Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries

Yinlin Deng
University of Illinois
Urbana-Champaign
yinlind2@illinois.edu

Chunqiu Steven Xia
University of Illinois
Urbana-Champaign
chunqiu2@illinois.edu

Chenyuan Yang
University of Illinois
Urbana-Champaign
cy54@illinois.edu

Shizhuo Dylan Zhang
University of Illinois
Urbana-Champaign
shizhuo2@illinois.edu

Shujing Yang
University of Illinois
Urbana-Champaign
shujing6@illinois.edu

Lingming Zhang
University of Illinois
Urbana-Champaign
lingming@illinois.edu

ABSTRACT

Bugs in Deep Learning (DL) libraries may affect almost all downstream DL applications, and it is crucial to ensure the quality of such systems. It is challenging to generate valid input programs for fuzzing DL libraries, since the input programs need to satisfy both the syntax/semantics of the supported languages (e.g., Python) and the tensor/operator constraints for constructing valid computational graphs. Recently, the TitanFuzz work demonstrates that modern Large Language Models (LLMs) can be directly leveraged to implicitly learn all the language and DL computation constraints to generate valid programs for fuzzing DL libraries (and beyond). However, LLMs tend to generate ordinary programs following similar patterns/tokens with typical programs seen in their massive pre-training corpora (e.g., GitHub), while fuzzing favors *unusual* inputs that cover edge cases or are unlikely to be manually produced.

To fill this gap, this paper proposes FuzzGPT, the first approach to priming LLMs to synthesize unusual programs for fuzzing. FuzzGPT is mainly built on the well-known hypothesis that historical bug-triggering programs may include rare/valuable code ingredients important for bug finding. Meanwhile, while traditional techniques leveraging such historical information require intensive human efforts to both design dedicated generators and ensure the syntactic/semantic validity of generated programs, FuzzGPT demonstrates that this process can be *fully automated* via the intrinsic capabilities of LLMs (including fine-tuning and in-context learning), while being generalizable and applicable to challenging domains. While FuzzGPT can be applied with different LLMs, this paper focuses on the powerful GPT-style models: Codex and CodeGen. Moreover, FuzzGPT also shows the potential of directly leveraging the instruction-following capability of the recent ChatGPT for effective fuzzing. The experimental study on two popular DL libraries (PyTorch and TensorFlow) shows that FuzzGPT can substantially outperform TitanFuzz, detecting 76 bugs, with 49 already confirmed as previously unknown bugs, including 11 high-priority bugs or security vulnerabilities.

ACM Reference Format:
Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623343

1 INTRODUCTION

Deep Learning (DL) has been widely adopted in various application domains, including scientific discovery [52], healthcare [6], finance [63], and transportation [56]. Such DL applications are commonly constructed using DL libraries (e.g., PyTorch [49] and TensorFlow [62]) where developers utilize library APIs to build, train, and deploy DL models. Similar to any other complicated software systems, DL libraries can also be buggy. Moreover, bugs in DL libraries can cause serious consequences as they can potentially affect almost all downstream DL applications, including safety-critical ones [42, 54, 81]. As a result, it is crucial to ensure the quality of DL libraries.

Fuzzing [5, 61, 80], a powerful methodology for bug finding via random input generation, has been widely studied for testing DL libraries in recent years. Meanwhile, it is extremely challenging to generate arbitrary input programs for DL libraries, since the programs need to satisfy both the syntax/semantics of the supported languages (such as Python with dynamic typing) and the tensor/-operator constraints for constructing valid computational graphs. For example, in multiplication operations, two tensors must have matching dimensionality. To simplify the problem, prior DL library fuzzing techniques mainly work on model-level [23, 25, 34, 48, 68] or API-level fuzzing [14, 69, 73, 75]. Model-level fuzzers either reuse/mutate existing seed models [25, 48, 68], or generate DL models from scratch [23, 34]. Due to the intricate tensor/operator constraints, such model-level fuzzers either only focus on manipulating shape-preserving APIs [68] or require manually-written specifications for each API [23, 34] to preserve model validity. As a result, they can only cover limited DL APIs and program patterns. On the other hand, API-level fuzzers focus on testing each individual API via effective input generation [69, 73] or oracle inference [14, 75]. While API-level

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623343

fuzzers can easily cover a large number of APIs, they cannot find any bugs arising from interactions of different DL APIs.

With the recent enormous advances in Large Language Models (LLMs), TitanFuzz [13] has been proposed to directly leverage LLMs for fuzzing DL libraries and beyond. The key insight is that LLMs are pre-trained on billions of code snippets in different languages from open source, which can include numerous valid DL programs for popular DL libraries; in this way, LLMs can implicitly learn both the language syntax/semantics and the tensor/operator constraints for valid DL computations. TitanFuzz has been shown effective in generating *valid* input DL programs and substantially outperforms both traditional model-level and API-level fuzzers. More importantly, compared with traditional fuzzing techniques that require intensive human efforts for building generation/mutation strategies [16, 28, 31, 32, 60, 76, 82], TitanFuzz is fully automated, and can be easily generalized to different application domains and programming languages. Meanwhile, TitanFuzz directly leverages the generative capability of LLMs, which is based on *token naturalness* [26] and aims to resemble what they saw in the training corpora. In this way, TitanFuzz can easily generate ordinary *human-like* DL programs.

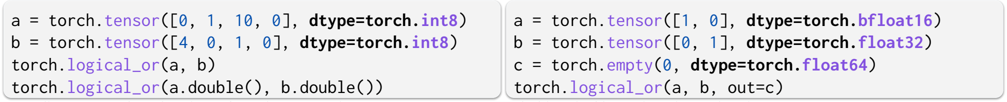
However, such ordinary programs can only cover a limited set of common/standard DL library behaviors, which may not be important or interesting for testing. In contrast, *unusual programs* exhibit less common behaviors and are more likely to cover code paths that are not sufficiently tested, potentially revealing new bugs or vulnerabilities. For instance, unusual programs may construct edge-case inputs or use unconventional parameter combinations and API sequences. Compared to Figure 1a, Figure 1b shows a historical bug-triggering code for `logical_pr`, where instead of using the same `dtype`, a bug occurs when the `dtype`s are different. Such non-standard (unusual) code snippets can be difficult for pre-trained LLMs to directly generate as it does not conform to the large amounts of well-formed code snippets seen during pre-training.

Our Work. This paper proposes FuzzGPT, the first approach to guiding LLMs to directly synthesize *unusual* input programs for effective fuzzing. *While the recent TitanFuzz work samples programs from the natural probability distribution encoded in pre-trained LLMs, FuzzGPT aims to shift the distribution towards unusual programs in the search space to explore code paths rarely hit by ordinary programs.* FuzzGPT is mainly built on the well-known hypothesis that historical bug-triggering programs may include edge-case/valuable code ingredients important for bug finding. In the literature, researchers have proposed various techniques to recompose such code ingredients or insert them into new code contexts for exposing new interesting bugs/vulnerabilities [9, 28, 59, 84]. However, such techniques require intensive human efforts to both design such dedicated generators and ensure the syntactic/semantic validity of the generated programs. For example, according to prior work [28, 84], even resolving the very common undeclared-identifier issue can be non-trivial. Moreover, it is hard to generalize such techniques to different domains, not to mention the challenging DL library fuzzing problem. In contrast, our key insight is that recent advanced LLMs offer a natural, generalizable, and fully automated solution for leveraging such historical programs – they can be easily *prompted* [51] or *fine-tuned* [50] to digest such historical programs, and then generate more unusual programs that resemble the historical ones and effectively exploit their code ingredients. Compared with traditional techniques on leveraging such historical information, FuzzGPT can *implicitly* learn all the generation constraints (including language syntax/semantics, DL computation constraints, and the new unusualness constraints), and is fully automated. Moreover, while this paper focuses on the challenging problem of DL library fuzzing, the FuzzGPT idea is generalizable to other application domains or programming languages (e.g., testing/fuzzing for various compilers/interpreters, DB systems, SMT solvers, or any software libraries with accessible APIs).

To implement FuzzGPT, we first construct a dataset of bug-triggering code snippets by mining bug reports from open-source repositories of the target DL libraries. Built on this dataset, FuzzGPT includes the following strategies. **(1) In-context learning** [37]: we provide LLMs with either a few examples of historical bug-triggering programs (few-shot learning [7, 51]) or a partial bug-triggering program (zero-shot learning [47, 58]) to either generate a new code snippet or to autocomplete the partial code. **(2) Fine-tuning** [50]: we modify the model weights by training on the extracted historical bug-triggering programs to obtain fine-tuned LLMs that are specially designed to generate similar bug-triggering code snippets. From both learning strategies, FuzzGPT can *prime* the LLMs to generate bug-triggering programs by capturing code ingredients within either the local context examples or fine-tuning dataset.

