
Revisiting TitanFuzz: A Reproduction and Comparative Evaluation Against DeepREL

Chen En Lo

National Yang Ming Chiao Tung University

Abstract

Testing deep learning frameworks has long been a challenging task. TitanFuzz introduces an LLM-based fuzzing approach that combines autoregressive code generation and code infilling to generate diverse test programs. In this study, we reproduce TitanFuzz and evaluate its effectiveness against DeepREL, an API-level fuzzer, on PyTorch. We select 30 APIs across six categories and conduct ten independent runs per API for each fuzzer, with a fixed time budget of 60 seconds per run. By independently replaying up to 50 generated candidates per run, we measure additional PyTorch Python source lines, previously unseen ATen operator names, valid test yield, and potential bugs. On average, TitanFuzz covers 22.75 additional Python lines and 17.54 new ATen operator names per API-run, compared with 3.44 and 13.21 for DeepREL. However, TitanFuzz produces a lower valid yield (50.69% versus 68.42%). These results highlight a trade-off between broader program exploration and test validity.

1 Introduction

Although large language models have become widespread and extensively studied, testing deep learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) remains a promising research area.

Previous studies have primarily focused on two approaches: **API-level fuzzing** and **model-level fuzzing** (Deng et al., 2023, 2022; Gu et al., 2022). However, both approaches face two major limitations.

1) *Limited diversity of API sequences.* API-level fuzzers typically focus on testing individual DL APIs, making it difficult to detect bugs triggered by sequences of API calls. Although model-level fuzzers can potentially test API sequences, they often cover only a limited subset of APIs and interaction patterns. For example, Muffin, proposed by Gu et al. (2022), requires manually specified input/output constraints for supported DL APIs and uses additional reshaping operations to ensure valid connections.

2) *Limited ability to generate arbitrary code.* DL frameworks often expose their interfaces through Python, a dynamically typed language, making it difficult to statically infer API argument and return types. As discussed by Deng et al. (2023), existing DL framework fuzzers often rely on constrained program-generation or mutation strategies, which can limit the diversity of generated code.

Deng et al. (2023) hypothesize that LLMs trained on enormous corpora implicitly learn both Python syntax and semantics, as well as the complex type constraints and relationships among DL APIs. This knowledge enables LLMs to generate and mutate potentially valid DL programs for fuzzing. Based on this insight, they propose TitanFuzz, an LLM-based framework for testing DL libraries.

1.1 Approach

TitanFuzz, introduced by Deng et al. (2023), proposes a fuzzing approach that combines large language models with evolutionary fuzzing. First, it employs Codex (Chen et al., 2021) with step-by-step prompting to generate high-quality seed programs for target APIs. Next, an evolutionary fuzzing algorithm selects seed programs based on their fitness scores and dynamically chooses mutation operators using a multi-armed bandit (MAB) algorithm. The selected operators mask specific portions of the seed programs, which are subsequently filled by InCoder (Fried et al., 2022), a code-infilling model, to generate new mutants. The generated mutants are then filtered to eliminate execution failures and evaluated using a fitness function. Valid mutants are added to the seed bank, where higher-scoring seeds are prioritized for subsequent mutation rounds to improve the effectiveness and diversity of test cases. Finally, TitanFuzz performs differential testing by executing the generated programs on CPU and GPU backends to identify potential software bugs.

1.1.1 Initial Seed Generation

Figure 1 illustrates an example of the constructed prompt and its corresponding output. TitanFuzz employs step-by-step prompting to improve the quality of generated programs by instructing Codex (Chen et al., 2021) to perform three tasks sequentially: (1) import the target library, (2) generate input data, and (3) invoke the target API. The constructed prompt is then provided to Codex, which generates seed programs based on these instructions.

