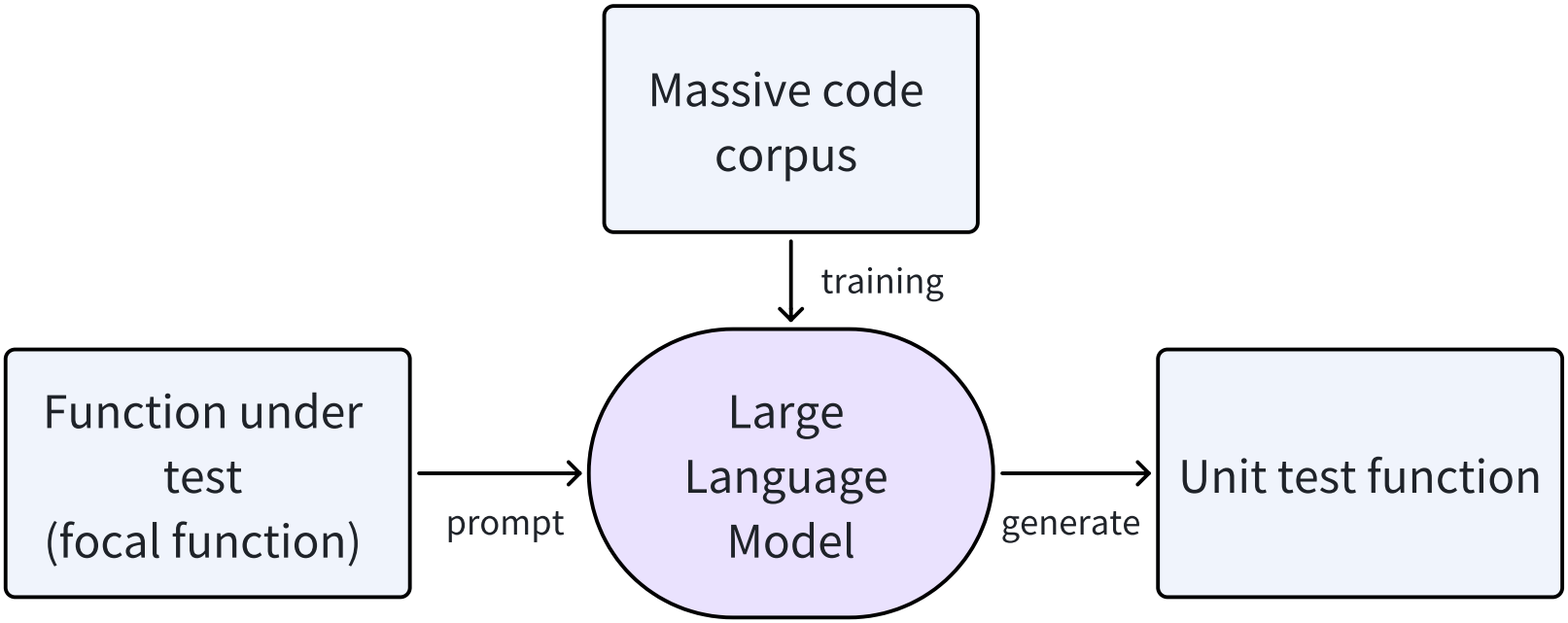
FuzzAug: Data Augmentation by Coverage-guided Fuzzing for Neural Test Generation

Yifeng He, Jicheng Wang, Yuyang Rong, Hao Chen

University of California, Davis

UCDAVIS
COMPUTER SCIENCE

LLM-based automatic unit test generation (neural test generation)



Using an LLM to generate unit tests involves three steps:

