VERT: Verified Equivalent Rust Transpilation with Few-Shot Learning

AIDAN Z.H. YANG*†, Carnegie Mellon University, USA
YOSHIKI TAKASHIMA*†, Carnegie Mellon University, USA
BRANDON PAULSEN, Amazon Web Services, USA
JOSIAH DODDS, Amazon Web Services, USA
DANIEL KROENING, Amazon Web Services, USA

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 safe and correct 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 can theoretically produce *correct* transpilations that maintain input-output equivalence to the original, they often yield unreadable Rust code that uses unsafe subsets of the Rust language. On the other hand, while LLM-based approaches typically produce more readable, maintainable, and safe code, they do not provide any guarantees about correctness. In this work, we present *VERT*, a tool that can produce readable Rust transpilations with formal guarantees of correctness. *VERT*'s only requirement is that there is Web Assembly compiler for the source language, which is true for most major languages. *VERT* first uses the Web Assembly compiler to obtain an *oracle* Rust program. In parallel, *VERT* uses an LLM to generate a readable candidate Rust program. This candidate is verified against the oracle, and if verification fails, we regenerate a new candidate transpilation until verification succeeds. We evaluate *VERT* by transpiling a suite of 1,394 programs taken from competitive programming style benchmarks. Our results show that *VERT* significantly improves an LLM's ability to generate correct Rust transpilations. Combining Anthropic's Claude-2 and *VERT* increases Rust transpilations passing property-based testing from 31% to 54% and bounded model-checking from 1% to 42% compared to using Claude alone. In addition, we evaluate *VERT*'s ability to generate non-trivial safe Rust on programs taken from real-world C projects that make significant use of pointers. Our results provide insights into the limitations of LLMs to write safe Rust.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Model checking**; • **Computing methodologies** → *Natural language processing*.

1 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. Microsoft estimates that 70% of all their security bugs are due to memory-safety issues [16], which could be mostly or entirely eliminated if the code were written in Rust. Citing Rust's security benefits, Rust has been used in major open source projects such as Firecracker [8], and Linus Torvalds recently announced Rust will be a supported language for Linux kernel development [50].

Rust's security and performance benefits have fueled interest in automatically transpiling existing code written in other languages into Rust [2, 45]. Existing works on transpilation broadly fit into two categories: *rule-based* and *large language model (LLM)-based*. Rule-based approaches use

*Equal contribution.
†Work done at Amazon.

Authors' addresses: Aidan Z.H. Yang, aidan@cmu.edu, Carnegie Mellon University, USA; Yoshiki Takashima, ytakashi@andrew.cmu.edu, Carnegie Mellon University, USA; Brandon Paulsen, bpaulse@amazon.com, Amazon Web Services, USA; Josiah Dodds, jldodds@amazon.com, Amazon Web Services, USA; Daniel Kroening, dkr@amazon.co.uk, Amazon Web Services, USA.

Fig. 1. Examples of transpilations from a source Go program using compile-then-lift, Anthropic's Claude LLM, and *VERT*.

hand-written rules and algorithms that translate a target program into a new language, typically in a statement-by-statement fashion. Rule-based approaches have human-understandable implementations that could, in theory, be proved correct. However, as we will show, they often result in unreadable and unidiomatic code that does not take full advantage of the target language's useful features, such as efficient native types.

On the other hand, LLM-based approaches train an LLM that takes a program in one language as input and attempt to output an equivalent program in the target language [43]. 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, the model will usually produce high quality, idiomatic transpilations [57]. However, these approaches come with no formal guarantees that the resulting code will maintain input-output equivalence with the original [34, 38, 55].

In this work, we focus on *general*, *verified*, and *readable* transpilation to Rust. By *general*, we mean that it can apply to most major languages. By *verified*, we mean that input-output equivalence of the final transpilation has been verified against the source program in some way. The verification techniques we experiment with are property-based testing (PBT), bounded model checking, and unbounded model checking. By *readable*, we mean that the transpilation resembles human-written code. While readability is a highly subjective measure, we show examples in Figure 1 that we believe makes this claim self-evident.

To the best of our knowledge, the only *general* rule-based approach for transpiling to Rust that could *theoretically* guarantee equivalence is to *compile-then-lift*. In this approach, we first compile the source program to an intermediate language like LLVM or Web Assembly, and then lift

the intermediate language to the target language (Rust in our case). For example, Web Assembly compilers exist for most major languages (e.g. C, C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, and Kotlin), and the recent work `rWasm` [11] can lift Web Assembly to Rust. While this approach is very general and, at least in theory, can guarantee equivalence, compile-then-lift approaches generally can only produce code that is as readable as the intermediate language itself, which for LLVM and Web Assembly is virtually unreadable [24, 41]. We give an example transpilation using a Web Assembly compiler and `rWasm` in Figure 1. As can been seen, `rWasm` produces Rust that looks like assembly rather than a high-level language.

There are also many works on transpiling without equivalence guarantees [35, 43, 45, 52, 53, 56], mainly using language models. While these approaches are also general and usually produce readable code, language models are notorious for outputting subtly incorrect code [55]. We show an example language model transpilation in Figure 1 using Anthropic's Claude-2, which is a state-of-the-art general purpose LLM. We can see the transpilation is far more readable than the result of the compile-then-lift approach. However, the LLM has changed a - to a +, which may escape human review. Such subtle errors may be difficult to debug and only manifest in corner cases.

To overcome these limitations, 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 Web Assembly compiler.

VERT first creates an *oracle* Rust transpilation by using the source language's Web Assembly compiler and `rWasm`, as previously described. This transpilation is equivalent by construction, but is unreadable. Next, we leverage an LLM to produce a candidate final transpilation, which is far more readable, 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 hand-crafted 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 [45, 61]. We focus on C++ and C since these two languages are often used for similar tasks as Rust [27, 51]. We further evaluate on Go as Rust is often the transpilation target when cross-platform support is a hard requirement [26]. We experiment with three state-of-the-art LLMs as the underlying LLM for *VERT*, namely CodeLlama-2 [42], StarCoder [31], and Anthropic Claude-2 [4]. With Claude-2 as the underlying LLM, our results show that *VERT* can produce transpilations that pass bounded verification for 42% of these programs, and differential testing for 54% of these programs. Moreover, *VERT* improves the capabilities of existing LLMs – *VERT* is able to produce a verified transpilation 45% more often on average (43% for C++, 46% for C, and 43% for Go) compared to using an LLM alone.

To measure *VERT*'s ability to produce non-trivial *safe* Rust, we gather and manually label an additional 14 programs that make significant use of pointers from the benchmarks of prior work on C to Rust transpilation [2, 21, 61]. Our results on these additional benchmarks show that *VERT* can produce Rust with relatively simple ownership models for small (less than 36 LoC) programs.

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 human-readable Rust. Our data and tool are available for 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.

2 BACKGROUND

We give a brief introduction of the key aspects of our tool. In particular: Rust, the `rWasm` compilation strategy, and auto-regressive large language models.

