RUG: Turbo LLM for Rust Unit Test Generation

Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang†, and Taesoo Kim
Georgia Institute of Technology, †George Mason University
{cxworks, fsang, yzhai60, taesoo}@gatech.edu, xiaokuan@gmu.edu

***Abstract*—Unit testing improves software quality by evaluating isolated sections of the program. This approach alleviates the need for comprehensive program-wide testing and confines the potential error scope within the software. However, unit test development is time-consuming, requiring developers to create appropriate test contexts and determine input values to cover different code regions. This problem is particularly pronounced in Rust due to its intricate type system, making traditional unit test generation tools ineffective in Rust projects. Recently, large language models (LLMs) have demonstrated their proficiency in understanding programming language and completing software engineering tasks. However, merely prompting LLMs with a basic prompt like "generate unit test for the following source code" often results in code with compilation errors. In addition, LLM-generated unit tests often have limited test coverage.**

To bridge this gap and harness the capabilities of LLM, we design and implement RUG, an end-to-end solution to automatically generate the unit test for Rust projects. To help LLM's generated test pass Rust strict compilation checks, RUG designs a semantic-aware bottom-up approach to divide the context construction problem into dependent sub-problems. It solves these sub-problems sequentially using an LLM and merges them to a complete context. To increase test coverage, RUG integrates coverage-guided fuzzing with LLM to prepare fuzzing harnesses. Applying RUG on 17 real-world Rust programs (average 24,937 LoC), we show that RUG can achieve a high code coverage, up to 71.37%, closely comparable to human effort (73.18%). We submitted 113 unit tests generated by RUG covering the new code: 53 of them have been accepted, 17 rejected, and 43 are pending for review.

***Index Terms*—Unit testing, Large language model, Rust**

I. INTRODUCTION

Unit testing is essential to ensure program quality throughout the development process, with the aim of comprehensively testing a function in all possible branches and execution paths. It has become an integral part of the software development cycle, and many companies such as Google and Meta have a strict coverage requirement [1], [2]. A good unit test includes the calling context to successfully execute the function, the proper input triggering different paths, and clear assertions reflecting the correctness of the result. Consequently, it is widely recognized that the production of high-quality unit tests requires substantial human energy and effort. For example, in the United States, software testing labor is estimated to cost $48 billion dollars per year [3].

To reduce developers' workload, several approaches have been proposed to automate test generation, including 1) traditional approaches like Search Based Software Testing (SBST) [4], fuzzing [5], program synthesis [6] and 2) latest Large Language Model (LLM)-based methods [7], [8]. Traditional approaches usually leverage program analysis to build the testing context and search for appropriate test input to trigger different paths. These tools achieve good testing coverage on programming languages such as Java (*e.g.,* EvoSuite [9]), Python (*e.g.,* Pynguin [10]) and C/C++ (*e.g.,* AFL [5]). Recently, many LLM-based tools such as CODAMOSA [11], ChatUniTest [8], TestGen-LLM [12] have been proposed to automate testing generation. Commercial products like Cody [13] and Copilot [14] are available for daily development.

Rust, an emerging system programming language that performs strict compilation checks, is gaining traction for its performance and memory safety advancements. It is adopted in crucial projects such as operating system kernels [15], [16], device drivers [17], web browsers [18], etc. Rust also has a plethora of over 130K packages to date. However, Rust packages are not well tested. We conducted a survey on crates.io, Rust's package repository, computing the unit tests' code coverage for the top 30K most downloaded crates[1]. From our study, we found that many crates are rarely tested, and 47.41% of the crates have less than 30% test coverage, which underscores the serious problem *lack of unit testing* in Rust projects.

The difficulty of Rust testing is due to its complex-type systems. For example, Rust's ownership system defines the single owner of each memory value, and the borrow checker ensures that each value should only have one writable reference at a time. These language features impose extreme stringency in compiler checks, making it challenging even for experienced developers to craft valid code that passes compilation [19]. When applying existing approaches in Rust, traditional approaches such as SBST and fuzzing suffer from complex type dependencies and the potential huge searching space, leading to limited test cases and test coverage. For LLM based approaches, Rust's strict compiler checks and complex program conditions largely undermine the ability of LLMs to generate valid testing code with high code coverage. This underscores the necessity of developing an automated tool specifically for the Rust programming language and pinpoints two key challenges that must be addressed: **A) Passing compilation checks.** and **B) Expanding code coverage.**

In this paper, we propose RUG, which leverages LLM to automatically generate compilable high coverage unit tests for Rust projects. To solve the challenge **A**, RUG proposes a bottom-up approach to divide the context construction problem into dependent sub-problems and iteratively interact with LLM

[1]In Rust, packages are called crates.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00097

to solve each sub-problem. Each subsolution will be verified and memorized from the bottom of the dependency graph to the top. The sub-solutions will be merged at the end to generate the final test context. For challenge **B**, RUG transforms the generated tests into fuzzing harnesses without breaking test body and leverages fuzzing to improve the test coverage. Regarding the fuzzing corpus, RUG prompts LLM to prepare sample test data during context generation and reuses them as initial corpora for the fuzzing process.

We evaluated RUG on the 17 most frequently downloaded crates of Rust (average 24,937 LoC). The result shows that RUG generates high-quality tests with better code coverage. Using the latest `GPT-4` model, RUG achieves 71.37% code coverage, which is comparable to human practice (73.18%), and even achieves higher test coverage than human practice on 8 crates. We submitted 113 unit tests generated by RUG containing code regions that developers failed to test as pull requests (PRs), and 53 of them have been merged into the project, taking 75. 14% of all the tests reviewed.

Contributions. This paper makes following contributions[2]:

- **A Bottom-up Context Building Algorithm.** We propose a semantic-aware bottom-up algorithm which simplifies the context construction problem for LLM and improve correctness of generated code by 65.14%.
- **Fuzzing based Input Exploration.** We demonstrate that fuzzing tools can be used to compensate for LLM's weakness in reasoning program conditions and expand testing coverage by 6.26% to 8.91%.
- **An automatic unit test generation tool for Rust.** We present RUG, an automatic unit test generation tool for Rust that leverages a deep combination of LLM and program analysis. RUG improves the code coverage of 13.21% to 29.68%, and 11.20% to 21.81% when compared to existing traditional approaches and existing LLM-based approaches, respectively.
- **Practical Usage.** We submitted the 113 generated unit tests to 12 different popular Rust projects, 53 of them are merged by maintainers with positive feedback, taking the 75.14% of all the reviewed unit tests.

II. BACKGROUND

In this section, we first outlines the issue of test generation alongside the most current state-of-the-art approaches. Then we introduce the basics of the Rust programming language, which guarantees memory safety and performance through strict compiler checks. Furthermore, we highlight the obstacles encountered in automatic testing code generation for Rust.

A. Automatic Unit Testing

Although unit tests have been proven to be helpful in program quality during software development [20], it is time-consuming for human developers to write unit tests for each unit. In order to address this issue, there are many efforts to automate the unit test generations, including the SBST, fuzzing approach and learning-based approach.

SBST Approach. Searching Based Software Testing (SBST) leverages the source code, including the library code and application code, to find the possible function sequences and input data that can test new code. It utilizes different search algorithms to automatically generate tests guided by the coverage goal, and evolutionary algorithms are demonstrated powerful for this searching process. However, when applying the SBST to Rust, there are two drawbacks: 1) Due to the complex type system of Rust, the mutation operators can not guarantee the correctness of the generated code. 2) the evolutionary algorithms primarily relies on existing user program to mutate, limiting their ability to generalize and adapt to broader testing scenarios.