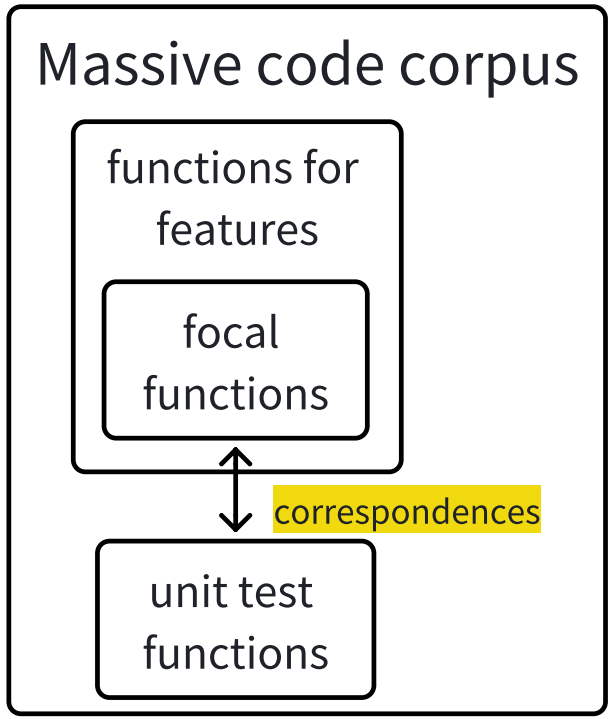
1. Train an LLM with a massive code corpus.
2. Prompt the LLM with the function under test (focal function).
3. Let the LLM generate a unit test function.

Test generation is challenging for LLM

Unit test functions and their focal functions have:

1. Fundamental correspondences.
2. Different representations.
3. Imbalanced data volumes.

Previous work attempted to address these challenges by: specialized training datasets with *aligned focal-test pairs*.



- CAT-LM [Rao+24] aligned focal-test pairs on file-level by matching names.
- UniTSyn [He+24] used LSP to align function-level focal-test pairs.

The remaining problem: limited dataset size

But still, current (small size) LLM-based models are not efficient enough:

1. Testing code is only 20% of the codebase [Rao+24].
2. Test corpus is limited in size for newer programming languages.
   - CAT-LM: Python and Java.
   - UniTSyn: Python, Java, Go, C/C++, JavaScript.

With newer languages getting popular in correctness-critical systems, for example *Rust*, the problem is more severe.

Our goal: a data augmentation (DA) technique to generate *meaningful* unit test data in such languages to train LLMs for unit test generation.

Key challenges of DA for test generation

1. Random data should not be purely noise.
   - DA should help in exploring the program's behavior space.
2. DA has to provide valid testing semantics.
   - The added tokens should provide a correct way to parse the input and invoke the focal function.

Our approach:

1. Use *coverage-guided fuzzing* to gather input data that are *interesting* to the focal function.
2. Use program transformation to convert existing *fuzz targets* into test templates with *valid testing semantics*.

FuzzAug: an overview

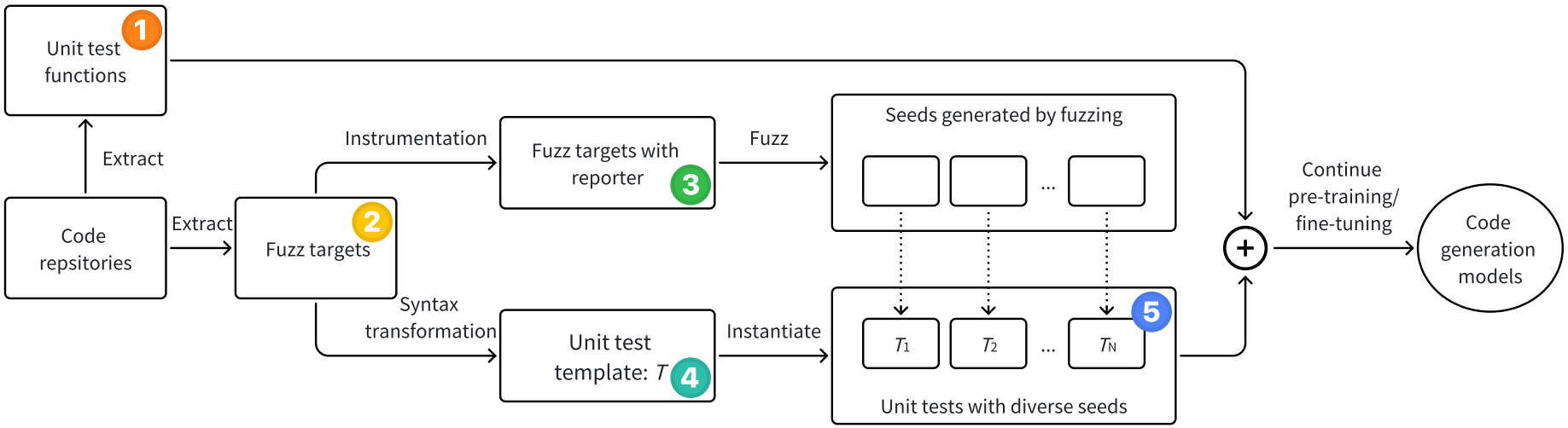


Figure 1: Data Augmentation by fuzzing for neural test generation.