2.1 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. In particular, when we want to create a variable that aliases a value (i.e., creating a new pointer to the value), we must transfer ownership to the new variable, either temporarily or permanently. Rust programs are challenging to write and synthesize as these ownership rules must be obeyed for the program to even compile. LLM-based Rust synthesis has the additional challenge that these typing rules are not found in popular languages that make up majority of the training dataset.

2.2 Migrating to Rust

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 [60], 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, 21, 61]. 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 scale to multiple 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.

2.3 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) [22] and Bounded

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

Model Checking [17, 18]. An example of a `bolero` harness for checking equivalence between an LLM-generated and a trusted reference program is given in Fig. 2.

```
#[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. 2. An example bolero harness checking equivalence between 2 functions.

Using a harness like the one in Fig. 2, `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.

`bolero` performs model checking through Kani [48], 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 loop iteration beyond the bound is not reachable. We say that a verification is *bounded* if unwind checks are turned off and exhaustiveness is not known. Conversely, a *full* verification includes unwind checks and thus is exhaustive with respect to all reachable executions. Even with full verification, the harness may not be able to cover enough values of unbounded types like vectors.

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 use both tools, using `bolero` and marking the harness for both PBT (`#[test]`) and model-checking (`kani::proof`).

2.4 Large Language Models

Deep learning (DL) has recently shown promise for program generation [36, 47]. The DL models that can achieve the closest capabilities to human-written results are large language models (LLMs), such as GPT-4 [37]. LLMs train billions of parameters using a massive amount of training data. LLMs' effectiveness for code generation [13] suggests that LLMs are capable of performing specialized software engineering tasks, such as program language transpilation.

Most modern LLMs are attention-based models. Attention-based models use the Transformer architecture [49]. In a Transformer architecture model, tokens exchange information across all other tokens using an attention matrix. LLMs typically produce text in a left-to-right manner (auto-regressive model), producing each token given its prefix context. In a auto-regressive models, the learned attention matrix is partially masked out. Specifically, the generation of a new token depends only on its prefix context (i.e., tokens on the left), and its suffix context (i.e., tokens on the right) are hidden (i.e., masked out). After a model finishes training all trainable parameters in a

user specified time, a model can make predictions on future tokens on a sequence of tokens the model has never seen before.

3 METHODOLOGY

In this section, we describe the key ideas behind our universal transpilation technique. Figure 3 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 3, 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 test 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.

3.1 Program repair on LLM output

LLMs often produce incorrect 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 create an automatic repair system based on **rustc** error messages. For each error, we first classify the error into one of three main categories: syntax, typing, and domain specific.

As seen in Figure 4, an example syntax error generated by the LLM is the wrong closing delimiter. For **rustc** to successfully compile, all syntax errors must be resolved. We track the error code location (e.g., line 10 in Figure 4), and we use the **rustc** provided initial delimiter to guide our repair strategy. For this case, we know to use the right curly bracket `}` to replace `]` on line 10.

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`.

Finally, domain-specific errors are compilation errors that are specific to the program. The error messages for domain-specific errors do not share the same structure, and therefore we only 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. It is often the case that even with the error message suggestion line, we cannot generate a repair that fixes all errors. In these cases, we regenerate an LLM output using the error as part of the new prompt and restart the process. Our error-guided repair is significantly faster than the LLM generation (discussed in Section 4.2), so we only regenerate an LLM output after exhausting all **rustc** helper messages.

3.2 Transpilation Oracle Generation

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. 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

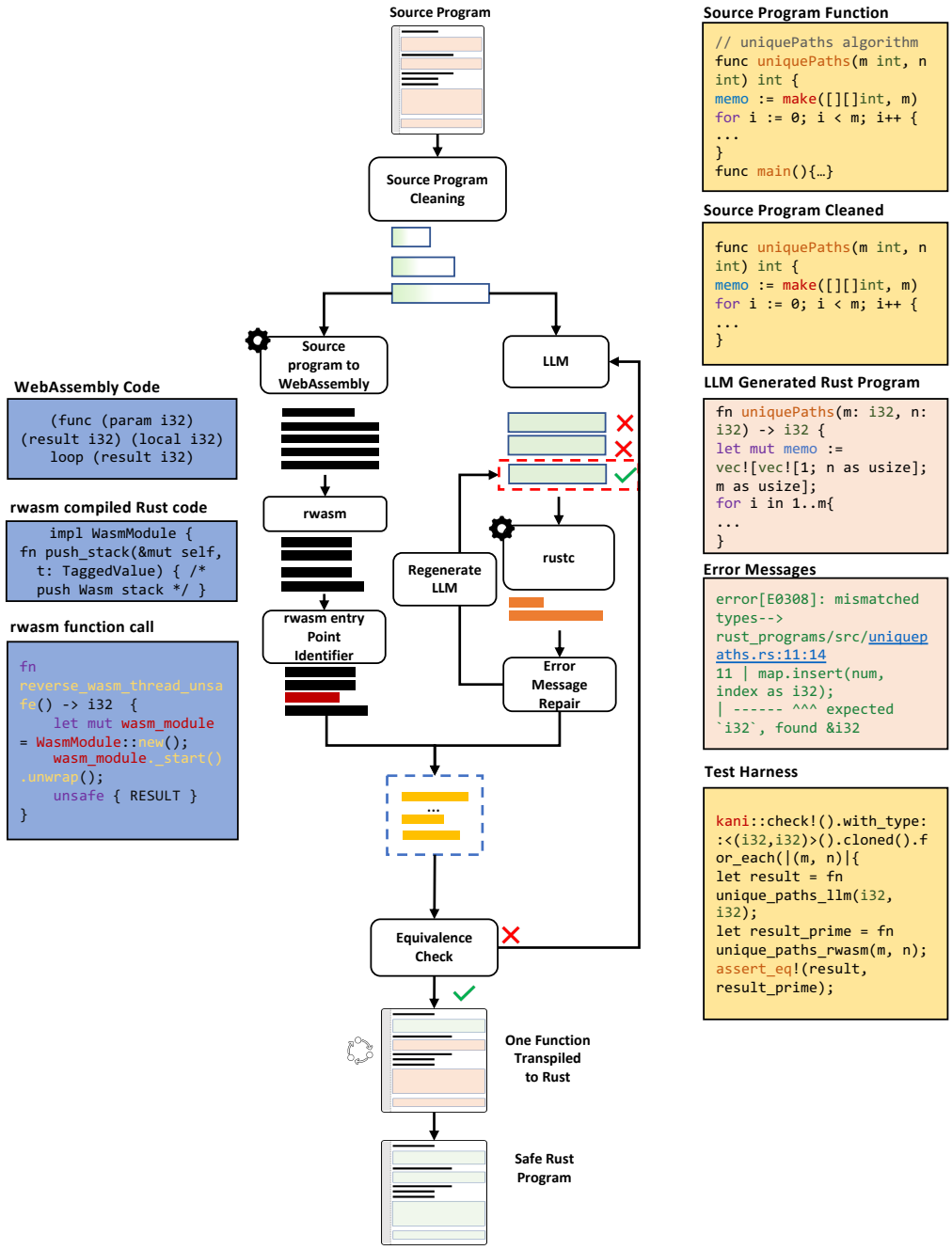

Fig. 3. *VERT* 's architecture, which takes as input a source program and produces a formally equivalent Rust program

