VERT: Polyglot Verified Equivalent Rust Transpilation with Large Language Models

Aidan Z.H. Yang[‡]* Yoshiki Takashima[‡][†]
Brandon Paulsen*, Josiah Dodds*, Daniel Kroening*
*Amazon Web Services [†]Yale Law School
[‡]Equal Contribution

***Abstract*—Rust is a programming language that combines memory safety and low-level control, providing C-like performance while guaranteeing the absence of undefined behaviors by default. Rust's growing popularity has prompted research on correct and idiomatic transpiling of existing code-bases to Rust. Existing work falls into two categories: rule-based and large language model (LLM)-based. While rule-based approaches are theoretically sound, they often yield unidiomatic and *unsafe* Rust code, and are limited to few source languages, which hinders maintainability and industrial application. By contrast, LLM-based approaches, while providing no guarantees, are polyglot and typically produce more idiomatic and safe Rust code. In this work, we present *VERT*, a formally correct, polyglot Rust translator with more idiomatic outputs. *VERT* supports any language that compiles to Web Assembly. Using the Web Assembly compiler, *VERT* obtains an *oracle* Rust program. Leveraging the LLM, *VERT* generates an idiomatic candidate Rust program. This candidate is verified against the oracle with model-checking to ensure equivalence.**

I. Introduction

Rust is a memory- and type-safe programming language that has performance on par with low-level languages like C. It is often referred to as a "safer C", because the Rust type checker can guarantee the absence of undefined behavior. Citing Rust's security benefits [14], Rust is used in major open source projects such as Linux [43] and AWS Firecracker [7]. Rust's security and performance benefits have fueled interest in automatically transpiling existing code written in other languages into Rust [2], [39]. Existing works on transpilation broadly fit into two categories: *rule-based* and *large language model (LLM)-based*.

Rule-based approaches use expert-written rules and algorithms that translate a source program by identifying known patterns in the source code and applying the rules. Rule-based approaches have implementations that could, in theory, be proven sound [22]. However, they often result in unidiomatic code that does not take full advantage of the target language's features, such as efficient native types. For example, CROWN [52], a state-of-the-art C to Rust transpilation tool, produced an average of 10.9 linter warnings per function translated. `rWasm` [10], a security focused Web Assembly (Wasm) to Rust transpiler, produced 10 times more lines of code compared to the source code before compiling to Wasm. While `rWasm` leverages Rust for security features and thus does not intend the output to be idiomatic, the same cannot be said of general transpilers.

LLM-based approaches apply an LLM that takes a program in one or more languages as input and attempt to output an equivalent program in the target language [36]. LLM-based approaches tend to produce code that is similar to their training data, and thus, if the model is trained on high quality, human written code in multiple languages, then the model will usually produce high quality, idiomatic transpilations [29], [36], [39], [44], [45], [49], [50] in the languages it trained on. However, these approaches come with no formal guarantees that the resulting code will maintain input-output equivalence with the original [28], [31], [48]. Language models are notorious for outputting subtly incorrect code [48]. Subtle errors such as flipping a `-` to a `+` may escape human review, be difficult to debug, and only manifest in corner cases.

In this work, we focus on *polyglot*, *verified*, and *idiomatic* transpilation of single programs to Rust. By *polyglot*, we mean that it can apply to most major languages. By *verified*, we mean that input-output equivalence of the final transpilation can be verified against the source program in some way. *VERT* uses bounded model checking (BMC) and property-based testing (PBT) to verify and test the translation. By *idiomatic*, we mean that the transpilation follows Rust conventions. While idiomaticity of the translation is a subjective measure, it is essential when using Rust translators in industrial settings.

To deliver polyglot, verified, and idiomatic transpilations, we combine rule-based and LLM-based transpilation with formal verification tools, and implement our approach in a tool *VERT*. Our algorithm takes a source program as input, and outputs a transpilation that is verified equivalent relative to a rule-based transpilation. Notably, *VERT* does not require any additional input beyond the source program. The main assumption of *VERT* is that the language of the source program has a Wasm compiler.

VERT first creates an *oracle* Rust transpilation by using the source language's Wasm compiler `rWasm`, and a single test case to drive the program. We choose Wasm as the intermediate as it is supported by over 75 languages [5], including the 20 most popular languages in the TIOBE index [12]. This transpilation is equivalent by construction, but is unidiomatic. Next, we leverage an LLM to produce a candidate final transpilation, which is far more idiomatic, but may have implementation errors, ranging from syntax errors to subtle logic errors. We then enter an iterative repair and verify process. We first attempt to fix compilation errors by applying a combination of handcrafted rules and re-prompting the LLM to re-transpile the source program until the program compiles. Once compiled, we attempt to verify equivalence using one of the previously mentioned verification techniques. If verification succeeds, then we stop and output the program. However, if verification fails, which is usually the case, we re-prompt the LLM to transpile the program.

We evaluate *VERT* on 1,394 transpilation tasks with source languages in C++, C, and Go curated from prior work [39], [52]. We focus on C++ and C since these two languages are often used for similar tasks as Rust [23], [33]. To assess *VERT*'s polyglot transpilation abilities, we further evaluate on Go.

With Anthropic Claude-2 [4] as the underlying LLM, our results show that *VERT* can produce transpilations that pass BMC for 42% of these programs, and differential testing for 54% of these programs. Moreover, using *VERT* improves the translation capabilities of Claude-2 – *VERT* with Claude-2 is able to produce a verified transpilation for 40% of inputs (41% for C++, 37% for C, and 46% for Go) compared to 1% for Claude-2 alone.

In summary, our main contributions are as follows.

- ***VERT*.** We propose and implement an end-to-end technique that can transpile any language that compiles to Wasm into idiomatic Rust. Our data and tool are available as open-source.[1]
- **Verified Equivalence with Input Code** We use Wasm generated from original input as a reference model and perform equivalence checking by automatically injecting test harnesses, allowing the user to verify that the LLM translation is free of hallucinations.
- **Empirical evaluation**. We evaluated *VERT* on a set of real world programs and competitive programming solutions, which include 569 C++ programs, 520 C programs, and 305 Go programs. We perform an extensive evaluation of several LLMs directly (CodeLlama-2), with fine-tuning (StarCoder), and with instruction-tuned few-shot learning (Anthropic Claude-2) on different source code languages.

II. Background

We give a brief introduction of the key aspects of our tool. In particular: migration to Rust, the `rWasm` compilation strategy, and tools to test or verify Rust.

A. Migrating to Rust

Rust is a systems programming language with a focus on performance, reliability, and safety. Rust's main goal is to eliminate memory safety errors through a *memory-ownership* mechanism. Rust's memory-ownership mechanism associates each value in memory with a unique *owner* variable, which guarantees safe static memory collection.

