Generation-based Differential Fuzzing for Deep Learning Libraries

JIAWEI LIU, State Key Laboratory for Novel Software Technology, Nanjing University, China
YUHENG HUANG and ZHIJIE WANG, University of Alberta, Canada
LEI MA, The University of Tokyo, Japan and University of Alberta, Canada
CHUNRONG FANG, MINGZHENG GU, XUFAN ZHANG, and ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Deep learning (DL) libraries have become the key component in developing and deploying DL-based software nowadays. With the growing popularity of applying DL models in both academia and industry across various domains, any bugs inherent in the DL libraries can potentially cause unexpected server outcomes. As such, there is an urgent demand for improving the software quality of DL libraries. Although there are some existing approaches specifically designed for testing DL libraries, their focus is usually limited to one specific domain, such as computer vision (CV). It is still not very clear how the existing approaches perform in detecting bugs of different DL libraries regarding different task domains and to what extent. To bridge this gap, we first conduct an empirical study on four representative and state-of-the-art DL library testing approaches. Our empirical study results reveal that it is hard for existing approaches to generalize to other task domains. We also find that the test inputs generated by these approaches usually lack diversity, with only a few types of bugs. What is worse, the false-positive rate of existing approaches is also high (**up to 58%**). To address these issues, we propose a *guided differential fuzzing approach based on generation*, namely, *Gandalf*. To generate testing inputs across diverse task domains effectively, *Gandalf* adopts the context-free grammar to ensure validity and utilizes a *Deep Q-Network* to maximize the diversity. *Gandalf* also includes 15 metamorphic relations to make it possible for the generated test cases to generalize across different DL libraries. Such a design can decrease the false positives because of the semantic difference for different APIs. We evaluate the effectiveness of *Gandalf* on nine versions of three representative DL libraries, covering 309 operators from computer vision, natural language processing, and automated speech recognition. The evaluation results demonstrate that *Gandalf* can effectively and efficiently generate diverse test inputs. Meanwhile, *Gandalf* successfully detects five categories of bugs with **only 3.1%** false-positive rates. We report all 49 new unique bugs found during

This work is supported partially by National Natural Science Foundation of China (61932012, 62141215, 62372228), CCF-Huawei Populus Grove Fund (CCF-HuaweiSY202306), Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003), Canada CIFAR AI Chairs Program, the Natural Sciences and Engineering Research Council of Canada (NSERC No. RGPIN-2021-02549, No. RGPAS-2021-00034, No. DGECR-2021-00019), as well as JST-Mirai Program Grant No.JPMJMI20B8, JSPS KAKENHI Grant No.JP21H04877, No. JP23H03372.

Authors' addresses: J. Liu, C. Fang (Corresponding author), M. Gu, X. Zhang, and Z. Chen, State Key Laboratory for Novel Software Technology, Nanjing University, 22 Hankou Road, Nanjing, Jiangsu, China, 210093; e-mails: jw.liu@smail.nju.edu.cn, fangchunrong@nju.edu.cn, {MF21320043, zhangxufan}@smail.nju.edu.cn, zychen@nju.edu.cn; Y. Huang and Z. Wang, University of Alberta, 9211 116 Street NW, Edmonton, Alberta, Canada, T6G 1H9; e-mails: {yuheng18, zhijie.wang}@ualberta.ca; L. Ma, The University of Tokyo, 7 Chome-3-1 Hongo, Tokyo, Tokyo, Japan, 113-8654 and University of Alberta, 9211 116 Street NW, Edmonton, Alberta, Canada, T6G 1H9; e-mail: ma.lei@acm.org.

1049-331X/2023/12-ART50 $15.00
https://doi.org/10.1145/3628159

the evaluation to the DL libraries' developers, and most of these bugs have been confirmed. Details about our empirical study and evaluation results are available on our project website.[1]

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Software testing, deep learning libraries, generation-based fuzzing

ACM Reference format:
Jiawei Liu, Yuheng Huang, Zhijie Wang, Lei Ma, Chunrong Fang, Mingzheng Gu, Xufan Zhang, and Zhenyu Chen. 2023. Generation-based Differential Fuzzing for Deep Learning Libraries. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 50 (December 2023), 28 pages.
https://doi.org/10.1145/3628159

1 INTRODUCTION

Deep learning (DL) libraries are foundational infrastructures for developing DL-based software in various domains, e.g., autonomous driving [55], medical diagnosis [12], and speech recognition [17]. To ease developers' work, DL libraries usually encapsulate complex DL algorithms into DL operators and provide these operators in the form of **Application Programming Interfaces (APIs)** [29]. The development process of DL software hence becomes more efficient, and DL libraries significantly reduce the learning cost for developers [61]. While the recent advance in DL libraries also drew much attention to their qualities. In practice, bugs usually exist in the operators of DL libraries. Once the buggy operators are used in DL software, even a simple bug can lead vast amounts of software to severe performance issues and degradation [1]. Considering that DL software could be used in various industrial applications, the quality assurance of DL libraries has become an urgent and essential need.

To address this, several recent attempts have been made to test DL libraries, among which a majority of these works adopt the differential fuzzing strategy [22, 24, 48, 58]. In the context of fuzzing DL libraries, a DL model with corresponding tensor data automatically generated from a seed model is considered as a test input [61]. A seed model is selected from existing task domain and built with DL library operators. Differential testing employs the same test input to run different DL libraries and measures the difference between their outputs. Any unexpected behavior (e.g., crashes, inconsistent outputs) indicates a bug [63]. However, even though the existing testing approaches have shown promising performance in some scenarios, it remains unclear how these approaches would work when testing DL libraries with seed models from a wide range of task domains. To bridge this gap, we first conduct an empirical study. In the study, we investigate how the differential and fuzzing techniques in the existing work have contributed to detecting bugs in different DL libraries. To this end, we leverage four representative testing approaches [8]: (1) CARDLE [48], (2) AUDEE [24], (3) LEMON [58], and (4) MUFFIN [22] to test 120 operators from four deep learning libraries: TensorFlow, Theano, CNTK, and MXNet. These approaches cover four different implementations frequently used in DL library testing. Our study aims to answer the following two research questions:

- **RQ1 (seeds of existing approaches):** How does fuzzing affect DL library testing? How do the seeds of fuzzing affect the existing testing approaches in different domains?
- **RQ2 (bugs found by existing approaches):** What kind of bugs can be found by the existing approaches? Do the existing differential approaches trigger any false positives?

[1] https://sites.google.com/view/gandalf4dll

Since the existing approaches for testing DL libraries rely on the seeds of fuzzing, RQ1 is proposed to evaluate the performance of these fuzzing approaches. To answer RQ1, we utilize seed models from three task domains, i.e., **computer vision (CV), natural language processing (NLP)**, and **automated speech recognition (ASR)** domains. We find that seed models employed by the existing fuzzing approaches have obvious impacts on testing **effectiveness** (i.e., the number of detected bugs). In addition, since the test inputs generated by the existing approaches are highly similar, the bug-triggering inputs generated by the existing fuzzing approaches may lack **diversity**. As a result, the capability of bug detection is limited. Considering the existing DL library testing approaches adopt the differential technique for identifying bugs, RQ2 is proposed to evaluate the performance of these differential approaches. To answer RQ2, we analyze the detected bugs in RQ1 and group them into four categories: (1) implementation bugs, (2) implementation differences, (3) precision bugs, and (4) random errors. Among the four categories, bugs related to implementation differences and random errors are **false positives** rather than real bugs. We find that most detected bugs of previous work are false positives caused by implementation differences such as initialization differences, data format differences, operator setting differences, and so on. Since the existing testing approaches do not take any solutions to address the DL library implementation differences, they may raise false positives caused by the flawed differential setting.

Our findings from the empirical study motivate us to propose and investigate a novel technique named *Gandalf*, a **g**ener**a**tio**n**-based **d**ifferenti**al** **f**uzzing approach for DL libraries. To ensure the **effective** test inputs generation in different domains, *Gandalf* generates test inputs by combining different operators following the ***context-free grammar*** **(CFG)** of DL models in different domains. *Gandalf* further employs a ***Deep Q-Network*** **(DQN)** as the guidance for combination to achieve high **diversity**. To reduce **false positives** caused by implementation differences and ensure the test inputs are equivalent for differential testing, we design equivalent **metamorphic relations (MRs)** for *Gandalf* according to initialization transformation, data format transformation, and setting transformation. With the support of equivalent MRs, *Gandalf* can automatically transfer operator graphs for different DL libraries. These equivalent MRs could also be generalizable to a large number of operators and DL libraries.

To evaluate the performance of *Gandalf*, we use *Gandalf* to test three representative open-source DL libraries with different development paradigms, i.e., TensorFlow with static graphs, PyTorch with dynamic graphs, and Jittor with fusion graphs. Our testing covers 309 different operators. Overall, we investigate the following three research questions when evaluating *Gandalf*:

- **RQ3 (effectiveness): How does *Gandalf* perform in detecting bugs of DL libraries? Does *Gandalf* reduce the effect of seed models from different task domains?** In RQ3, we conduct an evaluation to investigate the effectiveness of *Gandalf* on different task domains. Our experiment shows that *Gandalf* can effectively detect bugs in DL libraries of different domains. During the evaluation, we find 49 new unique bugs. We report all of them to library developers and receive positive confirmations on these bugs.
- **RQ4 (diversity and efficiency): Can *Gandalf* generate diverse test inputs efficiently?** In RQ4, we evaluate whether *Gandalf* can provide a significant enhancement in diversity while ensuring efficiency, compared with existing approaches. Our experimental results show that *Gandalf* can significantly enhance the diversity by a maximum of 80% and achieve higher efficiency (up to five times compared with other approaches).
- **RQ5 (bug categories): What kinds of bugs are found by *Gandalf*? How does *Gandalf* perform in dealing with the false positives?** In RQ5, we manually analyze the categories of the bugs detected by *Gandalf* and evaluate whether *Gandalf* could reduce false positives.

According to the results, *Gandalf* can detect bugs varied in five categories, while the majority are precision problems. *Gandalf* can also avoid all the false positives caused by implementation differences.

To summarize, this article makes the following contributions:

- We conduct an empirical study on four state-of-the-art approaches for testing DL libraries. We find that existing approaches suffer from the potential of ineffectiveness in certain task domains, lacking of diversity and usually triggering a lot of false positives.
- We propose the first guided differential fuzzing approach based on generation for deep learning libraries with the context-free grammar and deep Q-network to address the challenge in the existing approaches.
- We design 15 practical cross-libraries equivalent metamorphic relations for differential testing of deep learning libraries to reduce the false positive rates.
- We implement our approach as an open-source tool, *Gandalf*, and release our data to facilitate further development of deep learning libraries.
- We conduct an extensive evaluation on *Gandalf* with three representative DL libraries and models from CV, NLP, and ASR domains. The evaluation results demonstrate the effectiveness and efficiency of *Gandalf* when testing DL libraries by generating diverse test inputs and reducing the false positives significantly.

2 BACKGROUND

2.1 Differential Fuzzing for DL Libraries

Differential fuzzing is the technique for testing DL libraries by taking the advances of both fuzzing and differential testing. Fuzzing refers to an iterative process of testing the target DL libraries with test inputs (i.e., DL models) produced elaborately [38]. Differential testing aims to provide oracles for inputs by cross-checking the consistency of the corresponding outputs from different DL libraries [23].