```
1
2 For more information about this error, try 'rustc --explain E0433'.
3 error: mismatched closing delimiter: ']'
4   --> roman_to_integer_test.rs:1:33
5    |
6 1  | fn roman_to_int(s: &str) -> i32 {
7    |                                 ^ unclosed delimiter
8 ...
9 10 |     ]);
10   |     ^ mismatched closing delimiter
```

Fig. 4. Syntax error

```
1 error[E0308]: mismatched types-->
2 rust_programs/src/uniquepaths.rs:11:14
3 11 | map.insert(num,index as i32);| ------ ^^^ expected '&i32', found i32
4                                               help: consider borrowing here:
                                                    '&num'
```

Fig. 5. Mismatched type error

```
1 error[E0384]: cannot assign to immutable argument 'x'
2   --> reverse_integer_test.rs:16:3
3    |
4 1  | fn reverse(x: i32) -> i32 {
5    |- help: consider making this binding mutable: 'mut x'
6 ...
7 16 |         x = x / 10;
8    |         ^^^^^^^^^^ cannot assign to immutable argument
9
10 error: aborting due to 2 previous errors
11 For more information about this error, try 'rustc --explain E0384'.
```

Fig. 6. Immutable assignment error

```
1 func callReverse() int {
2   result := reverse(123)
3
4   if result == 321 {
5     return 0
6   } else {
7     return 1
8   }
9 }
```

Fig. 7. An entry point for the `reverse` function

use `rWasm` [11], 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

```
fn func_4(&mut self, ) -> Option<i32> {
  // ...
  let mut local_3 : i32 = 0i32;
  let mut local_4 : i32 = 0i32;
  v0 = TaggedVal::from(321i32);
  // mutant: v0 = TaggedVal::from(654i32);
  local_3 = v0.try_as_i32()?;
  v0 = TaggedVal::from(123i32);
  // mutant: v0 = TaggedVal::from(456i32);
  local_4 = v0.try_as_i32()?;
  // ...
}
```

Fig. 8. The difference between the original `rWasm` output and the mutated one (highlighted).

memory safety properties of safe Rust, we use it to generate trusted Rust code with same semantics as the original input.

3.3 Mutation Guided Entry Point Identification

Given the assembly-like output of `rWasm`, we must perform analysis to identify the entry point of the `rWasm` transpiled function. *VERT* provides the option for the user to manually identify the entrypoint, but we can find it automatically using a simple heuristic, such as a function call or a single test case. We note that this heuristic could be generated automatically using LLMs or search-based software testing and thus we can assume an entrypoint generator in the source language. *VERT* uses a function call in the source language with constant inputs to the function to be transpiled and an assertion on the output of that function. One such function is given in Fig. 7.

We leverage this function call to identify the input and output of the function. While one option for such analysis is to perform decompilation, we find that a mutation-guided approach is sufficient for our purposes. In Fig. 7, we know that the input is `123` and the output is `321`. Now, we wish to identify the equivalent constant in the `rWasm` output. While it is possible to just perform a linear scan of the `rWasm` output for this constant, that risks spurious matches, especially for simple types like `i32`. Instead, we guide this identification by leveraging the function call and mutating it. Suppose we swap `123` with `456` and `321` with `654` and re-transpile with `rWasm`. These constants will change, but the rest of the `rWasm` output remains the same. Taking the diff, we can identify inputs and outputs by what changed. The diff in the `rWasm` output is shown in Fig. 8.

3.4 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 property-based testing with random inputs and model-checking with symbolic 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 checking. Since the original program runs with a set of constant inputs, we must replace these constant inputs with the inputs of the harness like `input: i32`. The challenge here is to inject the inputs into the middle of the

Rust code representing a Wasm module. Instead of replacing the parameters to the function, we use globals in Rust to inject the inputs right at the location where constants used to be. An example is given in Fig. 9, with `func_4` being the Wasm equivalent of the test. *VERT* replaces constant inputs with global reads, generalizing the test and allowing us to vary the inputs fed into the Wasm-generated function. 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.

Now that we can feed various inputs into the Wasm function, we must also provide a way to assert that the output is equal. Recall that in Fig. 9, the output is compared to a baseline and 0 is returned if the check succeeds. Because of this comparison check, it is sufficient to inject the baseline value and then leverage the check to assert that the return value is 0. Injection is done in the same way as the inputs.

We note that while this approach is sound, it may falsely identify some equivalent programs as faulty due to semantic differences between Rust and the target language, or between the target language and `rWasm` embedding. We note two cases where we permit the analyst to add assumptions. First, when the input type is an unsigned integer. In this case, we have a mismatch were Wasm has only signed integers. So the output of `rWasm` will represent unsigned integers by encoding it in signed. However, the true valid range will be smaller (`u32` will become `i64` to the full range of `u32` values but the extra bits are not used). In this case, it is soundly permissible to assume that the values lie in the valid range of `u32`. Another case of valid assumptions occurs with strings: strings in C are ASCII while in Rust are Unicode. Therefore, a valid range of Rust strings will crash a C-derived Wasm module spuriously. We assume the string's range to valid ASCII only.

3.5 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 3 equivalence checking techniques with increasing levels of confidence and compute cost. This procedure is shown in Fig. 11. First, 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.

If the PBT stage succeeds, we now perform bounded verification with Kani. In the bounded verification phase, we run Kani with an unwinding bound of k and *no unwind checks*. This means that paths up to k loop iterations is exhaustively explored, but any divergences between the candidate and the oracle with traces containing more than k loop iterations are missed. We run this phase for 120 seconds, and terminate with 3 potential results. First, Kani returns with a counterexample that causes the oracle and the candidate to diverge or one of the two to crash. Second, Kani does not return with an answer within the time limit, which we also consider to be a failure as we cannot establish bounded equivalence. Finally, Kani verifies that, limited to executions with at most k loop iteration, there are no divergences or crashes. We consider the third case alone to be successful. Bounded verification does not check whether k is exhaustive.

If bounded verification succeeds, we perform full verification. *VERT* increases the unwinding bound until verification can be achieved with all checks enabled, including unwind checks that ensure the unwinding is fully exhaustive and the program cannot go beyond the code that was unwound. This ensures the equivalence and safety properties holds for every input the harness generates. Once again, three outcomes are possible. First, Kani can return a counterexample. Second, Kani can fail to return an answer within the time limit. Finally, Kani successfully verifies the harness. Again, we consider the third case alone to be successful. However, unlike with bounded verification,

