ClozeMaster: Fuzzing Rust Compiler by Harnessing LLMs for Infilling Masked Real Programs

Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, Baowen Xu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
{hongyangao2023, merlin, jiangchangwu}@smail.nju.edu.cn {yangyibiao, zhouyuming, bwxu}@nju.edu.cn

Abstract— **Ensuring the reliability of the Rust compiler is of paramount importance, given increasing adoption of Rust for critical systems development, due to its emphasis on memory and thread safety. However, generating valid test programs for the Rust compiler poses significant challenges, given Rust's complex syntax and strict requirements. With the growing popularity of large language models (LLMs), much research in software testing has explored using LLMs to generate test cases. Still, directly using LLMs to generate Rust programs often results in a large number of invalid test cases. Existing studies have indicated that test cases triggering historical compiler bugs can assist in software testing. Our investigation into Rust compiler bug issues supports this observation. Inspired by existing work and our empirical research, we introduce a bracket-based masking and filling strategy called `clozeMask`. The `clozeMask` strategy involves extracting test code from historical issue reports, identifying and masking code snippets with specific structures, and using an LLM to fill in the masked portions for synthesizing new test programs. This approach harnesses the generative capabilities of LLMs while retaining the ability to trigger Rust compiler bugs. It enables comprehensive testing of the compiler's behavior, particularly exploring edge cases. We implemented our approach as a prototype ClozeMaster. ClozeMaster has identified 27 confirmed bugs for *rustc* and *mrustc*, of which 10 have been fixed by developers. Furthermore, our experimental results indicate that ClozeMaster outperforms existing fuzzers in terms of code coverage and effectiveness.**

Index Terms—**Rust Compiler, fuzzing, large language model, bug detection**

I. Introduction

Rust, a modern, safe, and concurrent systems-level programming language [1]–[3], has garnered significant attention in both the software development and national-level cybersecurity fields in recent years [4]–[12]. According to a GitHub survey, Rust is the second fastest-growing programming language [13]. The White House administration advocates for developers to "preferentially use memory-safe languages such as Rust" in its latest report [14] in 2024, indicating the growing importance and potential of Rust in the software development and cybersecurity domains.

The adoption of Rust has increased, making bugs in the Rust compiler a significant concern for client users. Unnoticed errors during compilation can lead to undefined behavior or crashes at runtime, posing a serious risk for organizations relying on the Rust compiler for system development and critical tasks. Hence, systematically testing the Rust compiler is crucial [5].

Prior work on compiler testing can be classified into two categories: ❶generation-based fuzzers and ❷mutation-based fuzzers [15]. Generation-based fuzzers construct test cases using random syntax rules or templates [16]–[18], while mutation-based fuzzers generate additional test cases by creating equivalent programs or applying mutation templates and heuristics [19]–[23]. These methods have demonstrated effectiveness in testing compilers for mature languages like C/C++. However, due to the complex and diverse language features of Rust, developing a comprehensive generative fuzzer that covers its entire semantic landscape is challenging and time-consuming [5], [24], [25]. Furthermore, the evolving nature of Rust and the limited maturity of its analysis tools make implementing mutation-based fuzzers difficult [26].

Recently, the development of large language models (LLMs) has brought new advances to the field of software testing [27]–[32]. Titanfuzz proposes the direct generation of code snippets using LLM, synthesizing Python test cases for deep learning libraries by masking function bodies or parameters based on output results [30]. FuzzGPT fine-tunes LLM by collecting code segments that historically triggered bugs in deep learning libraries, generating Python test cases using textual prompts [29]. Fuzz4all and KernelGPT employ carefully designed prompts to guide LLM in generating test code for specific software or kernel programs [31], [32]. Despite notable achievements, these methods still face limitations in generating test programs for the Rust compiler. For instance, Titanfuzz and Fuzz4all rely on LLM for zero-shot learning or generating test programs based solely on textual prompts. However, due to Rust's status as an emerging programming language, existing open-source LLMs mostly generate meaningless or invalid Rust programs without considering program context. Although FuzzGPT addresses this by fine-tuning LLMs with code segments that historically triggered bugs, directly generating effective Rust compiler test programs solely based on the fine-tuned model remains limited due to the small scale of Rust programs [33].

Inspired by prior research [29], [34], [35], we observed that test code historically associated with software bugs exhibits potential characteristics for triggering bugs in software systems. Motivated by this, we analyzed code snippets that have historically triggered bugs in the Rust compiler. Considering the limitations of existing test case generation approaches using LLMs, we present our effective approach for the Rust compiler.

Approach. We introduce ClozeMaster, an innovative fuzzing framework specifically designed for testing the Rust compiler. ClozeMaster harnesses the power of LLMs and historical bug-triggering test inputs to address the unique challenges posed by the rigorous syntax of the Rust language. The core strategy adopted by ClozeMaster is a `clozeMask` approach. It involves masking specific code segments within parentheses in historical bug-triggering code snippets. Subsequently, an LLM is utilized to infill the masked code, completing the snippets. This process enables ClozeMaster to generate new and potential bug-triggering test programs for the Rust compiler, leveraging the LLM's capabilities for code completion. Our key insight is that historically bug-triggering code snippets encapsulate valuable domain-specific knowledge, which can effectively explore edge cases in the Rust compiler. Similar strategies have proven successful in testing other software systems, such as C compiler [34] and SMT solvers [35]. To maximize the utilization of the rich information present in historical bug-triggering inputs, ClozeMaster adoptes two-fold approach. Firstly, existing bug-triggering code snippets are collected, and specific code within parentheses is masked, serving as a foundation for generating new test code. Secondly, these bug-triggering code snippets are employed to fine-tune the LLM, enabling it to learn the syntax and semantic knowledge of code that can trigger Rust compiler bugs. This approach enhances the effectiveness of ClozeMaster in generating relevant and potentially bug-triggering test cases. To evaluate the effectiveness of ClozeMaster, we applied it to two practical Rust compilers actively used by developers: *rustc*, the official Rust compiler, and *mrustc*, an alternative Rust compiler developed by an individual developer. By employing ClozeMaster, we successfully identified have reported 37 bugs, out of which 27 are confirmed by developers. Notably, 10 of the 27 bugs have been fixed. Furthermore, compared to existing Rust compiler testing tools, ClozeMaster achieved higher code coverage and enhanced bug detection capabilities.

Contributions. We make the following major contributions:

- We analyze the bug reports in the Rust compiler and introduce a simple yet effective fuzzing approach named `clozeMask`, which involves masking the code segment within paired parentheses in the historical bug-triggering code snippets. Our proposed `clozeMask` strategy harnesses LLM's capabilities in code completion and maintains the code snippet's bug-triggering potential.
- We have implemented our proposed strategy in the form of a prototype called ClozeMaster. ClozeMaster, using LLM as a cloze master, can serve as a general and practical fuzzer for the Rust compiler.
- We conducted an extensive evaluation for ClozeMaster by applying it to two Rust compilers, namely *rustc* and *mrustc*. ClozeMaster successfully identified and reported 37 bugs within these compilers, of which 27 are confirmed and 10 of those confirmed bugs are fixed by developers. This underscores the effectiveness of ClozeMaster in exposing bugs for Rust compiler.

Artifacts. The implementation of ClozeMaster, as well as the list of bugs we have identified, have been publicly available at: **https://github.com/clozeMasterPro/clozeMaster**

Paper Organization. The rest of this paper is structured as follows. Section II presents an overview of Rust language, large language models, empirical analysis of Rust compiler bug reports, and the resulting motivation. Section III formalizes our approach and describes the implementation of ClozeMaster. Next, we elaborate on our extensive evaluation in detail in Section IV. In Section V, we formulate more discussions on our results. Finally, we survey related work in Section VI and the conclusion is in Section VII.

II. Background & Motivation

This section first presents a brief introduction to the Rust programming language and Large Pre-trained Language Models. Then, we motivate and illustrate our technique using empirical study and real examples.

A. Background

1) Rust Language

Rust, a systems programming language developed by Mozilla Research, was officially released in 2015 with the goal of providing solutions for high performance, memory safety, and concurrency [36], [37]. Compared to other programming languages, Rust exhibits higher syntactic complexity, primarily due to its ownership model and borrowing rules. Rust's ownership system handles memory with strict borrowing checks to ensure safety and prevent errors like null pointers and data races. It also features lifetimes for managing borrowing, requiring annotations with `<'a>` for safe access. While the syntactic features introduced by the Rust language significantly improve the safety of programs, their complex coding rules have also led Rust to be perceived as the "too intimidating, too hard to learn, or too complicated" in a survey among developers in the Rust community [24]. Furthermore, the stringent analysis enforced by the Rust compiler and the language's intricate syntax constraints reduce the likelihood of user-written code being accepted by the compiler [25].

