FuzzAug: Data Augmentation by Fuzzing for Neural Test Generation

YIFENG HE, University of California, Davis, USA
JICHENG WANG, University of California, Davis, USA
YUYANG RONG, University of California, Davis, USA
HAO CHEN, University of California, Davis, USA

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. In this paper, we introduce a novel data augmentation technique, *FuzzAug*, that introduces the benefits of fuzzing to large language models to preserve valid program semantics and provide diverse inputs. This enhances the model's ability to embed correct inputs that can explore more branches of the function under test. Our evaluations show that models trained with dataset augmented by FuzzAug increases assertion accuracy by 5%, improve compilation rate by more than 10%, and generate unit test functions with 5% more branch coverage. This technique demonstrates the potential of using dynamic software testing to improve neural test generation, offering significant enhancements in neural test generation.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Natural language processing*; *Learning latent representations*; • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: Software Testing, Data Augmentation, Fuzzing, Large Language Models, Unit Testing, Test Generation

1 INTRODUCTION

Testing is one of the 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 [58]. However, despite its importance, developers do not always contribute new tests due to the difficulty of identifying which code to test, isolating them as units, and finding relevant inputs [11]. Heuristic-based automatic unit test generation [17] is one solution to these issues, but the resulting tests are unsatisfactory in readability and correctness of relevant input-output pairs [49]. Other popular automatic randomized testing methods include fuzz testing [16, 62], which ignores readability and focuses on generating inputs to find new program behaviors via coverage monitoring, and property-based testing [10], which lets developers write formal specifications and only generate random inputs. 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 [21]. Therefore, finding semantic meaningful test cases correctly and effectively is still a remaining problem.

More recently, people have attempted to overcome these issues by leveraging the power of machine-learning models [25, 46, 52]. Large language models (LLMs) trained on large code corpora can write meaningful programs given text descriptions [6, 42, 57, 68]. Therefore, with sufficiently large code and test datasets, we expect that LLMs could generate high-quality unit tests to assist

Authors' Contact Information: Yifeng He, yfhe@ucdavis.edu, University of California, Davis, USA; Jicheng Wang, jicwang@ucdavis.edu, University of California, Davis, USA; Yuyang Rong, PeterRong96@gmail.com, University of California, Davis, USA; Hao Chen, chen@ucdavis.edu, University of California, Davis, USA.

human software engineers. However, unit testing functions typically occupy a minor fraction of a software repository, compared with regular functions for software features. Rao et al. found that in popular Python and Java repositories, test files comprise less than 20% of all code files. This deficiency in training data hampers the ability of LLMs to generate practical tests for production environments due to two primary reasons: first, the imbalance in training data causes the model to miss critical details in the units under test; secondly, the insufficient amount of testing code presents a significant challenge in learning the representations of unit tests adequately. Code-test alignment is an effective approach to solve the first issue, where CAT-LM [52] introduced file-level alignment and UniTSyn [25] improved the alignment at the more fine-grained function level, referred to as function-level code-test pairs. UniTSyn also tackles the second challenge by designing an extendable test collection framework to gather testing code from repositories in various languages to enhance the diversity and quantity of test cases. Although this approach can build large-scale datasets for test generation, the size of unit test functions in the dataset is still limited since they have to be created by human developers.

A more promising strategy to enhance the existing state-of-the-art unit test datasets is designing new specialized data augmentation (DA) methods [53]. In computer vision, data augmentation typically involves applying randomized geometric or color *transformations* or injecting *random noise* to images in the training set. In contrast, these methods face challenges in language processing [65]. Data augmentation for programming languages (PLs) is even more challenging than for natural languages. Techniques for augmenting natural language data, such as deleting, misspelling, or randomly replacing words, are unsuitable for tasks in program understanding and generation, including test generation. This is due to the fact that PLs are defined in formal grammars with strict semantics. 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 PL 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 is a powerful dynamic program analysis method that identifies vulnerabilities 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 [27, 80]. For implementing fuzzing data as a form of data augmentation, we perform code transformations on fuzz targets in libFuzzer [62] to create new unit test functions. FuzzAug significantly increases 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 open-source code generation models: StarCoder2 [42], CodeLlama [57], and CodeQwen [6]. Each model was trained on two datasets: on the original UniTSyn [25] 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 complete test functions that achieved higher code coverage. These findings underscore FuzzAug's potential to enhance the quality and effectiveness of unit test generation, effectively addressing the limitations of existing unit test datasets.

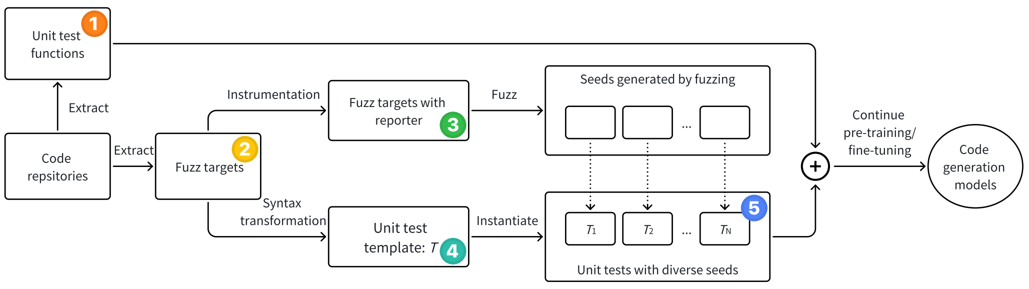


Fig. 1. Data Augmentation by fuzzing for neural test generation. To construct the augmented dataset, we first extract unit test functions (Listing 1) and fuzzing targets (Listing 2). We instrument each fuzz target with a reporter (Listing 3) to collect fuzzing seeds. We transform each fuzz target into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

In summary, our contributions are as follows:

(1) We introduce FuzzAug, a novel data augmentation method specifically designed for neural test generation datasets. FuzzAug addresses the critical issue of insufficient testing code in repositories and the need for precise and diverse inputs to invoke the focal function. To the best of our knowledge, FuzzAug is the first data augmentation method for neural test generation.
(2) We build and release the Rust version of UniTSyn comprising functional-level code-test pairs, aimed 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 autoregressive language models 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. This means that the data should not merely be random or arbitrary. For example in data augmentation for image classification, randomly rotating an image of a cat or injecting random noise into it does not change the fact that it is a cat. Similarly, although there will be no label, data augmentation for test generations requires the randomly generated data to be meaningful for testing the program. For example, when testing sorting algorithms, rather than using short or already-sorted data, meaningful inputs consist of long and diverse sequences that explore the deeper branches of the algorithm. The augmented data should help the model learn to handle diverse and complex testing situations, thus improving its ability to generate useful and effective test cases.

