Evaluating Deep Learning Fuzzers: A Comprehensive Benchmarking Study

Anonymous Author(s)

***Abstract*—In recent years, the practice of fuzzing Deep Learning (DL) libraries has gathered significant attention in the software engineering community. Many DL fuzzers have been proposed to test DL APIs via the generation of malformed inputs. Although most of these fuzzers have been demonstrated to be effective in detecting bugs and outperforming their respective prior work, there remains a gap in benchmarking these DL fuzzers regarding their effectiveness against ground-truth real-world bugs in DL libraries. Since the existing comparisons among these DL fuzzers mainly focus on comparing bugs detected by them, they cannot provide a direct, in-depth evaluation of different DL fuzzers.**

In this work, we set out to conduct the first evaluation of state-of-the-art DL fuzzers against real-world bugs. Specifically, we first manually created an extensive DL bug benchmark dataset, which includes 412 real-world DL bugs gathered from PyTorch and TensorFlow libraries that are detectable by fuzzing, i.e., can be triggered by malformed inputs. Then we apply six state-of-the-art DL fuzzers, i.e., FreeFuzz, DeepRel, NablaFuzz, DocTer, TitanFuzz, and AtlasFuzz, on the curated benchmark dataset by following their instructions. We find that these fuzzers can only detect 3.1% (13 out of 412) unique real-world bugs in our benchmark dataset. Our analysis further pinpoints three prevailing factors that impact the effectiveness of these fuzzers in identifying real-world bugs. These findings present opportunities to improve the performance of the fuzzers in future work. Overall, this work complements prior studies on DL fuzzers with an extensive evaluation and provides a benchmark for future DL library fuzzing studies.

1. Introduction

In recent years, fuzz testing [1]–[8], which detects bugs via generating malformed inputs, has been adopted to test DL libraries [9]–[11]. For example, FreeFuzz [9] is the first API level DL fuzzer, proposed to generate random inputs with various mutation strategies. Later, DocTer [10] proposed which first extracts DL-specific input constraints from the documentation of DL APIs and then the extracted constraints are used to generate valid and invalid inputs to test DL APIs. Along this line, DeepRel [11] extends FreeFuzz by using test inputs from one API to test other related APIs that share similar input parameters, with the assumption that APIs with similar input parameters can share input specifications. TitanFuzz [12] is the first study that leveraged the power of Large Language Models (LLMs) [13]–[15] to generate inputs tailored for testing DL APIs. AtlasFuzz [16] extended TitanFuzz by including unusual programs mined from open source by LLMs to guide the test input generation. NablaFuzz [17] is the first fuzzing tool that uses differential testing to expose bugs in DL libraries.

Although these tools have been demonstrated to be effective in detecting bugs in DL libraries and outperforming their prior work in finding more or different bugs, there remains a gap between the evaluated result and the tools' practical effectiveness. Specifically, prior studies failed to completely evaluate the tools' effectiveness, bug detection scope, and limitations against ground-truth DL library bugs. This leaves an unanswered question: *How effectively and thoroughly can these DL fuzzers find bugs in practice?* To our knowledge, there is no study in the literature to evaluate DL fuzzers for detecting real-world bugs from an extensive ground-truth bug dataset. In this paper, we conduct the first extensive empirical study to benchmark six state-of-the-art DL fuzzers, including DocTer [10], FreeFuzz [9], DeepRel [11], TitanFuzz [12], AtlasFuzz [16], and NablaFuzz [17] against real-world bugs from two most popular and open-source DL libraries (TensorFlow and PyTorch). Specifically, we first automatically extracted GitHub bugs using a keyword-matching approach [18] from TensorFlow and PyTorch repositories reported between November 2022 to October 2023. Since the automatic extraction may introduce false positives, we further conducted a manual analysis process to verify each bug collected and filter out non-applicable bugs, i.e., bugs that can not be triggered by malformed inputs. For each bugs, we further extract its trigger APIs, which will be the target of fuzz testing tools. As a result, 412 real-world DL bugs in 255 unique APIs from PyTorch and TensorFlow were collected. Then, we rigorously apply the selected DL fuzzers by following their instructions on the trigger APIs of the collected real-world bugs to evaluate their effectiveness in detecting ground-truth bugs in our benchmark dataset.

Our analysis profiles the cost, effectiveness, and efficiency of each DL fuzzers (Section 4) from the following three aspects: the total CPU hours it takes to run (i.e., cost), the number of real-world bugs that it detects (i.e., effectiveness), and test cases it generates to detect bugs on average (i.e., efficiency). Finally, we present a detailed quantitative and qualitative analysis of bugs detected by each DL fuzzer and we further check these fuzzers' implementations to identify the challenges they face and the factors that affect their bug detection abilities. In this work, we investigate the following research questions:

RQ1: *What are the characteristics of extracted real-world DL bugs?*

TABLE 1: Characteristics of DL fuzzers used in this study.

Fuzzer	Mutation Strategy						Test Oracles					
	Boundary-input	Shape mutation	Value/Type mutation	Prefix	Suffix	Method	Crash	CPU/GPU	Performance	Wrong Computation	AD	ND
FreeFuzz	Random	✓	✓	×	×	×	✓	✓	✓	×	×	×
DeepRel	Random	✓	✓	×	×	×	✓	×	×	✓	×	×
NablaFuzz	Random	✓	✓	×	×	×	✓	✓	×	×	✓	✓
DocTer	Rule-based	✓	✓	×	×	×	✓	×	×	×	×	×
TitanFuzz	Random	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×
AtlasFuzz	History-Based	✓	✓	×	×	×	✓	✓	×	✓	✓	✓

RQ2: *How effectively and thoroughly can SOTA DL fuzzers detect real-world DL bugs?*

RQ3: *What are the characteristics of detected DL bugs?*

RQ4: *What are the reasons for fuzzers to miss a bug?*

Overall, we find that these fuzzers failed to detect a large number of real-world bugs in our benchmark dataset, i.e., they only detected 13/412 (3.1%) of bugs. We observe that *Runtime Error* and *Check Failed* are the most frequent bug types in PyTorch and TensorFlow libraries. Additionally, we find that LLM-based DL fuzzers demonstrate superior bug detection effectiveness and generate more valid test cases. Finally, we discuss implications that can improve DL fuzzers by analyzing the root cause of bugs that cannot be detected by these fuzzers. Overall, this work complements prior studies on DL fuzzers with an extensive evaluation and provides a benchmark for future DL fuzzing studies. This paper makes the following contributions:

- We conduct the first empirical study to evaluate state-of-the-art DL fuzzers against real-world DL bugs, which provides a complementary perspective on prior studies regarding the comparison among DL fuzzers.
- We create the first benchmarking dataset for evaluating DL fuzzers rigorously, which includes 412 reproducible bugs from TensorFlow and PyTorch that are detectable by fuzzing-based approaches.
- We conduct an in-depth quantitative and qualitative evaluation of DL fuzzers and present findings regarding the cost, effectiveness, and efficiency of these DL fuzzers.
- We release the dataset and source code of our experiments to help other researchers replicate and extend our study[1].

The rest of this paper is organized as follows. Section 2 describes the methodology of our approach. Section 3 and Section 4 show the experimental setup and the evaluation results respectively. Section 5 discusses the threats to the validity of this work. Section 6 presents the related studies. Section 7 presents the threats to the validity of this work. Section 8 concludes this paper.

2. Approach

In this section, we present our approach to constructing a benchmark for evaluating the effectiveness of DL fuzzers. We first explain the process of selecting DL fuzzers for our benchmark analysis (Section 2.1). We then discuss the selection of subject DL libraries (Section 2.2). Finally, We explain how we gathered real-world bugs using a combination of automatic filtering and manual analysis (Section 2.3).

1. https://anonymous.4open.science/r/securityandprivacy2024-6B57/README.md

2.1. Selection of DL fuzzers

The focus of our research is on the examination of recent SOTA DL fuzzers, i.e., FreeFuzz [9], DeepRel [11], NablaFuzz [17], DocTer [10], TitanFuzz [12], and AtlasFuzz [16], which are currently being developed and freely available to the public. Even though there are multiple fuzzing tools proposed both in the academia [19]–[23] and the industry [4], [5], [24], in this paper, we give particular attention to fuzzers that concentrate on DL libraries Python client APIs. There are a couple of reasons behind this decision. First, DL libraries are commonly used via their Python APIs from the front end, while the actual computations are performed in the DL backend (which is mainly programmed in C/C++). While some general fuzzers use C/C++ APIs for fuzzing, testing DL libraries in their typical usage (i.e., via Python APIs) is the most effective way to detect bugs that are important to users. Second, DL libraries differ significantly from traditional software systems [25], [26] primarily due to their use of tensors–a unique data structure in which all computations occur at the tensor level. We chose to focus on DL fuzzers because they can simulate the tensor data structure for fuzzing, whereas general fuzzers are only capable of modeling preemptive data structures. Please note that we ignored DL compiler testing tools [27]–[30] since they focus on completely different subject programs, i.e., they test DL compilers such as TVM [31], TensorRT [32], and ONNXRuntime [33]. In contrast, our emphasis is on DL libraries, such as TensorFlow and PyTorch. This distinction in focus underscores our concentration on testing the foundational APIs used in DL application development, as opposed to the compilers that facilitate the execution of DL models on diverse hardware platforms.

