Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models

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

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

Haoran Peng
University of Science and
Technology of China, China
hurrypeng@mail.ustc.edu.cn

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

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

ABSTRACT

Deep Learning (DL) systems have received exponential growth in popularity and have become ubiquitous in our everyday life. Such systems are built on top of popular DL libraries, e.g., TensorFlow and PyTorch which provide APIs as building blocks for DL systems. Detecting bugs in these DL libraries is critical for almost all downstream DL systems in ensuring effectiveness/safety for end users. Meanwhile, traditional fuzzing techniques can be hardly effective for such a challenging domain since the input DL programs need to satisfy both the input language (e.g., Python) syntax/semantics and the DL API input/shape constraints for tensor computations.

To address these limitations, we propose TitanFuzz – the first approach to directly leveraging Large Language Models (LLMs) to generate input programs for fuzzing DL libraries. LLMs are titanic models trained on billions of code snippets and can autoregressively generate human-like code snippets. Our key insight is that modern LLMs can also include numerous code snippets invoking DL library APIs in their training corpora, and thus can implicitly learn both language syntax/semantics and intricate DL API constraints for valid DL program generation. More specifically, we use both generative and infilling LLMs (e.g., Codex/InCoder) to generate and mutate valid/diverse input DL programs for fuzzing. Our experimental results demonstrate that TitanFuzz can achieve 30.38%/50.84% higher code coverage than state-of-the-art fuzzers on TensorFlow/PyTorch. Furthermore, TitanFuzz is able to detect 65 bugs, with 44 already confirmed as previously unknown bugs.

This paper demonstrates that modern titanic LLMs can be leveraged to *directly* perform both generation-based and mutation-based fuzzing studied for decades, while being fully automated, generalizable, and applicable to domains challenging for traditional approaches (such as DL systems). We hope TitanFuzz can stimulate more work in this promising direction of LLMs for fuzzing.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598067

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Software reliability**.

KEYWORDS

Fuzz Testing, Test Generation, Large Language Model

ACM Reference Format:
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 '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598067

1 INTRODUCTION

Deep Learning (DL) is constantly providing revolutionary results and systems in critical fields like autonomous driving [34, 85], healthcare [9], and finance [68]. To build these systems, developers use popular DL libraries such as TensorFlow [66] and PyTorch [57] by composing individual library API calls, typically exposed in Python, to build models and perform computations. Due to the significance of detecting and fixing bugs in these DL libraries, researchers have applied various automated bug-finding techniques to test/analyze these libraries [16, 27, 45, 56, 70–72, 77]. One such popular methodology is fuzzing [8, 65, 84] – where a large set of inputs are generated and fed to the libraries to find potential bugs.

Previous work on fuzzing DL libraries mainly falls into two categories: API-level fuzzing [16, 72, 77] and model-level fuzzing [27, 44, 56, 71]. API-level fuzzing focuses on testing individual library APIs by generating various different inputs for each target API to discover potential crashes or result inconsistencies. On the other hand, model-level fuzzing techniques aim to generate diverse complete DL models and then compare the model outputs on different backends (e.g., different low-level libraries of Keras [36]) to discover potential bugs. While both model-level and API-level fuzzing techniques have been shown effective in bug finding, they still suffer from the following limitations:

1) Lack of diverse API sequences. Previous API-level fuzzers [16, 72] only focus on fuzzing each single DL library API in isolation. These techniques attempt to create many different inputs to a particular target API via simple mutation rules. However, these inputs are usually constructed by single code lines (at most one library input

```
x = torch.randn(3, 3, dtype=torch.float32)
print("Intermediate: ", torch.log(x * 2 - 1)) # Intermediate contains NaN
output = torch.matrix_exp(torch.log(x * 2 - 1)) # on CPU: Contains NaN
x = x.cuda()
output = torch.matrix_exp(torch.log(x * 2 - 1)) # on GPU: Does not contain NaN
```

a)

```
x = [-0, 0, -0] # generated by previous statements
y = 1 / torch.clamp(x, min=0, max=1)
# on CPU y = 1 / [-0, 0, -0] = [-Inf, Inf, -Inf] wrong!
# on GPU y = 1 / [0, 0, 0] = [Inf, Inf, Inf] correct!
```

b)

Figure 1: Bugs in DL Libraries

creation API, e.g., randomly initializing a tensor with a certain data type and shape) and cannot reveal bugs that are caused by chained API sequences. While model-level fuzzing techniques [27, 56, 71] can potentially test API sequences, the mutation rules usually have strict constraints, e.g., LEMON's layer addition rule cannot be applied to layers with different input and output shape [71], while Muffin needs to manually annotate input/output restrictions of considered DL APIs and uses additional reshaping operation to ensure valid connections [25]. As a result, model-level fuzzers can only cover a limited set of APIs with limited patterns [72], missing many diverse and interesting API sequences that can lead to bugs.

Figure 1a shows an example bug in PyTorch exposed by an API sequence. A random input is created and the code produces an intermediate variable by invoking the `log` API. The `log` function will produce `NaN` (Not a Number) for negative inputs. In theory, when `matrix_exp` is applied it should also contain `NaN` values. However, when running this code on GPU, it does not output any `NaN` values. More interestingly, this incorrect behavior on GPU cannot be reproduced if we just pass the intermediate tensor (which contains `NaN`) instead of the API call `log` to `matrix_exp`. The bug is only triggered by a synchronization error when we apply the API sequence. Prior API-level fuzzers cannot find this bug; model-level fuzzers can also hardly detect this bug as the specific sequence of `log` followed by `matrix_exp` is rarely used in building DL models where the focus is on layer APIs such as `Conv2d` or `MaxPool2d`.

2) Cannot generate arbitrary code. DL library APIs are exposed to the end user in Python which is not a statically typed language, making it hard to directly obtain the input and output argument types. Also, library APIs usually operate on input tensors where a shape mismatch (e.g., matrix multiplication with incorrect dimensions) can lead to runtime errors. Traditional program synthesis techniques [51, 54, 64] typically restrict themselves to a small set of functionalities in the language and cannot deal with a large number of library APIs, each with their own specific input and output parameters and types. As such, existing DL library fuzzers use predefined generation grammars that focus on mutating a small part of the program to minimize these errors. This limits the variety in both code structure and also the types of inputs that we can use to test library APIs. For example, FreeFuzz [72], a state-of-the-art API-level fuzzer, will first collect the valid argument space (e.g., type and shape of the input tensor) for a target API by mining open-source code snippets. During the fuzzing loop, FreeFuzz will perform small mutations of these valid inputs to generate new inputs, e.g., changing the data type (e.g., `float32` to `float16`). As such, FreeFuzz is limited by the traced argument space and predefined mutation rules. Meanwhile, Muffin [25], a recent state-of-the-art model-level fuzzer, generates diverse models via using manually annotated specifications for each manipulated API and predefined code structures (e.g. models consisting of sequential layers) in order to preserve model validity. As such, such prior DL library fuzzers cannot fully explore the huge search space that exists when it comes to using DL library APIs.

Figure 1b shows an example bug in PyTorch which cannot be detected by previous fuzzing techniques. The bug is caused by the `clamp` function not clamping negative zero to positive zero on CPU. Even though this bug is due to a single API, previous techniques cannot detect this bug as negative zero is *almost* zero. However, the bug is exposed when we apply 1 divided by the clamped list where the correct value should be Positive Inf not Negative Inf (significant value difference). Such Python basic expression is often used by developers in combination with library APIs. However, this bug is missed by prior work due to the restricted generation methods of both existing API-level and model-level techniques.