Figure 1 shows a typical workflow of the differential fuzzing for testing DL libraries. The challenges of differential fuzzing for testing DL libraries mainly lie in generating **test inputs** and analyzing **test outputs** [61]. The approaches for producing test inputs include generation-based and mutation-based approaches [53]. The generation-based and mutation-based fuzzing are two common approaches to produce test inputs in fuzzing [38]. The generation-based fuzzing produces test inputs based on a given constraint that describes the expected inputs, such as a grammar precisely characterizing the input format or less precise constraints such as magic values identifying file types. The mutation-based fuzzing produces test inputs by mutating or modifying existing inputs, e.g., altering, deleting, or replacing a subset of the input data. Some of the existing work employs mutation-based approaches to mutate existing DL models with manually designed mutation rules, e.g., LEMON [58] and AUDEE [24]. For mutation-based fuzzing, a well-formed corpus of seed models is required. However, it is difficult and time-consuming to manually design seed models covering diverse combinations of operators in DL libraries. In fact, existing work only takes a limited number of models as a seed corpus; while, in this article, we propose a generation-based approach for DL library testing by producing diverse test inputs following *context-free grammar*.

As each DL library under testing usually involves a self-defined input format, a unique challenge for differential testing of DL libraries is how to ensure the models deployed to each library are identical. Different from existing work relying on existing model converters, we propose a series of *equivalent metamorphic relations* to ensure identical models (seeds) are deployed when testing different DL libraries.

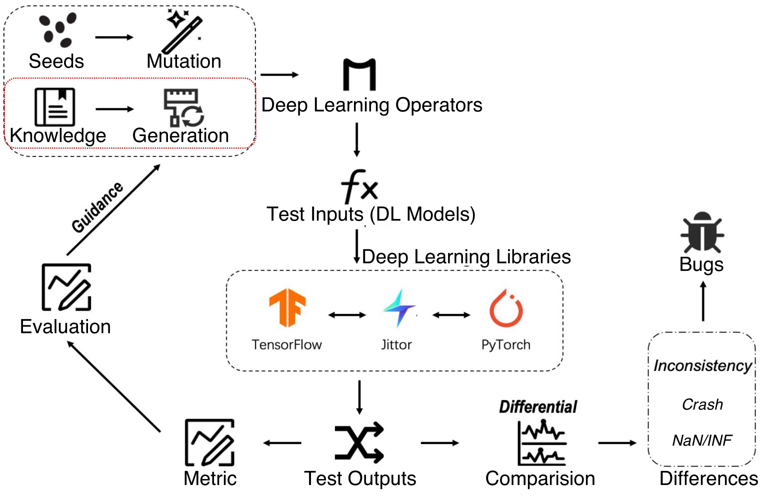


Fig. 1. Differential fuzzing for testing DL libraries.

2.2 Equivalent Metamorphic Relations

The **metamorphic relation (MR)** is an important technique in **metamorphic testing (MT)** [11]. MRs specify how particular changes to the input of the software under test would change the output [64]. Generally speaking, an MR for a function f is expressed as a relation among a series of inputs $x_1, x_2, \ldots, x_n$, where $n > 1$, and their corresponding outputs $f(x_1), f(x_2), \ldots, f(x_n)$ [49]. For example, a mathematical property of $sin(\cdot)$, i.e., $sin(x) \equiv sin(\pi - x)$ could be used to design an MR. When it is difficult to determine the expected output of $sin(1)$ when testing the function $sin(\cdot)$, the mathematical property of $sin(\cdot)$MR can help test $sin(\cdot)$ by checking whether $sin(1)$ equals $sin(\pi - 1)$.

The equivalent MRs are widely adopted in software testing, e.g., compiler testing [16, 33]. In compiler testing, given certain test inputs (i.e., programs), it is difficult to determine the expected outputs without manual efforts [9]. To alleviate such a problem, researchers propose a set of equivalent transformation rules and apply them to generate equivalent test inputs [44]. Given original test inputs I_1 and its equivalent inputs I_2, the compiler C under test should produce the same outputs $C(I_1)$ and $C(I_2)$, i.e., $I_1 \equiv I_2 \Rightarrow C(I_1) \equiv C(I_2)$ [54]. Inspired by the success of equivalent MRs in compiler testing, in this article, we employ equivalent MRs to create equivalent cross-library test inputs, i.e., equivalent DL models.

3 EMPIRICAL STUDY

3.1 Empirical Study Setup

To identify the current situation and challenges in differential fuzzing for testing DL libraries, we conduct an empirical study to understand how the differential and fuzzing techniques have contributed to the existing approaches. Our study is based on the results from using existing approaches [8], i.e., CRADLE [48], LEMON [58], AUDEE [24], and MUFFIN[22] in the same time period (i.e., one hour, as mentioned in their documents). These baselines cover four different implementations frequently used in DL library testing. Specifically, CRADLE employs 16 seed models as inputs. LEMON designs a heuristic-based fuzzing part with mutated inputs from 12 seed models. Both CRADLE and LEMON employ a differential part with front-end Keras to switch back-end libraries. AUDEE designs a search-based fuzzing part with 7 seed models and a

Table 1. The Basic Information of the Seeds and Datasets

Domain	Dataset	Size	DNNs	No.
CV	CIFAR-10 [32]	60,000 images	AlexNet	#1
	Fashion-MNIST [60]	70,000 images	LeNet5	#2
	MNIST [34]	60,000 images	LeNet5	#3
	ImageNet [14]	1,500 images	ResNet50	#4
			DenseNet121	#5
			MobileNetV1	#6
			InceptionV3	#7
			VGG16	#8
			VGG19	#9
			Xception	#10
NLP	Imdb [37]	50,000 reviews	LSTM	#11
	Reuters	11,228 reviews	LSTM	#12
ASR	Common Voice3	12 hours of audio clips	wav2vec2.0	#13
			ContextNet	#14

differential part with model converter MMdnn to transfer mutated inputs to multiple DL libraries. MUFFIN summarizes two model structure templates from neural architecture search [18] to fuzz the test inputs. After analyzing the four baselines we selected, we find that CRADLE and LEMON can be applied to TensorFlow, Theano, CNTK, and MXNet. AUDEE can be applied to TensorFlow, Theano, CNTK, and PyTorch. MUFFIN can be applied to TensorFlow and Theano. As a result, we select all DL libraries that could be tested by at least three approaches, including TensorFlow 1.14.0 [2], CNTK 2.7.0 [50], Theano 1.0.4 [3], and MXNet 1.5.1 [10], with the same settings in References [24, 48, 58]. To have a comprehensive understanding of the effects of different seed models in the fuzzing part of the existing approaches, we select 14 seed models, with a total of 120 operators, from different domains including **computer vision (CV), natural language processing (NLP)**, and **automated speech recognition (ASR)**, covering 8 datasets. Speech, text and image data are the common data in the mentioned domains and could cover a large number of DL tasks including image classification, object detection, text classification, machine translation, speech recognition, and so on. As a result, we select eight datasets in our experiments to cover the above-mentioned data domains. Table 1 gives the basic information of the seeds and datasets, including (1) the domain information provided by developers, (2) the dataset name, (3) the size of the dataset, and (4) the seed name. All these seed models are non-trivial. These datasets contain thousands to hundreds of thousands of commonly used data.

We investigate the following two research questions in this empirical study.

RQ1: How does fuzzing affect the DL library testing? How do the seeds of fuzzing affect the existing testing approaches in different domains? To answer RQ1, we first need to find models that cover a representative set of operators as seeds, as it is difficult to exhaust the set of operators that are constantly updated. To achieve this, we study how frequently the different operators are used in the open-source community (by 2022.08). We select operators with high frequency as representative operators and use models that cover these operators as seeds. We further conduct a study on four state-of-the-art approaches for testing DL libraries, i.e., CRADLE [48] (a random differential fuzzing technique), LEMON [58] (a guided differential fuzzing technique based on mutation), AUDEE [24] (a guided differential fuzzing technique based on mutation), and

MUFFIN [22] (a random differential fuzzing technique based on neural architecture search) with these seeds. The four methods are selected based on different types of fuzzing techniques. We analyze the number of effective test inputs and unique bugs found by these approaches. However, the mutated test inputs may be duplicated and in-diversified (e.g., inputs mutated from the same seed with the same mutation parameters). The lack of test input diversity in DL library testing could lead to the insufficient exploration of input space and low effectiveness in detecting bugs. To measure the diversity, we define the diversity of the DL library test inputs, i.e., DL models. Considering that DL models are implemented with the structure of graphs, the diversity of DL models is measured by the graph edit distance. The definition of diversity is shown as follows:

Definition 1 (Diversity of DL Models). Given a set of DL models $M = M_0, M_1, \ldots$, the diversity of M could be defined as the average extent to which all the models $M_0, M_1, \ldots$ in M differ from each other, i.e.,

$$diversity(M) = avg(|M_0 - M_1|, |M_0 - M_2|, \ldots), \tag{1}$$

where $|M_0 - M_1|$ denotes the minimum number of operator modifications required to complete the interconversion between model $M_0 = (O_{00}, O_{01}, \ldots, O_{0i})$ and model $M_1 = (O_{10}, O_{11}, \ldots, O_{1j})$, which could be measured as follows:

$$|M_0 - M_1| = modOp(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j) = 0 \\ \min \begin{cases} modOp(i-1,j) + 1 \\ modOp(i,j-1) + 1 \\ modOp(i-1,j-1) + 1_{(O_{0i} \neq O_{1j})} \end{cases} & \text{if } \min(i,j) \neq 0 \end{cases}. \tag{2}$$

RQ2: What kind of bugs can be found by the existing approaches? Do the existing approaches trigger any false positives? In RQ2, we aim to study the bug categories detected by existing approaches and investigate if there are false positives of bugs. To achieve this, we manually check whether a bug is a false positive with five professional engineers (three for coding checking and two for algorithm checking). Specifically, we adopt different analysis methods depending on the bug symptoms. For symptoms with crashes, we leverage the error message to analyze the causes. For symptoms with **NaN/INF (numerical errors)** and **IC (inconsistent outputs)**, we trace the output of libraries on each computation to analyze the causes.

3.2 Empirical Results for Answering RQ1

Table 2 shows the empirical results of **CRADLE (C), LEMON (L), AUDEE (A)**, and **MUFFIN (M)** on four DL libraries. For each approach, we calculate the number of effective test **inputs (I)** and unique **bugs (B)** found during the study. Considering the different effective test inputs may trigger the same bug, chances are that the number of unique bugs is smaller than that of effective test inputs. We further calculate the **mean edit distance (mED)** of each approach as an indicator of diversity. Most of the mEDs are 0, referring to the duplicated test inputs.