Fuzzing. Fuzzing has grown in popularity for software testing due to its efficiency and reproducibility. It generates and mutates the input to test the target program and collect code coverage as feedback. Based on coverage feedback and domain knowledge, modern fuzzers are proficient in exploring paths within testing programs and can even uncover bugs that have existed for years [21]. With its ability to identify input data that cover new code, fuzzing holds potential in aiding the generation of test data for unit tests. However, utilizing fuzzing solely for unit test generation presents the following issues. First, fuzzers require specific fuzzing harnesses to work with, and the quality of the fuzzing harnesses largely affects fuzzer performance. Although program analysis tools such as RULF [22], RPG [23] are proposed to automate this process, it still suffers from the ultimate search process and lack of readability. Second, the number of corpora generated during the fuzzing process is large, requiring post-processing steps to filter and select. Finally, the readability of the fuzzing corpora is poor and is difficult to directly adopt into the repository.

LLM Approach. Large Language Models, like ChatGPT [24] are state-of-the-art language models that are trained on vast amounts of language data. These advanced language models, with their neural architectures and expansive understanding of languages, are able to capture intricate patterns and relationships in language usage. As a result, they express an impressive capabilities in the software-related activities including unit test generation [25], [26], coding assistance [27] and program debugging [28]. Provided the context and questions as prompts, LLM is able to generate the relevant test program.

To evaluate the capabilities of LLM to generate unit tests, Yuan et al. [26] conducted a thorough evaluation in the context of JAVA. Their study reveals that ChatGPT is able to write a reasonable number of unit tests, but a large proportion (57.9%) encountered diverse compilation errors. Although the rest of the tests have been successfully compiled, only 24.8% of the executions succeed because the generated assertions are incorrect. Finally, the unit test generated from LLMs exhibited commendable readability. All three findings are consistent with our results under the Rust context, and in this work we propose our solutions for these problems.

[2]We will open source RUG upon publication.

B. Rust Programming Language

Rust is an emerging programming language for low-level and system development with memory safety guarantees. It has two parts; safe Rust is designed to achieve native performance in a memory-safe way guarded by the compiler, and unsafe Rust requires developers' help for memory safety. (Safe) Rust[3] employs a robust type system that imposes strict disciplines, effectively mitigating security concerns and ensuring memory safety. In addition to its advanced type system, Rust supports traits to define the common behaviors and generic parameters to extend its usability. Next, we will describe some necessary language features and concepts used in the latter context and highlight the challenges they present.

Type Checks. Rust is a statically typed language, all the variables are assigned a specific type at the compilation time. Rustc compiler will reason and check the type correctness and bounds for every variables in the program. As a result, the unit test generation tool needs to ensure the correctness of variable types and bounds.

Ownership. Rust's ownership ensures that every variable has a relative memory it binds to, and this memory is immediately reclaimed when the owner variable goes out of scope. This ownership can also be transferred, and the original variable loses the access to the value, ensuring the variable is valid when it's being accessed.

Traits and bounds. To further empower the flexibility, Rust supports generic parameters to allow developers reuse the functions. Trait bounds are used as desired restrictions for the generic parameters. Therefore, when synthesising programs for generic functions, synthesizers are expected to find qualified candidate types as generic parameters, otherwise the Rustc will find the missing or mismatch of generic parameters.

The aforementioned features along with other compiler checks(e.g. mutability, lifetime, etc.) introduce new challenges for test statement construction or program synthesis in Rust. For instance, the recombination operator in a genetic algorithm takes the parent programs(e.g., program `A`, `B`) as inputs and 'breeds' them to generate child programs. In Java or C/C++ language, with the help of the type analysis, synthesizers generate a program with cross-usage of variables (a variable is created in `A` and used as a parameter in `B`). However, this step does not always hold in Rust because, except type correctness, the variable's ownership, mutability, and lifetime need to be correct as well. We argue that managing these requirements simultaneously is not trivial under the constraints of safe Rust. Furthermore, since Rust performs these checks during compile time, it is difficult to bypass these challenges.

III. Challenges

In this section, we use a motivating example in Listing 1 to showcase the challenges to automatically generate unit tests. Based on the root causes of these challenges, we categorize them into two classes and propose our solutions in §IV.

[3]The goal of code generation is for safe Rust, we still call it Rust for simplicity.

Listing 1. Motivating examples for encode function. Details of struct/trait defintions are omitted. The challenge comes from lines 10-12; which provides an implicit definition of Config trait.

```
1  fn encode<E: Encoder> (&self :char, encoder: E)
2      -> Result<EncodeError> // target function
3  // impl for Encoder trait
4  impl<W:Writer, C:Config> Encoder for EncoderImpl
5  pub struct EncoderImpl<W: Writer, C: Config>
6  // impls for Writer Trait
7  impl Writer for SliceWriter
8  impl Writer for IoWriter
9  // proxy impls for Config Trait
10 impl<T> Config for T where T: R1 + R2 + R3
11 // def for Configuration, impls R1, R2, R3
12 pub struct Configuration<R1, R2, R3>
```

A. Motivating Example

The motivating example in Listing 1 shows a target testing function `encode` in line 1 and the related data structures in the `bincode` crate [29]. The function `encode` takes the first argument `self` as a *char* type, serializes it from memory into raw bytes and stores them in the specified location (specified in `encoder`). In order to test the target function, developers need to prepare two concrete variables: one is a simple variable with `char` type and the other is a complex instance of `E: Encoder` trait.

In this code snippet, `EncoderImpl` is a candidate implementation of `Encode`. However, `EncoderImpl` itself depends on two other traits: `W: Writer` and `C: Config`, representing any general `Writer` or `Config`. `Writer` controls the output location of the raw bytes, and `Config` controls the way to encode the memory object(*e.g.,* big endian or little endian). In lines 7-8, we can resolve a concrete instance of `Writer` named `IoWriter` or `SliceWriter`. As for the `Config` trait, instead of providing a direct definition, developers need to deduce that the 'proxy' definition of `Config` on line 10 implies that any type `T` satisfying the union of R_n traits can be an implementation of `Config`. Thus, RUG needs to search across the source codebase for the intersections of candidates implementing R_n trait, which is `Configuration`. After getting a valid `Writer` and `Config`, a valid testing context is built for the `encode` function and ready to test.

Generating a unit test for function `encode` highlights the two previous challenges. First, to pass the compiler check, the generation tool needs to infer the correct instance of the trait based on the function declaration, ensuring that the generated code adheres to the complex compiler rules. Through our evaluation on `GPT-3.5` and `GPT-4`, even all relevant source code, rustdoc, and sample code are presented in the prompt, it is still difficult for LLM to generate the correct tests. Second, to enhance the coverage of the code, the generated code should encompass various regions within the function. Even if the test generation step succeeds, both the LLM and SBST approaches face challenges in reasoning about the conditions, and thus cannot effectively improve code coverage. By analyzing the generated tests and their failure reasons, we identified the root

causes of these two challenges and classified them by their sources as: the challenges caused by Rust and the challenges caused by the LLM models.

B. Challenges to Pass the Compiler Check

The Rust compiler enforces strict checks for all variables, making it challenging even for humans to write Rust code, let alone automatic tools. Two common mistakes that often require significant time for human engineers to diagnose and resolve are trait errors and path errors.

Traits and bounds. `Trait` is extensively used in Rust to define the shared behaviors across different structs, while `trait bound` serves as the restrictions of available generic parameters can be used in Rust library. For example, in Listing 1, the `E: Encoder` indicates the generic paramter `E` must satisfy the `Encoder` bound. Therefore, to generate a concrete test, it is vital that traits and trait bounds are filled with appropriate instances[4]. In the motivating example, the second parameter `encoder` needs to be an instance of `Encoder` trait, which further relies on the `Writer` trait and `Config` trait. Moreover, the `Config` trait only has a proxy definition shown in line 10 in Listing 1. Therefore, these complex composition and proxy of trait & trait bounds become the burdens for LLM to correctly reason the proper candidates, which lead to compilation errors of type mismatch [30]. Besides, Rust's separation of data definition and implementations makes it more difficult for LLMs to identify correct type information from the mixed source code.

Definition paths. To ensure successful compilation, all type and function definitions within the testing program must be explicitly imported into the context. This is hard for the purely LLM based approach. In the motivating example, the target test function `encode` is part of the `Encode` trait, which is implemented for the `char` type as the `self` parameter. Hence, to trigger the `encode` function for `char`, the `Encode` trait must be explicitly imported into the testing context.

