FuzzAug: Data Augmentation by Coverage-guided Fuzzing for Neural Test Generation

Anonymous ACL submission

Abstract

Testing is essential to modern software engineering for building reliable software. Given the high costs of manually creating test cases, automated test case generation, particularly methods utilizing large language models, has become increasingly popular. These neural approaches generate semantically meaningful tests that are more maintainable compared with traditional automatic testing methods like fuzzing. However, the diversity and volume of unit tests in current datasets are limited, especially for newer but important languages. In this paper, we present a novel data augmentation technique, *FuzzAug*, that introduces the benefits of fuzzing to large language models by introducing valid testing semantics and providing diverse coverage-guided inputs. Doubling the size of training datasets, FuzzAug improves the performances from the baselines significantly. This technique demonstrates the potential of introducing prior knowledge from dynamic software analysis to improve neural test generation, offering significant enhancements in neural test generation.

1 Introduction

Testing is one of the most important processes in software engineering, ensuring the quality and reliability of large software applications. Unit tests are example-based self-assessment tests written and executed by the developer to demonstrate that the software works correctly as described in the design specification (Runeson, 2006). However, despite its importance, developers do not always contribute new tests due to the difficulty of identifying which code to test, isolating them as fine-grained units, and finding relevant inputs (Daka and Fraser, 2014). Heuristic-based automatic unit test generation (Pacheco and Ernst, 2007; Fraser and Arcuri, 2011) is one solution to these issues, but the resulting tests are unsatisfactory in readability, correctness, and diversity of relevant input-output pairs (Panichella et al., 2020). Other popular automatic randomized testing methods, *e.g.* fuzzing (Serebryany, 2016), often ignores readability and focuses only on generating inputs to find new program behaviors, *i.e.* new coverage or crashes. However, these randomized testing methods only provide the input that triggers the bug with no valid semantics. These reported input seeds are usually not as informative as unit test functions in practice (Goldstein et al., 2024). Therefore, finding semantic meaningful test cases correctly and effectively remains an unsolved problem.

More recently, people have attempted to overcome these issues by leveraging the power of generative language models (Nie et al., 2023; Rao et al., 2024; He et al., 2024). Large language models (LLMs) trained on large code corpora can write meaningful programs given text descriptions (Bai et al., 2023; Rozière et al., 2023; Lozhkov et al., 2024). Therefore, with sufficiently large code and test datasets, we expect that LLMs could generate high-quality unit tests to assist human software engineers.

However, testing functions typically occupy a minor fraction of a software repository, compared with regular functions for software features. Rao et al. (2024) found that in popular Python and Java repositories, test files comprise fewer than 20% of all code files. This deficiency in training data hampers the ability of LLMs to generate practical tests for production environments for two reasons: 1. the imbalance in training data causes the model to miss critical details in the units under test. 2. the insufficient amount of testing code presents a significant challenge in learning the representations of unit tests adequately. Previous work addressed the imbalance issue by aligning code and tests into pairs (Rao et al., 2024; He et al., 2024). However, the second issue remains unsolved, and is further amplified by the trend of switching to newer programming languages for better maintainability and reliability, *e.g.* redesigning software in Rust.

A promising strategy to further enhance the existing state-of-the-art unit test datasets is designing a new specialized data augmentation (DA) method for LLM-based test generation. In computer vision, data augmentation typically involves applying randomized geometric or color *transformations* or injecting *random noise* to images in the training set. However, these methods are unsuitable for programming languages (PLs) due to their formal grammar and strict semantics. Limited research (Yu et al., 2022) on DA for PL is not suitable for test generation, as they do not introduce new test cases that explore the behavior of the program. Unit test functions provide correct setups to invoke the functions under test (focal functions), and test inputs are fed to the focal functions to explore their functionality at run-time. Consequently, a valid data augmentation method for test generation must incorporate semantically meaningful unit test functions, coupled with randomized yet valid testing inputs tailored to the specific functions under test.

To address these challenges, we propose *FuzzAug*. FuzzAug, as depicted in Figure 1, is a direct and effective data augmentation technique utilizing fuzzing data to enhance test generation with LLMs. Fuzzing identifies vulnerabilities in software by randomly generating inputs to trigger new execution paths in software. These inputs capture the program's runtime behavior and thus can enhance the code understanding capabilities of LLMs (Zhao et al., 2023; Huang et al., 2024). For implementing fuzzing data as a form of data augmentation, we perform code transformations on fuzz targets in libFuzzer (Serebryany, 2016) to create new unit test functions. FuzzAug nearly doubles the limited amount of testing code in training datasets and provides a richer diversity of accurate and executable inputs for the focal functions. Training LLM-based test generation models with FuzzAug addresses the aforementioned issues by automatically providing unit test functions with high-quality test inputs. Thus, FuzzAug is a novel approach in training practical LLM-based unit test assistance, enhancing software robustness and maintaining test readability.

To assess the effectiveness of FuzzAug, we conducted experiments with three different state-of-the-art 7B open-source code generation models. Each model was trained on two datasets: on the original UniTSyn (He et al., 2024) dataset and its FuzzAug-augmented counterpart. All three models trained with FuzzAug consistently outperformed their counterparts trained on only UniTSyn, and outperformed the pre-trained/instruction-tuned baseline significantly. They demonstrated significant improvements in generating accurate test cases (assertions) and useful test functions that achieved higher code coverage.

Our contributions. **1**. We introduce FuzzAug, a novel data augmentation method specifically designed for neural test generation LLMs to address the limitations of existing training datasets. **2**. We build and release the Rust version of UniTSyn, aiming at training test generation models for Rust programs. Furthermore, we apply FuzzAug to this dataset and release the resulting augmented dataset, enhancing its utility for advanced model training. **3**. We validate the efficacy of FuzzAug by training generative LLMs on the UniTSyn dataset augmented by it. The notable improvement underscores the necessity and advantages of incorporating fuzzing-augmented testing functions into the training corpus, demonstrating the practical benefits of our approach.