(2) The augmentation modification must keep the syntax and semantic integrity of test functions. Since we are training a language model to generate code, the tokens in training data must follow the exact syntax of the language for the test function to be valid. Furthermore, we require this data augmentation method to generate code with testing semantics. This is because regular functioning code and testing code have purposes, and thus have different distributions of embedding. We want the model to generate better test cases and unit test functions, so it is necessary to overcome this challenge to keep the semantic integrity of test functions.

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 syntactic and 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. In other words, in order 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 is a widely used software testing method, which can generate or mutate inputs randomly and keep the inputs that explore more program paths by behavior monitor [16]. 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. Next, the program's behavior, particularly branch coverage, is monitored to detect any new behavior. If a new behavior is observed, the input is saved as a seed and prioritized for mutation; otherwise, it is discarded. Finally, the mutator modifies the input for the next cycle to explore new behaviors. Since 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 [27, 80] 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.

Fuzzing can be conducted on different levels of the software. AFL++ [16] feeds inputs to whole programs and tests the program entirely. LibFuzzer [62], on the other hand, 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, which ensures syntax correctness when invoking the corresponding focal function. If we can compile and run the fuzz target successfully, we are confident that the testing code is valid training data for the language model. Therefore, the validity of FuzzAug is guaranteed.

Furthermore, fuzzing is a programming language agnostic testing approach. LibFuzzer is part of the LLVM compiler infrastructure [37], which supports any language that can be compiled to LLVM intermediate representation. We choose Rust [45] to conduct our study to take advantage of its powerful build tool `cargo` [1]. `Cargo-fuzz` [69] 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

[1]https://doc.rust-lang.org/cargo/

```rust
#[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

```rust
#![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

```rust
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

```rust
#[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

```rust
#[test]
fn test_1() {
    let data = [3,44,12,3,21,2,255,12,4,34,12,4,12,3];
    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

Fig. 2. Simplified examples extracted from project base64 [44] in our collected Rust dataset. Each example listing corresponds to one step in Figure 1.

augmentation. In principle, our method can be generalized to all libFuzzer-supported languages, and their corresponding fuzz targets can be found in OSS-Fuzz [63].

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. We use `reporter` to denote the reporting process in the figure for simplicity. 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 Transformation for Code Representation

For code generation with causal language modeling (CLM), valid and complete training data with appropriate semantics within the tokens is beneficial. Therefore, we cannot append fuzzing data 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 [80] or uses different language modeling loss functions for two kinds of data [27]. However, these approaches are not applicable to generative models, so we need to design a different representation for fuzzing data to train test generation models with CLM.

We implement the syntax transformation in the compiler frontend for the fuzz targets to obtain valid new test functions to provide testing semantics for fuzzing data. We compiled these candidates into Abstract Syntax Trees (ASTs) and extracted the function bodies from each AST

Algorithm 1 Fuzzing as Data Augmentation

```
1:  function ReporterInstrumentation($fuzz\_target$)
2:      $AST \leftarrow$ Parse($fuzz\_target$)
3:      $entry \leftarrow$ GetBegin($AST$)                                   ▷ Pointer to the entry point
4:      $data \leftarrow$ GetParameters($AST$)[0]
5:      $AST' \leftarrow$ AddInstruction($AST$, $entry*$, Report($data$))   ▷ Add reporter the entry of AST
6:      $fuzz\_target' \leftarrow$ Dump($AST'$)
7:      return $fuzz\_target'$