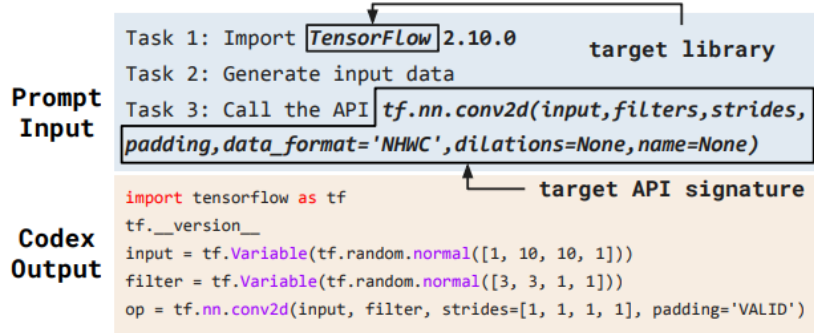


Figure 1: Example generation from the Codex model, as discussed by Deng et al. (2023).

1.1.2 Evolutionary Fuzzing

Algorithm 1 presents the evolutionary fuzzing process of TitanFuzz (Deng et al., 2023). Initially, the seed bank is populated with programs generated by Codex, and a prior distribution is initialized for each mutation operator. During each iteration, a seed is selected based on its fitness score, with higher-scoring seeds receiving greater selection probabilities through a softmax function. A Multi-Armed Bandit (MAB) algorithm then dynamically selects a mutation operator based on its previous performance. The selected operator masks portions of the seed program, which InCoder subsequently fills to generate new mutants. These mutants are evaluated to identify valid programs, and the results are used to update the mutation operator’s posterior distribution. Finally, valid mutants are scored according to their API interaction diversity and added to the seed bank for future iterations. This process continues until the time budget is exhausted.

1.1.3 Mutation Operators

As shown in figure 2, TitanFuzz employs four mutation operators: argument, prefix, suffix, and method (Deng et al., 2023). Each operator first identifies the target API call (e.g., torch.mm) in a seed program and replaces a specific portion of the code with a special token. For instance, the argument mutation operator masks the arguments of the target API call. The resulting masked code is then provided to a code infilling model, which generates new code to replace the masked region. By applying different mutation operators,

Algorithm 1: Evolutionary fuzzing algorithm

```

1 Function EvoFuzz(API, Seeds,  $T_{\text{Budget}}$ ):
   Input: The test target API, the seed programs Seeds, and the time budget  $T_{\text{Budget}}$ 
   Output: The generated programs
2   SeedBank  $\leftarrow$  Seeds;
3   InitializeMPrior();
4   while  $T_{\text{Elapsed}} \leq T_{\text{Budget}}$  do
5     CurrentSeed  $\leftarrow$  SelectSeed(SeedBank);
6     MutationOp  $\leftarrow$  SelectMutationOp();
7     MaskedInput  $\leftarrow$  Mask(CurrentSeed, MutationOp);
8     Samples  $\leftarrow$  InCoder(MaskedInput);
9     (ValidSamples, InvalidSamples)  $\leftarrow$  Evaluate(Samples);
10    UpdateMPosterior(MutationOp, Count(ValidSamples), Count(InvalidSamples));
11    FitnessScore  $\leftarrow$  FitnessFunction(ValidSamples);
12    SeedBank  $\leftarrow$  SeedBank  $\cup$  ValidSamples;
13  return SeedBank;

```

TitanFuzz can generate diverse program variants while preserving the target API call for testing.

1.1.4 Mutation Operator Selection

As shown in algorithm 2, TitanFuzz formulates mutation operator selection as a Multi-Armed Bandit (MAB) problem (Deng et al., 2023), since different mutation operators may perform differently across target APIs. Each operator is treated as an arm with an unknown success probability, defined by its ability to generate valid and unique programs. To balance exploration and exploitation, TitanFuzz employs Thompson Sampling (Chapelle and Li, 2011) with a Beta-Bernoulli model.

Initially, each operator is assigned a uniform Beta(1,1) prior. During each iteration, a value is sampled from each operator’s posterior distribution, and the operator with the highest sampled value is selected. After generating and evaluating mutants, the posterior distribution is updated according to the numbers of successful and failed samples. This adaptive strategy prioritizes effective operators while continuing to explore alternatives. The distributions are reinitialized for each target API.