Our Work. We propose TitanFuzz – the first fully automated approach for fuzzing DL libraries via Large Pre-trained Language Models (LLMs) [10]. As discussed earlier, DL libraries expose APIs mostly in Python (dynamically typed), making it challenging to directly apply traditional program synthesis to generate syntactically/semantically valid DL programs [6]. Moreover, DL APIs may involve complicated input/shape constraints for tensor computations that are extremely hard to satisfy without additional manual efforts. In contrast, modern LLMs can serve as a natural solution as they are built using the popular Transformer [69] architecture which allows for autoregressive generation (based on left context) or infilling (based on bi-directional context) trained using billions of code tokens to generate "human-like" programs. Our key insight is that *modern titanic LLMs can include numerous code snippets using various DL libraries in their training corpora (e.g., there are >400,000 TensorFlow/PyTorch projects on GitHub, which is an important training source for modern LLMs), allowing them to implicitly learn both Python syntax/semantics and intricate types/constraints of DL APIs to directly generate/mutate valid DL programs for fuzzing DL libraries.*

In TitanFuzz, we first use a generative LLM with a step-by-step input prompt [47] to produce the initial seed programs for fuzzing. To enrich the pool of test programs, we further adopt an evolutionary strategy to produce new test programs by using LLMs to automatically mutate the seed programs. This mutation process is done using multiple mutation operators designed to leverage an infilling LLM to replace only parts of the seed with new code. In order to generate more complicated and diverse API call relations, we design a fitness function which prioritizes seeds or mutated test programs based on data-dependency depth and number of unique library APIs, allowing us to discover bugs that can only be found when studying complex API relationships. Finally, we execute the generated test programs with differential testing on different backends to detect bugs. In fact, both bugs in Figure 1 which cannot be detected by any previous DL library fuzzers are detected by TitanFuzz and confirmed by developers as previously unknown bugs. While our approach is general and can be built upon any LLMs, we build our technique on Codex [12] and InCoder [21] as they have shown state-of-the-art results for generative and infilling tasks, respectively. Also, while we evaluate on two most popular DL libraries: TensorFlow and PyTorch, our idea of directly using

LLMs as the generation engine can be applied for fuzzing any DL libraries with little additional effort and can further be extended for fuzzing/testing software systems from other application domains. In summary, this paper makes the following contributions:

- **Dimension.** This paper opens a new dimension for fuzzing DL libraries (and beyond) by directly using LLMs as generation engines. To our knowledge, this is also the first work demonstrating that modern titanic LLMs can directly perform both generation-based [80] and mutation-based [50] fuzzing studied for decades, while being fully automated, generalizable, and applicable to domains challenging for traditional approaches (such as DL systems). Our approach can be easily extended to test software systems from other application domains (e.g., compilers, interpreters, DB systems, SMT solvers, smart contracts, and other popular libraries). Moreover, this paper demonstrates the promising future of directly leveraging modern LLMs for fuzzing and testing in general.
- **Technique.** We implement TitanFuzz, a fully automated fuzzer for DL libraries that first uses a generative LLM (Codex) to synthesize high-quality seed inputs and then combines an infilling LLM (InCoder) with an evolutionary algorithm to guide the generation towards a higher number of unique library API usages and valid/diverse DL programs.
- **Study.** We perform an extensive evaluation on two of the most popular DL libraries: PyTorch and TensorFlow. Our result shows that TitanFuzz is able to cover 1329 / 2215 APIs with 20.98% / 39.97% coverage on PyTorch and TensorFlow respectively, improving on the state-of-the-art fuzzing tools by 24.09% / 91.11% in API coverage and 50.84% / 30.38% in code coverage. In addition, TitanFuzz is able to detect 65 bugs, with 44 already confirmed as previously unknown bugs. Furthermore, we perform a broad ablation study to justify the design of components in TitanFuzz.

2 BACKGROUND AND RELATED WORK

2.1 Fuzzing Deep Learning Libraries

DL libraries (e.g., TensorFlow [66] and PyTorch [57]) serve as the fundamental building block for all DL pipelines by providing thousands of APIs for building, training, and deploying DL models. Figure 2 shows an example DL model that classifies an input image with its associated training and inference steps. The DL model consists of two convolutional (`Conv2d`) and one fully connected linear (`Linear`) layers. In the forward pass, the first convolutional layer with a non-linear activation function (`RELU`) produces an intermediate output, which is then passed to the second convolutional layer. Next, the fully connected layer is called to produce the final output. In short, using these sets of library APIs, which define the functionality of each layer, the DL libraries essentially create a computational graph, highlighting the flow of data in the model as shown on the right side of the figure. In order to train the model, we first initialize it together with an optimizer that updates the model weights. Next, we load the training data, and for each pair of input and its associated label, we obtain the model output. Finally, we compute the loss together with its gradient to perform back-propagation and update the model weights. To use the model for inference, we first load the trained model and then pass the chosen

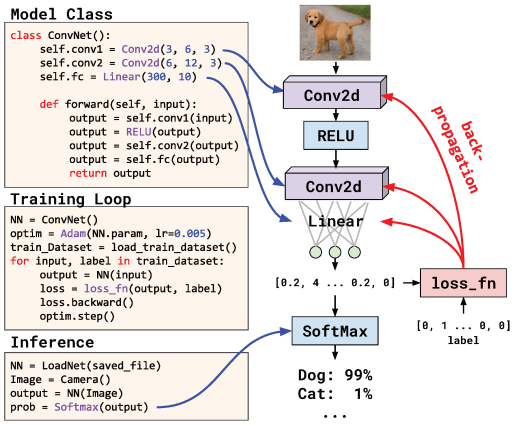

Figure 2: Deep Learning basics

image to get the model output. Further, we can use the `Softmax` API to obtain the probability representation of the output.

Prior work on fuzzing DL libraries can be mainly classified into two categories, namely model-level and API-level fuzzers. Model-level fuzzers attempt to leverage complete DL models (which cover various sets of DL library APIs) as test inputs. CRADLE [56] is one of the first work in the area that detects inconsistencies by running existing models on multiple low-level backends of Keras [36]. To generate more diverse models, LEMON [71] and AUDEE [27] further extend the idea of CRADLE to apply predefined mutation rules on seed models/inputs. Muffin [25] further applies a top-down approach to generate DL models for bug detection in both the inference and training phases. Very recently, NNSmith [44] leverages symbolic constraint solving and gradient-based search for high-quality model synthesis. While such model-level fuzzers are able to find bugs in DL libraries, due to the input/output constraints of DL APIs, model-level mutation/generation rules either are restrictive to certain shape-preserving APIs [71] or require manual annotation of the restrictions of all targeted APIs [25], leading to a limited number of unique APIs covered. Different from model-level fuzzing, API-level fuzzing focuses on finding bugs within a single API at a time. FreeFuzz [72] is an API-level fuzzer that first learns the valid inputs for each target API through mining open-source code snippets and then applies simple mutations to generate diverse inputs to test a target API. Similarly, DocTer [77] mines the input constraints from API documentation by learning the extraction rules with 30% manually annotated API parameters, and then generates valid and invalid inputs based on the extracted constraints to detect crashes. More recently, DeepREL [16] and ∇Fuzz [79] further leverage relational APIs (e.g., APIs that always return the same results/statuses given the same inputs) and automatic differentiation, respectively, as the test oracle for more effective API-level DL library fuzzing. While researchers have demonstrated that API-level fuzzing can cover many more DL library APIs than model-level fuzzing [16, 72, 79], API-level fuzzers cannot detect any bug that arises from interactions within a complex API sequence.

2.2 Large Pre-trained Language Models

Large Pre-trained Language Models (LLMs) typically follow the Transformer [69] architecture of an *encoder* to produce an encoded representation of the input and a *decoder* to generate output tokens.

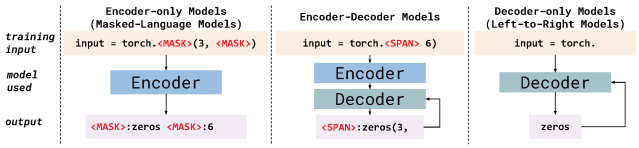

Figure 3: Overview of different LLM architectures

These LLMs are pre-trained on billions of available text on the internet and have been widely used in many different Natural Language Processing (NLP) tasks [10, 49, 81]. Due to the large amounts of available pre-training data, LLMs without any fine-tuning on specialized datasets can already be directly used for very specific downstream tasks. This is accomplished using prompt engineering [47], where a natural language description of the task together with a few demonstrations of the task is provided to the LLM first before the actual input. Researchers have demonstrated that this paradigm of directly leveraging LLMs through prompts can already achieve state-of-the-art performance on downstream tasks [10]. More recently, supported by code naturalness [28], researchers have begun to apply LLMs for programming languages [12, 17, 21, 78]. Similar to the impressive performance achieved on NLP tasks, LLMs can also excel in many code related tasks such as code completion [12], code synthesis [6, 46], and automated program repair [74–76].