Cascading Errors. The unit test composition process involves several steps. First, we need to figure out the calling context, *e.g.,* reasonable input data for each variable. Second, we need to properly passing those context to the target function. Finally, we need to resolve the correct assertion statement. When prompt LLM through this process, it requires complicated reasoning chain from the model. Even a minor error at any step could culminate in an inaccurate final prediction. Therefore, LLM must accurately navigate the challenges posed by the Rust language as mentioned previously.

C. Challenges of Low Code Coverage

Even after overcoming the challenge of generating unit tests that pass the compiler's checks, low code coverage remains a significant issue when composing unit tests for Rust programs.

[4]Although Rust provides *dyn* keyword to delay these bound checks until runtime, it's rarely used in the Rust crates.

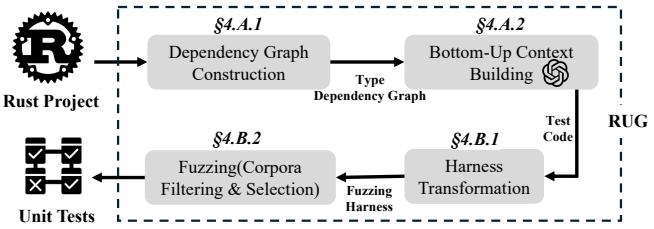

Fig. 1. The general workflow and components of RUG. After building the type dependency graph from the input Rust project, RUG leverages bottom-up context building to handle the compilation challenges and fuzzing to resolve the coverage challenges.

Difficulty in Path Exploration. In contrast to traditional approaches, LLM currently lacks the capability to examine the execution of the program. This limitation results in challenges in reasoning about branch conditions and exploring various code paths. Moreover, in the context of test assertions, the oracle is responsible for verifying the correctness of the function execution. While LLM is able to generate appropriate fields for verification, because of failure to statically reason the path that will be executed, they may yield incorrect values as oracles. Such a deficiency in accurately reasoning about data conditions and outcomes further complicates the LLM's capability to produce valid and reliable test assertions, underlining the need for enhanced techniques.

IV. RUG DESIGN

To address the challenges of test generation and coverage, we propose a new tool called RUG, which uses a semantic-aware approach to guide the LLM building the test and leverages fuzzing to expand testing coverage. The workflow of RUG is shown in Fig. 1. Taking a Rust project as input, RUG first constructs a type dependency graph for the parameters of the target function, then it resolves the concrete types and the corresponding code for each node in the dependency graph from the bottom. After the root node is resolved, the unit test context is complete. Finally, RUG uses fuzzing to explore valid inputs and enhance code coverage.

In this section, we first introduce the context-building approach of RUG in §IV-A and show the fuzzing process in §IV-B.

A. Testing Context Construction

To build the unit test context, RUG follows a two-step process for each target function: 1) RUG applies static analysis to construct a type dependency graph for the parameters of the target function; 2) Starting from the leaf node, RUG adopts the bottom-up approach to automatically build the concrete code with the help of LLM.

1) Constructing Type Dependency Graphs: To build the unit test context, RUG first statically builds the type dependency graph. To better explain our approach, we use $G = (V, E)$ to denote the type dependency graph, where G is a directed

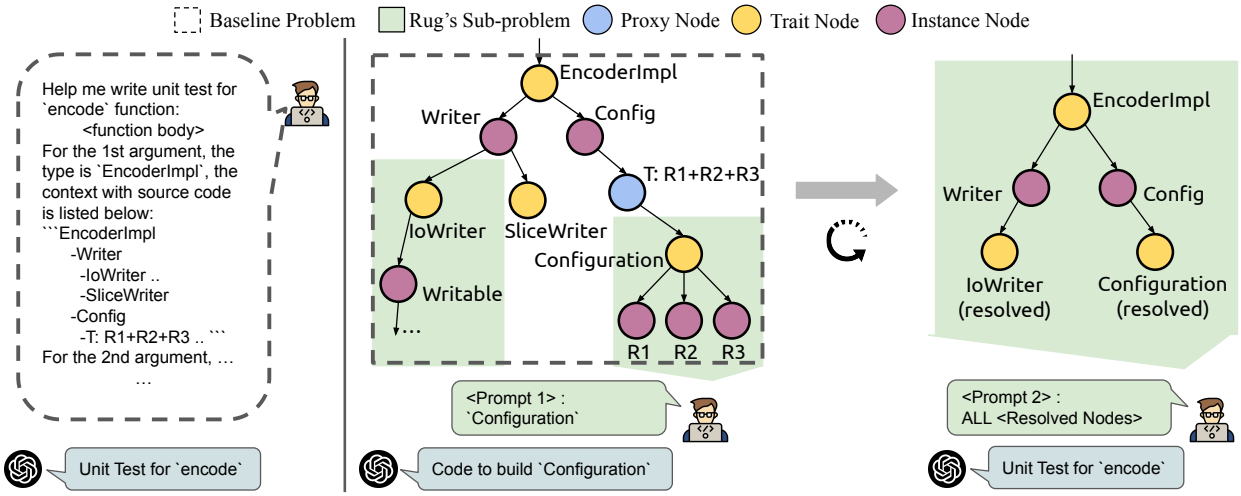

Fig. 2. RUG's bottom-up context building example. The left side represents the baseline's one-shot approach, requiring LLM to generate the test with a long context, leading to buggy output. RUG automatically divides the task into subproblems and simplifies the task. The prompt of the RUG is in Fig. 3 and Fig. 4.

graph; V denotes the nodes in the graph and E denotes the edges. The details of the graph are defined as follows:

- $\forall v \in V$, $v \in \{v_{trait}, v_{instance}, v_{proxy}\}$, where v_{trait} denotes the vertices with trait bounds constraints; $v_{instance}$ represents the vertices with concrete types like `struct` or `enum`; v_{proxy} is a kind of special vertices for a proxy definition (line 10 in Listing 1), denoting the logical substitution of type compositions.
- $\forall e \in E$, e is a directed edge indicating that the start vertex depends on the end vertex. For example, in Listing 1, line 4 can be represented as $EncoderImpl \rightarrow Writer$, $EncoderImpl$, and $Writer$ are vertices in the graph.

In this graph, the leaf vertices are defined as those with concrete types. For example, in Fig. 2, which illustrates part of the type dependency graph for Listing 1, the vertex `Writer` is considered an instance vertex, but since it does not have a concrete type, it is not a leaf vertex.

2) Bottom-Up Code Generation: Given the type dependency graph G, treat each node in the graph as a subproblem, and the unit test generation problem can be modeled as follows:

Given a type dependency graph G and parameters p_i of the target function, for each p_i, the context of the unit test to be generated is a subgraph $g \in G$, such that $\forall s \in S$, s starts with p_i and ends with primitive type nodes, where S denotes all topological sequence permutations of g.

In Fig. 2, the subgraph of `EncoderImpl` is the scope of the target parameter to be solved. For existing LLM based auto-testing tools [7], [8], [30], [31], testing code generation is finished in one shot like the left part in Fig. 2: they collect the context and ask for the testing code, then check the output and apply automatic fixes or retries if necessary. This one-shot approach usually produces a long context for LLM to infer, increasing the difficulties of code generation due to the strict compiler checks and cascading errors in §III-B.

RUG divides the generation of the test code into small steps as checkpoints, reducing the difficulties of each subproblem and guiding LLM to solve the final problem. Like the right part of Fig. 2, RUG begins with nodes that can be directly instantiated, using LLM to construct the unit test context. Once the unit test code for a node is generated, the results are used to resolve its parent node in the graph. A node is ready to be resolved once all its dependent nodes have been addressed. For example, in Fig. 2, after resolving the R_n types, RUG starts with the `Configuration` node, which can be resolved using the output of R_n type sub-problems. With the prompt shown in Fig. 3, the LLM generates the concrete code for `Configuration`. To ensure the correctness of the generated code for each node, RUG uses an oracle compiler to verify that the code is compilable. Once validated, the results are used to generate the test code for the type `T` as a proxy node, denoting the composition of type `R1+R2+R3`.

