Towards Translating Real-World Code with LLMs: A Study of Translating to Rust

Large language models (LLMs) show promise in code translation – the task of translating code written in one programming language to another language – due to their ability to write code in most programming languages. However, LLM's effectiveness on translating real-world code remains largely unstudied. In this work, we perform the first substantial study on LLM-based translation to Rust by assessing the ability of five state-of-the-art LLMs, GPT4, Claude 3, Claude 2.1, Gemini Pro, and Mixtral. We conduct our study on code extracted from real-world open source projects. To enable our study, we develop FLOURINE, an end-to-end code translation tool that uses differential fuzzing to check if a Rust translation is I/O equivalent to the original source program, eliminating the need for pre-existing test cases. As part of our investigation, we assess both the LLM's ability to produce an initially successful translation, as well as their capacity to fix a previously generated buggy one. If the original and the translated programs are not I/O equivalent, we apply a set of automated feedback strategies, including feedback to the LLM with counterexamples. Our results show that the most successful LLM can translate 47% of our benchmarks, and also provides insights into next steps for improvements.

I. INTRODUCTION

The task of program translation between programming languages is becoming particularly relevant, given the recent interest in safe programming languages such as Rust, and the expectation of translating potentially buggy, legacy code into such modern languages. While "rule-based" translation tools have been developed [1]–[3] that target a fixed source and target language (e.g. C to Rust), recent work [4]–[7] provides hope that large language models (LLMs) can accomplish this task for any source and target language.

Prior work in using LLMs for code translation [4]–[9] has almost exclusively focused on translating code taken from competitive programming websites [10], educational websites [11], or hand-crafted coding problems [12], [13]. While useful, such benchmarks are not representative of real-world code. For example, these benchmarks are typically a single function using only primitive data types, whereas real-world code has many functions and user-defined data types (e.g. structs).

In this work, we take a step towards answering the question: Can LLM's translate real-world code? Towards this end, we develop FLOURINE, an end-to-end code translation tool capable of producing validated Rust translations. FLOURINE first uses an LLM to obtain a candidate translation, then applies a compilation driven repair, where we make use of the Rust compiler's error messages as described in [14]. Once the translation compiles, FLOURINE uses cross-language differential fuzzing to test the I/O equivalence between the source program and the Rust translation. Notably, our cross-language differential fuzzer removes the need for unit tests – prior work assumed test cases already exist in the target language, or they were hand-written as part of the study, making a substantial investigation difficult. If a counterexample is discovered, FLOURINE executes a *feedback strategy*, which provides feedback to the LLM to fix the counterexample.

For the dataset we extract benchmarks from seven open source projects written in C and Go. We do not use the entire projects because LLMs cannot fit them in their context window. We choose these languages because Rust, C, and Go are typically used for low-level programming tasks, such as systems development, so C and Go are likely candidates for translation to Rust. The open source projects are from a diverse set of domains: audio processing, text processing, geometry, banking, 2D triangulation, graph algorithms, and sound card emulation. To automate and reduce bias in the selection of code samples, we develop a methodology and tool for extracting them from projects. We use this tool to extract code samples that contain between 1 and 25 functions and use only standard libraries, and which also use features such as global variables, user-defined, dynamically-allocated, data structures, array pointers, type casts, enumeration types etc..

For example, Figure 1 contains a program extracted from the *ACH* library featuring a global variable `moov_io_ach_stringZeros`, which is initialised with the function call `moov_io_ach_populateMap(94, "0")`. This kind of initialization of a global variable is not allowed in Rust, making it non-trivial to find an equivalent translation without resorting to unsafe code. Claude3 managed to find the following translation:

```
static MOOV_IO_ACH_STRING_ZEROS:
    Lazy<HashMap<usize, String>> =
    Lazy::new(|| populate_map(94, "0"));
```

This snippet uses `once_cell::Lazy`, which stores a value that gets initialized on the first access.

As another example, Figure 2 contains a program we extracted from the *go-gt* library, featuring a user-defined type `Env` that assembles several arrays, pointers and numeric data. Mapping `Env` to an exact counterpart in Rust is not obvious as a slice `[]int64` in Golang can be represented by a vector in Rust `Vec<i64>`, which is a growable, owning array-like data

```go
var (
    moov_io_ach_stringZeros map[int]string =
        moov_io_ach_populateMap(94, "0")
)

func moov_io_ach_populateMap(max int, zero string)
    map[int]string {
    out := make(map[int]string, max)
    for i := 0; i < max; i++ {
        out[i] = strings.Repeat(zero, i)
    }
    return out
}
```

Fig. 1: Code sample from *ACH*

type, or a borrowed-slice `&'a [i64]`, a non-growable, non-owning array-like data type. Our cross-language differential fuzzer handles translations of the function `add` that use both variants for `Env` by correctly mapping between the Go and Rust representations of its inputs (where the receiver `e` of type `*Env` is one of the inputs).

The code in Figure 3, extracted from the *go-edlib* library, returns all the longest common subsequences of two input strings (for brevity, we omit the callees). This code presents several challenges, for instance, finding a correct mapping between different styles of error handling. In Golang, a failable computation output is typically expressed by a pair of the target output type and the error type, as shown in the signature of `LCSBacktrackAll`. On the other hand, in Rust, this is often expressed by an optional output type, such as `Result<Vec<String>, Error>`. Moreover, this program contains casts from strings to arrays, and array manipulation, which need to be correctly mapped to their corresponding Rust representation.

We evaluate the LLM's ability to produce compilable translations, as well as translations that preserve the source behavior. Given that semantic equivalence is critical to program translation, we further investigate the LLM's potential to take feedback and fix equivalence errors. We develop and compare four different feedback strategies. Three of our feedback strategies provide the the LLM with counterexamples returned by the fuzzer. We compare these with a baseline strategy that repeats the original prompt, relying on randomness in LLM inference to obtain a new candidate translation.