Based on the model architectures and pre-training tasks, LLMs can be categorized into: **Decoder-only**, **Encoder-only** and **Encoder-decoder**. Figure 3 shows the three LLM types. Decoder-only models [10, 12] use the decoder to predict the probability of the next token based on all previous tokens. These models can be used in an auto-regressive manner to perform auto-completion given all previous (or left) context. Encoder-only models [17, 26] aim to provide a representation of the input through the use of the encoder component. Such models are trained using the Masked Language Model (MLM) objective where a percentage (e.g., 15%) of the tokens during training are masked out, and the goal is to recover the true values of these masked tokens based on both the context before and after. Encoder-decoder models [5, 59] use both the encoder and decoder component and are most commonly trained using the Masked Span Prediction (MSP) objective. Instead of replacing each chosen token with a masked token, MSP replaces a sequence of tokens with a single masked span token. The model is then asked to recover the entire sequence during training. During inference, these models can be used to directly *fill in* code in the middle using both the context before and after. However, training both the encoder and decoder components can be time-consuming. As such, recently, researchers have proposed the InCoder [21] model which uses only the decoder component but can fill in text/code in the middle through the Causal Language Model training objective [21]. Instead of using the decoder to autoregressively predict the next token in the original training data, similar to MSP, InCoder also replaces random sequences in the training data with masked span tokens. Using this processed training data, InCoder only autoregressively predicts the original masked sequence given the processed input. With this training strategy, the resulting InCoder model is also able to perform infilling and achieve state-of-the-art results.

In summary, LLMs can perform two main types of code generation tasks: *generative* and *infilling*. Generative tasks involve auto-completing a complete code snippet given the left context only (e.g., some starting code or a natural language description), typically done using decoder-only models. Infilling tasks aim to insert the most natural code based on bi-directional context (e.g., in the middle of a code snippet), which can be done using encoder-only, encoder-decoder, and also decoder-only models that are trained for infilling, such as InCoder. In this work, we leverage modern LLMs to perform both types of generation tasks for fuzzing DL libraries. Besides generative models, inspired by recent work [74, 75] on infilling-style program repair (where LLMs generate correct patches by directly filling in the correct code given the context), we also leverage infilling models to perform mutations by replacing a small part of an input program with masked tokens and then filling in generated code to produce even more diverse programs.

2.3 Testing using Deep Learning Models

Due to the recent advances in Deep Learning (DL), researchers have looked into using DL models to facilitate automated test generation or fuzzing of different software systems. Traditionally, such techniques rely on training a neural network to produce code snippets automatically. Seqfuzzer [86] employs a Recurrent Neural Network (RNN) [13, 29] and GANFuzz [32] leverages the Generative Adversarial Network (GAN) [24] for protocol fuzzing. Learn&Fuzz [23], DeepSmith [15] and DeepFuzz [48] each trains a RNN to generate programs for PDF file parsers, OpenCL and C, respectively. Similarly, Montage [40] targets JavaScript engines by training a RNN to mutate subtrees of a seed input to produce valid JavaScript programs. None of the above work has leveraged LLMs for fuzzing.

More recently, COMFORT [82], has been proposed to fine-tune GPT-2 (with 1.5B parameters) [58] on open-source JavaScript programs. COMFORT then uses the fine-tuned GPT-2 model to synthesize JavaScript programs to test specific engines. While COMFORT has demonstrated the potential of LLMs for fuzzing, it did not leverage state-of-the-art LLMs for code and thus requires an extensive fine-tuning dataset. Moreover, COMFORT cannot perform end-to-end test generation using GPT-2, and has to rely on additional heuristics to generate inputs for the synthesized programs. In contrast, our work demonstrates for the first time that directly leveraging state-of-the-art LLMs (e.g., Codex with 12B parameters) can already perform end-to-end input generation for fuzzing real-world systems (without any further fine-tuning). Also, our work shows for the first time that step-by-step prompt engineering [60] can substantially help boost fuzzing. Moreover, to our knowledge, we are the first to apply infilling models (e.g., InCoder) to directly perform mutation-based fuzzing [39, 50] to generate more diverse input programs in an evolutionary fuzzing loop.

In addition to using DL techniques for fuzzing, another very recently explored direction involves using LLMs for automated unit test generation, e.g., GitHub Copilot has been shown to be promising for such purposes [7]. Different from fuzzing which focuses on general approaches for testing complex real-world software systems, unit test generation involves targeting particular modules or functions. As such, unit test generation requires additional knowledge from the program under test such as callable modules (e.g., constructors) and functions. TeCo [52] fine-tunes the CodeT5 [83] model to perform test completion for any targeted method under

test and the test signature written by human developers. TestPilot [63] directly uses Codex by prompting with the source code and example usages of the method under test to automatically generate unit tests. Additionally, TestPilot also involves an adaptive component which re-generates failed unit tests by querying Codex given the error message. CodaMosa [41] combines traditional search-based software testing (SBST) [19, 20] with LLMs (e.g., Codex) for effective unit test generation. Such existing techniques on LLM-based unit test generation are project/system specific by requiring precise information such as detailed code units under test and/or dynamic test execution traces combined with prompting to elicit generation of unit tests. In contrast, our approach directly leverages the pre-training strategies of LLMs to auto-complete/infill code and therefore can easily generalize to arbitrary real-world systems such as compilers/interpreters of different programming languages, DB systems, SMT solvers, smart contracts, and additional popular libraries with sufficient code examples in the massive pre-training corpora of LLMs. Furthermore, such unit testing techniques can only assist developers and require manual interaction since even current largest LLMs (e.g., PaLM [14] and ChatGPT [62]) cannot reliably produce oracles for unit tests, while TitanFuzz on the other hand is a *fully automated* approach through the usage of effective fuzzing oracles (such as differential testing) at the system level, and has already detected various bugs for real-world systems. Additionally, TitanFuzz is the first work to demonstrate that LLMs can perform both generation-based [30, 80] and mutation-based [39, 50] fuzzing studied for decades [84], while being fully automated, generalizable, and applicable to domains challenging for traditional approaches (such as DL systems).

3 APPROACH

Figure 4 shows the overview of our TitanFuzz approach. Given any target API, TitanFuzz first uses a generative LLM to generate a list of high-quality seed programs for fuzzing (Section 3.1). This is done by providing the model with a step-by-step prompt [60] to generate code snippets that directly use the target API. For the generated seeds, we further apply an evolutionary fuzzing algorithm to iteratively generate new code snippets (Section 3.2). In each iteration, we start by selecting a seed program with a high fitness score from the seed bank. We systematically replace parts of the selected seed with masked tokens using different mutation operators (Section 3.2.1) to produced masked inputs. Mutation operators are selected using a multi-armed bandit algorithm [67] (Section 3.2.2) aiming to maximize the number of valid and unique mutations generated. Using the masked inputs, we leverage the ability of infilling LLMs to perform *code infilling* to generate new code that replaces the masked tokens (Section 3.2.3). For each generated mutant, we first filter out any execution failures and use our fitness function (Section 3.2.4) to score each mutant. We then place all generated mutants into the seed bank, and for future mutation rounds, we prioritize seeds that have a higher score, allowing us to generate a more diverse set of high-quality code snippets for fuzzing. Finally, we execute all generated programs using differential testing oracle on different backends (CPU/GPU) to identify potential bugs (Section 3.3).

While our approach is general for any pair of generative and infilling LLMs, in this work, we use Codex [12] and InCoder [21] as our generative and infilling models, respectively. Codex is a state-of-the-art generative code model based on the popular GPT-3 [10] architecture where the model weights are first initialized using GPT-3 weights trained on natural language text and then fine-tuned on a large corpus of open-source code files. Codex can be used to perform auto-completion where the input is simply a description of the task (known as a prompt [10, 43]). In TitanFuzz, we use Codex to automatically create the high-quality seed programs for our evolutionary fuzzing algorithm. To obtain mutations from the seed programs, we use the InCoder model to perform *code infilling*. Unlike the generative models such as Codex which only uses the context before, InCoder is able to *fill in* code in the middle by using both the context before and after. In TitanFuzz, we combine the power of both types of LLMs by first using the generative model (Codex) to produce high-quality seed programs and then using the infilling model (InCoder) to generate additional mutated programs.

