Your Fix Is My Exploit: Enabling Comprehensive DL Library API Fuzzing with Large Language Models

Kunpeng Zhang*, Shuai Wang*✉, Jitao Han†, Xiaogang Zhu‡, Xian Li§, Shaohua Wang†✉, and Sheng Wen§
*The Hong Kong University of Science and Technology
{zkp0625@outlook.com, shuaiw@cse.ust.hk}
†Central University of Finance and Economics
{hanjitao1@gmail.com, davidshwang@ieee.org}
‡The University of Adelaide
xiaogang.zhu@adelaide.edu.au
§Swinburne University of Technology
{xli1, swen}@swin.edu.au

***Abstract*—Deep learning (DL) libraries are widely used to form the basis of various AI applications in computer vision, natural language processing, and software engineering domains. Despite their popularity, DL libraries are known to have vulnerabilities, such as buffer overflows, use-after-free, and integer overflows, that can be exploited to compromise the security or effectiveness of the underlying libraries. While traditional fuzzing techniques have been used to find bugs in software, they are not well-suited for DL libraries. In general, the complexity of DL libraries and the diversity of their APIs make it challenging to test them thoroughly. To date, mainstream DL libraries like TensorFlow and PyTorch have featured over 1,000 APIs, and the number of APIs is still growing. Fuzzing all these APIs is a daunting task, especially when considering the complexity of the input data and the diversity of the API usage patterns.**

Recent advances in large language models (LLMs) have illustrated the high potential of LLMs in understanding and synthesizing human-like code. Despite their high potential, we find that emerging LLM-based fuzzers are less optimal for DL library API fuzzing, given their lack of in-depth knowledge on API input edge cases and inefficiency in generating test inputs. In this paper, we propose DFuzz, a LLM-driven DL library fuzzing approach. We have two key insights: (1) With high *reasoning* ability, LLMs can replace human experts to reason edge cases (likely error-triggering inputs) from checks in an API's code, and transfer the extracted knowledge to test other (new or rarely-tested) APIs. (2) With high *generation* ability, LLMs can synthesize initial test programs with high accuracy that automates API testing. DFuzz provides LLMs with a novel "white-box view" of DL library APIs, and therefore, can leverage LLMs' reasoning and generation abilities to achieve comprehensive fuzzing. Our experimental results on popular DL libraries demonstrate that DFuzz is able to cover more APIs than SOTA (LLM-based) fuzzers on TensorFlow and PyTorch, respectively. Moreover, DFuzz successfully detected 37 bugs, with 8 already fixed and 19 replicated by the developer but still under investigation.

I. Introduction

To date, deep neural networks (DNNs) and their enabled applications have been extensively utilized in various real-world scenarios, such as autonomous driving [26, 42, 44], software vulnerability detection [33, 36, 57, 62] and medical diagnosis [11, 52]. Typically, those DNN applications are built on top of deep learning (DL) libraries, such as TensorFlow [6] and PyTorch [4], which offer a comprehensive set of API functions to facilitate developing DNN models and execute them. Given that DNN applications have been actively employed in reliability-sensitive scenarios [7, 30, 51], it is demanding to thoroughly test DL libraries and uncover underlying bugs in those API functions. Among all popular methods, fuzzing is deemed the mainstream approach with high potential, given its high flexibility and capability in detecting real-world defects [12, 24, 25, 47, 49, 74].

While traditional fuzzing techniques have been used to find bugs in software [9, 22, 23, 43, 71], recent works show that they are not well-suited for DL libraries [14–18, 27, 60, 64]. In general, the complexity of DL libraries and the diversity of their APIs make it challenging to test them thoroughly. To date, mainstream DL libraries like TensorFlow and PyTorch have featured over 1,000 APIs, and this number continues to increase. Performing fuzzing on all of these library APIs is a challenging endeavor, particularly when taking into account the input data complexity, emerging data types, and diverse API implementation patterns. Recent DL library fuzzing works primarily rely on manually crafted mutators [14], which require expensive human efforts and are less scalable.

Large language models (LLMs) are transformer-based neural networks that have achieved state-of-the-art (SoTA) performance in a wide range of natural language and code processing tasks [13, 19, 21, 32, 34, 35, 58, 63, 67, 68]. It is shown that LLMs can reason and generate human-like code, given that they have been trained on large-scale corpora which often subsume common sense knowledge and programming expertise. Recent works [15, 16] use LLMs to perform DL library API fuzzing with promising results. However, we find that those emerging LLM-based fuzzers often suffer from a lack of in-depth knowledge on API input edge cases and inefficiency in generating test inputs (most test inputs are not helpful). This is mainly due to the fact that LLMs were merely employed to mutate API inputs using either scheduling algorithm-based schemes [15] or based on historical data [16]. As a result, they

✉Corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00041

essentially treat DL library API fuzzing as a *black-box problem*, where the underlying API implementations are *neither* well understood *nor* utilized.

Our Insights. This paper presents a novel and comprehensive DL library fuzzing approach, by bridging de facto LLMs with a "white-box view" of DL library APIs. We have two key insights: (1) LLMs manifest high *reasoning ability* and possess pre-knowledge over a considerable number of APIs. They can reason those API input checks (e.g., `TORCH_CHECK` in PyTorch) widely seen in API low-level code, and infer edge cases (likely error-triggering API inputs) accordingly. Moreover, these edge cases are transferable to effectively stress other similar APIs and likely uncover more bugs rapidly. (2) LLMs also manifest high *generation ability*, in that with proper guidance and feedback, they can synthesize programs invoking a target API with high accuracy, thus automating the API testing process with high efficiency.

With the above insights, we propose DFUZZ, a novel LLM-based fuzzing framework for DL libraries. DFUZZ features a three-step approach to perform effective API fuzzing, with the usage of LLMs in different steps. DFUZZ first employs LLMs to summarize input checks (e.g., `TORCH_CHECK`) found in the low-level code of a DL library API. DFUZZ forms edge cases that can provoke those checks, and further lift the edge cases to an abstract, context-free form that can be easily transferred to test other APIs with similar input types. Next, DFUZZ uses LLMs to synthesize initial test programs to invoke a target API, where we form a feedback-driven process to guide the generation. Third, with edge cases and initial test programs on hand, DFUZZ employs LLMs to synthesize diverse inputs to fuzz a given API. We further design a set of optimizations to make DFUZZ highly efficient and practical.

We evaluate DFUZZ to fuzz two mainstream DL libraries, TensorFlow and PyTorch. We compare DFUZZ with SoTA LLM-based fuzzers, TitanFuzz [15] and FuzzGPT [16], and SOTA type-aware mutation-based fuzzer, IvySyn [14]. We show that DFUZZ achieves higher API coverage with much lower LLM usage. In fuzzing PyTorch and TensorFlow, DFUZZ uses only 7.12% and 16.81% of TitanFuzz's LLM usage (in initial program generation) respectively, yet covers 126 and 125 more APIs than TitanFuzz, respectively. Moreover, we show that DFUZZ finds 37 bugs in the latest versions of PyTorch and TensorFlow, with 8 already fixed and 19 replicated by the developer but still under investigation. More than 20 bugs exist in previous versions of TensorFlow and PyTorch which have been extensively tested by TitanFuzz, FuzzGPT, and IvySyn, yet none of these fuzzers discovered them. In sum, our contributions are as follows:

- We for the first time advocate a *white-box view* in the context of LLM-based DL library API fuzzing. The high reasoning and generation abilities of LLMs facilitate inferring edge cases and generating test programs, which are essential for comprehensive DL library API fuzzing.
- We present a novel LLM-based fuzzing framework, DFUZZ [2], that implements the above insight. DFUZZ features a three-step approach to delivering fuzzing. DFUZZ features a set of design principles and optimizations to make it highly efficient and practical.
- Our results show that DFUZZ can consistently outperform SOTA (LLM-based) fuzzers, in terms of API coverage and bug finding. 37 bugs have been found in the latest versions of TensorFlow and PyTorch, with 24 bugs already existing in TensorFlow/PyTorch fuzzed by previous tools.