Given the memory-safety properties of Rust, there is a strong incentive to migrate existing codebases to Rust. While several notable projects have been rewritten in Rust [51], the translation to Rust remains a challenge owing to the enormous manual effort. For C to Rust translation in particular, several tools have been developed to automatically translate C functions to Rust [2], [20], [52]. These tools use the semantic similarity between C and Rust and apply re-writing rules to generate Rust code. However, these re-write rules are specific to the source language, and do not generalize to other languages, especially those whose semantics is not similar to Rust. To the best of our knowledge, no rule-based automatic translator exists from a garbage-collected language like Go or Java to Rust.

In contrast to transpilers, `rWasm` differs significantly in its intent. `rWasm` converts Web Assembly (Wasm) programs into Rust and leverages the memory-safety properties of safe Rust as a sandbox to eliminate the Wasm runtime overhead. Since many programming languages already target Wasm as an intermediate representation [5], we can leverage `rWasm` for multi-language support.

B. Rust Testing and Verification

To establish trust in the LLM-output, we perform equivalence verification of two Rust programs. We use existing tools that operate on Rust to prove equivalence between the LLM-generated and the `rWasm`-generated oracle Rust programs. We use `bolero`, a Rust testing and verification framework that can check properties using both Property-Based Testing (PBT) [21] and Bounded Model Checking (BMC) [15], [16]. An example of a `bolero` harness for checking equivalence between an LLM-generated and a trusted reference program is given in Fig. 1.

```
#[test]
#[cfg_attr(kani, \kani::proof)]
fn eq_check()  {
    bolero::check!()
        .with_type()
        .cloned()
        .for_each(|(a, b) : (i32, i32)| llm_fn(a
            , b) == reference_fn(a, b));
}
```

Fig. 1: An example bolero harness checking equivalence between 2 functions.

Using a harness like the one in Fig. 1, `bolero` can check properties via two different methods. The first method is by random PBT. PBT works by randomly generating inputs to the function under test, running the harness with these inputs, and asserting the desired properties (e.g. equivalence between two functions). PBT repeatedly runs this procedure to check the property over the range of inputs. PBT is a valuable tool for catching implementation bugs, however it is generally infeasible to run PBT for long enough to exhaust all possible inputs to a program.

The second method available via `bolero` is model checking through `kani` [42], a model-checker for Rust. When run with `kani`, `bolero` produces symbolic inputs rather than random concrete inputs. Executing the harness with symbolic inputs, we can cover the entire space of inputs in one run and the model-checker ensures the property holds for all possible inputs. Since

[1]https://zenodo.org/records/10927704

symbolic execution does not know how many times loops are run, kani symbolically executes loops up to an user-provided bound. To prove soundness of this bound, kani uses *unwind checks*, asserting that the loop iteration beyond the bound is not reachable.

While kani can prove properties of programs, complex programs can take too long to prove. Conversely, PBT runs exactly as quickly as the program runs, but is not exhaustive. Given the complementary properties of PBT and kani, we allow users to apply both tools, and mark the harness to enable both PBT (`#[test]`) and BMC (`kani::proof`).

III. *VERT* Rust Transpiler

In this section, we describe the key ideas behind our universal transpilation technique. Figure 2 gives an overview of *VERT*'s entire pipeline. The technique takes as input a source program, and outputs a verified equivalent Rust program. As shown in Figure 2, we parse the program into separate functions during the cleaning phase, then split the pipeline into two paths. The first path outputs an LLM-generated Rust transpilation. The second path produces rWasm Rust code that is compiled directly from the original program through Wasm. Finally, we create a equivalence-checking harness based on the original program to verify equivalence of the two paths' outputs, and only after a successful verification we output a correct and maintainable Rust transpilation. In the following sections, we describe each component of *VERT* in further detail.

A. Untrusted Translation Pipeline: LLM

VERT produces the candidate translation by prompting an LLM to perform the translation to Rust. The LLM-based translation pipeline is shown in the right hand side of Figure 2.

LLMs often produce syntactically incorrect or ill-typed code. When prompting an LLM for Rust code, any slight mistake could cause the strict Rust compiler (**rustc**) to fail. Fortunately, **rustc** produces detailed error messages when compilation fails to guide the user to fix their program. We automatically repair these typing errors by re-prompting the LLM to fix it. While we omit the exact prompt for brevity, it is given in the first half of Figure 4 of the appendix.

B. Trusted Translation Pipeline: rWasm

Since the LLM output cannot be trusted on its own, we create an alternate trusted transpilation pipeline for generating a reference Rust program against which the LLM output is checked. Given in the left hand side of Fig. 2, the alternate pipeline does not need to produce maintainable code, but it needs to translate the source language into Rust using a reliably correct rule-based method. We use Wasm as the intermediate representation because many languages have compilers to Wasm, allowing it to serve as the common representation in the rule-based translation. Once the input programs are compiled to Wasm, we use rWasm [10], a tool that translates from Wasm to Rust by embedding the Wasm semantics in Rust source code. While the original authors intended rWasm as a sandboxing tool that leverages the memory safety properties of safe Rust, we use it to generate trusted Rust code with same semantics as the original input.