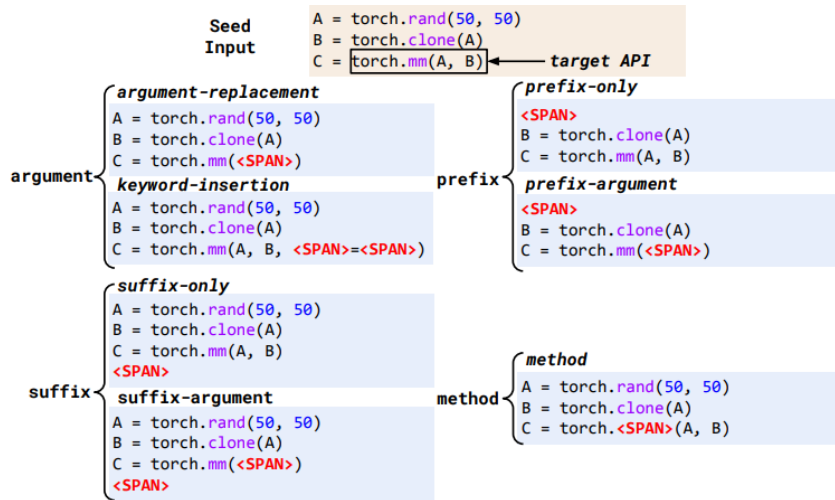


Figure 2: Mutation operator outputs (masked inputs to the infilling model; see Deng et al., 2023).

Algorithm 2: Mutation operator selection algorithm

```
1 Function InitializeMPrior():  
2   for  $m \in \text{MutationOps}$  do  
3      $m.S, m.F \leftarrow 1, 1$ ;  
4 Function SelectMutationOp():  
   Output: The chosen mutation operator  $m$   
5   for  $m \in \text{MutationOps}$  do  
6      $\theta_m \sim \text{Beta}(m.S, m.F)$ ;  
7    $m^* = \arg \max_m \theta_m$ ;  
8   return  $m^*$ ;  
9 Function UpdateMPosterior( $m, \text{NumValid}, \text{NumInvalid}$ ):  
10   $m.S \leftarrow m.S + \text{NumValid}$ ;  
11   $m.F \leftarrow m.F + \text{NumInvalid}$ ;
```

1.1.5 Code Generation

TitanFuzz leverages InCoder (Fried et al., 2022) and its code-infilling capabilities to generate diverse and semantically meaningful mutations. Unlike traditional mutation techniques, which often produce invalid programs due to tensor shape mismatches or incorrect API arguments, InCoder uses bidirectional context to generate code that better preserves compatibility with surrounding operations.

1.1.6 Fitness Function

TitanFuzz uses a fitness function (Deng et al., 2023) to prioritize seed programs that exhibit complex and diverse API interactions. The fitness score considers three factors: **dataflow graph depth (D)**, representing the longest dependency path, **the number of unique API calls (U)**, and **the number of repeated API calls with identical inputs (R)**. The fitness function is defined as:

$$\text{Fitness}(C) = D + U - R$$

By rewarding deeper dataflow dependencies and diverse API usage while penalizing redundant calls, TitanFuzz prioritizes promising seed programs for subsequent mutation rounds, increasing the likelihood of discovering bugs.

1.1.7 Oracle

After generating test programs, TitanFuzz employs differential testing (Deng et al., 2023) to identify potential bugs by executing the same programs on both CPU and GPU backends. It focuses on two types of bugs: **wrong computation** and **crashes**. Wrong-computation bugs are detected by comparing intermediate variable values across backends, using a tolerance threshold to account for numerical differences. Crashes, such as segmentation faults, unexpected aborts, and internal assertion failures, are also monitored.

2 Implementation

We conduct a reproduction of TitanFuzz (Deng et al., 2023) and compare it with DeepREL (Deng et al., 2022) on PyTorch. Unlike the original TitanFuzz study, which evaluated PyTorch 1.12 and TensorFlow 2.10, our study focuses on PyTorch 2.6.0 with Python 3.11. We use the original TitanFuzz evolutionary search with observation hooks and DeepREL’s native relation verification and argument mutation.