In total, we perform 8160 code translation experiments across 408 code samples, four feedback strategies and five state-of-the-art LLMs – GPT4, Claude 3, Claude 2.1, Gemini Pro, and Mixtral. Overall, the LLMs achieve successful translation rates of 21% to 47% on our code samples, and feedback strategies are responsible for up to 8% absolute of this success rate. Somewhat unsurprisingly, we find that larger programs (in LoC) are less likely to translate successfully than smaller programs. Surprisingly, we also find that our feedback strategies that include counterexamples in the prompt actually *perform worse* than the simple baseline strategy. We discuss why this may be case, and suggest directions for future work.

```go
func (e *Env) add(i, p int64) {
    var j int64
    e.S[i] = true
    e.Prev[i] = p
    for j = 0; j < e.N; j++ {
        if e.Lx[i]+e.Ly[i]-e.G.Get(i, j) < e.Slack[i
            ] {
            e.Slack[i] = e.Lx[i] + e.Ly[i] - e.G.Get
                (i, j)
            e.Slackx[i] = j
        }
    }
}

func (m Matrix) Get(i int64, j int64) int64 {
    return m.A[i*m.N+j]
}

type Env struct {
    N                    int64
    G                    *Matrix
    S                    []bool
    Slack, Slackx, Prev []int64
    Lx, Ly               []int64
}

type Matrix struct {
    N int64
    A []int64
}
```

Fig. 2: Function `add` from *go-gt*

```go
func LCSBacktrackAll(str1, str2 string) ([]string,
    error) {
    runeStr1 := []rune(str1)
    runeStr2 := []rune(str2)

    if len(runeStr1) == 0 || len(runeStr2) == 0 {
        return nil, errors.New("Can't process and
            backtrack any LCS with empty string")
    } else if Equal(runeStr1, runeStr2) {
        return []string{str1}, nil
    }
    return processLCSBacktrackAll(
        str1,
        str2,
        lcsProcess(runeStr1, runeStr2),
        len(runeStr1),
        len(runeStr2),
    ).ToArray(), nil
}
```

Fig. 3: Function `LCSBacktrackAll` from *go-edlib*

We claim the following contributions:

- We develop FLOURINE, a tool capable of producing validated Rust translations without the need for hand-written test cases
- We build a cross-language fuzzer, capable of passing inputs and outputs between languages
- We use FLOURINE to conduct the first substantial study of using LLMs to translate real-world code
- We demonstrate that LLMs are capable of translating parts of real-world projects, and that directly providing counterexamples as feedback to an LLM is less effective

```rust
struct Env {
    n: i64,
    g: Box<Matrix>,
    s: Vec<bool>,
    slack: Vec<i64>,
    slackx: Vec<i64>,
    prev: Vec<i64>,
    lx: Vec<i64>,
    ly: Vec<i64>,
}

struct Matrix {
    n: i64,
    a: Vec<i64>,
}

fn add(e: &mut Env, i: i64, p: i64) {
    let mut j: i64 = 0;
    e.s[i as usize] = true;
    e.prev[i as usize] = p;
    for j in 0..e.n {
        if e.lx[i as usize] + e.ly[i as usize] - get
            (&e.g, i, j) < e.slack[i as usize] {
            e.slack[i as usize] = e.lx[i as usize] +
                e.ly[i as usize] - get(&e.g, i, j);
            e.slackx[i as usize] = j;
        }
    }
}

fn get(m: &Matrix, i: i64, j: i64) -> i64 {
    m.a[(i * m.n + j) as usize]
}
```

Fig. 4: Rust translation of function `add` from *go-gt*

than repeating the original prompt

- We open source all code, benchmarks, results to reproduce our experiments*

II. Related Work

In this section, we discuss closely related work from the literature under several categories.

Code Translation. The most closely related code translation works use LLMs for translation where the source and target languages are different. Most of them [4]–[7], [9] evaluate exclusively on competitive programming style code. In contrast, our work evaluates on real-world code, allowing us to draw stronger conclusions about LLM's ability for code translation. Others use some real-world code: [15] evaluates on real-world API code, translating Java to C#. Their technique requires additional fine-tuning, unlike ours. Another work, [8], uses real-world benchmarks but does not produce syntactically correct code for those examples. Two of the works [8], [9] conclude that counterexamples can be useful feedback, which does not match our conclusion. We compare our results with theirs in Section VI-C1.

Other LLM/ML code translation works focus on problems where the source an target language are the same [16], [17]. We consider this a different task than ours, because the goals are different. Meta studies have been conducted on code translation as well in [18], [19], though they do not provide insight on translating real-world code. Finally, several works have developed rule-based techniques for specific source and target language pairs such as C to Rust, [3], [20], [21], C to Go [1], and Java to C# [2]. While rule-based approaches can theoretically guarantee correctness of the translation, they require significant engineering effort to build, and they can produce unidiomatic code as we demonstrate in our results.

Cross-Language Differential Fuzzing. While differential fuzzing/testing has a rich literature, the majority do not consider comparing implementations in two different languages. There are many works that compare programs in the same language using symbolic execution [22]–[25] and fuzzing [26]–[29]. Such works do not need to solve the problem of mapping data from one language to another, though they are likely complementary to our work – they could be used to improve the coverage achieved by our fuzzer. Works in fuzzing multi-language systems [30] do not address this problem either. Only one work [31] attempts general cross-language testing like we do by compiling both languages down to a shared IR. As we will discuss in Section IV-B, this approach cannot effectively handle user-defined data types, and is heavily dependent on the IR compiler preserving structure of the original source program.

Feedback Strategies for LLMs. Only a limited number of works have tried to develop feedback strategies for LLMs. Recent work in automated program repair [32], [33] reports success with an approach that provides counterexamples as feedback. We discuss their results in relation to ours in Section VI-C1. While any automated program repair technique could be used as a feedback strategy, we focus only on feedback strategies that use an LLM to fix errors.

III. Overview

In this section, we define the task of code translation, provide an overview of our algorithm for code translation with LLMs, and then illustrate on a concrete example.

A. Code Translation

We first formally define the problem of code translation. Let l be a programming language, and P_l the set of all valid programs written in l. Assume we have a program $p \in P_l$ that we wish to translate to a different language l'. That is, we wish to find $p' \in P_{l'}$ that has the same behavior as p with respect to a mapping between the values of l and l'.

In our work, a program (i.e. p or p') is a set of functions, user-defined types (e.g. struct definitions), global variables, import/include statements etc. One of the functions in a program is the *entry point* function. Note that the entry point function is *not* necessarily `main()` – the inputs and outputs of the entry point could be primitive data types, user-defined types (e.g. structs, classes), and even pointer types.

For simplicity of notation, we define p and p' as operating on *program states*. A program state contains the values of the inputs outputs of the program, as well as variables defined in

*Artifact can be downloaded at https://d34gtk3knhjgeg.cloudfront.net/artifact.tar.gz

the global scope. Letting S_p and $S_{p'}$ be the set of all program states for p and p', respectively, we have $p : S_p \to S_p$ and $p' : S_{p'} \to S_{p'}$. We write $p(s_{in}) = s_{out}$ where $s_{in}, s_{out} \in S_p$ to denote the result of executing p on s_{in}.