From the perspective of the fuzzing DL libraries, we have several interesting findings:

- **Finding 1.1. The testing effectiveness (i.e., the number of detected bugs) is significantly affected by seeds from different task domains.** For example, CRADLE fails to detect any bugs of DL libraries in the NLP domain and MUFFIN cannot work in the ASR domain and MXNet. Even within the same domain, existing approaches perform differently with different seeds, e.g., AUDEE fails to detect bugs with mutation seed {#11, #12} while succeeds with seed #5. Therefore, it is necessary to involve seeds from different task domains when generating test inputs.
- **Finding 1.2. The inputs that trigger bugs lack diversity.** Since most of the test inputs of existing approaches are generated from given seeds by mutations or templates, the

Table 2. The Number of Effective Inputs (I) and Unique Bugs (B) on Each Library (CRADLE [48], LEMON [58], AUDEE [24]), and MUFFIN [22])

No.	TensorFlow								CNTK								MXNet								Theano								Edit Distance			
	C		L		A		M		C		L		A		M		C		L		A		M		C		L		A		M		C	L	A	M
	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B	I	B				
#1	3	2	38	16	1	0	0	0	1	0	12	1	1	0	1	0	1	1	16	2	1	0	-	-	13	0	12	1	1	0	5	3	0	1.5	2.25	2.6
#2	0	0	8	4	0	0	1	0	2	2	0	0	2	1	1	0	0	0	8	5	0	0	-	-	0	0	8	5	0	0	4	2	0	1	1.5	3
#3	0	0	7	0	0	0	0	0	2	2	7	0	2	1	0	0	0	0	7	0	0	0	-	-	0	0	7	0	0	0	7	2	0	0.8	1.1	2.3
#4	25	4	0	0	21	2			1	0	0	0	1	0			23	2	21	7	25	2			1	0	0	0	1	0			0	0.04	0.07	
#5	24	3	1	0	21	2			1	0	1	0	1	0			9	2	1	0	5	2			5	1	1	0	6	2			0	0	1	
#6	1	0	0	0	1	0			1	0	0	0	1	0			1	0	39	13	1	0			1	0	0	0	1	0			0	0.13	0.5	
#7	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	-	-	1	0	1	0	1	0	2	2	0	0	0	2.4
#8	6	2	10	0	5	1			1	0	10	0	1	0			5	1	10	0	4	1			4	2	10	0	4	0			0	0	0	
#9	0	0	5	0	0	0			1	0	5	0	1	0			1	0	5	0	1	0			0	0	5	0	0	0			0	0	0	
#10	0	0	10	3	0	0			1	0	1	0	1	0			1	0	1	0	1	0			0	0	1	0	0	0			0	0.1	1.5	
#11	0	0	20	12	0	0	0	0	1	0	21	12	1	0	2	0	0	0	29	12	0	0	-	-	0	0	0	0	0	0	5	3	0	2.2	3.3	2
#12	0	0	25	21	0	0	0	0	1	0	23	17	1	0	1	0	0	0	34	25	0	0	-	-	0	0	0	0	0	0	6	4	0	4	2.1	2.9
#13	1	1	3	2	0	0	-	-	1	0	2	1	1	0	-	-	0	0	2	1	0	0	-	-	0	0	0	0	0	0	-	-	0	1.1	1.65	-
#14	2	1	1	1	1	0	-	-	1	0	1	1	1	0	-	-	0	1	3	2	1	0	-	-	2	0	0	0	0	1	-	-	0	0.9	1.35	-

inputs are highly similar to each other (some of them are even duplicated). Specifically, CRADLE, an early work in this area, merely employs existing model architectures as test inputs without implementing any structural modifications. AUDEE and LEMON move one step further, introducing self-defined mutation rules to generate mutated test inputs based on existing model structures. However, these adaptations remain constrained. Some rules only introduce Gaussian noise into the model weight without any structural modifications. Other rules selectively modify specific layers while leaving the majority of the model unchanged. In summary, all three of these methodologies heavily depend on the structure of the initial seed models, leading to a limited diversity in the generated test inputs. However, as illustrated in Table 2, the number of bugs detected is highly correlated to the diversity of the test inputs. This suggests a potential limitation of existing approaches.

- **Finding 1.3. There is room for improvement in the seed selection and test input generation of existing DL library fuzzing.** Seed selection and test input generation constitute critical procedures in related DL library fuzzing techniques. The test effectiveness of existing approaches is limited because of their inherent flaws in these two stages. By incorporating domain selection on existing seed models and explicitly adding diversity when generating new models, both stages can be improved, leading to the final performance enhancement.

3.3 Empirical Results for Answering RQ2

After the manual analysis of the bugs detected in RQ1, we group these bugs into four categories: (1) **implementation bugs (IBs)**, (2) **implementation differences (IDs)**, (3) **precision bugs (Ps)**, and (4) **random errors (REs)**. Table 3 shows the number of unique bugs detected by existing approaches in different categories. IBs will be triggered if there exist bugs in the implementation of DL libraries. IDs are raised when different DL libraries implement certain operators in different ways, e.g., the *padding* operator is implemented differently in TensorFlow and Theano. The IDs consist of **initialization differences (INIT), data format differences (DAF)**, and **operator setting differences (SET)**. Ps are caused by the floating-point precision in the DL libraries. REs are raised by the inevitable random error caused by operators such as *Dropout*. Among the four categories, IDs and REs are considered as false positives.

Table 3. The Number of Unique Bugs in Different Categories

Approach	REs	IDs			Ps	IBs	Total	False Positives
		INIT	DAF	SET				
CRADLE	2	13	3	5	3	1	27	23
LEMON	0	0	2	1	10	0	13	3
AUDEE	0	1	0	1	8	0	10	2
MUFFIN	0	4	3	3	13	0	23	10

From the perspective of the differential testing for DL libraries, we obtain the following interesting finding:

- **Finding 2.1. The existence of implementation differences leads to a large number of false positives by existing testing approaches.** Even MUFFIN, the state-of-the-art differential fuzzing approach for DL libraries, suffers from a remarkably high number of false positives, with a false positive rate of up to 58%. The root cause of these false positives lies in how the existing approaches adopt differential testing. Differential testing necessitates consistency in test inputs across various libraries. Unfortunately, existing methodologies do not provide this assurance. Even though the test input models utilized share identical logical structures across different libraries, divergent implementations adopted by these libraries could lead to inconsistencies at the front end. For example, during the initialization of a new model, DL libraries will randomly assign parameters. This randomness cannot be preserved uniformly across all libraries, potentially leading to inconsistencies, which can be misinterpreted as bugs by existing testing approaches.

  To reduce the false positives that suffer from these implementation differences, we can consider strategies such as initialization transformations, data format transformations, and setting transformations for specific operators.
- **Finding 2.2. The ability of the existing approaches to find IBs is limited.** The DL libraries that could be tested by the existing approaches are restricted by their final test inputs for target DL libraries. In specific, CRADLE, LEMON, and MUFFIN rely on Keras to run final test inputs on other back-end libraries, and AUDEE relies on MMdnn to generate final test inputs. As a result, the DL libraries under testing are of high similarity, i.e., the development paradigm of static graph. The implementations of these libraries tend to be same, even if there are bugs in the implementation. To tackle this problem, DL libraries with different development paradigms should be taken into consideration.

4 APPROACH

In this section, we introduce *Gandalf*, a novel generation-based differential fuzzing technique for DL libraries. The generation of test inputs can be considered as two steps. The first step generates combinations of operators in the graph format, which can serve as the primary test inputs. The second step generates final test inputs for target DL libraries by transferring the intermediate graphs into the corresponding library-specific format following the API documentation. These two steps resemble the inverse process of the typical DL libraries' computation, where the front-end first translates the library-specific models into intermediate representations that share a similar structure (graph), and the back-end hardware will further execute operator graphs. A good design of a DL library should make these two steps independent of each other [8]. Our design is inspired by such observation, and our workflow is shown in Figure 2. At a high level, we first generate operator graphs as primary test inputs based on a grammar-based diversity-guided strategy (Section 4.1). The **context-free grammar (CFG)** in our design ensures the generated

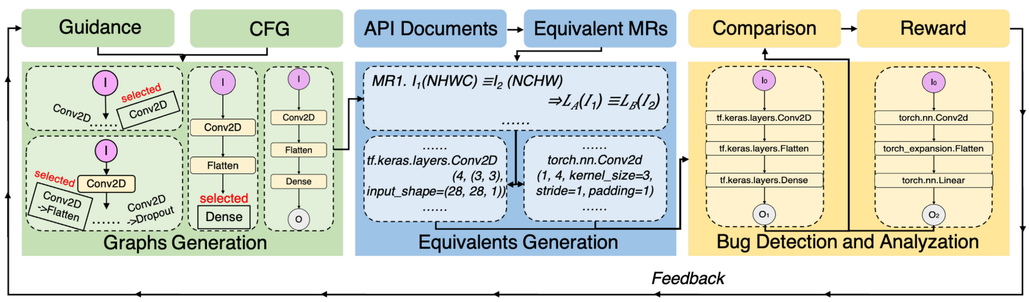


Fig. 2. The workflow of *Gandalf*.

DL models are valid. The generation process is optimized by a **Deep Q-Network (DQN)**. One of the optimization goals is diversity (Equation (3)), which aims to find more diverse bugs in the DL libraries. The second part generates semantically equivalent inputs for different DL libraries with equivalent **Metamorphic Relations (MRs)** (Section 4.2) to promote differential testing. These MRs can reduce false positives. After collecting the generated inputs, we execute them and cross-check the outputs of DL libraries (Section 4.3). We further analyze the testing results and use them as rewards for the DQN to improve our testing efficiency.

4.1 Graphs Generation

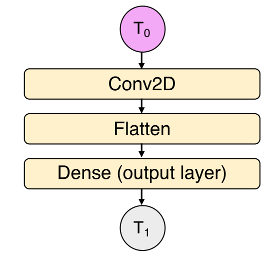
Based on our empirical study, we observe the problem that it is the selected seeds that limit the diversity of test inputs produced by existing approaches. Compared with using models with established structures as seeds, using seeds with a more flexible low-level structure can effectively reduce the limits of diversity. Thus, in this article, we employ the smallest units that constitute the models, i.e., operators, as seeds. By combining operators, we could generate operator graphs for test inputs. To generate valid operator graphs for test inputs, we propose a generation procedure based on the CFG of DL models (Section 4.1.1). Further, we employ the DQN as a guide, aiming to explore the bug-exposing operator graphs (Section 4.1.2).

4.1.1 CFG of DL Models. Considering the legitimate operator graphs are combined following specific constraints, randomly generated graphs can be potentially invalid and may incur a lot of false positives. In this article, we employ CFG of DL models to generate valid operator graphs. The CFG of DL models defines how the DL operators are combined. CFG can ensure the effectiveness of DL models as test input and is applicable across a wide range of task domains. However, since the building blocks of DL models are operators rather than simple codes, the CFG of DL models should be defined in the form of operators. In this article, we define the CFG of a DL model. Following the practical CFG [4], the CFG of DL models can be defined as Definition 2.

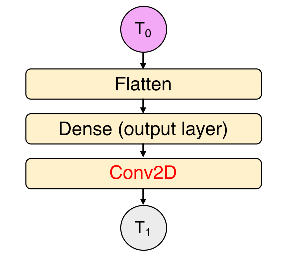
Definition 2 (CFG of DL Models). Given a set of operators $O = \{O_0, O_1, \ldots\}$ and a set of **partially ordered relations (PRs)** $R = \{O_0 \rightarrow O_1, O_0 \rightarrow O_2, \ldots\}$ of O, the CFG of DL models $C = (V, \Sigma, R, S)$ defines the valid procedure to combine DL operators as follows:

- select a start symbol $s \in S$, where S is a set of input operators $O_S \subset O$;
- recursively select a non-terminal symbol $v \in V$, where V is a set of non-terminal operators $O_V \subset O$ and a partially ordered relation $\{v_0 \rightarrow v\} \in R$ exists between v and the predecessor v_0 of v;
- select a terminal symbol $\sigma \in \Sigma$, where Σ is a set of output operators $O_\Sigma \subset O$;
- generate $\alpha \leftarrow s(\bigoplus v)^{\bigotimes} \bigoplus \sigma$,

where $\bigoplus$ denotes injections of operators and $\bigotimes$ denotes repetitions of operators.