Table 1 outlines the key characteristics of the three fuzzers analyzed in this paper. We performed a manual analysis of the implementation and documentation for each fuzzing tool to assess its capabilities and limitations. We characterize DL fuzzers mainly based on their mutation strategies and types of test oracles. This comprehensive analysis allows us to compare the effectiveness of each fuzzer.

FreeFuzz [9] is an API-level fuzz testing tool that performs test case generation using random value and type mutation. It also supports boundary input values to generate test cases that find edge cases in DL libraries. However, the

boundary input generation is random, unlike DocTer [10], where rules are used to guide the boundary input generation. FreeFuzz uses three types of oracles: CRASH, CPU/GPU, and Performance.

DeepRel [11] extends FreeFuzz by using test inputs from one API to test other related APIs that share similar input parameters. It hypothesizes that APIs with similar input parameters can share input specifications. Like FreeFuzz, it also supports type and value mutation of preemptive data structures as well as tensor data structures. Its major difference when compared to FreeFuzz and DocTer is the oracle where DeepRel compares two semantically related APIs in terms of equivalence value and status.

NablaFuzz [17] Is an API-level fuzzer that tests DL APIs with and without Automatic Differentiation (AD). The hypothesis is that direct execution of DL APIs is not sufficient for bug detection, rather DL APIs should be tested with a critical component of deep learning which is automatic differentiation [34]. NablaFuzz uses two AD modes for gradient calculation including reverse mode AD calculation and forward mode AD calculation. NablaFuzz also uses Numerical Differentiation (ND) In terms of fuzzing, NablaFuzz employs the previous DL fuzzer, FreeFuzz [9] for test input collection and fuzzing operation (via applying the same mutation strategies).

DocTer [10] is a fuzzing tool that uses two major techniques for testing DL APIs: *DL-specific constraint extraction* and *DL-specific input generation*. Initially, DocTer extracted input constraints specific to DL libraries from API documentation. It generates rules from syntactic patterns found in API descriptions and applies them to a large number of API documents in popular DL libraries to extract their input parameter constraints. The constraints are then used to guide its fuzzer to generate conform, violate, and boundary inputs to DL APIs. Once the test cases are generated, DocTer executes them and performs stack trace analysis to look for crash bugs such as *Segmentation Fault*, *Bus Errors*, *Aborts*, and *Floating point exceptions*.

TitanFuzz [12]:is one of the most recent DL fuzzer, that harnesses the power of LLMs [35], [36] for conducting API-level fuzzing on TensorFlow and PyTorch libraries. As shown in Table 1, TitanFuzz goes beyond supporting shape and value/type mutation; it extends its capabilities to include mutation in additional code regions. Specifically, it performs mutations in the line preceding API call sites (prefixes), the line following the API call site (suffixes), and even the mutation of method names (Method), i.e., based on the contact of the code. The inclusion of these mutation operators is crucial, given the increasing complexity of bug patterns in deep learning libraries. To effectively generate test cases and perform fuzzing, it is essential to model and explore various code regions, considering the evolving nature of bugs in DL libraries.

AtlasFuzz [16]: is the extension of TitanFuzz where it uses historical bug data collected from the open source to guide their fuzzer generator with LLMs. AtlasFuzz initially extracts bug reports from DL library repositories to gather historical code snippets triggering bugs. AtlasFuzz then focuses on specific DL library APIs, requiring corresponding buggy API labels for each code snippet. However, bug reports may not explicitly mention the buggy API. Hence, AtlasFuzz uses a self-training approach, LLMs to automatically generate buggy API labels based on a few manually labeled examples. With pairs of bug-triggering code snippets and buggy API labels, AtlasFuzz initiates the fuzzing process to generate edge-case code snippets.

2.2. DL libraries selection

In this study, we chose TensorFlow [37] and PyTorch [34] as our subjects, which have been widely used in the literature [38]–[41], for two main reasons. First, there are numerous usages of TensorFlow and PyTorch in various application domains, including image classification [42], [43], big data analysis [44], pattern recognition [45], self-driving [25], [46], [47] and natural language processing [48]–[50]. Second, the six benchmark fuzzers studied have used TensorFlow and PyTorch as their target DL libraries for fuzzing.

2.3. Automated collection of real-world DL bugs

This study aims to evaluate the effectiveness of six state-of-the-art DL fuzzers in practice. To this end, it is essential to collect a set of real-world bugs. In this paper, we use the term *bug* in a general sense to refer to any kind of software defect, including security vulnerabilities, logical bugs, and performance bugs. We start from collecting bugs from the GitHub repository of TensorFlow and PyTorch[2]. We adopt the following two steps for automatic data collection: 1) we filter GitHub issues with the labels *bug* and *bug-fix* and 2) for each bug collected in step 1), we automatically analyze the title, body, and comments and search for bug-related keywords following existing empirical studies [18], [26], [51]–[53]. Specifically, the following shows an excerpt of keywords we use for data collection[3]:

Security vulnerabilities: *buffer overflow*, *integer overflow*, *cross-site scripting*, *remote code execution*, *memory leak*, *race condition*, *heap buffer overflow*, *null pointer dereference*. **Logical errors**: *wrong result*, *unexpected output*, *incorrect calculation*, *inconsistent behavior*, *unexpected behavior*, *incorrect logic*, *wrong calculation*, *logic error*. **Performance errors**: *slow*, *high CPU usage*, *high memory usage*, *poor performance*, *slow response time*, *performance bottleneck*, *performance optimization*, *resource usage*.

The reason behind using a keyword-based approach for filtering irrelevant issues is threefold. Firstly, it's an alignment with prior studies as many existing empirical studies adopted a keyword-based approach for data collection [18], [26], [51]–[53]. Second, keywords are specific terms directly

2. The GitHub repository can be considered a reliable source of bugs since the users often report bugs as issues

3. Please note that due to brevity, we do not include all security keywords in the manuscript. We list all keywords in the GitHub repository of this paper.

related to bug-related issues, enhancing the precision of issue retrieval [54]. Fortunately, PyTorch and TensorFlow have very strict policies in opening issues in their GitHub repository. This increases the chance of finding more relevant issues using the keyword-based approach. Ultimately, Keywords are selected based on common bug patterns or issues that developers typically report. This approach helps in capturing relevant and recurring bugs.

At the end of this automated process, 2,678 issues from PyTorch and 476 issues from TensorFlow were collected for our manual validation (shown in Table 2). The reason behind this unbalanced number of collected data is that PyTorch has a higher number of reported issues compared to TensorFlow. Please note that CVE records were not included in our benchmark dataset. Concerning PyTorch, developers typically do not report bugs in the CVE repository[4]. In the case of TensorFlow, although bug reports exist, we chose not to incorporate them for two main reasons. Firstly, CVE reports are not up-to-date compared to issues reported in the GitHub repository of TensorFlow. Notably, the latest entry in the advisory dates back to March 2023, while the most recent bug-related issue in the TensorFlow GitHub repository is from October 19, 2023. This disparity is crucial, particularly given the evolving complexity of bug patterns in the TensorFlow library [55]. Ensuring a more current dataset is essential for the robust testing of DL libraries. Secondly, the GitHub repository of TensorFlow serves as the most suitable and accessible channel for reporting bugs within the TensorFlow community. It is worth noting that, frequently, CVE reports originate from issue reports within TensorFlow.

While the automatic mining of GitHub repositories proved efficient in gathering a substantial number of issues, it also led to the inclusion of numerous unrelated or non-bug-related issues. To ensure the integrity of our manual analysis and focus solely on potential bugs, we chose to manually analyze the collected issues gathered from TensorFlow and PyTorch repositories. This thorough manual analysis helped us identify and validate actual bugs effectively. Please note that we exhaustively compared all reported issues in the replication packages of DL fuzzers with our collected benchmark data to check whether they match or not to remove any potential bias from our experiments and results.

2.4. Manual validation to create our benchmark dataset

2.4.1. Manual Analysis Criteria. Once we automatically collected the related bugs from the GitHub repository of TensorFlow and PyTorch, we performed manual analysis to filter false-positive instances. The following are the different scopes of bugs that we used to guide our manual analysis:
Security Vulnerability: A security vulnerability or weakness is a flaw in DL libraries that external attackers can exploit to take control of the running DL program [26], [51], [56]. **Logical Bug**: A logical bug [57] in a DL library refers to an error in the code that is not related to syntax or other obvious mistakes but rather to a flaw in the reasoning or logic behind the algorithm. **Performance Bug**: A performance bug [58] in a DL library refers to a bug that degrades the efficiency or speed of the DL model's training or inference process. Performance bugs can occur for a variety of reasons, such as inefficient algorithm design, suboptimal hardware utilization, or poor memory management.

For each issue, we examined its title, body, comments, linked PRs, and commits to determine whether the bug is detectable by fuzzing-based approaches, i.e., can be triggered by malformed inputs. We exclude the following types of unrelated issues:

- Bugs that are specific to certain platforms, such as Windows, Apple M chips, Android, or IOS[5].
- Bugs require devices other than GPU and CPU[6].
- Bugs that are related to building and library updates.
- Bugs in the backend, no high-level APIs are involved for triggering these bugs.
- Bugs in external libraries such as torchvision, torchaudio, transformers, etc.[7].