2 Design of FuzzAug

2.1 Challenges

Generating meaningful test functions as training data for neural test generation models is a complex and critical challenge. To introduce high-quality random data for training test generation models, a data augmentation method should satisfy the following requirements: 1. The randomly generated data must be meaningful and valid to the software testing context, *i.e.*, the random data should be able to explore the program's behavior space. 2. The augmentation modification must provide valid testing semantics in the unit test

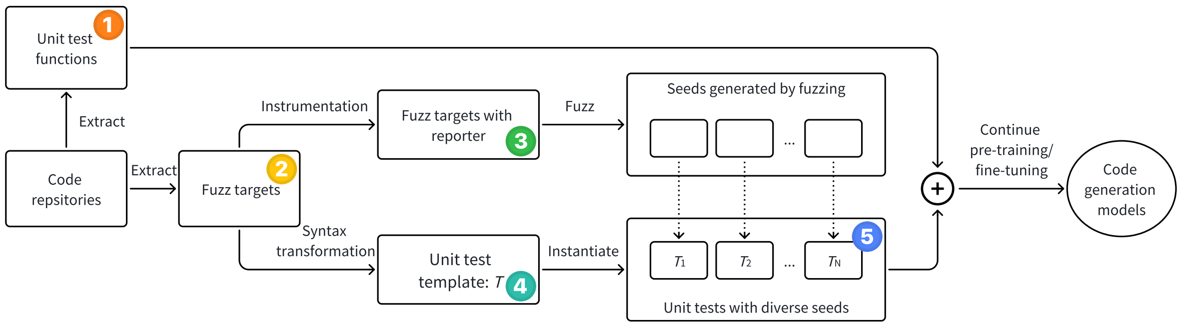

Figure 1: Data Augmentation by fuzzing for neural test generation. To construct the augmented dataset, we first extract unit test functions (Listing 1) and fuzzing targets (Listing 2). We instrument each fuzz target with a reporter (Listing 3) to collect fuzzing seeds. We transform each fuzz target into a unit test template (Listing 4). Finally, we instantiate the templates with valid test inputs to create the augmented training dataset (Listing 5). Please refer to Figure 2 for examples of each step.

functions. As stated by Pacheco and Ernst (2007), unit test functions must correctly parse the random input, set up the state by invoking the focal function, and assert the result of the final call is desired when possible.

Therefore, designing data augmentation to train test generation models involves creating a sophisticated balance. On the one hand, introducing sufficient variability to train the models under diverse conditions is essential to generate high-quantity test cases. On the other hand, maintaining the semantic integrity of augmented test functions is crucial to ensure the validity of training data. This makes the development of FuzzAug not only challenging but also vital for advancing the capabilities of neural test generation with language models.

2.2 Fuzzing for Random Input

The first requirement ensures that the randomly generated data is beneficial to model training. High-quality test cases are expected to reflect the behavior of the programs, which is hard to achieve by data augmentation for natural language data. To improve the model's ability to generate useful test cases, the data augmentation method needs to be aware of the program's structure and behavior.

Fuzzing. Fuzzing is a widely used software testing method that generates inputs randomly to explore unseen program behaviors (Zeller et al., 2019). Coverage-guided fuzzing can be summarized as a four-stage loop consisting of input generation, program execution, behavior monitoring, and input ranking. First, the program is executed with a given input. During execution, the program's dynamic behavior, particularly branch coverage, is monitored to collect coverage information. If a new behavior is observed, the triggering input is saved in a seed queue and prioritized for next round of mutation; otherwise, it is discarded. Finally, the mutator modifies the input for the next cycle to explore new behaviors. Various mutation, behavior monitoring, seed scheduling strategies have been studied to enhance the quality of input seeds during fuzzing (Böhme et al., 2016, 2017; Chen and Chen, 2018; She et al., 2019), and are integrated to the modern fuzzers like LibFuzzer (Serebryany, 2016).

Fuzzers select input seeds by executing the programs, these inputs embed the program's dynamic behavior and are thus able to discover bugs and vulnerabilities in the program. Previous studies (Zhao et al., 2023; Huang et al., 2024) show that fuzzing input-output pairs are helpful for language models to understand programs. Therefore, we argue that random inputs generated by fuzzers are also suitable to contribute to randomized mutation for testing function data augmentation. Thus, this first requirement is satisfied by engaging fuzzing in the data augmentation process.

LibFuzzer (Serebryany, 2016) allows users to define custom fuzz targets to specify the most important functions as entry points for testing. We select libFuzzer for its function-level fuzzing feature to ensure syntax correctness when invoking the corresponding focal function. If we can compile and run the fuzz target

```
#[test]
fn encode_all_bytes_url() {
    let bytes: Vec<u8> = (0..=255).collect();
    assert_eq!(
        "...", // expected result
        &engine::GeneralPurpose::new(&URL_SAFE,
                PAD).encode(bytes)
    );
}
```

Listing (1) Unit test function extracted from repository

```
#![no_main]
#[macro_use] extern crate libfuzzer_sys;
extern crate base64;
use base64::*;
mod utils;
fuzz_target!(|data: &[u8]| {
    let engine = utils::random_engine(data);
    let _ = engine.decode(data);
});
```

Listing (2) Fuzz target extracted from repository

```
fuzz_target!(|data: &[u8]| {
    report(data); // example reporter
    let engine = utils::random_engine(data);
    let _ = engine.decode(data);
});
```

Listing (3) Fuzz target instrumented with reporter

```
#[test]
fn test_template() {
    let data = []; // example template
    let engine = utils::random_engine(data);
    let _ = engine.decode(data); }
```

Listing (4) Test template transformed from fuzz target

```
#[test]
fn test_1() {
    let data = [3,44,12,3,21,2,255,12,4,34,12,4,12,3];  // example recorded test input
    let engine = utils::random_engine(data);
    let _ = engine.decode(data); }
```

Listing (5) Unit test function instantiated from test template with a seed generated by fuzzing

Figure 2: Simplified examples from `base64` (Pierce, 2024) in our collected Rust dataset. Each example listing corresponds to one step in Figure 1. Please refer to Section A.2 for details of unit testing in Rust.

successfully, we are confident that the testing code is valid training data for the language model. Therefore, the validity of FuzzAug is guaranteed. To collect inputs with the program's dynamic behavior from the fuzzing loop, we instrument a `reporter` to each fuzz target as shown in Figure 1. After all the fuzz targets in the project are instrumented, we start the fuzzing loop for each target and save the reported inputs as a randomly generated portion of our data augmentation process.