Consequently, the steep learning curve of the Rust language and the rigorous syntax checks enforced by the Rust compiler make it highly challenging to generate Rust test programs through traditional methods reliant on manually constructed templates or rules [5], [25], [38], [39].

2) Large Pre-trained Language Models

Large Language Models (LLMs) have demonstrated impressive performance across multiple domains within Natural Language Processing (NLP), trained on massive internet-extracted text data [40]–[42]. These models employ deep learning techniques, typically based on the Transformer architecture. The Transformer effectively captures contextual relationships and generates coherent outputs. For example, in code generation or code completion tasks, LLMs undergo supervised learning, and training on extensive code data to grasp syntax, structure, and common patterns [33], [43], [44]. By maximizing accuracy in predicting the next code token, the model gradually improves understanding and generation of code semantics. Recently, researchers have leveraged pre-trained LLMs from open-source libraries, employing carefully designed prompt strategies or

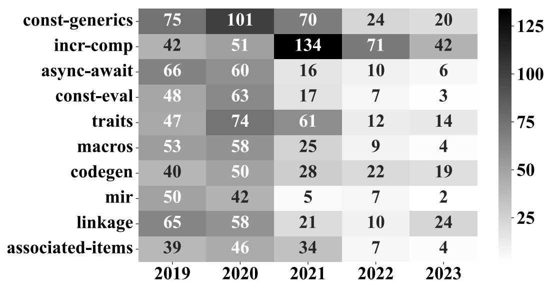


Fig. 1: Evolution of Bug Reports in *rustc* Components from January 2019 to August 2023: A Heatmap Analysis

fine-tuning techniques for specific tasks like domain adaptive code generation [45].

Although LLMs perform well on many NLP tasks, they struggle with downstream tasks of generating compiler test programs for emerging languages like Rust due to the scarcity of training data. Analysis of models such as StarCoder and Incoder shows that Rust constitutes less than 1/20th of their pre-training corpora [33], [43], leading to a long-tail problem [46]. Directly generating Rust programs with general-purpose ChatGPT4 is also challenging as it rarely triggers compiler bugs [29]. To address this issue, we have designed the `clozeMask` strategy to more effectively harness the power of LLMs for Rust-specific test code generation tasks.

B. Preliminary Study

We conduct a preliminary study on bug issues in the *rustc* compiler to understand typical characteristics of test cases historically triggering Rust compiler bugs. From this data analysis, we make two important observations.

1) Statistics of rustc Issues

A web crawler was developed to collect code snippets and label information from bug issues in *rustc*. As we focus only on compiler-related bugs, during the collection phase, we specifically targeted issues with the "`C-bug`" and "`T-compiler`" labels, resulting in a total of 7,474 issues (of which 5,242 are closed). We utilized BeautifulSoup [47], which is a Python library for parsing HTML and XML documents, to extract label information for each bug issue. Through regular expression matching, we extracted a total of 6,088 code snippets (including those mentioned in comments) from 3,819 resolved bug issues.

Our analysis shows it takes about 209 days on average to resolve a Rust compiler bug, with some taking over a decade (e.g., **#Issue10186**). This highlights the significant delay and complexity in fixing Rust compiler issues, which can persist across versions.

2) Findings

By statistically analyzing the code snippets and labels extracted from bug issues that historically triggered compiler errors, we have obtained two crucial observations.

① **The code that historically triggered compiler bugs contains a rich set of features, involving various components of previously released compilers.** Based on our analysis of code extracted from issues, we found that 31% (2016 out of 6088) of the code contains the declaration "**`#![feature(...)]`**". Of

```rust
#![feature(unsize, coerce_unsized)]
use std::{
    ops::CoerceUnsized,
    marker::Unsize,
};
#[repr(C)]
struct Ptr<T: ?Sized>(Box<T>);
impl<T: ?Sized, U: ?Sized> CoerceUnsized<Ptr<U>> for Ptr<T>
where
    T: Unsize<U>,
{}
fn main() {
    let foo = Ptr(Box::new(5)) as Ptr<dyn std::any::Any>;
    let arr: [u8; 4] = [1, 2, 3, 4];
    unsafe {
        println!("Array: {:?}", arr);
    }
}
```

Fig. 2: An Example: Test Code from *rustc* **Issue#98322**

the code snippets containing feature declarations, each code segment contains 1.15 feature declarations on average, with the maximum combination containing 8 features. These explicit feature declarations in the code can enable or disable compiler pipelines, which is a significant factor leading to compiler bugs. By examining the labels provided by developers for bug issues, we further analyzed the compiler components involved in these code samples. Figure 1 presents compiler components' top 10 ranked label information associated with confirmed *rustc* issue reports from January 2019 to August 2023. We discovered that the compiler code triggering historical bugs involves various components, such as const generics, incremental compilation, async/await, const evaluation, traits system, macros, code generation, intermediate representation, linkage, and associated item handling. Notably, const generics and incremental compilation, two components of *rustc*, have consistently shown a higher number of reported issues, indicating that they are components that are more prone to trigger bugs. From Figure 1, it can be observed that as *rustc* continues to iterate and developers improve the functionality of its components, there is a relative decrease in the number of newly added bug issues for the internal components of the compiler. However, there still remains room for testing and exploration. According to the observations by Zhong [34] and Deng [48], code that historically triggers compiler defects can explore edge cases in the compiler.

★ ***Conclusion***: Therefore, due to the ongoing development of certain features declared in "**`#![feature(...)]`**", they are not sufficiently stable and may contain potential bugs. When constructing test cases for the Rust compiler, special consideration should be given to these features.

② **The majority of test code that triggers the Rust compiler exhibits abundant usage of bracket structures.** These bracket structures, including "()", "{}", "[]", and "<>", play a crucial role in defining the syntax and semantics of Rust code. They are used for various purposes, including function and method invocations, control flow statements, data structure definitions, and pattern matching, among others. The widespread use of bracket structures indicates that Rust's syntax heavily relies on these constructs to provide expressive and flexible language features. Taking Figure 2 as an example, `#![feature(unsize,coerce_unsized)]` and `#[repr(C)]` are both attribute declarations, where the square brackets [] denote attributes. `#![feature(unsize,coerce_unsized)]` enables two experimental features, namely `unsize` and `coerce_unsized`. `#[repr(C)]` is applied to the `Ptr` structure to ensure its memory layout is compatible with C language structs. The syntax `<T: ?Sized>` represents a generic constraint, known as a generic bound, which limits the behavior of the type parameter T. In this context, it restricts T to be a type that may or may not have a known size. The expression `[u8; 4]` initializes an array of size 4 with elements of type `u8`. The `unsafe {}` block is used to indicate that the enclosed code contains operations that bypass Rust's memory and safety checks. It allows certain operations that are not considered safe by the Rust compiler. The purpose of using `unsafe {}` is to circumvent some of Rust's safety guarantees and gain additional flexibility or performance, but it requires the programmer to ensure correctness and safety manually.

★ ***Conclusion***: The presence of diverse bracket structures in code highlights compilers' need to handle various grammar cases accurately. When performing code generation tasks for Rust test code, it can be beneficial to incorporate code snippets featuring a range of bracket structures as contextual input to guide the LLMs. This approach ensures coverage of different syntactic combinations and boundary scenarios, thereby facilitating the generation of comprehensive test code that maximizes coverage across various compiler components.

III. Approach

In this section, we present our method, ClozeMaster, designed to discover new bugs in the Rust compiler. Our approach harnesses the power of LLM and a curated collection of code snippets that have historically exposed bugs in the Rust compiler.