To complete our definition of code translation, we define $M : S_p \to S_{p'}$ and $M' : S_{p'} \to S_p$, which are mapping functions that map states of program p to states of p', and vice versa. Formally, translation's goal is to discover a program p' such that:

$$\forall s \in S_p . p(s) = M'(p'(M(s)))$$

B. Our Code Translation Algorithm

Next, we present our iterative algorithm for code translation. We again assume we have a source program p, and we wish to discover a translation p' with the same behavior.

Let $G : Q \to P_{l'}$ be an LLM that takes a natural language query $q \in Q$ and outputs a candidate translation $p' \in P_{l'}$. q contains the original source program p and natural language instructions to translate p into the target language l'. Note that in practice, the resulting p' may have a top-level function whose function signature is incompatible with p and therefore the mapping functions M, M' cannot be defined, or the program output by the LLM may not compile. We find that the former rarely happens, and we address the latter through a compilation repair phase, which is based on the approach in [14].

We also assume the existence of a *fuzzer*. We define this as FUZZER(p, p'), which takes the original source program and translation, and returns two sets of examples E^+ and E^-. E^+ is a set of positive examples where p and p' agree. Positive examples have the form (s_{in}, s_{out}), where $s_{in}, s_{out} \in S_{p'}$. E^- is the set of counterexamples where the output produced by p disagrees with p'. A counterexample is a triple of states from S'_p of the form $(s_{in}, s_{exp}, s_{act})$, which are the initial state, expected output state, and the actual output state.

Finally, we have a routine FEEDBACK(q, p', E^-, E^+) which takes the query q, the candidate translation p', and the examples E^+, E^- returned by the fuzzer, and returns a new query that can be provided to G to generate a new candidate translation.

The routine for code translation with feedback strategies is shown in 1. We first use G to generate a candidate program p' from the initial query q, which we then pass to the compilation driven repair routine. If this is unsuccessful at making p' compile, we exit the loop and fail. Otherwise, we invoke the fuzzer to check for counterexamples. If none are found, we assume p' is correct, and return it. Otherwise, we invoke a feedback routine, which generates a new q, and repeat the process until a program is found that passes the fuzzer check, or until some fixed budget is reached and we fail.

C. Motivating Example

We now illustrate our Rust translation approach with the concrete example in Figure 2. In the example, our source program p is the function `add` from the *go-gt* library. This is a subroutine of the Hungarian algorithm [34] for finding maximum matching, which adds two edges to an alternating path during the search, and records the output by mutating the receiver `e`.

We first create an initial query containing the Go code and instructions describing the translation task, which is given to the LLM to generate a candidate translation. If we continue past compilation driven repair, the candidate translation p' in Figure 4 is guaranteed to compile, but not to be I/O equivalent to the original source program. To check for I/O equivalence, p and p' are passed to the fuzzer, which uses an off-the-shelf fuzzing technique to generate input states, execute both programs, and check that they produce the same output state. We capture side-effects by comparing whole program states, rather than just the explicit output.

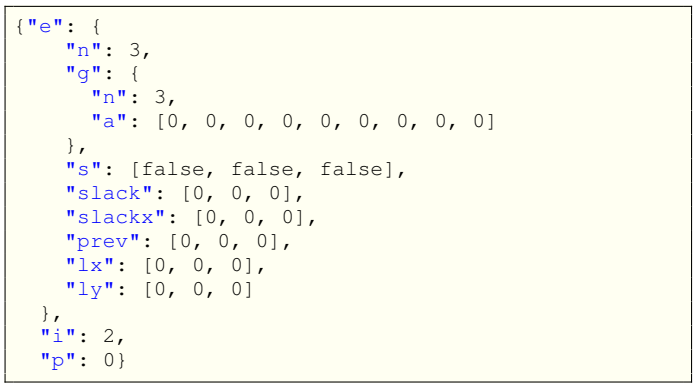
One of the challenges that we face is executing p and p' in two different languages on matching input states, and then comparing their output state. Specifically for our running example, we must convert primitive types as well as user-defined data structures: `Env` has distinct representations in Go and Rust; arguments `i` and `p` have type `int64` in Go, but `i64` in Rust; `e` is a pointer to an `Env` value in Go, but a mutable reference in Rust.

To solve this challenge, we develop a technique based on serializing then de-serializing to exchange data between languages. We use the JSON [35] format, because most languages support it. Most data types, including complex data types and pointers can be automatically serialized into JSON, thus it allows us to easily support real-world code. For our example, Fig. 5 denotes a serialized valid input state. Once the two programs are executed, the Go output state is again serialized to JSON, deserialized to Rust, and compared against the Rust output state. For our example, the expected output state, as obtained by executing the Go code, is the same as the input state in Fig. 5, with the only difference that the last element of field `s` is set to `true` instead of `false`. The translation in Figure 4 computes the expected output state, and it is thus deemed I/O equivalent to the original Go code, and returned by FLOURINE.

Algorithm 1 Iterative Code Translation Algorithm

Require: p : The program to translate, q: The initial task description, FEEDBACK: A feedback strategy, b: A budget

```
while b > 0 do
    p' ← G(q)
    if ¬COMPILATION-REPAIR(p') then
        break
    end if
    E⁻, E⁺ ← FUZZER(p, p')
    if E⁻ = ∅ then
        return p'
    end if
    q ← FEEDBACK(q, p', E⁻, E⁺)
    b ← b − 1
end while
return FAIL
```



```
{"e": {
    "n": 3,
    "g": {
      "n": 3,
      "a": [0, 0, 0, 0, 0, 0, 0, 0, 0]
    },
    "s": [false, false, false],
    "slack": [0, 0, 0],
    "slackx": [0, 0, 0],
    "prev": [0, 0, 0],
    "lx": [0, 0, 0],
    "ly": [0, 0, 0]
  },
  "i": 2,
  "p": 0}
```

Fig. 5: Serialized JSON input state for function `add`

Human:
 # Preamble
You are given a C/Go program. We need to translate it to Rust.

 # Code to be translated
{C/Go Program}

 # Instruction
Give me a Rust translation of the above C/Go code.
 # Constraints
Here are some constraints that you should respect:

- Give me only the translated code, don't add explanations or anything else. *# formatting guideline*
- Use only safe Rust. *# code characteristic*
- Do not use custom generics. *# fuzzer limitation*
- ...