3.1 Initial Seed Generation

To generate the initial seed programs for a target DL API, we first query the Codex model with a step-by-step prompt and sample multiple completions. Codex is trained using causal language modeling where the model aims to predict the next token using all previous generations. Given a training sequence of tokens $T = \{t_1, t_2, ..., t_n\}$, let $T_{<m} = \{t_1, t_2, ..., t_{m-1}\}$ be the token sequence generated so far ($m \leq n$) and P be the Codex model which outputs the probability of generating a token. The Codex loss function is defined as:

$$\mathcal{L}_{Codex} = -\frac{1}{n}\sum_{i=1}^{n} log\ (P\ (t_i \mid T_{<i})) \tag{1}$$

Figure 5 shows an example of the constructed prompt and model output. In the prompt, we wrap our task description in a docstring following [12]. More specifically, we include the target library (e.g., `TensorFlow`) and the target API signature (e.g., `tf.nn.conv2d(input, filters, ...)`) in the prompt. The API signature is automatically extracted from the API documentation with an HTML crawler. We also design a step-by-step instruction (i.e., `Task 1: ... Task 2: ... Task 3: ...` in Figure 5) to improve the model's performance following [2, 47, 53, 73]. More precisely, we instruct the model to perform three tasks sequentially: (1) import the target library; (2) generate input data; and (3) invoke the target API. The constructed prompt serves as the initial input for Codex and the raw seed programs are obtained by sampling the autocompletion from Codex.

3.2 Evolutionary Fuzzing

Algorithm 1 describes the main evolutionary fuzzing algorithm of TitanFuzz. We start by initializing the seed bank with the Codex generated seeds (Line 2). The seed bank maintains the list of code snippets that have been generated so far. Next, we initialize the prior distribution of each mutation operator, which we will use and update in the main loop for selecting mutation operators (Line 3). We then enter the generation loop by selecting a current seed for mutation according to a fitness score (Line 5). This seed selection process first prioritizes those with higher fitness scores by choosing the top N seeds with the highest fitness score. Out of the top N seeds, we perform a *softmax* [22] operation on their exact fitness scores to determine the probability of picking each seed.

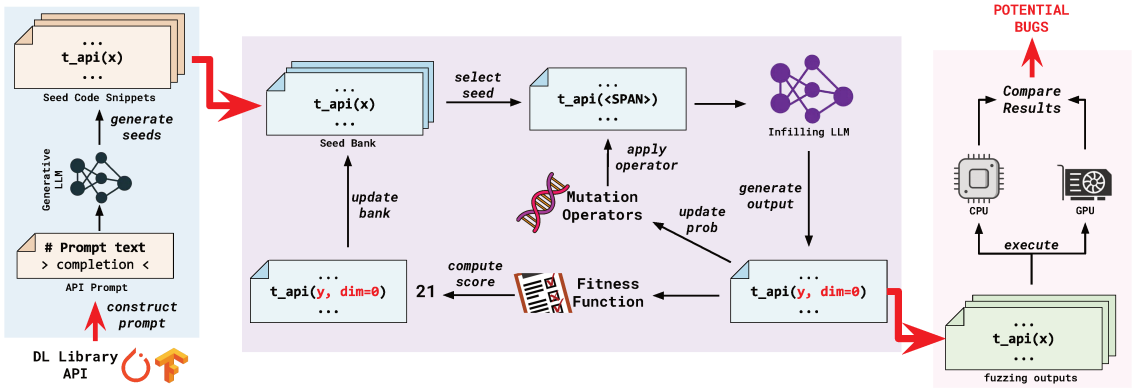

Figure 4: Overview of TitanFuzz

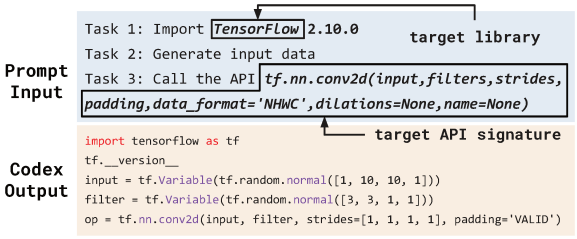

Figure 5: Example generation from the Codex model.

Besides the seed selection, we also decide which mutation operator will be applied on the seed (Line 6). Since the mutation operators that work well in helping the model to generate valid and unique mutations can be different for different target APIs, we dynamically learn the operator prioritization strategy using a Multi-Armed Bandit (MAB) [67] algorithm. Each mutation operator will mask out one or multiple segments of the current seed program with a special <SPAN> token (Line 7). The masked input is then fed into the InCoder model to generate code snippets that infill the masked regions (Line 8) with sampling. For each sample generated, we run and statically analyze the code snippet (Line 9). Specifically, we determine the code snippets that can be compiled (ValidSamples). We then update the posterior distribution of the mutation operator according to the number of valid and invalid samples it produced (Line 10). For each valid sample, we compute the fitness score according to our defined fitness function (FitnessFunction), which is designed to prioritize a diverse set of seeds that have a high number of unique interactions between different APIs, enabling us to discover more potential bugs. Using the fitness score, we add these samples into the seed bank for next iteration of seed selection (Line 12). Finally, when the time budget is exhausted, we terminate the generation and return the seed bank, now filled with a number of unique code snippets using the target API. Next, we will detail the key components for our algorithm.

3.2.1 Mutation Operators. We use four basic types of mutation operators: **argument, prefix, suffix and method**. Figure 6 shows the example masked inputs generated using each of our mutation operators. We start by identifying the target API (e.g., torch.mm) in the seed code snippet. Each mutation operator replaces a particular location of the chosen seed code with a masked span token (<SPAN>). For example, the argument-replacement mutation operator will

Algorithm 1: Evolutionary fuzzing algorithm

```
1  Function EvoFuzz(API, Seeds, T_Budget):
       Input  : The test target API, the seed programs Seeds, the time budget T_Budget
       Output : The generated programs
2      SeedBank ← Seeds
3      InitializeMPrior ()
4      while T_Elapsed ≤ T_Budget do
5          CurrentSeed ← SelectSeed (SeedBank)
6          MutationOp ← SelectMutationOp ()
7          MaskedInput ← Mask (CurrentSeed, MutationOp)
8          Samples ← InCoder (MaskedInput)
9          ValidSamples, InvalidSamples ← Evaluate (Samples)
10         UpdateMPosterior (MutationOp, Count (ValidSamples), Count
             (InvalidSamples))
11         FitnessScore ← FitnessFunction (ValidSamples)
12         SeedBank ← SeedBank ∪ ValidSamples
13     return SeedBank
```

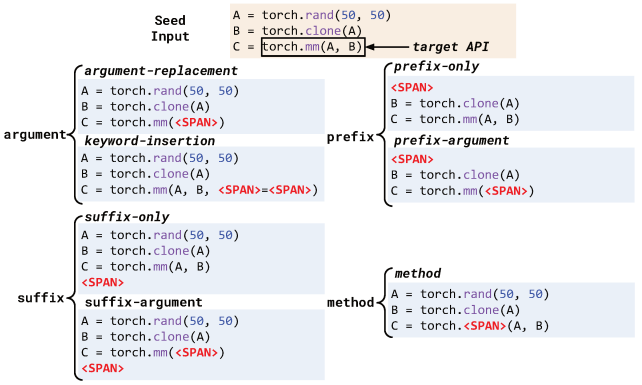

Figure 6: Mutation operators outputs (inputs for the model)

replace the argument of the target API call with the span token. The key idea is to create inputs which leverage the ability of LLMs to generate code snippets which replace each span token at the desired location, i.e., replacing the <SPAN> token with the model generation. We now define each of our mutation operator types:

Argument. The first is the argument-replacement mutation operator which replaces the API call arguments with the span token. Using this masked input, the model can fill-in different arguments to the API, generating unique and different program behaviors. Note, the argument-replacement mutation operator is not limited to just the target API and can be applied on any arbitrary library API in the code snippet. Furthermore, we use the keyword-insertion mutation operator which attempts to allow the model to generate additional keywords for a particular library API. Different from

Algorithm 2: Mutation operator selection algorithm

```
1  Function InitializeMPrior():
2      for m ∈ MutationOps do
3          m.S, m.F ← 1, 1
4  Function SelectMutationOp():
       Output: The chosen mutation operator m
5      for m ∈ MutationOps do
6          Sample θ_m ~ Beta(m.S, m.F)
7      m* = argmax_m θ_m
8      return m*
9  Function UpdateMPosterior(m, NumValid, NumInvalid):
10     m.S ← m.S + NumValid
11     m.F ← m.F + NumInvalid
```

the argument-replacement operator which replaces the entire argument list (which can include more than one argument), the keyword-insertion operator appends two span tokens at the end of the argument list. The two span tokens sandwich an equal sign, which signifies to the model that the two spans together should form a keyword. The equal sign is important, as without it, the model may think that an additional argument should be added here instead of a keyword. Keywords can unveil many interesting bugs because typically only the base case keyword values are tested; this is especially true for API sequences, as the combination of keywords in multiple API calls can lead to previously undiscovered bugs.