We employ a `clozeMask` strategy (Section III-C) to synthesize new test cases. Our empirical analysis (SectionII-B2) shows the test code historically triggering Rust compiler bugs exhibits rich language features. Therefore, the core assumption of ClozeMaster is code snippets triggering historical bugs can explore vulnerable components and logic branches of the compiler. Leveraging these diverse code snippets, we can generate novel test cases to effectively fuzz the Rust compiler. Previous work in this direction required significant manual effort, where experienced developers had to devise a set of tailored rules and constraints specific to the compiler under test. Moreover, directly generalizing these approaches to testing the Rust compiler was challenging. In contrast, recent Large Language Model (LLM) advancements provide a natural, generalizable, and fully automated solution. Modern LLMs easily ingest historical programs via prompting or fine-tuning, generating programs similar to historical ones, effectively utilizing their code components. Figure 3 illustrates an overview of ClozeMaster. We systematically mine bug reports from the *rustc* code repository to collect code snippets historically triggering bugs in the Rust compiler. Our original code corpus includes the official Rust test suite [49]. Furthermore, the Rust-lang team provides the Glacier [50], which contains Rust code snippets and shell execution scripts that historically triggered internal compiler errors (ICEs). We also include these code snippets in our dataset. Leveraging these code snippets that trigger historical compiler bugs, ClozeMaster can generate new rust programs to effectively fuzz the Rust compiler, aiming to discover new bugs.

In our research, the ClozeMaster follows a four-step workflow, which can be outlined as follows:

- **Step-1. *Dataset construction***. This step involves web scraping to collect data for building a code dataset for Step-3 and a dataset with augmented data for Step-2. (Section III-A)
- **Step-2. *Fine-tuning***. The LLM is fine-tuned using historical test code that triggered bugs. This process enables the LLM to learn the characteristics of test code that historically triggered bugs. (Section III-B)
- **Step-3. *ClozeMask***. A code snippet is randomly selected from the code dataset, and the positions of bracket structures within the snippet are identified using a depth-first search (DFS) strategy. The content enclosed within the brackets is substituted with a "mask" label, and an LLM is used to populate the masked portion, thereby generating new test cases. (Section III-C)
- **Step-4. *Oracle and Duplicated Bug Filtering***. The test oracle is used to determine whether a bug is present. If a bug is identified, a bug deduplication process is performed. For new bugs, the test case is added to the code dataset for dataset updates. (Section III-D)

In the following sections, we will provide a more comprehensive explanation of each step in ClozeMaster.

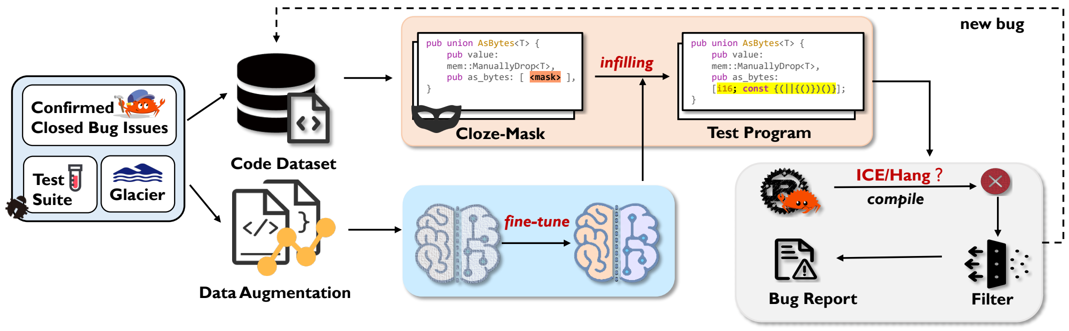
A. Dataset Construction

1) Building the Code dataset

Firstly, we implemented an HTML crawler to collect all bug issues related to *rustc* and extracted Rust code snippets for inclusion in our dataset. Specifically, we selected closed issues tagged with "`T-compiler`" and "`C-bug`" up until August 13, 2023. We used regular expressions to extract Rust code segments from these issues. Additionally, to supplement our dataset, we extracted suitable *.rs* files from the official Rust test suite and the Glacier repository, as well as Rust code from *.sh* files. These additions were incorporated into our code dataset. Importantly, to avoid interfering with experimental results, we removed code snippets from Glacier and the test suite that could trigger compiler bugs originally.

2) Data Augmentation

To fine-tune the LLM, it is necessary to augment the Rust code corpus in our raw code dataset. According to Dong et al.'s empirical research [51], data augmentation techniques that slightly disrupt source code syntax, such as random deletion (RD) and random swapping (RS), have proven effective in enhancing PL models pre-training. Therefore, we employ a random approach with a probability of p to delete code tokens at the code segment level in the original dataset. We manually set the value of p to 0.2, as this allows us to retain the original



```
pub union AsBytes<T> {
    pub value:
    mem::ManuallyDrop<T>,
    pub as_bytes: [ <mask> ],
}
```

```
pub union AsBytes<T> {
    pub value:
    mem::ManuallyDrop<T>,
    pub as_bytes:
    [i16; const {(||{()})()}];
}
```

Fig. 3: The Overview of CLOZEMASTER

semantic meaning of the code as much as possible when implementing the RD strategy. Additionally, we apply RS as a data augmentation technique for code segments in the original dataset. Specifically, for a given test program, we randomly select two statements and interchange their positions, repeating n times. The value of n is determined based on the number of tokens in the code. The original dataset contains 25248 code snippets. Through data augmentation, we expand the dataset to 100000 code snippets.

Algorithm 1: Main procedure of `clozeMask`

```
Function cloze($Code_{raw}$)
    $Pos_{mk}$ ← getMaskedPositions($Code_{raw}$)
    # DFS is employed to extract the positions of
    # bracket structures
    $Code_{mk}$ ← [ ]
    for $pos$ in $Pos_{mk}$
        $Code_{mk}$ ← $Code_{raw}$.replace(pos, "mask")
        $Codes_{mk}$.append($Code_{mk}$)
    end
    return $Codes_{mk}$
end
Function Infilling ($Code_{raw}$, $time_{max}$, $t$)
    $Codes_{new}$ ← [ ]
    $Codes_{mk}$ ← cloze($Code_{raw}$)
    for $Code_{mk}$ in $Codes_{mk}$
        if isSpecialMasked($Code_{mk}$) then
            # bracket pairs in "feature" block
            for $i \leftarrow 0$ to $time_{max}$
                $t_r \leftarrow$ randomFloat$(0, 1)$
                $Code_{new}$ ← codeInfilling($Code_{mk}, t_r$)
                if $Code_{new} \neq Code_{raw}$ then
                    $Codes_{new}$.append($Code_{new}$)
                end
            end
        else
            $Code_{new}$ ← codeInfilling($Code_{mk}, t$)
            if $Code_{new} \neq Code_{raw}$ then
                $Codes_{new}$.append($Code_{new}$)
            end
        end
    end
    return $Codes_{new}$
end
```

B. *Fine-tuning*

We used fine-tuning to improve the code generation task for Rust compiler bugs. Fine-tuning is a transfer learning technique where a pre-trained model is further trained for specific task requirements. We used the Incoder model [33], an open-source LLM supporting Rust code completion with fewer parameters. We performed fine-tuning by treating code snippets as training samples, encoded using Byte Pair Encoding (BPE) [52], [53]. The last few layers were trained to maximize the probability of predicting the next token. The Adam optimizer [54] expedited convergence. Retraining the Incoder model took over 48 hours, resulting in a language model capable of generating test programs from seed code snippets.

C. *ClozeMask*

To generate Rust test programs, we employed the `clozeMask` method to transform test code from an existing code dataset. The `clozeMask` algorithm consists of two processes: `cloze` and `infilling`, illustrated in Algorithm 1.

1) *Cloze*

The provided Rust code utilizes a stack and recursion to preserve the occurrences of bracket pairs (including "()", "{}", "[]", and "<>") within the given code text. By traversing each bracket pair's position in the stack, the original code snippets are replaced with the "mask", resulting in various code cloze frameworks for completion. This approach enables the creation of different code fill-in-the-blank templates by systematically identifying and masking the bracket pairs within the code text.

2) *Infilling*

Masked codes generated from the traversal of the original code are iteratively populated. In cases where the insertion point of the mask in a masked code is considered special, additional fillings are performed with varying frequencies and temperature parameters. This approach aims to obtain test codes that are more likely to trigger compiler bugs. If the populated code remains identical to the original code after filling, the generated result is discarded to ensure high-quality of test cases.

D. *Oracle and Duplicated Bug Filtering*

We assess the Rust compiler's performance using generated outputs with general testing oracles. Additionally, we develop a methodology for identifying repetitive bugs unique to the Rust compiler.