To summarize, this paper makes the following contributions:

- **Dimension.** This paper opens up a new direction for generating *unusual* input programs for effective fuzzing via LLMs. This paper is the first to show that LLMs can be easily prompted/fine-tuned to resemble historical bug-triggering programs or even directly follow human instructions to generate unusual programs for fuzzing real-world systems. Compared with traditional fuzzers for unusual program generation, FuzzGPT is fully automated, generalizable, and applicable to challenging application domains (especially for software systems with accessible APIs).
- **Technique.** While our idea is generalizable, we have implemented FuzzGPT as an LLM-based fuzzer for DL libraries in this paper. We implement three variants of FuzzGPT based on in-context learning and fine-tuning: 1) few-shot learning: a few examples of previous bug-triggering code snippets are provided, 2) zero-shot learning: a partially complete bug-triggering program is given, and 3) fine-tuning: training a specialized LLM via learning bug-ingredients from the historical programs. While FuzzGPT can be applied with different LLMs, we build our strategies based on state-of-the-art proprietary and open-source GPT-style LLMs for code, Codex [10] and CodeGen [43]. Moreover, we also build a specific zero-shot FuzzGPT variant by directly leveraging the instruction-following capability of ChatGPT [45] without any historical information.
- **Extensive Study.** We study all FuzzGPT variants on two popular DL libraries (PyTorch [49] and TensorFlow [62]). Our results show that FuzzGPT achieves 60.70%/36.03% higher coverage than state-of-the-art TitanFuzz on PyTorch/TensorFlow. Overall, FuzzGPT found 76 bugs on the latest versions of PyTorch and TensorFlow. 49 have already been confirmed as new bugs, with 11 high-priority bugs or security vulnerabilities.



```
a = torch.tensor([0, 1, 10, 0], dtype=torch.int8)
b = torch.tensor([4, 0, 1, 0], dtype=torch.int8)
torch.logical_or(a, b)
torch.logical_or(a.double(), b.double())
```

a) ordinary example code snippet from documentation

```
a = torch.tensor([1, 0], dtype=torch.bfloat16)
b = torch.tensor([0, 1], dtype=torch.float32)
c = torch.empty(0, dtype=torch.float64)
torch.logical_or(a, b, out=c)
```

b) historical bug-triggering code snippet

Figure 1: Example code snippets. a) shows an *ordinary* code snippet where `logical_pr` is used in a very standard way by initializing two tensors with the same `dtype`. This code snippet is taken from the API documentation. b) shows an *unusual* code snippet for `logical_pr`, where instead of using the same `dtype`, a bug occurs when the `dtype`s are different. This code snippet is taken from a historical bug report.

2 BACKGROUND AND RELATED WORK

2.1 Large Language Model

Large Language Models (LLMs) have demonstrated impressive performance across a wide range of NLP tasks by training on large corpora of text data scraped from the Internet [20]. Recently, researchers have begun adopting LLMs for code-related tasks, e.g., via further fine-tuning LLMs on code snippets from open-source repositories [17]. LLMs can be classified based on variations of the popular Transformer architecture [64] into: *Encoder-only*, *Decoder-only* and *Encoder-Decoder* models. Decoder-only LLMs (e.g., GPT [7, 46], Codex [10] and CodeGen [43]) focus on autoregressive completion tasks by learning to predict the probability of the next token given previously generated tokens. Encoder-only (e.g., CodeBERT [17] and GraphCodeBERT [24]) and Encoder-Decoder (e.g., CodeT5 [79] and PLBART [4]) models on the other hand are designed for infilling tasks, where the pre-training objective is to recover masked-out tokens or token spans in the training data by using bi-directional context.

In order to apply LLMs for down-stream applications, there are two main paradigms: *Fine-tuning* [50] and *In-context learning* [7, 51]. Fine-tuning is the process of updating the model weights by learning the desired output from the given input of a specific downstream task dataset. The resulting fine-tuned LLM can be treated as specialized models designed to perform a particular task such as code summarization [4] and program repair [72]. Different from fine-tuning, which typically requires large downstream datasets to update the model, in-context learning only requires a few examples/demonstration of the downstream task. In-context learning directly uses the pre-trained LLM without any weight modification by constructing an input which contains multiple (i.e., few-shot) examples of input/output demonstrations and then the final task to be solved. Through looking at the context, LLMs can directly learn the goal of the specific task and expected output format without fine-tuning. Various techniques have been proposed to improve in-context learning, in particular via example selection [35, 53], and via multi-step reasoning using Scratchpad [44] or Chain-of-Thought [70]. Together, in-context learning and fine-tuning are two different approaches to prime LLMs for downstream tasks.

While the approach of FuzzGPT is general for different LLMs, in this paper we mainly focus on the recent powerful GPT-style models: Codex [10] and CodeGen [43], which are state-of-the-art proprietary and open-source models for code, respectively. Moreover, we also directly leverage the instruction-following capability of ChatGPT [45] for fuzzing.

2.2 Fuzzing with Historical Bugs

Fuzzing with historical bugs has been extensively studied for over a decade. One of the pioneering techniques is LangFuzz [28]: LangFuzz first parses historical tests which have been known to cause incorrect behaviors and extracts individual code fragments; then, LangFuzz will generate code and further replace partial code with extracted code fragments. Researchers have also mined fuzzer configurations (e.g., grammar/statement probabilities) from historical bug-triggering programs for more diverse generation [9, 59]. More recently, JavaTailor [84] leverages extracted code ingredients from historical bugs to synthesize new input programs for Java Virtual Machine (JVM) fuzzing. Besides such generation-based or hybrid approaches, mutation-based fuzzers have also widely utilized such prior bug reports or regression tests as high-quality input seeds for future mutations, targeting C compilers [31], SMT solvers [71], and database systems [66, 86]. Moreover, a recent study [85] shows that directly reusing code snippets with minimal modification from historical bug reports can also find interesting bugs in other C/C++ compilers.

FuzzGPT directly learns from historical bug-triggering code via fine-tuning and in-context learning with LLMs. Unlike prior generation- or mutation-based techniques which require extensive human efforts to build dedicated generators and ensure syntactic/semantic validity of generated programs [28, 84], FuzzGPT is fully automated and generalizable. Moreover, it is extremely challenging to build such traditional generators/mutators for DL programs due to the complicated language and tensor/operator constraints, while FuzzGPT can implicitly learn all such constraints using modern LLMs.

2.3 Fuzzing via Deep Learning

In addition to traditional fuzzing approaches, researchers have also developed fuzzing tools based on Deep Learning (DL) techniques. SeqFuzzer [83] is a network protocol fuzzer built on Long Short-term Memory (LSTM) [27], a Recurrent Neural Network (RNN) [11]. RNN-based fuzzers have also been used for fuzzing other systems such as OpenCL compilers [12], C compilers [38], and PDF file parsers [22]. Likewise, Montage [33] has been proposed to fuzz JavaScript engines by directly training the tree-based RNN to mutate existing seed ASTs. Moreover, COMFORT [77] proposes to fine-tune the pre-trained GPT-2 model to generate JavaScript programs, and further relies on additional heuristics to generate inputs for the synthesized programs. While the above work leverages the development of DL models for fuzzing, they did not yet study modern LLMs for code. Recently, TitanFuzz [13] demonstrates for the first time that modern LLMs can be directly leveraged to perform end-to-end fuzzing of real-world systems, and can simulate both generation- and mutation-based fuzzing studied for decades. TitanFuzz first leverages Codex [10] to generate high-quality seed programs and then leverages InCoder [19] with an evolutionary fuzzing strategy to generate diverse code snippets. TitanFuzz has demonstrated state-of-the-art results for fuzzing DL libraries and can find bugs that can only be uncovered with complex API sequences. However, TitanFuzz is not specifically designed to generate unusual programs, nor does it utilize historical bugs to guide the fuzzing process.

In this work, we propose to leverage the historical bug-triggering code information to further guide LLMs towards more effective fuzzing. To our knowledge, this is the first work demonstrating that LLMs can easily perform history-driven fuzzing (widely studied for over a decade), while being fully automated, generalizable, and applicable to the challenging domain of DL library fuzzing. Our work differs from the recent TitanFuzz work in terms of both the overall approach/idea and the concrete techniques. TitanFuzz directly leverages LLMs to generate valid programs resembling the training corpora, where the vast majority would be common usages. In contrast, FuzzGPT either prompts or fine-tunes LLMs to resemble the historical bug-triggering programs, which would be extremely rare in the training data and important for testing/fuzzing.

3 APPROACH

In this section, we describe our FuzzGPT approach of exploiting historical bugs via Large Language Models (LLMs) to automatically fuzz DL libraries. The key idea of FuzzGPT is to use LLMs and directly *learn* from historical reported bugs to generate similar *bug-triggering* code snippets to find new bugs. Existing work along this direction requires extensive human efforts, and can hardly generalize to the challenging domain of DL library fuzzing. In contrast, the recent advances in LLMs offer a natural, generalizable, and fully automated solution – modern LLMs can be easily prompted or fine-tuned to digest such historical programs and then generate programs that resemble the historical ones via effectively exploiting their code ingredients.

Figure 2 shows the overview of FuzzGPT. We first systematically mine bug reports from the target DL library repositories to collect historical bug-triggering code snippets (Section 3.1). As FuzzGPT aims to target specific DL library APIs, each bug-triggering code snippet requires a corresponding buggy API label. However, oftentimes the exact buggy API may not be explicitly indicated within the bug report. As such, FuzzGPT employs a self-training approach to automatically generate buggy API labels using LLMs by prompting with a few manually labeled examples. Next, using these pairs of extracted bug-triggering code snippets and buggy API labels, FuzzGPT can start the fuzzing procedure to generate *edge-case* code snippets (Section 3.2). In this work, we investigate two learning methods: **1) In-context learning:** we prompt the pre-trained LLM directly with either examples of bug-triggering code and buggy API pairs (few-shot), or a partial/complete bug-triggering code (zero-shot), and let the model generate/edit programs for a target API, **2) Fine-tuning:** using the extracted dataset, we fine-tune the LLM to learn from the bug-triggering code examples and patterns to generate new bug-triggering code snippets for a target API. Finally, the generated programs are executed with test oracles (e.g., CPU/GPU [69], or automatic differentiation oracle [75]) for bug detection (Section 3.3).