C. Equivalence Harness Generation

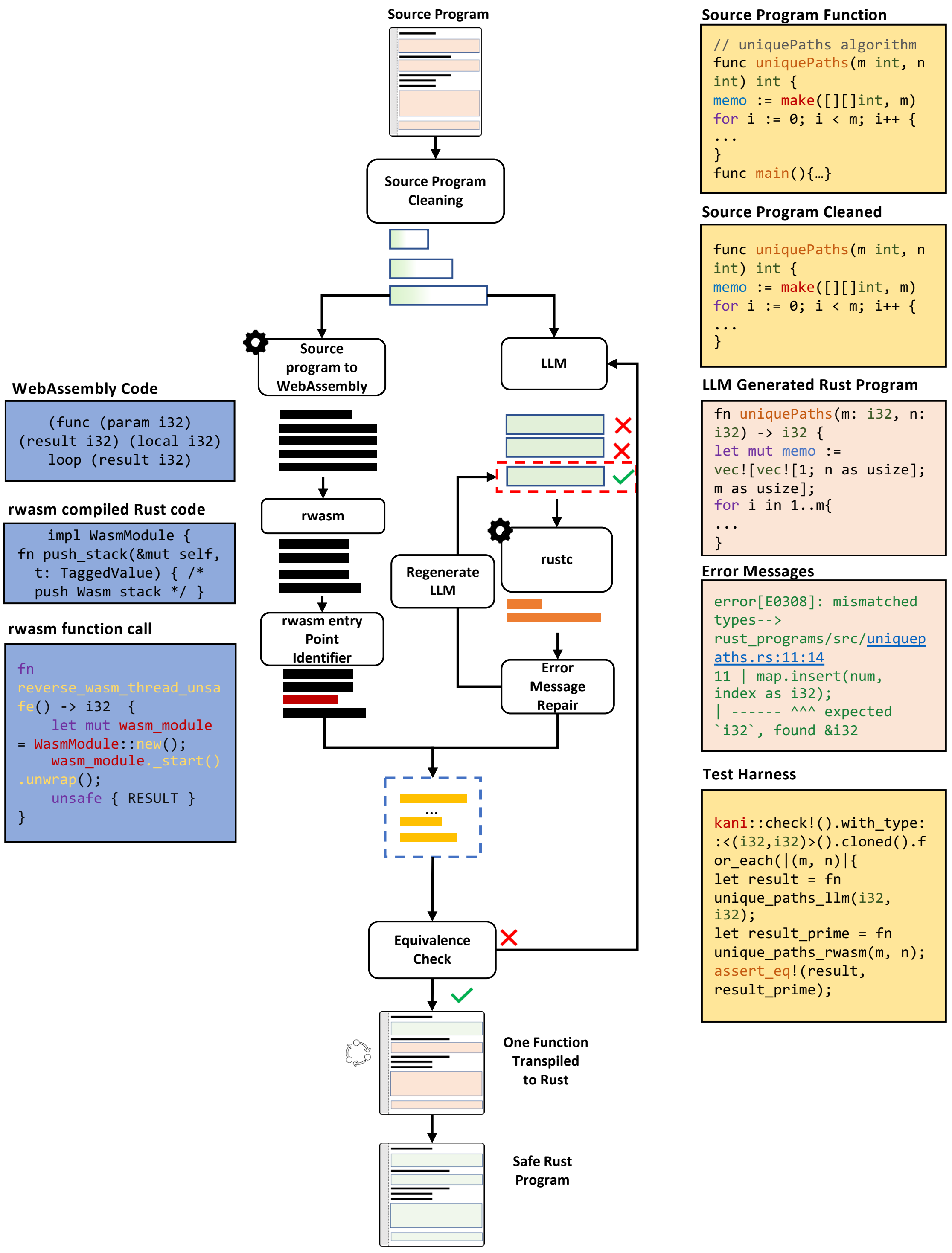
In our final step, we generate harnesses to check for equivalence given the input and output locations. We define equivalence here in functional terms: for all inputs, running both functions yields no crashes and identical outputs. To check this property holds, we automatically generate a wrapper to the Wasm function and a harness where the LLM-synthesized and wrapped rWasm functions are called with the same inputs, and the outputs are asserted to be equal. To ensure this equivalence holds for all inputs, we leverage model-checking with symbolic inputs and property-based testing with random inputs. For the remainder of this section, we refer to both of them together as "the input."

The wrapper consists of two parts: input injection, and output gathering. We replace constants inputs with the inputs of the harness like `input: i32`. After identifying the relevant variables using dependency analysis, we use globals in Rust to inject the inputs right at the location where the variables used to be. An example is given in Fig. 3, with `func_4` being the Wasm equivalent of the test. Note that, while this injection requires unsafe code, it is fine as this is only done in the oracle and the oracle is discarded once the equivalence is checked. We collect the output of the oracle program in the global variable `OUTPUT_1` and compare with the output of the LLM-generated program.

Given the low-level nature of the oracle program, we support complex types through their primitive parts. Given a struct or enum, we construct values of that type by abstracting the primitive parameters of that type and any required discriminants for enums. For types of finite size, this is sufficient. Moreover, we provide bounded support for handling vector types. The challenge here is to vary the length of the vector in the rWasm output, which is done using a fixed-length vector of varying inputs and then pruning the length down to the actual length dynamically. Our approach is sound and complete for primitive types, and by extension, any type that comprises solely of primitive types such as tuples of primitives. For unbounded types like vectors, hashmaps and user-defined types containing such, *VERT* synthesizes harnesses that generate inputs up to the size encountered in the sample function call. As a limitation, any divergences that require larger vectors than encountered will be missed.

D. Equivalence Checking

With the equivalence checking harness built, we must now drive the harness and check that the equivalence property holds for all inputs. *VERT* provides two equivalence checking techniques with increasing levels of confidence and compute cost. We use existing tools that operate on Rust to prove equivalence between the LLM-generated and the rWasm-generated oracle Rust programs. We use bolero, a Rust testing and verification framework that can check properties using both



Source Program Function

```
// uniquePaths algorithm
func uniquePaths(m int, n
int) int {
memo := make([][]int, m)
for i := 0; i < m; i++ {
...
}
func main(){…}
```

Source Program Cleaned

```
func uniquePaths(m int, n
int) int {
memo := make([][]int, m)
for i := 0; i < m; i++ {
...
}
```

LLM Generated Rust Program

```
fn uniquePaths(m: i32, n:
i32) -> i32 {
let mut memo :=
vec![vec![1; n as usize];
m as usize];
for i in 1..m{
...
}
```

Error Messages

```
error[E0308]: mismatched
types-->
rust_programs/src/uniquep
aths.rs:11:14
11 | map.insert(num,
index as i32);
| ------ ^^^ expected
`i32`, found &i32
```

Test Harness

```
kani::check!().with_type:
:<(i32,i32)>().cloned().f
or_each(|(m, n)|{
let result = fn
unique_paths_llm(i32,
i32);
let result_prime = fn
unique_paths_rwasm(m, n);
assert_eq!(result,
result_prime);
```

WebAssembly Code

```
(func (param i32)
(result i32) (local i32)
loop (result i32)
```

rwasm compiled Rust code

```
impl WasmModule {
fn push_stack(&mut self,
t: TaggedValue) { /*
push Wasm stack */ }
```

rwasm function call

```
fn
reverse_wasm_thread_unsa
fe() -> i32  {
    let mut wasm_module
= WasmModule::new();
    wasm_module._start()
.unwrap();
    unsafe { RESULT }
}
```

Fig. 2: *VERT* 's architecture, which takes as input a source program and produces a formally equivalent Rust program

```
static mut INPUT_1 = 0;
static mut OUTPUT_1 = 0;
impl WasmModule {
  fn func_4(&mut self, ) -> Option<i32> {
    // ...
    v0 = TaggedVal::from(unsafe {INPUT_1});
    self.func_5(v0); // Wasm representation
    unsafe {OUTPUT_1 = v0.try_as_i32()?;};
    // ...
  }
}
fn equvalence_checking_harness() {
  bolero::check!()
  .for_each(|(input: i32)| {
    let llm_fn_output = llm_generated();
    unsafe {INPUT_1 = input;};
    let mut wasm_module = WasmModule::new();
    wasm_module._start().unwrap();
    assert!(OUTPUT_1 ==  llm_fn_output);
  });
}
```

Fig. 3: Equivalence-checking harness for `func_4`.

Bounded Model Checking (BMC) [15], [16] and Property-Based Testing (PBT) [21].

1) BMC: We perform bounded verification with a combination of `bolero` and `kani`. We customize `bolero` to perform model checking through `kani` [42], a model-checker for Rust. When run with `kani`, `bolero` produces symbolic inputs rather than random concrete inputs. Executing the harness with symbolic inputs, we can cover the entire space of inputs in one run and the model-checker ensures the property holds for all possible inputs. Since symbolic execution does not know how many times loops are run, `kani` symbolically executes loops up to an user-provided bound. To prove soundness of this bound, `kani` uses *unwind checks*, asserting that the loop iteration beyond the bound is not reachable.