Assistant:

Fig. 6: LLM Prompt for obtaining translations.

Conversely, if a counterexample is discovered by the fuzzer, then we invoke a feedback method, which uses the counterexample to create a new query to the LLM and generates a new candidate translation. Designing a suitable feedback method is another challenging aspect of the translation task. There are many ways to re-query the LLM for a new translation, each with their own likelihood of success. Moreover, most state-of-the-art LLMs are operated as API services which charge per input token, so different query strategies will have different dollar costs. To address this, we propose and evaluate a set of feedback strategies.

IV. LLM-Based Code Translation

A. Obtaining Translations

As mentioned in the previous sections, we are considering the problem of translating a program written in C or Go to Rust. We use zero-shot prompting and follow the best practices given by the LLM's provider. We construct the initial query q (to be input to the LLM) as sketched in Figure 6.

We start with a preamble describing the overall task. Then, we supply the program to be translated, and, finally, we provide specific constraints to be followed by the LLM. In particular, we have three types of constraints: formatting guidelines, code characteristics and fuzzer constraints. Formatting guidelines describe how the generated code should look, simplifying parsing and extraction of relevant information from the response. For code characteristics, we instruct the LLM to produce safe Rust code, and to maintain the same function names, parameter names, and return types from the input code. Finally, the fuzzer constraints ensure that the generated code can be handled by our fuzzer (more details on this in Section IV-B).

The translation generated by the LLM may not initially compile. We address this with approach in [14]. At a high level, we iteratively query the LLM to fix the error, until the code becomes compilable. Each time, we provide both the faulty translation and the error message from the Rust compiler to the LLM, and ask it to use a specific format for the suggested fixes, applying them only to the affected lines of code.

B. Checking Translations

To test the I/O equivalence between the original source program p and a candidate Rust translation p', we develop a cross-language differential fuzzer. For a given p and p', we automatically generate a fuzzing harness in Rust, which uses Bolero and libfuzzer [36] to perform fuzzing. The test harness generates program states from $S_{p'}$, which are directly invoked on p'. We implement the mapping function $M' : S_{p'} \to S_p$, using JSON de/serialization. We serialize the Rust program state s' into JSON format, and then instrument the source program p to deserialize the JSON into a program state of S_p. The instrumented p is invoked on the serialized s' from Rust using a foreign function interface. To compare outputs, we map the output state of p to one of p' using JSON de/serialization as well, which can then be directly compared.

We use JSON serializers for two reasons. First, the mapping between fields of user-defined data types in the source and target language are automatically determined based on the field names. This requires the LLM to produce data types with field names that match the source program, but in our benchmarks LLMs always do this. Second, most languages support automatic serialization of primitive, pointer, and user-defined types.

We note an alternative approach, taken by [31], is to compile both p and p' down to a common IR, such as LLVM, and then perform fuzzing on the IR. However, we find that IR compilers for different languages typically discard type and layout information (e.g. user-defined data types are represented as a void pointer). This makes it nearly impossible for a fuzzer to generate any meaningful inputs.

Soundness & Limitations. Our fuzzer can only make heuristic based guarantees (e.g. coverage) on the equivalence of p and p'. This is a limitation of fuzzing and testing in general. However, our fuzzer achieves an average line coverage of 97%.

In addition, JSON serialization is not automatically supported for all types. For example, features in Rust like trait

definitions, IMPL traits, and lifetimes in data type definitions are only partially supported. This means that the equivalence check may fail because serialization fails. We report these errors in Section VI-B. In addition, we do not support features like concurrency, network, and file I/O. Our benchmark selection excludes these features.

V. Feedback Strategies

In this section, we present four feedback methods that can be used if the fuzzer finds a counterexample E^- for the correctness of the translation p' by the LLM in Alg. 1.

*a) Simple Restart **Restart**:* We discard the generated code p' and re-query the model with the same prompt q.

*b) Hinted Restart **Hinted**:* This builds on the previous strategy by adding positive and negative examples from the fuzzer, E^+ and E^-, to the original prompt q. The intention is to suggest desirable behaviours to the LLM, as well as known faulty cases to avoid. We separately group the examples in E^+ and E^- based on the paths they exercise in p'. Intuitively, this corresponds to splitting them into equivalence classes, where each equivalence class corresponds to a particular program path. Then, the query constructed by **Hinted** only contains positive and negative examples from a single equivalence class, respectively.

*c) Counterexample-Guided Repair (**BaseRepair**):* Discarding the generated code p' when the fuzzer check fails may not always be the optimal choice. For instance, if p' is close to passing the fuzzer, trying to repair it might work better. As part of **BaseRepair**, we give counterexamples from the fuzzer to the LLM. Similarly to **Hinted**, a query only contains negative examples from the same equivalence class, which correspond to bugs associated with the same program path. The expectation is that the candidate translation generated in the next iteration of Alg. 1 will produce the correct outputs for the given examples. A sketch of the prompt used for **BaseRepair** is given in Figure 7 (excluding the lines colored in magenta). In Alg. 1, if the translation generated by G for the query q constructed by **BaseRepair** still fails the fuzzer check, then this last faulty translation will be considered by the next call to **BaseRepair**.

*d) Conversational Repair (**CAPR**):* Recent work in code translation [8] and automated program repair [37], have proposed *conversational* repair approaches, wherein previous incorrect code is included in the prompt to the LLM to discourage the LLM from producing the same code again. The **CAPR** approach begins with the same prompt as **BaseRepair**, however they differ if the new translation still fails the fuzzer check. In **BaseRepair**, we create a new prompt from scratch, but in **CAPR**, we keep the prompt, and append a new piece of dialogue to it as shown in magenta Figure 7. This process can be repeated multiple times, meaning the prompt is a dialogue of failed translations.

The methods **Restart** and **Hinted** cost less than **BaseRepair** and **CAPR** as they don't include the incorrect translation in the prompt. Therefore the former use about half the input tokens of the latter.

Human:

```
# Preamble
You are given a C/Go program and its faulty Rust
translation. We need to repair the faulty Rust
program.

# Code to be translated
{C/Go Program}

# Code to be repaired
{Faulty Rust Program}

# Instruction
Make changes to the given code to obtain expected
outputs for the given test inputs.

# Constraints
Here are some constraints that you should respect:
...

# Counterexamples
CE1
CE1
```

Assistant:

```
{LLM generated rust translation}
```

Human:

```
That is incorrect on the following inputs:
# Counterexamples
CE1
CE2
```

Assistant:

Fig. 7: LLM Prompt for **BaseRepair** and **CAPR**. **BaseRepair** is shown in black. **CAPR** is shown in black and magenta.