While FuzzGPT is general for any generative LLMs, in this paper, we focus on two specific LLMs: Codex [10] and CodeGen [43]. Codex is a state-of-the-art generative model fine-tuned on open-source code repositories initialized from the GPT-3 [7] model weights. Unlike Codex whose model weights and training data are not released, CodeGen is an open-source generative model. Since we target DL library APIs exposed in Python, we use the Python version of CodeGen, fine-tuned on Python GitHub code [43]. In FuzzGPT, we directly use both Codex and CodeGen models – the larger Codex model allows us to show the full potential of fuzzing when using LLMs, while the open-source CodeGen models can be used to evaluate the scaling effect and test out fine-tuning strategies. We next describe each step in FuzzGPT in more detail.

3.1 Dataset Construction

To facilitate learning from historical bugs, FuzzGPT requires a bug-triggering code dataset which contains pairs of bug-exposing code snippet and its corresponding buggy DL library API.

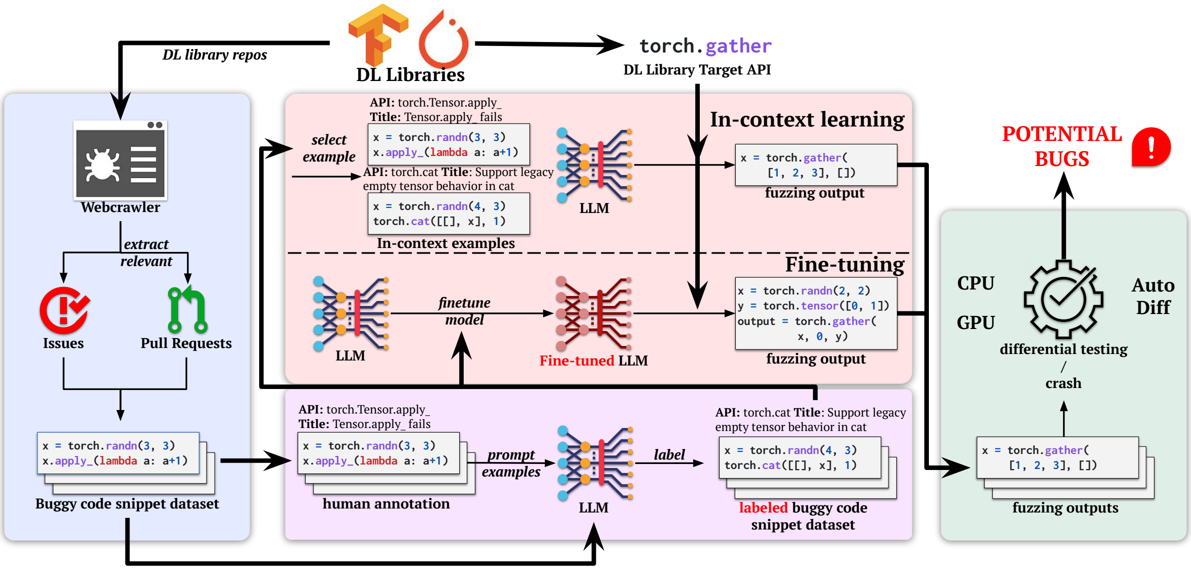
3.1.1 *Mining Bug History from GitHub.*

First, we implement an HTML crawler to collect all the issues and pull requests (PRs) from the GitHub issue-tracking systems of our target libraries. Then, we focus on identifying bug-triggering code snippets from two sources: (1) Issues associated with accepted or pending PRs. We search in these bug reports for all code blocks, which contain code snippets to reproduce the bug, and concatenate them together. (2) PRs containing code blocks in their commit messages. We further consider PRs because some PRs may fix bugs without corresponding issues. For each extracted issue or PR, we extract its title as well and include that in prompts for few-shot learning and fine-tuning. After mining bug-triggering code snippets, we will next perform automated annotation to further label each code snippet with a corresponding buggy API.

3.1.2 *Automated Buggy API Annotation.*

Each code snippet in our dataset often involves multiple DL APIs. As such, the exact buggy API cannot be directly extracted. In order to annotate the buggy API for each bug-triggering code example, we propose a self-training [55, 87] approach where we feed a few-shot prompt (shown in Figure 3) to LLM and use the model completion as the buggy API. We first provide manual annotation of the buggy API name for several randomly chosen bug-triggering code snippets. These manually annotated pairs become the few-shot examples used as part of the input to the LLM. In Figure 3, the few-shot examples are constructed with the chosen bug-triggering code snippet, the extracted title of the corresponding issue/pull request (e.g., `Tensor.apply_ fails`), and finally, the manually annotated buggy API name (e.g., `torch.Tensor.apply_`). Next, we combine the few-shot examples with a target bug-triggering code snippet to create a prompt, and then query the LLM to obtain the predicted buggy API name for the given code snippet. Note that the annotation overhead is *minimal*, since we only need to annotate a few examples (6 for this work as shown in Section 4) for the targeted library due to the in-context learning capability of modern LLMs.

Our proposed method is similar to self-training [55, 87] where a classifier is first trained on a small labeled dataset and then used to label a larger unlabeled dataset. In our case, we directly use LLMs with few-shot learning from a small number of manual annotations to provide annotations for the large amounts of extracted bug-triggering code snippets in our dataset. Our main goal with labeling is to steer the LLMs to generate many programs for each targeted DL API by providing pairs of API and code examples. Note that our automated API annotation procedure may mislabel the precise buggy API in a bug-triggering program. Meanwhile, even if some issue is mislabeled with a different API that is also called in the program, the LLM can still correctly learn the basic task of generating a program that calls the target API for effective fuzzing (as further confirmed by our experimental results in Section 6.3.4).



```
API: torch.Tensor.apply_
Title: Tensor.apply_ fails
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
API: torch.cat Title: Support legacy
empty tensor behavior in cat
x = torch.randn(4, 3)
torch.cat([[], x], 1)
```

```
x = torch.gather(
    [1, 2, 3], [])
```

```
x = torch.randn(2, 2)
y = torch.tensor([0, 1])
output = torch.gather(
    x, 0, y)
```

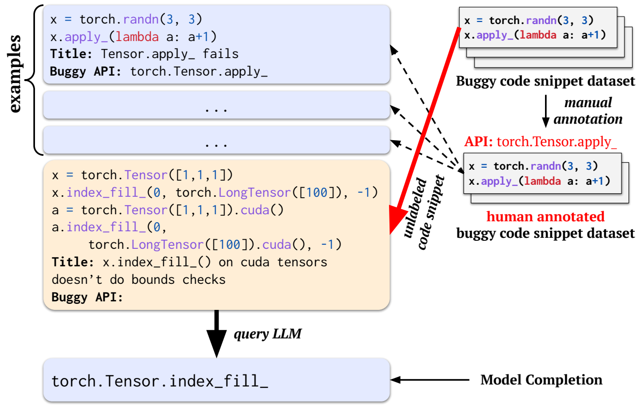
```
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

```
API: torch.Tensor.apply_
Title: Tensor.apply_ fails
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

```
API: torch.cat Title: Support legacy
empty tensor behavior in cat
x = torch.randn(4, 3)
torch.cat([[], x], 1)
```

```
x = torch.gather(
    [1, 2, 3], [])
```

Figure 2: Overview of FuzzGPT.



```
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
Title: Tensor.apply_ fails
Buggy API: torch.Tensor.apply_
...
...
x = torch.Tensor([1,1,1])
x.index_fill_(0, torch.LongTensor([100]), -1)
a = torch.Tensor([1,1,1]).cuda()
a.index_fill_(0,
      torch.LongTensor([100]).cuda(), -1)
Title: x.index_fill_() on cuda tensors
doesn't do bounds checks
Buggy API:
```

```
torch.Tensor.index_fill_
```

```
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

```
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

Figure 3: Prompt for buggy API annotation.

3.2 In-context learning and Fine-tuning

Using the extracted and annotated buggy code snippet dataset, FuzzGPT starts the process of *learning* to generate *edge-case* code. At a high level, LLM-based learning methods can be separated into two main approaches: **In-context learning** and **Fine-tuning**. In-context learning involves directly using a pre-trained LLM without adjusting any of the model parameters. Instead, the input to the LLM is prepended first with instructions and examples demonstrating the task before providing the current input. In-context learning allows the LLM to prime its output by learning not only the desired output format (e.g., provide a yes or no answer) but also the task domain (e.g., translate English to French) from the surrounding input context. Fine-tuning on the other hand aims to modify the LLM parameters through training on a specific dataset to create a specialized model.