II. MOTIVATION

A. Related Work and Limitations

Fuzzing DL libraries is a challenging and demanding task faced by the community. Based on seed differences, existing research on fuzzing DL libraries can be categorized into two types: model-level and API-level fuzzers [15]. Model-level fuzzers primarily test APIs related to model construction and training, treating complete DL models as inputs while simultaneously testing multiple APIs [27, 29, 40, 48, 59]. The testing scope of API-level fuzzers encompasses all APIs within the target DL framework, generating programs for each API and subsequently applying mutations [14–17, 60, 64, 69].

DFUZZ belongs to the latter category, focusing on API-level fuzzing given its high comprehensiveness over APIs. However, with the high complexity and diversity of DL library APIs, we find that SoTA methods in this category feature a rather *opaque* understanding of the underlying API usage patterns and codebases, even if LLMs may have been employed. To elaborate, existing DL framework fuzzing methods can be categorized into the following three types:

① Scheduling algorithm-based approach, such as TitanFuzz [15]. Such a method pre-defines basic mutation operators, and then applies these operators to mutate programs based on a scheduling algorithm. However, the optimization goal of such methods is often not bug discovery, but rather metrics like the complexity of generated programs. The mutation is undirected and lacks guidance to effectively stress APIs. Importantly, given that there are often thousands of APIs in real-world DL frameworks, our tentative experiments show that such methods fail to allocate sufficient testing time to each API, making them hardly comprehensive to test each API instance. For example, in TitanFuzz's experiments, only 60 seconds were performed to each API [15], likely struggling to achieve good results.

② Approach based on predefined input mutators, such as IvySyn [14] and DocTer [64]. These methods require experts to analyze existing bug codes, summarize features, and thus construct suitable mutators. In comparison to ①, mutations based on predefined mutators are more directed and targeted. However, the construction of mutators is labor-intensive and time-consuming, and the mutators are often not comprehensive enough to cover all possible edge cases. In IvySyn, researchers manually analyzed 240 CVEs and summarized dozens of mutators. However, this relies on heavy expert experience and human resources for analysis, thus lacking automation and scalability. In DocTer, the tool relies on the provided documentation to extract input constraints for API functions.

③ Approach based on historical bug-triggering code, such as FuzzGPT [16]. Holistically, history-based methods can be seen

as an enhanced version of ①, where the mutation is guided by historical bug cases. Nevertheless, we find that such approaches require collecting a large number of error codes from the Internet; such error codes may be likely incomplete and not representative of the entire API input space. Moreover, there exists a noticeable gap between bug codes and test program generation. Bug codes often consist of a limited number of statements that are genuinely pertinent to the bug, whereas other statements are irrelevant and frequently lead to erroneous program generation. Also, due to the substantial differences in the syntactic forms and usages of APIs, error-triggering features gathered from existing bug codes may be likely unusable when fuzzing other APIs. DFUZZ achieves notably better results than FuzzGPT, as shown in our evaluation (Sec. V).

B. Insights Derived from a White-Box Perspective

Conceptually, we view that the limitations of existing methods can be addressed by shedding a "white-box" perspective on the DL library codebase. In general, DL frameworks like TensorFlow and PyTorch typically comprise multiple layers of abstraction, and at the lowest level, native low-level APIs implement specific operations (e.g., tensor operations). Importantly, these implementations include extensive checks on input parameters to ensure they meet the expected conditions of the API. In case of errors, corresponding error messages are generated. We see that these checks contain a rich set of information about edge cases — extreme or special situations of API inputs that are likely to trigger bugs. To ease understanding, we present a real example below.

```
1 Tensor& abs_(Tensor& self) {
2     TORCH_CHECK(!self.is_complex(), "In-place abs is not supported
3     for complex tensors.");
4     return unary_op_impl_(self, at::abs_out);
}
```

Fig. 1. Sample source code in PyTorch.

Real-World Example. Fig. 1 illustrates the low-level source code of PyTorch API `torch.Tensor.abs_`. This API computes the absolute value of a tensor and modifies the tensor in place without creating a new one. Since `torch.Tensor.abs_` does not yet support handling tensors of complex type (i.e., an "edge case"), line 2 checks whether the input tensor is complex; if so, an error is thrown.

The check statement (line 2) can be abstracted into an edge case, *a tensor parameter is a complex tensor*, which likely crashes the API without the check. Moreover, our manual analysis shows that when testing another API `torch.all`, which has the same input parameter type as `abs_`, feeding the edge case into `torch.all` triggers an unknown bug. This example implies that an edge case extracted from one API is often valuable for testing other APIs. We present the following definition and key observations that motivate our approach.

Edge Case. Through analyzing the source code of PyTorch and TensorFlow, we find that the source code of DL frameworks contains a rich set of information that can be used to infer edge cases. This offers a unique opportunity to leverage LLMs to extract edge cases from the source code, and then transfer the extracted knowledge to test a group of APIs with high similarity. Table I defines edge cases considered, which are three special conditions over API inputs. Using such edge cases to test APIs can effectively stress them and likely uncover defects, if the APIs are not properly handling those edge cases.

TABLE I
EDGE CASE CATEGORIES AND EXAMPLES.

Category	Input Example	Expected	Edge Case
Special Type	x is int	x is int	x is string
Abnormal Value	x is int	$x > 0$	$x < 0$
Special Type Attribute	x is tensor	x.dtype is int	x.dtype is float

Observation I: API Checks Reflect Edge Cases. In developing DL frameworks, developers commonly incorporate check statements to ensure that the input parameters meet the expected conditions of the API, or alert users if certain inputs are yet supported. For instance, in the PyTorch low-level code, we see many check statements, such as `TORCH_CHECK` and `AT_CHECK`, where each checks one or several edge cases. This suggests that checks contain rich information to reflect edge cases.

This paper extracts edge cases by analyzing check statements and then infer the edge cases based on the context, input types, and error messages. However, in practice, DL libraries like PyTorch and TensorFlow often have two kinds of check statements: 1) those encode the edge conditions directly, and 2) those encode expected conditions of the API inputs. DFUZZ indeed considers both types of check statements to extract edge cases: to extract the former, we directly ask the LLM to infer the edge case (Sec. III-C); to extract the latter, we ask the LLM to infer an edge case that violates the expected conditions (by slightly tweaking prompts used in Sec. III-C). To avoid verbosity, we unified the two types of checks as "edge cases" in this paper.

Observation II: APIs with Same Input Types Often Sharing Edge Cases. Importantly, the edge cases that an API needs to handle are often determined by their input types. For example, in the case of `torch.all` and `torch.abs_`, where both APIs take a tensor as its input, they share the same edge cases to handle undefined and empty tensors. We view this illustrates an important insight: if APIs have the same input parameter types, they shall likely share edge cases. Thus, we can transfer knowledge of edge cases learned from one API to test another API with the same input types. Moreover, since edge cases are determined by API input types, we see that knowledge transfer of edge cases also occurs in a *cross framework setting*, e.g., an edge case learned from PyTorch can be used to test TensorFlow APIs, and vice versa. We validate this insight empirically in Sec. V-B; we extract edge cases from PyTorch core libraries and use these edge cases to test TensorFlow to detect 27 bugs.

C. Pilot Study

Despite the promising observations, PyTorch/TensorFlow has thousands of APIs, where each API may contain multiple checks. Extracting the "edge case" from a check statement is not trivial, which requires reasoning the check context, input types, error messages, and possibly data flow constraints. Relying on manual analysis would require a significant amount of time

and effort, and certain pattern match (e.g., regular expression) based approaches are also not feasible.

