Mutation-Based Deep Learning Framework Testing Method in JavaScript Environment

Yinglong Zou
State Key Laboratory for Novel Software Technology
Nanjing University
China
652023320004@smail.nju.edu.cn

Juan Zhai
University of Massachusetts Amherst
United States
juanzhai@umass.edu

Chunrong Fang*
State Key Laboratory for Novel Software Technology
Nanjing University
China
fangchunrong@nju.edu.cn

Jiawei Liu
State Key Laboratory for Novel Software Technology
Nanjing University
China
jw.liu@smail.nju.edu.cn

Tao Zheng
State Key Laboratory for Novel Software Technology
Nanjing University
China
zt@nju.edu.cn

Zhenyu Chen*
State Key Laboratory for Novel Software Technology
Nanjing University
China
zychen@nju.edu.cn

ABSTRACT

In recent years, Deep Learning (DL) applications in JavaScript environment have become increasingly popular. As the infrastructure for DL applications, JavaScript DL frameworks play a crucial role in the development and deployment. It is essential to ensure the quality of JavaScript DL frameworks. However, the bottleneck of limited computational resources in the JavaScript environment brings new challenges to framework testing. Specifically, JavaScript DL frameworks are equipped with various optimization mechanisms (e.g., cache reuse, inference acceleration) to overcome the bottleneck of limited computational resources. These optimization mechanisms are overlooked by existing methods, resulting in many bugs in JavaScript DL frameworks being missed. To address the above challenges, we propose a mutation-based JavaScript DL framework testing method named DLJSFuzzer. DLJSFuzzer designs 13 tensor mutation rules targeting the cache reuse mechanism to generate test input tensors. Besides, DLJSFuzzer designs eight model mutation rules targeting the inference acceleration mechanism to generate test input models. To evaluate the effectiveness of DLJSFuzzer, we conduct experiments on the most widely-used JavaScript DL framework, TensorFlow.js. The experimental results show that DLJSFuzzer outperforms state-of-the-art methods in both effectiveness and efficiency. DLJSFuzzer successfully detects 21 unique crashes and 126 unique NaN & Inconsistency bugs. All detected crashes have been reported to the open-source community, with 12 of them already confirmed by developers. Additionally, DLJSFuzzer has improved by over 47% in model generation efficiency and over 91% in bug detection efficiency compared to all baselines.

KEYWORDS

Deep learning, framework testing, JavaScript environment

ACM Reference Format:
Yinglong Zou, Juan Zhai, Chunrong Fang, Jiawei Liu, Tao Zheng, and Zhenyu Chen. 2024. Mutation-Based Deep Learning Framework Testing Method in JavaScript Environment. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695478

1 INTRODUCTION

With the popularization of web applications, JavaScript has become the primary language for frontend development [11]. As developers increasingly choose to integrate deep learning (DL) models into web applications, the demand for developing and deploying DL models in JavaScript environments is growing [2]. The development and deployment of JavaScript DL applications rely on JavaScript DL frameworks (e.g., TensorFlow.js [39]), which provide developers with various functional operators. Unlike common DL frameworks like TensorFlow [1], JavaScript DL frameworks have limited computational resources [12]. To overcome this bottleneck, JavaScript DL frameworks design a series of optimization mechanisms (e.g. cache reuse, inference acceleration), which brings new challenges to DL framework testing in JavaScript environment. Unfortunately, existing methods failed to identify this important and unique challenge. Once overlooked, the testing will miss certain bugs (e.g., crashes arising from inference acceleration, see Section 4.4). The ignorance manifests in two main components of the existing methods: **test input model generation** and **test input tensor generation**.