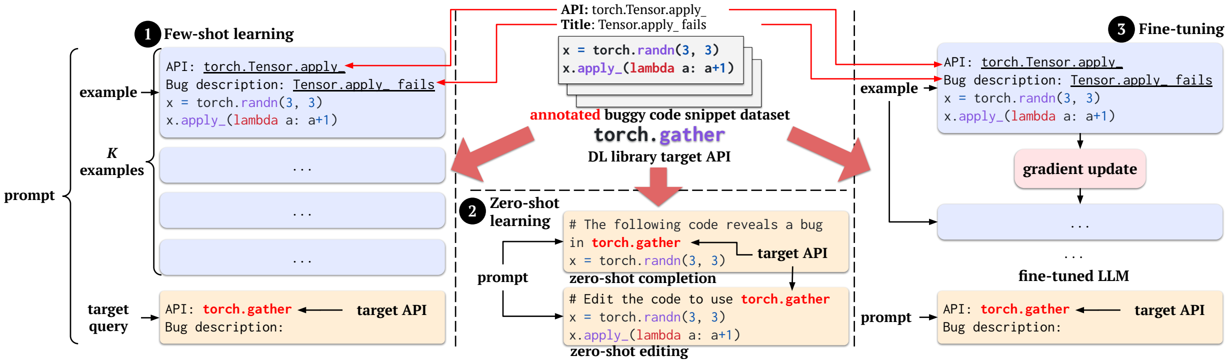
In FuzzGPT, we follow prior work [7] and systematically explore different learning settings. Specifically, we design three different learning strategies: few-shot, zero-shot, and fine-tune, each uses the annotated bug-triggering code dataset differently to generate fuzzing outputs. Figure 4 illustrates these three strategies, which are presented in more detail below.

3.2.1 *Few-shot Learning.*

We follow the classic few-shot in-context learning [7] by prepending the target query with examples from the annotated bug-triggering code snippet dataset. Figure 4 (❶) shows how the few-shot learning prompt is constructed. Our example format consists of the API name (e.g., `torch.Tensor.apply_`), bug description which is obtained from the title of the issue/PR (e.g., `Tensor.apply_ fails`), and finally the bug-triggering code snippet. The purpose of these examples is twofold. Firstly, they are intended to prime the LLM towards generating the desired output format. Secondly, they enable the model to learn to produce similar *edge-case* code snippets by observing historical bug-triggering code snippets, without having to modify the model parameters. In few-shot learning, each prompt consists of K examples (denoted as K-Shot) and the actual query (the target API and bug description header). Then, LLMs can generate new predictions based on the prompt.

Chain-of-Thought Prompting. An important component of few-shot learning is the prompt used. Our prompt design is inspired by chain-of-thought (CoT) [70] prompting, where instead of directly generating the final output, the prompt asks the model to perform the task in a step-by-step manner. Following the format of in-context examples, we first include the target API name (e.g., `API: torch.gather`) in our query, and then we ask the model to produce a description of a possible "bug" (`Bug description:`) before generating the actual "bug-triggering" code that invokes the target API. The predicted bug description provides an additional hint to the LLM, indicating that the generated code should try to cover specific potential buggy behavior.

Let $\mathcal{M}$ be the LLM that outputs the probability of generating a sequence. Let $E_K = \{< p_1, d_1, c_1 >, ..., < p_K, d_K, c_K >\}$ be the concatenation of K examples consisting of tuples of example API p_i, bug description d_i, and code snippet c_i. Let p_{target} be the target API query and d_{fs} be the bug description generated using few-shot learning. The probability of generating the final output code c_{fs} using



```
API: torch.Tensor.apply_
Bug description: Tensor.apply_ fails
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

```
API: torch.gather
Bug description:
```

```
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

```
# The following code reveals a bug
in torch.gather
x = torch.randn(3, 3)
```

```
# Edit the code to use torch.gather
x = torch.randn(3, 3)
x.apply_(lambda a: a+1)
```

Figure 4: Fine-tuning, zero-shot, and few-shot learning for DL Library Fuzzing.

few-shot learning can be formalized as this conditional probability: $\mathcal{M}(c_{\text{fs}}|E_K, p_{\text{target}}) = \mathcal{M}(c_{\text{fs}}|E_K, p_{\text{target}}, d_{\text{fs}}) \cdot \mathcal{M}(d_{\text{fs}}|E_K, p_{\text{target}})$

3.2.2 *Zero-shot Learning.*

We now describe two FuzzGPT variants under the zero-shot learning scenario:

Zero-shot Completion (default). In this variant, the input is only made up of a partial code snippet. The partial code snippet is created from historical bug-triggering code snippet dataset where we randomly remove a portion of the suffix code. Figure 4 (❷) shows how the zero-shot completion input can be created. We first include a natural language comment `# The following code reveals a bug in {target_api}` which allows us to apply zero-shot learning to any arbitrary target API. We then randomly pick an example bug-exposing code snippet from the dataset. Following, we randomly remove a portion of the selected bug-triggering code snippet's suffix and only keep the prefix lines as the input for the LLM to complete it.

Again let $\mathcal{M}$ be the LLM, c_e be the selected code snippet with the first j lines ($c_e[:j]$), and p_{comp} be the completion prompt. The probability of producing zero-shot completion code $c_{\text{zs-comp}}$ can be formalized as $\mathcal{M}(c_{\text{zs-comp}}|p_{\text{comp}}, c_e[:j])$, with $c_e[:j] + c_{\text{zs-comp}}$ being the complete fuzzing code snippet, including the partial code.

Zero-shot Editing. In this variant, the input is created from a complete code snippet in our historical bug-triggering dataset. Figure 4 (❷) also shows an example of the zero-shot editing. To begin with, we use the natural language comment `# Edit the code to use {target_api}` which indicates to the LLM that we want to perform editing and we should directly reuse a large part of the original code. Similar to zero-shot completion, we randomly select a bug-triggering code and attach it to the end of the input. Different from zero-shot completion, where the model autocompletes the end of the code snippet, editing is designed to allow the LLM to reuse all parts of historical bug-triggering code snippet to generate new fuzzing output. Let p_{edit} be the editing prompt. The zero-shot editing output $c_{\text{zs-edit}}$ can be formalized as this conditional probability: $\mathcal{M}(c_{\text{zs-edit}}|p_{\text{edit}}, c_e)$

Compared with few-shot learning, where multiple historical bugs are provided to the LLM as examples, zero-shot learning can directly use the existing code portions from historical bugs and only need to generate a partial program (completion) or replace a small portion of the code to use the new target API (editing). These concrete code lines from prior bug-triggering code can be useful to test new APIs as they can include special values (e.g., `NaN`), edge-case tensor shapes/dimensions, and other code patterns/ingredients useful for bug finding (e.g., unconventional API usages). By using zero-shot learning, FuzzGPT can directly make use of these historical bug-triggering code snippets to target additional APIs.

While TitanFuzz [13] also utilizes LLMs in a zero-shot manner, it cannot be easily modified in order to achieve the goal of this paper. TitanFuzz relies on using the InCoder model (infilling LLM) to mutate existing seed code snippets generated using Codex. However, this mutation process requires the generated seeds to be well-formed and contain vast amounts of valid DL APIs which can be extremely difficult to obtain in historical bug-triggering code snippets.

3.2.3 *Fine-tuning.*

Apart from in-context learning (few-shot and zero-shot) to learn from historical bugs which only uses the original pre-trained LLMs, fine-tuning directly modifies the model parameters by training on the historical bug code snippet dataset. Figure 4 (❸) shows how the fine-tuning procedure works and also the input to the fine-tuned model during inference time to produce the fuzzing outputs. Each training sample follows the same format as few-shot examples – made up of the API name, bug description and also the bug-triggering code snippet. We start with the original pre-trained LLM and then update the model weights through gradient descent by training to auto-regressively predict each training sample.

Let $T = \{t_1, t_2, ..., t_n\}$ be the training token sequence obtained by tokenizing the training sample, $T_{<s} = \{t_1, t_2, ..., t_{s-1}\}$ be the token sequence generated by the model so far and $\mathcal{M}$ be the LLM which outputs the probability of generated the next token given the previously generated tokens. The fine-tuning loss function is defined as: $\mathcal{L}_{\text{fine-tune}} = -\frac{1}{n}\sum_{i=1}^{n} log(\mathcal{M}(t_i|T_{<i}))$

For each DL library, we fine-tune a separate model using the bug-triggering code snippet collected from that library. By fine-tuning on these historical bug-triggering code snippets, the LLM can learn from different kinds of bug-triggering patterns/ingredients for the targeted library. The input to the fine-tuned model follows the same pattern as the few-shot approach where we construct a prompt based on the specific target API. Let $\mathcal{M}_{\text{ft}}$ be the fine-tuned LLM which outputs the probability of generating a sequence with target API prompt p_{target}. The fine-tune model output code c_{ft} can be formalized as this conditional probability: $\mathcal{M}_{\text{ft}}(c_{\text{ft}}|p_{\text{target}}) = \mathcal{M}_{\text{ft}}(c_{\text{ft}}|p_{\text{target}}, d_{\text{ft}}) \cdot \mathcal{M}_{\text{ft}}(d_{\text{ft}}|\, p_{\text{target}})$

3.3 Oracle

Using the generated fuzzing outputs, we test the DL libraries through both general and DL-specific oracles:

Crashes. We detect bugs caused by unexpected crashes found when executing the fuzzing output. These crashes can include aborts, segmentation faults, and `INTERNAL_ASSERT_FAILED`. Bugs exposed by such crashes may even further trigger security vulnerabilities.