2.3 Unifying Code Representation

For code generation with causal language modeling, valid and complete training data with appropriate semantics within the tokens is beneficial. Therefore, to avoid any distribution shift between unit test functions and data augmentation, we cannot append inputs generated by fuzzing to training data directly due to the distinct representations between raw fuzzing inputs and meaningful unit test functions. Fuzzers treat all inputs as bytes and apply byte-level random mutations, for example, bit-flip. Previous work on using fuzzing data for code understanding tasks decodes the raw inputs into strings and append the inputs to the program (Zhao et al., 2023) or uses different language modeling loss functions for two kinds of data (Huang et al., 2024). However, these approaches do not apply to generative models, so we need to design a different representation for fuzzing data.

We implement a syntax transformation in the compiler frontend to obtain valid new test functions to keep testing semantics. We compiled these candidates (Listing 2) into Abstract Syntax Trees (ASTs) and extracted the function bodies from each AST using `proc_macro` (David Tolnay and Alex Crichton, 2024) and `syn` (David Tolnay, 2024). Then we rewrite the macro for fuzz targets into valid function definitions with the `#[test]` attribute on top to help test discovery (Listing 4). We call the result of syntax transformation *test template*. We demonstrate a fuzz target and its transformed test template in Figure 1. These test templates are stored for actual data augmentation at a later stage.

2.4 Fuzz Augmentation

To ensure the quality of the augmented data, we employed an input selection algorithm as shown in Algorithm 1. Raw inputs collected from fuzzing have two drawbacks. First, there will be repeated or overlapping inputs collected from fuzzing. Fuzzing applies mutation on inputs that explore new paths in the program. Therefore, consecutive inputs differ only in small parts, which should be avoided.

Second, since the input data are generated

Algorithm 1 Fuzzing as Data Augmentation

function FuzzAug($repo$, N, L, $timeout$)
  ▷ $repo$ = repository to apply FuzzAug
  ▷ N = number of training examples to generate
  ▷ L = maximum input length for collection
  ▷ $timeout$ = maximum allowed fuzzing time
  $dataset_{\text{aug}} \leftarrow []$
  for all $t \in$ GetFuzzTarget($repo$) **do**
    $t' \leftarrow$ ReporterInstrumentation(t)
    $inputs \leftarrow$ Fuzz(t', $timeout$)
    $inputs' \leftarrow$ Filter($\lambda x : \text{len}(x) < L, inputs$)
    $selected \leftarrow$ Sample(N, $inputs'$)
    $templates \leftarrow$ SyntaxTransformation(t)
    $aug \leftarrow$ Instantiate($templates[: N]$, $selected$)
    $dataset_{\text{aug}} \leftarrow dataset_{\text{aug}} + aug$
  return $dataset_{\text{aug}}$

Dataset	# Repo	# Focal	# Pairs	# Tokens
Unit tests	249	14 633	7881	2.5M
Fuzz	179	14 790	6811	2.2M
All	249	29 423	14 692	4.7M

Table 1: Dataset statistics. Unit tests: the base dataset we collected from code repositories using UniTSyn (He et al., 2024). Fuzz: the dataset we transformed from fuzz targets using Algorithm 1, where $N = 40$. Augmented dataset: the combination of unit tests and fuzz.

randomly by libFuzzer (Serebryany, 2016), the token length for those inputs can be excessively long. This behavior happens especially commonly when the input type is a vector or long number (`i64`, `f64`, etc) since the length of the vectors or numbers is not a problem for fuzzing. However, for generative models, the acceptable token length is much smaller, so such long inputs will harm the performance of the model. To overcome the aforementioned issues, we designed our selection algorithm to first shuffle the inputs and then sample the desired inputs within a given length. Our algorithm samples N fuzzing inputs that satisfy the requirements to instantiate the test templates for unique data augmentation (Listing 5).

3 Experimental Setup

3.1 Data Collection

We chose Rust language to conduct this research for three reasons. First, Rust projects are highly structured with `src/`, `tests/`, and `fuzz/` directories on the top level. With the `cargo` package manager, we can build and run the project without solving dependency issues. Second, the Rust compiler has built-in support for unit testing and fuzzing, so collecting unit tests and fuzzing data is straightforward. Third, Rust's syntax for libFuzzer passes a closure to a predefined macro, so we can apply syntax transformation described in Section 2.3 to the fuzz targets. Rust is one of the most popular languages for security-critical software, and yet is new compared to older languages like C/C++, further lighting the necessary for effective data augmentation. We follow UniTSyn (He et al., 2024) to collect the training data from open-source repositories on GitHub.

Unit test collection. Different from previous work training on file-level code-test pairs (Rao et al., 2024), we follow previous work (Nie et al., 2023; He et al., 2024) to collect our training data as function-level code-test pairs since it suits our data augmentation method. We implement the Rust hook for the UniTSyn (He et al., 2024) based on the `#[test]` attribute on top of the Rust unit test functions. To find the call to the focal function, since assertion in Rust is a macro instead of a keyword or function as in UniTSyn, we extend the framework to handle this marco special case. From the downloaded repositories, we found 14 633 calls to the focal functions in the unit tests, and collected 7881 focal-test pairs as training data.