When providing concrete types to the LLM, different strategies are employed for different types of nodes. For instance nodes, RUG resolves the correct definition paths for each type within the context and recursively gathers *relevant* items, including structure definitions, target function definitions, trait definitions, structure field types, implementation relationships. For trait nodes, RUG queries the intersection of bound constraints to identify all candidate types. If no valid candidate is found, RUG generates a prompt describing the trait and uses LLM to implement it. Finally, for proxy nodes, RUG applies its proxy definitions within the compilation context to find candidate types.

After resolving and validating all the unit test codes for each parameter, RUG leverages a test generation prompt (Fig. 4) to ask LLM to generate the unit test code. The generated code is then validated using the compiler. The whole algorithm is shown in Algorithm 1: $getDependent$ finds direct dependents (if any) of the input type for instance nodes and valid candidate types for trait & proxy nodes, $descriptionGen$ generates the prompt description of the target type, $G[k]$ means accessing

Please help me fill in the sample code by creating an initialized local variable named {var-name} with type {var-type} using its constructor method or structural build in {crate-name} crate's {file-location} file. Fill in any sample data if necessary, …
<Rust sample code to fill>
(Optional) For the {dependent-type}, please reuse below sample code to construct.
<Dependent code from previous round>

Fig. 3. <Prompt 1>: template RUG used for each sub-problem. The optional paragraph is reusing the previous output to cut the context.

The target function is {fn-name} in {crate-name} crate's {file-location} file, its definition path is {def-path} and source code is like below:
<Target test function>
For n-th argument, {parameter-n-type} can be used, please use following sample code to construct it:
<Dependent code sample to reuse>
Please help me build unit test,
[Instructions to reuse the context]

Fig. 4. <Prompt 2>: Final test generation template RUG used to combine the sub-solution together and build the target unit test.

the value in the map G by the key k, and $\bar{D}$ is a set of verified candidates for the input parameter type t.

Algorithm 1 RUG's context construction algorithm. t is the target parameter type, G is the type dependency graph. Corner cases are omitted for simplicity.

```
1:  procedure BUILD_CONTEXT(t, G)
2:      D̄ ← {}
3:      S̄ ← getDependent(t, G)
4:      for s ∈ S̄ do
5:          if not s is resolved then
6:              build_context(s, G)
7:          D̄[s] ← G[s]
8:      if t is Instance then
9:          G[t] ← llmRequest(t, D̄)
10:         if not compileVerify(t, G[t]) then
11:             G[t] = descriptionGen(t)
12:     else
13:         if D̄ is Empty then
14:             D̄ = llmRequest(t, descriptionGen(t))
15:         G[t] = D̄
16:     return G[t]
```

3) Corner Cases: Although RUG's bottom-up building approach improves the quality of the generated code by minimizing the relevant context, there are several corner cases need to be handled. One of the corner cases is the loop in the subgraph g, indicating the presence of cyclical type dependencies. RUG will break the cycle by randomly determining orders and will remove the sample code in the prompt shown in Fig. 4.

In addition, for trait/proxy nodes, sometimes there is no valid instance type in the compilation scope and RUG will prompt LLM to implement the traits. Meanwhile, for the trait/proxy nodes with multiple instances, RUG provides different candidate selection strategies according to the user's preferences. The occurrence of nodes with multiple candidates available takes around 17.48% and by default RUG will choose the candidate in the local crate.

Finally, due to the uncertainties of LLM, sometimes the model fails to give a correct answer, RUG will mark the subtask as unfinished and continue the next step with descriptions in natural language. In practice, we found, even without a sample code, that sometimes LLM can generate the correct code for the current sub-problem.

B. Fuzzing for Input Exploration

Although the divided context building approach helps LLM to write executable tests, the generation of useful test data is another burden for LLM. In §III-C, we show that the root cause of this challenge is lack of the ability to inspect the state of the program during execution, so it is struggling for LLM to generate the corresponding data to trigger different conditions in the code path. In our motivating example, for the `char` type to encode, LLM usually prepares a valid ASCII symbol or one or two simple UTF-8 characters as test data. With the help of fuzzing, the test input is quickly scaled to complex UTF-8 characters and triggers the missed region. Thus, without fuzzing, it is challenging for LLM to prepare reasonable test data for high code coverage. In this section, we demonstrate how we prepare the fuzzing harness from generated tests and how we handle the redundant fuzzing corpora as postprocessing.

1) Fuzzing Harness Transformation: Fuzzers need fuzzing harnesses to test with, where all input data is provided by the fuzzers as raw bytes. To convert existing test code into a fuzzing harness, RUG leverages program transformation to construct test data from raw inputs and preserve the semantics of the test body. During the transformation process, RUG first identifies all the primitive data in the original test code and replaces them with local variables of the same type built from the fuzzer input bytes. Meanwhile, RUG records these initial data as seeds and saves them as the initial corpus for the fuzzing process. Second, to ensure the graceful execution of fuzzer, RUG disabled all assertions to help fuzzer finish its execution. Although this step may miss some bugs, it ensures the graceful execution of the fuzzer and improves the code coverage. After fuzzing and postprocessing, RUG will review these assertions and try to replace the value based on the fuzzing result.

2) Fuzzing Corpora Selection: By applying fuzzers to an existing test program, we can efficiently expand our test coverage through the new input data found by the fuzzers. Because of its efficiency, a few seconds of fuzzing can execute the program thousands of times with hundreds of corpora generated, which is far beyond the requirement for the number of tests. To further manage these corpora, we develop a source

code coverage-based, path weight guided corpora selection algorithm to filter and rank the corpora based on their coverage. The post-processing goes with two stages. First, RUG filters the corpora based on source code coverage, instead of fuzzers' bitmap coverage to remove the redundant inputs. Second, RUG uses *weight* to measure the importance of each code region [32] and the overall *weight* for each corpora to rank them. To calculate the weight, RUG assigns a default weight of **1** for each code region. For all the n corpora that touched this region, they will share the weight, gaining the score of $\frac{1}{n}$. The goal of this weight sharing design is to find the corpus that covers more unique code regions, and the algorithm is shown in Algorithm 2: src_cov is a function that takes a specific corpus and returns a set of code regions $\bar{R}$ that are covered by the given input. $\bar{W}$ contains the weights for each input corpus. Finally, based on the weight score and the number of code regions covered, RUG ranks the corpora and selects them according to the given threshold.

Algorithm 2 RUG's corpora ranking algorithm. $\bar{I}$ is the input corpora as a set.

procedure RANK-CORPORA($\bar{I}$)
  $\bar{W}, \bar{C}, \bar{D} \leftarrow \{\}, \{\}, \{\}$
  for $i \in \bar{I}$ **do**
    $\bar{R} \leftarrow src\_cov(i)$
    $\bar{D}[i] \leftarrow \bar{R}$
    for $r \in \bar{R}$ **do**
      $\bar{C}[r] \leftarrow getOrDefault(\bar{C}, r, 0) + 1$
  for $i \in \bar{I}$ **do**
    $weights \leftarrow 0$
    for $r \in \bar{D}[i]$ **do**
      $weights \leftarrow weights + 1.0/\bar{C}[r]$
    $\bar{W}[i] \leftarrow weights$
  sort $\bar{I}$ by $\bar{W}[i], \forall i \in \bar{I}$ in descending order
  return $\bar{I}$

V. IMPLEMENTATION

We implemented RUG for the Rust toolchain nightly-2022-12-10[5]. As shown in Fig. 1, RUG leverages LLM and fuzzer as black boxes, implements a testing context builder that deeply combines static analysis and LLM, fuzzing harness transformer, and corpora postprocessor.

A. Static Analysis

Wrapped as a compiler plugin, RUG works with middle-level intermediate representation (MIR) in Rustc to retrieve the semantic information. Specifically, for the target types in context, RUG recursively collect the relevant items including: 1) all the types' definitions; 2) all the definitions of the inner types; 3) all the types that implement the relevant traits; and 4) all the function definitions and the rustdoc of the relevant types. RUG's searching scope is limited to the target cage, and when a foreign trait or type outside of the current compilation scope is included, RUG stops searching for its relevant items.