CPU/GPU oracle. We detect wrong-computation bugs by identifying inconsistencies between the output values across two execution backends (CPU and GPU). We follow prior work [13, 69] to use a significance tolerance threshold for comparison in order to account for the non-deterministic nature of particular library APIs when executed on different backends.

Automatic Differentiation (AD) oracle. We further detect gradient computation bugs in the crucial AD engines of DL libraries, which support efficient training of DL models. We apply the AD oracle proposed in prior work [75] and compare the computed gradients between reverse-mode AD (the most commonly used mode in DL libraries), forward-mode AD, and numerical differentiation (ND).

4 IMPLEMENTATION

Dataset construction. We scraped the GitHub repositories of targeted libraries using the *requests* library [3], and finally collected 1750 and 633 bug-triggering code snippets for PyTorch and TensorFlow, respectively, from their historical issues or PRs. The lower number of TensorFlow is because it has fewer PRs than PyTorch, as TensorFlow developers are not as active in confirming bugs/PRs, and rarely include code blocks in their PRs. Note that we perform additional cleaning on the extracted code to filter out error messages and remove code lines that contain only the inputs and outputs of executions. We also did not consider the code snippets that fail to pass syntax checking or are longer than 256 tokens.

For buggy API annotation, we manually annotate $K = 6$ randomly sampled examples and use them as in-context examples. For each unlabeled example, we query Codex to complete the buggy API with $temperature = 0$ (deterministic greedy decoding) to get the most confident prediction following [67].

Language models. We evaluate on two specific generative LLMs Codex (code-davinci-002) and CodeGen (350M/2B/6B-mono). Unlike Codex whose training data and model weights are not released, CodeGen [43] is a widely-used open-source generative model [15, 41], which provides trained models of various sizes and allows fine-tuning. We access Codex through its API and use the PyTorch implementation of the CodeGen models on Hugging Face [29].

Fine-tuning. We perform fine-tuning on the CodeGen models because Codex is not open source. We fine-tune a separate model for each studied DL library. To update the model parameters for each targeted library, we use batch_size=32, learning rate=5e-5, and train the models with AdamW optimizer [39] for 10 epochs, using a linear learning rate scheduler with 10% warmup proportion.

5 EVALUATION

5.1 Research Questions

We investigate the following research questions in our experiments:

- **RQ1:** How does different learning paradigms of FuzzGPT compare against each other?
- **RQ2:** How does FuzzGPT compared against existing fuzzers?
- **RQ3:** How do the key components of FuzzGPT contribute to its effectiveness?
- **RQ4:** Is FuzzGPT able to detect new bugs?

5.2 Experimental Setup

Subject systems. We evaluate FuzzGPT on fuzzing PyTorch and TensorFlow, two of the most popular open-source DL libraries. For RQ1, we separately run FuzzGPT with few-shot, zero-shot, and fine-tune settings to evaluate their individual effectiveness for fuzzing (denoted as FuzzGPT-FS, FuzzGPT-ZS, and FuzzGPT-FT respectively). We use FuzzGPT-FS as the default implementation for FuzzGPT if not specified explicitly. For RQ2, We compare our approach with prior work on PyTorch (v1.12) and TensorFlow (v2.10), the same version as the most recent work TitanFuzz[13], and we use the same set of public Python APIs as TitanFuzz as well. For RQ3, due to huge costs, we conducted our ablation study experiments for all three settings of FuzzGPT on 50 APIs randomly sampled from one example DL library PyTorch, and report the average results of 5 runs following prior work [13]. For RQ4, we run all the generated tests on the nightly version of PyTorch and TensorFlow to find previously unknown bugs (Section 6.4).

Baselines. We compare FuzzGPT to state-of-the-art DL library fuzzers, including state-of-the-art API-level (FreeFuzz [69], DeepREL [14], and ∇Fuzz [75]) and model-level (Muffin [23]) fuzzers, as well as the most recent TitanFuzz [13]. We run each tool with its default configuration on both libraries, except that Muffin was only executed on TensorFlow since it does not support PyTorch.

Environment. We use a 64-core workstation with 256 GB RAM and running Ubuntu 20.04.5 LTS with 4 NVIDIA RTX A6000 GPUs.

Fuzzing budget. Our default setting generates 100 programs for each target API. In the Few-shot approach, for each target API, we *independently* construct 10 prompts, each with 6-shot examples picked randomly, and feed each prompt to the LLM to sample 10 generations. Similarly, in the zero-shot approach, we randomly choose 10 different examples from our dataset, and use the partial/complete code to construct 10 prompts to perform completion/editing. In the Fine-tune approach, we use a fixed task description, and query the model for 10 times to generation all 100 programs for a target API.

Generation. Our default setting when using all LLMs for generation uses top-p sampling with $p = 0.95$, $temperature = 0.8$, and $max\_token = 256$ following prior work [13].

5.3 Metrics

Detected bugs. Following prior work on DL library fuzzing [13, 48, 65, 68, 69, 73, 75], we report the number of unique detected bugs.

Unique crashes. Besides counting all bugs, we also count the number of unique crashes as another proxy for fuzzing effectiveness. Unique crashes are widely used in the literature for evaluating fuzzing technique [18, 30, 40]. Note that our definition in this work is more strict: First, we manually examine the root cause of each crash. If different programs crash due to the same reason, we count them as one unique crash. Second, all the unique crashes reported in our study have been confirmed by developers as **unique and real crash bugs**.

Code coverage. Code coverage has been widely adopted in software testing and recently DL library/compiler testing [13, 23, 36, 69, 75]. We follow recent DL library fuzzing work [13, 23, 69, 75] and measure Python line coverage with the *coverage.py* tool [1]. We excluded additional coverage added by the oracle checking for fair comparison.

API coverage. We evaluate the number of covered DL APIs as an important metric of test adequacy following prior work on fuzzing DL libraries [13, 14, 69, 75].
Unique valid programs. A generated program is considered valid if the program executes successfully without exceptions and actually invokes the target API at least once. We also remove the programs already generated and only consider unique programs.

6 RESULT ANALYSIS

6.1 Comparison of Learning Paradigms.

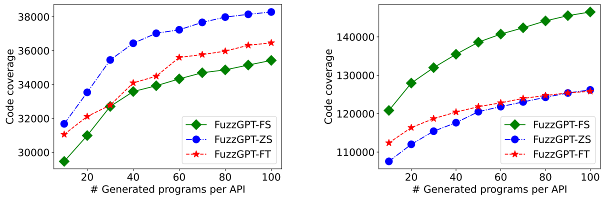
We first compare all our three FuzzGPT variants (few-shot, zero-shot, fine-tune) against each other to understand their performance. Table 1 summarizes the results. Columns **#APIs**, **#Prog.**, and **Cov** present the number of APIs, unique programs, and lines covered. More specifically, **Valid** means only unique programs without runtime errors are considered, **All** means all generated unique programs are considered. Lastly, **Valid(%)** computes the ratio of valid programs over all generated unique programs. Figure 5 further shows the coverage trend with respect to the number of generated programs per API.

We can observe that FuzzGPT-FS has the highest number of covered APIs and unique (valid) programs on both PyTorch and TensorFlow. The reason could be that it provides the LLM (i.e., Codex) with a rich context (including K = 6 bug examples, which very likely contain different buggy APIs), enabling it to learn and combine a variety of bug patterns and use a diverse set of APIs.

FuzzGPT-ZS has a much lower valid rate compared with other variants. The reason is that it is required to complete existing partial programs. In this way, the search space is more constrained compared to other variants, and the task is more challenging since the newly generated code needs to be compatible. Meanwhile, FuzzGPT-ZS may trigger more interesting interactions between APIs in the existing partial programs and newly generated APIs. Furthermore, the partial code reused from bug history can also be very valuable, and may already cover interesting program paths/behaviors. As a result, FuzzGPT-ZS even achieves the highest coverage on PyTorch. FuzzGPT-ZS performs relatively worse on TensorFlow, potentially because there are fewer snippets (only 633, which can hardly cover all 3316 TensorFlow APIs) to reuse for TensorFlow.

FuzzGPT-FT achieves comparable code coverage on both libraries, and has the highest valid rate on PyTorch, even with a smaller model (CodeGen-6B). The results suggest that fine-tuning can be a very effective approach for fuzzing a specific library, since the fine-tuned model has learned from all collected buggy patterns via updating model parameters, and can "select" or "mix" the learned buggy ingredients to target a specific API during generation. On the contrary, few-shot consumes a limited number of in-context examples, and zero-shot relies on one partial example at each inference step. Nevertheless, fine-tuning requires collecting a (high-quality) fine-tuning dataset, and training a different LLM for every different task (which can be costly in terms of computation resources and storage).

FuzzGPT demonstrates the ability to generate fuzzing inputs using techniques from both in-context learning and fine-tuning. From the coverage trend in Figure 5, we observe that in all three variants, the coverage does not saturate even after all 100 code snippets get generated, showing the power of LLMs in learning from historical bug-triggering datasets to continuously generate valuable fuzzing programs that can obtain more coverage.



(a) PyTorch (b) TensorFlow

Figure 5: Coverage trend of FuzzGPT-FS/-ZS/-FT.

Table 1: Comparison of learning paradigms.