For each collected bug, we retrieved the bug type and release information for the involved APIs (the release in which the bugs can be replicated). We also collect the affected and patched releases. In our experiments, we randomly chose one affected release that had not yet been patched. Users may report numerous releases affected by the bug, but we opt to consider only one affected release per bug when running the fuzzers.

2.4.2. Manual Analysis Procedure. Starting with the 2,678 and 476 automatically extracted bugs for PyTorch and TensorFlow repositories, respectively, the authors conducted a manual analysis with two rounds:
Round 1: Three authors independently reviewed PyTorch and TensorFlow and collected GitHub bugs. The authors extracted multiple pieces of information from each bug, i.e., the buggy API, affected release, CUDA version (if the issue is tested under CUDA), the impact of the bug(log message or stack traces), whether the bug relates to tensor calculation, how the bug can be triggered, and a description if it is necessary. Once the information is extracted, the authors cross-check the labeled bugs to mark possible disagreements (e.g., lack of consensus on the bug type). In this step, the disagreement rates were 2.3% and 17.4% for PyTorch and TensorFlow, respectively.
Round 2: In this round, all authors were involved in the manual analysis of the records in PyTorch and TensorFlow

4. https://github.com/pytorch/pytorch/security

5. Please note that we only run the studied DL fuzzers on Linux operating system.

6. Kindly be informed that our machine is limited to CPU and CUDA capabilities only. Consequently, we filtered out bugs that necessitate different hardware platforms for replication, focusing on issues compatible with our available hardware configurations.

7. If the issue is merely from the external libraries, we filter them. However, there are some bugs due to the usage of DL APIs with external libraries, we also include them in our benchmark dataset.

TABLE 2: Characteristics of our benchmark dataset.

Library	All Issues	Automatic Filtering	# Real-World Bugs	# Unique APIs
PyTorch	$5k^+$	2,678	303	155
TensorFlow	$1.9k^+$	476	109	100
Total	-	3,154	412	255

TABLE 3: Statistics on the number of overlapping APIs for each DL fuzzer on PyTorch and TensorFlow.

Tools	PyTorch		TensorFlow	
	# Covered API	# Analyzed APIs	# Analyzed API	# Analyzed APIs
FreeFuzz	1,071	65	1,902	44
DeepRel	1,071	65	1,902	44
NablaFuzz	1,071	69	1,902	44
DocTer	498	42	911	35
TitanFuzz	1,329	95	2,215	70
AtlasFuzz	1,377	95	2,309	70

from Round 1, and disagreements were resolved with group discussions. The rate of disagreements at the end of this round was 0.6% and 1% for PyTorch and TensorFlow, respectively. At the end of the round, we discarded any bug on which the authors could not reach a consensus. Finally, 303 and 109 bugs for PyTorch and TensorFlow are left in the dataset.

3. Experiment Setup

3.1. Experiment Data

Table 2 shows the number of collected bugs and unique APIs in our benchmark. The total number of bugs (# Bugs) we could get after our manual analysis is 303 and 109 for PyTorch and TensorFlow. Since the internal database of the fuzzers may not cover all APIs we collected, We consider overlapping APIs (shown in Table 3), i.e., if its buggy DL API exists in the internal database of the target fuzzer.

3.2. Configuration for running DL fuzzers

In this section, we describe the steps we took to set up the DL fuzzers used in our benchmark analysis. It should be noted that we followed the provided instructions in the replication documentation of each DL fuzzer.

FreeFuzz [9]: To run FreeFuzz, we enabled crash, cuda, and performance oracles. We also set the float difference cut-off value as its default value of $1e-2$. We also set *max_time_bound* and *time_thresold* to the default values of 10 and $1e{-}3$ respectively. Regarding the mutation strategies, we enabled value mutation, type mutation, and database mutation. We also set the number of times FreeFuzz performs testing for each API to 1000.

DeepRel [11]: DeepRel is built upon FreeFuzz, and compared to FreeFuzz, there are three more hyperparameters to be set. We set the test number (i.e., the number of times each API is executed) to 1000, *top_k* parameter to 3, and the number of iterations to 1 (as instructed). Note that for both FreeFuzz and DeepRel, we use the same internal database for both TensorFlow and PyTorch libraries.

NablaFuzz [17]: To execute NablaFuzz, we employ the default hyperparameters outlined below. The number of mutants generated for each API is set to 1000, and the tool is executed in CUDA-enabled mode. Additionally, the timeout for each API is configured to be 300 seconds.

DocTer [10]: There are multiple DocTer hyperparameters that we have to configure. The most important parameter is the fuzzing mode which we set as *Conforming Inputs*. We also set *fuzz_optional_p=0.2*, *mutate_p=0.4*, and *time-out=10*. We fuzz each API 1000 times as suggested and use DocTer's own constraints and internal database for fuzzing.

TitanFuzz [12]: We opt for the default hyperparameters when running TitanFuzz. This entails enabling CUDA, employing the fitness function as the seed selection algorithm, and utilizing Thomson Sampling (TS) as the default operator selection algorithm. The seed pool size is set to 10, with a corresponding batch size of 10. Each API is allotted a running budget of 60 seconds, and the random seed used is 420.

AtlasFuzz [16]: To run AtlasFuzz, we adhere to their provided running instructions and adopt their default hyperparameters. Specifically, we choose a few-shot learning scenario for test case generation, as recommended in the original paper. In this configuration, we set *k_shotk_shot* to 10 and utilize the *Salesforce/codegen-350M-mono* model with a maximum length of 256. Each iteration involves generating a total of 100 test cases, with a batch size of 10. The codex engine selected is *code-davinci-002*, and the temperature is set to 0.8. Additionally, we configure the maximum tokens to be 256, and *top_p* is set at 0.95.

We experiment on a machine equipped with Intel(R) Core(TM) i7-10700F CPU @ 2.90GHz, NVIDIA GTX 1660 Ti GPU, 16GB of RAM, Ubuntu 22.04, Python 3.9 for different releases of PyTorch and TensorFlow. The time cost of running each fuzzer is shown in Table 4.

TABLE 4: Time cost of running each fuzzer (in terms of CPU hours)

Tools	PyTorch			TensorFlow			
	2.0.0	2.0.1	2.1.0	2.11.0	2.12.0	2.13.0	2.14.0
FreeFuzz	0.29	0.35	0.28	0.30	0.34	0.22	0.27
DeepRel	5.18	5.09	5.11	1.89	1.92	1.79	1.95
NablaFuzz	2.88	2.84	2.81	1.80	1.81	1.81	1.82
DocTer	1.72	1.56	1.53	1.71	1.78	1.81	1.86
TitanFuzz	6.05	7.51	6.10	0.30	1.16	0.28	0.27
AtlasFuzz	2.88	2.84	2.81	1.80	1.81	1.81	1.82

3.3. Evaluation Metrics

In this study, we use the following metrics to measure the effectiveness of the six DL fuzzers:

Number of generated test cases is a typical metric to assess the effectiveness of DL fuzzers. This metric counts the number of distinct test cases generated by each fuzzer. We averaged the number of generated test cases over different releases of PyTorch and TensorFlow.

Number of true bugs detected determines the number of real-world bugs discovered by a fuzzer. To calculate this metric, we performed a manual analysis on the log message of every test case generated by DL fuzzers and bugs (reported log message for each issue) in our benchmark dataset.

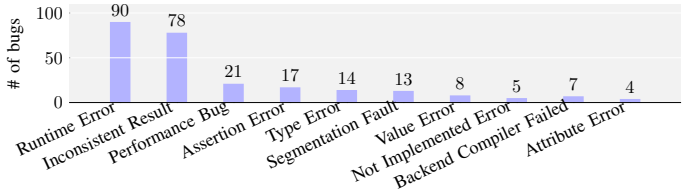

Figure 1: Top bug types in PyTorch library.

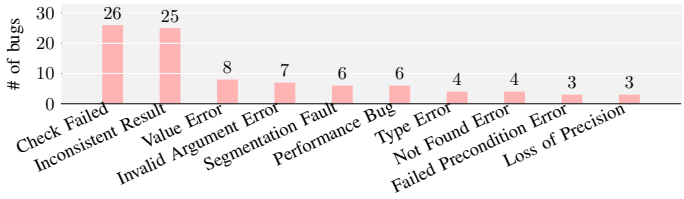

Figure 2: Top bug types in TensorFlow library.

We analyzed each log message in two scopes including the type of the bug and the message of the log message. Please see Section 4.2 for more details on our manual analysis.

Execution time: Given the distinct implementations and design assumptions of these tools, variations in execution times are expected. Consequently, we measure their performance in terms of CPU hours.

4. Results and Analysis

4.1. RQ1: Distribution of Bugs Types

Approach: In this RQ, we investigate the distribution of bug types in our benchmark dataset. As the basic information of each bug has already been collected in our data collection (Section 2.3), we directly analyze our bug dataset for answering this RQ. We also show the number of bugs in our curated benchmark dataset reported on each release of PyTorch and TensorFlow (shown in Table5).