In **test input tensor generation**, existing methods fail to flexibly adjust the shape and data types of test input tensors, making it challenging to effectively detect framework bugs that may arise from cache reuse mechanism. Specifically, existing DL framework testing methods define the shape and data types of test input tensors before testing and keep them unchanged throughout testing,

*Chunrong Fang and Zhenyu Chen are the corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695478

which brings two problems. Firstly, the fixed shape and data types of the test input tensors limit the ability to thoroughly test the cache reuse mechanism (e.g., it fails to test modules that deal with varying tensor shapes). Secondly, the fixed shapes and data types of test input tensors can hardly trigger misjudgments in the framework, which overlooks related bugs. For example, JavaScript DL frameworks occasionally misjudge two tensors with identical shapes as completely identical tensors, leading to incorrect cache reuse (see Section 2.2.1 for more information about this misjudgment).

In **test input model generation**, the models generated by existing methods lack model structures that can trigger inference acceleration, making it difficult to effectively detect framework bugs that may arise from inference acceleration mechanisms. Specifically, existing framework testing methods are designed for common DL frameworks like TensorFlow and PyTorch, without targeting the inference acceleration mechanism in the JavaScript environment. The models generated by existing methods hardly trigger inference acceleration strategies such as node optimization, operator reordering, and operator fusion. As a result, it is almost impossible to detect bugs in the JavaScript DL frameworks caused by inference acceleration by existing methods (see Section 4.4).

To better test JavaScript DL framework, we propose a mutation-based method called DLJSFuzzer, which targets widely used optimization mechanisms (including cache reuse and inference acceleration) in JavaScript DL frameworks to generate test input tensors and models. DLJSFuzzer designs 13 tensor mutation rules to generate new test input tensors flexibly in the shape and data types, which contributes to testing the JavaScript DL framework components related to the cache reuse mechanism. In addition, DLJSFuzzer designs eight model mutation rules targeting three different inference acceleration strategies (including node optimization, operator reordering, and operator fusion) to generate new models successfully triggering the inference acceleration mechanism. We apply DLJSFuzzer to test the most widely used JavaScript DL framework, TensorFlow.js [39], and compare it against three state-of-the-art methods (including LEMON [41], Muffin [14], and Gandalf [29]). Experimental results show that DLJSFuzzer outperforms all baselines in effectiveness, and achieves an over 47% improvement in model generation efficiency and an over 91% improvement in bug detection efficiency. DLJSFuzzer has successfully detected 21 unique crashes and 126 unique NaN & Inconsistency bugs. All detected crashes have been reported to the open-source community, with 12 of them already confirmed by developers. The main root causes of the detected crashes include cache reuse, implementation bug, inference acceleration, and others. The main root causes of the detected NaN & Inconsistency bugs include precision, environment, cache reuse, and randomness. However, all crashes detected by baselines stem from cache reuse, lacking those from implementation bug, inference acceleration, and others. All NaN & Inconsistency bugs detected by baselines stem from random and precision, lacking those from environment and cache reuse.

Our main contributions are as follows:

- We propose a mutation-based JavaScript DL framework testing method, which is the first to target optimization mechanism in JavaScript DL frameworks.
- We design 13 mutation rules for generating test input tensors targeting the cache reuse mechanism in the JavaScript environment, supporting the shape and data type of the test input tensors to change flexibly during method execution.
- We design eight mutation rules for generating test input models targeting the inference acceleration mechanism in JavaScript DL frameworks, which contributes to generating more models covering inference acceleration strategies.
- We implement the method as a tool named DLJSFuzzer and apply DLJSFuzzer to detect the most widely used JavaScript DL framework, TensorFlow.js. Experimental results show that DLJSFuzzer successfully detects 21 crashes and 126 NaN & Inconsistency bugs. In addition, DLJSFuzzer achieves an over 47% improvement in model generation efficiency and an over 91% improvement in bug detection efficiency.

We make all the data and code used in our paper publicly available on Github: https://github.com/DLJSFuzzer/DLJSFuzzer.