Augmented test collection. We chose LLVM libFuzzer (Serebryany, 2016) to utilize the predefined fuzz targets in the code repositories. For Rust, libFuzzer is supported as `cargo-fuzz`. We instrumented each fuzz target in the repository to report the input fuzzing data. We transform the body of the fuzz target macro to an equivalent unit test template, as described in Figure 1. We fuzzed all targets for one minute following previous work (Zhao et al., 2023; Huang et al., 2024) on fuzzing for code understanding. All fuzzing processes are performed on a server with dual 20-core, 40-thread x86_64 CPUs and 692 GB of RAM. Out of the 249 repositories we downloaded, 179 of them can be compiled successfully for fuzzing. For the main experiments, we set $N = 40$ so that the augmented data is at the same scale as the original unit test dataset, and explore the effects of scaling N later in Section 4.4. We collected in total of 6811 additional code-test pairs generated by FuzzAug. The statistics of the collected

Method	Base Model		
	StarCoder2	CodeQwen1.5	CodeLlama
UniTSyn	UnitCoder	UnitQwen	UnitLlama
FuzzAug	FuzzCoder	FuzzQwen	FuzzLlama

Table 2: Our model selection for evaluation. Base Model: names of the baseline models used for applying the fine-tuning methods.

unit test dataset and data augmentation are summarized in Table 1.

3.2 Baseline Models

We select three baselines to evaluate FuzzAug. StarCoder2 (Lozhkov et al., 2024) is the successor of UniTSyn's base model SantaCoder (Allal et al., 2023). We follow EvalPlus (Liu et al., 2023) to select the best-performing 7B code generation model CodeQwen1.5 (Bai et al., 2023). Finally, we experiment on CodeLlama (Rozière et al., 2023) to compare against its instruction-tuned baseline. The complete model selection and naming are in Table 2. Our training details are in Section A.1.

3.3 Research Questions

To evaluate FuzzAug, we structure our experiments around the following research questions on the quality of generated unit tests:

RQ.1. Can FuzzAug improve the accuracy of generated test cases? Software testing aims to discover hidden bugs in the code. The prerequisite of this aim is to have *accurate* test cases, where the generated input and output to the focal function match with the ground truth. Therefore, accuracy of generated test cases is an essential metric for software testing. Generating accurate test cases requires the model to learn both the semantics and runtime behavior of the focal function, which is challenging for language models (Gu et al., 2024). We follow previous work (Chen et al., 2023a; He et al., 2024) to extract the first 10 generated test cases to examine their standalone correctness. We compile and execute these test cases against the ground truth focal function independently.

RQ.2. Can FuzzAug improve the validity and completeness of generated unit tests? Accurate assertions are essential for unit testing, while completeness and validity are necessary for generated test functions to be practical. A generated test function is *valid* if it can be compiled and executed. On the other hand, a test function is *complete* if it can cover all of the branches of the focal function. Therefore, we follow UniTSyn to use the compile rate of the whole generated unit test functions and branch coverage on the focal functions to check the validity and completeness of the generated unit test functions. We use `grcov` (Marco Castelluccio, 2024) to measure the branch coverage.

RQ.3. Can FuzzAug generalize to other models? Data augmentation is a training-time technique that should improve the performance of all models in the same task.

RQ.4. The effect of further scaling FuzzAug. It is possible to further scale-up FuzzAug, so we explore the effects of hyperparameter N.

3.4 Evaluation Setup

Benchmark dataset. We follow UniTSyn to evaluate the models on HumanEval-X (Zheng et al., 2023), a hand-crafted benchmark for code generation tasks that contains Rust. HumanEval-X has 164 different problems, where each of them is composed of description prompt in natural language, function declaration (header), canonical solution (ground truth implementation), and unit test function. We follow UniTSyn to use the canonical solution as the focal function, and let the model generate the corresponding test function.

Prompts. We follow Chen et al. (2023a) to guide the language models in generating assertions (Listing 6). We use natural language "Check the correctness of `` `function_name` ``" in comments to instruct the model to complete the test function. We guide the generation of assertions by providing the language-specific assert keyword and the incomplete invocation of the focal function. We allow the model to predict at most 1024 new tokens for the synthesized assertions for all models. We set the generation temperature to 1 for all the models to encourage output diversity. We concatenate the import statements, the focal function implementation, the natural language instruction in the comment, and the test header together as the import prompt to the language model.

Post-processing. We avoid overly intricate processing of the generated test functions to

```
fn has_close_elements(numbers: Vec<f32>,
    threshold: f32) -> bool { ... }
// Check the correctness of `
    has_close_elements`
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_has_close_elements() {
        assert_eq!(has_close_elements(
```

Listing 6: Example prompt used for test generation. Import statements are removed for simplicity.

keep our evaluation results faithful. We first count the number of the curly brackets. If the numbers do not match, we check if the last generated line ended with a semicolon to see if the last line is complete. If not, we remove that line. Then we add the missing closing curly brackets to complete the generated test.

4 Evaluation Results

We report our experimental results on the performance of neural test generation in this section. We categorize the models into three groups: pre-trained (PT), instruction-tuned (IT), and fine-tuned (FT) models. PT and IT models are the baselines, while FT models are further trained with UniTSyn and FuzzAug.

4.1 Test Case Correctness

We follow CodeT (Chen et al., 2023a) to guide the language models in generating independent test cases (assertions). Since the assertions are independent, we can parse them and evaluate each one of them individually. We present the evaluation results in Table 3. Notably, CodeQwen1.5 is the strongest model in this assertion compile rate evaluation, where we observe an increase of *+14.38%* over CodeQwen1.5 and *+7.37%* over UnitQwen. For assertion accuracy, We observe a *+10.49%* increase over CodeQwen1.5 and a *+6.16%* increase over UnitQwen.