1) *Oracle*

To evaluate the generated outputs, we use two testing oracles: ICE and Hang, specifically designed for testing the

Rust compiler. The ICE issue in the Rust compiler refers to "Internal Compiler Error". When the compiler encounters an unhandled or unexpected situation, it crashes and displays an ICE error message. The Hang issue in the Rust compiler refers to situations where the compilation process becomes stuck or unresponsive. When the compiler gets stuck in an infinite loop, consumes resources for an extended period, or cannot proceed with execution, a Hang issue occurs. Both types of compiler bugs can be detected during compilation by examining stack information and observing compilation duration, without requiring a comparative compiler.

2) Duplicated Bug Filtering

We collect stack trace information for test cases triggering ICE errors into a bug dataset and gather *time-passes* information for test cases triggering Hang issues into the bug dataset as well. For test cases triggering bugs, we search the bug dataset. If the stack trace information or *time-passes* information matches an entry in the bug dataset, we discard the test case. If there is no match, we add the corresponding stack trace information or *time-passes* information to the bug dataset, update it, and mark the test case as "interesting" to generate a bug report. We store the test case in the code dataset to enrich it for subsequent iterations. This process eliminates duplicate bugs' impact on the method and ensures new bug detection.

IV. Evaluation

In this section, we conduct extensive evaluations to investigate the effectiveness of our method. Specifically, the experiments aim to answer the following research questions:

- **RQ1:** What is ClozeMaster's capability in bug hunting? (Section IV-B)
- **RQ2:** How does ClozeMaster compare against existing fuzzers in terms of effectiveness? (Section IV-C)
- **RQ3:** How do the key components of ClozeMaster contribute to its effectiveness? (Section IV-D)

A. Evaluation Setup

Target Compilers. We applied the ClozeMaster to two compilers, *rustc* and *mrustc*, as indicated in Table I. Of these two compilers, *rustc* represents the official Rust language compiler, while *mrustc* is an individually developed Rust compiler that currently supports a limited subset of Rust syntax. Hence, our experimental investigations primarily center around *rustc*, which holds dual importance as the sole compiler officially recognized by the Rust language development team and for its extensive support of the latest and most comprehensive Rust language syntax.

TABLE I: Target compilers we have tested.

Compilers	Versions	Build No.
rustc	v1.74-stable	79e9716c9
	v1.73-stable	cc66ad468
	v1.72-stable	d5c2e9c34
mrustc	v1.29.100	4c7e8171

- ***rustc*,** the official compiler of the Rust programming language, is developed in Rust itself. *rustc* follows a "self-compilation" methodology and employs the LLVM framework as its backend architecture.
- ***mrustc*,** a Rust compiler variant developed by an individual programmer using C++, represents a "simplified" version of the compiler that supports a subset of Rust syntax.

Bug Type. To ensure the reliability of the Rust compiler, our evaluation primarily focuses on the bugs mentioned in Section III-D, namely ICE and Hang.

- **ICE:** When the *rustc* compiler crashes during compilation due to an unexpected trigger, resulting in the output "`internal compiler error`" or "`compiler unexpectedly panicked`", we consider it an ICE issue. Similarly, if the *mrustc* compiler interrupts compilation and outputs "`BUG`" with "`core dump`", it is also a potential ICE problem.
- **Hang:** The "Hang" issue in compilers refers to a situation where the compiler becomes unresponsive or stuck during the compilation process, causing it to stop making progress or producing any output. According to industry conventions, a compiler is considered to have a hang issue if its compilation time exceeds 60s. Taking into account that the Rust compiler tends to have slower compilation speeds compared to traditional compilers [55], we have relaxed this timeout threshold T and set it to 180s. The 100% confirmation rate of bug issues related to Hang, as reported by us, validates the reasonableness of setting threshold T.

Competitive Baselines. We selected RustSmith [5], Rustlantis [56], and SPE [23] as our baseline methods. RustSmith and Rustlantis are open-source generative fuzzers, and we utilized their latest versions, RustSmith-1.30.0 and Rustlantis-0.1.0 in our experiments. SPE is a well-known compiler testing case generation method based on Skeletal Programs Enumeration. It was originally designed as a mutation-based fuzzer for C compilers. We have re-implemented SPE to meet the testing requirements of the Rust compiler. For a fair comparison, the seed programs used by SPE are identical to those used by ClozeMaster. Specifically, we extract the variables from the seed programs to obtain a variable set V, and the program skeleton P after the variable extraction. We then rearrange the variables and fill them back into P to generate new test programs. Since the rearrangement of the variable set V is a full permutation problem, we simplify the process by filtering the full permutations for each skeleton P (if the full permutation result exceeds a threshold T_θ=64, we randomly select 32 from the full permutation result) before refilling the skeleton. The proportion of cases where the full permutation results of the variable set V exceed the threshold T_θ is 4.84%.

Implementation and Evaluation Platforms. Our program generator is developed using PyTorch v2.1.0 and CUDA version 12.1. The implementation combines Rust and Python languages, with the core code comprising approximately 500 lines. The evaluation platform we employed is a multi-core server equipped with a 20-core CPU, 4 Tesla V100-SXM2-32GB GPUs, and 216GB of RAM. The server runs on the Ubuntu 18.04 operating system with Linux kernel version 4.15. All LLMs are executed on this server using GPUs.

TABLE II: Bugs found by ClozeMaster in *rustc* and *mrustc*.

Status	**rustc**		**mrustc**		**Total**
	ICE	*Hang*	*ICE*	*Hang*	
Reported	17	15	5	0	37
Confirmed	9	15	3	0	27
Fixed	5	3	2	0	10
Duplicate	4	0	0	0	4
Won't fix	4	0	0	0	4

B. RQ1: Effectiveness

From August 2023 to November 2023, we extensively tested *rustc* and *mrustc* using ClozeMaster. To ensure that all detected bugs were new, we always ran ClozeMaster on the latest trunk of the compilers for testing purposes.

Bug Count. In general, ClozeMaster generated over 1000 potential bug-triggering instances, and all confirmed bugs were triggered by the Rust compiler's basic optimization level, `-O0`. Following the bug deduplication principle mentioned in Section III-D, we reported a total of 37 bugs involving *rustc* and *mrustc*, out of which 27 bugs have been confirmed or fixed by the developers, as detailed in Table II. It is worth noting that all the confirmed bugs were previously unknown and could not be triggered by any historically known bug-triggering code. In total, these confirmed bugs resulted in 12 ICEs. Additionally, ClozeMaster revealed 15 confirmed Hang issues in *rustc*, which are significant vulnerabilities in the Rust compiler. Some of the vulnerabilities that developers deemed "*won't fixed*" were primarily due to the test cases utilizing internal features of the compiler that are not publicly accessible (**#116783**, **#116784**). This is mainly because we used certain code snippets from the testsuite that are not exposed to external developers.

Affected compiler components. To assess the performance of ClozeMaster in covering different components of the compiler, we classified the bugs that have been confirmed with their root causes based on developers' feedback and labeled information. Table III presents the categorization of these bugs. Taking *rustc* as an example, we found that the bugs identified by ClozeMaster span across multiple components of the compiler, including *parser*, *macros*, *associated-items*, *traits*, and *lifetimes*. This observation indicates that ClozeMaster has the potential to thoroughly explore different components and unveil compiler bugs. Additionally, we discovered that out of all the confirmed bugs, 8 were attributed to features, with seven being ICE issues and one being a Hang issue. Notably, the "`generic_const_exprs`" feature was responsible for the majority of these errors, occurring in 8 out of 5 instances. This finding suggests that increasing the number of mutations involving "`#feature[. . . ]`" during the `clozeMask` stage is more likely to generate test cases that expose compiler errors, corroborating our pre-study in Section II-B1.