1. extract unit test functions (Listing 1)
2. extract fuzzing targets (Listing 2).
3. instrument fuzz targets with a reporter (Listing 3) for DA inputs.
4. transform fuzz targets into a unit test template (Listing 4).
5. instantiate the templates with valid test inputs collected (Listing 5).

Fuzzing for coverage-guided random inputs

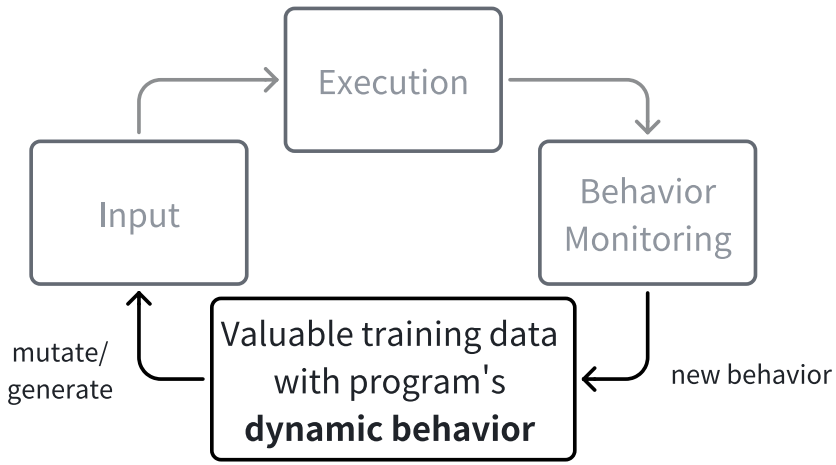


Figure 2: Fuzzing loop for dynamic program testing

- Previous studies [Zha+23; Hua+24] show that input-output pairs from fuzzing are helpful for code understanding (clone detection, code searching, ...).
- In FuzzAug, inputs triggering new program behaviors are collected for augmentation.

Unifying code representations

Inputs collected from fuzzing are great for code understanding tasks with BERT, but how can we use them to train generative models?

1. Compiler-frontend [1] semantic-preserving transformation.
   fuzz target (Listing 2) → test template (Listing 4).
2. Instantiate test template with valid inputs (Listing 5).
3. Result: valid "unit test" functions that read and execute a coverage-guided fuzzing input.

[1] For Rust, we can use procedural macros to modify the AST. We did not customize the Rust parser.

Fuzz augmentation

Algorithm 1 Data Augmentation By Coverage-Guided Fuzzing

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
    $dataset_{aug} \leftarrow []$
    for all $t \in$ GetFuzzTarget(*repo*) **do**
        $t' \leftarrow$ ReporterInstrumentation(t)
        inputs $\leftarrow$ Fuzz(t', *timeout*)
        inputs$'$ $\leftarrow$ Filter(λx : len(x) $<$ *L*, *inputs*)
        selected $\leftarrow$ Sample(*N*, *inputs*$'$)
        templates $\leftarrow$ SyntaxTransformation(t)
        aug $\leftarrow$ Instantiate(*templates*[: *N*], *selected*)
        $dataset_{aug} \leftarrow dataset_{aug} + aug$
    return $dataset_{aug}$

Our augmented dataset

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+24]. Fuzz: the dataset we transformed from fuzz targets using 1, where $N = 40$. Augmented dataset: the combination of unit tests and fuzz, which is used to train the fuzz models.

Evaluation setup

- Language: Rust.
- Training dataset: UniTSyn [He+24] + FuzzAug.
- Evaluation benchmark: HumanEval-X [Zhe+23].
- Compared models (details in Table 4):
  - Baseline: StarCoder2 [Loz+24], CodeLlama [Roz+23], CodeQwen1.5 [Bai+23].
  - UniTSyn (no DA): UnitCoder, UnitLlama, UnitQwen.

Our research questions to answer

1. What makes generated unit tests good?
   - Can FuzzAug improve the accuracy of generated test cases?
   - Can FuzzAug improve the validity and completeness of generated unit tests?
2. Can FuzzAug generalize to different models?
3. Can we further scale up the dataset with FuzzAug?