4.2 Test Validity and Completeness

To evaluate if FuzzAug can help the model generate valid unit test functions, we evaluate the generated unit test functions without extracting the individual assertions. Results for this experiment are shown in Table 4. For whole test function compile rate, FuzzAug also shows stable improvements on all models. On the strongest model, CodeQwen1.5, we observe an increase of *+4.88%* over CodeQwen1.5 and *+12.80%* over UnitQwen.

Model	Type	Assert. CR	Acc
StarCoder2	PT	64.09	31.83
UnitCoder	FT	65.73	32.99
FuzzCoder	FT	**70.98**	**35.50**
CodeLlama	IT	64.57	32.13
UnitLlama	FT	70.79	34.70
FuzzLlama	FT	**75.67**	**37.07**
CodeQwen1.5	PT	66.52	41.71
UnitQwen	FT	73.54	46.04
FuzzQwen	FT	**80.91**	**52.20**

Table 3: Accuracy of tests generated by LLMs. The best results are highlighted in bold. Assert. CR: the compile rate of the individual assertions. Acc: accuracy of individual assertions.

Model	Type	Func. CR	Cov
StarCoder2	PT	45.73	9.88
UnitCoder	FT	48.17	11.92
FuzzCoder	FT	**59.56**	**17.09**
CodeLlama	IT	54.88	15.75
UnitLlama	FT	64.02	16.23
FuzzLlama	FT	**71.95**	**19.52**
CodeQwen	PT	68.29	20.90
UnitQwen	FT	60.37	20.76
FuzzQwen	FT	**73.17**	**24.63**

Table 4: Evaluations of usefulness of generated unit tests. Func. CR: the compile rate of generated unit test functions. Cov: the average branch coverage of generated unit test functions on the focal functions.

FuzzAug also improves the average branch coverage consistently. For CodeQwen1.5, we observe an increase of *+3.73%* over CodeQwen1.5 and *+3.87%* over UnitQwen. Achieving high branch coverage is a hard task for LLMs, as it requires deep understanding and reasoning ability over the function's control flow. For reference, even with known overfitting issues (Jain et al., 2024), GPT-4 can only achieve an average branch coverage of 47.94%.

4.3 Generalizability of FuzzAug

Useful data augmentation methods should work on different models. We fine-tune three different models with FuzzAug and evaluate their performance, where all models trained with FuzzAug show improvements over the baseline pre-trained models and UniTSyn.

4.4 Scaling FuzzAug

We explore the effects of scaling FuzzAug to construct larger training datasets. To assess the impact of varying amounts of fuzzing inputs, we train models with $N = 40, 60, 80, 100$ fuzzing samples for this experiment.

As shown in Appendix Figure 5, the impact of scaling FuzzAug is not consistent across models. In particular, for the stronger base model CodeQwen1.5, increasing N does not lead to significant changes. Conversely, for weaker base models, scaling N improves both assertion accuracy and compile rate. When evaluating the test function compile rate, both FuzzLlama and FuzzCoder exhibit a positive correlation with increasing N. Additionally, FuzzLlama's accuracy improves with larger N, while other metrics show no clear trend.

The results suggest that dataset size alone is not the primary factor influencing model performance. Instead, the quality of data augmentation, driven by the test semantics of the fuzz targets and coverage-guided inputs, plays a more crucial role. Therefore, we recommend selecting N at a scale comparable to the original training dataset, which should be enough.

5 Related Work

5.1 Fuzzing

Fuzz testing (Zeller et al., 2019), or fuzzing, is a popular execution-based dynamic testing technique with randomized inputs in various software domains (Rong et al., 2020; Chen et al., 2023b; Rong et al.). Fuzzing aims to generate a set of inputs based on the provided set of seeds to achieve high code coverage. The fuzzer uses behavior monitoring to find inputs with high branch coverage and favors those inputs for future input generation (Chen and Chen, 2018; She et al., 2019; Rong et al., 2024). LibFuzzer (Serebryany, 2016) is integrated into the LLVM compiler infrastructure (Lattner and Adve, 2004), and can also be used in other mainstream languages (Intelligence, 2024; Google).

Fuzzing for machine learning. Inputs generated by coverage-guided fuzzing can benefit language models in understanding programs, as they contain information about the program's dynamic behavior (Zhao et al., 2023; Huang et al., 2024). Fuzzing was also adopted as a data augmentation tool to improve the robustness of neural networks (Gao et al., 2020).

5.2 Test Generation via LLMs

Using especially LLMs to generate test cases is a new trend in automatic software testing. This method is referred to as *neural test generation*. The direct approach toward neural test generation is to instruct pre-trained code generation LLMs (Rozière et al., 2023; Lozhkov et al., 2024), or foundation models (Achiam et al., 2023; Schäfer et al., 2024; Tang et al., 2024). The other approach is to train test-specific models that are specialized in generating test cases or test functions (Watson et al., 2020; Tufano et al., 2021; Dinella et al., 2022; Alagarsamy et al., 2023). The more recent work (Nie et al., 2023; Rao et al., 2024; He et al., 2024) proposed to train the test generation model on *aligned* data that includes the correspondence between the unit test and the function under test (focal).

6 Conclusion

We developed FuzzAug, a data augmentation method for unit test function generation. FuzzAug combines the advantages of coverage-guided fuzzing and generative large language models to generate tests that are not only semantically meaningful but also strategically comprehensive. We applied FuzzAug to fine-tune three state-of-the-art 7B open-source code generation models to demonstrate the effectiveness of FuzzAug. We collect our experimental dataset on Rust crates that have pre-defined fuzzers as a Rust extension to UniTSyn. Our method can be generalized to all languages that OSS-Fuzz supports with slight modifications. Our results show the effectiveness of employing dynamic program analysis to generate high-quality inputs to augment the code corpus in training language models. We believe FuzzAug can spur the development of unit test generation by large language models and contribute to the field of AI for software engineering and testing. Our code and artifacts are available anonymous (link), and will be publicly available after publication.

Limitations and Future Work

In this section, we discuss the potential concerns of our design and limitations. We structure each concern we foresaw and the discussion of them as subsections.