Significance. Many bugs discovered by our tool have received prompt feedback and active discussions from the developers. For example, in the case of **Issue#116647** reported on *rustc*, the developers acknowledged the issue and provided a comment within one day, stating, "*Interestingly, the time gets significantly worse if the trailing comma for the last line is removed.*" Furthermore, to gain a better understanding of the significance of identified bugs, we conducted a study on how the confirmed bugs discovered by ClozeMaster impact the historical versions of *rustc*. Specifically, we input the submitted triggering test code, which causes new bugs, into a series of official release versions of *rustc* and observed whether any abnormal results occurred. The formal stable versions we selected for this study were 1.56 (released on November 1, 2021),1.60, 1.62, 1.64, 1.68, and 1.70 (released on May 31, 2023), which were the versions available before our experiment. These versions have undergone extensive testing with various tools, such as RustSmith and Rustlantis. As shown in Figure 4, we identified 17 bugs that initially appeared in *rustc* stable version 1.56. In other words, they evaded detection by other fuzzers and remained latent in *rustc* for over two years. Additionally, we discovered that some test cases triggered ICE bugs in certain versions but manifested as Hang bugs in other versions. This observation signifies that the test cases produced by our tool effectively investigate various edge cases of the Rust compiler. Furthermore, as subsequent versions of the compiler are developed, developers may encounter difficulties in effectively addressing these intricate edge cases.

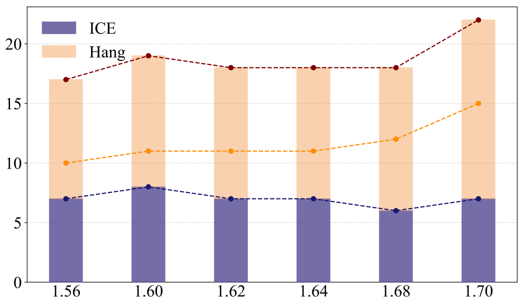


Fig. 4: Confirmed bugs that affect the corresponding release versions of *rustc*

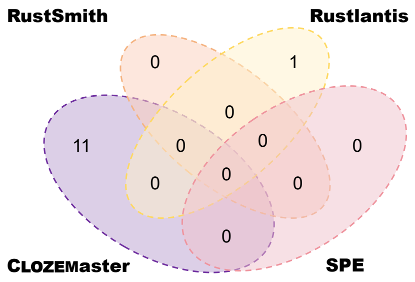


Fig. 5: The distribution of unique bugs detected by ClozeMaster and the baselines on the stable version 1.73 of *rustc*.

TABLE III: Compiler Components Implicated in Bugs Confirmed with Root Causes by Developers.

Compiler	**Component**	**Brief Description**	**Bugs**
rustc	parser	The parsing of Rust source code to an AST.	3
	macros	All kinds of macros (*custom derive, macro_rules!, proc macros, ..*)	1
	associated items	Associated items such as associated types and consts.	2
	traits	Trait system	2
	lifetime	lifetime related	1
	const-genetics	const generics (parameters and arguments)	1
	feature	when including '`#feature[...]`' in source code	8
mrustc	genetic-type-check	generic type was copied to a scope with no generics	1
	macros	All kinds of macros (*custom derive, macro_rules!, proc macros, ..*)	1

TABLE IV: Code coverage achieved by different fuzzers. The highest coverage is highlighted. The baseline fuzzers are bolded.

Techniques	*Code Coverage*
RustSmith	32.84%
Rustlantis	29.55%
SPE	62.02%
CLOZEMASTER	**64.34%**

Summary on Effectiveness: CLOZEMASTER possesses the capability to synthesize effective test cases that reveal novel bugs within the Rust compiler. These test cases expose bugs across various components of the Rust compiler, including those that have remained undiscovered in previous versions by developers.

C. RQ2: Comparison with Existing Techniques

In this research question (RQ2), we investigate whether our approach improves upon existing techniques, including Rustsmith, Rustlantis, and SPE. In the experiment, RustSmith and Rustlantis utilized randomly generated integer values as random seeds to guide the generation of their test cases. The SPE method, however, requires existing test programs as input. To ensure the fairness of the experiment, the original seed program dataset from CLOZEMASTER was used as the input for SPE. Compared to the conventional generation strategies employed by other fuzzers, the results demonstrate the effectiveness of our proposed framework, which leverages LLM as a cloze master to generate high-quality Rust test cases. Here, we primarily focus on comparing their bug-finding capabilities and the achieved code coverage across different testing tools.

1) Bug-finding Capability

To evaluate the bug-finding capability of CLOZEMASTER, we re-ran CLOZEMASTER and the baseline tools for 24 hours. Besides, to mitigate the potential influence of data leakage, we selected the stable version 1.73 of *rustc* (released on October 3, 2023) as the test subject in our experiment. The bug-triggering inputs used for training are primarily reported and resolved prior to the release of this version, thereby avoiding the data leakage problem. The results are presented in Figure 5. We observed that CLOZEMASTER identified the highest number of bugs, totaling 11, compared to the other tools. Furthermore, we noted that there was minimal overlap between the bugs detected by CLOZEMASTER and the other tools. Furthermore, we found that template-based test program generators such as RustSmith and Rustlantis can limit the diversity of the test programs they produce. For instance, the test programs generated by RustSmith often exhibit similar macro declarations and expression structures, while Rustlantis utilizes a singular feature declaration approach along with pattern matching. This is why they were able to find a relatively large number of bugs in the early versions of *rustc*, but as *rustc* has been continuously improved, these two methods have difficulty detecting new bugs in the newer versions of *rustc*. In contrast, SPE exhibits a strong dependence on the positioning of variables within the seed programs, and its mutation process is restricted to variable replacement alone. Conversely, the CLOZEMASTER supports more flexible mutations enabled by LLM that can better comprehend the contextual semantics of the code. This indicates that CLOZEMASTER exhibits strong complementarity with existing fuzzers and demonstrates a higher degree of bug-finding capability.

2) Code Coverage

We conducted a comparative analysis of code coverage on the Rust compiler between baseline methods and CLOZEMASTER, generating an equal number of 10,000 test cases for evaluation to ensure a fair comparison. These test cases were also executed using the stable version 1.73 of *rustc*. We employ an instrumentation-based approach within the Rust compiler, *rustc*, to collect code coverage metrics for test cases generated by different methods. Individual code coverage for each method was assessed, and the results are presented in Table IV. Our findings indicate that test programs generated by CLOZEMASTER achieved significantly higher coverage on *rustc*, with a relative increase of 95.92% compared to RustSmith and 117.73% compared to Rustlantis. Furthermore, CLOZEMASTER also outperformed the traditional variable-filling mutation-based method SPE.

Summary on Tools Comparison: Compared to both generation-based and mutation-based fuzzers, CLOZEMASTER outperforms in both bug discovery capability and code coverage for the Rust compiler, establishing its superiority in these aspects.

D. RQ3: Contribution of key components

Our methodology consists of five main components: historical code snippets triggering bugs, the `clozeMask` strategy, data augmentation, fine-tuning, and prompts based on code context. Therefore, we have designed five variants of CLOZEMASTER to assess their contributions to our approach.

- **CLOZEMASTER$_{nh}$**: It leverages regular Rust code snippets rather than the historical bug-triggering code snippets for fine-tuning the Incoder model in CLOZEMASTER. With the fine-tuned Incoder model, we then generate new test code using the `clozeMask` strategy by infilling the masked regular Rust code snippets.
- **CLOZEMASTER$_{nc}$**: It adopts a random line masking strategy rather than the `clozeMask` strategy for the historical bug-triggering code snippets. In other words, a line is randomly chosen for masking before utilizing the fine-tuned Incoder model for code completion.
- **CLOZEMASTER$_{nf}$**: It utilizes an untuned Incoder model to infill the historical bug-triggering code snippets which are masked by the `clozeMask` strategy.
- **CLOZEMASTER$_{na}$**: It leverages a fine-tuned model using non-augmented data and utilizes the fine-tuned model to populate historical bug-triggering code snippets masked by the `clozeMask` strategy.
- **CLOZEMASTER$_{nm}$**: Instead of employing any mask strategy, it directly utilizes a fine-tuned Incoder model to solely generate Rust code from scratch as test cases.

In RQ3, we consider the results of CLOZEMASTER as the baseline. We investigated the code coverage and bug-hunting results achieved by various implementation variants in this experiment. Following the experimental setup consistent with RQ2, we evaluated the coverage of various variants within a 24-hour timeframe on the stable version 1.73 of *rustc*. The code snippet dataset utilized by CLOZEMASTER$_{nh}$ is sourced from the Rust open-source projects, specifically the nomicon [57], rust-by-example [58], and rust-cookbook [59] repositories. It should be noted that the coverage of all the original test programs (including test cases that have historically triggered defects and the *rustc* test suite) is 73.64%. Considering that our method employs random algorithms when generating new test programs, not all test programs are utilized.