```
static mut INPUT_1 = 0;
static mut OUTPUT_1 = 0;
impl WasmModule {
  /// returns 0 if the output matches
  fn func_4(&mut self, ) -> Option<i32> {
    // ...
    let mut local_3 : i32 = 0i32;
    let mut local_4 : i32 = 0i32;
    v0 = TaggedVal::from(unsafe {INPUT_1});
    local_3 = v0.try_as_i32()?;
    v0 = TaggedVal::from(unsafe {OUTPUT_1});
    local_4 = v0.try_as_i32()?;
    // ...
  }
}
/// equivalence-checking harness.
fn equvalence() {
  bolero::check!()
  .for_each(|(input: i32)| {
    let llm_fn_output = llm_generated_reverse();
    unsafe {
      INPUT_1 = input;
      OUTPUT_1 =  llm_fn_output;
    }
    let mut wasm_module = WasmModule::new();
    wasm_module._start().unwrap();
    assert!(wasm_module.func_4().unwrap() == 0);
  });
}
```

Fig. 9. Equivalence-checking harness for `func_4`.

successful full verification guarantees that the translation is without any error. If the oracle crashes at any point in the equivalence checking, *VERT* provides the user with a counterexample which can be used to diagnose the crash in the original program.

We support complex types through their primitive parts. Given a struct or enum, that Kani or PBT does not initially support, 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. However, 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 by having 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 bigger vector than encountered will be missed.

3.6 Few-shot Learning

The main focus of this work is on verifying the output of LLMs for program transpilation, and not LLM prompt engineering. Therefore, we keep the prompts simple and short. Complicated