Applying to Different Languages

On the high level, fuzzing is a programming language agnostic testing approach. LibFuzzer is part of the LLVM (Lattner and Adve, 2004), which supports any language that can be compiled to LLVM intermediate representation. Currently, OSS-Fuzz (Serebryany, 2017) supports C/C++, Rust, Go, Python, and Java/JVM code, and other LLVM-supported languages.

Syntax transformation from fuzz targets to unit test templates differs for languages. However, the general framework can be defined in a language-agnostic manner. UniTSyn (He et al., 2024) is a multi-lingual framework to collect unit test functions based on tree-sitter, which can be extended to syntax transformation.

We choose Rust (Matsakis and Klock, 2014) to conduct our study to take advantage of its powerful build tool `cargo`[1]. `Cargo-fuzz`[2] allows software developers to define their fuzz targets inside the repository, making it easier for us to execute the fuzz targets and apply our data augmentation. In principle, our method can be generalized to all libFuzzer-supported languages, and their corresponding fuzz targets can be found in OSS-Fuzz (Serebryany, 2017). To use FuzzAug in other languages, one could locate the fuzz targets in OSS-Fuzz. The current limitation of FuzzAug is that only languages supported by OSS-Fuzz can be used.

Applying to Different Datasets

We followed TeCo (Nie et al., 2023) and UniTSyn (He et al., 2024) to construct our dataset on function-level code-test pairs. File-level pairing approach used in CAT-LM (Rao et al., 2024) offers additional benefits by providing more relevant context, which is particularly useful in less modular, tightly coupling, complex software systems. FuzzAug is applicable to both function-level and file-level data to accommodate various types of datasets effectively. LibFuzzer maintains separate fuzz targets in different files. After syntax transformation and fuzz data collection, FuzzAug can insert augmented unit test functions into their original files and adopt CAT-LM's pairing strategy. This versatility enhances FuzzAug's ability to augment and improve various types of unit test datasets effectively. However, FuzzAug requires the software repositories to compile successfully.

Evaluation on Real-World Projects

In our experiments, we follow UniTSyn to assess the validity and completeness of generated unit test functions using HumanEval-X (Zheng et al., 2023). We did not use real-world Rust projects due to a few challenges. First, as discussed in UniTSyn, it is hard to eliminate data leakage when evaluating on open-source projects. He et al. (2024) conducted a detailed analysis of the data leakage issue, and conclude that user their dataset construction method, there will be no data leakage on HumanEval-X in the training process.

Second, we want to minimize the negative impacts of incorrect project setup. Generating unit tests in large open-source software (OSS) requires special setups for each project. These setups for defect testing are hard to construct and require human domain knowledge (Zhu and Rubio-González, 2023). Therefore, choosing to evaluate test generation on OSS introduces additional bias in the results, which is another thing we want to eliminate.

Finally, a hand-crafted and expert-verified benchmark like HumanEval-X offers an oracle implementation of the focal functions. If we use real-world projects to evaluate LLM-based unit test generation and an assertion failed, we have no directly way to distinguish whether the generated unit test is incorrect or there is an actual defect. Previous work (Pacheco and Ernst, 2007) in automated unit test generation uses very simple assertions as oracles, such as `assert o.equals(o)`, aimed at finding bugs in codebases. Our goal is to evaluate the completeness and correctness of the generated unit test functions, so we need a benchmark that can provide the oracle implementation of the focal functions. One interesting future work direction is to construct a ground-truth benchmark on selected real-world projects for neural test generation, where all the bugs are known and the oracle implementation is available. Examples

[1] `https://doc.rust-lang.org/cargo/`
[2] `https://github.com/rust-fuzz/cargo-fuzz`

in this direction include BugSwarm (Tomassi et al., 2019) and Magma (Hazimeh et al., 2020).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3test: Assertion-augmented automated test case generation. *arXiv preprint arXiv:2302.10352*.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. Santacoder: don't reach for the stars! *Preprint*, arXiv:2301.03988.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA. Association for Computing Machinery.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA. Association for Computing Machinery.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725.

Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023b. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 1600–1614, New York, NY, USA. Association for Computing Machinery.

Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, page 201–211, USA. IEEE Computer Society.

David Tolnay. 2024. syn: Parser for Rust source code.

David Tolnay and Alex Crichton. 2024. procmacro2: A substitute implementation of the compiler's 'proc_macro' API to decouple token-based libraries from the procedural macro use case.

Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2130–2141, New York, NY, USA. Association for Computing Machinery.

Gordon Fraser and Andrea Arcuri. 2011. 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*, ESEC/FSE '11, page 416–419, New York, NY, USA. Association for Computing Machinery.

Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1147–1158, New York, NY, USA. Association for Computing Machinery.

Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Google. Atheris: A coverage-guided, native python fuzzer. https://github.com/google/atheris.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3).

Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. Unitsyn: A large-scale dataset capable of enhancing the prowess of large language models for program testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, Vienna, Austria.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Jiabo Huang, Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2024. Code representation pre-training with complements from program executions. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track (EMNLP)*, pages 267–278, Miami, Florida, US. Association for Computational Linguistics.

Code Intelligence. 2024. jazzer: About coverage-guided, in-process fuzzing for the jvm. https://github.com/CodeIntelligenceTesting/jazzer.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Marco Castelluccio. 2024. grcov: Rust tool to collect and aggregate code coverage data for multiple source files.

Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104.

Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2111–2123. IEEE Press.

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA. Association for Computing Machinery.

Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J

Hellendoorn. 2020. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*, pages 523–533. IEEE.

Marshall Pierce. 2024. base64: encodes and decodes base64 as bytes or utf8.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2024. Cat-lm training language models on aligned code and tests. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 409–420. IEEE Press.

Yuyang Rong, Peng Chen, and Hao Chen. 2020. Integrity: Finding integer errors by targeted fuzzing. In *Security and Privacy in Communication Networks: 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part I 16*, pages 360–380. Springer.

Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. Irfuzzer: Specialized fuzzing for llvm backend code generation.

Yuyang Rong, Chibin Zhang, Jianzhong Liu, and Hao Chen. 2024. Valkyrie: Improving fuzzing performance through deterministic techniques. *J. Syst. Softw.*, 209(C).

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Per Runeson. 2006. A survey of unit testing practices. *IEEE software*, 23(4):22–29.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE.

Kostya Serebryany. 2017. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC. USENIX Association.

Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817.

Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Trans. Softw. Eng.*, 50(6):1340–1359.

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 339–349. IEEE Press.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit test case generation with transformers and focal context. *Preprint*, arXiv:2009.05617.

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM.

Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The fuzzing book.

Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2023. Understanding programs by exploiting (fuzzing) test cases. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 10667–10679, Toronto, Canada. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

Hao-Nan Zhu and Cindy Rubio-González. 2023. On the reproducibility of software defect datasets. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pages 2324–2335. IEEE Press.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427.

A Appendix

A.1 Training Details

We follow the previous work (Radford et al., 2019; He et al., 2024) to use an autoregressive signal for continual training of the pre-trained base model. We follow UniTSyn for the basic training configuration. Specifically, each training example is the concatenation of the focal function and the unit test function, joined by a `\n` new line symbol. Since most of the training data is around 250 tokens (see Figure 3), we set the maximum sequence length to 512 for the tokenizer. We use a batch size of 128, with gradient accumulation at every 32 steps. We use a $5e^{-5}$ learning rate for our training, with cosine annealing learning rate decay for each batch (Loshchilov and Hutter, 2016). Following Kirkpatrick et al., we use 0.05 weight decay to make the trained model robust to catastrophic forgetting. We apply LoRA (Hu et al., 2022) to the model with the rank $r = 16$, $\alpha = 16$, and 0.05 dropout. We train all the models, except StarCoder2, for 100 steps (approximately eight epochs) on four NVIDIA H100-80GB GPUs. StarCoder2 is trained for 200 steps due to its slower convergence rate and poor performance.

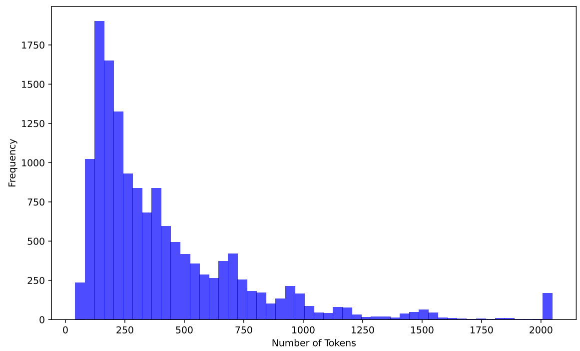

Figure 3: Token distribution of the dataset.

A.2 Testing in Practice

Unit testing is a software testing technique that focuses on assessing the correctness of basic software units (Zhu et al., 1997). In classical setups, unit tests contain three major stages: arrange, act, and assert (Khorikov, 2020). The arrange stage sets up the input data in the correct format, the act stage invokes the code under test, and the assert stage checks the output of the code. If passed, these unit tests can be used as regression tests to ensure the future correctness and security of the software (Pacheco and Ernst, 2007). Unit tests in software repositories are usually structured as *test functions*, each encapsulating the semantics of the aforementioned three components. Unit test functions can be identified using language-specific hooks (He et al., 2024).

Unit Testing in Rust. Unit testing in Rust is no different from that in other programming languages. Rust provides a built-in test framework that allows developers to specify unit test functions using the `#[test]` or `#[cfg(test)]` attribute. The `rustc` compiler can automatically identify these test functions at compile time and includes them only in the test build. Rust offers assertions through the `assert!` macro, with variants such as `assert_eq!` and `assert_ne!` for checking equality and inequality, respectively. These assertion macros are used to verify the expected behavior of the code when the tests are executed. An example of a Rust unit test function is shown in Listing 1, illustrating a simple arrangement on the first line, followed by the action and assertion within the `assert_eq!` macro on the next line.

Fuzzing in Rust. The `cargo-fuzz` tool provides fuzzing functionality for Rust using LibFuzzer (Serebryany, 2016). However, instead of being defined as a test function, a fuzz target is specified using the `fuzz_target!` macro, which takes a closure function as an argument. The closure function provides the appropriate testing semantics. Unlike unit test functions, where programmers hardcode test inputs during the arrange stage, fuzz targets supply randomized input data of type `&[u8]` (a slice of 8-bit unsigned integers) to the closure function. The closure function is then responsible for correctly parsing the input into the appropriate format for the arrange stage. After that, the closure function follows the same semantics as a unit test function: the act stage invokes the code under test, and the assert stage verifies its output. As shown in the example in Listing 2, the closure function performs the arrange stage on line 7. This key design of fuzz targets enables syntax transformation to convert a fuzz target into a unit test function, as described in Section 2.3.

A.3 Additional Results

Model	Type	Assert. CR	Acc
GPT-4	API	95.53	75.04

Table 5: Accuracy of tests generated by LLMs. The best results are highlighted in bold. Assert. CR: the compile rate of the individual assertions. Acc: accuracy of individual assertions.

Model	Type	Func. CR	Cov
GPT-4	API	93.90	47.94

Table 6: Evaluations of usefulness of generated unit tests. Func. CR: the compile rate of generated unit test functions. Cov: the average branch coverage of generated unit test functions on the focal functions.