The results are presented in Table IV. It can be observed that in the absence of historical-triggered defective code, the performance of CLOZEMASTER$_{nh}$ significantly deteriorates compared to the baseline. This validates the effectiveness of incorporating historical-triggered defective code in our approach. The strategy based on `clozeMask` is equally important. Although CLOZEMASTER$_{nc}$ relying on randomly masked fillings also utilizes the context of Rust code snippets to assist the LLM in generating test cases, it does not outperform the baseline method in terms of coverage and bug-hunting. This is because the parentheses structure better preserves the syntactic and semantic integrity of the original code. Additionally, many of Rust's complex syntax features are composed of parentheses structures, as demonstrated in Section II-B1. CLOZEMASTER$_{nf}$ and CLOZEMASTER$_{na}$ exhibit a relatively minor decrease in both code coverage and bug-finding ability compared to the baseline. This could be attributed to the scarcity of code that triggers Rust compiler defects in the training corpus used to train the original LLM. As a result, the effects of fine-tuning and data augmentation components are not as evident as we had anticipated. Another notable observation is that without the contextual guidance provided by Rust code snippets, relying solely on the generative capability of LLM, CLOZEMASTER's performance is significantly compromised. The programs generated by CLOZEMASTER$_{nm}$ consist mostly of invalid characters and incorrect syntax, indicating the necessity of code context to guide LLM in generating test cases for complex compiler scenarios.

TABLE V: Code coverage achieved by different variants of CLOZEMASTER. The column labeled *#Prog.* represents the number of test cases generated in 24h, *Code Cov.* represents code coverage and *Change* represents the relative percentage decrease in coverage obtained by each variant compared to the coverage achieved by CLOZEMASTER.

	#Prog.	*Code Cov.*	*Change*	*Bugs*
CLOZEMASTER	18775	**65.67%**	-	**11**
CLOZEMASTER$_{nh}$	19001	42.34%	35.52%	0
CLOZEMASTER$_{nc}$	18984	50.01%	23.85%↓	2
CLOZEMASTER$_{nf}$	18890	61.89%	5.76%↓	9
CLOZEMASTER$_{na}$	18779	60.46%	7.66%↓	10
CLOZEMASTER$_{nm}$	22736	7.01%	89.33%↓	0

Summary on Contributions of Components: In conclusion, our comprehensive evaluation indicates that the key components of CLOZEMASTER exhibit remarkable effectiveness, particularly in enhancing code coverage and bug-hunting capabilities.

V. DISCUSSION

In this section, we mainly explore the complexity of the test cases generated by the CLOZEMASTER, the influence of the selected large language model on the CLOZEMASTER 's performance, and the potential threats to the validity of our research findings.

A. *Complexity and Case Analysis of Rust Test Code Generated by* CLOZEMASTER

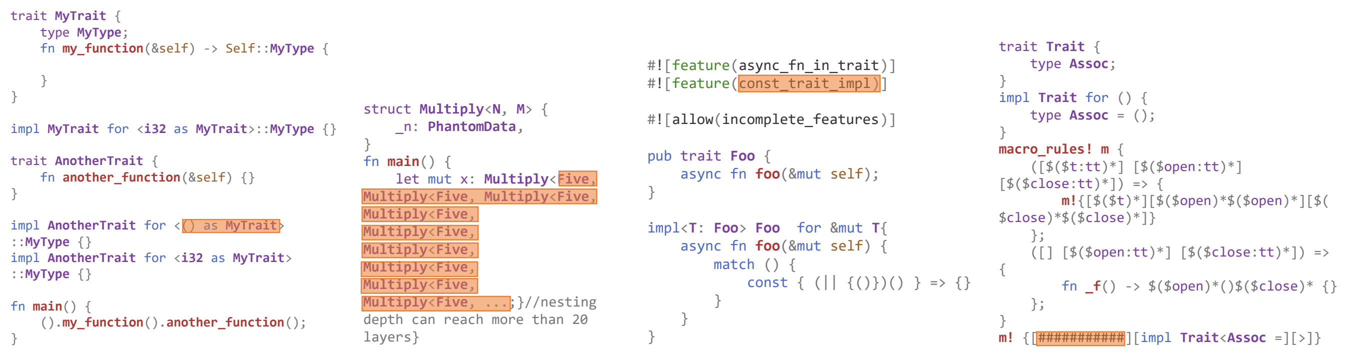
To evaluate the complexity of the code generated by the `clozeMask` strategy in CLOZEMASTER, we provide elucidation through two code evaluation metrics and exemplary analysis.

1) Metrics

We analyzed the 18775 test cases generated by CLOZEMASTER over a 24-hour period, focusing on the code snippets filled by the `clozeMask` strategy. The two code evaluation metrics used are:

Nested Parenthesis Structure. Our observations indicate that the average nesting depth of the `clozeMask-generated` segments is approximately 2, with the maximum nesting depth exceeding 20 levels.

Diversity of features. The `clozeMask` strategy generated 1230 different features. The most frequently occurring ones are `type_alias_impl_trait` (7.07%), followed by `const_generics` (6.28%), `generic_associated_types` (5.35%), `generic_const_exprs` (4.42%), and `const_fn`



```
trait MyTrait {
    type MyType;
    fn my_function(&self) -> Self::MyType {

    }
}

impl MyTrait for <i32 as MyTrait>::MyType {}

trait AnotherTrait {
    fn another_function(&self) {}
}

impl AnotherTrait for <() as MyTrait>
::MyType {}
impl AnotherTrait for <i32 as MyTrait>
::MyType {}

fn main() {
    ().my_function().another_function();
}
```

(a) #Issue116287

```
struct Multiply<N, M> {
    _n: PhantomData,
}
fn main() {
    let mut x: Multiply<Five,
Multiply<Five, Multiply<Five,
Multiply<Five,
Multiply<Five,
Multiply<Five,
Multiply<Five,
Multiply<Five,
Multiply<Five, ...;}//nesting
depth can reach more than 20
layers}
```

(b) #Issue116647

```
#![feature(async_fn_in_trait)]
#![feature(const_trait_impl)]

#![allow(incomplete_features)]

pub trait Foo {
    async fn foo(&mut self);
}

impl<T: Foo> Foo  for &mut T{
    async fn foo(&mut self) {
        match () {
            const { (|| {()})() } => {}
        }
    }
}
```

(c) #Issue117634

```
trait Trait {
    type Assoc;
}
impl Trait for () {
    type Assoc = ();
}
macro_rules! m {
    ([$($t:tt)*] [$($open:tt)*]
[$($close:tt)*]) => {
        m!{[$($t)*][$($open)*$($open)*][$(
$close)*$($close)*]}
    };
    ([] [$($open:tt)*] [$($close:tt)*]) =>
{
        fn _f() -> $($open)*()$($close)* {}
    };
}
m! {[###########][impl Trait<Assoc =][>]}
```

(d) #Issue116681

Fig. 6: Example cases generated by CLOZEMASTER that can reveal bugs in *rustc*. The portion generated using the `clozeMask` strategy is highlighted with light orange.

(3.34%). Among these, `generic_const_exprs` triggers the most Internal Compiler Errors (ICEs) (3/11), due to it being a compiler feature still under development.

2) Examples of reported bugs

Here, we present a case study to illustrate the key characteristics of the test cases generated by CLOZEMASTER capable of triggering bugs. Fig. 6 illustrates 4 bugs discovered in the *rustc* compiler by the test cases generated through CLOZEMASTER.

a. ***Grammatically correct statements based on context***: Fig. 6a is a Hang issue. The introduction of the “`() as MyTrait`” statement has added unexpected complexity to the interaction between the “`MyTrait`” and “`AnotherTrait`” implementations, causing the compiler to consume excessive resources when processing this part of the code, ultimately leading to a deadlock.
b. ***Complex nested structures***: Fig. 6b shows that the `clozeMask` strategy introduces a complex nested “`Multiply`” type structure, which overwhelms the compiler's type inference and parsing capabilities. The circular nested types generated by `clozeMask` exceed the compiler's processing limits. Consequently, the code's design surpasses the compiler's ability to handle it in a reasonable timeframe.
c. ***Different features***: Fig. 6c is an ICE issue. The combination of the “`const_trait_impl`” feature and the use of a constant closure within an async trait method implementation by the `clozeMask` strategy overloads the compiler's capabilities. The constant expression evaluation interacts poorly with async function handling, leading to the observed Internal Compiler Error. This complex interaction between language features has triggered an edge case in the Rust compiler.
d. ***Uncommonly-written code snippet***: Fig. 6d is a hang issue caused by the “`m!`” macro's recursive nature. Each invocation of “`m!`” adds more tokens to the macro's input without any termination condition. The “`###########`” triggers repeated calls to “`m!`”, resulting in infinite recursion. The compiler cannot handle the endless macro expansion, leading to a hang.