Prefix/Suffix. Just modifying the arguments of a specific API can only produce limited code mutants, since there are only a limited number of variables/literals available in the current scope for each argument in the API call. As such, to further augment the seed input, we apply prefix and suffix mutation operators, which choose a code segment (spanning one or multiple lines) before or after the target API invocation to insert the span token. In Figure 6, the prefix-only operator replaces the first line in the seed input with the span token. The model may then fill in this span token with different input generation methods instead of `torch.rand`. On the other hand, the suffix-only operator adds a span token after the target API. This allows the model to generate more code, potentially applying other DL APIs on the output of the prior code, covering additional interesting program behaviors/bugs. Since both the prefix and suffix can affect arguments of the API, we further combine the prefix and suffix together with the argument (i.e., the prefix-argument and suffix-argument operators) to allow more freedom for generating code before/after at the same time as argument generation.

Method. We include the method operator which replaces a randomly chosen library API method name with a span token. The idea is to generate a different library API invocation while using the existing arguments. The inspiration comes from prior work [16], which shows that it is common in DL libraries for related APIs to share the same input, and *borrowing* inputs from one API can help trigger bugs in its relational APIs. We can also leverage the method operator to generate more unique code snippets and test more APIs.

3.2.2 Mutation Operator Selection. We first formulate our mutation operator selection problem as a multi-arm bandit (MAB) problem [67], and then detail our algorithm. Our assumption is that the mutation operator that works well (e.g., generating more bug-triggering mutants) can be different for different DL APIs. Thus, we would like to adaptively learn the effectiveness of each operator in the generation loop. Intuitively, the validity of generated programs from a mutation operator can be a strong hint for prioritizing/de-prioritizing the operator. Thus, we model the mutation operator selection problem as a Bernoulli bandit problem [61] as follows:

Definition 1. ***Bernoulli Bandit.*** *Suppose there are K arms, and when played, each arm yields either a success or a failure. Each arm $i \in \{1, \ldots, K\}$ is associated with a success probability $\mu_i \in [0, 1]$, which is unknown to the agent. At each time step t, the agent will pull an arm i_t and observe a success/failure output drawn from the Bernoulli distribution $Ber(\mu_{i_t})$. The objective is to maximize the accumulated number of successes over T rounds of experimentation.*

Definition 2. ***Beta-Bernoulli Bandit.*** *For a Bernoulli bandit problem, let the agent adopt a Bayesian framework and choose the standard beta distribution [1] as the independent prior belief over each arm m. The probability density function of the beta distribution, for $0 \leq x \leq 1$, and parameters $\alpha > 0, \beta > 0$ is given by*

$$f(x; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

, where Γ denotes the gamma function [4]. If the prior is a $Beta(\alpha, \beta)$ distribution, the posterior will also be a beta distribution, with α or β increases by one with each observed success or failure, respectively.

For TitanFuzz, each mutation operator m can be seen as an arm associated with an unknown expected success probability, defined as the unique pass rate (percentage of generated code snippets when using a mutation operator that are valid and different from historical generations). When we"play" an arm at time t, we apply the mutation operator to generate programs, validate them, and interpret each program execution status as a success or failure.

To balance the exploitation and exploration trade-off in this beta-Bernoulli bandit problem, we leverage the classic Thompson Sampling (TS) algorithm [11, 67]. Algorithm 2 shows how TS, specialized for our mutation operator selection, proceeds. By initializing $m.S$ and $m.F$ to 1 (Lines 2-3), the algorithm assumes each arm m has prior $Beta(1, 1)$ (i.e., uniform distribution). After observing $m.S - 1$ successes and $m.F - 1$ failures of arm m, the posterior distribution of μ_m is updated as $Beta(m.S, m.F)$. To select an arm, we draw a sample θ_m from each of those posterior distributions (Line 6) and play the arm with the largest sampled value (which indicates that it has the highest probability of having the highest success rate). After generation using LLMs, we then update the posterior of the chosen mutation operator based on the execution statuses of generated programs (Lines 10-11). Compared to randomly picking mutation operators to use, this approach allows us to identify mutation operators that help generate more valid and unique code snippets. Note the *best* mutation operators can be different for different target APIs, therefore we start a separate MAB game and re-initialize the operator prior distribution for each end-to-end run of the evolutionary fuzzing targeting one API.

3.2.3 Code Generation. After using the selected mutation operator to produce the masked input for code generation, we use InCoder to generate new code to fill-in the masked-out location. InCoder is trained using causal masking objective [21] to perform *code infilling* by using bi-directional context to determine reasonable code snippets to place in the middle. Let $T_{masked} = \{T_1, T_2, \ldots \texttt{<SPAN>}, T_n\}$ be training code tokens where span mask tokens are inserted, $M = \{m_1, m_2, \ldots, m_k\}$ be the tokens masked out,

$M_{<g} = \{m_1, m_2, ..., m_{g-1}\}$ be the list of tokens generated so far ($g \leq k$), P be the InCoder model which outputs the probabilities of generating a token. The loss function of InCoder can described as:

$$\mathcal{L}_{\text{InCoder}} = -\frac{1}{k}\sum_{i=1}^{k} log\ (P\ (m_i \mid T_{masked},\ M_{<i})) \tag{2}$$

We leverage the ability of InCoder to generate arbitrary but related code snippets to target intricate library API relationships as the model can learn from the context (surrounding code which already focuses on library APIs) to generate additional APIs/code. As DL library APIs operate on Tensors, a shape mismatch (e.g., vector addition with incorrect dimensions) can lead to runtime errors. Traditional mutations (e.g., changing random code elements) do not work in generating valid DL programs since they can easily cause runtime errors by incorrectly mutating the input and argument space, and also cannot easily ensure the semantic validity [38, 55] of the generated programs due to the dynamically typed language. In contrast, InCoder is trained on millions of code snippets, many of which contain usages of these library APIs [28]. This allows InCoder to directly provide interesting/correct code based on the bi-directional context and generate potentially valid DL programs. Using the code generated by InCoder, we place them directly into the `<SPAN>` of the mutated input to produce new code snippets.

3.2.4 Fitness Function. Similar to the mutation operator, the seed programs we choose to mutate over are also important to generate unique and interesting code snippets for fuzzing. As such, we design a fitness function and score to rank each generated program. We apply static analysis to calculate the fitness scores of each test program. The intuition behind the fitness calculation is to give higher scores to the generated mutation programs with deeper execution path, more diverse computation graph, and more complicated API invocations. Specifically, we consider the following features:

- **Depth of dataflow graph.** We statically analyse the dataflow of variables within the generate code snippets to build a dataflow graph with each edge representing data dependencies between two operations. The depth of the dataflow graph (**D**) is defined as the maximum number of edges in any path of the graph.
- **Number of API calls.** We count the number of unique library API calls (**U**) that exist within each code snippets. Since LLM tends to generate many code snippets where code line(s) are repeated, we also count and penalize the number of library APIs that are repeatedly called with the same inputs (**R**).

Combining all these factors, given generated code snippet C, we define our fitness function to be:

$$\textbf{FitnessFunction}(C) = \mathbf{D} + \mathbf{U} - \mathbf{R} \tag{3}$$

According to the formula, TitanFuzz favors input programs involving long-chained API sequences and more unique APIs. In this way, it allows us to cover more interactions between different APIs, potentially triggering more interesting program behaviors/bugs. Meanwhile, using only the first two sub-terms would cause TitanFuzz to cover longer and longer API sequences with repeated API calls, making the fuzzing process less efficient. Therefore, TitanFuzz further penalizes the sequences with repeated API calls.

3.3 Oracle

After the generation loop, we leverage differential testing oracle to detect bugs by running the generated code snippets on two separate backends. In short, we execute the generated code snippets on CPU and GPU, record all the variables including the intermediate ones, and detect potential bugs. We focus on the following bug types:

Wrong-Computation. We compare the values of all intermediate variables across the two execution backends and find wrong-computation when values are significantly different. Due to the non-deterministic nature of certain computations leading to slightly different results on CPU or GPU, we follow previous work [72] and use a tolerance threshold to check if values are significantly different. Difference in computed values can indicate a potential semantic bug in different backend implementations of a library API or interactions between different APIs.