Result: Table 5 shows the bug distribution on different releases of PyTorch and TensorFlow. We also show the number of bugs related to tensor calculation. As shown in the table, a large proportion of bugs in each release is related to tensor calculation, showing the correctness of our curated benchmark dataset as the tensor calculation is the core of DL libraries [51], [52], [59]. Figure 1 and Figure 2 show the distribution of bug types in the PyTorch and TensorFlow libraries, respectively. In the figures, the x-axis represents bug types, and the y-axis shows the number of issues reported for each bug type. As we can see, *Runtime Error* is the most prevalent bug type, with a significantly higher occurrence than other categories accounting for 90 bugs in PyTorch. *Inconsistent Result* and *Performance Bug* (The detailed distribution is shown in Figure 3) also exhibit notable frequencies, i.e., 78 and 21, while categories such as *Not Implemented Error* and *Backend Compiler Failed* have lower occurrences accounting for 7 and 4 bugs. Regarding

TABLE 5: Number of bugs in our benchmark dataset from different releases of PyTorch and TensorFlow. The numbers inside the parenthesis show the percentage of bugs that relate to tensor calculation per release.

Library	PyTorch			TensorFlow			
	2.0.0	2.0.1	2.1.0	2.11.0	2.12.0	2.13.0	2.14.0
# of bugs	108(60)	81(75)	114(78)	16(93)	29(72)	43(90)	21(100)

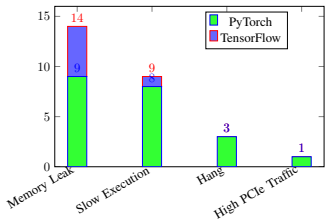

Figure 3: Detail distribution of *Performance Bug* types.

TensorFlow, it is observable that *Check Failed* is the most prevalent bug type, accounting for 26 issues. *Inconsistent Result* comes as the second most common, with 25 issues. The least frequent bug types are *Failed Precondition Error* and *Loss of Precision*, each accounting for 3 issues.

Answer to RQ1: *Runtime Error* and *Check Failed* are the most frequent bug types in the PyTorch and TensorFlow accounting for 90 and 26 bugs, respectively. *Inconsistent Result* is the second most frequent bug type in both libraries.

4.2. RQ2: Effectiveness of Fuzzers

Approach: RQ2 focuses on evaluating and discussing the correctness and quality of the generated test cases in connection with the ability of the studied DL fuzzers to detect real-world bugs effectively. As a result, the approaches to address RQ2 are as follows:

Step 1: Setup Execution Environments: We established multiple Anaconda environments for the targeted releases of PyTorch or TensorFlow, incorporating CUDA support and the necessary dependencies for the designated fuzzers. The installation process followed the instructions provided in the respective DL fuzzer's manual available within their replication package.

Step 2: Test Case Generation and Execution: With the Anaconda environments prepared, we executed DL fuzzers on each specific environment corresponding to the respective releases. The generated test cases were stored for subsequent execution. It is important to note that test case generation and execution were conducted as separate phases, primarily due to the limited computational resources of our machine. This approach ensured sufficient availability of CUDA and main memory, especially considering the substantial computational overhead associated with some of the investigated fuzzers (refer to Table 4). For test execution, we developed multiple Python and shell scripts to run the generated test cases on each environment(the scripts are available in the

replication package). Log messages of the test case executions were captured for subsequent analysis.

Step 3: Classify Generated Test Cases based on Execution Logs: In this phase, the three authors of the paper simultaneously analyzed the execution log messages of the generated test cases in a systematic approach, categorizing them into four main categories and seven subcategories, as illustrated in Table 6. The taxonomies were incrementally developed as the authors analyzed the log messages. This iterative process occurred in multiple rounds to address and resolve any potential disagreements.

Step 4: Bug Matching via Log Message Comparison: In this phase, we conducted a manual comparison between the ground truth log messages in our benchmark data to match bugs and the log messages of test cases to identify any matches. The three authors simultaneously examined two components of each log message: the type and the message. The types indicate the overall category of the bug, while the message precisely explains the impact of the bug.

Result: Table 7 shows the statistics of the generated test cases across different releases of PyTorch and TensorFlow. In this table, the first column shows the overall status categories of the generated test cases. The second column shows the breakdown for *Fail*, which is manually summarized by authors. The rest of the columns show the statistics of each fuzzer on PyTorch and TensorFlow libraries. Table 8 shows the number of true bugs detected by DL fuzzers.

Table 8 shows that *AtlasFuzz* outperforms other tools in detecting true bugs, boasting the highest counts for bug types and matching full log messages. However, this commendable bug detection performance is accompanied by a relatively high rate of failed test cases along with syntactically incorrect test cases. For instance, in PyTorch releases, *AtlasFuzz* exhibits the third-highest average count of *Invalid Test Input* (5.6k), surpassed only by *FreeFuzz* and *DeepRel*. Similarly, *AtlasFuzz* also ranks higher regarding *Invalid Test Case*, *Invalid Environment*, and *Syntax Error*. This trade-off between bug detection and failed test cases along with syntactically incorrect test cases indicates that *AtlasFuzz*, while excelling in identifying real-world bugs, experiences challenges in generating successful test cases. Please note that AtlasFuzz follows the same pattern on TensorFlow releases. *TitanFuzz* could only detect one bug among TensorFlow releases (2.14.0). The discrepancy in bug detection efficiency between *AtlasFuzz* and *TitanFuzz*, both being LLM-based fuzzers, could be attributed to their differing approaches in guiding the fuzzing process. *AtlasFuzz* appears to leverage a more sophisticated strategy by guiding its fuzzer generator based on real-world bug patterns. This targeted approach enables *AtlasFuzz* to align its test case generation with known bug scenarios, enhancing its ability to detect real-world issues efficiently. On the other hand, *TitanFuzz* relies on a set of in-place location-fixed mutation operators for fuzzing. The in-place mutation operators guide the fuzzer generator to generate code elements on predefined code location based on the nature of the operator (please see Table 1). While these operators may introduce diversity in the generated test cases, they might be less effective at mimicking specific real-world bug patterns. The use of broad mutation operators may result in a fuzzing process that lacks the precision and targeted approach seen in *AtlasFuzz*.

Regarding traditional DL fuzzers, *FreeFuzz* emerges as the most efficient tool, exhibiting superior performance on both PyTorch and TensorFlow releases when compared to *DeepRel* and *NablaFuzz*. The exceptional performance of *FreeFuzz* can be attributed to its utilization of multiple test oracles for bug detection. Unlike *DeepRel*, which relies solely on the inconsistency oracle between semantically related DL APIs, *FreeFuzz* employs a more diverse set of test oracles. This broader approach allows *FreeFuzz* to potentially capture a wider range of issues and anomalies within the DL libraries. On the other hand, *DocTer* follows a more specialized path, using only a crash oracle. While this approach might provide focused insights into potential bugs leading to crashes, it inherently limits its ability to detect a broader spectrum of bugs compared to *FreeFuzz*. The use of multiple test oracles by *FreeFuzz* contributes to its robust bug detection capabilities, setting it apart from *DocTer* and *DeepRel*. *DocTer* emerges as the second most efficient traditional fuzzer in real-world bug detection on PyTorch releases, trailing behind *FreeFuzz*. However, similar to *FreeFuzz*, *DocTer* faces the same challenges, notably a high number of *Invalid Test Input* and *Invalid Test Case* test cases. This trend underscores a general observation regarding traditional fuzzers—they showcase relatively good performance in terms of real-world bug detection. Nevertheless, this effectiveness is counterbalanced by a notable downside: a significant number of failed and syntactically incorrect test cases.

In total, the studied DL fuzzers successfully identified 13 unique bugs. Notably, AtlasFuzz emerges as the most proficient DL fuzzer, showcasing exceptional effectiveness in detecting real-world bugs across both PyTorch and TensorFlow. Following closely, FreeFuzz stands out as the second most effective fuzzer, successfully detecting three bugs in PyTorch releases. In contrast, both DocTer and TitanFuzz exhibit lower effectiveness, each detecting only one bug and being considered the least effective tool in the study.

Answer to RQ2: Overall, using LLM-based DL fuzzers is a *Win huge, loss is tiny* scenario where they showcase superior bug detection effectiveness with fewer failed test cases. *FreeFuzz* excels among traditional DL fuzzers while suffering the generation of a large number of invalid test inputs. These observations highlight the nuanced trade-offs in the capabilities of different fuzzing tools, each presenting its unique strengths and weaknesses.

4.3. RQ3: Characteristics of Detected Bug Types

Approach: To answer this research question, we conducted additional manual verification of the identified bug types by each DL fuzzer, presenting a summary in Figure 4.

TABLE 6: Status of the generated test cases and the corresponding explanation.

Status Category	Status Breakdown	Explanation
Success	-	Test cases that executed without encountering errors or crashes.
Syntax Error	-	Test cases have indentation or syntax-related issues.
Timed Out	-	Test cases that exceeded the allowed execution time.
Out of Memory	-	Test cases that depleted available memory.
Fail	Crash	Test cases resulting in crashing the Python interpreter.
	Invalid Test Input	The generated test inputs are not compatible with API input constraints.
	Missing Test Input	The required test inputs are missing.
	Invalid Test Case	The generated test cases present an invalid coding pattern.
	Assertion Error	Test cases failing assertion checks.
	Missing/Invalid Environment	The generated test cases miss or have an invalid environment or configuration.
	AD/ND Fail	Test cases fail due to errors during automatic and numerical differentiation.