The `clozeMask` strategy in CLOZEMASTER generates complex test cases for the Rust compiler due to their deep nesting and use of diverse features. These complex cases can uncover compiler bugs by introducing unexpected interactions or edge cases. The findings suggest that CLOZEMASTER can provide valuable inputs for improving the Rust compiler's robustness.

B. Selection of Different LLMs

We propose the CLOZEMASTER framework, which can theoretically be combined with any pre-trained code generation LLM that supports the Rust language. However, our experimental results indicate that the model's performance on mainstream datasets (e.g., HumanEval [60]) does not necessarily make it more suitable for our CLOZEMASTER framework. Specifically, we evaluated the code infilling capabilities of the state-of-the-art open-source models for Rust code infilling tasks, namely Incoder [33], StarCoder [43], and CodeShell [61], on three Rust official tutorial datasets (nomicon [57], rust-by-example [58] and rust-cookbook [59]). We randomly selected 100 code snippets from these datasets, extracted their code templates (i.e., randomly masking the contents within pairs of brackets), and then used the LLM to fill in the missing code. The filled code was compiled using the *rustc* (stable version 1.73). The results, as shown in Table VI, indicate that the code filled by Incoder achieved the highest pass rate (with 11 programs successfully compiling and executing), while StarCoder and CodeShell had pass rates of 2% and 3%, respectively. Our manual analysis reveals that in the generated test programs, the latter two models incorrectly classify many Rust program errors as Python programs, resulting in the generated code being inconsistent with Rust syntax rules.

These results suggest that Incoder has greater potential compared to the other models for the task of Rust code filling. This is because Incoder employs a strategy of random masking between code snippets during the model training phase [33], which enhances its ability to fill in code within the middle of existing code. Therefore, Incoder is better suited for our CLOZEMASTER framework compared to other LLMs. We also

TABLE VI: Evaluation of Compilation Success and Code Coverage for Various LLMs in Rust Code Infilling Tasks

	Pass Rate	*Code Coverage*
StarCoder	2.00%	10.8%
CodeShell	3.00%	15.6%
Incoder	**11.00%**	**36.1%**

attempted to use GPT-4o [62] to directly generate test code for the Rust compiler. Within 24 hours, 11,883 test programs were generated, with a compilation pass rate reaching 32.22%, surpassing the lightweight open-source models listed in Table VI. However, no bugs were found. This is consistent with our observation in Section 6d. Considering that CLOZEMASTERis a prototype framework, using Incoder can achieve significantly better results than the current Rust compiler testing tools discussed in Section IV. We believe that applying a better model to our framework might yield better results in Rust compiler testing.

C. Threats to validity.

The primary threats to internal validity are bugs in implementation and experimentation, which have been mitigated through rigorous code review. The main threats to external validity come from the chosen subject systems, which have been reduced by selecting the two most popular Rust compilers, *rustc* and *mrustc* [5], [39], [63]. We have also used widely used metrics from prior fuzzing studies [35] to ensure real bug detection and code coverage.

VI. RELATED WORK

Generative fuzzers. Generative fuzzers which construct test cases based on random syntax rules [64]–[66] or templates [64], [67] and determine whether a bug is detected through differential testing or compilation crash is a common approach in compiler testing. For instance, tools like Csmith [16] and Yarpgen [17] are utilized to synthesize test cases for C language compilers, while CLSmith [18] is employed for OpenCL testing. RustSmith [5] is a prominent generative fuzzing method designed for testing the Rust compiler. It generates random Rust test code by constructing abstract syntax trees (AST) that adhere to the Rust grammar. The code generation process is context-aware, ensuring compliance with Rust's typing rules and semantics. In addition, Dewey et al. [39] have explored the automatic generation of Rust programs by formulating problems using Constraint Logic Programming (CLP). However, their work is limited to the Rust type system module. In contrast, CLOZEMASTER aims to uncover bugs in the Rust compiler by generating diverse test cases based on historical test programs that reveal bugs. These generated test cases exhibit greater diversity and cover a broader range of compiler components. To the best of our knowledge, we are the first to undertake such an endeavor on the Rust compiler.

LLM-based fuzzers. With the rise of Large Language Models (LLMs), LLMs have been utilized for software testing. WhiteFox [28] explores using LLMs for assisting white-box testing of compilers, while KernalGPT [31] attempts to leverage LLM for enhancing kernel fuzzing. Methods such as TitanFuzz [30] and FuzzGPT [29] have started incorporating LLMs into test case generation, but primarily focus on testing Python API libraries. Recently, the work by Gu [68] and Fuzz4all [32] demonstrated LLM effectiveness in generating compiler test cases, but are limited to mature programming languages like Python, Java, Go, and C++. To our knowledge, there is no research discussing the application of LLM-based fuzzers in Rust compiler testing, likely due to limited Rust training corpora and complex syntax. For instance, TitanFuzz designed methodology for testing Python API libraries, generating test cases from scratch using LLM and applying masking and filling transformations based on specific API or function calls. However, this approach cannot be directly applied to Rust due to ongoing language development and lack of mature corpus, making it challenging to generate test code solely using LLM without contextual information . Furthermore, the masking and filling techniques in TitanFuzz are limited to specific Python libraries and APIs, inapplicable to Rust. Some other works explored masking and filling methods, but predominantly for APR tasks [69]. FuzzGPT and HistFuzz [35] propose combining LLM with historically triggered code snippets to generate comprehensive test cases. However, these approaches rely on direct generation using LLM, which performs well on mature programming languages like Python and relatively SMT formula languages but is unsuitable for Rust, which is a relatively niche language, as discussed in section IV-D. Inspired by existing methodologies, we carefully selected an LLM suitable for Rust code completion tasks. By fine-tuning the LLM with input from historically triggered code snippets and utilizing Rust's bracket structures for masking and filling, we leverage the contextual information present in the original Rust corpus. Our approach focuses on generating targeted test cases specifically for the Rust compiler.

VII. CONCLUSION

We propose and implement CLOZEMASTER, the first approach for fuzzing Rust compilers by leveraging historical bug-triggering inputs. Our method involves using bug-triggering code collected from past incidents to generate cloze-masked code snippets and utilizing the contextual understanding capability of the LLM to perform fill-in-the-blank tasks and generate new test cases. To evaluate the CLOZEMASTER's effectiveness, we conducted a hunting campaign exposing real bugs in two widely used Rust compilers, *rustc* and *mrustc*. CLOZEMASTER successfully detected 27 confirmed bugs in *rustc* and *mrustc*, 10 already fixed by the developers. Furthermore, CLOZEMASTER outperformed state-of-the-art fuzzers in terms of code coverage and effectiveness. Our experimental results demonstrate potential for utilizing historical information for in Rust compiler testing.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their insightful suggestions on earlier versions of this paper. We are also indebted to the Rust compiler developers for inspecting and fixing our reported bugs. Yibiao Yang is the corresponding author. This work is partially supported by the National Natural Science Foundation of China (62072194, 624B2067, 62172205, and 62272214), the Jiangsu Natural Science Foundation (Grant BK20231402), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Fundamental Research Funds for the Central Universities (14380121).

References

[1] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 136:1–136:27, 2020. [Online]. Available: https://doi.org/10.1145/3428204

[2] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586037

[3] D. Hardin, "Hardware/software co-assurance for the rust programming language applied to zero trust architecture development," *Ada Lett.*, vol. 42, no. 2, p. 55–61, apr 2023. [Online]. Available: https://doi.org/10.1145/3591335.3591340

[4] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 763–779. [Online]. Available: https://doi.org/10.1145/3385412.3386036

[5] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1483–1486. [Online]. Available: https://doi.org/10.1145/3597926.3604919

[6] S. Thy, A. Costea, K. Gopinathan, and I. Sergey, "Adventure of a lifetime: Extract method refactoring for rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622821

[7] W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622841

[8] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 581–592. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678813

[9] J. Corbet, "Next steps for rust in the kernel," Website, 2022, https://lwn.net/Articles/908347/.