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

15: function FuzzAug($repo$, $N$, $L$, $timeout$)
16:                                                   ▷ $repo$ = repository to apply FuzzAug
17:                                                   ▷ $N$ = number of training examples to generate
18:                                                   ▷ $L$ = maximum input length for collection
19:                                                   ▷ $timeout$ = maximum allowed fuzzing time
20:     $dataset_{aug} \leftarrow []$
21:     for all $t \in$ GetFuzzTarget($repo$) do
22:         $t' \leftarrow$ ReporterInstrumentation($t$)
23:         $inputs \leftarrow$ Fuzz($t'$, $timeout$)                       ▷ Collect raw fuzzing inputs
24:         $inputs' \leftarrow$ Filter($\lambda x$ : len($x$) $< L$, Shuffle($inputs$))
25:         $selected \leftarrow$ Take($N$, $inputs'$)
26:         $templates \leftarrow$ Instaniate($N$, SyntaxTransformation($t$))
27:         $dataset_{aug} \leftarrow dataset_{aug}$ + Fill($templates$, $selected$)
28:     return $dataset_{aug}$
```

using `proc_macro` [13] and `syn` [12]. Then we rewrite the macro for fuzz targets into valid function definitions with the `#[test]` attribute on top to help test discovery. 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.

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 [25] designed a multi-lingual framework to collect unit test functions based on tree-sitter [1], and we believe it can be extended to syntax transformation. If further research finds a native compiler as a more desirable tool, an approach similar to ours can also be applied using `ast`[2] module for Python datasets. We discuss the application of FuzzAug to other languages in detail in Section 6.1.

Table 1. Dataset statistics. Unit tests: the base dataset we collected from code repositories using UniTSyn [25]. This dataset is used to train UnitCoder, UnitQwen, and UnitLlama in Section 3. Fuzz(100): the dataset we transformed from fuzz targets using Algorithm 1, where $N = 100$. Augmented dataset: the combination of unit tests and fuzz(100). This dataset is used to train FuzzCoder, FuzzQwen, and FuzzLlama in Section 3.

Type of Test Dataset	# Repo w/ Tests	# Focal Calls	# Focal-Test Pairs	# Tokens
Unit tests	249	12 452	6481	6.2M
Fuzz(100)	217	52 127	20 504	96.4M
Augmented dataset	249	64 579	26 985	102.6M

2.4 Fuzz Augmentation

Since fuzzing is high-throughput, we record an exceeding amount of inputs during the process. 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. These characteristics of fuzzing data will cause the trained model to generate repeated test cases. Second, since the input data are generated randomly by libFuzzer [62], 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, where inputs are stored in random access memory or disks. 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.

As described in Algorithm 1, our algorithm selected N fuzzing inputs that satisfy the requirements. Then for each transformed test template, we duplicate it for N copies. Finally, we instantiate each copy of the test template with valid selected fuzzing inputs to generate valid new unit test functions with diverse inputs as data augmentation. Thus, for every selected fuzz input, FuzzAug generates a unique unit test function.

3 EXPERIMENTAL SETUP

To evaluate FuzzAug, we build two test generation models **UnitCoder** and **FuzzCoder** based on StarCoder2 [42]. UnitCoder is trained on the Rust version of UniTSyn [25] containing focal functions and unit test functions, and FuzzCoder is trained on the same dataset but augmented by FuzzAug. UnitCoder serves as a baseline to demonstrate the effects of FuzzAug. We also apply the same setting to train other models as mentioned in Table 2. To explore the quality of test cases generated by the language models, we adopt the popular HumanEval-X [81] benchmark. In this section, we explain the details of our 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 [32] and fuzzing [69], so collecting unit tests and fuzzing data is straightforward. Third, Rust's syntax for libFuzzer passes a closure to a predefined macro, so

[2] https://docs.python.org/3/library/ast.html

we can apply syntax transformation described in Section 2.3 to the fuzz targets. This eases the problem that the model needs to learn two different representations for the fuzzing seeds and unit test functions.

Repository and Package Mining. We follow previous work [25, 28, 52] to collect open-source repositories from GitHub. Specifically, we follow UniTSyn [25] to select Rust repositories based on stars and the date of the last commit. We require the repositories to be popular (with more than ten stars) and under active development (with new commits after January 1st, 2020). We also de-duplicate the dataset by filtering out repositories that are archived, forked, or mirrored. In addition to the requirements on the project's quality as described above, we also require the repository to contain pre-defined unit tests and fuzzers in order to apply FuzzAug. For this purpose, we check the presence of `tests/` and `fuzz/` directories in the root of the project. In total, we find 249 suitable repositories to build our training dataset.

Unit Test Collection. Different from previous work training on file-level code-test pairs [52], we follow [25, 46] 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 [25] 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 12 452 calls to the focal functions in the unit tests, and collected 6481 focal-test pairs as training data.

Augmented Test Collection. We chose LLVM libFuzzer [62] to utilize the pre-defined fuzz targets in the code repositories. For Rust, libFuzzer is supported as `cargo-fuzz` mentioned in the previous sections. We instrumented each fuzz target in the repository to report the input fuzzing data. We used the body of the fuzz target macro to generate an equivalent unit test template, as depicted in Figure 1. We fuzzed all targets for 1 minute following previous work [80] on fuzzing for code understanding. The fuzzing process is 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, 217 of them can be compiled successfully for fuzzing. We collected in total of 20 504 additional code-test pairs generated by FuzzAug. We summarize the statistics of the dataset in Table 1.

3.2 Model Training

Table 2. Our model selection for evaluation. Baseline: names of the pre-trained or instruction-tuned baseline models. UniTSyn: corresponding models fine-tuned using the UniTSyn dataset. FuzzAug: corresponding models fine-tuned using UniTSyn augmented by FuzzAug.

Base Model / Method	StarCoder2 [42]	CodeQwen1.5 [6]	CodeLlama [57]
UniTSyn [25]	UnitCoder	UnitQwen	UnitLlama
FuzzAug	FuzzCoder	FuzzQwen	FuzzLlama