The experiments run as isolated Slurm jobs with one NVIDIA H200 GPU and eight CPU cores allocated per shard. Both fuzzers use the same allocation policy. We select 30 APIs by drawing five from each of six predefined groups (element-wise, reduction/statistics, shape, indexing, linear algebra, and neural-network operations) in a 60-API sampling frame. API sampling uses random seed 20261019. Each API is tested with ten paired repetition seeds

(2026101901–2026101910), giving $30 \times 10 \times 2 = 600$ tool-API-repetition units. Pair ordering is balanced, and each search has a 60-second active fuzzing budget.

2.1 Seed Programs and Tool Configuration

For each target API, we use `codex-cli0.163.0` to write three seed programs: a basic `float32` invocation, a `float64` variant, and a case using a non-contiguous tensor constructed by stacking and slicing. This yields 90 seed programs. Their expected behaviors are checked against independent expressions derived from the PyTorch API specifications before fuzzing. For example, one of our `torch.matmul` cases multiplies

```
a = torch.tensor([[2., -1.], [3., 4.]])
b = torch.tensor([[1., 2.], [-2., 1.]])
y = torch.matmul(a, b)
```

for which the independently computed result is `[[4., 3.], [-5., 10.]]`. This expectation follows from matrix multiplication, not from copying the current implementation’s output.

TitanFuzz uses the local `facebook/incoder-1B` model (Fried et al., 2022) for code infilling. The configuration uses batch size 4, temperature 1.0, top- $p = 0.95$, and a 50-token infilling limit. Seed selection uses the fitness function with a pool size of ten, mutation operators are selected by Thompson Sampling.

DeepREL converts seed programs into invocation records with its original instrumentation and preserves the official argument definitions, similarity ranking, and mutation pools (Deng et al., 2022). It considers the ten highest-ranked candidate API relations, applies native verification, and generates mutation batches of 100 for accepted relations until the deadline. When no relation is accepted, a run may terminate before 60 seconds. We confirm agreement in initial tensor shape, dtype, and argument metadata. However, DeepREL’s tensor serialization can reconstruct numerical values and discard non-contiguous strides. Consequently, the experiment controls the starting metadata, not identical concrete tensor contents and layouts for every execution. DeepREL’s working database is reset between runs to avoid cross-run information leakage.

2.2 Metrics

Following the evaluation criteria discussed by Deng et al. (2023), we assess the effectiveness of each fuzzing method in terms of code coverage, operator diversity, test validity, program diversity, bug detection, and computational cost. To ensure a comparable evaluation budget, our primary metrics are computed from fixed-size test suites that are replayed independently of the fuzzers.

Evaluation Protocol. For each method m , target API a , and repetition r , we sample up to 50 generated test programs without replacement. Let $S_{m,a,r}$ denote this set of raw candidates. We independently execute each candidate and classify it as valid only if it completes successfully and invokes the intended target API at least once. The subset of valid programs is denoted by $V_{m,a,r} \subseteq S_{m,a,r}$.

Additional Python Line Coverage. Our primary metric measures how many previously uncovered PyTorch Python source-code lines are exercised by the generated tests. For a target API a , let B_a denote the common set of manually authored seed programs, and let $L(p)$ represent the set of PyTorch Python source-line locations executed by program p . Each location is identified by its source file and line number.

We define additional line coverage as

$$\Delta L_{m,a,r} = \left| \left(\bigcup_{p \in V_{m,a,r}} L(p) \right) \setminus \left(\bigcup_{b \in B_a} L(b) \right) \right|. \quad (1)$$

This metric counts distinct executable lines reached by valid generated tests but not by the common seed baseline.

ATen Operator Diversity. In addition to Python-level coverage, we measure the diversity of internal PyTorch tensor operations exercised by the generated tests.

ATen (A Tensor Library) is a core tensor-operation library underlying PyTorch (PyTorch Contributors, a). Many high-level Python APIs ultimately invoke operators registered in PyTorch’s internal `aten::` namespace (PyTorch Contributors, b). For example, a two-dimensional `torch.matmul` operation may invoke `aten::mm`, while tensor addition may invoke `aten::add`.