2 BACKGROUND

2.1 DL in JavaScript Environment

With the development of web browsers and the Node.js platform, the application of deep learning in the JavaScript environment is becoming increasingly popular. The JavaScript DL frameworks are the infrastructure for DL applications, which are designed to meet the need for developing and deploying DL models in the JavaScript environment. Among them, TensorFlow.js is the most commonly used JavaScript DL framework [37]. Through TensorFlow.js, developers can implement applications in multiple areas such as image classification [4], object detection [48], natural language processing [6], and recommendation systems [26].

Despite the widespread application of JavaScript DL frameworks, they also have some limitations. Specifically, compared to the local environment, the deployment environment of JavaScript DL frameworks has more scarce computational resources. On the one hand, the browser cache size is limited. The browser cache is mainly used to temporarily store model parameters (including input tensors). The cache size of commonly used browsers is usually less than 100 MB, which is much smaller than the total size of parameters in the model (e.g., the parameter size of VGG reaches 140 MB). The significant disparity between the browser cache and the total size of model parameters results in excessive cache read and write operations during the model inference process, severely reducing the efficiency of model inference. On the other hand, the model inference speed is limited. JavaScript DL frameworks are often deployed on mobile devices (e.g., mobile phones). Compared to deploying in a local environment, the floating point arithmetic speed in mobile devices is more limited, leading to a slower model inference speed.

2.2 Optimization in JavaScript DL Framework

As introduced in Section 2.1, the scarcity of computing resources (e.g., limited browser cache and model inference speed) is a major bottleneck of developing and deploying DL models in the JavaScript environment. To overcome this bottleneck, a series of optimization mechanisms are designed in JavaScript DL frameworks. Among

them, **cache reuse** and **inference acceleration** are the most popular ones. In this section, we will take TensorFlow.js as an example to explain these optimization mechanisms in detail.

2.2.1 cache reuse [23]. Unlike traditional cache reuse mechanism, JavaScript-environment cache reuse mechanism is designed to adapt the small browser cache in the JavaScript environment, which relies on complex browser cache refresh strategies. If the tensors that need to be newly loaded are identical to the ones already loaded, the browser will no longer load the new tensors, but will directly use the ones that have been loaded. However, the JavaScript DL framework may make misjudgments when determining if two tensors are completely identical. If the newly loaded tensor has the same dimensions as the already loaded tensor, it may be mistakenly considered identical. This misjudgment can lead to loading parameters that have the same dimensions but are not completely identical, resulting in framework crashes or incorrect model inference results.

2.2.2 inference acceleration [27]. TensorFlow.js primarily accelerates inference by optimizing the computation graph corresponding to the model. The computation graph is a directed acyclic graph that represents the operators in the model and the data flow corresponding to the model. Common inference acceleration strategies include **node optimization**, **operator reordering**, and **operator fusion**. A more detailed introduction to these inference acceleration strategies is as follows.

Node Optimization. Node optimization aims to replace nodes in the model with nodes that have the same computational logic but are more suitable for deployment in a JavaScript environment. In the DL model, a node is a combination of an operator and tensors & parameters corresponding to this operator. Node optimization simplifies the computational formulas corresponding to nodes and maps the simplified formulas to more efficient nodes. Figure 1 shows an example of node optimization. The rectangles in this figure represent nodes, which include operators and their corresponding parameters. Node optimization replaces nodes corresponding to the operator *Batchnorm* with nodes corresponding to the operator *Scale*. In this way, it is possible to reduce computational complexity, decrease the model's parameter size, and alleviate the insufficient browser cache in the JavaScript environment.