(a) The input obeying CFG.



(b) The input violating CFG.

Fig. 3. Examples of operator graphs.

Specifically, the CFG defines the way and order in which different types of operators are combined into DL models, while the PRs constrain the specific sequences that can exist between each operator. The set of operators (e.g., *Conv2D*, *Batch_norm*) is collected from DL libraries and categorized into three sets (V input, O non-terminal, and Σ output) by their functionality defined in API documentations. To achieve valid PRs between DL operators, we traverse PRs in the example projects provided by DL libraries and open-source projects in GitHub and collect PRs verified by these projects. With the PRs of operators, we can derive valid injections and generate graphs of operators.

Figure 3 shows the operator graphs of two test inputs, with tensor data I_0 and I_1 as running examples for the functionality of CFG. The valid input in Figure 3(a) can be executed by DL libraries while the invalid input in Figure 3(b) with wrong graphs of operators can be rejected by DL libraries with compiling errors reported. Since an output operator, i.e., Dense, must be used as the final operator of the test inputs. If the generation procedure follows the CFG of DL models, then the invalid combination in Figure 3(b) could be avoided. Because there is neither a PR $Dense \rightarrow Conv2d$ nor $Conv2d$ belongs to the set of output operators.

4.1.2 Bug-exposing Guidance. In this paper, we select a DQN as the bug-exposing guidance of Gandalf. According to Definition 2, for each selected operator in the generation procedure, bunches of operators as its successor satisfying PRs can be selected. For example, there are a lot of PRs for $Conv2d$ to be selected, including $Conv2d \rightarrow Dropout$, $Conv2d \rightarrow ReLU$, and $Conv2d \rightarrow AveragePooling2D$. Since a lot of valid PRs can be chosen for each operator in the generation process, as is shown in Figure 5(a), there can be an exponential number of possible graphs in the search space. However, different operator graphs may have different bug detection capabilities. It remains a challenge how to generate bug-exposing graphs. In other words, we should select PRs that can help generate operator graphs toward the direction of amplifying the probability of exposing bugs. To tackle this challenge, *Gandalf* adopts **reinforcement learning (RL)** to assist the selection of PRs. With the RL technique, Gandalf could learn for exposing bugs based on selected and generated operator combinations, so the bug-exposing graphs in the huge search space can be found. The exponential amounts of valid PRs may lead to the redundancy of RL techniques. We prefer DQN to other RL techniques. In this article, we propose a DQN-guided generation approach to guide the selection of bug-exposing PRs, aiming at generating bug-exposing operator graphs.

We now briefly introduce the problem setting in reinforcement learning language, where *state*, *action*, and *reward* play an important role. As is shown in Figure 5(b), for each generation of a combination, the DQN guides the selection of each PR recursively. For each selection, the DQN takes the current operator as a *state* and the required PR as an *action* from current operator state

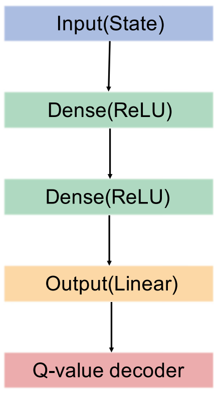
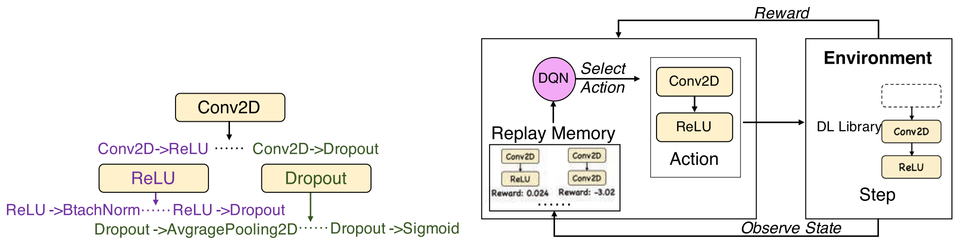


Fig. 4. The architecture of DQN in *Gandalf*.



(a) Generation without guidance.

(b) Generation with guidance.

Fig. 5. Generating graphs of DL operators.

to a new operator *state*. After each selection, the environment of DQN gives a *reward* (calculated by Equation (3)) for the latest selection, together with the new operator state as an *observation*. According to each *observation*, the DQN of Gandalf decides which *action* would be conducted next, i.e., which PR would be selected and added to the generated DL model. In this article, the DQN of Gandalf employs a Q-Network to decide the next *action* based on the *observation*. The Q-Network takes the *observation* as an input argument. Gandalf employs a **multi-layer perceptron (MLP)** structure with several densely connected layers as the Q-Network in DQN. The architecture of the Q-Network in Gandalf's DQN is shown in Figure 4, including two hidden layers with ReLU activation functions and an output layer. After the output layer, Gandalf uses a Q-value decoder layer to translate the output feature vector into a Q-value corresponding to each possible *action*. The Q-value indicates the estimated *reward* for each *action* to be selected. The *action* with the biggest value of *reward* indicates the highest possibility to trigger DL library bugs and would be selected by Gandalf.

To guide the generation with a high probability of triggering bugs, the DQN saves the recent selection and reward in a **replay memory (RPM)** and updates itself automatically. With DQN guiding the generation of models towards the direction of amplifying reward, *Gandalf* could generate bug-exposing test inputs. The higher reward is expected to obtain a higher probability of exposing bugs.

To further make the DQN guidance suitable for testing DL libraries, we employ two tricks in our approach, i.e., the top-k selection trick (*Trick* 1) and the no trap trick (*Trick* 2). *Trick* 1 selects PRs from the *k* PRs with the highest rewards at a certain probability. Without *Trick* 1, the guidance tends to always choose the PR with the highest known rewards, which may result in local

ALGORITHM 1: DQN-Guided Generation Using CFG

```
Input: $O$ : $Operators$; $L$ : $DL\ Library$; $n$ : $Target\ number$
Output: $C$ : $Generated\ graphs$
1:  while $Size(C) < n$ do
2:      $C \leftarrow C \cup RunEpisode()$
3:  end while
4:  return $C$
5:  function RunEpisode
6:      $c, obs, s \leftarrow reset(O.inOp), s \leftarrow random()$
7:      while $Size(c) < s$ do
8:          $action \leftarrow DQN.selectAction(obs)$
9:          $c.append(O.[indexOf(action.next))$
10:         $reward \leftarrow Reward(action, c)$
11:         $rpm.update(obs, action, reward)$
12:         $//\ Trick\ 2.\ No\ trap.$
13:         if $trapDetected()$ then
14:             $DQN.update(rpm.update(\gamma))$
15:         end if
16:         $obs \leftarrow DQN.nextObs()$
17:     end while
18:     return $c.append(o.ouOp)$
19: end function
20: function selectAction(obs)
21:     $dice \leftarrow random.uniform(0, 1)$
22:     if $dice < \epsilon$ then
23:         $action \leftarrow random(obs)$
24:     else
25:         $dice \leftarrow random.uniform(0, 1)$
26:         if $dice < 0.5$ then
27:             $action \leftarrow best(obs)$
28:         else
29:             $//\ Trick\ 1.\ Top - k\ selection.$
30:             $actions \leftarrow chooseTopKAction(obs, k)$
31:             $action \leftarrow RandomActionFrom(actions)$
32:         end if
33:     end if
34:     return $action$
35: end function
```

optimum. This can partially alleviate the exploration-exploitation dilemma. By using $Trick$ 1, we fully leverage all PRs with the potential for higher rewards. During the generation, the trap refers to the repeated generation of specific graphs. If a trap occurs, then $Trick$ 2 will update the DQN by adding a large negative reward to the RPM. Without $Trick$ 2, the guidance tends to satisfy the local optimum and repeatedly generates known graphs of high rewards. By using $Trick$ 2, our approach can potentially explore bug-exposing graphs in all candidate graphs.

4.1.3 Overall Algorithm. We formally describe our DQN-guided generation using CFG in Algorithm 1. The inputs include a set of operators O provided by DL libraries L under test and the target number n of generated operator graphs. The output is a set of generated graphs C for test inputs with operators in O. In this algorithm, n valid graphs for the differential fuzzing are generated (*Lines* $1-3$). During the generation of each combination c (*Line* 2), c and observation ob are reset with the **input operator** in O and the number of non-terminal operators is randomly generated (*Line* 6). During the selection of each **non-terminal operator**, the DQN selects a PR *action* as the next action according to ob (*Line* 8). To encourage the balance between exploration, i.e., generating different graphs, and exploitation, i.e., generating bug-exposing graphs, we employ *Epsilon-Greedy* [43] policy to select *action* (*Lines* $20-34$). During the selection, there is a probability of ϵ to randomly select (*Lines* $21-24$), and a probability of $\frac{1-\epsilon}{2}$ to select most bug-exposing *action* (*Lines* $25-27$), and $1-\frac{\epsilon}{2}$ to select with $Trick$ 1 (*Line* 28). After the selection, the new operator in *action* is appended into c (*Line* 9). The reward for the selection is calculated by Equation (3) (*Line* 10). The replay memory is updated by current information (*Line* 11). If a trap is detected in the current c (*Line* 13), then we add a large negative reward γ into the RPM and update the DQN (*Line* 14) according to $Trick$ 2. The DQN observes the current operator state obs (*Line* 16) and gets ready for the new iteration of selecting a non-terminal operator until the c is generated with enough operators (*Lines* $7-17$). *Line* 18 appends the **output operator** for c and returns an operator combination. These two tricks in the algorithm are the key to the testing effectiveness of *Gandalf*, and we will demonstrate corresponding experiment results in RQ4.

4.2 Equivalents Generation

Based on the empirical study, we find that to reduce false positives in DL library testing, it is necessary to ensure that when adopting differential testing in DL library testing, the test inputs executed by each library must be identical, including the structure and implementation of the models. To achieve this goal, we propose equivalents generation to transform the generated operator graphs into test inputs for different DL libraries, such that the generated test inputs for different DL libraries are identical. With our method, differential testing can be adopted to cross-check the consistency of the corresponding outputs of different DL libraries.

To achieve this goal, we design a series of equivalent MRs for generating equivalent test inputs. In this article, we refer the equivalent test inputs as the instances of DL models under different DL libraries, sharing the same operator graphs of specific DL operators.

When generating DL models out of graphs, we design a series of equivalent MRs to eliminate the effect brought about by the IDs of different DL libraries. According to the IDs of different DL libraries, we need to design different types of equivalent MRs separately. To identify the equivalent MRs, we adopt the symmetry **MR pattern (MRP)**, which refers to the differences between different DL libraries to implement DL models. The symmetry relations of DL libraries lie in the different implementations of the same operators. Following the symmetry MRP, we identify the equivalent MRs by investigating the implementation differences in each of the three phases correspondingly, i.e., initialization transformation, data format transformation, and settings transformation. We introduce them in detail in Table 4. In the table, I_1 and I_2 denote the test inputs for two DL libraries L_A and L_B, respectively, sharing the same operator combination. $L_A(I_1)$ and $L_B(I_2)$ denote the test outputs of I_1 and I_2.