Crash. During program execution, we also detect unexpected crashes, e.g. segmentation faults, aborts, `INTERNAL_ASSERT_FAILED` errors. Such crashes indicate failures to check or handle invalid inputs or corner cases, and can lead to security risks.

4 EVALUATION

We aim to investigate the following research questions:

- **RQ1:** How does TitanFuzz compare against existing DL library fuzzers?
- **RQ2:** How do the key components of TitanFuzz contribute to its effectiveness?
- **RQ3:** Is TitanFuzz able to detect real-world bugs?

4.1 Implementation

For seed generation, we use the Codex Completion model with `code-davinci-002` engine to sample 25 programs for each API. Since the Codex model is not open-sourced, we access it by interacting with the Codex API through HTTP requests from Python. Our default setting for code completion for Codex uses top-p (nucleus) sampling [31] with p = 0.95 following previous studies [12, 21], and max_tokens=256, $temperature$=0.4 tuned for our task. Since we pose a maximum token limit to Codex model for code completion, the generated program can end with an incomplete line. Thus, for each Codex-produced program, we iteratively remove the last line of the program until the syntax parsing succeeds. For fuzzing, we choose N=10 for seed selection and use the PyTorch implementation of the InCoder 1.3B model on Hugging Face [33]. Our default setting when using InCoder uses temperature = 1 with default settings of top p = 0.95 from previous studies [21]. We apply code filtering to remove unnecessary code generated by the model such as print statements. Furthermore, we apply dataflow analysis to perform dead code elimination.

4.2 Experimental Setup

Targeted DL libraries. We include both PyTorch (v1.12) [57] and TensorFlow (v2.10) [66], since they are two of the most popular DL libraries and are widely studied in prior DL library testing work [70, 72, 77].

Fuzzing budget. By default, we use a one-minute fuzzing budget per API for all possible APIs of both studied libraries. Meanwhile,

for RQ2, we randomly sample 100 public APIs in each library and conduct the ablation study experiments for five times and report the average following prior work [37]. Also, for RQ3, we extend the fuzz budget to four-minute per API for maximal bug finding.

Environment. We use a 64-core workstation with 256 GB RAM and running Ubuntu 20.04.5 LTS with 4 NVIDIA RTX A6000 GPUs. We use the coverage.py [3] tool to measure Python code coverage.

4.3 Metrics

Number of detected bugs. Following prior work on fuzzing DL libraries [56, 70–72, 77], we report the number of detected bugs.

Code coverage. Code coverage has been widely adopted in software testing and recently DL library/compiler testing [25, 45, 72]. We follow recent DL library fuzzing work (Muffin [25] and FreeFuzz [72]) and use line coverage.

Number of covered APIs. Following prior work [16, 72], we report the number of covered APIs as another important metric of test adequacy in DL libraries which typically have thousands of APIs.

Number of unique valid programs generated. 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 code snippets that have already been generated and only consider unique programs.

Execution time. Since TitanFuzz uses LLMs as the generation engines, it may take more time than existing fuzzers. As such, we also record the execution time following prior work [45, 71, 72].

5 RESULT ANALYSIS

5.1 RQ1: Comparison with Prior Work

We compare TitanFuzz against both state-of-the-art API-level (FreeFuzz [72], DeepREL [16]) and model-level (LEMON [71], Muffin [25]) fuzzers for testing DL libraries. Table 1 presents the number of library APIs covered by all studied techniques on TensorFlow and PyTorch. We run each tool with its default setting, and since LEMON and Muffin do not support PyTorch models, we only report their results on TensorFlow. Column **Total** presents the total number of APIs in each DL library. Note that we excluded the deprecated APIs and compatibility APIs in TensorFlow as they are no longer actively maintained by developers.

We observe that TitanFuzz is able to cover 2215 and 1329 APIs in TensorFlow and PyTorch, *achieving the highest number of APIs covered compared to state-of-the-art techniques.* TitanFuzz increases the number of APIs covered by 91.11% and 24.09% compared to the best-performing baseline DeepREL. Compared with model-level fuzzing techniques (LEMON and Muffin), LLM can greatly outperform them in terms of the number of covered APIs. This is due to the fact that model-level fuzzers use complete DL models that are implemented using a small set of layer-wise APIs such as `Conv2d`. On the other hand, TitanFuzz is able to generate arbitrary code through the use of both generative (Codex) and infilling (InCoder) LLMs to achieve state-of-the-art results in terms of API coverage.

Table 1: Comparison on API coverage

	TitanFuzz	DeepREL	FreeFuzz	Muffin	LEMON	Total
TensorFlow	2215	1159	581	79	35	3316
PyTorch	1329	1071	468	-	-	1593

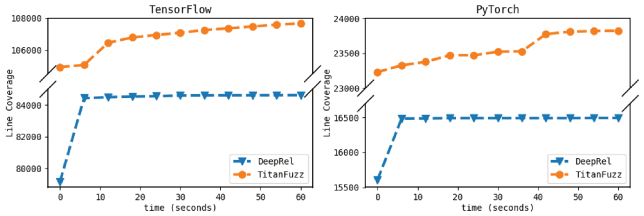

Figure 7: Coverage trend against DeepREL

Table 2 presents the overall code coverage rate. We choose the best-performing API-level and model-level baselines DeepREL and Muffin according to API coverage and run with their default settings. We observe that TitanFuzz significantly outperforms both DeepREL and Muffin, *achieving the state-of-the-art result of 20.98% and 39.97% line coverage on PyTorch and TensorFlow.* Compared with DeepREL, we increase the coverage result by 50.84% and 30.38% on PyTorch and TensorFlow. The time cost of TitanFuzz is higher due to the larger number of tested APIs and the use of LLMs. However, we observe that simply running TitanFuzz with only seed generation and targeting only the APIs that are covered by DeepREL (Row **TitanFuzz-seed-only (w/ DeepREL APIs)**) can already greatly outperform DeepREL with even less time, showing the power of directly using LLMs to produce high-quality seeds.

Figure 7 further shows the coverage trend of TitanFuzz against the best baseline DeepREL as we increase the time spent on fuzzing each target API. In this experiment, we run both techniques with a one-minute time budget for each API. We note that the DeepREL coverage barely increases after around 10 to 20 seconds of fuzzing. On the other hand, TitanFuzz does not suffer from the same coverage saturation. Even after 50 seconds of fuzzing, TitanFuzz can still generate new programs that improve coverage. We attribute this to both the usage of LLM to perform infilling and our *guided* seed and mutation operator selection in the generation process.

Table 2: Comparison with the best existing techniques

	PyTorch		TensorFlow	
	Coverage	Time	Coverage	Time
DeepREL	15794 (13.91%)	5.1h	82592 (30.65%)	9.3h
Muffin	-	-	79283 (29.42%)	6.8h
TitanFuzz-seed-only (w/ DeepREL APIs)	18447 (16.25%)	3.4h	89048 (33.05%)	4.9h
TitanFuzz-seed-only (w/ all APIs)	22584 (19.89%)	5.1h	103054 (38.35%)	11.9h
TitanFuzz	23823 (20.98%)	9.9h	107685 (39.97%)	21.1h

5.2 RQ2: Evaluation of Key Components

5.2.1 Seed Generation. We first study the various design choices for our seed generation which uses the Codex model with a carefully designed input prompt. The goal of the seed generation is to provide high-quality programs for as many APIs as possible. Therefore, we compare several variants of the input prompt and different Codex model hyperparameter values.

Figure 8 shows the API coverage and number of unique valid programs with different temperatures and prompts. **TitanFuzz** represents the default strategy presented in Section 3.1, where the prompt includes three steps to first import the DL library, generate input, and then call the target API. We also include the full API signature in the prompt to provide syntax guidance. **TitanFuzz-sig.**

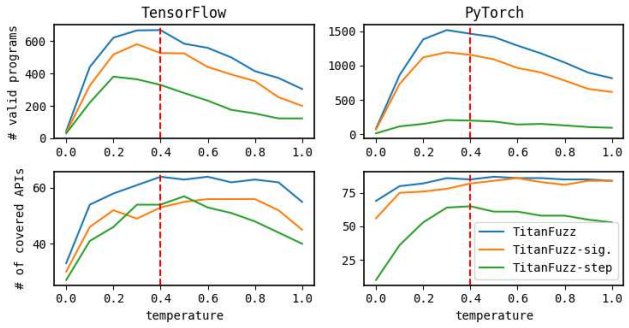

Figure 8: Codex seed generation trend

only provides the API name instead of API signature in the prompt (e.g. replacing the entire target API signature with `tf.nn.conv2d` in Figure 5). **TitanFuzz-step** removes the first two steps (import library and input generation) and only asks Codex to call the target API from the specified library version. First, we find that our default sampling temperature value of 0.4 (red dotted line) provides a good balance in terms of generating more valid programs and also covering more unique APIs on both PyTorch and TensorFlow. Second, we observe that by adding step-by-step instructions (first import the library and generate input data) to the prompt, we can substantially improve Codex's performance in both the number of unique valid programs generated and API coverage, demonstrating the power of prompt engineering for fuzzing for the first time. Furthermore, by adding the API signature to the prompt, we provide Codex with valuable information regarding the input parameter space to help Codex generate more valid programs.