```
{Original code}
...
Safe Rust refactoring of above code in {language}, with code only, no comments.
Use the same function name, same argument and return types.
Make sure the output program can compile on its own.
// If there exists counter examples from prior failed equivalence checking
Test that outputs from inputs {counter_examples} are equivalent to source program.
```

Fig. 10. LLM Prompt template.

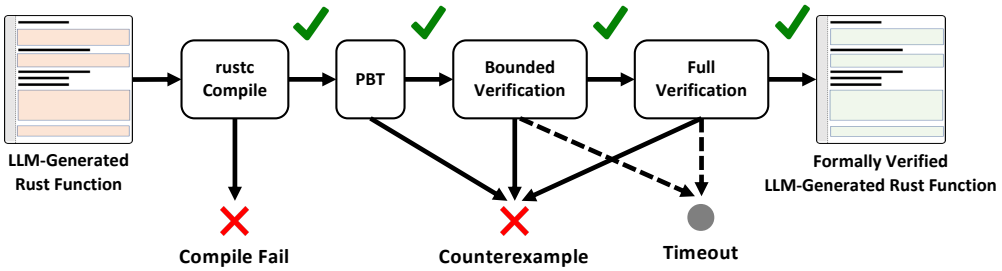

Fig. 11. Evaluation procedure.

and repeated querying of the same prompts do not provide additional benefits on the accuracy of outputs for small sized models, and too expensive for an average practitioner for industry sized models (i.e., Anthropic Claude). To achieve few-shot learning on our transpilation queries, each failed transpilation attempt provides its equivalence checking counter examples as a few-shot learning example for future transpilation attempts.

Figure. 10 shows our template for few shot learning. We start with querying the LLM to refactor the source code into safe Rust. Although we filter for safe Rust LLM output, we experimentally found that asking the LLM to always produce safe Rust gives more accurate results. We prompt the LLM to use the same argument and return types as the original, and can compile without external dependencies. Finally, we collect the counter examples from prior failed equivalence 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.

4 EVALUATION

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

RQ1. How does *VERT* perform vs. using the respective LLM by itself? 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.

RQ2. How does each component of *VERT*'s approach impact its performance? We conduct an ablation analysis, which shows that our prompting and error-guided refinement helps produce more well-typed and more correct programs. We further measure the runtime performance of each part of *VERT*, showing that time costs of error-guided refinement is reasonable and *VERT* spends most of the time in verification.

RQ3. Does *VERT* produce safe, readable, 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 lines of code between the translations produced by *VERT*, `rWasm`, and CROWN [61], 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.
RQ4. How extensible is *VERT* to future verifiers? While we use Kani [48] for most of the benchmarks to leverage automation, we encountered a large number of timeouts. We show that *VERT* is able to work with multiple verifiers by using Verus [30] and show that manual verification is possible albeit costly.

4.1 Setup

4.1.1 LLMs. We use the following LLMs to generate the candidate transpilations in *VERT*:

- **TransCoder-IR [45]**: 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 [43] 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 [42]**: A 13B parameter model initialized from Llama-2 [46], then further fine-tuned on 500 billion tokens of code data.
- **StarCoder Fine-tuned [31]**: A 15.5B parameter model trained on 1 trillion tokens sourced from The Stack [28]. StarCoder prompted achieves the highest HumanEval [13] score of 40.8 over comparable open-source LLMs, such as LLaMA-65B [46] with a score 23.7 and CodeGen-16B [36] with a score of 29.3. To investigate the effectiveness of Rust fine-tuning on prior LLMs, we fine-tune StarCoder for transpilation using LeetCode problems that have solutions for Rust, C, C++, and Go. In total, we collect solutions in each language for 94 LeetCode problems. We fine-tune the LLM to take C, C++, or Go solution as the input, and produce the corresponding Rust solution as output.
- **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.

4.1.2 LLM Fine-tuning. The availability of Rust code in open source is scarce as compared code written in most other programming languages. Incoder [23] estimates that Rust is only the 20th most common language in their training database, which is a 159GB code corpus taken from Github BigQuery [3]. Due to the lack of Rust data available on open-source, we opt to not train an LLM targeted at Rust code generation. Instead, we directly use an off-the-shelf industry grade LLM, and also fine-tune on a separate open-source pretrained LLM. Specifically, we use Anthropic Claude-2 [4] for the industry grade LLM, and StarCoder [31] for the pretrained LLM.

We use light weight and parameter efficient adapter layers [25, 33, 58] for fine-tuning StarCoder. Instead of retraining StarCoder entirely, we taken the final hidden states of StarCoder and add adapter layers at the end using small amounts of data. We collect 94 LeetCode type question solutions in C, C++, Go and Rust. Although there are existing code bases for all four languages, we

[2]https://console.cloud.google.com/marketplace/details/github/github-repos
[3]https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code
[4]https://www.anthropic.com/product

find that LeetCode has the most consistent translation between other languages and Rust. We were able to collect 94 LeetCode questions of which have a direct translation between all 3 languages. For each LeetCode type question, we have a corresponding source program (written in Go, C, or C++), and a target program (written in Rust). We encode all code words into tokens using the GPT-2 tokenizer. We fine-tune with 4 Transformer layers, 300 total epochs, and a final model dimension of 6144.

4.1.3 Benchmark selection. We draw our benchmarks from two sources. Our first source is the benchmark set from TransCoder-IR [45], 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) [9] 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, 31, 42]. 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 14 additional pointer-manipulating C programs from prior work on C to Rust transpilation [2, 21, 61]. We note, however, that 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, we select and customize snippets from these C projects to transpile to Rust. 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 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 4.2.

In total, we evaluate our approach on **569 C++** programs, **520 C** programs, and **341 Go** programs.

4.1.4 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, in which a higher score indicates the two outputs have many tokens in common. 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.

For programming languages, metrics based off of passing tests have been proposed. Roziere et al. [43] and Szafraniec et al. [45] use the computational accuracy (CA) metric, which counts a translation as correct if it passes a series of unit tests. However, there is no accepted standard for the number of required passing tests when using the CA metric. Furthermore, the CA metric does not take into account the quality or coverage 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. 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 9. Our full procedure is shown in Fig. 11. Since the three metrics used are significantly slower than checking a series of unit tests, we set a time limit for our metrics. For all three metrics, we set a 120

seconds limit. For PBT, no counterexamples within 120 seconds counts as success. For Bounded and Full verification, success requires establishing verified equivalence within 120 seconds. If any of the three verification step fails, *VERT* terminates.

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

4.2 Results

We present results on the TransCoder-IR benchmarks in Table 1. We present *VERT* operating in three different modes. *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 repair* 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 3.1 to the output of the LLM. Finally, *Few-shot repair & 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 3.6. *Few-shot repair & 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. As seen in Table 1, we only perform *single-shot* for Transcoder-IR (baseline) to replicate results from prior work. We perform *few-shot repair* on CodeLlama2 and StarCoder fine-tuned to investigate the effectiveness of few-shot and rule-based repair on open-source, non-instruction tuned LLMs. Finally, we perform *single-shot, few-shot repair, and few-shot repair 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.

RQ1. How does *VERT* perform vs. using the respective LLM by itself?

As seen in table 1, *VERT* with Claude-2 compiles for 76% more programs for C++, 75% for C, and 82% for Go as compared to baseline (i.e., Transcoder-IR). *VERT* with Claude-2 can pass PBT for 49% more programs for C++, 37% for C, and 56% for Go as compared to baseline. *VERT* with Claude-2 can pass bounded verification for 40% more programs for C++, 37% for C, and 47% for Go as compared to baseline. For passing full verification, *VERT* with Claude-2 can transpile 19 C++ programs, 15 C programs, and 9 Go programs. Transcoder-IR cannot pass full verification on any of the tested programs. *VERT* with both CodeLlama2 and StarCoder fine-tuned also improve upon baseline on number of programs passing compilation, PBT, bounded verification, and full verification. 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 [54], which indicates if 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, bounded-verification, and full-verification.

RQ1 Summary

VERT with CodeLlama2, StarCoder fine-tuned, and Anthropic Claude-2 can produce more PBT, bounded verification, and full verification passing Rust transpilations than baseline. In particular, *VERT* with Claude-2 can pass bounded verification for 40% more programs for C++, 37% for C, and 47% for Go as compared to baseline.

Table 1. *VERT* performance across with different LLMs and modes.

LLM	Source Lang	Technique	Compiled	PBT	Bounded-ver.	Full-ver.
Transcoder IR (Baseline)	C++ (569)	Single-shot	107	23	3	0
	C (520)	Single-shot	101	14	1	0
	Go (341)	Single-shot	24	3	0	0
CodeLlama2 13B	C++ (569)	Few-shot repair	307	25	6	2
	C (520)	Few-shot repair	160	18	4	2
	Go (341)	Few-shot repair	104	15	2	0
StarCoder fine-tuned 15.5B	C++ (569)	Few-shot repair	253	79	8	2
	C (520)	Few-shot repair	179	76	4	2
	Go (341)	Few-shot repair	134	59	2	0
Anthropic Claude-2 130B	C++ (569)	Single-shot	240	55	6	0
		Few-shot repair	539	292	41	2
		Few-shot repair & counter examples (VERT)	**539**	**295**	**233**	**19**
	C (520)	Single-shot	239	49	6	0
		Few-shot repair	339	195	29	4
		Few-shot repair & counter examples (VERT)	**339**	**209**	**193**	**15**
	Go (341)	Single-shot	126	26	3	0
		Few-shot repair	276	157	39	4
		Few-shot repair & counter examples (VERT)	**317**	**195**	**159**	**9**

RQ2. How does each component of *VERT* impact its performance? Table 1 shows the transpilation results across CodeLlama-2 and StarCoder fine-tuned in a few-shot setting. We observe that *VERT* with CodeLlama-2 and StarCoder fine-tuned improve over Transcoder slightly for compilable Rust translations. Since Rust is an underrepresented language in all LLMs trained on

Table 2. *VERT*'s average runtime per component for a Single-program translation

Component type	Component	Time (s)
LLM	Transcoder-IR	8
	CodeLlama-2	43
	Starcoder fine-tuned	45
	Anthropic Claude	30
Rust compilation	`rustc`	<1
	Error guided repair	1
	`rwasm`	<1
Testing and verification	PBT	25
	Bounded-ver.	52
	Full-ver.	**67**