Test Case Correctness

- The assertion's input and output have the correct type as the focal function.
- *Accurate*: the generated output matches the execution result.
- We extract the assertion from the generated test cases, compile, and execute them individually.

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 2: 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.

Test Validity and Completeness

- We measure if generated test *function* is *complete* and *Useful*.
- *Complete*: the test function has correct syntax and can be compiled.
- *Useful*: the test function has high branch coverage.
- We compile and execute the test function as a whole.

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**
CodeQwen1.5	PT	68.29	20.90
UnitQwen	FT	60.37	20.76
FuzzQwen	FT	**73.17**	**24.63**

Table 3: 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.

Further scaling FuzzAug I

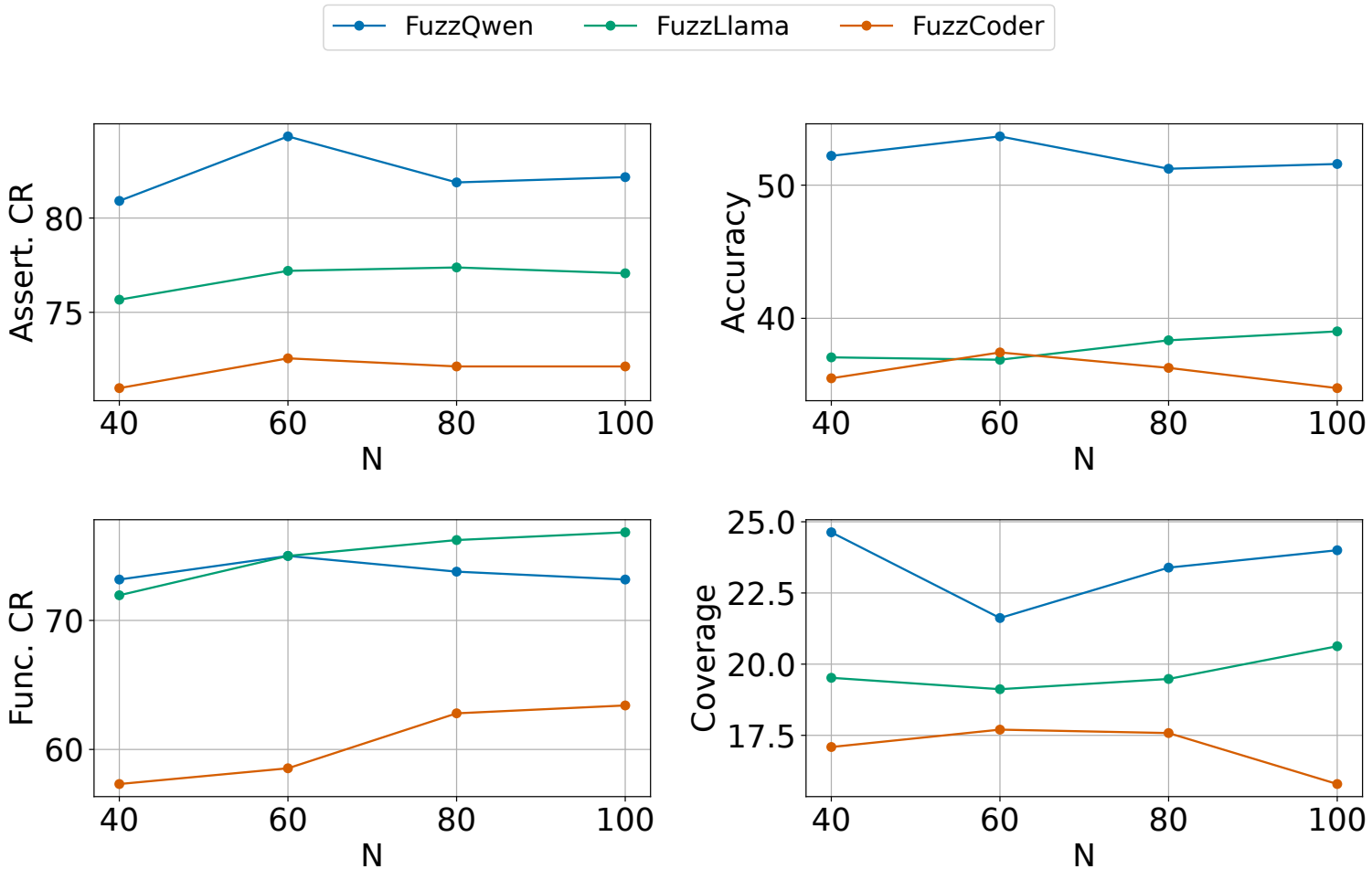


Figure 3: The impact of scaling FuzzAug on test generation performance.