2) PBT: PBT works by randomly generating inputs to the function under test, running the harness with these inputs, and asserting the desired properties (e.g. equivalence between two functions). PBT repeatedly runs this procedure to check the property over the range of inputs. PBT is a valuable tool for catching implementation bugs, however it is generally infeasible to run PBT for long enough to exhaust all possible inputs to a program.

We run the equivalence-checking harness with PBT using `bolero` up to the time limit, generating random inputs and checking equivalence of the outputs. If the candidate diverges from the oracle, then PBT will return the diverging input as a counterexample. If no counterexample is found within the time limit, we say this candidate passes PBT.

E. Re-Prompting with Model-Checking Error

When the equivalence checking fails, we use the LLM to repair the code and repeat equivalence checking. We collect the error messages and counter examples from prior failed checks as part of the prompt. Specifically, we ask the LLM to consider the specific inputs that caused a test or verification failure from the previous iterations. We observed that providing specific inputs as information to the LLM results in subtle bug fixes within the program output. The prompt template is given in Figure 4.

```
{Original code}
Safe Rust refactoring of above code in {language
      }.
Use the same function name, same argument and
      return types.
Make sure the output program can compile as a
      stand alone program.
// If there exists rustc error message from
      prior failed compilations
Ensure that the output avoids the above error
      message: {error_msg}.
// If there exists counter examples from prior
      failed equivalence checking
Test that outputs from inputs {counter_examples}
       are equivalent to source program.
```

Fig. 4: LLM Prompt template, which changes according to previous error messages or failed inputs.

```
error[E0308]: mismatched types-->
11 | map.insert(num,index as i32);| -- ^^^
      expected '&i32', found i32
help: consider borrowing here: '&num'
```

Fig. 5: Mismatched type error

F. Compilation Errors

`rustc` produces detailed error messages when compilation fails to guide the user to fix their program. We create an automatic repair system based on **`rustc`** error messages. For example, consider the typing error in Figure 5. Typing error messages in Rust generally have a similar structure. In particular, error messages are usually of the form `expected type a, found b`. Figure 5 shows the LLM generate a pass-by-reference variable `&i32` while **`rustc`** expects a pass-by-value `i32`. Using the compiler message's error localization and suggestion line (characterized by the keyword `help:`), we replace the variable `num` by `&num`.

For compilation errors that are specific to the program, we use the **`rustc`** error message suggestion line to generate a repair. In Figure 6, which shows the error message for an immutable assignment, the suggestion line indicates that if the variable `x` is converted to a mutable object, the immutable assignment error would be solved. Using this suggestion line, we replace `x` by `mut x`, and observe that the program compiles.

IV. Evaluation

We present our evaluation setup for the following research questions.

RQ1. How does *VERT* perform? We evaluate our technique's performance on a benchmark dataset, showing that *VERT* significantly increases the number of verified equivalent transpilations vs. using the LLM by itself. We further conduct an ablation analysis, which shows that our prompting and error-guided refinement helps produce more well-typed and more correct programs. Finally, we measure the runtime performance of each part of *VERT*, showing that time costs of error-guided

```
error[E0384]: cannot assign to immutable
     argument 'x'
1  | fn reverse(x: i32) -> i32 {
   |- help: consider making this binding mutable
       : 'mut x'
...
16 |         x = x / 10;
   |         ^^^^^^^^^^ cannot assign to
        immutable argument
```

Fig. 6: Immutable assignment error

refinement is reasonable and *VERT* spends most of the time in verification.

RQ2. To what extent does *VERT* produce safe and idiomatic Rust transpilations? To evaluate *VERT*'s ability to produce safe Rust, we collect programs from real world C projects that make use of pointers. In addition, we report on the frequency of linter warnings of transpilations produced by *VERT*, and compare the number of lines of code produced by *VERT*, `rWasm`, and CROWN [52], a rule-based C to Rust transpiler. *VERT*'s transpilations do not produce linter warnings, and has far fewer lines of code than the other approaches.

A. Setup

1) LLMs: We use the following LLMs to generate the candidate transpilations in *VERT*:

- **TransCoder-IR [39] (baseline)**: A language model trained by low-level compiler intermediate representations (IR) for the specific purpose of programming language translation. TransCoder-IR improves upon the TransCoder model [36] by incorporating IR into the training data and decompiling into IR as a training target. Both TransCoder and TransCoder-IR are trained on roughly 2.8 million repositories from GitHub[2]. Since TransCoder-IR's input is the original code alone and no prompt is taken, we do not perform error-guided few-shot prompting. To the best of our knowledge, TransCoder-IR is the only LLM-based general transpilation tool for Rust. Therefore, we use TransCoder-IR as baseline for our evaluation.
- **CodeLlama-2 [35]**: A 13B parameter model initialized from Llama-2 [41], then further fine-tuned on 500 billion tokens of code data.
- **StarCoder [26]**: A 15.5B parameter model trained on 1 trillion tokens sourced from The Stack [24].
- **Anthropic Claude-2 [4]**: A production-grade, proprietary LLM accessible through Anthropic's APIs with roughly 130 billion parameters. Claude-2 costs about \$0.0465 per thousand tokens.

2) LLM Prompting: We evaluate *VERT* operating in three different modes: single-shot, few-shot, and few-shot counter examples. *Single-shot* means that *VERT* uses the LLM *once* to create a single candidate transpilation, and then proceeds directly to verification. If verification fails, then *VERT* does not attempt to regenerate. *Few-shot* means that, if verification fails, then *VERT* will prompt the LLM to regenerate the transpilation repeatedly. In each iteration, we apply the syntactic repair described in Section III-A to the output of the LLM. Finally, *few-shot counter examples* means that we use counter examples produced by previous failed verification attempts as part of the LLM's few-shot learning, as described in Section III-E. *Few-shot counter examples* only works for instruction-tuned models. We re-prompt the LLM up to 20 times for few-shot modes. For each LLM and each mode of *VERT*, we report the number of transpilations that compiled and that passed the various verification modes. Prompts for translation are given in Figure 4.

We omit *single-shot* for CodeLlama2 and StarCoder because our program repair technique does not work in a *single-shot* setting (see section III-E), resulting in a failure to produce well-typed Rust code for open-source and non-Rust fine-tuned LLMs. We omit *few-shot counter examples* for CodeLlama2 and StarCoder because neither LLM was instruction-tuned (i.e., further trained for conversations), which is required for counter example feedback. Finally, we perform *single-shot, few-shot, and few-shot with counter examples* with Anthropic Claude-2 to investigate how each part of *VERT* impacts an instruction-tuned LLM's ability to perform Rust transpilation.

3) Benchmark selection: We draw our benchmarks from two sources. Our first source is the benchmark set from TransCoder-IR [39], which is primarily made up of competitive program solutions. In total, this benchmark set contains 852 C++ programs, 698 C programs, and 343 Go programs. We choose this dataset to avoid potential data-leakage (i.e., LLM memorization) [8] in our evaluation. We note that the Rust programs produced by TransCoder-IR were released after June 2022, which is the training data cutoff date of our chosen LLMs [4], [26], [35]. We select programs from the TransCoder-IR dataset that can directly compile to Wasm using `rWasm`. After filtering, we collect a benchmark set of 569 C++ programs, 506 C programs, and 341 Go programs. These types of benchmarks are common for evaluating LLMs' coding ability. However, the programs themselves often do not make extensive use of pointers, so they do not adequately challenge *VERT*'s ability to generate safe Rust.