VI. Evaluation

In this section, we present our results for the following research questions.

RQ1: How do LLMs perform on translating code taken from real-world projects to Rust? We gather a large number of benchmarks by extracting code samples from real-world projects, and we use LLMs to generate translations which are then checked for correctness by the fuzzer, and fixed if needed by applying feedback strategies. We answer the following concrete questions.

(*RQ1.1*) How many benchmarks can each LLM translate from each of our projects? We report the percentage of benchmarks from each project that are successfully translated for each LLM. We show that success rates vary widely based on the benchmark and LLM. LLMs achieve up to 80% success rate on benchmarks from our "easiest" project, and between 15-40% on our "hardest" project.

(*RQ1.2*) How does code complexity affect the success rate of translation? We look at how lines of code and number of functions in a benchmark influence the success rate. We show lines of code strongly influences success rates.

(*RQ1.3*) How idiomatic is the Rust produced by LLMs? We run Clippy [38], Rust's standard linter, on the successful translations, and analyze the rates of different categories of

linter warnings. We show that LLMs occasionally (1-15% of the time) produce code with linter warnings, suggesting that the translations could be made more performant, concise, or use unsafe code.

RQ2: How effective are feedback strategies at fixing translation bugs? In addition to overall translation success rates, we record the initial success rates – the rate at which the first translation passes the fuzzer – and compare this to the overall success rate. We answer two concrete questions.

(*RQ2.1*) How much do feedback strategies increase the translation success rate? We compare overall success rates directly to initial success rates. We show that the most effective feedback strategy increases the success rate by an absolute 6-8% on average for the best LLMs.

(*RQ2.2*) Which feedback strategies increase success rates the most? We compare the increase in success rates for each feedback strategy. We show that, surprisingly, **Restart** and **Hinted** outperform **BaseRepair** and **CAPR** consistently. We provide a plausible explanation for this result.

RQ3: How do LLM translations compare to rule-based translation tools? We compare LLM translations to translations produced by the rule-based translation tool C2Rust [3]. While C2Rust theoretically can guarantee the correctness of the translation, we show LLMs produce far more concise and idiomatic translations.

RQ4: Why do translations fail? Translation can fail for several reasons beyond the fuzzer finding counterexamples. We report failure rates for different failure reasons.

A. Experimental Setup

1) Implementation: We implement an end-to-end translation tool FLOURINE, which takes as input (1) a program, (2) a feedback strategy to apply, and (3) a budget. FLOURINE outputs either a corresponding Rust translation that passes the fuzzer, or it fails with an error. Internally, FLOURINE implements Algorithm 1. FLOURINE is written entirely in python, except for the fuzzer, which is written in Rust. FLOURINE currently supports C and Go for the input program. FLOURINE is implemented as a framework, which can be extended with new LLMs, feedback strategies, and language support for the input program. We use GNU Parallel [39] to run experiments in parallel.

2) LLMs: We limit our study to LLMs hosted by third party providers. This is in part because they are the highest performing on coding tasks, and they are the most accessible in that they do not require the user to own powerful compute resources. We use five LLMs in our evaluation: GPT-4-Turbo [40], Claude 2.1 [41], Claude 3 Sonnet [41], Gemini Pro [42], and Mixtral [43]. The first four are likely very large (1T+ parameters). On the other hand, Mixtral is relatively small (45B parameters), but is known for performing well on coding tasks, and costs less than the others. We access GPT-4-Turbo and Gemini Pro through OpenAI's and Google's APIs. We access Claude and Mixtral through AWS Bedrock. Due to lack of access to GPU machines, we do not attempt to run open source LLMs like CodeLLaMA.

TABLE I: Benchmark details

Project	Lang.	#Benchs	Min/Max/Avg LoC	Min/Max/Avg #Func
libopenaptx	C	31	13 / 173 / 69	1 / 9 / 2.9
opl	C	81	19 / 460 / 67	1 / 15 / 2.8
go-gt	Go	43	9 / 213 / 51	1 / 16 / 3.5
go-edlib	Go	36	13 / 597 / 62	1 / 25 / 3.1
ach	Go	121	43 / 194 / 64	3 / 7 / 3.4
geo	Go	67	13 / 70 / 35	3 / 7 / 4.1
triangolatte	Go	29	9 / 164 / 38	1 / 10 / 2.5

3) Benchmarks: We collect benchmarks from real-world projects hosted on GitHub. We focus on C and Go as the source program languages for multiple reasons. First, C, Go, and Rust are typically used for lower-level programming tasks, unlike other popular languages like Java or Python. Thus they are likely candidates for translating to Rust. Second, and more pragmatically, projects written in C and Go make less use of third party libraries, which we do not attempt to support for this work. Conversely, most Java and Python projects make heavy use of third party libraries.

We choose seven projects with the aim of getting a diverse set of application domains. Our projects are:

- **ACH** [44]: a Go library implementing a reader, writer, and validator for banking operations
- **geo** [45]: a math-focused Go library implementing common geometry functions and interval arithmetic
- **libopenaptx** [46]: a C library for audio processing
- **opl** [47]: a C library for sound card emulation
- **go-gt** [48]: a Go library for graph algorithms
- **go-edlib** [49]: a Go library string comparison and edit distance algorithms
- **triangolatte** [50]: a 2D triangulation library in Golang

As we will show in our experiments, LLMs are still not capable of translating entire projects. To create benchmarks of manageable size, we develop a tool for automatically extracting benchmarks from these projects. Our tool takes as input the entire project and a specific function identifier f in the project. The tool then analyzes the project to find all of f's dependencies, including all functions called by f (including transitive calls), type definitions, standard libraries, global variables, etc. and extracts them intro a single, compilable file. The translation task is then to write a compilable Rust file with a function equivalent to f's behavior. Our methodology for selecting benchmarks is to iterate over all functions in a project, create a benchmark for it, and keep it if it meets the following criteria: (1) it does not use 3rd party libraries, (2) the maximum depth of the call graph is less than 4.

Details on the benchmarks are given in Table I. The total number of benchmarks extracted from each project is given in the column "#Benchs". LoC and number of functions for individual programs vary from 13 to 597 and from 1 to 25, respectively.