GitHub open-source repositories and The Stack dataset [28], we see that light-weight fine-tuning on a small dataset shows immediate improvement. In particular, we observe that StarCoder fine-tuned has fewer transpilations than CodeLlama-2 passing compilation, but more transpilations than CodeLlama-2 passing bounded verification. Fine-tuning with Rust code has an immediate impact on transpilation accuracy. 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 fine-tuned compiles 47% fewer programs for C++, 41% fewer for C, and 63% fewer programs for Go as compared to *VERT* with Claude-2. While adding fine-tuning on Rust syntax increases the number of compilable translation generated, we observe that an industry-grade LLM with more trainable parameters and a larger training dataset performs significantly better for our metrics.

We observe that *VERT* using few-shot plus repair with either StarCoder fine-tuned or Claude-2 yields better transpilation across all our three languages and three metrics. In particular, few-shot plus repair 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. Table 1 does not show single-shot results for CodeLlama-2 and StarCoder fine-tuned as we observed no transpilations passing PBT. Few-shot plus repair with Claude-2 passes 6% more bounded-verification 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. For C++ and C in particular, few-shot and repair with Claude-2 does not provide any additional passes on bounded verification nor full verification as compared to only few-shot with Claude-2. We observe that few-shot learning with counter examples of failed previous verification attempts provides the largest improvements on both bounded-verification and full-verification. Modern LLMs that are instruction-tuned can learn to generate more correct program when given specific test failures in few-shot settings.

Table 2 shows the average runtime of each of *VERT*'s components across our entire evaluation dataset. We observe that in the non-timeout failure cases (i.e., Kani does not establish equivalence within 120s), Kani's full verification (full-ver.) uses an average of 67 seconds per program. Kani's bounded verification uses an average of 52 seconds per program, and Bolero's property testing uses an average of 25 seconds per program. 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.

RQ2 Summary

Our ablation study shows that fine-tuning an LLM with Rust yields a higher accuracy of transpiled programs, as seen by a higher number of programs passing PBT and bounded verification by StarCoder fine-tuned compared to CodeLlama2. However, few-shot learning with counter examples provides the largest improvements on transpilation accuracy. Finally, we observe that *VERT* spends most of its runtime in verification.

RQ3. Does *VERT* produce safe, readable, 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 C programs described previously. Table 3 presents the results of *VERT* on these 14 benchmarks as well as several metrics that provide a rough idea of the complexity of the benchmarks. Specifically, we present the number of pointer variables, function definitions, LoC, and the number of structs defined in each benchmark. The `avl_*` benchmarks are taken from a library that implements an AVL tree. The `brotli_*` benchmarks 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.

VERT can produce transpilations for 7 of the 14 C programs that pass PBT, and 2 of those can pass bounded verification. 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, have more pointer variables, and also on programs with multiple functions. Still on smaller programs, the LLM can still determine basic ownership needs. For example, it can determine if a variable or parameter reference needs a `mut` permission. On the `avl_insert` benchmark, the LLM successfully assigns ownership to the newly created node. To evaluate the readability, we compare lines of code in the transpilations produced by *VERT*, `rWasm`, and CROWN [61], the rule-based C to Rust transpiler. After running `rustfmt` [44], the official formatter of Rust, CROWN's output is more than 5x larger than *VERT*, and `rWasm`'s output more than 10x as large. Given the strong negative association between LoC and code readability [12], we conclude that *VERT*'s outputs are more readable than CROWN and `rWasm`.

To evaluate the idiomaticity, we run Clippy[5], the official linter of Rust, on *VERT*'s transpilations. Clippy checks Rust code for potential issues in the categories of correctness (e.g. checking if an unsigned int is greater than 0), performance (e.g. unnecessarily using a Box or collection type), stylistic (e.g. not following naming conventions, unnecessary borrowing), and conciseness (e.g. unnecessary type casting or parentheses). On average, Clippy produces 10.9 warnings per function for CROWN, and 372 warnings per function for `rWasm`. Clippy does not produce any warnings on *VERT*'s transpilations, thus we conclude that they are reasonably idiomatic.

VERT targets the broader and more difficult problem of general, verified translation to Rust, whereas CROWN only targets unsafe to safe rust (after running C2Rust [2]) without verification. For the 14 programs, both *VERT* and CROWN output safe Rust. However, *VERT* 's output is more Rust-native than CROWN's, using standard Rust types while CROWN and C2Rust use C-foreign

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

Table 3. Benchmark information and results on the 14 C pointer manipulation programs. The symbols ✔, ✗, and ● indicate pass, fail (with counterexample), and timeout.

Benchmark	Structs	Functions	Pointer variables	LOC	Compiled	PBT	Bounded Ver.
avl_minvalue	1	1	4	17	✔	✔	✔
avl_insert	1	2	4	30	✔	✔	●
avl_rotate	1	3	7	32	✔	✔	●
avl_delete	1	4	27	111	✗	✗	✗
brotli_parse_int	0	1	2	15	✔	✔	●
brotli_filesize	1	1	1	28	✔	✗	✗
buffer_new	1	1	3	16	✔	✔	●
buffer_resize	1	3	3	22	✔	✗	✗
ht_hashkey	0	1	2	13	✔	✔	✔
ht_create	2	1	3	36	✗	✗	✗
libcsv_get_opts	1	1	1	29	✔	✗	✗
libcsv_init	1	1	4	55	✗	✗	✗
libtree	0	1	1	7	✔	✔	●
urlparser	0	9	28	158	✗	✗	✗

types/functions. *VERT* 's lack of reliance on C-foreign functions is a qualitative strength. *VERT* 's output is more self-contained and reviewable to Rust programmers [7]. *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. Finally, we note that CROWN assumes correctness on their output, and only runs deterministic test suites on 6 example benchmarks (corresponding to 4/14 of our selected pointer benchmarks). *VERT* performs three layers of equivalence checking on all its output.

RQ3 Summary

VERT can produce transpilations for 7 of the 14 C programs that pass PBT, and 2 of those can pass bounded verification. *VERT* tends to struggle as programs have more pointer variables, or have multiple functions. However, *VERT* is far more readable than prior work. *VERT* produces 5X less LoC than CROWN, 10x less LoC than `rWasm`, and its transpiled Rust programs do not show any linter warnings.

RQ4. How extensible is *VERT* to future verifiers? We observe in Table 1 that few transpiled Rust programs can pass full-verification with Kani, which is a bounded model checker (BMC). Full-verification using a BMC results in complete unrolling of a program, which does not scale to programs that loop over their inputs. We consider using Verus [30] as the verifier instead of Kani. Given that the verification failures are due to Kani unrolling loops, we use Verus to specify loop invariants and algebraic specifications for proving equivalence.

VERT handles multiple verifiers for the equivalence checking step. This is useful when different verifiers have different strengths and weaknesses. For example, Kani is a bounded model checker so loops are difficult to verify. Verus is a autoactive verifier [30], so it can verify loops more effectively via invariants, but at the cost of lower automation. To understand the need for an autoactive verifier, we run lightweight analysis using regex matching on our benchmarks that Kani failed to verify. 86% of the benchmarks had explicit iteration over an input. Furthermore, this is an undercount because it ignores loops that happen in API calls and library functions like `strcpy`.

Table 4. We manually verify 5 timeout cases in RQ4 using Verus. LoC is the lines of code while Spec. Loc is the lines of specification. No Spec. LoC is given when verification is not successful. The symbols ✔ and ● indicate pass and feature limitation respectively.

Benchmark	Code LoC	Spec. LoC	Verus Ver.
avl_insert	30	-	●
avl_rotate	32	-	●
brotli_parse_int	15	92	✔
buffer_new	16	1	✔
libtree	7	6	✔

Given the manual effort required to manually verify each of the LLM's outputs, we limit ourselves to the 5 cases in the CROWN benchmark where PBT succeeded but bounded Kani failed. The results are show in the Table 4. We succeed in verifying equivalence for 3 of the 5 failed benchmarks while we fail to verify AVL benchmarks due to Verus's limitations on returning mutable pointers. In each of the successful cases, checking equivalence required fully specifying and verifying the resulting programs. Given that, the spec to code ratio varied substantially depending on the benchmark. For `brotli_new` the program simply allocates a large buffer that caused Kani to time out. So the specification is a one line, and would have probably passed with Kani using the new contract feature or with a lower buffer size. `libtree` benchmark is a function that, given a list of integers, checks if it is sorted. This one required a quantifier-based specification to assert an invariant over the range of the array that has checked as sorted. While the spec size is not substantial, the presence of quantifiers over unbounded arrays will have been difficult to specify with Kani. The final and heaviest benchmark is `brotli_parseint`, which required 92 lines of specification and supporting proof. This function parses an integer from an array of chars, and specifying that required recursive specs involving exponentiation that would be difficult with Kani. The sheer size of the spec also stems from proofs required to show that the spec is well-behaved while recursing over the array. Overall ratio of spec to code is 2.6.

We go over `brotli_parseint` detail to describe the supporting specification proofs for equivalence. We give the specification used to prove `brotli_parseint` function correct in Fig. 12. The main specification is `right_parse`, which defines parsing of the array from right to left upto tthe given index. The induction proof `right_parse_continues` extends this specification over the array of `char`. We also prove the absence of arithmetic overflows in the output program by relating the maximum array length to the size of the parsed integer. We omit the precise specification for conciseness.

We give the LLM's output and supporting specifications in Fig. 13. We use the `ensures` to specify that a valid vector parses correctly and invalid or out-of-range values are not returned. The main loop verifies through the invariant that the integer parsed out is correct up to the current index of the array. For this to hold, we also trivially specify that the vector remains valid through the loop. Finally, we put an invariant that the integer parsed is below 10^i, which we use to show no integer overflow occurs. The rest of the requirements dispatches naturally through the conditional cases. We note that full specification, while possible, is not compatible with the automatic nature of the tool. We expect that further improvements, such as automatic invariant synthesis through LLMs [59] might make this approach more amenable to users who have neither the specification nor the formal methods expertise to use tools like Verus.