[10] T. Claburn, "Microsoft is busy rewriting core windows code in memory-safe rust," Website, 2023, https://www.theregister.com/2023/04/27/microsoft_windows_rust/.

[11] Huggingface, "candle," 2023, https://github.com/huggingface/candle.

[12] L. H. Newman, "The rise of rust, the 'viral' secure programming language that's taking over tech." 2022, https://www.wired.com/story/rust-secure-programming-language-memory-safe/.

[13] Github, "The top programming languages," Website, 2022, https://octoverse.github.com/2022/top-programming-languages.

[14] U. government, "The case for memory safe roadmaps," 2024, https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps.

[15] Y. Zhao, J. Chen, R. Fu, H. Ye, and Z. Wang, "Testing the compiler for a new-born programming language: An industrial case study (experience paper)," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 551–563. [Online]. Available: https://doi.org/10.1145/3597926.3598077

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, jun 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[17] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428264

[18] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 65–76. [Online]. Available: https://doi.org/10.1145/2737924.2737986

[19] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," pp. 216–226, 2014. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[20] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "Slemi: Equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 335–346. [Online]. Available: https://doi.org/10.1145/3377811.3380381

[21] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," pp. 65–76, 2015. [Online]. Available: https://doi.org/10.1145/2737924.2737986

[22] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527317

[23] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 347–361. [Online]. Available: https://doi.org/10.1145/3062341.3062379

[24] R. language, "Rust survey 2018 results," 2018, https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.htmlhttps://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html.

[25] M. Coblenz, A. Porter, V. Das, T. Nallagorla, and M. Hicks, "A Multimodal Study of Challenges Using Rust," 3 2023. [Online]. Available: https://kilthub.cmu.edu/articles/conference_contribution/A_Multimodal_Study_of_Challenges_Using_Rust/22277326

[26] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, "rust-code-analysis: A rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, p. 100635, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711020303484

[27] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li, X. Lian, G. Meng, X. Peng, H. Sun, L. Shi, B. Wang, C. Wang, J. Wang, T. Wang, J. Xuan, X. Xia, Y. Yang, Y. Yang, L. Zhang, Y. Zhou, and L. Zhang, "Deep learning-based software engineering: Progress, challenges, and opportunities," *SCIENCE CHINA Information Sciences*, vol. 68, no. 1, 2025. [Online]. Available: http://www.sciengine.com/publisher/ScienceChinaPress/journal/SCIENCECHINAInformationSciences/68/1/10.1007/s11432-023-4127-5

[28] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "White-box Compiler Fuzzing Empowered by Large Language Models," *ArXiv preprint*, vol. abs/2310.15991, 2023. [Online]. Available: https://arxiv.org/abs/2310.15991

[29] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[30] 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*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[31] C. Yang, Z. Zhao, and L. Zhang, "KernelGPT: Enhanced Kernel Fuzzing via Large Language Models," *ArXiv preprint*, vol. abs/2401.00563, 2024. [Online]. Available: https://arxiv.org/abs/2401.00563

[32] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 126:1–126:13. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[33] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," *ArXiv preprint*, vol. abs/2204.05999, 2022. [Online]. Available: https://arxiv.org/abs/2204.05999

[34] H. Zhong, "Enriching compiler testing with real program from bug report," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556894

[35] M. Sun, Y. Yang, M. Wen, Y. Wang, Y. Zhou, and H. Jin, "Validating smt solvers via skeleton enumeration empowered by historical bug-triggering inputs," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 69–81. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00018

[36] C. Chen, Z. Zhang, H. Tian, S. Yan, and H. Xu, "Oom-guard: Towards improving the ergonomics of rust oom handling via a reservation-based approach," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 733–744. [Online]. Available: https://doi.org/10.1145/3611643.3616303

[37] Y. Zhang, A. Kundu, G. Portokalidis, and J. Xu, "On the dual nature of necessity in use of rust unsafe code," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 2032–2037. [Online]. Available: https://doi.org/10.1145/3611643.3613878

[38] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1269–1281. [Online]. Available: https://doi.org/10.1145/3510003.3510164

[39] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 482–493. [Online]. Available: https://doi.org/10.1109/ASE.2015.65

[40] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[41] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: https://jmlr.org/papers/v21/20-074.html

[42] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. M. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. C. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. García, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Díaz, O. Firat, M. Catasta, J. Wei, K. S. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *J. Mach. Learn. Res.*, vol. 24, pp. 240:1–240:113, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247951931

[43] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. M. V, J. T. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" *Trans. Mach. Learn. Res.*, vol. 2023, 2023. [Online]. Available: https://openreview.net/forum?id=KoFOg41haE

[44] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Conference on Empirical Methods in Natural Language Processing*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258685677

[45] Z. Tang, J. Ge, S. Liu, T. Zhu, T. Xu, L. Huang, and B. Luo, "Domain adaptive code completion via language models and decoupled domain databases," *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 421–433, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261030382

[46] Y. Zhang, B. Kang, B. Hooi, S. Yan, and J. Feng, "Deep long-tailed learning: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 9, p. 10795–10816, sep 2023. [Online]. Available: https://doi.org/10.1109/TPAMI.2023.3268118

[47] https://pypi.org/project/BeautifulSoup/.

[48] Y. Deng, C. S. Xia, C. Yang, S. Dylan Zhang, S. Yang, and L. Zhang, "Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT," *arXiv e-prints*, p. arXiv:2304.02014, Apr. 2023.

[49] rust lang, "rustc-testsuite," 2010, https://github.com/rust-lang/rust/tree/master/tests.

[50] rust lang, "glacier," 2015, https://github.com/rust-lang/glacier.

[51] Z. Dong, Q. Hu, Y. Guo, Z. Zhang, M. Cordy, M. Papadakis, Y. L. Traon, and J. Zhao, "Boosting source code learning with data augmentation: An empirical study," *CoRR*, vol. abs/2303.06808, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.06808

[52] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[53] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:160025533

[54] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[55] J. Jackson, "Where does the time go? rust's problem with slow compiles," 2024, https://thenewstack.io/where-does-the-time-go-rusts-problem-with-slow-compiles/.

[56] A. Wang, "Rustlantis," 2023, https://github.com/cbeuw/rustlantis.

[57] rust lang, "nomicon," 2017, https://github.com/rust-lang/nomicon.

[58] rust lang, "rust by example," 2014, https://github.com/rust-lang/rust-by-example.

[59] rust lang, "rust cookbook," 2017, https://github.com/rust-lang-nursery/rust-cookbook.

[60] Openai, "human eval," 2021, https://github.com/openai/human-eval.

[61] T. K. C. L. at Peking University, "Codeshell," 2023, https://github.com/WisdomShell/codeshell.

[62] OpenAI, :, A. Hurst, A. Lerer, and . J. W. Goucher, "GPT-4o System Card," *arXiv e-prints*, p. arXiv:2410.21276, Oct. 2024.

[63] Z. Ren and H. Xu, "Detect stack overflow bugs in rust via improved fuzzing technique," in *The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023, KSIR Virtual Conference Center, USA, July 1-10, 2023*, S. Chang, Ed. KSI Research Inc., 2023, pp. 175–180. [Online]. Available: https://doi.org/10.18293/SEKE2023-122

[64] A. Calvagna, A. Fornaia, and E. Tramontana, "Assessing the correctness of jvm implementations," in *Proceedings of the 2014 IEEE 23rd International WETICE Conference*, ser. WETICE '14. USA: IEEE Computer Society, 2014, p. 390–395. [Online]. Available: https://doi.org/10.1109/WETICE.2014.33

[65] A. Calvagna and E. Tramontana, "Automated conformance testing of java virtual machines," in *Proceedings of the 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, ser. CISIS '13. USA: IEEE Computer Society, 2013, p. 547–552. [Online]. Available: https://doi.org/10.1109/CISIS.2013.99

[66] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 95–110. [Online]. Available: https://doi.org/10.1145/3062341.3062349

[67] J. Hua and S. Khurshid, "Edsketch: Execution-driven sketching for java," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 162–171. [Online]. Available: https://doi.org/10.1145/3092282.3092285

[68] Q. Gu, "Llm-based code generation method for golang compiler testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 2201–2203. [Online]. Available: https://doi.org/10.1145/3611643.3617850

[69] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 172–184. [Online]. Available: https://doi.org/10.1145/3611643.3616271