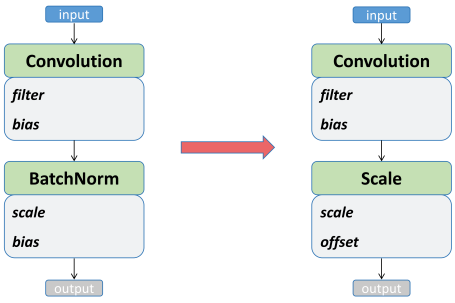

Figure 1: An Example of Node Optimization

Operator Reordering. Operator reordering reduces the computational workload of model inference by rearranging the topological order of operators in the model while maintaining the same inference accuracy. The goal of operator reordering is to improve the efficiency of model inference by reorganizing the computational sequence of operators, reducing unnecessary computations and data transfers. Figure 2 shows an example of operator reordering. In this example, the computational process is simplified by exchanging the order of the operator *MaxPool* and the operator *Sigmoid*.

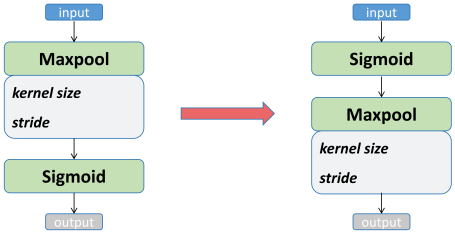

Figure 2: An Example of Operator Reordering

Operator Fusion. Operator fusion is a model compression technique that merges multiple related operators into a single operator based on the equivalence relationship between operators and operator combinations. In DL models, a large number of repeated calculations and frequent memory accesses lead to inference latency and energy consumption issues. Operator fusion analyzes the model's topology, identifies operators that can be merged, and combines them into a new operator according to certain rules. Operator fusion can effectively reduce the number of nodes and edges in the computation graph corresponding to the model. Reducing the number of nodes means fewer memory accesses, and reducing the number of edges means fewer computational steps. Therefore, operator fusion can reduce the computational complexity and storage usage, thereby accelerating the model's inference. Figure 3 shows an example of operator fusion. In this example, the operator sequence ($Convolution$, $BatchNorm$, $RELU$) is fused to a CBR (Conv_Batch_Relu) block.

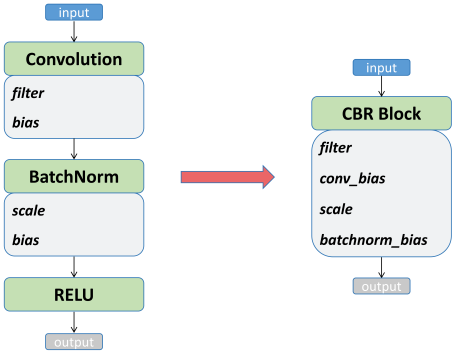

Figure 3: An Example of Operator Fusion

Unlike traditional inference acceleration, JavaScript-environment inference acceleration employs new optimization strategies (e.g., In-place Node Optimization) and unique implementations of common strategies (e.g., depending on the browser API), introducing unique types of bugs to JavaScript-environment DL frameworks (see Section 4.4).

3 METHODOLOGY

3.1 Overview

In this work, we propose DLJSFuzzer, a mutation-based JavaScript DL framework testing method. Figure 4 shows the overall workflow of DLJSFuzzer. Firstly, DLJSFuzzer generates seed tensors and designs tensor mutation rules targeting the cache reuse mechanism to generate new test input tensors flexibly in the shape and data types (Section 3.2). Secondly, DLJSFuzzer generates a seed model and designs model mutation rules targeting inference acceleration strategies (e.g., node optimization, operator reordering, and operator fusion) to generate new test input models (Section 3.3). Based on generated test inputs, DLJSFuzzer detects DL framework bugs by analyzing the testing results under different frameworks and browsers (Section 3.4), and heuristically guides the next round based on these results (Section 3.5).