```
spec fn right_parse(s: &Vec<char>, upto: int, value: int) -> bool
    decreases upto,
{
    if 0 <= upto && upto <= s.len() {
        if upto > 0 {
            (value % 10 == (s[upto - 1] as u32 - '0' as u32)) &&
              right_parse(s, upto - 1, value / 10, )
        } else { value == 0 }
    } else {false}
}
proof fn right_parse_continues(s: &Vec<char>, upto: int, value: int)
    requires
        valid_vector(s),
        right_parse(s, upto, value),
        0 <= upto < s.len(),
    ensures
        right_parse(s, upto + 1, (value * 10) + ((s[upto] as u32) - ('0' as
            u32))),
{ /* Verus figures out by definition you
can append a digit by parsing one more char. */}
```

Fig. 12. Specification of `brotli_parseint`. We prove this spec on both the LLM and reference programs.

RQ3 Summary

We manually verify 5 programs with Verus that previously timed-out using Kani. We succeeded in verifying equivalence for 3 of the 5 failed benchmarks by leveraging loop invariants.

5 RELATED WORK

5.1 Language model transpilers

Recent advances in language modelling using code as training data have shown that LLMs can perform code completion [20] and generate code based on natural language [40] with impressive effectiveness. Large Language Models (LLMs) have raised performance on these tasks using significantly more trainable parameter and training data [13]. Transcoder and Transcoder-IR [43, 45] use unsupervised machine translation to train neural transcompilers. As both Transcoder versions are trained on program translation pairing, 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 [10, 13, 39]. Transcoder-IR [45] show in their evaluation that a significant portion of their translated code do not compile, especially for target programming languages that are underrepresented in their training data (e.g., Rust). Our work helps both generate memory-safe code and establish equivalence, to safely harness the code output of an LLM.

5.2 Rust transpilers

Prior Rust transpilers convert C/C++ to Rust. C2Rust [2] automatically converts large-scale C programs to Rust while preserving C semantics. Citrus [3] and Bindgen [1] both generate Rust FFI bindings from C libraries, and produce Rust code without preserving C semantics. Bosamiya et al. [11] embedded WebAssembly (Wasm) semantics in safe Rust code for the Rust compiler to emit safe and executable Rust code. Bosamiya et al. [11] implemented all stages of their tool in safe Rust,