4) LLM Hyperparameters: All LLMs use a *temperature* parameter for controlling the randomness/creativity of its output. To make our results more deterministic, we use a lower temperature (i.e. less random) of 0.2. Other hyperparameters, e.g. topP and topK, are set to the default value recommended by the LLM's provider.

5) FLOURINE *Hyperparameters:* We set the budget b in Algorithm 1 to 5. For the **Hinted** and **BaseRepair** strategies we provide 4 examples in the prompt (more examples appeared to reduce efficiency as the context window grew). For the **CAPR** strategy, we keep conversation window size as 3, which means that only the latest 3 incorrect translations are made available to the LLM. A translation is deemed equivalent if 5 minutes of fuzzing does not return any counterexamples.

6) Compute Resources: We run our experiments on a machine with an AMD EPYC 7R13 Processor with 192 cores and 380 GB of RAM. Each translation task is run sequentially in a single thread (we do not parallelize individual translation tasks or fuzzing). As previously mentioned, all LLMs are accessed through APIs provided by third party services.

B. Results

We run a translation experiment for each of our five LLMs, four feedback strategies, and 408 benchmarks for a total of 8160 translation experiments. A translation is successful if it compiles and passes the fuzzer. A translation is failed if it: (1) does not compile, (2) the fuzzer cannot de/serialize the types used in the translation, or (3) the fuzzer finds a counterexample in the translation and the budget is reached if applicable. We answer our research questions based on these results.

RQ1: How do LLMs perform on translating code taken from real-world projects to Rust? Our LLMs achieve overall success rates of 47.7% (Claude 2), 43.9% (Claude 3), 21.0% (Mixtral), 36.9% (GPT-4-Turbo), and 33.8% (Gemini Pro). We present detailed results for each LLM in Figures 8, 9, 10, and 11. The success rate is the total number of successful translations divided by the total number of translation experiments in the category (experiments for different feedback strategies are averaged together). We answer our sub-questions below.

(*RQ1.1*) How many benchmarks can each LLM translate from each of our projects? Figure 8 shows success rates by benchmark and LLM. The best LLMs achieve success rates of 20-60% depending on the benchmark, with one outlier of 80% by Claude 2 on ACH. The outlier is in large part due to ACH having ∼40 extremely similar benchmarks, which Claude 2 nearly always gets right. If we remove these similar benchmarks, the success rate for Claude 2 drops to 55%, which is in line with the other LLMs. A consistent trend is that Mixtral, while somewhat capable, has 5-20% lower success rates than the other much larger and more expensive LLMs. However, the cost of running Mixtral (both in dollars and compute) is at least 10x less than the other LLMs. Other trends are that Claude 2, Claude 3, and GPT-4-Turbo perform similarly on most benchmarks, and they outperform Gemini in most cases.

(*RQ1.2*) How does code complexity affect the success rate of translation? We use lines of code and number of functions as proxy metrics for complexity, and we show success rates for benchmarks grouped by level of complexity in Figures 9 and 10. The general trend is that increasing complexity, especially in lines of code, reduces success rate. The spikes for 3 functions and 48-82 lines of code are again due to the ACH benchmarks mentioned in the previous research question. Removing these flattens the spike. In particular, success rates tend to drop off somewhere around 100+ lines of code. We discuss approaches for handling larger benchmarks in section VI-C2.

(*RQ1.3*) How idiomatic is the Rust produced by LLMs? Figure 11 shows the rate of different categories of linter warnings produced by Clippy [38], Rust's standard linter. We limit our analysis to successful translations. Clippy reports five types of warnings, and we add **unsafe**. We describe them below, and give specific examples of the warnings most frequently reported by Clippy on the Rust translations.

Correctness: reports code that may have correctness bugs. The common examples we find are: checking if an unsigned integer is greater than 0, and using `MaybeUninit::uninit().assume_init()` (i.e. assuming that potentially unitialized data is initialized)

Suspicious: the same as Correctness, but could be a false positive

Style: code that is unidiomatic, but still correct. The common examples we find are: not following naming conventions, unnecessary borrows, using `return` statements, unnecessary closure expressions (e.g. `xs.map(|x| foo(x))` instead of `xs.map(foo)`), using class types (e.g. `String`) when a simple primitive type will suffice (e.g. `str`), and not using idiomatic statements (e.g. using `x <= z && z <= y` instead of `(x..y).contains(z)`)

Complexity: code that could be simplified. Common examples are: unnecessary casting or type conversion, unnecessary parentheses, and unnecessarily putting a `Box<..>` around the type of a function parameter

Performance: code that could be written to run faster. The most common example is unnecessarily putting a `Box<..>` around local variables or collection types (e.g. `Vec`)

Unsafe: code wrapped in an `unsafe` block

Overall, LLMs produce very few correctness warnings, however they occasionally (1-15% of the time) produce code that could be more idiomatic (Style warnings), more concise (Complexity warnings), or more performant (Performance warnings). Gemini's high rate of Style warnings is due to its preference for using `return` statements when not necessary. We suspect that a large number of these warnings could be eliminated through prompting (e.g. instructing the LLM to prefer not using return statements, follow specific naming conventions, and not use `Box<..>` in certain cases). We also observe occasional use of `unsafe`, however the use of

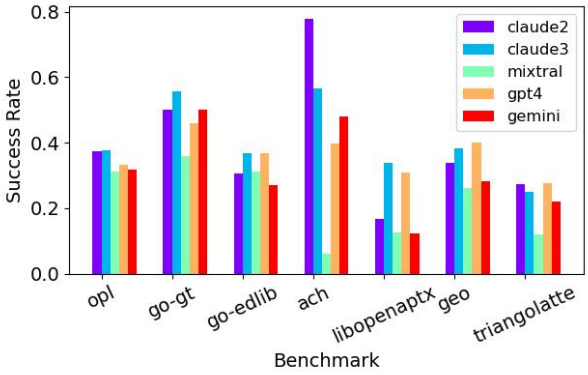


Fig. 8: Success rate for each LLM on each benchmark. Averaged across all feedback strategies.