Before the computations, the implementation differences of DL libraries arise in the random initializations, including the initialization of weights, kernel, and bias. $MR_1 - MR_3$ focus on the initialization transformation. For the operators in DL libraries, the randomness is mainly introduced by the random *initializer* within operators. To eliminate such randomness across different DL libraries, MR_1 loads the same weights for the inputs under different libraries. For the random *initializer* in *kernel* and *bias*, MR_2 and MR_3 load the same weights to *initializer* as the constant.

Table 4. Equivalent MRs for DL Libraries

ID	*Equivalent Metamorphic Relation*	
1	$I_1 \equiv I_2$ when initialized with same weights $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	**Initialization Transformation**
2	$I_1 \equiv I_2$ when the kernel are normalized with same constants $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	
3	$I_1 \equiv I_2$ when the bias are normalized with same constants $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	
4	$I_1 \equiv I_2$ with input shape as (N, H, W, C) and $(N, C, H, W) \Rightarrow L_A(I_1) \equiv L_B(I_2)$	**Data Format Transformation**
5	$I_1 \equiv I_2$ with input shape as (N, C, L), (N, L, C) and $(L, N, C) \Rightarrow L_A(I_1) \equiv L_B(I_2)$	
6	$I_1 \equiv I_2$ with input shape as $channel\_first$ and $channel\_last \Rightarrow L_A(I_1) \equiv L_B(I_2)$	
7	$I_1 \equiv I_2$ with $float$ and $float32 \Rightarrow L_A(I_1) \equiv L_B(I_2)$	
8	$I_1 \equiv I_2$ with int and $int32 \Rightarrow L_A(I_1) \equiv L_B(I_2)$	
9	$I_1 \equiv I_2$ when received multi inputs or automated patched inputs $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	**Settings Transformation**
10	$I_1 \equiv I_2$ when separable and depthwise set as $depth\_multiplier = 1$ or default $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	
11	$I_1 \equiv I_2$ when $padding =' SAME'$ or dealing with same padded tensor $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	
12	$I_1 \equiv I_2$ when received box param or box transformed from list param $\Rightarrow L_A(I_1) \equiv L_B(I_2)$	
13	$I_1 \equiv I_2$ when $I_1.momentum + I_2.momentum = 1 \Rightarrow L_A(I_1) \equiv L_B(I_2)$	
14	$I_1 \equiv I_2$ with $affine = True$ or default $affine \Rightarrow L_A(I_1) \equiv L_B(I_2)$	
15	$I_1 \equiv I_2$ with $track\_running\_stats = True$ or default $track\_running\_stats \Rightarrow L_A(I_1) \equiv L_B(I_2)$	

During the computations, the implementation differences of DL libraries are attributed to the storage order and format of the data. $MR_4 - MR_8$ focus on the data format transformation. For the operators without parameters for input shape, inputs with the shape of (N, H, W, C) are equivalent to that of (N, C, H, W) (MR_4). MR_5 takes the inputs with the shape of (N, C, L), (N, L, C), or (L, N, C) as equivalents. For the operators with parameters for channel dimension C in the input shape, the inputs declared as $channel\_first$ are equivalent to that of $channel\_last$ (MR_6). For the undeclared data bits in the operators, MR_7 and MR_8 treat them equivalent as 32 bits.

After the computations, the implementation differences of DL libraries lie in the default parameters of the operators. $MR_9 - MR_{15}$ focus on the setting transformation. MR_9 treats the multi inputs and automated patched inputs equivalent. For the operators of *depthwise* and *separable*, MR_{10} calculates the equivalent *depth_multiplier* for default. For the operators regarding *padding*, MR_{11} treats the inputs padded within the operator and before the operator equivalent. The inputs regulated with *box* list are equivalent to those with separated *box* (MR_{12}). The inputs with complementary *momentum* are equivalent (MR_{13}). For the input with default $affine$ and $track\_running\_stats$, MR_{14} and MR_{15} treat it equivalent to that of $True$.

4.3 Bugs Detection

In this part, we execute the generated inputs with DL libraries under test and detect bugs in DL libraries. We detect three kinds of bugs in DL libraries with different symptoms, i.e., *CRASH*, NaN/INF, and *inconsistency*. *CRASH* and NaN/INF could be detected by checking outputs directly, while *inconsistency* could be detected with differential testing. The failures or abortions of DL libraries lead to *CRASH*. NaN/INF consists the existence of *NaN* and *INF* in the test outputs. DL libraries produce *NaN* when an undefined or unpresentable value is calculated. When an infinite value is calculated, DL libraries produce *INF*. With differential testing, DL libraries produce *inconsistency* bugs when they yield inconsistent outputs for the same test inputs.

Following the three symptoms of bugs, we define a quantitative metric $Metric_D$ to measure the differences between DL libraries, as is shown in Equation (3). β, γ, and δ are considered as the exploration probability of the three bug symptoms. All bug symptoms are important in DL library testing. Therefore, it is important to take the detection of multiple bug symptoms into account when testing DL libraries. To this end, when designing the guidance and the rewards of Gandalf, we take all bug symptoms into account by setting exploration probability parameters. The values of β, γ, and δ could be revised according to the testing purpose. For example, if the *CRASH* bug symptoms are more preferred, then γ would achieve a higher value compared with β and δ. In this article, we value all bug symptoms equally, so we choose to take balanced values of β, γ, and δ. For the *Inconsistency*, $Metric_D$ is calculated by the distance between the outputs. I and I_c denote the differences in outputs and the confidence coefficient of outputs. For the *Crash* and *NaN/INF*, $Metric_D$ is calculated by the mean value m_d of the tensor data, such as that of MNIST. The results of differential testing could be calculated with $Metric_D$ by comparing the outputs of different DL libraries. Further, $Metric_D$ is considered as a reward for each generated test input to DQN.

$$Metric_D = \begin{cases} \beta * I * I_c & Inconsistency \\ \gamma * m_d & Crash \\ \delta * m_d & NaN/INF \end{cases} \tag{3}$$

5 EVALUATION

5.1 Research Questions

Previous sections reveal the limitations of previous work through answering RQ1–RQ2. Section 4 introduces *Gandalf* to overcome these limitations. In this section, we conduct a comprehensive study to evaluate *Gandalf*. We perform our evaluation along with the following three research questions.

RQ3 (effectiveness): How does *Gandalf* perform in detecting bugs of DL libraries? Does *Gandalf* reduce the effect of mutation seeds from different task domains? To overcome the limitations of the mutation-based approaches, *Gandalf* is designed as a generation-based approach without mutation. By getting rid of the burden of collecting mutation seeds in different domains, *Gandalf* can easily test DL libraries across different domains. In this RQ, we want to validate whether our design goal is met. We select eight widely used datasets for evaluation, as the same with RQ1 for a fair comparison. All of the operators studied in RQ1 and RQ2 are taken into consideration. The effectiveness of *Gandalf* is measured by the number of effective test inputs generated, i.e., the test inputs that trigger bugs in DL libraries.

RQ4 (diversity and efficiency): Can *Gandalf* generate diverse test inputs efficiently? As suggested by Finding 1.2 in RQ1, the generated test inputs of current testing frameworks suffer from the lack of diversity. In this article, the bug-exposing guidance of *Gandalf* is designed to generate diverse test inputs to trigger bugs effectively. In this RQ, we aim to investigate whether the guidance of *Gandalf* can indeed provide a significant improvement in the diversity of test inputs. However, optimizing the test generation process with diversity guidance can increase testing overhead, which may potentially influence the testing efficiency. As a result, diversity and efficiency need to be measured together. We measure the diversity of test inputs through *mean Edit Distance* and investigate the efficiency of *Gandalf* by counting how many effective test inputs are generated in the same period. We compare *Gandalf* with state-of-the-art approaches, i.e., random guidance in CRADLE [48] and MUFFIN [22], **Markov Chain Monte Carlo (MCMC)** guidance in LEMON [58], and **Genetic Algorithm (GA)** guidance in AUDEE [24] in the same time period (i.e., one hour). Meanwhile, to evaluate whether the DQN outperforms other RL techniques in assisting Gandalf, we compare the performance of Gandalf with a simpler RL technique, Q-Learning, and a more complex RL technique, DDQN, in assisting Gandalf to find bugs of three different symptoms. We repeat the experiments three times and apply the *t-test* to identify whether the improvement in diversity is significant or not.

RQ5 (bug categories): What categories of bugs are found by *Gandalf*? How does *Gandalf* perform in dealing with false positives? RQ5 aims to identify the bugs found by *Gandalf* to provide a more comprehensive understanding of the performance of *Gandalf*. We categorize the

Table 5. The Number of Effective Test Inputs Generated by *Gandalf*

	TensorFlow				PyTorch				Jittor			
	IC	N/I	C	T	IC	N/I	C	T	IC	N/I	C	T
MNIST	66	15	16	97	4	15	10	29	27	15	1	43
Fashion -MNIST	90	6	13	109	53	6	63	122	81	6	1	88
CIFAR-10	115	18	6	139	47	18	16	81	72	18	1	91
CIFAR-100	108	45	3	156	3	45	25	73	45	45	1	91
ImageNet	14	16	3	33	14	16	27	57	23	16	1	40
Imdb	23	4	1	28	3	4	3	10	9	4	1	14
Reuters	75	19	0	94	7	19	1	27	45	19	1	65
Common Voice3	126	198	0	324	6	198	12	216	45	198	1	244

bugs, which are confirmed by manual analysis after contacting developers and provide in-depth analysis of these bugs with different types. We also discuss the **false positives (FPs)** found by *Gandalf* and analyze to what extent *Gandalf* reduces the FPs.

5.2 Setup

To answer the above research questions, we select three libraries with different development paradigms as test subjects: (1) TensorFlow is known for static graphs [2], (2) PyTorch adopts dynamic graphs [47], (3) Jittor uses meta-operators to enable the fusion of static and dynamic graphs [27]. Nine versions of the three libraries are taken into consideration, i.e., 2.0.0, 2.4.0, 2.7.0 of TensorFlow, 1.8.1, 1.9.1, 1.10.1 of PyTorch, and 1.1, 1.2, 1.3 of Jittor. These three libraries cover all the popular technologies currently used in DL software development. Meanwhile, these libraries have been updated frequently in the past three years and have active developer communities. For the operators involved in testing, we collect open source projects as example projects from the trustworthy community (with at least 30 stars) to explore operators that have been frequently used in the development. According to how many times the different operators are used in the example projects, we rank the frequency of the DL operators, as shown on our website. Following the frequency, we select the operators, which are used more than 20 times in the example projects, with a frequency greater than 0.13%. The remaining operators are the ones that were almost developed a long time ago and are barely used at present. To comprehensively evaluate *Gandalf*, we further compare it with the state-of-the-art approaches in RQ1 and RQ2. For the comparative study, we evaluate all the approaches on the same experimental subjects to detect bugs in DL libraries with the same setup for fairness. To further evaluate the usefulness of *Gandalf*, we report the new bugs detected by *Gandalf* to the DL library developers for confirmation and feedback.