Further scaling FuzzAug II

We train models with $N = 40, 60, 80, 100$ fuzzing samples.

- Scaling up FuzzAug is not the primary factor influencing performance.
- The quality of data augmentation plays a more crucial role.
  - driven by the test semantics of the fuzz targets and coverage-guided inputs.
- We recommend N at a scale comparable to the original training dataset.
  - For tasks with a clear oracle (i.e, software engineering), keep a balance between real and syntactic data.

Conclusion: key take-aways

- We propose FuzzAug, the first data augmentation technique for LLM-based unit test generation.
- Inputs obtained from fuzzing are useful; they provide diversity and coverage to execute the focal functions.
- Fuzz targets are useful after transformation; they provide testing semantics for parsing inputs and invoking focal functions.
- We evaluate FuzzAug on Rust and show that it improves the quality of LLM-based unit test generation:
  - **individual test cases** (assertions), in both compile rate (type checking) and accuracy (matching outputs)
  - **unit test functions**, in both completeness (correct syntax, can be compiled) and usefulness (branch coverage).

Thanks For Your Attention!

Any questions?

Paper

Code on GitHub

[Rao+24] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. “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. Echternach, Luxembourg: IEEE Press, 2024, pp. 409–420. ISBN: 9798350329964. DOI: 10.1109/ASE56229.2023.00193. URL: https://doi.org/10.1109/ASE56229.2023.00193.

[He+24] Yifeng He et al. “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, Sept. 16–20, 2024. URL: https://doi.org/10.1145/3650212.3680342.

[Zha+23] Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. "Understanding Programs by Exploiting (Fuzzing) Test Cases". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 10667–10679. DOI: 10.18653/v1/2023.findings-acl.678. URL: https://aclanthology.org/2023.findings-acl.678/.

[Hua+24] Jiabo Huang et al. "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)*. Miami, Florida, US: Association for Computational Linguistics, Nov. 2024, pp. 267–278. DOI: 10.18653/v1/2024.emnlp-industry.21. URL: https://aclanthology.org/2024.emnlp-industry.21/.

[Zhe+23] Qinkai Zheng et al. “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. Long Beach, CA, USA: Association for Computing Machinery, 2023, pp. 5673–5684. ISBN: 9798400701030. DOI: 10.1145/3580305.3599790. URL: https://doi.org/10.1145/3580305.3599790.

[Loz+24] Anton Lozhkov et al. *StarCoder 2 and The Stack v2: The Next Generation*. 2024. arXiv: 2402.19173 [cs.SE]. URL: https://arxiv.org/abs/2402.19173.

[Roz+23] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2023. arXiv: 2308.12950 [cs.CL].

[Bai+23] Jinze Bai et al. *Qwen Technical Report*. 2023. URL: arXiv%20preprint%20arXiv:2309.16609.

[Pie24] Marshall Pierce. *base64: encodes and decodes base64 as bytes or utf8*. Sept. 2024. URL: %7Bhttps://github.com/marshallpierce/rust-base64%7D.

```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];  // 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 5: Simplified examples extracted from project `base64` [Pie24] in our collected Rust dataset. Each example listing corresponds to one step in Figure 1.

Models

Table 4: Our model selection for evaluation.
Baseline: names of the pre-trained or instruction-tuned baseline models.
Type: pre-trained (PT), instruction-tuned (PT).
UniTSyn: corresponding models fine-tuned using the UniTSyn [He+24] dataset.
FuzzAug: corresponding models fine-tuned using UniTSyn + FuzzAug.

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

Prompt and post-processing

```
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     #[test]
7     fn test_has_close_elements() {
8         assert_eq!(has_close_elements(
```

Listing 1: Example prompt used for test generation. Import statements are removed for simplicity.

- We use the focal function as the prompt, concatenated with a NL instruction and the test header, following UniTSyn [He+24].
- We avoid overly intricate post-processing, just adding the missing brackets and semicolons.