To provide insight into *VERT*'s ability to write safe Rust, we gather pointer-manipulating C programs from prior work on C to Rust transpilation [2], [20], [52] that have tests (which enable us to locate Bolero harness entry points), and have unbounded loops over pointers (to test BMC verifiability). The benchmarks in these prior works use open-source programs written before our chosen LLM's training data cutoff (June 2022). To avoid LLM data-leakage (i.e., the LLM having already seen and memorized the C and Rust pairings), we only select programs contained in repositories created or committed after June 2022 to transpile to Rust, and have no Rust equivalent in any open-source repositories. We manually label the input output pairs for each snippet for verifying equivalence on the transpiled Rust programs. Many of the benchmarks we select involve multiple functions. The explicit goal when selecting benchmarks from these projects is to discover the limitations of *VERT* in terms of

[2]https://console.cloud.google.com/marketplace/details/github/github-repos

writing safe Rust, therefore we gather benchmarks of increasing complexity in terms of the number of pointer variables, and the number of functions in the benchmark. We present several complexity metrics for the benchmarks and discuss them in more detail in Section IV-C. In total, we evaluate our approach on **569 C++** programs, **520 C** programs, and **341 Go** programs.

B. Evaluation Metrics

Neural machine translation (NMT) approaches use metrics that measure token similarity between the expected output and the actual output produced by the LLM. While these approaches are often meaningful when applied to natural language, for programming languages, small differences in the expected output and actual output could result in different compilation or run-time behavior. Conversely, two programs that share very few tokens (and hence have a very low text similarity score) could have identical compilation or run-time behavior. Metrics based off of passing tests have been proposed, such as the computational accuracy (CA) metric [36], [39]. However, the CA metric has no accepted standard for the number of required passing tests, or the coverage and quality of the unit tests.

To improve upon the metrics used for prior NMT approaches and remove the overhead of writing high-coverage unit tests, we use formal methods to measure the correctness of the output. In particular, we use property based testing (PBT) and bounded model checking (BMC). We insert the LLM-generated code and `rWasm`-generated code in an equivalence-checking harness that asserts equal inputs lead to equal outputs. An example of such a harness is given in Figure 3. Since the two metrics used are significantly slower than checking a series of unit tests, we set a time limit for our metrics. For both metrics, we set a 120 seconds limit. For PBT, no counterexamples within 120 seconds counts as success. For BMC, success requires establishing verified equivalence within 120 seconds. If either of the step fails, *VERT* terminates.

We present results on the TransCoder-IR benchmarks in Table I. We apply *VERT* in three different modes: *single-shot* (transpile with the first LLM generation), *few-shot* (prompt repeatedly with repair but without feedback), and *few-shot counter examples* (using counter examples for LLM feedback). We perform *single-shot* for Transcoder-IR (baseline) to replicate results from prior work. Transcoder-IR does not have support for few-shot instruction tuning. We perform *few-shot* on CodeLlama2 and StarCoder to investigate the effectiveness of few-shot and error-based repair on open-source, non-instruction tuned LLMs.

The experiments for all benchmarks were run on an Ubuntu 22 instance with 32 Intel Xeon 2.30 GHz processors, 240 GB RAM, and 4 Tesla V100 GPUs.

C. Results

RQ1. How does *VERT* perform?

As seen in table I, *VERT* with Claude-2 produces PBT-passing programs for 49% of all inputs (52% for C++, 40% for C, and 56% for Go) and BMC-passing programs for 40% of all inputs (41% for C++, 37% for C, and 46% for Go). *VERT* with Claude-2 in addition improving baseline Claude-2, *VERT* with Claude-2 greatly outperforms the three prior translation tools.

VERT with both CodeLlama2 and StarCoder also improve upon baseline on the number of programs passing compilation, PBT, and BMC. We observe that few-shot learning with rule-based repair on general code-based LLMs can perform more accurate Rust transpilations than an LLM trained with transpilation as its main target. To confirm that *VERT* yields a statistically significant improvement over baseline, we perform a Wilcoxon rank test [46], which indicates whether the metric performance between *VERT* and baseline are statistically different. We use the Wilcoxon signed-rank test to see if the statistically significant difference is also positive (i.e., our approach is different and better as measured by our three metrics). We observe Wilcoxon signed-rank p-values ranging from 1×10^{-5} to 4×10^{-5} for PBT and BMC.

Table I shows the transpilation results across CodeLlama-2 and StarCoder in a few-shot setting. *VERT* with CodeLlama-2 and StarCoder improve over Transcoder for compilable Rust translations. As the size of the LLM increases, the overall correctness of its compilable output increases as well. Few-shot prompting for Claude-2 yields a greater improvement over single-shot compared to our repair technique. Few-shot learning with counter examples of failed previous verification attempts provides the largest improvements on BMC. Modern LLMs that are instruction-tuned can learn to generate more correct program when given specific test failures in few-shot settings. For a more detailed analysis of the contribution of each of *VERT*'s components to performance, see IV-C2.

To disaggregate the gains from using Claude-2 with the gains from using *VERT*, we provide a comparison between single shot, few shot, and *VERT* on Claude-2 in Table II. While Claude-2 is good at producing well-typed Rust programs, it struggles with correctness producing PBT-passing programs at roughly 1/4 the rate of *VERT* and almost never succeed in passing BMC.

`kani`'s BMC uses an average of 52 seconds per program, and `bolero`'s property testing uses an average of 25 seconds per program. The average runtime of each of *VERT*'s components can be found in the Table III. Kani verification timeout is the main reason for lower BMC pass rates.

1) Ablation study: Table I shows the results across all our evaluated LLMs in the three prompting modes. StarCoder has slightly fewer transpilations than CodeLlama-2 passing compilation, but more transpilations than CodeLlama-2 passing BMC. We note that as the size of the LLM increases, the overall correctness of its compilable output increases as well. StarCoder's results are limited by its ability to pass compilation, even with *VERT*'s `rustc` error guided program repair in place. *VERT* with StarCoder compiles 47% fewer programs for C++, 41% fewer for C, and 63% fewer programs for Go as compared to *VERT* with Claude-2. An industry-grade LLM with more trainable parameters and a larger training dataset performs significantly better for our metrics due to its ability for generating compilable Rust code.

TABLE I: *VERT* performance across with different LLMs and modes. PBT is the number of programs that passed Property-Based Testing and compilation. BMC is the number that passed Bounded Model Checking of those that passed PBT.