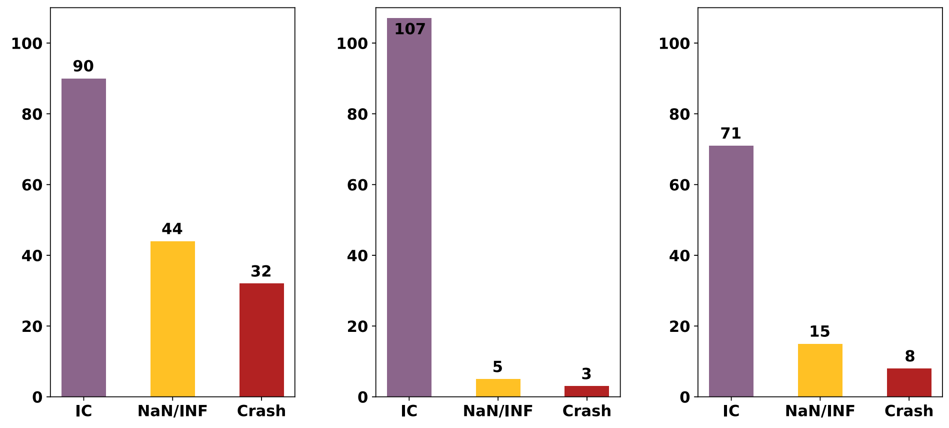
All experiments are conducted on a GNU/Linux system with Linux kernel 4.15.0 on one 20-core 2.5.0 GHz Intel Xeon Gold 6248 CPU with 64 GB RAM equipped with an NVIDIA Corporation GV100GL GPUs. The CUDA version is 10.0, Anaconda version is 4.10.3, and Python version is 3.6.

5.3 RQ3: Effectiveness

Table 5 summarizes the number of effective test inputs generated by *Gandalf* for the three DL libraries. The effective test inputs refer to the inputs that could trigger bugs. The total number of effective test inputs generated by Gandalf is listed in column "T." These effective test inputs can

Table 6. The Number of Unique Bugs in TensorFlow (TF), PyTorch (PTH), and Jittor (J)

	Existing			Fixed			New					
							Reported			Confirmed		
	CV	NLP	ASR	CV	NLP	ASR	CV	NLP	ASR	CV	NLP	ASR
TF	1	0	0	1	0	0	14	6	6	8	3	2
PTH	0	0	0	0	0	0	3	3	3	2	1	2
J	0	0	0	0	0	0	8	5	5	1	1	1



(a) *Gandalf*. (b) Without *Trick* 1. (c) Without *Trick* 2.

Fig. 6. Effectiveness of two tricks in *Gandalf*.

trigger bugs with three different symptoms, including **inconsistency bugs (IC), NaN/INF bugs (N/I)**, and **Crash bugs (C)**, corresponding to column "IC," column "N/I," and column "C." The result shows that for different datasets in different domains, *Gandalf* is able to generate effective test inputs. More specifically, we exclude the duplicated test inputs that share the same graphs of operators and parameters. Different from Finding 1.1 in RQ1, regardless of the domains, *Gandalf* can generate effective test inputs to test DL libraries.

To further evaluate the effectiveness of *Gandalf*, we check the number of unique bugs found by *Gandalf* for different domains. We categorize the triggered bugs in Table 5 and present the results in Table 6. In total, *Gandalf* found 50 unique bugs, with one bug previously fixed in the existing issues and 49 new bugs. These bugs cover three task domains, and we report all of the new bugs to DL library developers. Twenty-one of them have been confirmed by the developers and two have been fixed.

According to our evaluation, *Gandalf* is effective in testing DL libraries. According to our preliminary experiments, the effectiveness of *Gandalf* mainly comes from the two tricks detailed in Algorithm 1. To validate our observations, we conduct the evaluation of our approach on different policies, i.e., *Gandalf* in Algorithm 1, *Gandalf* without top-k selection trick (*Trick* 1), *Gandalf* with no trap trick (*Trick* 2) to demonstrate the effectiveness of the two tricks. Figure 6(a) demonstrates that the two tricks contribute to *Gandalf* in detecting bugs with all symptoms. They enable *Gandalf* to detect more *NaN/INF* and *Crash* bugs, as well as *inconsistency* (IC) bugs. In contrast, without them, it is difficult to detect bugs with *Crash* or *NaN/INF*, as is shown in Figures 6(b) and 6(c). This proves that these two tricks are the keys to the effectiveness of *Gandalf*.

Table 7. The Diversity of Generated Test Inputs and Efficiency of Each Approach

	Gandalf		*Gandalf$_r$*		*Gandalf$_{MCMC}$*		*Gandalf$_{GA}$*	
	mED	EIs	mED	EIs	mED	EIs	mED	EIs
CV	3.41	12	4.72	9	1.49	7	2.69	13
NLP	2.88	12	3.17	5	1.40	10	2.60	19
ASR	3.38	113	4.62	49	3.23	11	2.97	99

5.4 RQ4: Diversity and Efficiency

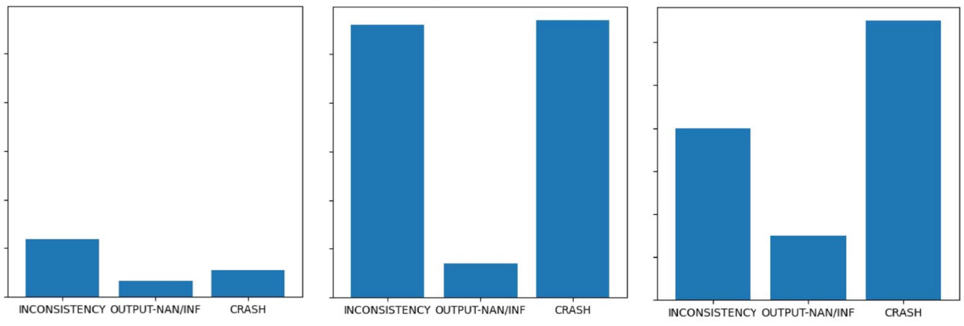
Table 7 depicts the diversity of effective test inputs generated by ***Gandalf***, *Gandalf$_r$* (variant of *Gandalf* with random guidance adopted by **CRADLE** and **MUFFIN**), *Gandalf$_{MCMC}$* (variant of *Gandalf* with MCMC guidance adopted by **LEMON**), and *Gandalf$_{GA}$* (variant of *Gandalf* with GA guidance adopted by **AUDEE**), calculated by ***mean edit distance*** **(mED)**. Among all of the approaches, *Gandalf* achieves a higher diversity, compared with the diversity of *Gandalf$_{MCMC}$* and *Gandalf$_{GA}$*. The diversity of *Gandalf* increased by 80% and 17% compared to *Gandalf$_{MCMC}$* and *Gandalf$_{GA}$*, respectively, based on the average results of the three domains. At the same time, *Gandalf* generates a large number of **effective inputs (EIs)**. According to the data in Table 7, a total of 137 bugs are found with the EIs generated by *Gandalf*, which is far more than that of *Gandalf$_r$* (63 bugs) and *Gandalf$_{MCMC}$* (28 bugs). It is demonstrated that the RL technique is superior to other simpler techniques such as MCMC in generating bug-exposing DL models. Meanwhile, the number of bugs found by the *Gandalf$_{GA}$* (131 bugs) is also lower than that found by *Gandalf*. Thus, the RL-based guidance of *Gandalf* is more effective than the heuristics adopted by existing approaches in the given time limit and achieves high efficiency. Based on the overall results on the three domains, *Gandalf* improves efficiency by up to five times and two times compared with *Gandalf$_{MCMC}$* and *Gandalf$_r$*, respectively. Although *Gandalf$_r$* performs well in terms of diversity, it is the least efficient. Thus, compared to *Gandalf$_r$* and *Gandalf$_{MCMC}$*, *Gandalf* could achieve high efficiency while ensuring diversity of generated test cases.

Overall, according to Table 7, *Gandalf* performs well in diversity compared with *Gandalf$_{GA}$*, while they share similar efficiency. We further conduct *t-test* between *Gandalf* and *Gandalf$_{GA}$* to identify whether *Gandalf* can significantly enhance the diversity of test inputs. In terms of diversity, the two-tailed P value of t-test results equals 0.0692, paired by domains. In other words, the enhancement of *Gandalf* on diversity could be considered significant compared with *Gandalf$_{GA}$*. Therefore, among the four approaches, *Gandalf* is most suitable for testing DL libraries, since it could provide a significant enhancement in diversity while ensuring efficiency.

Figure 7 illustrates the bug number of three different symptoms, i.e., outputs with *CRASH*, NaN/INF, and *inconsistency*, exposed in the same time period when using the three guidances. With Q-Learning as the guidance, Gandalf, the approach proposed in this article, shows poor performance in finding bugs in any of the symptoms. When Q-Learning is changed to DQN, Gandalf shows superior performance in detecting bugs for all symptoms. However, when taking DDQN as the guidance, although the number of detected bugs is close to that of DQN, additional resources are consumed due to multiple Q-Networks. Overall, Gandalf chooses DQN as the guidance to achieve a better performance and reduce the waste of resources compared with other RL techniques.

5.5 RQ5: Bug Categories

In addition to the number of bugs detected by *Gandalf*, we categorize them to investigate whether the proposed approach can find diverse bug types. We conduct a manual analysis of the bugs and summarize them into the following five typical categories:



(a) Guided by Q-learning. (b) Guided by DQN. (c) Guided by DDQN.

Fig. 7. The bug distribution guided by different RL techniques.

Precision (P) problems regarding floating-point numbers. (56.2%) Floating-point-related computations are crucial in many building blocks of DL models. Since a typical DNN model involves countless such computations, slight precision errors can accumulate and eventually lead to serious errors in the final output. For example in Figure 8(a), in a real reported bug, precision errors are accumulated in the computations of *LayerNorm2d*, leading to an inconsistency bug and localized in *Conv2d*.

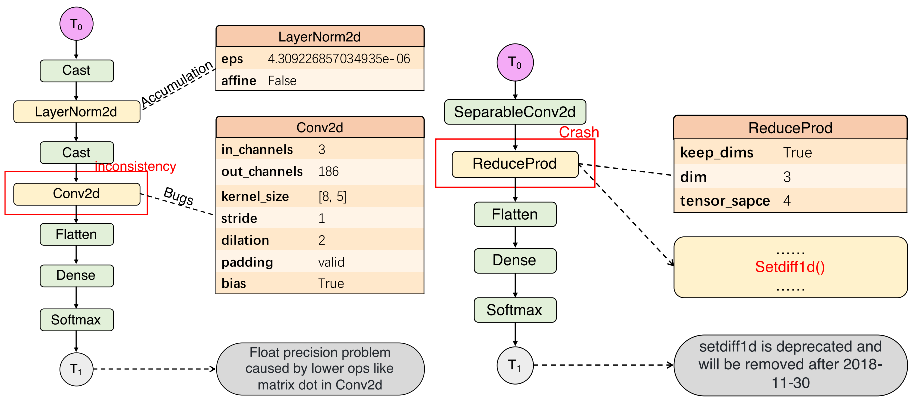
Wrong Reference (WR) regarding underlying APIs. (3.1%) In the implementation of DL libraries, operators may contain references to underlying APIs. These underlying APIs may be upgraded or discarded, but the references in these DL operators are not always updated in time, causing bugs in DL libraries. For example in Figure 8(b), we find that a bug is raised when the operator *reduce_prod* invokes an underlying API $setdiff1d$. However, $setdiff1d$ is already discarded and this case triggers a *Crash* bug of TensorFlow 2.7.0.