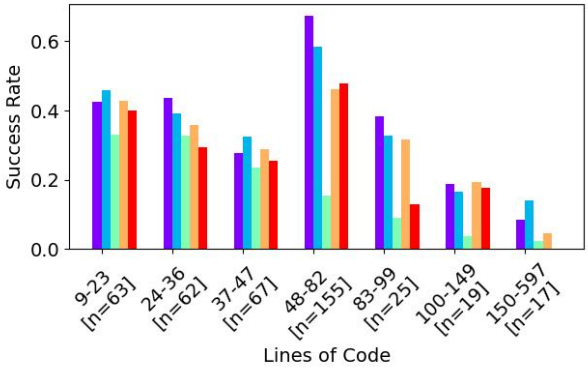


Fig. 9: Success rate for each LLM on benchmarks grouped by lines of code.

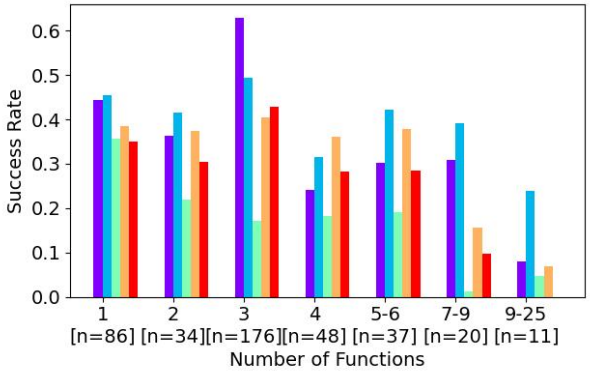


Fig. 10: Success rate for each LLM on benchmarks grouped by number of functions.

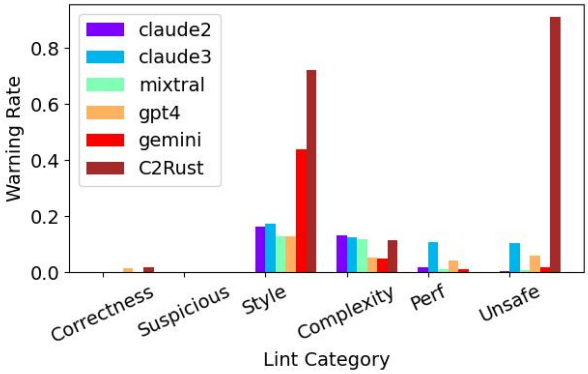


Fig. 11: Rates of different types of linter warnings for each LLM.

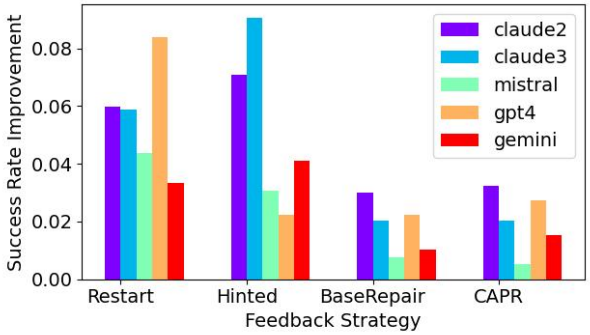


Fig. 12: Absolute improvement in success rates after applying feedback strategies as compared to initial success rate.

unsafe code was not necessary in those cases.

RQ2: How effective are feedback strategies at fixing translation bugs? We answer this question by comparing the initial success rate – the rate at which the first translation passes the fuzzer, after fixing compilation errors – to the final success rate after applying feedback strategies to the unsuccessful translations.

(*RQ2.1*) How much do feedback strategies increase the translation success rate? Figure 12 shows the final success rate subtracted from the initial success rate. Disappointingly, the best feedback strategy only improves success rates by 6-8% absolute across all of our benchmarks.

(*RQ2.2*) Which feedback strategies increase success rates the most? Figure 12 also shows, surprisingly, that the most reliable strategy is **Restart** (simply repeating the same prompt). In fact, our results suggest that providing counterexamples in the prompt may actually confuse the LLM. We discuss this trend further in Section VI-C1.

RQ3: How do LLM translations compare to rule-based translation tools? We compare the idiomatic-ness of LLM generated Rust to C2Rust [3] on our opl benchmark (C2Rust failed to produce code for most of libopenaptx). Rates of linter warnings are presented in Figure 11. Overall we can see that the majority of code produced by C2Rust is unsafe and it is far less idiomatic, as indicated by the rate of style warnings. In addition, we observe that C2Rust produces

far more verbose Rust than LLMs. On average C2Rust translations have 1.98x more LoC than LLM translations.

RQ4: What is the main cause of translation failure? There are three reasons a translation can fail. (1) A compiling translation cannot be found. This accounts for only 7.0% of failures. (2) The fuzzer cannot de/serialize the data types. These account for 52.6% of failures. (3) A counterexample is found in the final translation. These account for 40.3% of failures. The implication of this result is that we likely under-report the true translation success rate, because at least some serialization failures might be successes.

C. Discussion & Future Work

1) Improving Feedback Strategies: Our result that counterexamples harm performance contradicts several recent works' results [8], [9], [32], [33]. We note that two of the works [8], [9] do not compare with a simple baseline like **Restart**, so they cannot conclude if counterexamples helped or hurt. However, the other two [32], [33] do report benefit from counterexamples relative to a baseline, and they evaluate on real-world code (though their task is automated program repair as opposed to code translation). The most likely explanation for their success and our failure is that their counterexamples use inputs from human-written test cases, so they might be more "intuitive" to an LLM. On the other hand, our counterexamples come from random inputs generated by a fuzzer. Our own manual analysis reveals that random fuzzer inputs are not intuitive to a human and their textual representation can be very large and unintuitive to an LLM as well. A future direction is to study what types of counterexamples are useful for LLMs. Studying input selection and input reduction techniques would likely be immediately fruitful.

2) Handling Larger Benchmarks: We conjecture that the stochastic nature of LLMs' next token prediction poses fundamental limitations for translating large source programs in one go. Larger input source programs require more Rust code to be generated by the LLM, or in other words, more tokens to be predicted. Each time a token prediction is made, there is some probability that an erroneous prediction is made. Letting e be the probability of an erroneous prediction, the probability that the LLM correctly predicts n tokens is $(1-e)^n$. This probability quickly goes to 0 as n increases. A future direction that (at least theoretically) solves this problem is to develop a solution that partitions the input source program into chunks that can be individually translated and validated. The most obvious way to partition an input program is by function, but one could imagine even going down to the basic block level.

3) Improving the Fuzzer: Given the high rate of failure due to serialization limitations, improving the serializer in our fuzzer to handle more features data types is likely necessary to make additional progress as well.