	Paradigm	# APIs		# Prog.		Valid(%)	Cov
		Valid	All	Valid	All		
PyTorch	FuzzGPT-FS	**1377**	**1588**	**42496**	**154904**	27.43%	35426
	FuzzGPT-ZS	1237	1553	7809	132111	5.91%	**38284**
	FuzzGPT-FT	1223	1546	31225	112765	**27.69%**	36463
TensorFlow	FuzzGPT-FS	**2309**	**3314**	**54058**	**310483**	**17.41%**	**146487**
	FuzzGPT-ZS	1460	3157	4650	233887	1.99%	126193
	FuzzGPT-FT	1834	3292	31105	253216	12.28%	125832

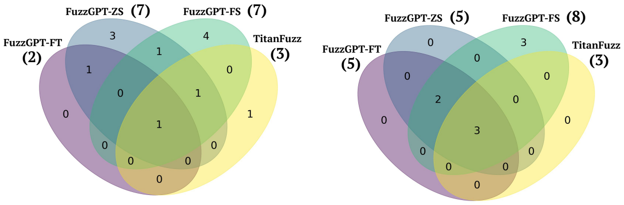
6.2 Comparison with Prior Work

We compare FuzzGPT-FS/-ZS/-FT against state-of-the-art fuzzer TitanFuzz and other recent DL library fuzzers. All techniques are applied under their default configurations.
API and code coverage. As shown in Table 2, all three variants of FuzzGPT significantly outperform all existing fuzzers including state-of-the-art TitanFuzz in code coverage. In particular, the best-performing variants FuzzGPT-FS/FuzzGPT-ZS achieve state-of-the-art results of 54.37%/33.72% line coverage on TensorFlow/PyTorch, 36.03%/60.70% improvement over TitanFuzz. We also observe an interesting fact that FuzzGPT has similar API coverage with TitanFuzz but has much higher code coverage. This demonstrates that FuzzGPT can cover much more interesting code behaviors/paths for DL libraries. Both FuzzGPT and TitanFuzz rely on LLMs to fully automatically generate (or mutate) programs and significantly outperform prior techniques (FreeFuzz, DeepREL, ∇Fuzz, Muffin) in terms of API coverage, showing the superiority of LLMs for fuzzing.

Table 2: Comparison with prior work.

	PyTorch		TensorFlow	
	Code Cov	API Cov	Code Cov	API Cov
Codebase Under Test	113538 (100.00%)	1593	269448 (100.00%)	3316
FreeFuzz	15688 (13.82%)	468	78548 (29.15%)	581
DeepREL	15794 (13.91%)	1071	82592 (30.65%)	1159
∇Fuzz	15860 (13.97%)	1071	89722 (33.30%)	1159
Muffin	NA	NA	79283 (29.42%)	79
TitanFuzz-seed-only	22584 (19.89%)	1329	103054 (38.35%)	2215
TitanFuzz	23823 (20.98%)	1329	107685 (39.97%)	2215
FuzzGPT-FS-25	32305 (28.45%)	1296	130312 (48.36%)	1937
FuzzGPT-FS	35426 (31.20%)	**1377**	**146487 (54.37%)**	**2309**
FuzzGPT-ZS	**38284 (33.72%)**	1237	126193 (46.83%)	1460
FuzzGPT-FT	36463 (32.12%)	1223	125832 (46.70%)	1834

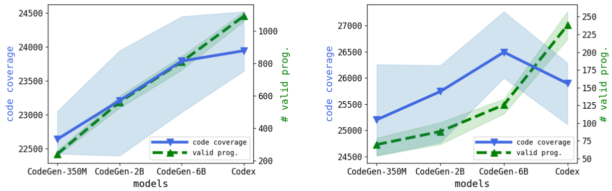
Crash detection. We compare the bug finding capabilities of FuzzGPT and TitanFuzz on an example library PyTorch using the number of unique crashes as a metric. We did not include inconsistency bugs in this comparison, because crashes are easier to measure and can serve as an approximation of bug finding capabilities [30]. We



(a) Direct crashes.

(b) AD Crashes.

Figure 6: Venn diagram of unique crashes



(a) Few-shot

(b) Zero-shot

Figure 7: FuzzGPT-FS/-ZS with model size scaling

run both tools with their default settings, and execute the programs directly to detect crashes. In addition to direct execution, we also execute all programs with AD oracles to detect more crashes. Figure 6 shows the Venn diagram comparison of FuzzGPT-FS/-ZS/-FT and TitanFuzz where the number in the parenthesis is the number of total crashes found by each technique. First, we observe that all three of our techniques can find more crashes in total (including both AD and direct execution crashes) compared with the baseline of TitanFuzz, e.g., our default FuzzGPT-FS detects **2.5** times as many unique crashes as TitanFuzz. Moreover, by combining FS/ZS/FT together, FuzzGPT can detect 19 distinct crashes in total, with 14 unique crashes that cannot be found by TitanFuzz, while only 1 crash is uniquely found by TitanFuzz, highlighting the effectiveness of FuzzGPT in generating unusual crash-triggering programs with the help of historical bug-triggering code snippets.

Generation efficiency. We further discuss the generation efficiency for FuzzGPT and the strongest baseline TitanFuzz. Since TitanFuzz uses Codex to first generate 25 seed programs and then performs mutations with InCoder, we apply FuzzGPT-FS to only generate 25 programs using Codex (denoted as FuzzGPT-FS-25). According to Table 2, even FuzzGPT-FS-25 can substantially outperform TitanFuzz with much lower cost (TitanFuzz further involves additional mutations with InCoder for each API), demonstrating that LLMs like Codex can effectively leverage historical bug-triggering programs to generate valuable programs for fuzzing. Moreover, the mutation phase of TitanFuzz does not bring significant coverage gain compared to its seed-only version (i.e., TitanFuzz-seed-only), while FuzzGPT's coverage does not saturate even after 100 generations (Figure 5). Please note that overall it is hard to precisely compare the efficiency of techniques using different LLMs (due to different CPU/GPU/Cloud costs), and we tried our best to make the discussion fair here.

6.3 Ablation Study

6.3.1 *Few-shot Learning.*

We first study the various design choices for FuzzGPT-FS which provide the LLMs with several real bug-triggering code examples in the prompt. We compare different models and variants of the prompting strategies.

Model size. We first evaluate the performance of FuzzGPT-FS with the model size scaling. Figure 7a plots the code coverage and # of valid programs generated as we increase the model size for all 5 runs (with lines representing the average values). We can see that larger models are able to generate more (unique) syntactically and semantically correct programs. We can also observe a clear gain in coverage with the increase of model parameters (from CodeGen 350M to 2B and finally 6B), and that CodeGen-6B can already achieve comparable performance compared to Codex in terms of coverage.

Chain-of-Thought prompting. We now examine the effectiveness of the bug description in our few-shot template. Our default prompting strategy **w/ CoT** provides natural language explanation to the buggy code (shown in Figure 4), which can be seen as chain-of-thought prompting as it instructs the LLM to first generate the possible bug reason (in natural language) and then generate code conditioned on it. The baseline strategy **w/o CoT** removes the `Bug description: ...` component from the prompt and only asks the model to generate programs from the specified API. As shown in Table 3, including the bug description significantly improves the code coverage, indicating that it is beneficial to give LLMs some intermediate context information to "reason" about the buggy patterns. Table 3 also shows that including natural language description in each example encourages the model to generate more unique APIs, potentially suggesting that the model first generates more diverse natural language description which affects the later code generation. The lower valid rate is probably because the generated programs are more likely to cover some edge-cases and trigger run-time exceptions.

Table 3: FuzzGPT-FS w/ or w/o CoT prompting.

Prompt	Valid APIs	All APIs	Valid Prog.	All Prog.	Valid(%)	Cov
w/ CoT	**190**	**428**	1092	**4885**	22.36%	**23945**
w/o CoT	181	377	**1346**	4798	**28.05%**	22922

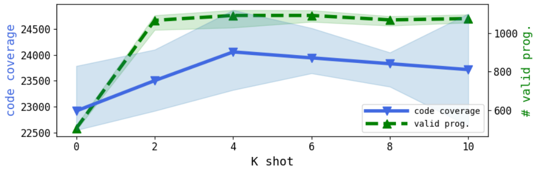


Figure 8: FuzzGPT-FS as the number of examples K increases.

# of examples. We study the effect of K, the number of examples in the context. Figure 8 shows trend of the code coverage and # of valid programs as we increase the number of examples. We first notice that having no examples ($K = 0$) is by far the worst in terms of both coverage and valid programs generated. As we slowly increase K, the coverage improves drastically, demonstrating the benefit of using few-shot learning to provide in-context examples for the LLM to learn from. However, we see that the coverage actually begins to decrease as we add more examples. This could be due to having too many prior historical bug-triggering examples, causing the LLMs to restrict its generation creativity and get distracted. In fact, it has been also observed in prior work [74] on NLP that the distracting prompt structure, with more few-shot examples, can decrease the LLM's performance.