B. Bottom-up Test Context Generation with LLM

RUG is implemented in Python. During the evaluation, we use `gpt-3.5-turbo-16k-0613` model as `GPT-3.5` and `gpt-4-1106` model as `GPT-4`. We set `presence penalty` to '-1' to encourage the model to reuse the context provided by us and leave the other configurations unchanged[6]. To handle the uncertainties of LLM outputs, RUG adds `system` prompts asking LLM to conform to specific formats and always provides a template code to fill out, which helps postprocess generated code. However, due to the nondeterministic nature of LLM, sometimes it is difficult for LLM to find the correct answer, so we set a maximum limitation of three times to try for each individual question. After three attempts, even if the answer is still wrong, RUG will cache the result and use natural language descriptions as hints for the next questions. After receiving the answers from LLM, RUG checks the correctness of the answer by compiling the code and detecting the types of local variables using an oracle compiler plugin.

C. Fuzzing Transformation and Postprocessing

RUG implements harness transformer with the help of Rustc compiler and 'syn' crate to transform the testing code into fuzzing harnesses while semantically preserving the test logic. For a given unit test, RUG first identifies and extracts the primitive test data as local variables, then converts these local variables from the input of the fuzzer so that the fuzzers can manipulate them for testing. The original test data are saved as the initial corpus seed for the fuzzer. In addition, RUG temporarily disabled all the assertions in the original test since these assertions may not hold because the test data are changed. Finally, RUG utilizes *bolero* [33] and libFuzzer to launch fuzzing testing. After the fuzzing process, RUG applies the corpora filtering algorithm to filter redundant corpora and the ranking algorithm to order corpora according to their importance.

VI. EVALUATION

To demonstrate the effectiveness of RUG, we conducted a comprehensive evaluation motivated by the following research questions:

- **RQ1:** How does the test generation performance of RUG, in terms of coverage, compare to traditional tools?
- **RQ2:** Compared with other LLM-based tools, does RUG show any benefits?
- **RQ3:** How does each factor contribute to RUG's testing coverage?
- **RQ4:** How is RUG's usability for real world applications?
- **RQ5:** How robust is RUG in different scenarios? Specifically, how does RUG perform on crates that the LLM has not been trained on? Additionally, how do different type selection approaches impact the results?

[5]The effort to support other toolchains is to handle the Rust compiler's MIR changes across different versions.

[6]RUG uses default values for temperature=1, top-p=1, frequency-penalty=0.

A. RQ1: Comparison with Traditional Tools

To answer Q1, we compare RUG with four of the latest different test generation tools for Rust, including RustyUnit [4] as a SBST approach, SyRust [6] as a constraint solving program synthesizer, and RULF [22], RPG [23] as fuzzing-based tools.

Comparison with General Searching Based Tools. RustyUnit is a SBST approach that leverages the DynaMOSA algorithm to mutate the existing Rust codebase and generate the unit tests. And SyRust is a semantic-aware program synthesizer that uses a SAT solver to generate valid Rust programs. To ensure a fair comparison, we applied RUG to their original benchmarks separately and used LLVM source code coverage [32] to measure the code region coverage and function coverage. When calculating the code coverage, we considered only functional code, excluding testing code and compiler-generated code.

We conducted our experiments on servers equipped with two AMD EPYC 7452 CPUs and 256GB of memory, running Ubuntu 22.04. The Rust toolchain version used is nightly-2022-12-10, and all tools are assessed using their default configurations. Since RustyUnit provided different evolutionary algorithms, we selected the most powerful DynaMOSA algorithm, which achieves the highest code coverage, for comparison with RUG. For SyRust, we run the synthesizer with their default timeout of 10 hours for each project[7].

The evaluation results are shown in Table I. RUG achieves 54.84% coverage on RustyUnit's and 52.28% on SyRust's, outperforming all the works. RustyUnit's seeded DynaMOSA algorithm, which is part of the genetic algorithm family, relies on existing code as 'parents' to mutate unit test statements. However, because the usage of each function within a crate is often imbalanced, this approach struggles with less frequently used functions. SyRust, on the other hand, requires a manually crafted template of function arguments for synthesis, necessitating developer input and limiting the synthesis to the template input. Overall, RUG's superior coverage is primarily due to its ability to construct calling contexts for a wider range of target functions.

Comparison with Fuzzing Tools. Besides comparing with SBST tool and program synthesis approach, RUG is evaluated against fuzzing-based tools: RULF [22] and RPG [23]. RULF constructs type dependency graphs to assist in generating fuzzing harnesses, producing a sequence of API calls through graph traversal. RPG extends RULF by adding support for generic parameters and prioritizes functions containing unsafe regions using a pool-based generator. We run all tools on RULF's benchmarks under default configurations to ensure a fair comparison. For each individual fuzzing harness, we set a timeout of 24 hours for RULF and 4 hours for RPG, since RPG generates much more fuzzing harnesses compared with RULF[8]. For tests generated by RUG, we conduct fuzzing for 60 seconds per function, which is a reasonable time for unit test generation. All fuzzing experiments were repeated three times, and the average values are reported.

[7]The crates are: bitvec, crossbeam, dashmap, imrc, ndarray, num-rational, slab for **data structure** and csv-core, encode-unicode, encoding-rs, hcid, sval, urlencoding, utf8-width for **encoding**.

[8]This setting outfits RPG's evaluation settings of 4 to 6 hours total timeout for each project.

Crate	Func	Region	Func	Region
	RustyUnit		RUG	
gamie	55.54%	30.79%	68.67%	72.24%
humantime	45.55%	26.67%	50.33%	64.92%
lsd	32.58%	40.23%	37.66%	43.98%
quick-xml	17.38%	24.61%	54.5%	62.76%
tight	24.70%	30.27%	32.24%	36.90%
time	75.26%	70.78%	68.13%	56.94%
mean	**37.23%**	**34.70%**	**49.96%**	**54.84%**
	SyRust		RUG	
data-structure	26.11%	31.19%	52.10%	56.03%
encoding	30.69%	28.51%	55.47%	48.54%
mean	**28.40%**	**30.65%**	**53.79%**	**52.28%**

Table I
COVERAGE COMPARISON BETWEEN RUG, RUSTYUNIT AND SYRUST. RUG OUT PERFORMS RUSTYUNIT BY 20.14% AND SYRUST BY 21.57%.

The results are shown in Fig. 5, where RUG achieves an average coverage of 54.9%, outperforming RULF's 25.2% and RPG's 41.7%. RULF's harnesses generation process is an NP-Complete problem, using a heuristic threshold to limit the generation process. Besides, RULF does not implement static analysis for generic parameters, preventing it from triggering functions that use traits (e.g. `json` crate). RPG improves upon RULF by supporting generic parameters and utilizing a pool-based generator to produce more fuzzing harnesses. However, RPG encounters a similar search problem as RULF and prioritizes functions with unsafe regions during the generation process, which affects the total number of harnesses. In addition, if no valid candidate is found for the given generic parameters, the RPG may not locate a candidate outside the current compilation scope, leading to a missing fuzzing harness. In contrast, RUG leverages LLM to generate the testing code, avoiding the search problem and covering more testing functions.

B. RQ2: Comparison with LLM-based Tools

To answer the RQ2 and demonstrate RUG's effectiveness in tackling the two challenges, we select 17 crates from the most downloaded on crates.io averaging 24,937 LoC per crate. Beyond their popularity, theese crates represent a diverse spectrum of Rust applications in system development: *mio*, *crc32fast* for system call and hardware instructions; *json*, *toml*, *bincode* for data serialisations; *hashes*, *uuid* for crypto computations; and *num-traits*, *ryu* for numerical computations.