Having that said, we see the encouraging potential of LLMs in this context. LLMs are trained on millions of lines of code available on the Internet, and are shown to manifest promising understanding and reasoning capabilities in relevant software engineering tasks like code completion, summarization, and bug fixing [8, 20, 28, 34, 39, 45, 46, 50, 54, 65, 66, 70, 73]. In fact, we use the code from Fig. 1 to form a prompt, and ask GPT-3.5 (ChatGPT) [1] to extract edge cases from line 2. We find that GPT-3.5 can output the edge case: *"Tensor self is a complex tensor"* in an accurate and succinct manner.

This above result is promising, and suggests that LLMs can be used to extract edge cases from check statements. At this step, we conduct a pilot study to quantify the feasibility of LLMs in extracting edge cases from DL framework source code. Without loss of generality, we take PyTorch and GPT-3.5 as representative examples to validate our approach. The PyTorch framework comprises front-end, back-end, and bindings. The front-end, primarily the Python APIs, empowers users to construct and debug DL models with ease. The back-end, centered around the C++ APIs, implements elementary operations. Through bindings, PyTorch's front-end interacts with the C++ implementation in the back-end, blending Python's user-friendly interface with C++'s high performance. In the back-end, `Aten (pytorch/aten/src/ATen)` serves as the underlying tensor library, supporting tensor operations of various purposes. Most PyTorch high-level APIs and functionality are built upon `Aten`, making it a core component of the entire framework.

We collect Aten's native functions (`ATen/native`) for our study. Aten's native functions serve as the contemporary approach for integrating operators and functions into `ATen`. We randomly select 50 functions from `ATen/native`, and for each function, we extract its function header and all check statements (in the form of `PYTORCH_CHECK`) to form a code block. Then, for each block, we ask GPT-3.5 four questions: (1) *"How many check statements are contained in each block?"* (2) *"What are the variables related to each check statement?"* (3) *"What are the types of variables related to each check statement?"* and (4) *"Extract the edge cases checked by each check statement."* We also manually analyze the selected code blocks to obtain the ground truth. Then, we assess the accuracy of LLM outputs, whose results are in Table II.

TABLE II
ASSESSING GPT-3.5 ACROSS FOUR TASKS TO COMPREHEND CHECKS.

Tasks	Accuracy (Success/Total)
The number of checks in a code block	100%(50/50[1])
The involved variables of a check	97.26%(71/73[2])
The involved variable types of a check	97.26%(71/73)
The edge cases corresponding to a check	95.89%(70/73)

[1] 50: There are a total of 50 code blocks.
[2] 73: There are a total of 73 checks among all 50 code blocks.

GPT-3.5 achieves an accuracy of over 95% on all four tasks. LLM can infer the meaning of callee functions based on the context provided. Among all 73 `TORCH_CHECKs`, 54 involve callee functions. The errors occurred only in analyzing two code blocks (including three checks). One error occurs because the target code block contains too much irrelevant information, leading the LLM to inaccurately analyze it. The other one was misled by the unclear error message string (*"target(3)"*) in the check. We confirm that without the error message string, the LLM can correctly analyze the edge case: *'grad_output' is a 3D tensor*. Overall, we interpret the pilot study as highly promising, and the results suggest that LLMs can be used to extract edge cases from DL framework source code with high accuracy.

III. DESIGN OF DFUZZ

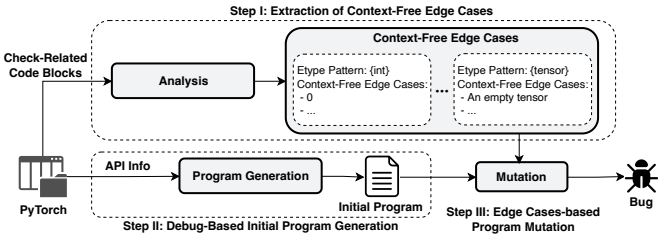

Fig. 2. The workflow of DFUZZ.

With the key insights presented in Sec. II, we now present the design of DFUZZ, a framework for comprehensive API fuzzing. As in Fig. 2, DFUZZ comprises three core steps:

Step I: Edge Case Extraction. Following the discussion in Sec. II, we use LLM to extract edge cases from the source code of DL library APIs. We identify and address several domain challenges in this step (see details in Sec. III-A), whose outputs will be edge cases (conditions over input parameters) in a *context-free* format. This way, the extracted edge cases can be smoothly used to test other similar APIs, despite various differences in the input syntactic forms like variable names.

Step II: Initial Program Generation. The next step prepares an initial test program that invokes a target API for fuzzing. Despite the fact that LLMs may fail to generate valid programs for certain APIs, we propose an iterative procedure, where errors encountered during execution are given to the LLM to regenerate the program and enhance chances of success.

Step III: Edge Cases-Based Mutation. With edge cases and initial test programs, DFUZZ performs edge cases-centric mutation. Per our observation, we deem an edge case beneficial for testing a target API, if the input types of the target API match or subsumes the types of the edge case. With this heuristic, we instruct the LLM to select beneficial edge cases for fuzzing, achieving high comprehensiveness in covering even rarely-tested and new APIs as long as their input types match certain edge cases. Whenever a bug is found, we record the case and report it to the developers for fixing.

Application Scope. DFUZZ is designed for testing DL libraries; it is fully automated and requires no additional resources besides the source code of the target DL framework. In contrast, recent works may require input templates pre-defined by human experts [14], or historical bug reports that are collected online [16]. DFUZZ can be used by developers and also normal users who want to test DL libraries (but may lack expertise

in library implementation). While DFUZZ is evaluated over two mainstream DL libraries, PyTorch and TensorFlow, it can be applied to other DL libraries when source code is available. DFUZZ extracts edge cases from the input checks available in the low-level code of the DL frameworks. We clarify that such inline checks are the primary way to handle edge cases in DL libraries. Therefore, analyzing those checks offers a comprehensive coverage of edge cases. Looking ahead, DFUZZ may be extended to test other types of software, such as those financial software, where inline checks appear also prevalent [55].

A. Edge Case Extraction

DFUZZ extracts edge cases from the source code of DL libraries and reuses them to test other APIs. The most straightforward approach is to split the check statements from the source code and ask LLM to reason edge cases reflected by these check statements. For instance, we can use prompts like "*Extract edge cases detected by the following check statement for fuzzing APIs*" to guide LLMs. Nevertheless, our tentative exploration shows the following challenge.

Transferability. LLMs often focus solely on triggering edge cases within a specific context, without considering whether these extracted edge cases can be applied to testing other APIs. For instance, in Fig. 1, the edge case extracted from line 2 is "`self` *is a complex tensor.*" However, this edge case can only be used when testing a specific API, `torch.Tensor.abs_`, due to the reason that other APIs may not have an input variable named "`self`". Even worse, our analysis focuses on the low-level implementation of APIs, where variable names may presumably differ from those at the high level. For example, the variable `self` at the high level (Python layer) corresponds to `Tensor input`. Therefore, directly utilizing those edge cases extracted by LLMs is not effective.

To address the challenge, we propose a three-step approach to extracting *context-free* edge cases. We refer the three steps as Code Extractor, Analyzer, and Standardizer. Code Extractor retrieves *check-related code blocks* from the DL framework source code, including check statements and the interface information (function name and input parameters) of their respective functions. Then, for each check statement within the extracted code block, Analyzer uses an LLM to extract a *context-based edge case* and the *context description*; the former refers to the involved input parameters and their description, and the latter encodes the names and types of the related parameters. We further convert context-based edge cases into context-free ones. For each context-based edge case and its context description, Standardizer first removes variable names in an edge case, and then uses the data types of the involved variables in an edge case (referred to as etype pattern) to form a context-free edge case. After analyzing all check statements, we cluster all (etype pattern, context-free edge case) tuples; tuples in each cluster share the same etype pattern.