6.3.2 Zero-shot Learning. **Model size.** Figure 7b shows the code coverage and # of valid programs generated using FuzzGPT-ZS as we vary the size of the model used. We first observe a clear trend of improvement as we increase model size, except that the coverage drops for Codex. One reason could be Codex generates much more valid programs, and may miss coverage obtained during exception-handling code (covered by invalid programs). Additionally, this could also be due to the usage of partial programs in the zero-shot setting where we directly re-use part of the historical bug-triggering code. As such, smaller models like CodeGen can obtain a high coverage results without needing to generate a complete code snippet required in the few-shot setting. Nevertheless, we still observe that larger models like Codex is able to achieve the highest number of generated valid programs which are important to test DL libraries.

Prompting Strategy. We evaluate three different zero-shot prompting strategies: **editing** (to edit a complete program), **completion** (to complete a partial program, our default zero-shot variant), and a baseline **completion-NL** where only a natural language description and no code is given to the LLM for completion. As shown in Table 4, completion achieves significantly higher coverage than completion-NL, highlighting the effectiveness of including partial historical bug-triggering programs in the prompt. It is also noteworthy that completion has lower valid rate than completion-NL, because the partial code contains unusual patterns and thus is harder to generate semantically valid completions. The extremely low valid rate of editing is because editing an existing program to use new APIs fully automatically is an even more challenging task for Codex.

Table 4: FuzzGPT-ZS with different prompting strategies.

Prompt	Valid APIs	All APIs	Valid Prog.	All Prog.	Valid(%)	Cov
editing	22	304	20	1609	1.22%	19440
completion-NL	**192**	**400**	**506**	**4972**	**10.17%**	22917
completion	112	362	238	3470	6.86%	**25893**

6.3.3 Fine-tuning. **Model size.** We study the fine-tuning performance with different model sizes. Table 5 shows that fine-tuned CodeGen-6B, the largest CodeGen model studied, does achieve the highest code coverage. Compared to the original CodeGen (row 6B w/o FT), our fine-tuned model significantly improves the number of valid programs and code coverage. Note that although the original CodeGen model can cover more library APIs (given the huge number of code tokens seen during pre-training), it cannot achieve high code coverage as the model is not specialized for DL code generation and has a much lower valid program rate (only 10.17%). With fine-tuning, another interesting observation is that larger models are not always better. For example, the 2B model has the most unique valid programs. This could potentially be explained with the *validity-unusualness* trade-off: as we fine-tune LLMs towards unusualness-favored generation, we may lose some validity-preserving information, while both contribute to code coverage and the ultimate bug-finding capability.

Prompting strategy. According to Table 6, similar to the few-shot results, including bug description in the fine-tuning process also helps the fine-tuned LLM to generate much more diverse outputs (i.e., more unique and valid APIs/programs) and achieve higher coverage. Another interesting finding is that, unlike in FuzzGPT-FS, CoT contributes to an increased percentage of valid programs in FuzzGPT-FT. Our hypothesis is that, in the fine-tuning process, since the LLM was trained on the entire dataset, CoT can assist it in better associating a reasonable buggy pattern with a target API, compared to the few-shot scenario. We would like to further clarify that getting more *valid* programs is *not* the goal of our CoT prompting or our overall approach. Therefore, we do not expect CoT to consistently improve this metric. The main advantage of CoT prompting is to guide the model to generate *unusual* programs more effectively by first describing the bug. Our evaluation shows that CoT does consistently improve code coverage in both the few-shot and fine-tuning settings. This indicates that CoT enables the generation of unusual programs that cover more (interesting) code paths, even if there are fewer valid programs in the few-shot scenario.

Table 5: FuzzGPT-FT with different LLM sizes.

Model	Valid APIs	All APIs	Valid Prog.	All Prog.	Valid(%)	Cov
350M	118	**329**	682	**4236**	16.10%	23352.2
2B	118	292	**835**	3393	**24.60%**	22688.8
6B	**131**	311	768	3532	21.75%	**24420.8**
6B w/o FT	**192**	**400**	506	**4972**	10.17%	22917

Table 6: FuzzGPT-FT with different prompting strategies.

Prompt	Valid APIs	All APIs	Valid Prog.	All Prog.	Valid(%)	Cov
w/ CoT	**131**	**311**	**768**	**3532**	**21.75%**	**24420.8**
w/o CoT	83	250	434	2659	16.34%	24242.8

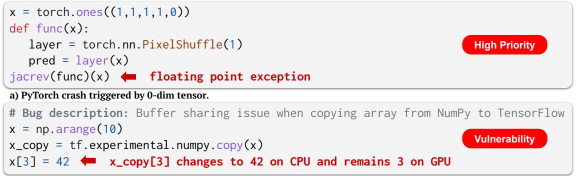
6.3.4 Buggy API Annotation. Finally, we study the impact of our API annotation method (Section 3.1.2) on fuzzing performance. As a baseline, we constructed a randomly annotated dataset by selecting a random DL API *from the bug-triggering program* as the buggy API. We then fine-tune CodeGen-6B on this dataset and compare it with FuzzGPT-FT. Table 7 summarizes the results. We evaluate the labeling accuracy on a random sample of 100 PyTorch issues/PRs, and find our Codex-based labeling achieves 76% precision, while random annotation has 26% precision. When used for fine-tuning, our Codex-based labeling achieves not only a higher ratio of valid programs but also higher coverage compared to random labeling. This is because the wrong API label can be less-aligned with the bug description and code, leading to decreased performance. Interestingly, we further find that even the randomly annotated dataset can still guide FuzzGPT-FT to outperform the CodeGen-6B baseline (without fine-tuning). This demonstrates that the annotation does not need to be fully precise – if the mislabeled API always appears in the code, such (API, code) pairs can still guide the model to learn the fundamental task of generating a program that calls a given target API.

Table 7: FuzzGPT-FT with random API annotation.

Method	Label Acc(%)	Valid(%)	Cov
FuzzGPT-FT	76%	21.75%	24421
FuzzGPT-FT-Random	26%	19.48%	23468
CodeGen-6B w/o FT	N/A	10.17%	22917

6.4 Bug Finding

Due to the extensive human cost in bug finding/reporting, in this RQ, we mainly focus on our default setting: FuzzGPT-FS with all the



```
x = torch.ones((1,1,1,1,0))
def func(x):
   layer = torch.nn.PixelShuffle(1)
   pred = layer(x)
jacrev(func)(x) ⬅ floating point exception
```

a) PyTorch crash triggered by 0-dim tensor.

```
# Bug description: Buffer sharing issue when copying array from NumPy to TensorFlow
x = np.arange(10)
x_copy = tf.experimental.numpy.copy(x)
x[3] = 42 ⬅ x_copy[3] changes to 42 on CPU and remains 3 on GPU
```

b) TensorFlow bug when copying array.

Figure 9: Example bugs found by FuzzGPT.

oracles in Section 3.3. Meanwhile, we expect FuzzGPT-ZS/FuzzGPT-FT to be also effective in bug finding (given their performance in code coverage and crash detection) and may contribute additional bugs.

Bug statistics are summarized in Table 8. In total, FuzzGPT detected 76 bugs, with 61 confirmed, including 49 confirmed as previously unknown bugs (6 of them have already been fixed). Besides, Column **Pending** presents the bugs not yet confirmed and **Won't Fix** shows bugs rejected by developers (usually due to precision issues or efficiency concerns). Notably, Column **High Prio** presents the number of high-priority bugs or security vulnerabilities newly detected by FuzzGPT. Note that fewer bugs are confirmed or fixed on TensorFlow, because TensorFlow developers are less active (as discussed in Section 4 there are fewer PRs in TensorFlow). Out of the 49 confirmed new bugs (including 25 crashes and 24 inconsistencies; within them 30 are AD-related), only 11 can be found by running TitanFuzz (augmented with our oracles), and 2 can be found by directly rerunning historical bug-triggering programs with our oracles. We next present two exemplary bugs detected by FuzzGPT.

Table 8: Summary of detected bugs.

	Total	Confirmed (Fixed)		Pending	Won't Fix	High Prio
		Unknown	Known			
PyTorch	43	33 (6)	5 (1)	1	4	3
TensorFlow	33	16 (0)	7 (0)	5	5	8
Total	76	49 (6)	12 (1)	6	9	11

Figure 9a shows a crash bug when we apply `PixelShuffle` on a special tensor of 0-dimension shape and then compute gradient with `jacrev`. Normally `PixelShuffle` accepts a 4-D tensor where the last dimension is the width of of an image, typically larger than 1. FuzzGPT has learned from an in-context historical bug example [21] where zero-dimension tensors can trigger crashes, and generates this unusual input of shape `(1,1,1,1,0)` for `PixelShuffle` which triggers floating point exception during gradient computation. As `PixelShuffle` is a commonly-used API in computer vision applications [57] where crashes can lead to security risks, this bug is labeled by PyTorch developers as **high-priority** and immediately fixed.

Figure 9b presents a TensorFlow bug where the generated fuzzing code snippet first makes a copy of a numpy array `x` and then modifies its value. In this case, the tensor `x_copy` should remain the same. However, we find that on CPU, after we assign a new value to `x[3]`, the value of copied tensor `x_copy` is also modified. Previous work (including TitanFuzz which also uses LLMs for generation) cannot trigger this bug because it requires calling the `copy` API and then modifying the value of the original data - a series of unnatural operations. FuzzGPT successfully finds this bug because our CoT prompting instructs it to first predict a plausible bug reason "buffer sharing issue" together with its triggering condition "when copying array ..." as a bug description. Following this description, FuzzGPT can generate the unusual program which is very rare in the training dataset and thus hard to be generated by TitanFuzz. This bug enables data manipulation attacks by silently changing copies of the original array, and has been further confirmed as a **security vulnerability** by the Google security team.