Let $O(p)$ denote the set of distinct `aten::` operator names observed during the execution of program p . We define additional operator diversity as

$$\Delta O_{m,a,r} = \left| \left(\bigcup_{p \in V_{m,a,r}} O(p) \right) \setminus \left(\bigcup_{b \in B_a} O(b) \right) \right|. \quad (2)$$

The metric counts operator names that are exercised by valid generated tests but absent from the seed baseline. Each distinct operator is counted once, regardless of how frequently it is invoked. A higher value indicates that the fuzzer explores a broader range of internal tensor operations.

Valid Test Yield. We evaluate the ability of each fuzzer to produce executable tests under a fixed candidate budget. The valid yield is defined as

$$Y_{m,a,r} = \frac{|V_{m,a,r}|}{50}. \quad (3)$$

A test contributes to the numerator only if it executes successfully and calls the target API.

Program Diversity. We additionally measure the number of structurally distinct valid programs within each replayed sample. Two programs are considered structurally identical when their parsed Python abstract syntax trees (ASTs) are identical, regardless of irrelevant formatting differences such as whitespace (Python Software Foundation).

Let $AST(p)$ denote the parsed syntax tree of program p . The number of AST-distinct valid programs is

$$D_{m,a,r} = |\{AST(p) \mid p \in V_{m,a,r}\}|. \quad (4)$$

This metric helps distinguish fuzzers that repeatedly generate similar programs from those that explore different program structures. However, syntactic diversity does not necessarily imply distinct execution behavior or higher code coverage.

3 Experiments

3.1 Test-Suite Quality and Coverage

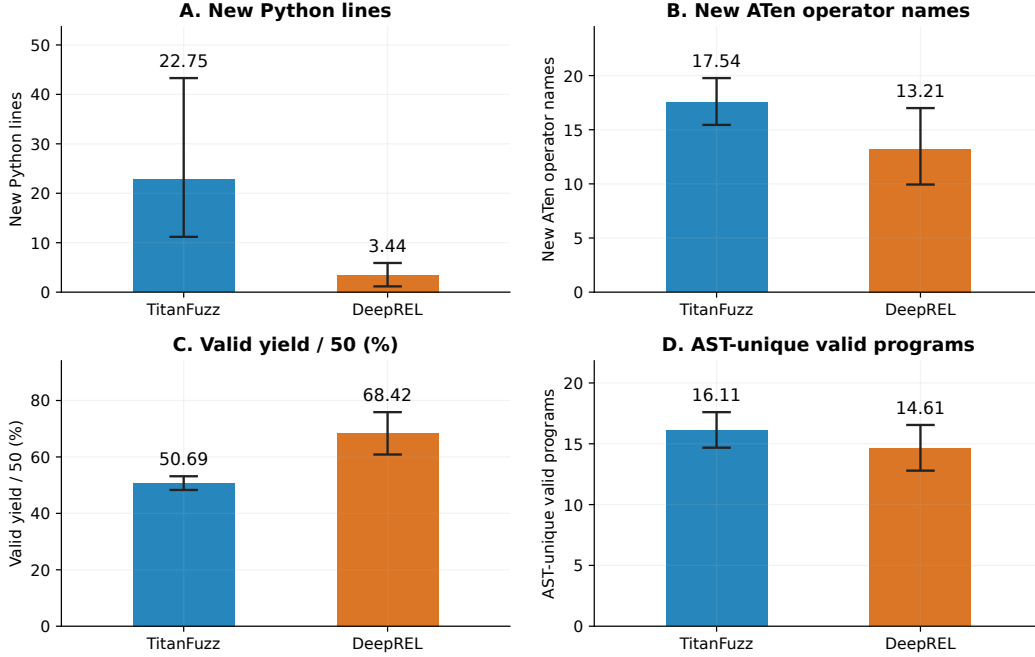
Table 1 and Figure 3 summarize the complete 30-API comparison. TitanFuzz reaches an average of 22.75 new PyTorch Python lines per API-repetition unit, versus 3.44 for DeepREL. TitanFuzz also exercises more previously unseen ATen operator names (17.54 versus 13.21). In contrast, DeepREL achieves a higher valid yield: 68.42% versus 50.69%, a TitanFuzz–DeepREL difference of -17.73 percentage points. Thus, the LLM-based approach explores more code and operator types in this bounded setting, whereas the relation-based approach produces a larger proportion of executable test cases.