LLM	Source Lang	Technique	Compiled	PBT	BMC
Transcoder-IR	C++ (569)	Single-shot	107	23	3
	C (520)	Single-shot	101	14	1
	Go (341)	Single-shot	24	3	0
CodeLlama2 13B	C++ (569)	Few-shot	307	25	6
	C (520)	Few-shot	160	18	4
	Go (341)	Few-shot	104	15	2
StarCoder 15.5B	C++ (569)	Few-shot	253	52	7
	C (520)	Few-shot	179	46	5
	Go (341)	Few-shot	134	21	2
Claude-2 130B	C++ (569)	with ctr. examples (VERT)	**539**	**295**	**233**
	C (520)	with ctr. examples (VERT)	**339**	**209**	**193**
	Go (341)	with ctr. examples (VERT)	**317**	**195**	**159**

TABLE II: *VERT* performance with Claude-2 on 3 different prompting modes.

LLM	Source Lang	Technique	Compiled	PBT	BMC
Claude-2 130B	C++ (569)	Single-shot (Baseline)	240	55	6
		Few-shot	539	292	41
		with ctr. examples (VERT)	**539**	**295**	**233**
	C (520)	Single-shot (Baseline)	239	49	6
		Few-shot	339	195	29
		with ctr. examples (VERT)	**339**	**209**	**193**
	Go (341)	Single-shot (Baseline)	126	26	3
		Few-shot	276	157	39
		with ctr. examples (VERT)	**317**	**195**	**159**

We observe that *VERT* using few-shot with either StarCoder or Claude-2 yields better transpilation across all our three languages and three metrics. In particular, few-shot with Claude-2 passes 43% more PBT checks for C++, 46% more for C, and 43% more for Go as compared to single-shot with Claude-2. Few-shot with Claude-2 passes 6% more BMC checks for C++, 4% more for C, and 12% more for Go as compared to single-shot with Claude-2. We find that the few-shot prompting for Claude-2 yields a greater improvement over single-shot compared to our repair technique. Few-shot learning with counter examples of failed previous verification attempts provides the largest improvements on BMC. Modern LLMs that are instruction-tuned can learn to generate more correct program when given specific test failures in few-shot settings.

2) Runtime: Table III shows the average runtime of each of *VERT*'s components. In the non-timeout failure cases (i.e., `kani` does not establish equivalence within 120s), `kani`'s BMC uses an average of 52 seconds per program, and `bolero`'s property testing uses an average of 25 seconds per program. Of our failure cases, 17% of were non-timeout. Of the LLMs, both CodeLlama-2 and StarCoder use about 3 seconds per each prompt attempt, and Anthropic Claude-2 about 2 seconds. Not counting the failure cases (i.e., the LLM does not generate any program that can pass equivalence after 20 attempts), we observe an average of 15 tries before the LLM can achieve compilation. Transcoder-IR uses 8 seconds on average per transpilation, which we prompt only one time as the baseline of our evaluation.

TABLE III: *VERT*'s average runtime per component for a Single-program translation

Component type	Component	Time (s)
LLM	Transcoder-IR	8
	CodeLlama-2	43
	Starcoder	45
	Anthropic Claude	30
Rust compilation	`rustc`	< 1
	Error guided	1
	`rwasm`	< 1
Testing and verification	PBT	25
	Bounded-ver.	52

3) Pointer-manipulating C programs: Table IV gives the number of pointer variables, function definitions, and LoC in each benchmark. The `avl_*` benchmarks are taken from a library that implements an AVL tree. The `brotli_*` benchmarks are from the Brotli compression library. The `buffer_*` benchmarks allocate and resize a buffer respectively. The `ht_*` benchmarks compute a hash key, and create a hash table, respectively, the `libcsv_*` benchmarks initialize a struct with pointer variables, and retrieve members from the struct. `libtree` determines if an array of pointers to `int64`s is sorted. `urlparser` parses a URL.

RQ1 Summary

VERT with CodeLlama2, StarCoder, and Anthropic Claude-2 can produce more compiling, PBT, and BMC passing Rust transpilations than baseline, with increasing effectiveness as model size grows. *VERT* with Claude-2 can pass BMC for 40% more programs for C++, 37% for C, and 47% for Go as compared to baseline.

RQ2. To what extent does *VERT* produce safe and idiomatic Rust transpilations? To measure *VERT*'s ability to generate safe Rust, we use *VERT* few-shot + repair with Claude-2 to transpile the 14 pointer-manipulating C programs. Table IV presents several metrics that provide a rough idea of the complexity of the benchmarks.

VERT can produce transpilations for 7 of the 14 C programs that pass PBT, and 2 of those can pass BMC. Two benchmarks cannot pass compilation due to Rust's borrow checker (`ht_create` and `urlparser`). In particular, *VERT* was unable to generate safe Rust on `ht_create` due to transferring a variable into byte representation in two lines of code. The results show that *VERT* tends to struggle as the programs get larger, and have more pointer variables across multiple functions. On smaller programs, the LLM can still determine basic ownership requirements. For example, it can determine if a variable or parameter reference needs a `mut` permission.

To evaluate Rust idiomaticity, we compare lines of code in the transpilations produced by *VERT*, `rWasm`, and CROWN [52], the rule-based C to Rust transpiler. After running `rustfmt` [38], the official formatter of Rust, CROWN's output is more than **5x** larger than *VERT*, and `rWasm`'s output is more than **10x** as large. Given the strong negative association between LoC and code readability [11], we conclude that *VERT*'s outputs are more idiomatic than CROWN's and `rWasm`'s. To further evaluate idiomaticity, we run Clippy[3], the official linter of Rust, on *VERT*'s transpilations. Clippy checks Rust code for potential issues in the categories of correctness (e.g. unsigned int is greater than 0), performance (e.g. unnecessarily using a Box type), stylistic (e.g. unnecessary borrowing), and conciseness (e.g. unnecessary type casting). On average, Clippy produces **10.9** warnings per function for CROWN, and **372** warnings per function for `rWasm`. By contrast, Clippy produces **0** warnings on *VERT*'s transpilations, thus we conclude that they are reasonably idiomatic. Although all 14 of CROWN's outputs pass compilation, none can pass PBT or BMC as CROWN does not provide test-harnesses for any of its outputs.

VERT targets the broader and more difficult problem of polyglot, verified translation to Rust, whereas CROWN only targets unsafe to safe rust (after running C2Rust [2]) without testing or verification. The output from *VERT* is more Rust-native than CROWN's, using standard Rust types while CROWN and C2Rust use C-foreign types/functions. The lack of reliance on C-foreign functions is a qualitative strength. The output from *VERT* is more self-contained and reviewable by Rust programmers [6]. *VERT* can catch buggy C API calls in the input program instead of translating the incorrect API calls to Rust $libc$ :: calls that remain buggy.

[3]https://doc.rust-lang.org/clippy/

RQ2 Summary

VERT can produce transpilations for 7 of the 14 C programs that pass PBT, and 2 of those can pass BMC. *VERT* produces far-more idiomatic Rust code: 5X fewer LoC than CROWN, 10x fewer LoC than `rWasm`, and its transpiled Rust programs do not trigger any Linter warnings.