Partial-support for Input Space (PIS). (6.2%) Some operators in the DL libraries only support partial input space for certain data formats and do not provide constraints or exceptions for illegal data. These problems are typically detected in operations such as obtaining the inverse of a number, leading to NaN/INF problems. For example in Figure 9(a), with the operator *ReduceMax* repeatedly employed, the operator *Rsqrt* comes across tensors $\frac{1}{x}$ with unsupported shape and a NaN bug occurs.

Design or Mechanism Bug (D/MB). (9.4%) Some operators in the DL libraries may inherently have imperfect designs or mechanisms, which can lead to bugs. For example in Figure 9(b), during the evaluation, we find the design of the operator *Conv2dtranspose* is buggy. When employing *Conv2dtranspose* with $padding = SAME$, TensorFlow may automatically set the $pad = kernel\_size/2$ and $out\_padding = stride - 1$, leading to conflicts with $padding = SAME$ and resulting in an inconsistency bug.

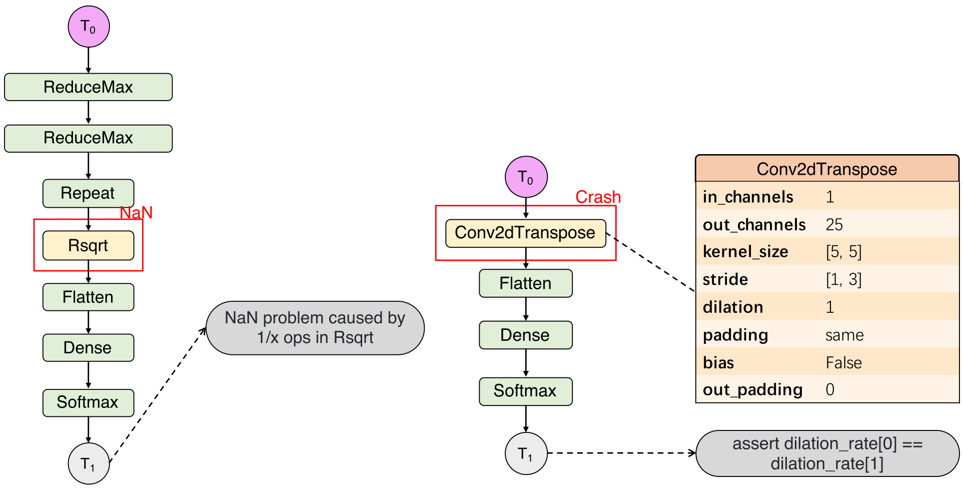
Wrong or Ambiguous Documentation (W/AD). (21.9%) As part of the DL libraries, the API documentation defines how operators are accessed by DL developers. However, there are inconsistencies between the documentation and the implementation of some operators, or the documentation does not specify the default constraints in the code, which has led to many issues. During our evaluation, for all the operators regarding *depthwise*, it is claimed in API documentation that all dimensions for *stride* are supported, which is actually not in the case (shown in Figure 10) with $stride = [2, 3]$.

During the evaluation, we also identify the false positives found by *Gandalf*, which are caused by the inevitable random errors in DL libraries; specifically, the IC bugs raised in operator *Dropout*, which randomly sets input units to 0 with a certain frequency to prevent *overfitting*. We further



(a) A detected bug of P. (b) A detected bug of WR .

Fig. 8. Typical bug categories (P and WR).



(a) A detected bug of PIS. (b) A detected bug of D/MB .

Fig. 9. Typical bug categories (PIS and D/MB).

Table 8. FP Rate of Different Approaches

Approach	*Gandalf*	CRADLE	AUDEE	LEMON	MUFFIN
FP Rate (%)	3.1	36.2	23	10	58.8

compare the rate of generating false positives (FP Rate) on *Gandalf* and the existing approaches. According to Table 8, *Gandalf* significantly reduces the false positives.

6 THREATS TO VALIDITY

In this article, the internal threats to validity come from the implementations of *Gandalf*. When Gandalf generates DL models, both valid model graphs and corresponding model implementations under different DL libraries have to be generated, which increases the challenge of model

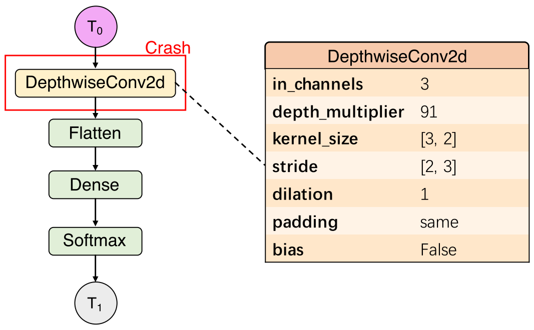


Fig. 10. A detected bug of W/AD.

generation and threatens the validity of this article. To reduce the threat, we decouple the combination generation and equivalent generation with each other. When *Gandalf* is extended with new operators or equivalent MRs, the generation of another one will not be affected. In addition, the implementation of Gandalf itself may also produce an internal validity threat. To reduce the threat, all artifacts regarding our approach, including codes, are carefully verified for weeks by five professional engineers. Among the five engineers, three of them work for coding review and two of them work for algorithm checking.

The external threats to validity come from the DL libraries employed in this article. The selection of DL libraries under test determines whether the findings of our empirical study and the evaluation of Gandalf can be generalized to other extensive circumstances. To further alleviate this threat, nine versions of TensorFlow, PyTorch, and Jittor are tested in this article, covering the popular libraries among DL communities.

The construct threats to validity come from the randomness of our evaluation. In DL libraries, there are a significant number of operators that employ randomness. These operators contribute to the performance of DL software, while introducing threats to testing DL libraries due to randomness. As a result, construct threats regarding randomness are inevitably brought about in our experiments. To reduce such a threat, we set the same time limit (i.e., one hour for each approach on each dataset), repeat each experiment three times, and take the average results. By repeating the experiments, the consequences caused by random error, noise, or other external factors can be minimized. The existing approaches repeat their experiments three times due to the threat of randomness. Therefore, this article repeats the experiments three times as well.

7 RELATED WORK

Fuzzing Technique. As one of the most widely deployed techniques for software testing [38], fuzzing produces a series of test inputs and checks whether the software under test violates a correctness policy [42]. According to how the test inputs are produced, the approaches for fuzzing are categorized into generation-based fuzzing and mutation-based fuzzing [53]. Previous generation-based work [31, 39] generates test inputs with specifications or knowledge, such as a grammar precisely characterizing the input format [15, 21]. Olsthoorn et al. combine the strength of grammar-based fuzzing and search-based generation to test the JSON parser libraries [45]. Gopinath et al. propose a general algorithm to derive the context-free approximation of input grammar from dynamic control flow [20]. In this article, we apply generation-based fuzzing to test DL libraries using well-exploited context-free grammar. This grammar guides how the operators are combined and forms the backbone of the testing inputs for the DL libraries.

Reinforcement learning in fuzzing. Bottinger et al. propose the first mathematical model to formalize fuzzing as an RL problem [6]. By defining states, actions, and rewards in the fuzzing context, they reduce the input fuzzing as a Markov decision process and generate new program inputs with high rewards as test inputs. Wang et al. propose a dynamic fuzzing approach, SyzVegas, which employs an RL scheme, i.e., **Multi-Armed Bandit (MAB)** problem, to identify the "most promising" strategy and seed, such as mutation count [56]. By abstracting the task/seed selection problem as an Adversarial MAB problem, SyzVegas could determine which type of task is the best, at each stage of fuzzing, and adapt its strategy accordingly. Su et al. propose a novel vulnerability-guided fuzzing approach, namely, RLF, for generating vulnerable test inputs to detect sophisticated vulnerabilities in smart contracts [51]. To handle the dynamic states of smart contracts during the fuzzing, RLF employs an RL framework and models the process of fuzzing smart contracts as a Markov decision process. Li et al. propose ALPHAPROG, a knowledge-guided RL-based approach to generating valid programs for compiler fuzzing [35]. Based on Q-learning, ALPHAPROG could generate a new program with best practices and provide a scalar reward for evaluating this synthesized program. Different from the existing work, since the complexity of testing DL libraries may lead to the exponential size of the input space, Gandalf employs DQN to guide the generation of the test inputs, i.e., DL models. To generate error-prone models, we define new metrics for reward and corresponding tricks for the DQN of Gandalf.

Differential Testing. Differential testing is an effective and acknowledged approach to deal with the test-oracle problem for complex software [40], such as compilers [52]. With more than one software implementation sharing the same functionality, differential testing compares the outputs from different implementations corresponding to the same input and considers the implementation that produces different output with other as a buggy one [9]. In differential testing, the same inputs that are expected as the outputs should be consistent theoretically [23]. When a large number of test inputs are employed, differential testing is promising in detecting semantic or logic bugs without manual efforts on labeling outputs [61]. In this article, we apply differential testing to a complex software suffering from the test-oracle problem, i.e., DL libraries. Since it is infeasible to generate identical inputs for different DL libraries because of their diverse software structures, we design MRs to generate semantically equivalent inputs so their outputs can be consistent.

Test input diversity. Input diversity has been investigated to support different aspects related to traditional software testing. Since executing similar test inputs tends to exercise similar parts of the software under test, it is likely to result in revealing duplicate bugs during testing. Therefore, testing with diverse test inputs should increase the exploration of the input space and thus increase bug detection rates [7, 13, 26]. Feldt et al. proposed TDSm, a diversity-based testing strategy [19]. TDSm employs the NCD metric to measure the diversity of test inputs. They demonstrate that diverse test inputs increase testing effectiveness and exhibit better bug detection capabilities. Hemmati et al. conducted an empirical study on similarity-based testing techniques for test inputs generated from state machine models [25]. They studied and compared over 320 variants that relied on different diversity metrics. Based on their experiments, they find that similarity-based testing techniques outperformed other techniques in bug detection rates and computational cost. Biagiola et al. introduced a web test generation algorithm that produces and selects candidate test inputs that are executed in the browser based on their diversity [5]. They verify that their diversity-based testing technique achieved higher code coverage and fault detection rates when compared to state-of-the-art, search-based web test generators [41]. In this article, we conduct an empirical study on the testing effectiveness and input diversity of the existing testing approaches for DL libraries. According to the empirical results, we find that the testing approach with high diversity could detect more bugs in DL libraries and achieve positive performance.

Deep Learning Library. DL libraries are the infrastructures for DL-based software in multiple domains, providing a number of operators as APIs [62]. Early DL libraries such as Theano encapsulate operators for multidimensional arrays [3], which can only be used for simple DL tasks. With DL libraries such as Caffe, DL-based software has been promoted in the CV domain [30]. Along the line, more mature DL libraries such as TensorFlow, PyTorch, and Jittor have been released [2, 27, 47]. One significant improvement of them is that they enable DL-based software in different domains such as NLP [46] and ASR [36]. As DL libraries develop, testing approaches for DL libraries should also evolve. As part of DL Library, DL APIs, DL operators are considered prone to bugs. References [61, 62] report precision bugs of multiple DL operators. Wei et al. [28, 57, 59] propose equivalence rules or mutations to test DL APIs. However, DL libraries are not simply collections of operators or APIs. Instead, DL libraries are used to handle large-scale computations based on operator combinations. Four representative differential fuzzing approaches have been proposed to tackle the problem. CRADLE [48] takes the well-established models as the test inputs. AUDEE [24] mutates the well-established models to generate the test inputs via mutation rules without changing the structures. LEMON [58] mutates the structures of the well-established models to generate the test inputs. MUFFIN [22] generates test inputs based on predefined DL model structure templates. According to our empirical results in Section 3, existing testing approaches are ineffective in testing DL libraries across multiple domains. Their generated test inputs lack diversity and report many false positives. In contrast, *Gandalf* in this article can find bugs across multiple domains. The diversified test inputs generated by *Gandalf* allow the effective testing of DL libraries in a broader range of contexts with few false positives.