We select StarCoder2 [42] to be our base model. StarCoder2, the successor of UniTSyn's [25] base model SantaCoder [5], is a state-of-the-art open-source code generation model with 7B parameters, which is powerful enough to generate valid code and efficient enough to evaluate the augmented data. We follow the previous work [25, 51, 57] 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. We set the maximum sequence length to 512 for each training example with a per-device batch size of 64. We use a $5e^{-5}$ learning rate for our training, with 50 warm-up steps and cosine annealing learning rate decay for each batch [41]. Following Kirkpatrick et al., we use 0.05 weight decay to make the trained model robust to catastrophic forgetting. We apply LoRA [26] to the model with the rank $r = 16$, $\alpha = 16$, and 0.05 dropout. We also train CodeQwen1.5 [6] and CodeLlama [57] to answer our research questions. The complete model selection and naming are detailed in Table 2. We trained all the models for 200 steps until the loss converged on eight NVIDIA H100-80GB GPUs.

Our training approach is to apply causal language modeling (CLM) on the pairs of focal function and unit test function (code-test pair). With CLM, the language model can learn as an unsupervised distribution estimation from a set of focal-test-pairs examples. By placing the focal function at the beginning of each training example, the language model can better understand the causal relationship between tokens in the focal function and those in the unit test function. This allows the model to generate the desired unit test functions when given the focal function as a prompt. This process is also called generative pre-training by earlier work [50, 51]. Let $\mathcal{F} = \{f_1, \ldots, f_p\}$ denote the tokens for the focal function and $\mathcal{T} = \{t_1, \ldots, t_q\}$ denote the tokens for the corresponding unit test, a training example (focal-test-pair) in our training corpus is represented as

$$\mathcal{U} = \mathcal{F} \oplus \{\backslash\text{n}\} \oplus \mathcal{T} = \{u_1, \ldots, u_r\} \tag{1}$$

where $r = p + q + 1$ is the length of the token sequence and $\oplus$ is vector concatenation. Following [50], our training objective is defined as maximizing the log-likelihood

$$L(\mathcal{U}) = \sum_i \log P\left(u_i | u_{i-k}, \ldots, u_{i-1}, \theta\right)$$

where $k = 1028$ is the maximum sequence length (context window size) and θ is the parameters of the models. In simpler terms, our objective is to predict the next token in a token sequence given all the previous tokens. We train θ parameters of all models using the Adam optimizer [30]. The models we experimented on are a decoder-only architecture using Transformer [71], where each Transformer Block uses multi-headed self-attention. With this training object and the aforementioned hyper-parameter settings, we convert the pre-trained code generation model to a unit test generation model to aid automatic software testing.

3.3 Research Questions

To evaluate our data augmentation method, we structure our experiments around the following two 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, measuring the accuracy of generated test cases is essentially to evaluate if the language model is useful in software testing. Generating the correct input-output of the focal function requires the model to learn both the semantics and runtime behavior of the focal function. Predicting the correct inputs and corresponding outputs is challenging for language models [23], so it is an important direction for models to improve on. We follow previous work [7, 25] to extract the first 10 generated test cases to examine their standalone correctness. We 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 [25] 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. To evaluate the coverage fairly, we only apply some basic post-process (details in Section 3.4) on the generation results, run the entire test function, and record the branch coverage using `grcov` [43]. Using branch coverage as a metric for neural test coverage allows us to better evaluate the usefulness of the test generation models and of FuzzAug.

RQ.3. Can FuzzAug generalize to other models? Data augmentation methods only operate on the training set, thus applying them should improve all the models in the same task. The same augmentation method should produce similar performance improvement trends on different models. To evaluate the generalizability of FuzzAug, we apply the same training and evaluation setting for training CodeLlama [57] and report the effects of FuzzAug.

RQ.4. How does models fine-tuned with FuzzAug compare with instruction-tuned code LLMs? Instruction-tuning (IT) enhances LLM's ability to follow natural language instructions [79]. Similar to the UniTSyn dataset consisting of focal-test function pairs, IT further trains LLMs on instruction-output paired datasets. Since our evaluation setting is using instructive prompts (Section 3.4.2), we also evaluate our models against the IT version of the base model to explore the differences between the two training methods in neural test generation.

3.4 Evaluation Setup

3.4.1 Evaluation datasets. We evaluate all the models on HumanEval-X [81]. HumanEval-X is a hand-crafted multilingual benchmark for code generation tasks, containing problems in popular languages including 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. HumanEval-X was designed for code generation and code translation in different languages, but is also useful for test generation. In order to turn the Rust subset of HumanEval-X into a test generation benchmark, we follow UniTSyn [25] to concatenate the function declaration and canonical solution as the focal function, and use the focal function to prompt the model to generate the corresponding unit test.

We choose to evaluate the test generation models on HumanEval-X instead of on downloaded real-world projects for three considerations. First, we try to eliminate data leakage. The training dataset of the state-of-the-art code generation models is likely to contain the popular repositories on GitHub, including the ones we used to build our dataset. Therefore, evaluating the models on the repositories we collected from GitHub might be unfair due to data leakage. Although comparing with other state-of-the-art models is not our goal, the effect of data leakage on evaluating FuzzAug is unknown. To minimize such effects, we choose to use a hand-crafted benchmark, and HumanEval-X is the only hand-crafted benchmark we found that contains Rust problems.

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 [82]. Therefore, choosing to evaluate test generation on OSS introduces additional bias in the results, which is another thing we want to eliminate.

Third, Rust programs are hard to compile in a complex codebase. Different from other popular languages in code generation, for example, Python and JavaScript with dynamic types and C/C++ with weak types, Rust's type system is both static and strong, making code generation in Rust hard to pass the type checking and therefore fail to compile. In addition to static and strong typing,