V. Limitations and Discussion

1) Limitations of the Equivalence Checking: VERT's equivalence checking is performed in 2 stages with increasing guarantees: PBT and BMC. These phases do not provide any exhaustive guarantees. The quality of PBT depends heavily on the inputs generated, and thus the seed used to generate the random inputs. Bounded verification may miss bugs or divergences in LLM and `rWasm`-generated programs if they occur beyond the loop unwinding bound.

During our evaluation we did not find any examples of missed bugs, but BMC failed due to unbounded loops. Takashima [40] extended our work to manually verify 3 out of 5 benchmarks that timed out in our RQ2 using Verus [25], an auto-active verifier. Verus at the time did not support the remaining two. This suggests that these translations were correct but could only be verified equivalent using loop invariants.

2) Trusted Computing Base (TCB) [37]: The TCB is the set of all software and hardware the user must trust in order to trust the result of the full verification. We note that *VERT* has a large TCB compared to traditional formal verification frameworks like Coq [17] or Isabelle/HOL [30]. To trust the results of *VERT*, the user must trust Kani, its CBMC backend, the original language's Wasm compiler, `rWasm`, the Rust compiler, and all of the transitive dependencies of the respective software. We posit that this TCB is still more trustworthy and easily checked than the LLM.

3) Scaling Beyond Single Programs: VERT focuses on translating single programs. A potential line of future work is to scale it to full projects by cutting projects along module boundaries to a programs that *VERT* can handle, run *VERT* at each stage to translate each module and verify both individual translations and compositions.

4) LLMs: Threats to *internal validity* are concerned with the degree of confidence on our dependent variables and our results. Using LLMs as part of our tool exhibits the potential for *training data contamination* (i.e. our evaluation datasets could be included in the LLM training data). To mitigate this threat, we select and evaluate our tool on Rust solutions written and labeled after June 2022. Furthermore, our evaluation is primarily to show that our iterative repair procedure can significantly improve the number of correct transpilations produced by an LLM, and that we can verify equivalence between `rWasm` Rust and the LLM produced Rust. This result

TABLE IV: Benchmark information and results on the 14 C pointer manipulation programs. The symbols ✔, ✗, and ● indicate pass, fail (with counterexample), and timeout.

Benchmark	Funcs	Pointer vars	LOC	Compiled	PBT	BMC
avl_minvalue	1	4	17	✔	✔	✔
avl_insert	2	4	30	✔	✔	●
avl_rotate	3	7	32	✔	✔	●
avl_delete	4	27	111	✗	✗	✗
brotli_parse_int	1	2	15	✔	✔	●
brotli_filesize	1	1	28	✔	✗	✗
buffer_new	1	3	16	✔	✔	●
buffer_resize	3	3	22	✔	✗	✗
ht_hashkey	1	2	13	✔	✔	✔
ht_create	1	3	36	✗	✗	✗
libcsv_get_opts	1	1	29	✔	✗	✗
libcsv_init	1	4	55	✗	✗	✗
libtree	1	1	7	✔	✔	●
urlparser	9	28	158	✗	✗	✗

is not affected by training data contamination. Thus we believe potential training data contamination does not invalidate our results. Threats to *external validity* lie in whether results on our benchmarks will generalize to real-world contexts. One of our benchmark sets is a collection of competitive programming solutions across three languages, used by Transcoder-IR [39]. Although the collected programs cover a wide range of input and output types, real-world code bases are often much more complicated than competitive solution programs. To reduce this threat, we evaluate on selected functions from real world programs taken from the *CROWN* and *Laertes* benchmark [20], [52] that contain complex, unbounded loops over pointers.

VI. Related Work

a) Language model transpilers: Recent advances have shown that LLMs can perform code completion [19] and generate code based on natural language [34]. Transcoder and Transcoder-IR [36], [39] use unsupervised machine translation to train neural transcompilers. As both Transcoder versions are trained on program translation pairings, they perform better on program translation tasks than similar sized but generic auto-regressive LLMs. However, recent work shows that LLMs can generate buggy and vulnerable programs [9], [13], [32]. Transcoder-IR [39] show in their evaluation that a significant portion of their translated code does not compile, especially for target programming languages that are underrepresented in their training data (e.g., Rust). While several approaches have been developed to improve the quality of the output with prompting [18], [47], *VERT* helps generate memory-safe code and establish equivalence, to safely harness the code output of an LLM.

b) Rust transpilers: C2Rust [2] automatically converts large-scale C programs to Rust while preserving C semantics. Several works aim to reduce the amount of *unsafe* code in C2Rust's output. Emre et al. [20] and Zhang et al. [52] use ownership and type analysis, and Ling et al. [27] removes *unsafe* code by shrinking the blocks to a minimum required by the Rust compiler.

While the C2Rust family of translators refactor the Rust output, refactoring may be done on the C++ side as well. CRAM [22] is a semi-automatic technique that refactors existing C++ code to transpilation friendly subsets C++, then produces human-friendly and idiomatic Rust. VERT is fully automatic.

Citrus [3] and Bindgen [1] both generate Rust FFI bindings from C libraries, and produce Rust code without preserving C semantics. Bosamiya et al. [10] embedded WebAssembly (Wasm) semantics in safe Rust code for the Rust compiler to emit safe and executable Rust code. Bosamiya et al. generate only safe Rust, and no stage of compiler needs to be further verified or trusted to achieve safety. *VERT* focuses on generating idiomatic and maintainable Rust code with Rust semantics directly.

VII. Conclusion

Rust's improvements on security and performance over other languages have prompted recent research on transpiling existing code-bases to Rust. However, rule-based transpilation approaches are unidiomatic and fail to follow the target language conventions. ML-based approaches (i.e., LLMs) cannot provide formal guarantees, thus removing the security benefits of Rust. We introduce *VERT*, which uses both LLMs and formal verification to transpile verified and idiomatic Rust programs.

VIII. Acknowledgment

We thank the reviewers for constructive feedback. This work was partially funded by the National Science Foundation via grants CCF1901136 and CNS2114148 and by the Oscar M. Ruebhausen Fund.