1) Code Extractor: Code Extractor extracts code blocks related to edge cases, whose approach is generally applicable to various C++ libraries containing check statements. To ease presentation, consider the ATen library in PyTorch that uses the `TORCH_CHECK` macro to form check statements. We locate all `TORCH_CHECK` statements across source code files under ATen. While there exists a large number of `TORCH_CHECK` statements, we only need to consider cases where the input parameters of APIs are directly checked by `TORCH_CHECK` statements, given that these checks reflect edge cases that can be triggered by mutating input parameters. Furthermore, for each function, we assemble a *check-related code block*, which consists of the function interface and all `TORCH_CHECK` statements within the function. For instance, for the function `pool2d` in Fig. 4(a), the Code Extractor only retains the function interface and the targeted `TORCH_CHECK` statements, resulting in the check-related code block shown in Fig. 4(b).

2) Analyzer: After extracting the check-related code blocks, Analyzer extracts the context-based edge case for each check statement. To do so, one approach is to directly employ LLMs to analyze each check statement and determine its corresponding edge case. Though it is feasible, the analysis results are challenging to use subsequently because: (1) The randomness of the output format makes it difficult to extract information in a unified manner. For instance, our employed LLM (GPT-3.5) sometimes provides edge cases directly corresponding to each check, while at other times, it divides the analysis outputs into multiple paragraphs, making our subsequent analysis tedious. (2) Lacking relevant variable information makes it difficult to convert context-based edge cases into context-free forms. For example, the edge case extracted by LLM for Fig. 4(b) is *the "stride_arg" should be empty.* Yet, for other APIs, it is unclear what *"stride_arg"* refers to.

Prompt Design. To form context-free edge cases, we instruct the LLM to reason edge cases and the context description (variable names and types) associated with each check. The prompt is shown in Fig. 5. We divide the analysis of a `TORCH_CHECK` into four steps: 1. *"What variables does the* `TORCH_CHECK` *examine?"* 2. *"What are the data types of these variables?"* Here, we consider seven most commonly used types: (`Tensor`, `Int`, `Bool`, `Str`, `Float`, `Scalar`, `List`), but it is easy to extend to other types. 3. *"What edge cases does the* `TORCH_CHECK` *check?"* 4. *"To standardize the output and reduce irrelevant information, summarize the output in JSON format."* After each JSON item, we also provide the expected output format and examples. As an example, for the check-related block in Fig. 4(b), the analysis results are in Fig. 4(c).

3) Standardizer: Analyzer uses LLM to obtain the context-based edge cases. However, as previously noted, context-based edge cases are hardly useful. Many context-based edge cases are redundant, and since variable names remain in an edge case, they can only be used in specific syntactic contexts. Additionally, these context-based edge cases are extracted from low-level code, whereas our API fuzzing will be launched at the Python layer (noted in Sec. II-B).

Analyzer has prepared sufficient information for Standardizer to convert context-based edge cases in a format usable at the Python layer, i.e., context-free edge case. In particular, for each context-based edge case, we gather the types of relevant variables into a set, called the *etype pattern*. Etype pattern

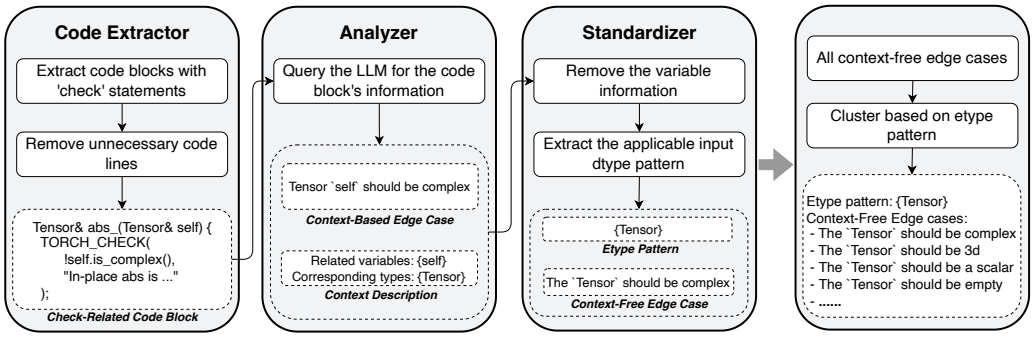

Fig. 3. The workflow for extracting edge cases.

```
Tensor pool2d(……
     IntArrayRef stride_arg, ……) {
     ……
     TORCH_CHECK(!stride_arg.empty(), "Stride cannot be empty!");
     ……
     return convert(v_output);
}
```

(a) Original codes

```
Tensor pool2d(……
     const IntArrayRef kernel_arg, ……) {
     TORCH_CHECK(!stride_arg.empty(), "Stride cannot be empty!");
```

(b) Extracted check-related code block

```
{
  "The 1st TORCH_CHECK statement": {
     "The statement": "TORCH_CHECK(!stride_arg.empty(),……)",
     "The checked variables": "stride_arg(List) ",
     "The checked edge case": "the `stride_arg` should be empty"
  }
}
```

(c) Analyzer sample outputs.

```
{
  "The 1st TORCH_CHECK statement": {
     "etype pattern": "{List}",
     "context-free edge case": "the `List` should be empty",
  }
}
```

(d) Standardizer sample outputs.

Fig. 4. Edge case extraction example.

Algorithm 1: Initial Program Generation Algorithm

Input: The target API, api. The interface information of the target API, api_info.
Output: A valid program that invokes api, P.

1 $init\_cnt \leftarrow 0$
2 **while** $init\_cnt < INIT\_MAX$ **do**
3 $dialogue \leftarrow [\,]$
4 $prompt \leftarrow construct\_prompt(api, api\_info)$
5 $dialogue.append(prompt)$
6 $P \leftarrow LLM(dialogue)$
7 $status, msg \leftarrow exec(P)$
8 $debug\_cnt \leftarrow 0$
9 **while** $debug\_cnt < DEBUG\_MAX$ **do**
10 **if** $status == SUCCESS$ **then**
11 $return\ P$
12 $error\_info \leftarrow get\_msg(msg)$
13 $dialogue.append(P)$
14 $dialogue.append(error\_info + \text{“}Regenerate\text{”})$
15 $P \leftarrow LLM(dialogue)$
16 $status, msg \leftarrow exec(P)$
17 $debug\_cnt \leftarrow debug\_cnt + 1$
18 $init\_cnt \leftarrow init\_cnt + 1$

reflects the types of input parameters that an API expects in accordance with the analyzed edge case. For example, suppose an edge case's etype pattern is {`tensor, tensor`}, then this edge case can be used to test APIs that expect two tensor inputs, such as `torch.add(input: Tensor, other: Tensor)`. And to obtain context-free edge cases, we replace all variable names in the context-based edge cases with the corresponding types. For example, for the context-based edge cases and the context descriptions extracted from `pool2d` in Fig. 4(b), we present the extracted etype pattern and context-free edge cases in Fig. 4(d). Finally, for all (etype pattern, context-free edge cases) tuples, we eliminate duplicates and cluster them based on etype pattern.

B. Initial Program Generation

To fuzz a target API i, we need to generate a test program that invokes i. The test program should be non-trivial, i.e., it can pass input values to the target API and check the output. With the high generation ability, we employ LLMs to synthesize the program, thus alleviating the burden of manually writing test programs. However, our tentative exploration implies that certain APIs are presumably not covered by LLM training data, especially for newly added or less commonly used APIs. This raises a hurdle for LLM to generate valid test programs.

To tackle this challenge, we divide the initial program generation into two stages: the generation stage and the debug stage. In the generation stage, we integrate the interface information of the target API into a prompt that is fed into the LLM to synthesize the initial test program. The prompt design is shown in Fig. 6. However, per our observation, generating a program that is both syntactically and semantically correct is challenging; the LLM-generated programs are often not executable. Therefore, we propose a debug stage, where we instruct the LLM to reflect the error message and regenerate the program.