7 DISCUSSION

Table 9: FuzzGPT w/o historical information with ChatGPT.

System Message: You are a pytorch fuzzer.	# APIs		# Prog.			
Prompt: Please generate a program to use conv2d ...	Valid	All	Valid	All	Valid(%)	Cov
to demonstrate the example usage **(baseline)**	102	183	2298	4209	**55.00%**	20454
in a way you have not seen in your training dataset	190	297	**2306**	4756	48.00%	**21077**
in a very creative way	201	318	2104	4801	44.00%	20971
in a non-conventional way	**212**	**330**	2038	**4880**	42.00%	20759
in a way that is rarely used by developers in practice	200	306	2056	4841	42.00%	20759
in a very strange way	181	291	1735	4854	36.00%	20377

FuzzGPT w/o historical information. So far we have leveraged historical bug-triggering programs to guide LLMs for unusual program generation. Meanwhile, with the recent advances in the instruct-following capability of LLMs, it is also possible to directly instruct LLMs (without any historical information) to generate unusual programs for fuzzing. To this end, we have tested state-of-the-art ChatGPT [45] (which has a knowledge cutoff at September 2021) with a list of representative prompts on the 50 PyTorch APIs used in our ablation study. More specifically, we first instruct ChatGPT to generate typical example usages of an API as the baseline; then, we instruct ChatGPT to generate unusual programs using various other prompts. Table 9 shows that all the studied prompts can help ChatGPT cover much more APIs than the baseline, demonstrating ChatGPT and similar models (e.g., GPT-4 [46]) can understand the instructions and generate more interesting programs that may cover interesting library paths/behaviors. Another interesting observation is that the other prompts all have lower valid rates than the baseline, since less common programs may more likely fail.

Impact of example selection. Besides the random example selection used in our default FuzzGPT-FS variant, we have also investigated other strategies to select in-context examples. Intuitively, examples with APIs similar to the target API may provide more relevant bug-triggering patterns. Conversely, a diverse set of examples can provide complimentary bug-triggering ingredients. As such, we design a set of smoothed maximum-marginal-relevance (MMR) guided selection strategies [8, 78], including strategies favoring similarity and diversity, and strategies in-between. Interestingly, we observe that the default random strategy is competitive compared with all studied variants. The main reason could be that modern LLMs are powerful enough to learn even from dissimilar examples for program generation; in this way, random selection can provide a diverse set of ingredients to facilitate effective generation.

Threats to validity. The main threats to internal validity lie in the potential bugs in our implementation and experimentation. To mitigate such threats, we performed rigorous review for our code. The main threats to external validity lie in the subject systems used. To reduce the threats, we select two most popular DL libraries, PyTorch and TensorFlow, which have also been widely studied in recent work [13, 14, 25, 69, 75]. Lastly, we also adopt widely used metrics in prior fuzzing work [13, 69, 75], such as real bug detection and code coverage.

8 CONCLUSION

We have introduced FuzzGPT, the first approach to leveraging historical bug-triggering programs to prime LLMs for fuzzing with edge cases. Compared to traditional fuzzing techniques on leveraging such historical information studied for over a decade, FuzzGPT is fully automated, generalizable, and applicable to challenging domains, such as DL library fuzzing. Moreover, FuzzGPT also shows the potential of ChatGPT for edge-case program generation without any historical information. The experimental results show that FuzzGPT substantially outperforms existing DL library fuzzers, and can detect various bugs for PyTorch and TensorFlow.

Artifact Availability. We make our artifact available at [2].

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCF-2131943 and CCF-2141474, as well as research awards from Google, Meta, and Kwai Inc.

REFERENCES

[1] 2023. Coverage.py. https://github.com/nedbat/coveragepy.

[2] 2023. FuzzGPT artifact. https://github.com/ise-uiuc/FuzzGPT.

[3] 2023. requests. https://pypi.org/project/requests/.

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL]

[5] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021), 79–86.

[6] Dalvin Brown. 2021. Hospitals turn to artificial intelligence to help with an age-old problem: Doctors' poor bedside manners. *The Washington Post* (2021). https://www.washingtonpost.com/technology/2021/02/16/virtual-ai-hospital-patients/.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 335–336.

[9] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 305–316.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[12] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105.

[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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 2023)*.

[14] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. https://doi.org/10.1145/3540250.3549085

[15] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. *arXiv preprint arXiv:2212.10007* (2022).

[16] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155.

[18] 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 (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.

[19] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[20] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027* (2020).

[21] GitHub. 2022. torch.nn.PixelShuffle crash with floating point exception when input has 0 size in the last three dimensions. (2022). https://github.com/pytorch/pytorch/issues/85155.

[22] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 50–59. https://doi.org/10.1109/ASE.2017.8115618

[23] J. Gu, X. Luo, Y. Zhou, and X. Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1418–1430. https://doi.org/10.1145/3510003.3510092

[24] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366 [cs.SE]

[25] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498.

[26] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 837–847.

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80.

[28] Christian Holler, Kim Herzig, Andreas Zeller, et al. 2012. Fuzzing with Code Fragments.. In *USENIX Security Symposium*. 445–458.

[29] HuggingFace 2022. Hugging Face. https://huggingface.co.

[30] 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*. 2123–2138.

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

[32] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.

[33] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language model-guided javascript engine fuzzer. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2613–2630.

[34] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *ASPLOS*. 530–543.

[35] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What Makes Good In-Context Examples for GPT-3?. In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*. Association for Computational Linguistics, Dublin, Ireland and Online, 100–114. https://doi.org/10.18653/v1/2022.deelio-1.10

[36] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (apr 2022), 26 pages. https://doi.org/10.1145/3527317

[37] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[38] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

[39] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bkg6RiCqY7

[40] M. Zalewski 2016. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt.

[41] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023. Learning Performance-Improving Code Edits. *arXiv preprint arXiv:2302.07867* (2023).

[42] Tanya Mohn. 2022. Can A.I. All but End Car Crashes? The Potential Is There. *The New York Times* (2022). https://www.nytimes.com/2022/04/19/technology/ai-road-car-safety.html.

[43] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[44] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114* (2021).

[45] OpenAI. 2023. ChatGPT. (2023). https://openai.com/blog/chatgpt.

[46] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[47] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language Models as Knowledge Bases?. 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)*. 2463–2473.

[48] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[49] PyTorch 2023. PyTorch. http://pytorch.org.

[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] Maithra Raghu and Eric Schmidt. 2020. A survey of deep learning for scientific discovery. *arXiv preprint arXiv:2003.11755* (2020).

[53] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning To Retrieve Prompts for In-Context Learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2671.

[54] Adam Satariano and Cade Metz. 2023. Using A.I. to Detect Breast Cancer That Doctors Miss. *The New York Times* (2023). https://www.nytimes.com/2023/03/05/technology/artificial-intelligence-breast-cancer-detection.html.

[55] H. Scudder. 1965. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory* 11, 3 (1965), 363–371. https://doi.org/10.1109/TIT.1965.1053799

[56] Danny Shapiro. 2023. Transportation Generation: See How AI and the Metaverse Are Shaping the Automotive Industry at GTC. *Nvidia Blog* (2023). https://blogs.nvidia.com/blog/2023/02/16/ai-metaverse-shaping-automotive-industry-gtc/.

[57] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. 2016. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1874–1883.

[58] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *EMNLP 2020*. 4222–4235.

[59] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs From Hell. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1138–1153.

[60] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 849–863.

[61] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[62] TensorFlow 2023. TensorFlow. https://www.tensorflow.org.

[63] Alina Tugend. 2021. A Smarter App Is Watching Your Wallet. *The New York Times* (2021). https://www.nytimes.com/2021/03/09/business/apps-personal-finance-budget.html.

[64] 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).

[65] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries. (2022).

[66] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.

[67] Xingyao Wang, Sha Li, and Heng Ji. 2022. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810* (2022).

[68] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

[69] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 995–1007. https://doi.org/10.1145/3510003.3510041

[70] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[71] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[72] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. *arXiv preprint arXiv:2303.10494* (2023).

[73] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[74] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2021. An explanation of in-context learning as implicit bayesian inference. *arXiv preprint arXiv:2111.02080* (2021).

[75] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *International Conference on Software Engineering (ICSE)*. to appear.

[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] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 435–450.

[78] Xi Ye, Srinivasan Iyer, Asli Celikyilmaz, Ves Stoyanov, Greg Durrett, and Ramakanth Pasunuru. 2022. Complementary Explanations for Effective In-Context Learning. *arXiv preprint arXiv:2211.13892* (2022).

[79] Shafiq Joty Yue Wang, Weishi Wang and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP 2021*.

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

[81] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.

[82] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 347–361.

[83] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An Industrial Protocol Fuzzing Framework from a Deep Learning Perspective. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 59–67. https://doi.org/10.1109/ICST.2019.00016

[84] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *ICSE 2022*. 1133–1144.

[85] Hao Zhong. 2022. Enriching Compiler Testing with Real Program from Bug Report. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[86] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

[87] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. 2020. Rethinking pre-training and self-training. *Advances in neural information processing systems* 33 (2020), 3833–3845.