References

[1] bindgen. https://github.com/rust-lang/rust-bindgen
[2] C2rust. https://c2rust.com/
[3] citrus. https://gitlab.com/citrus-rs/citrus
[4] claude. https://www.anthropic.com/index/introducing-claude
[5] Akinyemi, S.: Awesome WebAssembly Languages (Aug 2023), https://github.com/appcypher/awesome-wasm-langs, original-date: 2017-12-15T11:24:02Z
[6] Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe Rust? Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–27 (2020)

[7] Barr, J.: Firecracker – lightweight virtualization for serverless computing. AWS News Blog https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/

[8] Biderman, S., PRASHANTH, U., Sutawika, L., Schoelkopf, H., Anthony, Q., Purohit, S., Raff, E.: Emergent and predictable memorization in large language models. Advances in Neural Information Processing Systems **36** (2024)

[9] Black, S., Gao, L., Wang, P., Leahy, C., Biderman, S.: GPT-Neo: Large scale autoregressive language modeling with Mesh-Tensorflow (2021)

[10] Bosamiya, J., Lim, W.S., Parno, B.: *Provably-Safe* multilingual software sandboxing using *WebAssembly*. In: USENIX Security. pp. 1975–1992 (2022)

[11] Buse, R.P.L., Weimer, W.R.: Learning a metric for code readability. IEEE Trans. Softw. Eng. **36**(4), 546–558 (July 2010). https://doi.org/10.1109/TSE.2009.70

[12] BV, T.S.: TIOBE Index (2025), https://www.tiobe.com/tiobe-index/

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

[14] Cimpanu, C.: Microsoft: 70 percent of all security bugs are memory safety issues. ZDNET (2019), https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues

[15] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**, 7–34 (2001)

[16] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15

[17] Coquand, T., Huet, G.P.: Constructions: A higher order proof system for mechanizing mathematics. In: Invited Lectures from the European Conference on Computer Algebra—Volume I. pp. 151–184. EUROCAL '85, Springer (Apr 1985)

[18] Deligiannis, P., Lal, A., Mehrotra, N., Rastogi, A.: Fixing Rust compilation errors using LLMs (Aug 2023). https://doi.org/10.48550/arXiv.2308.05177, arXiv:2308.05177 [cs]

[19] Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., Roy, S.: Program synthesis using natural language. In: International Conference on Software Engineering. pp. 345–356 (2016)

[20] Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.: Translating C to safer Rust. Proceedings of the ACM on Programming Languages (OOPSLA) **5**, 1–29 (2021)

[21] Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes **22**(4), 74–80 (1997)

[22] Grammatech Inc.: Migration to memory safe code (2024), https://www.grammatech.com/cyber-security-solutions/migration-to-memory-safe-code/

[23] Jung, R.: Understanding and Evolving the Rust Programming Language. Ph.D. thesis, Universität des Saarlandes (2020)

[24] Kocetkov, D., Li, R., Allal, L.B., Li, J., Mou, C., Ferrandis, C.M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., et al.: The stack: 3 TB of permissively licensed source code. arXiv preprint arXiv:2211.15533 (2022)

[25] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying Rust programs using linear ghost types. Proc. ACM Program. Lang. (OOPSLA) **7** (2023). https://doi.org/10.1145/3586037

[26] Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al.: StarCoder: May the source be with you! arXiv preprint arXiv:2305.06161 (2023)

[27] Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E.: In Rust we trust: a transpiler from unsafe C to safer Rust. In: International Conference on Software Engineering: Companion Proceedings. pp. 354–355. ACM/IEEE (2022). https://doi.org/10.1145/3510454.3528640

[28] Min, M.J., Ding, Y., Buratti, L., Pujar, S., Kaiser, G., Jana, S., Ray, B.: Beyond accuracy: Evaluating self-consistency of code LLMs. In: International Conference on Learning Representations (ICLR) (2023)

[29] Ni, A., Ramos, D., Yang, A.Z., Lynce, I., Manquinho, V., Martins, R., Le Goues, C.: SOAR: A synthesis approach for data science API refactoring. In: International Conference on Software Engineering (ICSE). pp. 112–124. IEEE (2021)

[30] Nipkow, T., Wenzel, M., Paulson, L.C., Goos, G., Hartmanis, J., Van Leeuwen, J. (eds.): Isabelle/HOL, LNCS, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9, http://link.springer.com/10.1007/3-540-45949-9

[31] Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.: Lost in translation: A study of bugs introduced by large language models while translating code. In: International Conference on Software Engineering (ICSE). pp. 1–13 (2024)

[32] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: An empirical cybersecurity evaluation of GitHub Copilot's code contributions. ArXiv abs/2108.09293 (2021)

[33] von Perponcher-Sedlnitzki, G., Christian, P.: Integrating the future into the past: Approach to seamlessly integrate newly-developed Rust-components into an existing C++-system. Ph.D. thesis, Technische Hochschule Ingolstadt (2024)

[34] Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Programming Language Design and Implementation. pp. 419–428 (2014)

[35] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al.: Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)

[36] Roziere, B., Lachaux, M.A., Chanussot, L., Lample, G.: Unsupervised translation of programming languages. Advances in Neural Information Processing Systems **33** (2020)

[37] Rushby, J.: The design and verification of secure systems. In: Eighth ACM Symposium on Operating System Principles (SOSP). pp. 12–21 (Dec 1981), (ACM *Operating Systems Review*, Vol. 15, No. 5)

[38] Rust Contributors: rustfmt, https://github.com/rust-lang/rustfmt

[39] Szafraniec, M., Roziere, B., Leather, H., Charton, F., Labatut, P., Synnaeve, G.: Code translation with compiler representations. arXiv preprint arXiv:2207.03578 (2022)

[40] Takashima, Y.: Testing and Verifying Rust's Next Mile (4 2024). https://doi.org/10.1184/R1/25451383.v1, https://kilthub.cmu.edu/articles/thesis/Testing_and_Verifying_Rust_s_Next_Mile/25451383

[41] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)

[42] VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in Rust. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 321–330 (2022)

[43] Vaughan-Nichols, S.: Linus Torvalds: Rust will go into Linux 6.1. ZDNET https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

[44] Weisz, J.D., Muller, M., Houde, S., Richards, J., Ross, S.I., Martinez, F., Agarwal, M., Talamadupula, K.: Perfection not required? Human-AI partnerships in code translation. In: Intelligent User Interfaces (IUI). pp. 402–412 (2021)

[45] Weisz, J.D., Muller, M., Ross, S.I., Martinez, F., Houde, S., Agarwal, M., Talamadupula, K., Richards, J.T.: Better together? An evaluation of AI-supported code translation. In: 27th International Conference on Intelligent User Interfaces. pp. 369–391 (2022)

[46] Woolson, R.F.: Wilcoxon signed-rank test. Wiley Encyclopedia of Clinical Trials pp. 1–3 (2007)

[47] Wu, X., Cheriere, N., Zhang, C., Narayanan, D.: RustGen: An augmentation approach for generating compilable Rust code with large language models. Workshop on Challenges in Deployable Generative AI at International Conference on Machine Learning (ICML) (2023)

[48] Xu, Z., Jain, S., Kankanhalli, M.: Hallucination is inevitable: An innate limitation of large language models. arXiv preprint arXiv:2401.11817 (2024)

[49] Yang, A.Z.: SOAR: Synthesis for open-source API refactoring. In: Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. pp. 10–12 (2020)

[50] Yang, A.Z., Kolak, S., Hellendoorn, V.J., Martins, R., Goues, C.L.: Revisiting unnaturalness for automated program repair in the era of large language models. arXiv preprint arXiv:2404.15236 (2024)

[51] Zhang, H.: 2022 Review | The adoption of Rust in business (Jan 2023), https://rustmagazine.org/issue-1/2022-review-the-adoption-of-rust-in-business

[52] Zhang, H., David, C., Yu, Y., Wang, M.: Ownership guided C to Rust translation. In: Computer Aided Verification (CAV). LNCS, vol. 13966, pp. 459–482. Springer (2023)