5.2.2 Evolutionary Generation. Next we look at the different factors and choices in the evolutionary fuzzing algorithm.

Mutation Operators. We examine the effectiveness of each of our additional mutation operator types (except the default argument operators). Table 3 shows results when we remove each type from TitanFuzz. Column **Valid** considers only unique valid programs and Column **All** also includes programs with runtime errors. We observe that the highest number of unique programs and coverage is obtained when we use full set of mutation operator. This shows that each of the mutation operators can help in producing more unique programs and covering additional lines.

Table 3: Ablation study of operators

Variants	PyTorch				TensorFlow			
	# Unique Prog.		Coverage		# Unique Prog.		Coverage	
	Valid	All	Valid	All	Valid	All	Valid	All
TitanFuzz	**6969**	**18245**	**17411**	**17957**	**5173**	**16865**	**84447**	**86536**
-Suffix	5770	15813	16709	17691	4642	14501	81145	85294
-Method	6239	16943	16886	17615	3492	12519	83405	85454
-Prefix	6211	17082	17075	17797	3359	12345	83435	85645

Fitness Function. We compare our default fitness function against its variants, as well as random selection (**Random**) and a simplistic coverage guided [18, 35, 42, 50] (**Coverage**) baseline in Table 4. The fitness function variants are constructed by removing each subterm from the original fitness function (Equation 3). We observe that our chosen fitness function (**D+U-R**) is able to achieve close to the highest coverage and number of unique programs generated for both TensorFlow and PyTorch. Compared to random selection, our chosen fitness function approach is able to obtain higher coverage. This is due to the fitness function's ability to *guide* the fuzzing process towards using seeds with more unique APIs and longer chained API sequences, leading to covering more lines of code. Compared to our coverage-guided baseline which only adds programs with new coverage to the seed bank for later mutation, our fitness function has minimal additional overhead. This allows TitanFuzz to spend more time on the generation, leading to not only higher coverage but also more unique code snippets for testing.

Table 4: Ablation study of fitness function

Variants	PyTorch				Tensorflow			
	# Unique Prog.		Coverage		# Unique Prog.		Coverage	
	Valid	All	Valid	All	Valid	All	Valid	All
D+U-R	**6960**	**18245**	17411	17957	**5173**	**16865**	**84447**	**86536**
D+U	5817	15609	**17725**	**18415**	2993	11253	82963	85455
D-R	5872	16916	17229	18046	2876	11861	83563	85599
U-R	6234	17321	16894	17820	4315	15495	84057	86286
Random	7288	20720	16674	17586	3274	13237	83440	85045
Coverage	5098	15300	16715	17617	3210	12880	83030	84194

Operator Selection Algorithm. We compare our default Thompson Sampling operator selection algorithm (**TS**) with a uniformly random selection baseline (**Random**). Table 5 summarizes the results. The TS bandit algorithm helps to generate more unique valid programs and achieve higher code coverage in both libraries compared to the random strategy. Specifically, the TS strategy can generate around 2X more valid unique programs for TensorFlow; in PyTorch, although TS can generate fewer unique programs in total, it can still produce 12.5% more valid ones, demonstrating the effectiveness of our MAB-based operator prioritization.

Table 5: Evaluation of operator selection algorithms

Library	Algorithm	#Unique programs		Coverage	
		Valid	All	Valid	All
PyTorch	TS	**6960**	18245	**17411**	**17957**
	Random	6185	**18504**	17003	17683
TensorFlow	TS	**5173**	**16865**	**84447**	**86536**
	Random	2612	11816	83238	85469

InCoder vs Codex. Lastly, we also take a closer look at the contribution of both Codex and InCoder in generating unique test programs (**# Unique Prog. per API**) and time cost per unique program (**Time**) in Table 6. We observe that while Codex can provide high-quality seed programs, it is relatively slow compared to the smaller InCoder model, demonstrating the benefits of leveraging infilling LLMs and evolutionary mutation to further complement the powerful but costly large generative LLMs for fuzzing.

Table 6: Generation efficiency of Codex and InCoder

Library	Model	# Unique Prog. per API		Time per Prog. (s)	
		Valid	All	Valid	All
PyTorch	Codex	13.55	23.16	0.82	0.48
	InCoder	92.38	450.68	0.51	0.10
TensorFlow	Codex	6.85	22.26	1.69	0.52
	InCoder	67.17	358.06	0.67	0.13

5.3 RQ3: Detected Bugs

Table 7 summarizes the statistics of bugs detected by TitanFuzz. In total, TitanFuzz detected 65 bugs, with 55 confirmed (including 20 crash and 35 wrong-computation bugs), including 44 confirmed as

```
input_file = ['https://.../iris_training.csv',
              'https://.../iris_test.csv']
training_dataset = tf.data.experimental.
                   CsvDataset(input_file[0], ..., header=True)
for e in range(10):
    # The following operation is causing Check Fail
    training_dataset = training_dataset.shuffle(1000).repeat().batch(512)
```

Target API: tf.data.experimental.CsvDataset
Catch: Check failed: 0 <= new_num_elements ... (core dumped)

a)

```
x = torch.randn(10, 10).log() # x contains NaN
y = torch.histc(x, bins=10, min=0, max=1)
# On CPU: [48, ...] counts all NaN
# On GPU: [2,  ...] does not count any NaN
```

High Priority

Target API: torch.histc
Catch: Inconsistency between GPU and CPU

b)

```
indices = tf.constant([1, 2, 3, 4])
data = [1.0, 2.0, 3.0, 4.0]
output = tf.raw_ops.ParallelDynamicStitch(indices=indices, data=data)
# On CPU: [7.6904807, ...] out-of-bound read
# On GPU: [0, ...]
```

Security Vulnerability

Target API: tf.raw_ops.ParallelDynamicStitch
Catch: Inconsistency between GPU and CPU

c)

```
X = tf.constant([[1, 2, 3], [4, 5, 6]], dtype=tf.int32)
Z = tf.bitwise.right_shift(X, -1)
# On CPU: [[1, 2, 3], [4, 5, 6]]
# On GPU: [[0, 0, 0], [0, 0, 0]]
```

Implementation -defined

Target API: tf.bitwise.right_shift
Catch: Inconsistency between GPU and CPU

d)

Figure 9: Bugs detected by TitanFuzz

previously unknown bugs (21 of which already fixed). Out of the 55 confirmed bugs, only 9 can be also found by the studied API-level fuzzers while none can be found by model-level fuzzers. Notably, 10 confirmed bugs are found by directly using the Codex generated seeds without any mutation. We next present example bugs that can only be detected by TitanFuzz, as well as one rejected bug:

Table 7: Summary of detected bugs

	Total	Confirmed	Unknown (Fixed)	Rejected
PyTorch	37	32	25 (13)	5
TensorFlow	28	23	19 (8)	5
Total	65	55	44 (21)	10

Figure 9a shows an overflow bug found when we repeatedly batch an instance of `CsvDataset`. The code incorrectly crashes with `Check failed` message when it instead should throw a catchable overflow exception. What makes this bug hard to detect is that we first must create the dataset, which TitanFuzz correctly generates by calling `CsvDataset` (the two hyperlinks are in fact valid links to obtain data). Furthermore, the bug is triggered by using the `for` loop to repeatedly call the `batch` function. This type of unique input generation and program structure (`for` loop) makes it impossible for previous fuzzers to generate this test program. TitanFuzz through the use of LLMs can successfully generate the dataset creation code and also Python-specific code (e.g., `for` loop) to expose this bug.