```
1 fn has_close_elements(numbers: Vec<f32>, threshold: f32) -> bool { ... }
2 // Check the correctness of  `has_close_elements`
3 #[cfg(test)]
4 mod tests {
5     use super::*;
6
7     #[test]
8     fn test_has_close_elements() {
9         assert_eq!(has_close_elements(
```

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

Rust introduced linear logic [20] (linear types [72]) into its type system to manage the lifetime and ownership of pointers, adding additional difficulty in generating type-correct code in the complex codebase. To this end, we found HumanEval-X to be a more suitable benchmark since the problems are simple enough without any over-complex types, yet are challenging enough that the focal functions contain multiple branches to evaluate code coverage. As a result, we select HumanEval-X to evaluate FuzzAug instead of using the OSS projects.

3.4.2 Prompts. For all our experiments, we use the same prompting method as in CodeT [7] to guide the language models in generating assertions. As shown in 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 as shown on Line 9. 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.

3.4.3 Post-processing. We avoid overly intricate processing of the generated test functions to keep our evaluation results faithful. We only apply the most basic post-processing to make the generated test function complete. We first count the number of the curly brackets. If the number of opening brackets matches the number of closing brackets, we consider the generated test function as complete. Otherwise, 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 function.

4 RESULTS

4.1 Test Case Accuracy

We follow CodeT [7] 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 accuracy of the baseline StarCoder2, UnitCoder, and the competitor FuzzCoder in Table 3 under columns "Assert. CR" for the compile rate of individual assertions and "Accuracy" for the accuracy score of the individual assertions. Our model FuzzCoder trained on the augmented dataset outperforms StarCoder2 and UnitCoder on both metrics. In particular, given the same maximum token length for generation, we obtained *+5.42%* and *+3.54%* absolute gains in assertion compile rate and accuracy compared to StarCoder2, and *+2.13%* and *+2.02%* absolute gains compared to normal fine-tuning on UniTSyn. These performance improvements indicate that training with FuzzAug can help LLM learn the way to invoke the focal function correctly. Rust is a strong and statically typed language, so the improvement in compile rate indicates that FuzzCoder can generate input-output pairs that are type valid to the focal function and thus pass the type checking. This improvement is due to the additional focal function invocations with valid and executable inputs

Table 3. Evaluations of the accuracy and the usefulness of generated unit tests. Each number in the table is the mean score of all 164 tasks. Please refer to Table 2 for the names of the models. The best results are highlighted in bold.
Type: the method used to train the model, including pre-trained (PT) and fine-tuned (FT).
Assert. CR: the compile rate of the individual assertions we parsed from the generated text.
Accuracy: the number of accurate assertions over the total number of parsed assertions.
Func. CR: the compile rate of generated unit test functions.
Branch Coverage: the average branch coverage of generated unit test functions on the focal functions.

Model	Type	Assert. CR	Accuracy	Func. CR	Branch Coverage
StarCoder2	PT	64.09	31.83	45.73	9.88
UnitCoder	FT	67.38	33.35	54.88	13.66
FuzzCoder	FT	**69.51**	**35.37**	**59.15**	**14.87**
CodeQwen	PT	66.52	41.71	68.29	20.90
UnitQwen	FT	72.62	45.98	59.15	20.15
FuzzQwen	FT	**85.91**	**50.43**	**75.00**	**22.25**

collected from fuzzing, from which the model can learn the relationship between the desired input type and the type signature of the focal function.

RQ.1. Can FuzzAug improve the accuracy of generated test cases?

FuzzAug can improve the neural test generation model on all aspects of test case accuracy. FuzzCoder trained with FuzzAug shows a higher assertion compile rate and accuracy than pre-trained StarCoder2 and UnitCoder trained on UniTSyn without FuzzAug. FuzzCoder achieved +2.13% and +2.02% absolute improvement on average individual assertion compile rate and test case accuracy, respectively.

4.2 Unit Test Function 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 3 under columns "Func. CR" for function compile rate of the generated unit test functions. For whole test function compile rate, FuzzCoder achieved *+13.42%* absolute improvement over the baseline StarCoder2 and *+4.27%* absolute improvement over UnitCoder. The improvements on whole generated test function compile rate show the ability of FuzzAug in guiding LLMs to write complete unit test functions with correct syntax and types.

While generating complete unit test functions is important for LLMs, it's equally crucial that these generated tests are applicable and valuable in real-world scenarios. Coverage is a key metric for determining if the program's behavior is explored by the test efficiently [21, 33]. Therefore, we follow UniTSyn to evaluate the completeness of generated unit test functions based on their branch coverage of the corresponding focal functions. We show the result of this evaluation in Table 3 under the column "Branch Coverage." FuzzCoder achieved *+4.99%* absolute increase over StarCoder2 and *+1.32%* absolute increase over UnitCoder on branch coverage. This indicates the superior performance of FuzzAug to improve neural test generation in real-world software engineering applications, where the unit test functions are not simply asserting the input-output pairs. These improvements in branch coverage suggest that FuzzAug can help the model to understand the structure of the focal function and generate more complete unit test functions to discover potential bugs and vulnerabilities in the software. In practice, evaluating the generated unit tests as a whole function is more accurate than individual assertions for their better readability and maintainability with richer context.

RQ.2. Can FuzzAug improve the usefulness of generated unit tests?

In a real-world software engineering setting, FuzzCoder not only generates more unit test functions with correct setups, but also reaches higher branch coverage on the focal functions than UnitCoder. Our results indicate that models trained with FuzzAug can be more practical in writing valid and complete unit test functions for focal functions.

4.3 Generalizability of FuzzAug

Useful data augmentation methods should work on different models. Therefore, we also apply the same training and evaluation in RQ.1. and RQ.2. to the 7B parameter version of CodeLlama CodeQwen1.5 [6]. We choose to experiment on CodeQwen1.5 since it is the best-performing 7B code generation model on EvalPlus [39]. Through our experiment, we found CodeQwen1.5 performs better than StarCoder2 in test generation by a large margin. Applying FuzzAug to fine-tune the models further widens their performance leads. Our model FuzzQwen (fine-tuned with FuzzAug) shows a performance improvement in all four metrics mentioned in RQ.1. and RQ.2.

FuzzQwen demonstrates superior performance in test case accuracy evaluation (RQ.1). For the individual assertion compile rate, FuzzQwen achieved a *+19.39%* absolute increase over the baseline CodeQwen1.5 and a *+13.29%* absolute increase over UnitQwen, which is fine-tuned solely with UniTSyn. In terms of assertion accuracy, FuzzQwen outperforms CodeQwen1.5 by *+8.72%* and UnitQwen by *+4.45%*. Therefore, FuzzQwen not only enhances the reliability of generated test cases but also significantly improves the precision of assertions, showing the effectiveness of FuzzAug in neural test generation.