TABLE 7: Statistics of the generated test cases averaged on different releases of PyTorch (PT) and TensorFlow (TF).

Status Categories	Status Breakdown	FreeFuzz		DeepRel		NablaFuzz		DocTer		TitanFuzz		AtlasFuzz	
		PT	TF	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF
Success	-	65k	67.5k	2.1k	1.7k	28.9k	15.7k	6.3k	14.1k	0	0	1.7k	1.5k
Syntax Error	-	9k	0	2.4k	0	-	-	0	0	6.7	0	442.3	315.3
Timed Out	-	7	2	0	5	-	-	0	0	0	0	0	0
Out of Memory	-	0	53.3	9	288.5	-	-	1	1	1	0	6.3	26.7
Fail	Crash	1	1.1k	0	563	41.7	17.9	0	727.5	0	1	0	7.3
	Invalid Test Input	85k	8.7k	4.8k	4.5k	-	-	30k	16.3k	634.3	241.0	5.6k	6.3k
	Missing/Corrupted Test Input	0	0	0	0	-	-	792.7	657	1	0	38.0	72.5
	Invalid Test Case	317.3	871.3	381.3	882.3	-	-	3.4k	2k	532	233.8	1.3k	1.2k
	Assertion Error	3.5k	0	0	0			0	0	34.7	2	27.0	2.0
	Invalid Environment	1.5k	0	0	1	-	-	197.3	6.7	2.0	1	106.3	124.5
	AD Fail	-	-	-	-	397	63k	-	-	-	-	-	-

TABLE 8: Total number of true bugs detected by DL fuzzers.

Tools	PyTorch			TensorFlow			
	2.0.0	2.0.1	2.1.0	2.11.0	2.12.0	2.13.0	2.14.0
FreeFuzz	1	1	1	0	0	0	0
DeepRel	1	0	0	0	0	0	0
NablaFuzz	0	0	0	0	0	0	0
DocTer	1	0	0	0	0	0	0
TitanFuzz	0	0	0	0	0	0	1
AtlasFuzz	4	1	1	1	1	1	3

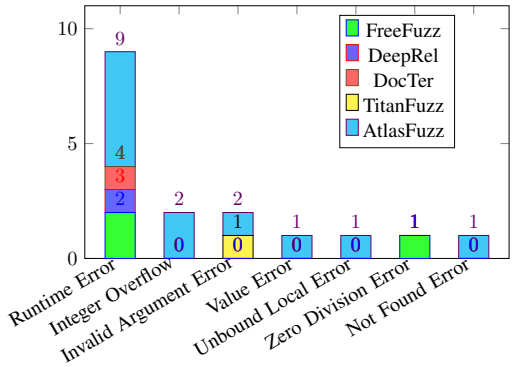

Figure 4: Distribution of detected bug types.

More precisely, all three authors of the paper simultaneously examined the log messages associated with the generated test cases from each DL fuzzer across multiple releases. For each log message, the bug types are then compared with the bug types within the log messages of actual bugs present in our benchmark dataset for the corresponding releases.

Result: Figure 4 shows the summary of bug types detected by the studied fuzzers. According to the depicted results, *Runtime Error* emerges as the most commonly detected bug type among the studied DL fuzzers, totaling 9 instances. Following, *Integer Overflow* and *Invalid Argument Error* stand out as the second most frequently identified bug types, accounting for 2 instances for each bug type respectively. *Value Error*, *Unbound Local Error*, *Zero Division Error*, and *Not Found Error* are the least detected bug types, each detecting only one instance.

Answer to RQ3: The most frequently identified bug type among the studied DL fuzzers is *Runtime Error*, with *Integer Overflow* and *Invalid Argument Error* ranking as the second most commonly detected bug types. Despite *Inconsistent Result* being the second most frequent bug type in our curated dataset, DL fuzzers face challenges in effectively detecting this complex bug type.

4.4. RQ4: Root Causes for Missing Bugs

Approach: To address this RQ, we conduct a comprehensive manual examination of the implementation and documentation of the studied DL fuzzers. Our goal is to

gain insight into their fuzzing capabilities and bug detection mechanisms by scrutinizing their mutation strategies and the support they provide for test oracles. Subsequently, we checked the real-world bugs overlooked by these fuzzers and developed taxonomies to categorize the underlying root causes.

Result: Table 9 shows the summary of the root causes of the missing real-world bugs. The first two columns show the overall broad categories as well as their corresponding breakdowns. The remaining columns show the missing bugs of the studied fuzzers over each DL library. We summarize the root causes into three high-level categories, i.e., *API Usage Context*, *Test Case Context*, *Test Input Context* with the corresponding breakdowns. We categorized the rest, which does not belong to any other category, as *Others*.

4.4.1. API Usage Context. We have summarized three subcategories of *API Usage Context* as shown in Table 9 which are explained in detail in the following paragraphs.

Lack of Modeling Multiple API Usage Patterns: While creating our benchmark data, we find that certain bugs manifest as a result of invoking specific APIs consecutively or adhering to particular usage patterns involving multiple API calls [26], [59]. Taking the bug with issue id 105901 from PyTorch library[8] as an example. The issue lies in the inability of *torchdynamo.export* to capture the default parameters of *torch.nn.MaxPool2d*. Unfortunately, all six DL fuzzers studied in this paper are incapable of modeling such usage scenarios in terms of DL APIs during their test case generation. Among traditional DL fuzzers, FreeFuzz and DocTer incorporate a single API within the generated test case, accompanied by the inclusion of input data. On the other hand, DeepRel and NablaFuzz attempt to model multiple API usages. However, it is worth noting that the complexity of the API usages they support is limited and falls short of capturing intricate usage patterns. Specifically, DeepRel restricts itself to incorporate semantically similar APIs using distance measures [60] (API pairs that manifest similar functionality, e.g., *torch.nn.AdaptiveAvgPool3d* and *torch.nn.AdaptiveMaxPool3d* in PyTorch). However, under this root cause category, not all API pairs have semantic similarity in terms of functionality. For example, in the above bug, the APIs *torch.nn.MaxPool2d* and *torchdynamo.export* are completely unrelated to each other in terms of functionality. While *torch.nn.MaxPool2d* performs 2D pooling operation on input tensors, *torchdynamo.export* is responsible for exporting torch models. In the case of NablaFuzz, its utilization involves testing DL APIs using automatic differentiation, while not all bugs in DL libraries are related to inconsistency during automatic differentiation.

LLM-based fuzzers, i.e., TitanFuzz and AtlasFuzz, also suffer the same issue. Specifically, TitanFuzz utilizes mutation operators such as*Suffix* and *Prefix* to instruct the LLM to generate code elements at already-specified locations in the test case body. However, it is important to note that these operators, while versatile, do not guarantee that LLM will consistently generate API calls in the intended locations as desired. For example, to detect the bug mentioned previously[9], TitanFuzz should use *Prefix* and *Postfix* (and also switch the operators through multiple rounds of test case generation which is also computationally expensive, as shown in Table 4). AtlasFuzz is designed with the expectation of generating test cases that assess multiple usages of an API by leveraging historical data known to trigger bugs. However, the performance of AtlasFuzz is heavily dependent on the quality of its constructed historical dataset. Unfortunately, AtlasFuzz historical dataset construction, which guides its fuzzer generator, is performed in an ad-hoc fashion, i.e., there is no certainty that how many bug records are specifically related to the usage of multiple APIs that are known to be buggy if being called together.

Lack of Modeling Function Decorators: In DL libraries, more specifically Python client APIs, function decorators are a powerful feature that allows the modification or extension of the behavior of DL APIs[10]. For example, *@tf.function* in TensorFlow, which is used to convert the target Python function into a TensorFlow computation graph, enabling performance improvements through graph optimization and the potential for running computations on specialized hardware like GPUs. Improper or missing usage of function decorations may introduce bugs in DL libraries. For example, the bug[11] that arises when applying the *@tf.function(input_signature=[_b])* decorator to the method *f* within the class *C*. When it comes to traditional fuzzers, none of them extend their modeling to test case elements beyond API calls. Both FreeFuzz and DocTer limit themselves to calling a single API within the generated test case, utilizing the created test inputs. On the other hand, DeepRel and NablaFuzz model API pairs but lack any incorporation of function decorations in their usage. Regarding LLM-based fuzzers, TitanFuzz encounters a limitation in the form of an arbitrary in-place mutation operator. While the *Prefix* operator aligns well with modeling function decorations, its arbitrary nature implies that it instructs LLM to generate any code elements before the target API without specific guidance. Also, the *Prefix* can be any code element and increases the false positive rate, if we only expect the test case to have function decorators along with the API calls.