For the baseline approach, we integrate three LLM-based approaches: ChatUniTest [8] for code generation, TestGen-LLM [12] for oracle compiler check, and RustAssistant [30] for code repair. The process is as follows: 1) We collect the relevant source code context of the target function and send it, along with the prompt, to the LLM to generate test cases. 2) The generated test code is then checked for correctness by an oracle compiler. 3) If the LLM output fails to pass the compiler

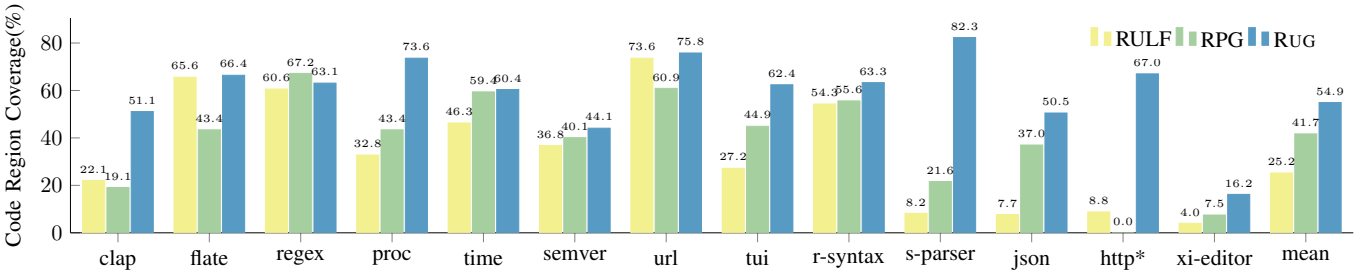

Fig. 5. Coverage comparison between RUG and RULF, RPG. *: RPG encounters an unexpected crash while running for `http` crate.

Crate Name (Downloads)	Tests ac/rej	GPT-3.5				GPT-4					Human Test Coverage
		Base		RUG		Base		RUG		Newly API Cov Rate	
		w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing		
bincode(49M)	4/0	1.57%	1.57%	22.92%	23.91%	16.63%	18.79%	44.67%	47.91%	74.11%	**64.58%**
chrono(128M)	22/13	37.88%	44.07%	47.2%	58.29%	54.04%	59.24%	56.90%	62.67%	73.05%	**76.66%**
hashes(266M)	**P**(7)	43.84%	43.84%	68.28%	68.28%	57.71%	57.71%	68.96%	**85.16%**	61.41%	**85.17%**
humantime(98M)	**P**(5)	63.09%	64.40%	67.02%	75.39%	74.08%	75.92%	74.61%	**80.37%**	40.00%	79.32%
itoa(221M)	1/0	26.00%	26.00%	82.00%	96.00%	96.00%	98.00%	100.00%	**100.00%**	83.33%	86.00%
json(203M)	-	28.10%	35.69%	44.60%	52.07%	62.26%	67.00%	70.25%	70.49%	47.33%	**72.36%**
mio(145M)	-	20.47%	20.47%	25.20%	25.20%	26.77%	26.77%	33.86%	**33.86%**	38.89%	24.19%
nom(114M)	6/1/**P**(14)	25.81%	25.81%	39.93%	40.04%	51.13%	51.17%	53.84%	53.87%	28.64%	**76.20%**
num-traits(185M)	-	36.02%	36.47%	43.20%	43.95%	46.94%	46.94%	47.23%	47.98%	90.36%	**50.58%**
demangle(93M)	**P**(14)	21.32%	21.62%	21.83%	65.99%	20.00%	74.82%	26.60%	**76.55%**	18.75%	72.25%
crc32fast(104M)	-	62.35%	64.71%	70.59%	71.76%	87.06%	88.24%	87.06%	**88.24%**	92.86%	68.24%
ryu(185M)	0/3	52.51%	95.28%	61.65%	97.64%	76.40%	99.42%	81.72%	**99.42%**	100.00%	87.85%
semver(168M)	18/0	61.40%	62.96%	62.54%	73.36%	72.36%	74.64%	74.22%	76.50%	95.24%	**84.33%**
textwrap(134M)	1/0	88.84%	92.56%	90.15%	94.31%	92.78%	94.75%	92.56%	**94.97%**	83.34%	87.53%
time(200M)	**P**(3)	33.08%	35.06%	48.98%	51.72%	55.34%	55.34%	79.89%	79.89%	66.06%	**96.48%**
toml(125M)	-	32.43%	37.06%	47.28%	49.02%	59.58%	64.40%	38.90%	38.90%	25.14%	**70.81%**
uuid(108M)	1/0	58.66%	64.44%	69.30%	77.20%	73.86%	75.08%	75.68%	**76.60%**	88.89%	61.40%
mean	-	40.79%	45.41%	53.69%	62.60%	60.17%	66.37%	65.11%	71.37%	65.14%	**73.18%**
Identifier	Ⓟ	Ⓐ	Ⓑ	Ⓒ	Ⓓ	Ⓔ	Ⓕ	Ⓖ	Ⓗ	Ⓘ	Ⓙ

Table II
CODE COVERAGE EVALUATION FOR RUG ON 17 POPULAR RUST CRATES. '**P**' INDICATES THE PR IS STILL PENDING FOR RESPONSE. THE THREE DIMENSIONS FOR SENSITIVITY TESTS ARE: GPT MODEL VERSIONS, GENERATION APPROACHES AND WHETHER APPLYING FUZZING. THE `newly API coverage` DENOTES THE APIS THAT `baseline` FAILED TO TEST AND ONLY COVERED BY RUG.

checks, we collect the error message, line numbers, and reasons for the failure, then ask LLM to fix the problems within the same context. Since ChatUniTest and TestGen-LLM are not implemented for Rust, we reimplemented them specifically for Rust during the preparation of the baseline experiment. We treat this pipeline of three approaches as the baseline, representing the best existing LLM-based practice.

Compared to `baseline`, RUG does not have an additional error fixing stage and employs a bottom-up building algorithm to generate the compilable test code. In addition, RUG transforms the test code into a fuzzing harness and takes advantage of fuzzing as a final step.

The results across different experiment configurations are shown in Table II. Compared with Ⓐ Ⓓ and Ⓔ Ⓗ, we show that RUG's overall approach improves the testing coverage by 21.81% and 11.20% under different LLMs. The improvement number of `GPT-4` is relatively smaller because the remaining untested code regions are limited, especially we only focus on the actual functional code and do not generate tests for derived functions[9].

[9]Rustc allows `#[derive]` to generate default implementations like `clone`, which will be counted as separate functions in LLVM source code coverage.

C. RQ3: Ablation Study

To demonstrate the effectiveness of RUG's two techniques(bottom-up building and fuzzing) and its sensitivity to different LLM models, we conducted sensitivity experiments for each of them, showing their individual contributions to the final results shown in Table II.

Bottom-up Building. To demonstrate the effectiveness of the bottom-up building approach of RUG, we evaluate the "newly covered APIs" shown in Ⓘ, representing target APIs that the baseline tools failed to generate the test context but can be resolved by the bottom-up approach of RUG. RUG successfully covers 65.14% of these APIs, highlighting its effectiveness. In addition, comparing between Ⓐ Ⓒ and Ⓔ Ⓖ, we show that bottom-up building helps improve code coverage by 12.90% and 4.94%, which accounts for about half of the total coverage improvement.

Fuzzing Approach. To show the improvement contributed by fuzzing, we launch RUG's fuzzing harness transformer on both generated tests. The result shows that the fuzzing component increases code coverage by 8.91% in Ⓒ Ⓓ and 6.26% in Ⓖ Ⓗ. Even when the initial method achieved high coverage,

such as 60.17% in Ⓔ and 65.11% in Ⓖ, fuzzing still provided an additional coverage boost of around 6%. This reinforces our insight that applying fuzzing can effectively enhance code coverage.

Large Language Model Versions. Finally, we test RUG's sensitivity to different version of LLM models(`GPT-3.5` and `GPT-4`). Clearly, `GPT-4` is more intelligent than `GPT-3.5` by showing about 20% improvement between Ⓐ and Ⓔ. The experiment result shows code coverage are all further improved by applying RUG for 21.81% in `GPT-3.5` and 11.20% in `GPT-4`. Besides, in terms of testing coverage, applying RUG on `GPT-3.5` Ⓓ achieves a higher coverage number than vanilla `GPT-4` approach Ⓔ.