The line-coverage distribution is strongly skewed: many units add no Python lines, whereas a few show substantial gains. For example, one TitanFuzz run targeting `relu` reaches 1,602 new lines. Consequently, the mean improvement does not represent the typical gain for every API. As shown in Figure 4, category-level effects vary. For example, the reduction/statistics category has an estimated line difference of only $+0.38$ with an interval spanning zero, while the neural-network category has a much larger but highly uncertain mean difference.

Table 1: Equally weighted results across 30 APIs and ten paired repetitions. Differences are TitanFuzz minus DeepREL; 95% intervals use a stratified hierarchical bootstrap.

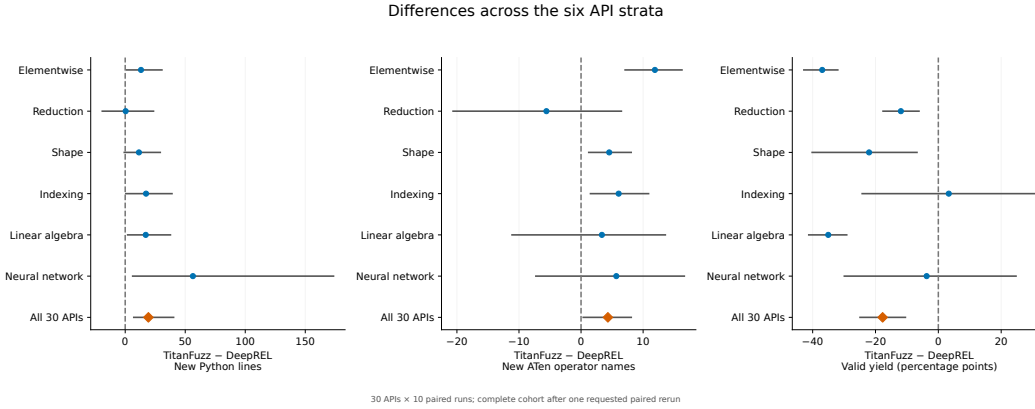
Metric (per API/run)	TitanFuzz	DeepREL	Difference [95% CI]
New Python lines	22.75	3.44	+19.31 [7.15, 40.31]
New ATen operator names	17.54	13.21	+4.33 [0.35, 8.09]
Valid yield (%)	50.69	68.42	-17.73 [-24.92, -10.48] pp

Overall test-suite quality — equally weighted APIs



30 APIs × 10 paired runs; complete cohort after one requested paired rerun

Figure 3: Measured quality of up to 50 raw proposals per API and repetition. Error bars show hierarchical 95% bootstrap intervals; program counts refer to independently replayed candidates, not the full generated corpus.



30 APIs × 10 paired runs; complete cohort after one requested paired rerun

Figure 4: Paired TitanFuzz–DeepREL effects by API category. Horizontal bars denote 95% hierarchical bootstrap intervals. An interval crossing zero does not establish a consistent advantage for that category.

3.2 Original-Paper Metrics and Run-to-Run Variation

To connect our measurements with the original paper, we separately aggregate the union of executable Python lines and deduplicated valid tests over all 30 APIs for each repetition, overlapping line locations are not counted multiple times. Table 2 gives the mean and sample standard deviation across ten runs. TitanFuzz covers more target APIs and records more distinct valid sampled programs per repetition, but Python line coverage has substantial run-to-run variation.

Table 2: Paper-aligned secondary measurements (mean $\pm$ sample SD over ten repetitions, pooling 30 APIs within each repetition). Valid programs are from independent replay, not the complete search corpus.