A.4 Additional Figures

Algorithm 2 Fuzzing as Data Augmentation

function ReporterInstrumentation($fuzz\_target$)
    $AST \leftarrow$ Parse($fuzz\_target$)
    $entry \leftarrow$ GetBegin(AST) ▷ Pointer to the entry point
    $data \leftarrow$ GetParameters(AST)[0]
    $AST' \leftarrow$ AddInstruction(AST, $entry*$, Report($data$)) ▷ Add reporter the entry of AST
    $fuzz\_target' \leftarrow$ Dump(AST')
    return $fuzz\_target'$

function SyntaxTransformation($fuzz\_target$)
    $AST* \leftarrow$ Parse($fuzz\_target$)
    $body \leftarrow$ ExtractBodyNode($AST*$)
    $test\_header \leftarrow$ ... ▷ Language-specific header
    $data\_template \leftarrow$ ... ▷ Declaring data variable
    $test\_ending \leftarrow$ ... ▷ Closing this test definition
    return $test\_header$ + $data\_template$ + $body$ + $test\_ending$

function FuzzAug($repo$, N, L, $timeout$)
    ▷ $repo$ = repository to apply FuzzAug
    ▷ N = number of training examples to generate
    ▷ L = maximum input length for collection
    ▷ $timeout$ = maximum allowed fuzzing time
    $dataset_{\text{aug}} \leftarrow$ []
    for all $t \in$ GetFuzzTarget($repo$) **do**
        $t' \leftarrow$ ReporterInstrumentation(t)
        $inputs \leftarrow$ Fuzz(t', $timeout$) ▷ Collect raw fuzzing inputs
        $inputs' \leftarrow$ Filter(λx : len(x) $< L$, $inputs$)
        $selected \leftarrow$ Sample(N, $inputs'$)
        $templates \leftarrow$ Take(N, SyntaxTransformation(t))
        $dataset_{\text{aug}} \leftarrow dataset_{\text{aug}}$ + Instantiate($templates$, $selected$)
    return $dataset_{\text{aug}}$

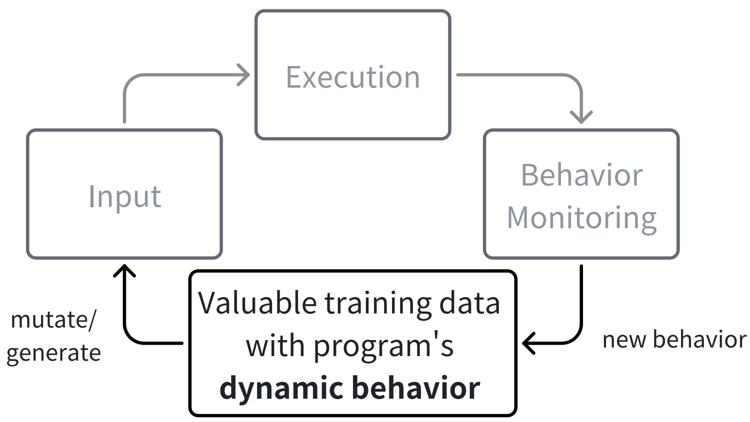

Figure 4: Fuzzing loop for dynamic program testing. This loop shows the process of the collection of randomized generated data for augmentation.

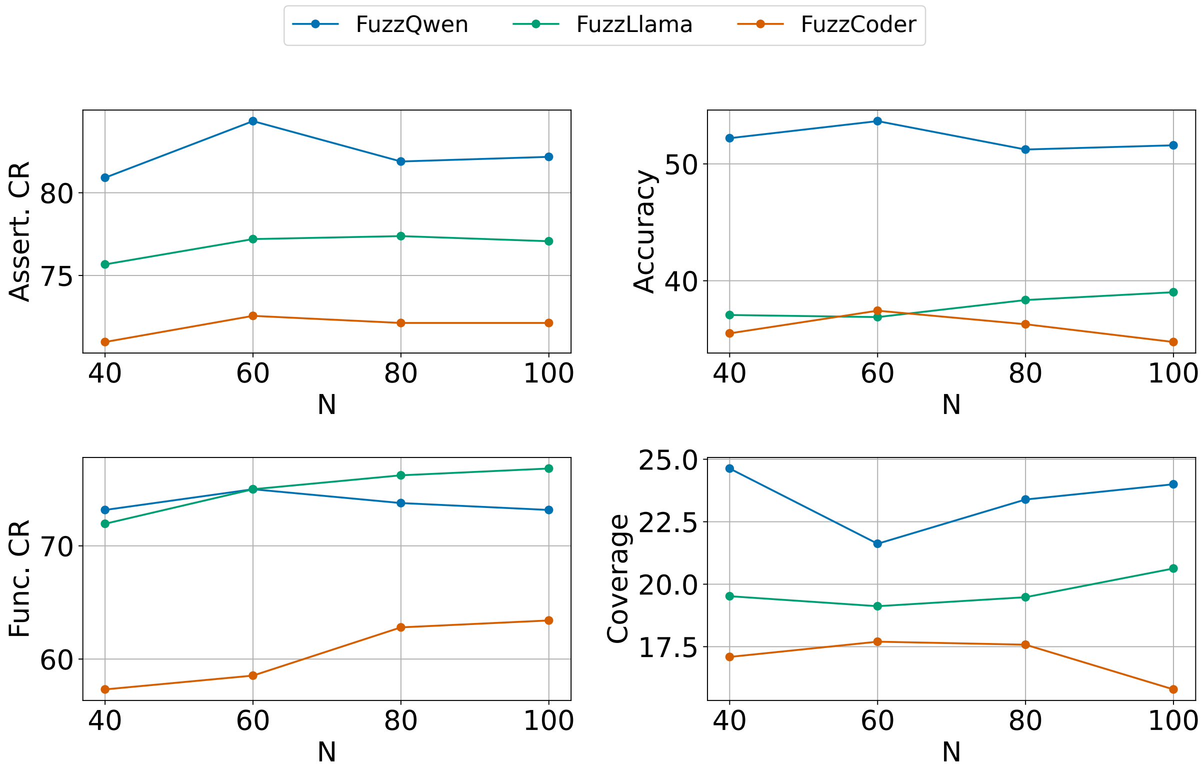

Figure 5: The impact of scaling the number of sampled fuzzing inputs on test generation performance.