FuzzQwen also surpasses the baselines in completeness and comprehensiveness (RQ.2.). FuzzQwen achieved a *+6.71%* absolute increase in whole test function compile rate over CodeQwen1.5 and *+15.85%* over UnitQwen. For branch coverage, FuzzQwen outperforms CodeQwen1.5 and UnitQwen by absolute increases of *+1.35%* and *+2.10%*. We observe that fine-tuning CodeQwen1.5 on UniTSyn leads to some performance decrease in this evaluation setting. FuzzAug overcomes this performance decrease caused by UniTSyn and outmatches the performance of the baseline, further indicating the power of our data augmentation method in real-world software engineering.

RQ.3. Can FuzzAug generalize to other models?

FuzzAug is a general data augmentation method for the neural test generation task. Training dataset augmented by FuzzAug can be used to train different test generation models and gain performance improvement in whole function compile rate and branch coverage on the target focal function, thus improving the usefulness in real software testing scenarios.

4.4 Comparison with Instruction-tuning

Instruction-tuning enhances an LLM's ability to generate content that aligns with user instructions. Since we incorporate natural language comments in the prompt to guide the models during the generation stage, we also compare the performance of instruction-tuned models and fine-tuned models trained on UniTSyn and FuzzAug. For this experiment, we use CodeLlama [57] as the base model to train UnitLlama and FuzzLlama. CodeLlama is a popular open-source code language model that offers an instruction-tuned version and is available in a 7B size. For comparison, we also include evaluation results of CodeGemma [68].

Table 4. Comparison between instruction-tuned and fine-tuned models for neural test generation. Columns are the same as Table 3. The "IT" type is short for instruction-tuned.

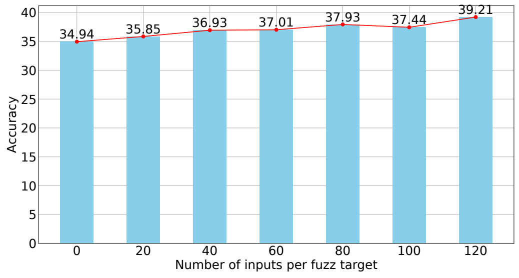
Model	Type	Assert. CR	Accuracy	Func. CR	Branch Coverage
CodeLlama	IT	64.57	32.13	54.88	15.75
CodeGemma	IT	50.24	29.94	71.95	17.02
UnitLlama	FT	69.82	34.94	65.85	17.64
FuzzLlama	FT	**74.70**	**37.44**	**75.00**	**21.69**

The results of this experiment are presented in Table 4. FuzzLlama significantly outperforms the baseline instruction-tuned CodeLlama across all four metrics. Specifically, we observed an absolute increase of *+10.13%* in individual assertion compile rate, *+5.31%* in assertion accuracy, *+20.12%* in whole-function compile rate, and *+5.94%* in branch coverage compared to CodeLlama. While CodeGemma performs better than CodeLlama out of the box, FuzzLlama surpasses it after our fine-tuning with FuzzAug. Therefore, we argue that the performance gain is contributed by FuzzAug instead of the difference in model architecture. The consistent improvement observed when comparing FuzzLlama with UnitLlama further strengthens our conclusions from RQ.1 and RQ.2. This highlights the advantages of incorporating additional fuzzing data as a form of data augmentation.

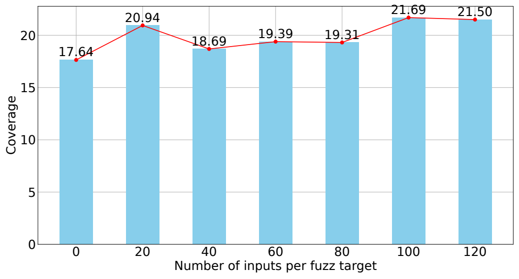
RQ.4. Can FT with FuzzAug out-perform IT in neural test generation?

Our experiments show that FuzzLlama, fine-tuned with FuzzAug, outperforms both the baseline instruction-tuned CodeLlama and CodeGemma across all key metrics, including compile rate and branch coverage. This demonstrates the benefits of incorporating additional fuzzing data for enhanced model performance for neural test generation.

5 ABLATION STUDY



(a) Assertion accuracy.



(b) Branch coverage.

Fig. 3. Change of assertion accuracy and branch coverage with respect to the number of fuzzing inputs used for data augmentation. Zero input per fuzz target is the dataset we used to train UnitCoder and UnitCoder.

We explore the effects of the hyper-parameter N in the data augmentation process shown in Algorithm 1. During the data augmentation process, the hyper-parameter N determines the number of new training examples generated by FuzzAug by using N recorded fuzz inputs. A larger N results in a larger training set. We investigate the impact of different numbers of fuzzing inputs on assertion accuracy and branch coverage by experimenting with $N = 0, 20, 40, 60, 80, 100, 120$ on

CodeLlama, as shown in Figure 3. Our findings suggest that all numbers of fuzzing data improve accuracy and branch coverage compared to the baseline. Assertion accuracy shows an almost linear increase as N increases. Branch coverage does not follow a clear pattern, though models trained with more fuzzing data ($N = 100, 120$) achieve higher coverage on focal functions. We selected $N = 100$ for our general evaluations because it provides the highest coverage and strikes a practical balance between performance gains and computational resource requirements.

6 DISCUSSIONS

6.1 Applying FuzzAug to Different Languages

As briefly introduced earlier, FuzzAug can be applied to any language supported by OSS-Fuzz [63] and libFuzzer [62]. This section details how syntax transformation is extended to popular programming languages such as C/C++, Java, and Python. For LLVM-supported languages like C/C++, libFuzzer is natively supported, and reporter instrumentation is implemented via LLVM's analysis and transform passes. Similarly, Java Virtual Machine (JVM) languages can benefit from fuzzing through Jazzer [29]. Using `javaagent` for reporter instrumentation, FuzzAug is applicable to Java datasets and unit test generation in Java. Python is supported through Google's `atheris` [22] for fuzzing, with syntax transformation leveraging the `ast` module and decorators for instrumentation. By supporting LLVM, JVM, and Python, FuzzAug provides broad compatibility, advancing data augmentation across widely used programming languages and contributing significantly to program testing by deep language models.

6.2 Applying FuzzAug to Different Datasets

We followed TeCo [46] and UniTSyn [25] to construct our dataset on function-level code-test pairs, In our training setting, each training example consists of a focal function and a unit test function concatenated by a newline symbol, providing efficiency and strong performance. However, the file-level pairing approach used in CAT-LM [52] offers additional benefits by providing more relevant context, which is particularly useful in less modular, tightly coupling, complex software systems. To accommodate both function-level and file-level data, a robust data augmentation method must be applicable to both levels. FuzzAug satisfies this requirement by supporting file-level pairing, as 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 adopts CAT-LM's pairing strategy. This versatility enhances FuzzAug's ability to augment and improve various types of unit test datasets effectively.

6.3 Function Specification as Test Template

Property-based testing (PBT) [10] is a hybrid approach between self-assessment testing like unit tests and randomized testing like fuzzing. In PBT, the developers identify a set of properties that the function should satisfy and define the input domain for each of the properties. The framework generates suitable inputs based on the instruction to test these properties randomly. The set of properties defined for a function is called its *specification*. PBT is also commonly used in software engineering to find unexpected behaviors in the program, especially in functional programming where each function is pure and specializes in one purpose, making the specification easier to define. PBT verifies the correctness of the focal function by checking if the specifications hold for all possible inputs.

Since PBT also generates random inputs to test the program, we can use these specifications as our test template. Recent studies have combined the coverage-guided input generation feature of fuzzing and PBT as coverage-guided property-based fuzzing [36, 48]. Combining the advantages