As shown in Alg. 1, DFuzz first generates an initial program based on the interface information of the API and runs it to obtain the execution status (lines 3–7). If there is an error, we extract the error function along with the specific error message (line 12). Then, we feed this error information into the LLM to regenerate a program that can resolve the current error (lines 12–15). If debugging DEBUG_MAX times still fails to obtain a valid program, we ask LLM to regenerate a new initial program (back to line 5). This is important, as due to the stochastic nature of LLM, the same prompt can yield programs of different quality, helping us to avoid getting stuck in "local minima" (lines 9–17). Based on our preliminary exploration, we set INIT_MAX to 2 and DEBUG_MAX to 3 to balance the trade-off between the quality of the generated program and the time cost. See our evaluation in Sec. V-A.

C. Edge Case-Based Mutation

Through the previous steps, we have obtained the context-free edge cases and the initial program to invoke a target API

```
<Check-related code block>
```
For each statement containing TORCH_CHECK, answer the following questions:
(1) Which input variables does the TORCH_CHECK statement check? List the name of the input variables.
(2) What type are these variables respectively? Considering only the following variable types: {Tensor, Int, Bool, Str, Float, Scalar, List}.
(3) What edge case does the TORCH_CHECK statement check?
(4) Output the results in the following format:
{
  "The Nth TORCH_CHECK statement": {
    "The statement in the source code": "", # the statement containing TORCH_CHECK in the source code
    "The checked variables": "", # such as "var1(type1), var2(type2)".
    "The checked edge case": "", # follow "`<variable name>` should <requirement>", such as "the `input_t` should be a 2D tensor".
  }
}
```

Fig. 5. The prompt used by Analyzer.

```
Write a program to generate input data and invoke <API_NAME> to process input data.
Here are some information of this api:
```
<API_INFO>
```
```

Fig. 6. The prompt of initial program generation.

$i$. To determine which context-free edge cases are presumably beneficial for testing $i$, we collect the etype pattern of $i$, denoting a combinatorial set of its parameter types. For example, for `torch.add(input: Tensor, other: Tensor)`, its etype pattern is {`Tensor, Tensor`}. Then, we iterate each subset of the etype: {}, {`Tensor`}, {`Tensor`}, {`Tensor, Tensor`}, and search for all collected context-free edge cases to find a match. Once matched, we concretize the type names in the context-free edge cases with the names of the input parameters in the target API. For example, when testing `torch.all(input: Tensor)`, we then transform "*'Tensor' is a complex tensor*" to "*'input' is a complex tensor*".

Finally, we assemble the collected context-free edge cases, the definition of the target API $i$, and the corresponding initial program that invokes $i$ into a prompt; a sample is in Fig. 7. Then, we let the LLM generate a test program capable of triggering the edge cases, to see if it can uncover any bugs.

As shown in Alg. 2, DFUZZ begins by retrieving the initial program and etype pattern of the target API (lines 2–3). Then, leveraging the etype pattern, DFUZZ identifies the context-free edge cases applicable to the API (line 4). To reduce overhead, we select a subset of matched edge cases for mutation (line 5). More specifically, we categorize edge cases into two types based on the etype pattern: individual edge cases and compound edge cases. The former's etype pattern is only related to a single variable, while the latter involves multiple variables. In selecting individual cases, we prioritize variables that appear earlier in the API as they are considered more crucial. Particularly, for the first two variables, all individual edge cases related to them will be selected. 25% of the individual edge cases related to variables in positions 3–4 of the API will be randomly selected, while the remaining variables will be chosen at a rate of 12.5%. Due to the relatively limited quantity of compound edge cases, all matched compound edge cases will be selected for mutation. Finally, any bugs triggered by the mutated program are then collected for further analysis (lines 9–10). Following the convention in this field [15, 16, 60], we consider two types of bugs: crashes and CPU/GPU inconsistency. The former asserts that fuzzing triggers any "crashes," including aborts, segmentation faults, etc. For the latter, we run mutated programs separately on the CPU and GPU to check if the outputs are consistent.

**Algorithm 2:** Edge Case-Based Mutation Algorithm

**Input:** The target API, $api$. A library of context-free edge cases, $CEC$.
**Output:** A set of programs that trigger bugs, $BUG$.

```
1  Bugs ← [ ]
2  P_i ← get_init_program(api)
3  etype_pattern ← get_etypes(api)
4  matched_edge_cases ← match(etype_pattern, CEC)
5  selected_edge_cases ← select(matched_edge_cases)
6  for edge_cas in selected_edge_cases do
7  |   P_m ← mutation(P_i, edge_case)          ▷ Based on Figure 7
8  |   status, msg ← exec(P_m)
9  |   if status == BUG then
10 |   |   Bugs.append(P_m)
```

```
1. The initial program that invoke `<API>`:
```
<INIT_PROGRAM>
```
2. The information of `<API>`:
```
<API_INFO>
```
Mutate the initial program to generate a program satisfy that following requirements:
<Edge Case>
```

Fig. 7. The prompt of mutation.

IV. Implementation

The core testing process of DFUZZ is implemented in Python with approximately 4,000 lines of code; see our artifact at [2].

Check Identification. Without losing generality, we extract edge cases from the checks found in the source code of PyTorch. We see that PyTorch inline checks are well-structured, in the format of `TORCH_CHECK`, making it easier to analyze and validate. Also, PyTorch updates frequently, resulting in considerably more checks than other DL libraries on the market.

Per our investigation, PyTorch's tensor operation library, Aten, contains a substantial number of checks over different API input types and attributes. This is reasonable: Aten is one core library of PyTorch that has been developed and constantly updated by the PyTorch team for decades. Aten contains a wide range of computation operations, such as matrix multiplication, convolution, and activation functions. These operations are the foundation of PyTorch's high-level APIs, such as `torch.nn` and `torch.optim`. Thus, we decide to use PyTorch's ATen library (`pytorch/aten/src/ATen/native`) to extract edge cases. We find that using this library offers a good balance between the number of checks (large enough to cover a wide range of APIs) and the complexity of the code (not too complex to analyze).

We extracted a total of 20 kinds of etype patterns and their associated 132 edge cases. The extracted etype patterns include not only 7 basic types (`Tensor`, `Int`, `Bool`, `Str`, `Float`, `Scalar`,
```

`List`), but also 13 compound types composed of multiple basic types, such as (`Tensor_1`, `Tensor_2`), (`Int_1`, `Int_2`), and so on. In compound types, there are often constraints between variables, such as *Tensor_1 has a larger last dimension than Tensor_2* or *Both integers Int_1 and Int_2 are negative*. With these extracted context-free edge cases, we conducted tests on the APIs of PyTorch in Sec. V. Moreover, it is important to note that the extracted types and edge cases are transferable to other DL libraries; we also test them on TensorFlow in Sec. V.

V. Evaluation

We aim to answer the following research questions: **RQ1**: How does DFuzz perform in terms of API coverage? **RQ2**: How many (new) bugs can be discovered by the DFuzz? and **RQ3**: how effective is DFuzz when using alternative LLMs? We also conduct an ablation study in **RQ4**.

DL Libraries. We test two mainstream DL libraries, PyTorch and TensorFlow, for the following two reasons: (1) PyTorch and TensorFlow are the most widely-used DL libraries, and bugs discovered within them hold greater value. (2) PyTorch and TensorFlow have already been tested by many fuzzing tools. If we can still uncover long-standing bugs from them, it convincingly shows the effectiveness of DFuzz. As a fair comparison, we use a consistent set of target APIs with TitanFuzz and FuzzGPT. In **RQ1**, we test API coverage using the same versions of PyTorch (v1.12) and TensorFlow (v2.10) as those in the TitanFuzz and FuzzGPT papers. In **RQ2**, we conduct tests on the latest versions of PyTorch (v2.2.1) and TensorFlow (v2.15) to uncover previously unknown bugs.

Environment. All evaluations are conducted on a system with 256 GB RAM and running Ubuntu 20.04 with one NVIDIA GeForce RTX 3090 GPU.