Lack of Modeling Specific Devices: Usually, DL APIs provide support for running computations on different devices primarily to leverage hardware acceleration and parallelism [26], [31], [51], [61]. Running on different devices allows developers to take advantage of specialized hardware such as GPUs and TPUs to speed up training and inference processes [32], [33]. However, sometimes DL APIs fail to run on specific devices. For example, a bug in the PyTorch library[12] revolves around the use of the generator parameter in the *DataLoader* class. The generator parameter is used to set the random number generator for the data loading

8. https://github.com/pytorch/pytorch/issues/105901

9. https://github.com/pytorch/pytorch/issues/105901
10. https://book.pythontips.com/en/latest/decorators.html
11. https://github.com/tensorflow/tensorflow/issues/61712
12. https://github.com/pytorch/pytorch/issues/98792

TABLE 9: Summary of root causes of missing bugs of the six studied DL fuzzers.

Root Cause Categories	Breakdown	FreeFuzz		DeepRel		NablaFuzz		DocTer		TitanFuzz		AtlasFuzz	
		PT	TF	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF
API Usage Context	Lack of Modeling Multiple API Usage Pattern	20	3	20	3	20	3	18	4	44	8	44	8
	Lack of Modeling Function Decorators	1	1	1	1	1	1	1	2	5	2	5	2
	Lack of Modeling Specific Devices	8	1	8	1	8	1	6	1	13	1	13	1
Test Case Context	Lack of Modeling DL Execution Modes	-	2	-	2	-	2	-	-	3	3	3	3
	Lack of Supporting External Library	4	-	4	-	4	-	4	-	8	-	8	-
	Lack of Modeling Specific Code Pattern	5	1	5	1	5	1	6	-	10	1	10	1
Test Input Context	Lack of Modeling Specific Inputs	20	6	20	6	20	6	17	9	29	10	29	10
	Lack of Supporting Specific Data Types	6	8	6	8	6	8	11	6	14	11	14	11
	Lack of Generating Tensors with Incompatible Shapes	-	4	-	4	-	4	-	3	-	9	-	9
	Lack of Generating Large Input Tensor	2	8	2	8	2	8	2	1	4	12	4	12
	Lack of Generating Sparse Input Tensor	1	-	1	-	1	-	2	-	1	-	1	-
	Lack of Generating NaN input tensor	1	-	1	-	1	-	-	-	1	-	1	-
	Lack of Generating Empty Input Tensor	1	-	1	-	1	-	2	-	2	2	2	2
	Lack of Generating Zero Input Tensor	-	-	-	-	-	-	-	1	-	1	-	1
	Lack of Generating Large Integer Argument	1	5	1	5	1	5	-	2	1	8	1	8
	Lack of Generating Large List Element	1	1	1	1	1	1	-	-	1	-	1	-
	Lack of Generating Invalid String	-	-	-	-	-	-	-	1	-	-	-	-
	Lack of Generating Non ASCII Characters	1	-	1	-	1	-	1	-	1	-	1	-
Others	Others	3	2	3	2	3	2	6	2	11	2	11	2

process. The code passes a CUDA device generator to the *DataLoader*, but it raises a *Runtime Error* with the message *Expected a 'cpu' device type for generator but found 'cuda'*. This is because the current implementation of *DataLoader* in PyTorch expects the generator to be on the CPU, not on a CUDA device. DL fuzzers are unsuccessful in detecting this bug due to a contextual misunderstanding. Specifically, they are not aware that the input data, particularly tensors, possesses an optional parameter known as *device*, which informs the backend about the target device. Also, DL fuzzers need to be customizable according to which device they want to run. For example, they need to be able to model multiple other devices such as *MkldnnCPU*, *QuantizedCUDA*, *Meta*, *MPS backends*, etc[13].

4.4.2. Test Case Context. The root causes within this category pertain to various aspects encompassing the content of the test case. These include intricacies related to execution models, the representation of external libraries in the test case, configurations of the testing environment embedded within the test case, and distinctive code patterns that can inadvertently trigger bugs.

Lack of Modeling DL Execution Modes: Generally, there are two execution modes in DL libraries [62], [63], i.e., graph mode and eager mode. In graph mode, also known as JIT (Just-In-Time) Compilation, instead of executing operations eagerly, PyTorch and TensorFlow construct a computational graph that represents the operations and their dependencies. This graph is then compiled and optimized for efficient execution. Graph mode can lead to improved performance for certain workloads, especially when running on hardware accelerators like GPUs or TPUs [31]–[33]. In the eager mode, on the other hand, operations are executed eagerly and the results are returned immediately. It is similar to the way operations are executed in imperative programming languages like Python. In certain situations, utilizing DL APIs with identical input data but under different execution modes can lead to *Inconsistent Result*. A notable illustration of this inconsistency is evident in a bug within the TensorFlow library[14] shown in Figure 5, where the outcomes of *tf.raw_ops.Acos* exhibit discrepancies between JIT compile mode and eager mode. From the perspective of DL fuzzers, it is important to note that they overlook such bugs due to their inability to generate test cases that run APIs under both graph and eager modes. This limitation results in a substantial number of inconsistency bugs being overlooked during the fuzzing process.

```
1  @tf.function(jit_compile=True)
2  def fuzz_jit():
3      y = tf.raw_ops.Acos(
4          x = x
5      )
6      return y
7  def fuzz_normal():
8      y = tf.raw_ops.Acos(
9          x = x
10     )
11     return y
12 y1 = fuzz_jit()
13 print('[+] JIT ok')
14 y2 = fuzz_normal()
15 print('[+] Normal ok')
16 np.testing.assert_allclose(y1.numpy(), y2.
       numpy(), rtol=1e-4, atol=1e-4)
```

Figure 5: An example of *Inconsistent Result* bug from TensorFlow library missed by DL fuzzers due to *Lack of Modeling DL Execution Modes*. The function decorator in line 1 converts *fuzz_jit* to a computation graph.

13. For the full list of devices, please see the data which is available in the replication package.

14. https://github.com/tensorflow/tensorflow/issues/60764

Lack of Supporting External Library: One of the unique characteristics of DL applications is their high dependency on external libraries [26], [51]. This intricate dependency on external libraries often results in different bugs. For example, in a bug from PyTorch library[15], the usage of hugging face GPT models in conjunction with *fully_sharded_data_parallel.FullyShardedDataParallel* results in *Runtime Error*. The reason DL fuzzers missed this bug is that they typically focus on generating input data for DL APIs, exploring variations in tensor shapes, data types, and other parameters related to the DL library. Hence, they cannot handle scenarios where DL APIs are used in conjunction with external libraries. More specifically, DL fuzzers are often designed to target specific DL frameworks and their APIs. They may not be equipped to understand or handle interactions with external libraries that are beyond the scope of the DL framework. Moreover, DL fuzzers generate inputs to explore different code paths, but they might not effectively create inputs that involve interactions with external libraries. Generating meaningful inputs for such scenarios requires a deep understanding of how DL APIs can be used with external libraries[16].

Lack of Modeling Specific Code Pattern: In the process of curating our benchmark data, we noticed that certain issues only manifest when specific code patterns are developed by the users. One illustrative example is a bug identified in the PyTorch library[17], wherein an inconsistency arises between two calls to *torch.tensor* with identical input data but in different orders. To elaborate, when *torch.Tensor.__getitem__ = None* is invoked between the two API calls, the second API call encounters a failure. The bugs in this category are very hard to detect since the DL Fuzzers should have a solid understanding of API code context and how to invoke them.

4.4.3. Test Input Context. The root causes of missing detecting bugs in this category are related to the inputs that are fed into DL APIs.

Lack of Supporting Specific Inputs Generation: The pivotal aspect in the development of DL applications lies in the data. DL APIs are designed to accommodate a diverse array of data formats and types through specific combinations. Unfortunately, introducing malicious or invalid data to DL APIs often leads to unpredictable behavior or runtime errors. For instance, consider a bug in the PyTorch library[18], illustrated in Figure 6. In this case, *torch.matmul* receives a batched Compressed Sparse Row (CSR) matrix intended for CUDA processing. Regrettably, the API is unable to handle the CSR matrix and raises a *RuntimeError: Sparse CSR tensors do not have strides*. Unfortunately, this bug was missed by the studied DL fuzzers since they are not able to generate CSR matrices.

15. https://github.com/pytorch/pytorch/issues/100069
16. We have summarized external libraries that trigger bugs when used with DL APIs in our replication package.
17. https://github.com/pytorch/pytorch/issues/98948
18. https://github.com/pytorch/pytorch/issues/98675

```
1 import torch
2 device = |torch.device("cuda:0")|
3 a = torch.tensor([[ [1.0, 0.0], [2.0, 1.0]],
4              [ [0.1, 0.1], [0.0, 2.0], ]
5 ]).to_sparse_csr().|to(device)|
6 b = torch.randn(2, 2).to(device)
7 print(torch.matmul(a, b))
```

Figure 6: An example bug from PyTorch library missed by DL fuzzers as the tensor *a* defined in line 3 is targeted to run on GPU. However, these fuzzers did not support tensor multiplication over GPU.

```
1 import torch
2 a = torch.randn((2,3), dtype=|torch.
    float8_e5m2|)
3 b = torch.randn((2,3), dtype=torch.
    float8_e5m2)
4 a = torch.tensor(1.2, dtype=torch.float8_e5m2
    )
5 b = torch.tensor(2.3, dtype=torch.float8_e5m2
    )
```

Figure 7: An example bug from PyTorch library was missed by DL fuzzers as they did not support the custom floating point data type *torch.float8_e5m2)*.