```
fuzz_target!(|data: &[u8]| {
    let encoded = STANDARD.encode(data);
    let decoded = STANDARD.decode(&encoded).unwrap();
    assert_eq!(data, decoded.as_slice());
});
```

Listing 7. Example fuzz targets added by FuzzAug from base64 that employs property-based fuzzing.

of fuzzing and PBT, this approach is widely accepted in testing memory-safe languages. In the augmented dataset we used to train FuzzCoder, some of the augmented unit test functions are converted from property-testing fuzz targets as shown in Listing 7. Through our experiments, we find that adding the knowledge of function specification in the training set contributes to the completeness and coverage of the generated unit test function. The feature of introducing specifications and properties in data augmentation further enhance our contribution to neural test generation by designing and building FuzzAug.

6.4 Evaluation on Real-World Projects

In our experiments, we assessed the validity and completeness of generated unit test functions using HumanEval-X [81]. We did not use real-world Rust projects due to the challenges outlined in Section 3.4.1, where we discuss the difficulties in efficiently compiling and evaluating generated Rust code within complex environments. For analysis, we executed the tests in UniTSyn, which are contributed by each project's maintainers. We found that most test functions covered less than 5% of the codebase individually. This suggests that utilizing LLMs to generate these functions may not yield substantial comparative insights. Additionally, we experimented with test function generation using CodeQwen1.5 [6], UnitQwen, and FuzzQwen, where we replaced the original unit test with the generated ones in the codebase and attempted to compile them. However, all the unit test functions produced by these models failed to compile successfully. Therefore, our evaluation was confined solely to HumanEval-X. Once code LLMs enhance their ability to generate syntactically correct code and manage types in strict programming languages like Rust, researchers will be able to evaluate the performance of FuzzAug or similar tools on real-world projects.

7 RELATED WORKS

7.1 Fuzzing

Fuzz testing [78], or fuzzing, is a popular execution-based dynamic testing technique with randomized inputs, which aims to generate a set of inputs based on the provided set of seeds to achieve high code coverage. The fuzzing process can be described as a coverage-guided input modification method. The fuzzer uses behavior monitoring to find inputs with high branch coverage and favor those inputs for future input generation. Behavior monitoring enables the fuzzer to focus on exploring new program branches that are not yet tested to detect unseen security issues like memory errors. AFL++ [16] and libFuzzer [62] are two of the most popular fuzzers that are widely adopted in testing real-world software. Libfuzzer is integrated into the LLVM compiler infrastructure [37] to test projects in LLVM-compiled languages, like C/C++, Rust, Swift, and Julia. One of the advantages of libFuzzer is to discover memory errors via the memory sanitizer, but it is also useful to find other unexpected behaviors and crashes in memory-safe language with garbage collection like JVM-based languages [29] and Python [63]. Fuzzing is a robust method to find bugs in various software domains [9, 54, 55].

Coverage-guided grey-box fuzzing verifies the software by repeatedly generating inputs for the software to execute and prioritize the inputs that reach new paths in the program. One of the key components is to determine if the input seed triggers new and interesting behaviors. Several related work have proposed methods to improve fuzzing in this direction, including using deterministic [56] and gradient-based [8, 64] techniques. This coverage-guided input generation feature allows the selected and generated input seeds to carry information on the program's dynamic runtime behavior, which is shown successfully and useful in training machine learning models for code understanding [27, 80].

Fuzzing was adopted as a data augmentation tool to improve the robustness of neural networks. SenSei [19] introduced fuzzing to improve the robust generalization of DNNs in training. Their key insight is that guided search like fuzzing and genetic programming can be more effective in finding optimal variants compared to other DA-produced random variants as described by Krizhevsky et al.. The authors also pointed out that SenSei can help the training process by skipping training data that is considered "not interesting" by fuzzing. This work mainly focuses on improving the robustness of neural networks, and is not a work to improve generative language models and neural test generation. Although both incorporated fuzzing to mutate the training dataset, SenSei and our work FuzzAug are fundamentally different in approach and application.

7.2 Test Generation via Large Language Models

Using deep neural networks, especially LLMs, to generate test cases is a new trend in automatic software testing. This method is referred to as *neural test generation*. Some approach toward neural test generation is to instruct pre-trained code generation LLMs like CodeLlama [57] and StarCoder2 [42]. Similarly, large foundation models like ChatGPT [3] can also be prompted to generate test cases [60, 67]. The other approach is to train test-specific models that are specialized in generating test cases or unit test functions. This category of neural test generation includes ATLAS [73], AthenaTest [70], TOGA [14], A3Test [4], TeCo [46], CAT-LM [52], and UniTSyn [25]. The more recent work [25, 46, 52] proposed to train the test generation model on *aligned* data that includes the correspondence between the unit test and the function under test (also referred to as *focal function*). TeCo and UniTSyn are in the direction of function-level alignment to provide fine-grained correspondence, whereas CAT-LM adopted file-level alignment for the efficiency of pairing and richer contexts.

At inference time, previous research can be divided into only generate test cases (input-output pair in assertions) [4, 14, 46, 73] and generate complete test files [52] or functions [25]. Generating only the test cases might not be useful in real-world software engineering as they are difficult to set up for execution as part of continuous integration (CI). Writing the complete test file or test function is more promising in practice. However, generating a whole file at once has limitations. First, it demands a long context window and more tokens, which is costly in practice for large software repositories. Second, the LLM might be distracted by the long context, causing the generated unit tests to reach insufficient coverage on each of the focal functions. Therefore, generating each unit test function for every single focal function is preferable.

Previous work [60, 67] evaluated the code coverage of unit test functions generated for real-world projects. These studies employed significantly larger models, like ChatGPT, for test generation and selectively retained only those results that successfully compiled, filtering out non-compilable outputs. This approach, in practice, is more similar to randomized testing methods such as fuzzing but leverages GPU resources instead. It is important to note that these experiments were primarily conducted on dynamically interpreted languages like JavaScript and Python, or on languages that run on a virtual machine, such as Java. These languages facilitate execution even when the entire generated function contains errors. Given the challenges in compiling Rust code, we have opted not to adopt their evaluation scheme. Instead, we follow the methodology outlined in UniTSyn to assess branch coverage on the popular public code generation benchmark HumanEval-X [81].

7.3 Data Augmentation for Language Processing

Data augmentation was first developed for computer vision (CV) and later adopted by natural language processing (NLP). With the advancement of LLMs, data augmentation techniques gain additional interest from low-resource domains and new tasks due to the scaling law of large models [15]. The challenge of data augmentation for natural languages is the discreteness of features, where methods like injecting random noise, cropping, and rotating for images introduced by [35, 66] cannot be directly applied to NLP. DAs for NLP are mainly rule-based [24, 59, 74, 75] or model-based [18, 34, 47, 61]. Rule-based DA applies techniques like random insertion, deletion, and swap to text at the token level. Model-based DA utilizes language models to translate or rewrite existing text, or generate new text as new data. These methods are effective but still lack quality control since the augmented data might contain unnatural or grammatically incorrect text, leading to models learning incorrect patterns.

DA for programming languages is still insufficiently studied compared to DA for NL due to the stricter grammar. Yu et al. proposed to augment code data by source-to-source program transformation. This approach is rule-based and operates on the program based on the denotational semantics. Similar to program transformation techniques used to generate simple program for compiler testing [38, 40, 55, 76], Yu et al. designed a set of grammar transformation rules to alter small parts of the program to produce a new valid program. This approach can generate a large amount of programs efficiently but only works on simple programs whose complexity cannot compare with real-world software. Therefore, this method is not ideal for training large generative models. Up until we wrote this paper, there existed no DA for unit test datasets and neural test generation like our method FuzzAug.