D. RQ4: Practical Usability Evaluation

In this section, we evaluate the usability of RUG by comparing its generated tests with human-written tests to answer RQ4. For the 17 crates in Table II, we note that due to their popularity, these crates are actively maintained and well tested by the developers, achieving an average coverage of 73.18%. We refer to their coverage as the best that human developers can achieve in practice, and evaluate RUG's generated tests against the human-written tests to demonstrate its practical usage.

Code Coverage. As shown in the column Ⓙ, the developers achieve an average of 73.18% code coverage on these 17 crates, and by leveraging the latest GPT-4, RUG can achieve 71.37% in Ⓗ, which is comparable to human tests. In addition, among the 17 crates, RUG and human developers both achieve higher coverage on 8 crates and get the similar testing coverage for `hashes`, indicating the potential usage of RUG in the software testing process. In addition, for crates like `chrono`, `nom` and `time`, RUG largely expands the testing coverage than developers', showing the effectiveness of RUG.

For crates where RUG does not outperform (e.g. `bincode, toml`), the main reasons are as follows: 1) These crates are template libraries for (de)serialization, with few type implementations in the code base, making it difficult for RUG to find the valid candidates. 2) The code requires highly structured input to be fuzzed effectively, which is challenging for fuzzers without concrete structure definitions.

Readability. To evaluate RUG's tests' readability, we collect the test coverage of RUG and human developers and find ten missing tests for existing human-written tests. We directly leverage RUG's generated tests, without changing test bodies and send them as PRs to the open source projects. To our surprise, the developers are happy to merge these machine generated tests. RUG generated a total of 248 unit tests, of which we submitted 113 to the corresponding crates based on their quality and priority. So far, 53 of these unit tests have been merged with positive feedback.

Developers chose not to merge 17 tests for two main reasons: first, the target functions are imported from external libraries(16), and the developers do not intend to include tests; second, the submitted tests are closed after a long pending period(1). We are still awaiting feedback on the remaining 43 unit tests. In general, 75. 71% of the tests reviewed are merged by the developers.

Crates	Base	Human	RUG		
			LS	RR	UF
metrics_evaluation	53.25%	**70.45%**	69.48%	71.43%	63.31%
coolssh	25.37%	N/A	36.76%	31.27%	35.66%
cbored	45.67%	23.51%	**50.11%**	44.53%	50.70%
rust_mc_proto	16.07%	N/A	44.60%	43.49%	45.98%
atomic-waitgroup	36.51%	**80.95%**	41.27%	36.51%	41.27%
osrm_client	5.29%	**10.20%**	9.02%	8.04%	9.41%
europe-elects-csv	14.06%	N/A	20.31%	18.75%	25.00%
xelis_hash	49.51%	77.67%	**80.58%**	85.44%	77.18%
behindthename	21.71%	41.09%	**43.41%**	37.21%	43.41%
utapi-rs	14.24%	N/A	15.82%	15.19%	16.46%
mean	28.17%	**50.64%**	41.14%*	39.18%	40.84%

Table III
ROBUSTNESS EVALUATION OF RUG ON UNLEARNED CRATES AFTER THE LLM TRAINING CUT-OFF. THE 'N/A' DENOTES THE CRATES DON'T HAVE TESTS. '*': THE RUG WITH **LS** STRATEGY ACHIEVES **48.98%** EXCLUDING THE NO-TEST CRATES, CLOSE TO THE **50.64%** COVERAGE BY DEVELOPERS.

Token Consumption. Although LLMs are getting more and more cheap nowadays[10], automatic tools quickly consume large number of tokens by sending relevant source code as prompts. For example, under the `GPT-4` model, the cost of the `baseline` approach is around $1000. However, with the help of type-aware caching, RUG only takes 51.3% of the total tokens in the `baseline` approach and improves the coverage of the testing by 10.4%. RUG consumes less tokens because, for each method under the same structs with dependent types, RUG feeds each unique dependency to the LLM only once, caching and reusing the result without saving the prompt. This approach saves tokens for subsequent functions with the similar dependencies. In contrast, baseline approaches must feed the entire context, including transitive dependencies each time, leading to higher token usage.

E. RQ5: Robustness Evaluation

To evaluate RUG's robustness, we conduct two experiments to measure RUG's performance under unlearned crates and different candidate selection strategies for corner cases in §IV. For `gpt-3.5-turbo-0125`, its knowledge cut-off is September 2021 [34], and we carefully select ten crates that are **created** after January 2022 from crates.io as our benchmark: *coolssh*, *xelis-hash* for algorithm calculations, *behindthename*, *cbored*, *utapi* for FFI wrappers, *metrics-evaluation*, *europs-elects-csv* for string parsing, *osrm*, *mc-proto* for binary data paring and *atomic-waitgroup* for concurrency.

Unlearned Crates. The evaluation result is shown in Table III. Compared with the baseline LLM approach, RUG achieves an average improvement of 12.93%. When comparing against human-written tests, excluding the four crates without human tests, RUG achieves 48.98% on the remaining crates, just 1.66% less than the developers' tests. In addition, RUG achieves a

[10]As of March 2024, the cost of 1M tokens as inputs is $30 for `GPT-4` and $0.5 for `GPT-3.5`

higher coverage rate on half of the crates compared to human developers, showing RUG's robustness on unlearned crates.

Candidate Selections. Regarding candidate selections, we compare three selection strategies for the same crates used in the data leakage analysis. Those selection strategies are widely used : local crate selection(LS), round robin(RR), and unsafe first(UF). The LS is the default strategy of RUG, trying to cover more local types, and only conducts random selection when there are multiple local candidate types. In addition, we further evaluated two more strategies: RR tries to fairly cover the candidate types while selection and UF leverages the program analysis to find the candidate type with highest number of unsafe regions, which is close to RPG's ranking algorithm. The result of different candidate selections are shown in the Table III: compared with RR and UF, local crate selection(LS) prefers to use the instance from the current crate, increasing the chance of local code coverage. For RR and UF, they may select the outside instance as final candidate, leading to the potential drop in the local testing coverage.

VII. Related Work

A. LLM based Tools for Rust Automatic Testing

The powerful induction ability of LLMs shows their application in software engineering [25], especially in code generation testing, debugging and problem analysis [31], [35]–[40]. Among these works, RustAssisstant [30] leverages the iterative communications with LLM to fix Rust compilation errors. RustGen [7] took the nature English description as input and output a reasonably functional Rust code. Rust-Lancet [41] automatically fixes the compiler errors resulting from the violation of the Rust ownership rules. They are all different applications of LLM compared with RUG. Regarding unit test generation, CODAMOSA [11] integrates the traditional SBST approach with LLM to generate unit test code by asking LLM for mutation seeds when the search process is stuck. However, CODAMOSA is tailored for the Python language, which, due to its weak typing, does not present the same challenges associated with inferring correct types as seen in strongly typed languages like Rust. Therefore, the general difficulties related to the synthesis of Rust programs remain unaddressed, leading to low code coverage. In contrast, the RUG approach effectively addresses these challenges using static analysis to guide LLM to infer types.

B. Traditional Automatic Testing Tools

Before the emergence of the LLM, there is already a lot of work dedicated to building the automatic testing tools, including searching-based software testing (SBST) and fuzzing. SBST approach requires the existing calling context for the target function as input [42]–[52]. Then it uses evolutionary algorithms to develop new unit tests, aiming to cover sections of the code that have not been previously tested [9], [53]–[63]. For example, RustyUnit [4] utilizes the DynaMOSA algorithm to automatically generate the unit test for Rust.