Lack of Modeling Specific Data Types: DL APIs have a wide variety of data types in their input constraints. However, errors may occur when developers want to use their custom data types for targeted devices. For example in a bug from PyTorch library[19], as shown in Figure 7, when the developer tries to create tensor variables(lines 1 and 3) with the custom data type *torch.float8_e5m2*, the backend throws *Runtime Error*. This bug was missed by the studied fuzzers because they are not able to cover custom data types. They merely support typical data types that are available in DL input constraints such as *torch.int8*, *torch.int16*, *torch.int32*, and *torch.int64*, etc.

Answer to RQ4: The root causes for missing detecting bugs can be summarized into three main categories, i.e., *API Context*, *Test Case Context*, and *Test Input Context*, each with specific breakdowns.

5. Implications for Improving DL Fuzzers

In this section of the paper, based on the root cause analysis of missed bugs, we propose a set of actionable guidelines to improve the studied DL fuzzers in terms of effectively detecting new bugs. Given space constraints, we only provide improvements based on the major root cause patterns.

19. https://github.com/pytorch/pytorch/issues/107087

5.1. Modeling Multiple API Usage Patterns

It is crucial to underscore the significance of DL fuzzers, in effectively modeling the intricate API usage sequences. We recommend the following improvements:

Improvement 1: Mine API Usage Sequence Patterns: This approach revolves around mining frequent API usage sequence patterns from bug reports using frequent itemset or graph mining algorithms [59], [64], [65]. The resulting sequences can be used as either rules to guide traditional fuzzer generators or as prompts that guide LLM-based fuzzer generators. For example, to improve TitanFuzz, the API sequences can be used along with in-place (please see Table 1) operators to increase the precision of the generated test cases, which can guide TitanFuzz to insert bug-prone API calls specifically in locations indicated by *Prefix* and *Postfix* operators, rather than randomly inserting arbitrary code statements.

Improvement 2: Parameter Exploration: Even though the most important step is to model API usage sequences and their interactions, the fuzzers must parameterize the test inputs for API calls to cover a diverse range of scenarios. Vary input sizes, data types, and other relevant parameters to explore different paths of API usage together. In other words, it is advisable to conduct parameter exploration for both the API under test and the bug-prone APIs inserted into the generated test case, which allows for simulating various API usage sequences. Additionally, we recommend ensuring that the generated test case incorporates diverse test inputs for the involved APIs.

Improvement 3: Environment-Aware Fuzzing: The DL fuzzing tools investigated in this study have the potential for extension to produce test cases that are environment-aware. This involves incorporating a single API with identical test input data in diverse execution models, namely, eager mode or graph mode. Additionally, these test cases must incorporate an inconsistency oracle, enabling the detection of different output values resulting from the same APIs that are being called under different APIs.

5.2. Extend Test Input Exploration

In this subsection, we suggest two improvements regarding extending the test inputs to DL APIs.

Improvement 1: Generate Extreme Corner Case Values The examined DL fuzzers have the potential for expansion beyond the mere generation of random corner case values for various parameter types within DL APIs. Instead of solely depending on random value generation, we propose a shift towards utilizing historical knowledge sourced from open repositories to inform the process of generating corner cases. For instance, traditional DL fuzzers, as observed in this study, typically generate large input tensors or integer arguments at random. However, a more effective approach involves extending these tools to generate substantial test inputs based on historical data, specifically targeting inputs known to have triggered bugs in the past.

Improvement 2: Model Parameter Correlations Given that a significant portion of DL bugs arises from tensor calculations (as detailed in Table 5), we suggest the studied DL fuzzers enhance their mutation operators to cover the intricate correlation among the input parameters within DL APIs, specifically tensors and their corresponding index arguments. This correlation modeling deliberately violates DL API input constraints, thereby effectively triggering bugs in the backend implementation of DL libraries.

6. Related Work

6.1. Fuzzing traditional software

In recent years, benchmarking studies in software engineering have attracted significant attention to assessing the efficiency and shortcomings of various software engineering methodologies, tools, or procedures in domains including but not limited to static bug detection [66]–[71], reviewer recommendation system [72], API recommendation [59], [73], and automatic test case generation [74]. Also, there have been multiple studies on Benchmarking fuzz testing tools on general software [75]–[78]. The reason behind multiple studies benchmarking fuzz testing tools is twofold. First, the evaluation of fuzzers is challenging since there is no solid metric to compare fuzzers. Crash oracles are the most straightforward technique, though they suffer from deduplication problem [79], [80]. Second, the lack of a benchmark dataset results in an unreliable comparison of fuzzers, which may introduce bias in the results. Magma [75] is one of the first steps towards benchmarking fuzzers for general software to tackle the aforementioned issues. It created a benchmark dataset that contains real bugs collected from general software to enable uniform fuzzer evaluation and comparison. Magma allows for the realistic evaluation of fuzzers against a broad set of targets and enables the collection of bug-centric performance metrics independent of the fuzzer. Metzman et al. [76] introduced FuzzBench as a free, open-source platform for fuzzer evaluation. Natella et al. [77] proposed a benchmark for stateful fuzzing of network protocols called ProFuzzBench. The benchmark consists of open-source programs that implement major network protocols, and tools for automating fuzzing experiments. Bohme et al. [78] is the most recent study on benchmarking fuzz testing tools on general software. The authors evaluated 10 fuzzers on 24 targets containing 13.2M lines of code and 268 known bugs in total. The study focuses on investigating the code coverage as a proxy measure w.r.t the detection effectiveness of the studied fuzzers.

6.2. Fuzzing DL compilers

Recently, many researchers have been attracted to research DL compilers [29], [61], [81]. Hence high-performance computing is essential in DL application development, often DL models are compiled and optimized [32], [33] based on specific platforms for safety-critical application domains. One of the first approaches for DL compiler

testing is proposed by [81] where they proposed TVMFuzz as a proof-of-concept application of their root cause categorization of bugs in DL compilers. They implemented TVMFuzz based on TVM tests that have been developed already. NNSmith [61] is one of the most recent DL compiler fuzzers proposed in the literature. As generating valid inputs to computation graphs is important, NNSmith uses a set of abstract operator models for the valid generation of computation graphs. NNSmith is designed in such a way that it is compatible with the ONNX format. TZer [29] proposed a functional tensor compiler fuzzer that incorporates coverage guidance and utilizes a combined approach of IR (Intermediate Representation) and pass mutation. Diverging from conventional compiler fuzzers, Tzer conducts simultaneous IR and pass mutation, enabling the exploration of diverse program states. In contrast to the aforementioned research directions, our current focus is on benchmarking DL fuzzers that undergo testing typical DL libraries,i.e., PyTorch and TensorFlow, as opposed to the evaluation of DL compilers. However, it's important to note that our future direction will involve benchmarking DL compilers, thereby serving as a complementary extension to our ongoing work.

6.3. Fuzzing DL libraries

Unlike the previous line of research in which benchmarking fuzzers on traditional software and DL compilers has received significant attention, DL fuzzers [9]–[12], [16], [17] are a very new area of research, there have not been a lot of benchmarking studies on them yet. Furthermore, benchmarking studies necessitate a large and carefully curated dataset of bugs for evaluation [75], which is difficult to obtain for DL fuzzers. In this study, we take the first step toward benchmarking DL fuzzers. To accomplish this, we use a curated dataset of real-world bugs, which includes more bugs than previous studies, as we collected 412 bugs while existing studies have used 268 known bugs [78]. We run three recently introduced DL fuzzers, including DocTer [10], FreeFuzz [9], and DeepRel [11], NablaFuzz [17], TitanFuzz [12], and AtlasFuzz [16] on TensorFlow and PyTorch libraries with more than 4.3 million lines of code. Based on the obtained results, our study performs an in-depth root-cause analysis of why the studied fuzzers miss real-world bugs. We then categorized the root causes into high-level and fine-grained root causes, which we believe are significant in improving the studied fuzzers. Based on the root cause analysis, our study provides a set of solid recommendations to improve the fuzzers in terms of detection effectiveness.

There is also a recently introduced DL fuzzer named IvySyn [82]. IvySyn is designed to facilitate automatic test case generation by focusing on the statically typed nature of the backend implementation in TensorFlow and PyTorch, i.e., written in C/C++. The underlying hypothesis is that the static typing inherent in C/C++ enables type-aware fuzzing. Given their dynamic nature, this proves beneficial in contrast to the computational overhead incurred when attempting to capture DL types during the fuzzing of Python client APIs. IvySyn, utilizing these types, constructs a set of proof-of-concept mutation operations, which it then employs in the test case generation process. It is important to clarify that, for our evaluation in this paper, we have chosen to exclude IvySyn. The rationale is that IvySyn has heterogeneous instrumentation and fuzzing where it primarily concentrates on the functions that are implemented in C++, as opposed to the studied DL fuzzers where they directly perform fuzzing on front-end APIs.