8 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 model 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 are supported by OSS-Fuzz with slight modifications. Our results show the effectiveness of employing dynamic program analysis to generate high-quality inputs to argument 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.

DATA AVAILABILITY

Our code and data are available [2].

REFERENCES

[1] [n. d.]. *Tree-sitter Introduction.* https://tree-sitter.github.io/tree-sitter/

[2] 2024. *FuzzAug.* https://doi.org/10.5281/zenodo.13751399

[3] 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* (2023).

[4] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).

[5] 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! arXiv:2301.03988 [cs.SE] https://arxiv.org/abs/2301.03988

[6] 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. arXivpreprintarXiv:2309.16609

[7] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*.

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

[9] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1600–1614.

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266

[11] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211.

[12] David Tolnay. 2024. syn: Parser for Rust source code. https://github.com/dtolnay/syn

[13] David Tolnay and Alex Crichton. 2024. proc-macro2: A substitute implementation of the compiler's 'proc_macro' API to decouple token-based libraries from the procedural macro use case. https://github.com/dtolnay/proc-macro2

[14] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. https://doi.org/10.1145/3510003.3510141

[15] Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for NLP. *arXiv preprint arXiv:2105.03075* (2021).

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

[17] 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*. 416–419.

[18] Fei Gao, Jinhua Zhu, Lijun Wu, Yingce Xia, Tao Qin, Xueqi Cheng, Wengang Zhou, and Tie-Yan Liu. 2019. Soft contextual data augmentation for neural machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 5539–5544.

[19] 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*. 1147–1158.

[20] Jean-Yves Girard. 1987. Linear logic. *Theoretical computer science* 50, 1 (1987), 1–101.

[21] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[22] Google. [n. d.]. *Atheris: A Coverage-Guided, Native Python Fuzzer.*

[23] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065* (2024).

[24] Bharath Hariharan and Ross Girshick. 2017. Low-shot visual recognition by shrinking and hallucinating features. In *Proceedings of the IEEE international conference on computer vision*. 3018–3027.

[25] 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)* (2024-09-16/2024-09-20). Vienna, Austria. https://doi.org/10.1145/3650212.3680342

[26] 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*.

https://openreview.net/forum?id=nZeVKeeFYf9

[27] Jiabo Huang, Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. 2023. Code Representation Pre-training with Complements from Program Executions. arXiv:2309.09980 [cs.SE]

[28] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[29] Code Intelligence. 2024. jazzer: About Coverage-guided, in-process fuzzing for the JVM. https://github.com/CodeIntelligenceTesting/jazzer.

[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[31] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[32] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press, USA.

[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[34] Sosuke Kobayashi. 2018. Contextual Augmentation: Data Augmentation by Words with Paradigmatic Relations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 452–457. https://doi.org/10.18653/v1/N18-2072

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[36] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[37] 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.* IEEE, 75–86.

[38] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[39] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[40] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[41] Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

[42] 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. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[43] Marco Castelluccio. 2024. grcov: Rust tool to collect and aggregate code coverage data for multiple source files. https://github.com/mozilla/grcov

[44] Marshall Pierce <marshall@mpierce.org>. 2024. base64: encodes and decodes base64 as bytes or utf8. https://github.com/marshallpierce/rust-base64

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

[46] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2111–2123.

[47] Yuyang Nie, Yuanhe Tian, Xiang Wan, Yan Song, and Bo Dai. 2020. Named Entity Recognition for Social Media Texts with Semantic Augmentation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 1383–1391. https://doi.org/10.18653/v1/2020.emnlp-main.107

[48] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.

[49] 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)*. IEEE, 523–533.

[50] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[51] 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 (2019), 9.

[52] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J Hellendoorn. 2023. CAT-LM training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

[53] Sylvestre-Alvise Rebuffi, Sven Gowal, Dan Andrei Calian, Florian Stimberg, Olivia Wiles, and Timothy A Mann. 2021. Data augmentation can improve robustness. *Advances in Neural Information Processing Systems* 34 (2021), 29935–29948.

[54] Yuyang Rong, Peng Chen, and Hao Chen. 2020. Int egrity: 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*. Springer, 360–380.

[55] Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. 2024. IRFuzzer: Specialized Fuzzing for LLVM Backend Code Generation. arXiv:2402.05256 [cs.SE]

[56] Yuyang Rong, Chibin Zhang, Jianzhong Liu, and Hao Chen. 2024. Valkyrie: Improving fuzzing performance through deterministic techniques. *Journal of Systems and Software* 209 (2024), 111886.

[57] 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. arXiv:2308.12950 [cs.CL]

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

[59] Eli Schwartz, Leonid Karlinsky, Joseph Shtok, Sivan Harary, Mattias Marder, Abhishek Kumar, Rogerio Feris, Raja Giryes, and Alex Bronstein. 2018. Delta-encoder: an effective sample synthesis method for few-shot object recognition. *Advances in neural information processing systems* 31 (2018).

[60] 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 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[61] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving Neural Machine Translation Models with Monolingual Data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 86–96. https://doi.org/10.18653/v1/P16-1009

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

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

[64] 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)*. IEEE, 803–817.

[65] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. 2021. Text data augmentation for deep learning. *Journal of big Data* 8, 1 (2021), 101.

[66] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[67] 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 (mar 2024), 1340–1359. https://doi.org/10.1109/TSE.2024.3382365

[68] CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. 2024. CodeGemma: Open Code Models Based on Gemma. arXiv:2406.11409 [cs.CL] https://arxiv.org/abs/2406.11409

[69] The rust-fuzz Project Developers. 2024. cargo-fuzz: A 'cargo' subcommand for fuzzing with 'libFuzzer'! Easy to use! https://github.com/rust-fuzz/cargo-fuzz/

[70] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE]

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

[72] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.

[73] 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. https://doi.org/10.1145/3377811.3380429

[74] Jason Wei and Kai Zou. 2019. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 6382–6388. https://doi.org/10.18653/v1/D19-1670

[75] Qizhe Xie, Zihang Dai, Eduard Hovy, Thang Luong, and Quoc Le. 2020. Unsupervised data augmentation for consistency training. *Advances in neural information processing systems* 33 (2020), 6256–6268.

[76] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[77] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data augmentation by program transformation. *Journal of Systems and Software* 190 (2022), 111304.

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

[79] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. 2023. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792* (2023).

[80] 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)*. Toronto, Canada.

[81] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 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*. 5673–5684.

[82] Hao-Nan Zhu and Cindy Rubio-González. 2023. On the reproducibility of software defect datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2324–2335.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009