LLMs. The current prototype of DFuzz uses GPT-3.5 (`gpt-3.5-turbo-1106`) for various LLM-based reasoning and generation tasks. We set the temperature to 0 to get the best results. In **RQ3**, we study the feasibility of replacing GPT-3.5 with llama2-7B-chat and llama2-70B-chat [3]; further discussions are in Sec. VI.

Fuzzers. When conducting **RQ1**, we compare DFuzz with SoTA tools, TitanFuzz and FuzzGPT. They are both LLM-based fuzzers and have successfully tested a wide-range of APIs on PyTorch and TensorFlow. For bug discovery (**RQ2**), we compare DFuzz with TitanFuzz, FuzzGPT, and IvySyn. IvySyn is another SoTA fuzzing tool for DL frameworks. We have introduced the design of these SoTA tools in Sec. II-A. In line with these previous tools, we mainly use two metrics, (1) API coverage (**RQ1**) and (2) number of detected bugs (**RQ2**), to assess DFuzz and compare with previous tools.

A. RQ1: API Coverage

API coverage is an important metric for DL library fuzzing, given that a bug is triggered only when the related APIs are invoked [15, 17, 60, 69]. Evaluating API coverage reflects the quality of the initial programs generated by DFuzz. We first report API coverage below, then measure the debugging procedure of DFuzz when synthesizing these initial programs.

We compare DFuzz with FuzzGPT and TitanFuzz, both of which are LLM-based fuzzers. IvySyn focuses on testing low-level, native DL (C/C++) code. It synthesizes code snippets in high-level languages (e.g., Python) only for the buggy code identified in the low-level code. Consequently, IvySyn's approach isn't optimized for API coverage, and therefore, we do not compare it. As a fair comparison, we conduct experiments on PyTorch (v1.12) and TensorFlow (v2.10) (the same setup used in the FuzzGPT and TitanFuzz papers). More specifically, we conduct experiments directly within the Docker environment provided by TitanFuzz. We use the default settings of TitanFuzz and FuzzGPT.

TABLE III
The comparison of API coverage. PyTorch has a total of 1,593 APIs, while TensorFlow has 3,316 APIs.

Target Library	DFuzz		FuzzGPT		TitanFuzz		Under Test
	Cov[1]	Times[2]	Cov	Times	Cov	Times	
PyTorch(v1.12)	**1455**	**2839**	1377	Unknown	1329	39825	1593
TensorFlow(v2.10)	**2340**	**13938**	2309	Unknown	2215	82900	3316

[1] Cov: API coverage. [2] Times: Number of times LLM has been invoked.

We measure the number of covered APIs and the number of LLM invocation times for each tool, whose results are in Table III. DFuzz and TitanFuzz both comprise two stages: initial program generation and mutation. As the mutation phase does not lead to additional gains in API coverage, we collect API coverage and LLM usage times from the initial program generation stage for both these fuzzers. For FuzzGPT, we directly use its final coverage data. DFuzz offers the highest API coverage in both PyTorch and TensorFlow. In particular, DFuzz covers 78 and 126 more APIs on PyTorch compared to FuzzGPT and TitanFuzz, respectively. Meanwhile, DFuzz invokes significantly less amount of LLMs than that of TitanFuzz. FuzzGPT has not been open-sourced at the time of writing, and its LLM usage counts are unknown to us. On PyTorch and TensorFlow, the LLM usage of DFuzz is only 7.12% and 16.81% of TitanFuzz's, respectively. We believe the results are promising, given that DFuzz is able to cover more APIs with such fewer LLM invocations.

Recall our synthesis follows a trial-and-fix strategy where a synthesis process deems failed if DFuzz cannot successfully debug an initial program within a certain number of attempts (Alg. 1). Here, we report the number of successful and failed debugging attempts when generating the initial programs in Table IV. Note that successful debug refers to the scenario where the program first synthesized by the LLM is *invalid*, yet, after debugging, a valid program is obtained; "failed debug" has been noted above.

TABLE IV
Debug success evaluation.

Target Library	Successful debug	Failed debug	Success rate
PyTorch	198	138	58.92%
TensorFlow	902	976	48.02%

Overall, DFuzz's debugging success rates on PyTorch and TensorFlow are 58.92% and 48.02%, respectively. PyTorch has a total of 1,593 APIs, where for the 1,257 APIs, DFuzz successfully generates valid programs with just one query to the

LLM, and the remaining 336 APIs require debugging. Among these, 198 APIs yield corresponding invocation programs after debugging, while the remaining 138 APIs cannot be debugged successfully. We report that debugging each PyTorch API requires invoking the LLM approximately 4.71 times. With further analysis, we find that the gap between PyTorch and TensorFlow is due to the complexity of the APIs. In particular, there are more rarely-used APIs in TensorFlow, for which our employed LLM appears to lack related knowledge. This makes the overall generation and debugging process more challenging compared to PyTorch. Overall, we believe the results are promising, indicating that when integrating error information into prompts, we effectively guide the LLM to generate valid programs.

B. RQ2: Bug Discovery

We conducted bug discovery [15, 16, 48, 56, 56] on the latest versions of PyTorch (v2.2.1) and TensorFlow (v2.15) by the time of writing. Note that due to the lack of automated tools for bug analysis, confirming each bug requires significant human effort. Therefore, we only analyze the fuzzing results of DFUZZ. However, to compare with other fuzzers, we examine if the bugs detected by DFUZZ exist in the older PyTorch and TensorFlow versions tested by other fuzzers. If so, it implies that those fuzzers were unable to identify these bugs (otherwise they would have been fixed).[1]

We report the bug discovery findings in Table V. On TensorFlow, DFUZZ identified 27 bugs, with 4 already fixed and 15 replicated by the developer but still under investigation. Importantly, 16 bugs exist in TensorFlow (v2.10) which has been tested by TitanFuzz and FuzzGPT. Nevertheless, neither TitanFuzz nor FuzzGPT were able to detect them. Meanwhile, 16 bugs exist in TensorFlow (v2.6), which has been tested by IvySyn. On PyTorch, DFUZZ discovered ten bugs, with 4 already fixed and 4 replicated by the developer but still under investigation. Among these, eight bugs exist in PyTorch (v1.12) which was tested by TitanFuzz and FuzzGPT, while six bugs already exist in PyTorch (v1.11) tested by IvySyn. The above results demonstrate that DFUZZ not only efficiently discovers bugs but also identifies long-standing bugs.

Bug Characteristics. DFUZZ uncovers 37 bugs in total (the last row in Table V), which can be categorized into four types: abort signals, segfaults, runtime errors, and inconsistent output. Runtime errors include `INTERNAL ASSERT FAILED`, `MKL FFT error`, and `cuFFT error`. We calculate the number of bugs for each category, whose results are in Table VI. Most of the bugs discovered by DFUZZ are due to the lack of relevant validation checks in APIs. Since we test various possible edge cases over input variables, most APIs tend to trigger abort signals or segfaults directly when encountering unexpected inputs, resulting in fewer cases of inconsistent outputs. Among all found bugs, 28 cause crashes. As noted in DocTer [64], "Despite receiving invalid inputs, DL API functions should not crash. Instead, they are expected to handle such inputs gracefully (e.g., through throwing an exception)." Thus, aligned with related works' focus (e.g., DocTer and IvySyn), we deem these crash bugs as critical.

Transferability. While our edge cases were extracted from PyTorch, we successfully discover 27 bugs in the latest version of TensorFlow. This indicates that the edge cases we extracted have good transferability and can be used across platforms. In fact, one could even interpret that the edge cases extracted from PyTorch are "more effective" in uncovering bugs in TensorFlow than that of PyTorch itself. We believe this is due to the fact that PyTorch has been tested by many fuzzing tools and the community is actively fixing bugs (still, we find ten bugs). Moreover, by transferring edge cases across platforms, DFUZZ for the first time enables a highly comprehensive fuzzing of TensorFlow, which is hardly achieved by existing tools (most of those 27 bugs are *not* found by previous works).