```
fn llm_parseint(s: &Vec<char>, low: u32, high: u32) -> (ret: Option<u32>)
    ensures
        valid_vector(s) && ret.is_some() ==> right_parse(s, s.len() as int,
        ret.unwrap() as int),
        valid_vector(s) && ret.is_some() ==> low <= ret.unwrap() <= high,
        !valid_vector(s) ==> ret.is_none()
{
    let mut value = 0u32;
    let mut i: usize = 0;
    while i < 5
        invariant
            valid_vector(s) && value < exp(i as int) &&

            right_parse(s, i as int, value as int),
    {
        let mut c: char = s[i];
        if (c as u32) < ('0' as u32) || (c as u32) > ('9' as u32) {
            return None;
        }
        assert(i < 6);
        assert(value < exp(6)) by {
            exp_monotone_any(value, i as int, 6);
        };
        assert(exp(6) == 1000000) by (compute_only);
        value = 10 * value + (c as u32 - '0' as u32);
        i += 1;
    }
    ... // Return None for error
    return Some(value);
}
```

Fig. 13. Code of `brotli_parseint`. We prove the `right_parse` specification from Fig. 12 onto the function with invariants. Note that both the original code in C and the LLM output assumes input length of at most 5.

and no stage of compiler needs to be further verified or trusted to achieve safety. Our work focuses on generating readable and maintainable Rust code with Rust semantics directly.

Existing tools for making unsafe Rust safer focus on converting raw pointers to safe references. Emre et al. [21] localized unsafe Rust code from C2Rust and converted unsafe Rust to safe Rust by extracting type and borrow-checker results from the `rustc` compiler. Zhang et al. [61] converts unsafe Rust from C2Rust to safe Rust by computing an ownership scheme for a given Rust program, which decides which pointers in a program are owning or non-owning at particular locations. Zhang et al. [61] evaluates their tool CROWN on a benchmark suite of 20 programs, and achieve a median reduction rates for raw pointer uses of 62.1%. Ling et al. [32] removed non-essential "unsafe" keywords in Rust function signatures and refined the scopes within unsafe block scopes to safe functions using code structure transformation. Our work is the first to propose a general Rust transpiler that does not depend on the source language's memory management properties to produce safe Rust.

5.3 Equivalence Verification

Equivalence verification has been studied in settings where two copies of should-be-equivalent artifacts are available. Churchill et al. applied equivalence checking to certify compiler optimizations

with program alignment [14, 15]. Antonopoulos et al. leverage an extension to Kleene Algebra with Tests (KATs) for a more formal calculus [6]. Rule-based approaches have been combined with machine learning by Kommrusch for automated equivalence verification [29]. Conversely, Dahiya and Bansal leverage equivalence-checking in black-box settings without relying on the translator [19]. Our work uses equivalence verification for program language translation. Our work is the first to combine LLMs with established equivalence checking techniques (i.e., property based testing and bounded model checking) to generate both readable and verified code.

6 LIMITATIONS AND DISCUSSION

Threats to *internal validity* are concerned with the degree of confidence on our dependent variables and our results. As *VERT* uses the LLMs StarCoder and Claude, which take as training data GitHub repositories up to June 2022, we can not fully mitigate the bias introduced by potential *training data contamination* – i.e. the possibility that the competitive coding questions in our evaluation datasets could be included in their 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 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 [45]. 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 further evaluate on 14 selected functions from real world programs taken from the *Crown* and *Laertes* benchmark [21, 61].

VERT's equivalence checking is performed in 3 stages with increasing guarantees: PBT, bounded, and full verification. The first 2 phases do not always 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. Full verification provides exhaustive guarantees but few programs actually reach that stage due to performance limitations of Kani. As Kani improves, the performance of unbounded verification will improve as well.

7 CONCLUSION

Rust is a growing language with C-like performance, but provides further safety guarantees. 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, fail to follow the target language conventions, and do not scale for larger programs. On the other hand, ML-based approaches (i.e., LLMs) cannot provide formal guarantees, thus removing the security benefits of Rust. In this work, we show how to use both LLMs and formal verification to transpile verified and readable Rust programs. We evaluate our tool *VERT* by transpiling 1,394 programs from C++, C, and Go. Our tool with the Claude LLM can verify with bounded model checking for 40% more programs for C++, 37% for C, and 47% for Go as compared to baseline.

8 DATA AVAILABILITY

The software that supports the experiments, the benchmark programs, and the LLM Rust transpilations is available at https://zenodo.org/records/10927704. The TransCoder-IR model can be obtained from https://github.com/facebookresearch/CodeGen.

REFERENCES

[1] [n. d.]. bindgen. https://github.com/rust-lang/rust-bindgen.
[2] [n. d.]. C2Rust. https://c2rust.com/.
[3] [n. d.]. citrus. https://gitlab.com/citrus-rs/citrus.
[4] [n. d.]. claude. https://www.anthropic.com/index/introducing-claude.
[5] Stephen Akinyemi. 2023. Awesome WebAssembly Languages. https://github.com/appcypher/awesome-wasm-langs original-date: 2017-12-15T11:24:02Z.
[6] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 20:573–20:603. https://doi.org/10.1145/3571213
[7] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.
[8] Jeff Barr. [n. d.]. Firecracker – Lightweight Virtualization for Serverless Computing. *AWS News Blog* ([n. d.]). https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/
[9] Stella Biderman, USVSN PRASHANTH, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. 2024. Emergent and predictable memorization in large language models. *Advances in Neural Information Processing Systems* 36 (2024).
[10] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large scale autoregressive language modeling with Mesh-Tensorflow. *If you use this software, please cite it using these metadata* 58 (2021).
[11] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. *Provably-Safe* multilingual software sandboxing using *WebAssembly*. In *31st USENIX Security Symposium (USENIX Security 22)*. 1975–1992.
[12] Raymond P. L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng.* 36, 4 (jul 2010), 546–558. https://doi.org/10.1109/TSE.2009.70
[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[14] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, 1027–1040. https://doi.org/10.1145/3314221.3314596
[15] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. *ACM SIGARCH Computer Architecture News* 45, 1 (April 2017), 313–326. https://doi.org/10.1145/3093337.3037754
[16] Catalin Cimpanu. [n. d.]. Microsoft: 70 percent of all security bugs are memory safety issues. *ZDNET* ([n. d.]). https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues
[17] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19 (2001), 7–34.
[18] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS (LNCS, Vol. 2988)*. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15
[19] Manjeet Dahiya and Sorav Bansal. 2017. Black-Box Equivalence Checking Across Compiler Optimizations. In *Programming Languages and Systems (LNCS)*, Bor-Yuh Evan Chang (Ed.). Springer, 127–147. https://doi.org/10.1007/978-3-319-71237-6_7
[20] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. 345–356.
[21] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.
[22] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.
[23] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
[24] Peter Garba and Matteo Favaro. 2019. Saturn-software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*. 27–38.
[25] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. *arXiv preprint arXiv:2304.01933* (2023).
[26] Shashank Mohan Jain. 2023. Why Choose Rust. In *Linux Containers and Virtualization: Utilizing Rust for Linux Containers*. Springer, 145–180.

[27] Ralf Jung. 2020. Understanding and evolving the Rust programming language. (2020).
[28] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).
[29] Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. 2023. Self-supervised Learning to Prove Equivalence Between Straight-Line Programs via Rewrite Rules. *IEEE Transactions on Software Engineering* 49, 7 (July 2023), 3771–3792. https://doi.org/10.1109/TSE.2023.3271065
[30] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. https://doi.org/10.1145/3586037
[31] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[32] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. 2022. In Rust we trust: a transpiler from unsafe C to safer Rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 354–355.
[33] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.
[34] Marcus J Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. 2023. Beyond Accuracy: Evaluating Self-Consistency of Code LLMs. In *The Twelfth International Conference on Learning Representations*.
[35] Ansong Ni, Daniel Ramos, Aidan ZH Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. Soar: a synthesis approach for data science api refactoring. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 112–124.
[36] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[37] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[38] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[39] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An empirical cybersecurity evaluation of github copilot's code contributions. *ArXiv abs/2108.09293* (2021).
[40] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 419–428.
[41] Alan Romano and Weihang Wang. 2020. Wasim: Understanding webassembly applications through classification. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1321–1325.
[42] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[43] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020).
[44] Rust Contributors. [n. d.]. rustfmt. https://github.com/rust-lang/rustfmt
[45] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578* (2022).
[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[47] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI conference on human factors in computing systems extended abstracts*. 1–7.
[48] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 321–330.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[50] Steven Vaughan-Nichols. [n. d.]. Linus Torvalds: Rust will go into Linux 6.1. *ZDNET* ([n. d.]). https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

[51] Graf von Perponcher-Sedlnitzki and Philipp Christian. 2024. *Integrating the future into the past: Approach to seamlessly integrate newly-developed Rust-components into an existing C++-system.* Ph. D. Dissertation. Technische Hochschule Ingolstadt.

[52] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*. 402–412.

[53] Justin D Weisz, Michael Muller, Steven I Ross, Fernando Martinez, Stephanie Houde, Mayank Agarwal, Kartik Talamadupula, and John T Richards. 2022. Better together? An evaluation of AI-supported code translation. In *27th International Conference on Intelligent User Interfaces*. 369–391.

[54] Robert F Woolson. 2007. Wilcoxon signed-rank test. *Wiley Encyclopedia of Clinical Trials* (2007), 1–3.

[55] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).

[56] Aidan ZH Yang. 2020. 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*. 10–12.

[57] Aidan ZH Yang, Sophia Kolak, Vincent J Hellendoorn, Ruben Martins, and Claire Le Goues. 2024. Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models. *arXiv preprint arXiv:2404.15236* (2024).

[58] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[59] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. 2023. Leveraging Large Language Models for Automated Proof Synthesis in Rust. arXiv:2311.03739 [cs.FL]

[60] Handong Zhang. 2023. 2022 Review | The adoption of Rust in Business. https://rustmagazine.org/issue-1/2022-review-the-adoption-of-rust-in-business

[61] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership guided C to Rust translation. In *Computer Aided Verification (CAV) (LNCS, Vol. 13966)*. Springer, 459–482.