Figure 9b shows a bug in the `torch.histc` API where on CPU the API incorrectly counts `NaN` values as part of the first bin in the histogram. This bug is only found through TitanFuzz as it relies on a chained API sequence of first generating the regular random input and then applying the `log` function which can generate `NaN` values for negative inputs. Due to the silent incorrect computation, PyTorch developers have labeled this as a **high priority** bug.

Figure 9c shows a bug when a certain output index is left unspecified for the API `ParallelDynamicStitch`. When running on CPU, this API can perform an out-of-bound read without throwing any exception. Although in theory, previous API-level fuzzers should be able to find this bug since it is isolated within a single API. In practice, this bug is missed since this API is not covered by any of the previous techniques. This is due to the fact that the particular API (`ParallelDynamicStitch`) is not commonly used. As such, previous work cannot generate valid inputs to cover this API since they rely on known valid input/API pairs obtained from databases created by scraping open-source code [16, 72]. TitanFuzz is able to successfully cover this API through the usage of Codex (with prompt engineering) to provide high quality seeds. Due to the potential for exploiting this silent out-of-bound read, TensorFlow developers have labeled this bug as a **security vulnerability**.

Since we use the differential testing oracle by comparing the values obtained when running on CPU and GPU, there could be false positive cases where inconsistencies are tolerated or intended. Figure 9d shows an obvious inconsistency detected by TitanFuzz but rejected by developers. The cause is due to the inconsistency when using `right_shift` with a negative value. While it is not explicitly stated in the documentation, the developers commented on the issue report that because the CPU and GPU use different lower-level shifting operators, the output result when shifting with negative values will be dependent on the implementation.

5.4 Threats to Validity

Internal. The main threat to internal validity comes from the implementation of TitanFuzz. To address this threat, the authors carefully performed testing and code review to validate that it was correctly implemented. Regarding randomness, while we only conduct RQ1 experiments for one run due to the large number of APIs, we analyze the 5 runs for 100 APIs in RQ2. Targeting just 100 sampled APIs, TitanFuzz's code coverage is 17957(±887) for PyTorch and 86536(±1499) for TensorFlow, substantially outperforming the strongest baseline (DeepREL) with p-value<0.001 for both libraries.

External. The main external threat to validity originates from our studied benchmarks. We mitigate this by evaluating on two most popular DL libraries: PyTorch and TensorFlow. Our result shows that TitanFuzz achieves the state-of-the-art results on both libraries.

6 CONCLUSION

We propose and implement TitanFuzz, the first approach for fuzzing DL libraries via Large Pre-trained Language Models. TitanFuzz first uses a generative LLM (e.g., Codex) to provide high-quality seed programs through prompt engineering, and then leverages an infilling LLM (e.g., InCoder) to mutate seed programs with an evolutionary fuzzing algorithm. Our extensive evaluation on two popular DL libraries (PyTorch and TensorFlow) demonstrates that TitanFuzz significantly improves the number of covered library APIs and code coverage. Furthermore, TitanFuzz is able to detect 65 bugs, 44 of which are confirmed to be previously unknown. Overall, this work demonstrates a promising future of directly leveraging modern LLMs for fuzzing and testing in general.

ACKNOWLEDGEMENTS

We thank the reviewers for their insightful feedback and comments to improve this paper. This work was partially supported by NSF grants CCF-2131943 and CCF-2141474. We also acknowledge support from Kwai Inc., Google, and Meta.

REFERENCES

[1] 2022. Beta distribution. https://en.wikipedia.org/wiki/Beta_distribution.

[2] 2022. Codex Documentation - Best Practices. https://beta.openai.com/docs/guides/code/best-practices.

[3] 2022. Coverage.py. https://github.com/nedbat/coveragepy.

[4] 2022. Gamma function. https://en.wikipedia.org/wiki/Gamma_function.

[5] 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]

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. https://arxiv.org/abs/2108.07732

[7] Rahul Banerjee. 2021. Writing Better Tests with AI and GitHub Copilot. *Codecov* (2021). https://about.codecov.io/blog/writing-better-tests-with-ai-and-github-copilot/.

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

[9] 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/.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165.

[11] Olivier Chapelle and Lihong Li. 2011. An Empirical Evaluation of Thompson Sampling. In *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger (Eds.), Vol. 24. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2011/file/e53a0a2978c28872a4505bdb51db06dc-Paper.pdf

[12] 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).

[13] 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

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

[15] 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.

[16] 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

[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] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[20] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[21] 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).

[22] Josiah Willard Gibbs. 1902. *Elementary principles in statistical mechanics: developed with especial reference to the rational foundations of thermodynamics*. C. Scribner's sons.

[23] 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

[24] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'14)*. MIT Press, Cambridge, MA, USA, 2672–2680.

[25] 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

[26] 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]

[27] 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.

[28] 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* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

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

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

[31] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. arXiv:1904.09751.

[32] Zhicheng Hu, Jianqi Shi, YanHong Huang, Jiawen Xiong, and Xiangxing Bu. 2018. GANFuzz: A GAN-Based Industrial Network Protocol Fuzzing Framework. In *Proceedings of the 15th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '18)*. Association for Computing Machinery, New York, NY, USA, 138–145. https://doi.org/10.1145/3203217.3203241

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

[34] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando Mujica, Adam Coates, and Andrew Y. Ng. 2015. An Empirical Evaluation of Deep Learning on Highway Driving. arXiv:1504.01716 [cs.RO]

[35] K. Serebryany 2015. libFuzzer a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[36] Keras 2020. Keras. https://keras.io/.

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

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

[39] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 216–226.

[40] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language {Model-Guided}{JavaScript} Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. 2613–2630.

[41] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering*.

[42] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[43] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 3045–3059. https://doi.org/10.18653/v1/2021.emnlp-main.243

[44] 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.

[45] 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

[46] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[47] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2022. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* (sep 2022). https://doi.org/10.1145/3560815

[48] 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.

[49] Yang Liu. 2019. Fine-tune BERT for Extractive Summarization. arXiv:1903.10318.

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

[51] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (mar 1971), 151–165.

[52] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *45th International Conference on Software Engineering*.

[53] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. https://doi.org/10.48550/ARXIV.2112.00114

[54] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. 2020. BUSTLE: Bottom-Up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381* (2020).

[55] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–19.

[56] 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

[57] PyTorch 2018. PyTorch. http://pytorch.org.

[58] 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.

[59] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* (jan 2020).

[60] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. arXiv:2102.07350.

[61] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. 2018. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning* 11, 1 (2018), 1–96.

[62] John Schulman, Barret Zoph, Jacob Hilton Christina Kim, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, Rapha Gontijo Lopes, Shengjia Zhao, Arun Vijayvergiya, Eric Sigler, Adam Perelman, Chelsea Voss, Mike Heaton, Joel Parish, Dave Cummings, Rajeev Nayak, Valerie Balcom, David Schnurr, Tomer Kaftan, Chris Hallacy, Nicholas Turley, Noah Deutsch, Vik Goel, Jonathan Ward, Aris Konstantinidis, Wojciech Zaremba, Long Ouyang, Leonard Bogdonoff, Joshua Gross, David Medina, Sarah Yoo, Teddy Lee, Ryan Lowe, Dan Mossing, Joost Huizinga, Roger Jiang, Carroll Wainwright, Diogo Almeida, Steph Lin, Marvin Zhang, Kai Xiao, Katarina Slama, Steven Bills, Alex Gray, Jan Leike, Jakub Pachocki, Phil Tillet, Shantanu Jain, Greg Brockman, and Nick Ryder. 2022. ChatGPT: Optimizing Language Models for Dialogue. (2022). https://openai.com/blog/chatgpt/.

[63] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. arXiv:2302.06527 [cs.SE]

[64] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

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

[66] TensorFlow 2020. TensorFlow. https://www.tensorflow.org.

[67] WILLIAM R THOMPSON. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (12 1933), 285–294. https://doi.org/10.1093/biomet/25.3-4.285 arXiv:https://academic.oup.com/biomet/article-pdf/25/3-4/285/513725/25-3-4-285.pdf

[68] 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.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. (2017). arXiv:1706.03762.

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

[71] 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.

[72] 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

[73] 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).

[74] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*.

[75] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning. arXiv:2207.08281.

[76] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[77] 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*.

[78] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[79] 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.

[80] 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.

[81] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2020. XLNet: Generalized Autoregressive Pretraining for Language Understanding. arXiv:1906.08237.

[82] 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.

[83] 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 *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

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

[85] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 132–142.

[86] 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

Received 2022-11-10; accepted 2023-01-16