7. Threats to validity

Internal Validity: In this section of the paper, we elaborate on the possible threats to the validity of our findings by benchmarking DL fuzzers. To avoid internal validity regarding running the studied fuzzers, we followed the original papers of the studied fuzzers to set up and run the fuzzers on the PyTorch and TensorFlow libraries. To eliminate potential biases in our collected dataset, we conducted a thorough comparison of all reported bugs in the replication package of DL fuzzers with the bugs we collected and removed the overlaps. To ensure the accuracy of our root cause analysis taxonomies, the three primary authors cross-checked the created taxonomies and resolved any disagreements through multiple rounds of collaboration.

External validity: To guard against external validity, we run the fuzzers on two widely used DL libraries with 4.3 million lines of code. We also collect a significant number of real-world bugs (412 bugs) from public and open-source domains such as the GitHub repository of the studied DL libraries. In our selection of DL fuzzers, we opted for widely used and open-source tools, both traditional and LLM-based fuzzers. This choice was made to enhance the reliability of our findings.

8. Conclusion

In this work, we conduct a benchmark study to evaluate state-of-the-art DL fuzzers, i.e., FreeFuzz, DeepRel, NablaFuzz, DocTer, TitanFuzz, and AtlasFuzz. Specifically, we first collected a curated dataset of real-world DL bugs, we then executed the selected fuzzers on PyTorch and TensorFlow libraries, having more than 4.3M lines of code. We find that *Runtime Error* and *Check Failed* are the most frequent bug types in PyTorch and TensorFlow. Moreover, we find that while LLM-based DL fuzzers show- case superior bug detection effectiveness with fewer failed test cases, FreeFuzz excels among traditional DL fuzzers in terms of overall performance, albeit with challenges related to the generation of valid test inputs. Ultimately, we conducted an in-depth root-cause analysis of why the studied fuzzers miss real-world bugs based on the obtained results. We then categorized the root causes into high-level and fine-grained root causes that are significant in improving the studied fuzzers. Based on the root cause analysis, we provided solid recommendations to improve the fuzzers' detection effectiveness.

References

[1] V. J. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1024–1036.

[2] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 765–777.

[3] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[4] Facebook. (2019) American fuzzy lop. [Online]. Available: https://github.com/google/AFL/releases

[5] K. Serebryany. (2015) libfuzzer a library for coverage-guided fuzz testing. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[6] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers." in *USENIX Security Symposium*, 2019, pp. 1949–1966.

[7] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, "Ems: History-driven mutation for coverage-based fuzzing," in *29rd Annual Network and Distributed System Security Symposium, NDSS*, 2022, pp. 24–28.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[9] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," *arXiv preprint arXiv:2201.06589*, 2022.

[10] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. 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*, 2022, pp. 176–188.

[11] Y. Deng, C. Yang, A. Wei, and L. 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*, 2022, pp. 44–56.

[12] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. 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*, 2023, pp. 423–435.

[13] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: whole-proof generation and repair with large language models," *arXiv preprint arXiv:2303.04910*, 2023.

[14] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," *arXiv preprint arXiv:2309.00608*, 2023.

[15] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," *arXiv preprint arXiv:2307.14991*, 2023.

[16] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *arXiv preprint arXiv:2304.02014*, 2023.

[17] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," *arXiv preprint arXiv:2302.04351*, 2023.

[18] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.

[19] J. Ba, G. J. Duck, and A. Roychoudhury, "Efficient greybox fuzzing to detect memory errors," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[20] A. Kallingal Joshy and W. Le, "Fuzzeraid: Grouping fuzzed crashes based on fault signatures," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[21] H. Lee, S. Kim, and S. K. Cha, "Fuzzle: Making a puzzle for fuzzers," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[22] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free dbms fuzzing," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[23] Y. Yu, X. Jia, Y. Liu, Y. Wang, Q. Sang, C. Zhang, and P. Su, "Htfuzz: Heap operation sequence sensitive fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[24] S. Groß. (2019) coverage-guided fuzzer for dynamic language interpreters based on a custom intermediate language. [Online]. Available: https://github.com/googleprojectzero/fuzzilli

[25] R. Simhambhatla, K. Okiah, S. Kuchkula, and R. Slater, "Self-driving cars: Evaluation of deep learning techniques for object detection in different driving conditions," *SMU Data Science Review*, vol. 2, no. 1, p. 23, 2019.

[26] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Characterizing and understanding software security vulnerabilities in machine learning libraries," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 27–38.

[27] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Finding deep-learning compilation bugs with nnsmith," *arXiv preprint arXiv:2207.13066*, 2022.

[28] Q. Su, C. Geng, G. Pekhimenko, and X. Si, "Torchprobe: Fuzzing dynamic deep learning compilers," *arXiv preprint arXiv:2310.20078*, 2023.

[29] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–26, 2022.

[30] W. Luo, D. Chai, X. Ruan, J. Wang, C. Fang, and Z. Chen, "Graph-based fuzz testing for deep learning inference engines," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 288–299.

[31] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[32] "Nvidia® tensorrt™, an sdk for high-performance deep learning inference, includes a deep learning inference optimizer and runtime that delivers low latency and high throughput for inference applications." [Online]. Available: https://developer.nvidia.com/tensorrt

[33] "Production-grade ai engine to speed up training and inferencing in your existing technology stack." [Online]. Available: https://onnxruntime.ai/docs/performance/graph-optimizations.html

[34] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[35] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *European conference on machine learning*. Springer, 2005, pp. 437–448.

[36] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Osdi*, vol. 16, no. 2016. Savannah, GA, USA, 2016, pp. 265–283.

[38] Z. Wang, M. Yan, J. Chen, S. Liu, and D. 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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 788–799. [Online]. Available: https://doi.org/10.1145/3368089.3409761

[39] H. V. Pham, T. Lutellier, W. Qi, and L. 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)*. IEEE, 2019, pp. 1027–1038.

[40] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks," *arXiv preprint arXiv:2203.02660*, 2022.

[41] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[42] G. Algan and I. Ulusoy, "Image classification with deep learning in the presence of noisy labels: A survey," *Knowledge-Based Systems*, vol. 215, p. 106771, 2021.

[43] F. Mahdisoltani, G. Berger, W. Gharbieh, D. Fleet, and R. Memisevic, "Fine-grained video classification and captioning," *arXiv preprint arXiv:1804.09235*, vol. 5, no. 6, 2018.

[44] F. Ertam and G. Aydın, "Data classification with deep learning using tensorflow," in *2017 international conference on computer science and engineering (UBMK)*. IEEE, 2017, pp. 755–758.

[45] Y. Lv, B. Liu, J. Zhang, Y. Dai, A. Li, and T. Zhang, "Semi-supervised active salient object detection," *Pattern Recognition*, vol. 123, p. 108364, 2022.

[46] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, and C. Rother, "Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 1025–1032.

[47] R. Kulkarni, S. Dhavalikar, and S. Bangar, "Traffic light detection and recognition for self driving cars using deep learning," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–4.

[48] S. Minaee and Z. Liu, "Automatic question-answering using a deep similarity neural network," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2017, pp. 923–927.

[49] R. G. Athreya, S. K. Bansal, A.-C. N. Ngomo, and R. Usbeck, "Template-based question answering using recursive neural networks," in *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*. IEEE, 2021, pp. 195–198.

[50] P. K. Roy, "Deep neural network to predict answer votes on community question answering sites," *Neural Processing Letters*, vol. 53, no. 2, pp. 1633–1646, 2021.

[51] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[52] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Automatic static vulnerability detection for machine learning libraries: Are we there yet?" in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 795–806.

[53] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021.

[54] Y. Ishida, T. Shimizu, and M. Yoshikawa, "An analysis and comparison of keyword recommendation methods for scientific data," *International Journal on Digital Libraries*, vol. 21, no. 3, pp. 307–327, 2020.

[55] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.

[56] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[57] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.

[58] X. Han and T. Yu, "An empirical study on performance bugs for highly configurable software systems," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.

[59] M. Wei, Y. Huang, J. Wang, J. Shin, N. S. Harzevili, and S. Wang, "Api recommendation for machine learning libraries: how far are we?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 370–381.

[60] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[61] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. 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*, 2023, pp. 530–543.

[62] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[63] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[64] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer, 2009, pp. 318–343.

[65] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.

[66] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 317–328.

[67] D. A. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 980–982.

[68] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *15th International symposium on software reliability engineering*. IEEE, 2004, pp. 245–256.

[69] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 97–106.

[70] G. Chatzieleftheriou and P. Katsaros, "Test-driving static analysis tools in search of c code vulnerabilities," in *2011 IEEE 35th annual computer software and applications conference workshops*. IEEE, 2011, pp. 96–103.

[71] D. A. Tomassi and C. Rubio-González, "On the real-world effectiveness of static bug detectors at finding null pointer exceptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 292–303.

[72] I. X. Gauthier, M. Lamothe, G. Mussbacher, and S. McIntosh, "Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 30–41.

[73] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. R. Lyu, "Revisiting, benchmarking and exploring api recommendation: How far are we?" *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1876–1897, 2022.

[74] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1548–1560.

[75] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[76] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1393–1403.

[77] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 662–665.

[78] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1621–1633.

[79] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, "Igor: Crash deduplication through root-cause clustering," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3318–3336.

[80] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[81] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 968–980.

[82] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "{IvySyn}: Automated vulnerability discovery in deep learning frameworks," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2383–2400.