Metric per repetition	TitanFuzz	DeepREL
Covered target APIs (out of 30)	30.00 $\pm$ 0.00	28.90 $\pm$ 0.32
Unique valid sampled programs	483.30 $\pm$ 57.52	438.40 $\pm$ 12.91
Python line coverage (%)	0.1282 $\pm$ 0.1498	0.0392 $\pm$ 0.0008
Search time (s; 30 APIs)	1800.10 $\pm$ 0.08	1680.67 $\pm$ 0.41

3.3 Bug Oracle and Assertion Reliability

The CPU/GPU oracle flags 50 mismatch occurrences in the TitanFuzz replay, all associated with `torch.topk`; these correspond to 37 distinct program ASTs. A specification-based review finds that all 37 are compatible with permitted tie-breaking or output-order behavior, consistent with the documented instability of indices for tied values in `torch.topk` (PyTorch Contributors, c). Therefore, *neither fuzzer yields a confirmed new PyTorch bug* in this study. These are rejected warnings, not 37 new bugs, and this observation does not establish a whole-library false-positive rate or the absence of defects.

Manual inspection also exposes weaknesses in generated assertions. For an `index_select` example that selects columns [2, 0] from `[[1, 2, 3], [4, 5, 6]]`, the specification requires `[[3, 1], [6, 4]]`. One generated candidate instead compares its result with an incorrect, differently shaped tensor using `torch.equal` but discards the returned Boolean. Successful execution therefore cannot be interpreted as a passed assertion. A separate generated `conv2d` case uses the tautology `assert torch.allclose(y, y)` and fails earlier because of undefined variables.

4 Conclusion

TitanFuzz (Deng et al., 2023) offers several notable advantages. First, it eliminates the need to manually specify API constraints by leveraging the knowledge embedded in large language models. Second, it is capable of generating complex test programs involving multiple API calls, enabling the exploration of interactions between different APIs. Third, it integrates both generation-based and mutation-based fuzzing techniques, allowing for the generation of initial seed programs and their subsequent evolution.

Despite these strengths, TitanFuzz also has several limitations. First, LLM-generated programs tend to follow common programming patterns, potentially limiting the exploration of rare edge cases. Second, TitanFuzz lacks direct coverage-guided feedback, as its fitness function primarily evaluates structural properties rather than actual code coverage (Deng et al., 2023). Third, invalid generated programs may introduce substantial computational overhead, reducing the overall efficiency of the fuzzing process.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng.

- TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283. USENIX Association, 2016. URL <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in Neural Information Processing Systems*, volume 24, pages 2249–2257, 2011. URL <https://proceedings.neurips.cc/paper/2011/hash/e53a0a2978c28872a4505bdb51db06dc-Abstract.html>.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL <https://arxiv.org/abs/2107.03374>.
- Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022. doi: 10.1145/3540250.3549085. URL <https://doi.org/10.1145/3540250.3549085>.
- Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 423–435, 2023. doi: 10.1145/3597926.3598067. URL <https://doi.org/10.1145/3597926.3598067>.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022. URL <https://arxiv.org/abs/2204.05999>.
- Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1418–1430, 2022. doi: 10.1145/3510003.3510092. URL <https://doi.org/10.1145/3510003.3510092>.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html>.
- Python Software Foundation. ast — abstract syntax trees. Python 3.11 Documentation. URL <https://docs.python.org/3.11/library/ast.html>. Accessed: 2026-10-18.
- PyTorch Contributors. ATen: Tensor library. PyTorch C++ Documentation, a. URL <https://docs.pytorch.org/cppdocs/api/aten/index.html>. Accessed: 2026-10-18.
- PyTorch Contributors. Extending PyTorch: The `__torch_dispatch__` mechanism. PyTorch Documentation, b. URL <https://docs.pytorch.org/docs/stable/notes/extending.html>. Accessed: 2026-10-18.
- PyTorch Contributors. torch.topk. PyTorch API Documentation, c. URL <https://docs.pytorch.org/docs/stable/generated/torch.topk.html>. Accessed: 2026-10-18.