Extracted Edge Case Types. The DFUZZ is capable of extracting a wide variety of edge cases. Except the three types in Table I, there are some other types: (1) edge cases related to multiple parameters, such as "`tensor_1` has a larger last dimension than `tensor_2`"; (2) some special parameter attributes related to program logic, such as "sparse", "dense", "conjugate", and "contiguous" for tensor. (3) restriction between the attributes of a parameter, such as "`tensor_1` with input.shape[-2] $<$ input.shape[-1]". LLMs employed by DFUZZ can handle these cases properly.

C. RQ3: Alternative LLMs

In this RQ, we evaluate if DFUZZ can effectively employ smaller LLMs for edge-case based mutation. In our experiments, the general LLMs have demonstrated greater proficiency in understanding our tasks, particularly in creating edge cases for a target API. Thus, we conduct experiments using two widely-used general open-source LLMs, llama2-70b-chat and llama2-7b-chat [3]. Recall we have obtained a set of bug-triggering programs in **RQ2**; we collect the prompts which are used to generate these programs, and then feed those prompts into llama2-70b-chat and llama2-7b-chat. With the same prompts, we check if they can also generate bug-triggering programs. We do not have powerful servers to run llama2-70b-chat, and therefore, we use the APIs provided by Replicate [5]. Replicate provides APIs capable of running open-source models. We set the temperature to 0.5 for each prompt and repeat the generation process three times for each prompt.

The results are in Table VII. Using llama2-70b-chat, we can still discover three bugs on PyTorch and ten bugs on TensorFlow, respectively. llama2-7b-chat results in discovering two bugs on PyTorch and four bugs on TensorFlow. Interestingly, llama2-7b-chat discovers two bugs on TensorFlow that are not found by llama2-70b-chat. Upon further analysis, we find that llama2-7b-chat exhibits more uncertainty in generating complex tensors. This increased uncertainty occasionally leads to correctly generating bug-triggering programs.

D. RQ4: Ablation Study

In DFUZZ, each test case is formed by conducting two tasks: initial program generation and edge case-based mutation. To

[1]At this step, we also search for bug information in the community (e.g., PyTorch Forums) to make sure they have not been reported by others before.

TABLE V
THE BUGS FOUND BY DFUZZ.

Target Library	Total	Fixed	Replicated	Existed in PyTorch (v1.12) / TF (v2.10) (Not found by TitanFuzz/FuzzGPT)	Existed in PyTorch (v1.11) / TF (v2.6) (Not found by IvySyn)
PyTorch	10	4	4	8	6
TensorFlow	27	4	15	16	16
Total	37	8	19	24	22

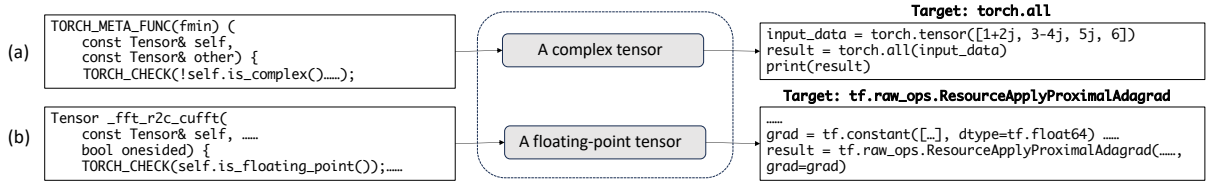

Fig. 8. The case study.

TABLE VI
BUG TYPES.

Library	abort signals	segfaults	runtime error	inconsistent
PyTorch	0	2	5	3
TensorFlow	24	2	0	1
Total	24	4	5	4

TABLE VII
EVALUATING OTHER LLMS.

Target Library	ChatGPT-3.5	Llama-2-70b-chat	Llama-2-7b-chat
PyTorch	10	3	2
TensorFlow	27	10	4
Total	37	13	6

analyze the contribution of each component to bug discovery, we analyze the bugs discovered by each component. We find that *all* bugs are discovered through the edge case-based mutations. This is unsurprising, because the API list we select is aligned with both TitanFuzz and FuzzGPT. TitanFuzz and FuzzGPT have already tested these APIs and discovered some rather easy-to-trigger bugs. Given DFUZZ's initial program generation is mainly to offer a program invoking the target APIs and thus expanding API coverage, it is expected to not discover any new bugs.

On the other hand, the results show the effectiveness of our edge case-based mutation. Although these APIs have already been tested by other tools, our edge case-based mutation approach still detected ten and 27 bugs respectively in the latest versions of PyTorch and TensorFlow (where most of them also exist in older versions tested by previous fuzzing tools). This illustrates that our tool effectively benefits from diverse edge cases, consequently leading to the discovery of more bugs.

E. Case Study

We present case studies to better illustrate the insights of DFUZZ. Fig. 8 presents two examples to show how DFUZZ extracts context-free edge cases from the source code and successfully triggers bugs. In Fig. 8(a), within `TORCH_META_FUNC`, `TORCH_CHECK` checks if the variable with the tensor type deems a complex tensor. Accordingly, we extract an edge case of tensor type: `complex tensor`. Since the input parameter of API `torch.all` is also of tensor type, we use the extracted edge case to fuzz it, by setting its input parameter as a complex tensor. This way, we discover an inconsistent output bug in PyTorch (v1.12) running on GPU and CPU, due to that `torch.all` does not consider the case of complex tensor. Given that `torch.all` is widely used, our finding received immediate attention from developers and was set as *high priority* for fixing.

In Fig. 8(b), we extract edge cases from PyTorch and apply them in fuzzing TensorFlow; this shows the transferability. Within `_fft_r2c_cufft`, `TORCH_CHECK` checks if the variable with tensor type deems a floating-point tensor. Similarly, we extract an edge case of tensor type: `floating-point tensor`, and use the extracted edge case to fuzz `tf.raw_ops.ResourceApplyProximalAdagrad`. We find that this API has a severe core dumped issue in handling floating tensors.

It is difficult for previous fuzzing tools to detect these two bugs. TitanFuzz applies basic mutation operators to mutate programs based on a scheduling algorithm. However, it lacks guidance for triggering edge cases, making it infeasible to trigger those two bugs. Similarly, for FuzzGPT to discover these bugs, three prerequisites must be met: first, there must be bug codes with similar root causes on the Internet. Second, FuzzGPT needs to properly crawl and collect these related bug codes. Third, FuzzGPT must correctly extract the root cause from these bug codes and apply it to test `tf.raw_ops.ResourceApplyProximalAdagrad`. However, these two bugs already existed in PyTorch (v1.12) and TensorFlow (v2.10), which have been tested by FuzzGPT; this indicates that FuzzGPT overlooked these bugs. Overall, the aforementioned three conditions are uneasy to meet in practice, making it difficult for FuzzGPT to discover these bugs. In contrast, DFUZZ manifests two advantages over FuzzGPT: (1) DFUZZ does not depend on information (bug codes) from the Internet; instead, it *only* needs the source code of DL libraries. (2) It can comprehensively test APIs, uncovering more edge cases that developers may not have considered nor reported online.

IvySyn manually designs mutators through the analysis of known 240 CVEs. This is a substantial effort, yet hard to be comprehensive. Particularly, for tensor types, IvySyn features a pool of tensor mutations containing tensors with large positive and negative values, tensors with empty shapes, and tensors containing random dimensions. Nevertheless, it does not include cases of complex tensors and floating-point tensors. As a result,

IvySyn fails to find these two bugs, although these bugs already exist in PyTorch (v1.11) and TensorFlow (v2.6) that have been tested by IvySyn. This illustrates the inherent challenge faced by manual efforts: it is difficult to be comprehensive and also keep up with the rapidly evolving DL libraries. In contrast, DFUZZ automatically extracts edge cases from the source code, not only eliminating the need for expert experience but also capturing a more comprehensive set of edge cases.