8 CONCLUSION

In this article, to understand the limitations of existing approaches, we first conduct an empirical study to evaluate the performance of four representative testing frameworks, namely, CRADLE, AUDEE, LEMON, and MUFFIN. We find that the test cases generated by all four approaches cannot generalize effectively across different task domains. Furthermore, the triggered bugs lack diversity and can usually arise false positives. To address these issues, we propose *Gandalf*, the first guided differential fuzzing approach based on generation for DL libraries. Through comprehensive evaluation, we demonstrate that *Gandalf* alleviates all aforementioned problems and can test DL libraries effectively and efficiently. During the evaluation, we also find 49 new unique bugs and report all of them to the developers of DL libraries. We receive their positive confirmation on these bugs.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] Reddit. 2021. Using PyTorch + NumPy? A bug that plagues thousands of open-source projects. Retrieved from https://www.reddit.com/r/MachineLearning/comments/mocpgj/p_using_pytorch_numpy_a_bug_that_plagues/

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. Retrieved from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] The Theano Development Team. 2016. *Theano: A Python Framework for Fast Computation of Mathematical Expressions.* CoRR abs/1605.02688 (2016).

[4] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* Association for Computing Machinery, New York, NY, 95–110. https://doi.org/10.1145/3062341.3062349

[5] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based web test generation. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 142–153. DOI:https://doi.org/10.1145/3338906.3338970

[6] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep reinforcement fuzzing. In *IEEE Security and Privacy Workshops*. IEEE Computer Society, 116–122. DOI:https://doi.org/10.1109/SPW.2018.00026

[7] Emanuela Gadelha Cartaxo, Patrícia D. L. Machado, and Francisco G. Oliveira Neto. 2011. On the use of a similarity function for test case selection in the context of model-based testing. *Softw. Test. Verific. Reliab.* 21, 2 (2011), 75–100. DOI:https://doi.org/10.1002/stvr.413

[8] Junjie Chen, Yihua Liang, Qingchao Shen, and Jiajun Jiang. 2022. Toward understanding deep learning framework bugs. *CoRR* abs/2203.04026 (2022).

[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A survey of compiler testing. *ACM Comput. Surv.* 53, 1 (2021), 4:1–4:36. DOI:https://doi.org/10.1145/3363562

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR* abs/1512.01274 (2015).

[11] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27. DOI:https://doi.org/10.1145/3143561

[12] Xuxin Chen, Ximin Wang, Ke Zhang, Kar-Ming Fung, Theresa C. Thai, Kathleen Moore, Robert S. Mannel, Hong Liu, Bin Zheng, and Yuchen Qiu. 2022. Recent advances and clinical applications of deep learning in medical image analysis. *Med. Image Anal.* 79 (2022), 102444. DOI:https://doi.org/10.1016/j.media.2022.102444

[13] Francisco Gomes de Oliveira Neto, Azeem Ahmad, Ola Leifler, Kristian Sandahl, and Eduard Enoiu. 2018. Improving continuous integration with similarity-based test case selection. In *13th International Workshop on Automation of Software Test*, Xiaoying Bai, J. Jenny Li, and Andreas Ulrich (Eds.). ACM, 39–45. DOI:https://doi.org/10.1145/3194733.3194744

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE Computer Society, 248–255. DOI:https://doi.org/10.1109/CVPR.2009.5206848

[15] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *ACM/IEEE International Conference on Automated Software Engineering*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 725–730. DOI:https://doi.org/10.1145/2642937.2642963

[16] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29. DOI:https://doi.org/10.1145/3133917

[17] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: Model-based quantitative analysis of stateful deep learning systems. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 477–487. DOI:https://doi.org/10.1145/3338906.3338954

[18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *J. Mach. Learn. Res.* 20, 1 (2019), 1997–2017.

[19] Robert Feldt, Simon M. Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 223–233. DOI:https://doi.org/10.1109/ICST.2016.33

[20] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. DOI:https://doi.org/10.1145/3368089.3409679

[21] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: an automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell*. Association for Computing Machinery, New York, NY, 13–20. https://doi.org/10.1145/2976002.2976017

[22] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. MUFFIN: Testing deep learning libraries via neural architecture fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, 1418–1430. DOI:https://doi.org/10.1145/3510003.3510092

[23] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *41st International Conference on Software Engineering: Software Engineering in Practice*, Helen Sharp and Mike Whalen (Eds.). IEEE/ACM, 71–80. DOI:https://doi.org/10.1109/ICSE-SEIP.2019.00016

[24] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. AUDEE: Automated testing for deep learning frameworks. In *35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 486–498. DOI: https://doi.org/10.1145/3324884.3416571

[25] Hadi Hemmati, Andrea Arcuri, and Lionel C. Briand. 2013. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.* 22, 1 (2013), 6:1–6:42. DOI: https://doi.org/10.1145/2430536.2430540

[26] Hadi Hemmati, Zhihan Fang, and Mika V. Mäntylä. 2015. Prioritizing manual test cases in traditional and rapid release environments. In *8th IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 1–10. DOI: https://doi.org/10.1109/ICST.2015.7102602

[27] Shi-Min Hu, Dun Liang, Guo-Ye Yang, Guo-Wei Yang, and Wen-Yang Zhou. 2020. Jittor: A novel deep learning framework with meta-operators and unified graph execution. *Sci. China Inf. Sci.* 63, 12 (2020). DOI: https://doi.org/10.1007/s11432-020-3097-4

[28] Li Jia, Hao Zhong, and Linpeng Huang. 2021. The unit test quality of deep learning libraries: A mutation analysis. In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 47–57. DOI: https://doi.org/10.1109/ICSME52107.2021.00011

[29] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside TensorFlow. In *25th International Conference on Database Systems for Advanced Applications (Lecture Notes in Computer Science,* Vol. 12112*)*, Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.). Springer, 604–620. DOI: https://doi.org/10.1007/978-3-030-59410-7_40

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia*, Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu (Eds.). ACM, 675–678. DOI: https://doi.org/10.1145/2647868.2654889

[31] Rauli Kaksonen, Marko Laakso, and Ari Takanen. 2001. *Software Security Assessment through Specification Mutations and Fault Injection*. Springer US, 173–183. DOI: https://doi.org/10.1007/978-0-387-35413-2_16

[32] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report.

[33] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Association for Computing Machinery, New York, NY, 386–399. https://doi.org/10.1145/2814270.2814319

[34] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. DOI: https://doi.org/10.1109/5.726791

[35] Xiaoting Li, Xiao Liu, Lingwei Chen, Rupesh Prajapati, and Dinghao Wu. 2022. ALPHAPROG: Reinforcement generation of valid programs for compiler fuzzing. In *36th AAAI Conference on Artificial Intelligence, 34th Conference on Innovative Applications of Artificial Intelligence, T12th Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 12559–12565. DOI: https://doi.org/10.1609/aaai.v36i11.21527

[36] Yuchen Liu, Jiajun Zhang, Hao Xiong, Long Zhou, Zhongjun He, Hua Wu, Haifeng Wang, and Chengqing Zong. 2020. Synchronous speech recognition and speech-to-text translation with interactive decoding. In *34th AAAI Conference on Artificial Intelligence, 32nd Innovative Applications of Artificial Intelligence Conference, 10th AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, 8417–8424. Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/6360

[37] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Dekang Lin, Yuji Matsumoto, and Rada Mihalcea (Eds.). The Association for Computer Linguistics, 142–150. Retrieved from https://aclanthology.org/P11-1015/

[38] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* 47, 11 (2021), 2312–2331. DOI: https://doi.org/10.1109/TSE.2019.2946563

[39] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 548–560. DOI: https://doi.org/10.1145/3314221.3314651

[40] William M. McKeeman. 1998. Differential testing for software. *Digit. Tech. J.* 10, 1 (1998), 100–107. Retrieved from http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[41] Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-based automatic testing of modern web applications. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 35–53. DOI: https://doi.org/10.1109/TSE.2011.28

[42] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44. DOI: https://doi.org/10.1145/96267.96279

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533. DOI:https://doi.org/10.1038/nature14236

[44] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *IEEE Asia Pacific Conference on Circuits and Systems*. IEEE, 676–679. DOI:https://doi.org/10.1109/APCCAS.2016.7804063

[45] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1224–1228. DOI:https://doi.org/10.1145/3324884.3418930

[46] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. 2021. A survey of the usages of deep learning for natural language processing. *IEEE Trans. Neural Netw. Learn. Syst.* 32, 2 (2021), 604–624. DOI:https://doi.org/10.1109/TNNLS.2020.2979670

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Annual Conference on Neural Information Processing Systems*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. Retrieved from https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[48] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries. In *41st International Conference on Software Engineering*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE/ACM, 1027–1038. DOI:https://doi.org/10.1109/ICSE.2019.00107

[49] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A survey on metamorphic testing. *IEEE Trans. Softw. Eng.* 42, 9 (2016), 805–824. DOI:https://doi.org/10.1109/TSE.2016.2532875

[50] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 2135. DOI:https://doi.org/10.1145/2939672.2945397

[51] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 36:1–36:12. DOI:https://doi.org/10.1145/3551349.3560429

[52] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *38th International Conference on Software Engineering*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 203–213. DOI:https://doi.org/10.1145/2884781.2884879

[53] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education.

[54] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *17th Asia Pacific Software Engineering Conference*, Jun Han and Tran Dan Thu (Eds.). IEEE Computer Society, 270–279. DOI:https://doi.org/10.1109/APSEC.2010.39

[55] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *40th International Conference on Software Engineering*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 303–314. DOI:https://doi.org/10.1145/3180155.3180220

[56] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael B. Abu-Ghazaleh. 2021. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2741–2758. Retrieved from https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[57] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: Creating equivalent graphs to test deep learning libraries. In *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, 798–810. DOI:https://doi.org/10.1145/3510003.3510165

[58] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 788–799. DOI:https://doi.org/10.1145/3368089.3409761

[59] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, 995–1007. DOI: https://doi.org/10.1145/3510003.3510041

[60] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *CoRR* abs/1708.07747 (2017).

[61] Xufan Zhang, Jiawei Liu, Ning Sun, Chunrong Fang, Jia Liu, Jiang Wang, Dong Chai, and Zhenyu Chen. 2021. Duo: Differential fuzzing for deep learning operators. *IEEE Trans. Reliab.* 70, 4 (2021), 1671–1685. DOI: https://doi.org/10.1109/TR.2021.3107165

[62] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: Precision testing of deep learning operators. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 400–412. DOI: https://doi.org/10.1145/3460319.3464843

[63] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. DOI: https://doi.org/10.1145/3213846.3213866

[64] Zhi Quan Zhou, Liqun Sun, Tsong Yueh Chen, and Dave Towey. 2020. Metamorphic relations for enhancing system understanding and use. *IEEE Trans. Softw. Eng.* 46, 10 (2020), 1120–1154. DOI: https://doi.org/10.1109/TSE.2018.2876433

Received 19 December 2022; revised 3 August 2023; accepted 20 September 2023