VII. Threats to Validity

Our main results are that (1) LLMs can translate real-world code, and that (2) providing counterexamples to the LLM is not effective feedback. We discuss threats to the validity of these conclusions. The biggest threat to (1) is that the fuzzer may miss counterexamples. While we acknowledge this limitation, we point out that the fuzzer achieves 97% coverage on average, thus we are confident that the translations are "mostly" correct. This limitation is also generally accepted in prior work, which uses test suites to assess correctness. Another threat to (1) is that our results do not generalize to other languages. We argue this is unlikely for popular languages like Java and Python, given that they more represented in LLM training data than C and Go. However, even if our results do not generalize to other languages, translating C and Go to Rust is still highly practical. The biggest threat to (2) is that other prompting strategies may improve the LLMs ability to use counterexamples. While possible, we believe this is unlikely due to the complexity of inputs in our benchmarks, as explained in Section VI-C1. Finally, recent work [51] has shown high non-determinism in code generated by LLMs, which poses a threat to both (1) and (2). While we don't run our experiments multiple times, executing our feedback strategies for multiple iterations has a similar effect. It is highly unlikely an additional run of experiments would give significantly different results, and doing so would be expensive (in dollar cost).

VIII. Conclusion

In this work, we study the ability of LLMs to translate real-world code to Rust. We present FLOURINE, an end-to-end Rust transpiler, and we use it to test the ability of five state-of-the-art LLMs to translate C and Go code taken from real-world projects to Rust. Our results demonstrate that LLMs are indeed capable of translating code to Rust, though there room for improvement. In addition, we show that counterexamples, at least random fuzzer generated counterexamples, are ineffective feedback for an LLM.

References

[1] "C to go translator." https://github.com/gotranspile/cxgo.

[2] "Sharpen - automated java-¿c# coversion." https://github.com/mono/sharpen.

[3] "C2rust transpiler." https://c2rust.com/.

[4] Z. Tang, M. Agarwal, A. Shypula, B. Wang, D. Wijaya, J. Chen, and Y. Kim, "Explain-then-translate: an analysis on improving program translation with self-generated explanations," in *Findings of the Association for Computational Linguistics: EMNLP 2023* (H. Bouamor, J. Pino, and K. Bali, eds.), (Singapore), pp. 1741–1788, Association for Computational Linguistics, Dec. 2023.

[5] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *NeurIPS*, 2020.

[6] B. Rozière, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," in *ICLR*, OpenReview.net, 2022.

[7] M. Szafraniec, B. Roziere, H. L. F. Charton, P. Labatut, and G. Synnaeve, "Code translation with compiler representations," *ICLR*, 2023.

[8] 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," 2024.

[9] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, "Attention, compilation, and solver-based symbolic analysis are all you need," *arXiv preprint arXiv:2306.06755*, 2023.

[10] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[11] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," *arXiv preprint arXiv:2108.11590*, 2021.

[12] 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, 2024.

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

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

[15] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 695–707, 2023.

[16] Q. Zhang, J. Wang, G. H. Xu, and M. Kim, "Heterogen: transpiling c to heterogeneous hls code with automated test generation and program repair," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), p. 1017–1029, Association for Computing Machinery, 2022.

[17] B. Mariano, Y. Chen, Y. Feng, G. Durrett, and I. Dillig, "Automated transpilation of imperative to functional code using neural-guided program synthesis," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[18] H. F. Eniser, V. Wüstholz, and M. Christakis, "Automatically testing functional properties of code translation models," *arXiv preprint arXiv:2309.12813*, 2023.

[19] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1529–1541, IEEE, 2023.

[20] H. Zhang, C. David, Y. Yu, and M. Wang, "Ownership guided C to Rust translation," in *Computer Aided Verification (CAV)*, vol. 13966 of *LNCS*, pp. 459–482, Springer, 2023.

[21] 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.

[22] Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "Hydiff: Hybrid differential software analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1273–1285, 2020.

[23] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Regression tests to expose change interaction errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 334–344, 2013.

[24] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: testing for divergences between software versions," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1181–1192, 2016.

[25] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *Acm Sigplan Notices*, vol. 46, no. 6, pp. 504–515, 2011.

[26] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 739–743, 2018.

[27] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *2010 Third international conference on software testing, verification and validation*, pp. 137–146, IEEE, 2010.

[28] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 176–187, IEEE, 2019.

[29] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*, pp. 615–632, IEEE, 2017.

[30] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "PolyFuzz: Holistic greybox fuzzing of Multi-Language systems," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 1379–1396, USENIX Association, Aug. 2023.

[31] J. J. Garzella, M. Baranowski, S. He, and Z. Rakamarić, "Leveraging compiler intermediate representation for multi- and cross-language verification," in *Verification, Model Checking, and Abstract Interpretation* (D. Beyer and D. Zufferey, eds.), (Cham), pp. 90–111, Springer International Publishing, 2020.

[32] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE*, IEEE, 2023.

[33] J. Kong, M. Cheng, X. Xie, S. Liu, X. Du, and Q. Guo, "Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs," *arXiv preprint arXiv:2403.01971*, 2024.

[34] H. W. Kuhn, "The hungarian method for the assignment problem," in *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art* (M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, eds.), pp. 29–47, Springer, 2010.

[35] E. T. Bray, "The javascript object notation (json) data interchange format," RFC 8259, RFC Editor, 12 2017.

[36] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, pp. 157–157, 2016.

[37] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[38] "Clippy: A bunch of lints to catch common mistakes and improve your rust code." https://rust-lang.github.io/rust-clippy/.

[39] O. Tange, "Gnu parallel 20240122 ('frederik x')," Jan. 2023. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.

[40] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[41] "Claude." https://www.anthropic.com/index/introducing-claude.

[42] "Gemini." https://blog.google/technology/ai/google-gemini-ai/.

[43] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand, *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[44] "Moov ach." https://github.com/moov-io/ach.

[45] "S2 geometry library in go." https://github.com/golang/geo.

[46] "Open source implementation of audio processing technology codec (aptx)." https://github.com/pali/libopenaptx.

[47] "Engine for making things with a ms-dos feel, but for modern platforms." https://github.com/mattiasgustavsson/dos-like/blob/main/source/libs/opl.h.

[48] "go-gt." https://github.com/ThePaw/go-gt.

[49] "String comparison and edit distance algorithms library." https://github.com/hbollon/go-edlib.

[50] "2d triangulation library." https://github.com/tchayen/triangolatte.

[51] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," 2023.