VI. DISCUSSION

Extension. Holistically, DFUZZ leverages the reasoning and generation abilities of LLMs to achieve comprehensive fuzzing of DL libraries. In this regard, we envision several promising directions to extend DFUZZ. First, we can further improve the reasoning ability of LLMs by incorporating more domain-specific knowledge. For example, we can leverage the existing knowledge in the form of API documentation, code comments, and bug reports to guide the reasoning process. As a common practice, this rich information can be dumped into local vector databases, and then be used to guide the reasoning process [37, 64]. Second, we can enhance the generation ability of LLMs by incorporating more advanced program synthesis techniques. Recent advances in LLM-based program synthesis have shown promising results in generating programs under various scenarios [10, 31, 38, 41, 53, 61, 72]. Nevertheless, we clarify that the benefit of enhancing generation ability may be marginal, given that when preparing the initial test programs, we often do not need to synthesize complex programs.

Threat to Validity. One potential threat to the validity of our study is the generalization of our findings. While we have demonstrated the effectiveness of DFUZZ on two popular DL libraries, TensorFlow and PyTorch, it is unclear whether DFUZZ can be generalized to other DL libraries. To mitigate this threat, we illustrate the high transferability of DFUZZ by showing that its extracted edge cases from PyTorch can be effectively used to test TensorFlow. This suggests that DFUZZ can be generalized to other DL libraries.

Another potential threat is the reproducibility of our findings. LLMs can be sensitive to the training data, the model architecture, and the hyperparameters. Particularly, commercial LLMs like GPT-3.5 only offer remote APIs which might undermine its reproducibility. To mitigate this threat, we have made our artifact available [2], and we have assessed the performance using alternative open-source LLMs. From another perspective, we observe that a certain level of randomness is helpful to fuzz (e.g., our findings in Sec. V-C).

VII. CONCLUSION

In this paper, we present DFUZZ, a novel white-box approach for LLM-based DL library API fuzzing. Using LLMs, we infer edge cases and generate initial test programs, which offer effective and efficient DL library API fuzzing. Evaluations show that DFUZZ consistently outperforms existing DL library fuzzers for PyTorch and TensorFlow.

ACKNOWLEDGMENTS

The HKUST authors are supported in part by a RGC GRF grant under the contract 16214723 and a RGC CRF grant under the contract C6015-23G.

REFERENCES

[1] Chatgpt. https://openai.com/chatgpt, 2024.

[2] Dfuzz. https://github.com/DFUZZ-ICSE/DFUZZ, 2024.

[3] llama. https://github.com/meta-llama/llama, 2024.

[4] Pytorch. http://pytorch.org, 2024.

[5] Replicate. https://replicate.com, 2024.

[6] Tensorflow. https://www.tensorflow.org, 2024.

[7] Sajad Saraygord Afshari, Chuan Zhao, Xinchen Zhuang, and Xihui Liang. Deep learning-based methods in structural reliability analysis: a review. *Measurement Science and Technology*, 2023.

[8] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. On hardware security bug code fixes by prompting large language models. *IEEE Transactions on Information Forensics and Security*, 2024.

[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[10] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[11] Mihalj Bakator and Dragica Radosav. Deep learning and medical diagnosis: A review of literature. *Multimodal Technologies and Interaction*, 2(3):47, 2018.

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

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

[14] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. {IvySyn}: Automated vulnerability discovery in deep learning frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2383–2400, 2023.

[15] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[17] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 44–56, 2022.

[18] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. Differential testing of cross deep learning framework {APIs}: Revealing inconsistencies and vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7393–7410, 2023.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. Large language models of code fail at completing code with potential bugs. *Advances in Neural Information Processing Systems*, 36, 2024.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[23] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1051–1065, 2022.

[24] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[25] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[26] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of field robotics*, 37(3):362–386, 2020.

[27] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1418–1430, 2022.

[28] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pages 12098–12107. PMLR, 2023.

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

[30] Sundas Iftikhar, Zuping Zhang, Muhammad Asim, Ammar Muthanna, Andrey Koucheryavy, and Ahmed A Abd El-Latif. Deep learning-based pedestrian detection in autonomous vehicles: Substantial issues and challenges. *Electronics*, 11(21):3551, 2022.

[31] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[32] Yi Li, Shaohua Wang, and Tien Nguyen. A context-based automated approach for method name consistency checking and suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 574–586. IEEE, 2021.

[33] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021.

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. Utango: untangling commits with context-aware, graph-based, code change clustering learning model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 221–232, 2022.

[35] Yi Li, Shaohua Wang, and Tien N Nguyen. Contextuality of code representation learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 548–559. IEEE, 2023.

[36] Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N Nguyen. Commit-level, neural vulnerability detection and assessment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1024–1036, 2023.

[37] Yitong Li. Documentation-guided fuzzing for testing deep learning api functions. Master's thesis, University of Waterloo, 2020.

[38] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. *arXiv preprint arXiv:2403.03997*, 2024.

[39] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, 2020.

[40] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 530–543, 2023.

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

[42] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2020.

[43] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. Ems: History-driven mutation for coverage-based fuzzing. In *NDSS*, 2022.

[44] Khan Muhammad, Amin Ullah, Jaime Lloret, Javier Del Ser, and Victor Hugo C de Albuquerque. Deep learning for safe autonomous driving: Current challenges and future directions. *IEEE Transactions on Intelligent Transportation Systems*, 22(7):4316–4336, 2020.

[45] Son Nguyen, Tien Nguyen, Yi Li, and Shaohua Wang. Combining program analysis and statistical language model for code statement completion. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 710–721. IEEE, 2019.

[46] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.

[47] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[48] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.

[49] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[50] Jonathan Pilault, Raymond Li, Sandeep Subramanian, and Christopher Pal. On extractive and abstractive neural document summarization with transformer language models. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pages 9308–9319, 2020.

[51] Mohammad Javad Shafiee, Ahmadreza Jeddi, Amir Nazemi, Paul Fieguth, and Alexander Wong. Deep neural network perception models and robust autonomous driving systems: practical solutions for mitigation and improvement. *IEEE Signal Processing Magazine*, 38(1):22–30, 2020.

[52] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[53] Dominik Sobania, Martin Briesch, and Franz Rothlauf. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pages 1019–1027, 2022.

[54] Liyan Tang, Zhaoyi Sun, Betina Idnay, Jordan G Nestor, Ali Soroush, Pierre A Elias, Ziyang Xu, Ying Ding, Greg Durrett, Justin F Rousseau, et al. Evaluating large language models on medical evidence summarization. *npj Digital Medicine*, 6(1):158, 2023.

[55] Chengpeng Wang, Gang Fan, Peisen Yao, Fuxiong Pan, and Charles Zhang. Verifying data constraint equivalence in fintech systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1329–1341. IEEE, 2023.

[56] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. Eagle: Creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering*, pages 798–810, 2022.

[57] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. Deepvd: Toward class-separation features for neural network vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2249–2261. IEEE, 2023.

[58] Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. Natural is the best: Model-agnostic code simplification for pre-trained large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE):586–608, 2024.

[59] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 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*, pages 788–799, 2020.

[60] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, pages 995–1007, 2022.

[61] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.

[62] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–357. IEEE, 2023.

[63] Michel Wermelinger. Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 172–178, 2023.

[64] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188, 2022.

[65] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[66] Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shanping Li. Selfpico: Self-guided partial code execution with llms. *arXiv preprint arXiv:2407.16974*, 2024.

[67] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N Nguyen. A learning-based approach to static program slicing. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):83–109, 2024.

[68] Aashish Yadavally, Tien N Nguyen, Wenbo Wang, and Shaohua Wang. (partial) program dependence learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2501–2513. IEEE, 2023.

[69] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1174–1186. IEEE, 2023.

[70] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. *arXiv preprint arXiv:2407.20898*, 2024.

[71] Michal Zalewski. American fuzzy lop, 2017.

[72] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

[73] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics*, 12:39–57, 2024.

[74] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.