Besides SBST, coverage-guided fuzzing approaches are used to assess different areas of the software.These approaches are particularly favored because they are capable of generating new inputs, thus extending the coverage of the code [3], [5], [64], [65]. For instance, `afl.rs` [66] is a fuzzing tool that depends on manually crafted fuzzing harnesses to probe the software. The combination of fuzzing with LLM can bypass the manual work of building a readable fuzzing harness and automate testing and analysis [38], [67]–[71]. Fuzz4All [68] is a recent work that proposes a universal fuzzy loop powered by LLMs to find bugs in different languages. However, Fuzz4All aims to find bugs in language compilers and rarely touches on the problem of low code coverage for generated code snippets. RUG builds a fuzzing harness transformer and adds fuzzing to further expand the coverage of the generated code.

Besides, RULF and RPG are the two related works that focus on automatically generating the fuzzing harnesses leveraging the API dependency graph. We note that compared to the API dependency graph of RULF or RPG, the dependency graph of type RUG is simplified by removing the function vertices, leading to a significant reduction in search space during the graph searching process.

VIII. Limitation

In this section, we discuss the potential limitations of RUG and possible future improvements. RUG relies on static type dependency analysis to divide the context for the following steps. Therefore, for other strong-typed languages such as Java, there is no fundamental burden to transplant RUG. For weak typed languages like Python, the accuracy and scope of the static analysis will affect the quality of the generated tests, and we argue that this is a general issue for these languages and other automatic testing tools have similar problems. In addition, RUG is bonded to the specific version of Rust and a general implementation can save time to accommodate changes.

To expand the code coverage of the generated tests, RUG uses fuzzing to explore the input space. In order to launch the fuzzer, RUG temporarily disables all assertions, which might miss some bugs while the fuzzing process. Apart from that, RUG's corpora selection algorithm may lose the code coverage based on the number limitation of generated tests. For the generated test, RUG does not consider the framework and coding style of the existing tests, which can be given as a sample prompt to LLM for better code quality.

IX. Conclusion

We propose RUG, which leverages LLM and fuzzing to automatically generate testing code for Rust projects. RUG proposes a bottom-up approach to address the difficulties of Rust program synthesis and fuzzing of the Rust program to expand the coverage. Evaluation shows that RUG achieves higher coverage than existing tools with efficiency and scalability.

X. Acknowledgment

We thank the anonymous reviewers for their helpful feedback and suggestions. This research was supported, in part, by the NSF award CNS-1749711, ONR under grant N00014-23-1-2095, the Technology Innovation Institute (TII), UAE, and gifts from Facebook, Mozilla, Intel, VMware and Google.

REFERENCES

[1] F. Taufiqurrahman, S. Widowati, and M. J. Alibasa, "The impacts of test driven development on code coverage," in *2022 1st International Conference on Software Engineering and Information Technology (ICoSEIT)*. IEEE, 2022, pp. 46–50.

[2] M. Siniaalto and P. Abrahamsson, "A comparative case study on the impact of test-driven development on program design and test coverage," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 275–284.

[3] M. Davis, S. Choi, S. Estep, B. Myers, and J. Sunshine, "Nanofuzz: A usable tool for automatic test generation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1114–1126.

[4] V. Tymofyeyev and G. Fraser, "Search-based test suite generation for rust," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 3–18.

[5] "American fuzzy lop." [Online]. Available: https://github.com/google/AFL

[6] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "Syrust: automatic testing of rust libraries with semantic-aware program synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 899–913.

[7] X. Wu, N. Cheriere, C. Zhang, and D. Narayanan, "Rustgen: An augmentation approach for generating compilable rust code with large language models," 2023.

[8] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *arXiv preprint arXiv:2305.04764*, 2023.

[9] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[10] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[11] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[12] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," *arXiv preprint arXiv:2402.09171*, 2024.

[13] Sourcegraph, "Cody," https://sourcegraph.com/cody, 2023, [Online; accessed 2024-03-22].

[14] GitHub, "Github copilot: Your ai pair programmer," https://github.com/features/copilot, 2023, [Online; accessed 2024-03-22].

[15] Rust for Linux Contributors. Rust-for-Linux/linux. GitHub. [Online]. Available: https://github.com/Rust-for-Linux/linux

[16] The Register. Microsoft to explore Windows 11 code written in Rust. Online article. [Online]. Available: https://www.theregister.com/2023/04/27/microsoft_windows_rust/

[17] Rust-GPU Contributors. Rust-GPU/Rust-CUDA. GitHub. [Online]. Available: https://github.com/Rust-GPU/Rust-CUDA

[18] Google Security Blog. Supporting Use of Rust in Chromium. Blog post. [Online]. Available: https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html

[19] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust: A mixed-methods study," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1269–1281.

[20] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[21] K. Serebryany, "{OSS-Fuzz}-google's continuous fuzzing service for open source software," 2017.

[22] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 581–592.

[23] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, "Rpg: Rust library fuzzing with pool-based fuzz target generation and generic support," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[24] OpenAI. Introducing chatgpt. [Online]. Available: https://openai.com/blog/chatgpt

[25] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[26] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[27] M. Wermelinger, "Using github copilot to solve simple programming problems," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 172–178.

[28] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," *arXiv preprint arXiv:2304.02195*, 2023.

[29] bincode org, "bincode: A binary encoder/decoder in rust," https://github.com/bincode-org/bincode, 2023.

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

[31] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[32] C. LLVM. Source-based code coverage - clang 11 documentation. LLVM Project. [Online]. Available: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[33] camshaft, "Bolero: fuzzing and property testing front-end framework for rust." [Online]. Available: https://github.com/camshaft/bolero

[34] "Openai platform models." [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo

[35] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2111–2123.

[36] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, "Cat-lm training language models on aligned code and tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 409–420.

[37] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 15–24.

[38] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[39] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[40] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[41] W. Yang, L. Song, and Y. Xue, "Rust-lancet: Automated ownership-rule-violation fixing with behavior preservation," 2024.

[42] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[43] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[44] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 342–357.

[45] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Empirical software engineering and verification. chapter search based software engineering: techniques, taxonomy, tutorial," 2012.

[46] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[47] O. Buehler and J. Wegener, "Evolutionary functional testing of an automated parking system," in *Proceedings of the International Conference on Computer, Communication and Control Technologies (CCCT'03) and the 9th. International Conference on Information Systems Analysis and Synthesis (ISAS'03), Florida, USA*, vol. 1. Citeseer, 2003.

[48] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[49] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 1–12.

[50] D. You, Z. Chen, B. Xu, B. Luo, and C. Zhang, "An empirical study on the effectiveness of time-aware test case prioritization techniques," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 1451–1456.

[51] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 213–224.

[52] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[53] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[54] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005-Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings 19*. Springer, 2005, pp. 504–527.

[55] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 727–737.

[56] S. Thummalapenta, J. De Halleux, N. Tillmann, and S. Wadsworth, "Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *Tests and Proofs: 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings 4*. Springer, 2010, pp. 77–93.

[57] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 193–202.

[58] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.

[59] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.

[60] X. Cheng and D. Devecsery, "Creating concise and efficient dynamic analyses with alda," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 740–752.

[61] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 486–498.

[62] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.

[63] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: automatic generation of fuzz driver using unit tests," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2676–2692.

[64] "American fuzzy lop plus plus." [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[65] R.-F. Team, "Rust-fuzz/arbitrary: Generating structured data from arbitrary, unstructured input." [Online]. Available: https://github.com/rust-fuzz/arbitrary

[66] T. rust-fuzz team, "afl.rs: Fuzzing rust code with afl," https://github.com/rust-fuzz/afl.rs, 2023, accessed: 2024-03-18.

[67] C. Yang, Z. Zhao, and L. Zhang, "Kernelgpt: Enhanced kernel fuzzing via large language models," *arXiv preprint arXiv:2401.00563*, 2023.

[68] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," *Proc. IEEE/ACM ICSE*, 2024.

[69] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[70] J. Eom, S. Jeong, and T. Kwon, "Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation," *arXiv preprint arXiv:2402.12222*, 2024.

[71] L. Huang, P. Zhao, H. Chen, and L. Ma, "Large language models based fuzzing techniques: A survey," *arXiv preprint arXiv:2402.00350*, 2024.