3.2 Test Input Tensor Generation

The test input tensor is an important part of the test input in DL framework testing. The cache reuse mechanism applied in JavaScript DL frameworks poses new challenges for generating test input tensors. Some false positives usually occur when judging whether two tensors are exactly the same. These false positives may lead to incorrect inference results or even cause crashes in DL frameworks (see Section 2.2). However, these quality issues cannot be successfully detected by existing DL framework testing methods. Specifically, existing DL framework testing methods require determining the shape and data type of the test input tensors before the testing method starts. Throughout the testing process, the shapes and data types of all test input tensors remain almost (if not completely) unchanged. The above process has the following two limitations. On the one hand, compared to the flexible and variable test input tensors in terms of shape and data types, the test input tensors generated by existing methods cannot fully and comprehensively test the cache reuse mechanism (e.g., modules related to handling different tensor shapes are not tested). On the other hand, test input tensors generated by existing methods with unchanged shapes and data types almost cannot trigger the cache reuse mechanism to misjudge whether two tensors are completely identical, thus overlooking relevant DL framework quality issues.

To overcome these limitations, DLJSFuzzer designs a mutation-based test input tensor generation method, which proposes a series of tensor mutation strategies to flexibly change the shape and data type of test input tensors. The detailed method is as follows: Firstly, DLJSFuzzer generates a seed tensor in the format of (N, C, H, W). DLJSFuzzer supports both randomly generating the seed tensor and loading the seed tensor from open-source datasets (e.g., MNIST). All adopted open-source datasets are listed in Section 4.1. After that, DLJSFuzzer randomly selects a tensor mutation rule to change the shape and data types of the test input tensor. Table 1 shows all tensor mutation rules in DLJSFuzzer. In this table, the first column is the index. The following columns are, respectively, the name, the description, and the type of the mutation rule. According to the different types of tensor operations, the tensor mutation rules in DLJSFuzzer are mainly divided into five categories: tensor copy, tensor padding, tensor transpose, tensor cropping, and data type transformation. Among them, the first four categories are the transformation of the tensor shape, while the last category is the transformation of the tensor data type. More detailed information about these mutation rules is as follows.

Tensor Copy. The **Mutation Rule 1-4** belong to the Tensor Copy category. The aim of these mutation rules is to double the size of the tensor along the corresponding dimension. These mutation rules copy the tensor and concatenate the original tensor with the newly copied tensor along the corresponding dimension.

Tensor Padding. The **Mutation Rule 5-8** belong to the Tensor Padding category. These mutation rules zero pad the tensor along the corresponding dimension. Unlike tensor copy, tensor padding results in a smaller increase in tensor size. The mutation rules of DLJSFuzzer support different granularities of tensor modifications, making the changes to tensor shapes more flexible.

Tensor Transpose. The **Mutation Rule 9** belongs to the Tensor Transpose category. This mutation rule transposes the tensor along the height and weight dimensions, which is a mutation rule that only changes the shape of the tensor but not the size of the tensor.

Tensor Cropping. The **Mutation Rule 10** belongs to the Tensor Cropping category. The aim of this mutation rule is to reduce the size of the tensor. The cropped tensor is a part of the original tensor. To ensure the flexibility of DLJSFuzzer in changing tensor shapes, the shape of the cropped tensor can be randomly specified.

Data Type Transformation. The **Mutation Rule 11-13** belong to the Data Type Transformation category. The aim of these mutation rules is to change the data type of the tensor. Since different data types require different storage space, changing the data type means changing the tensor size. DLJSFuzzer supports three common data types: 32-bit $float$, 64-bit $double$, and 16-bit $bfloat16$.

3.3 Test Input Model Generation

The test input model is another important part of the test input in DL framework testing. The existing DL framework testing methods lack targeted designs for testing JavaScript DL frameworks. The generated models contain almost no model structures suitable for inference acceleration, making it difficult to effectively detect bugs that may be caused by inference acceleration mechanisms. To overcome this limitation, DLJSFuzzer designs a series of model mutation rules targeting widely used inference acceleration strategies in JavaScript DL frameworks (e.g., Node Optimization, Operator Reordering, and Operator Fusion). DLJSFuzzer first generates a seed model (see Section 3.3.1), and then applies a mutation rule to generate a new model based on the seed model (see Section 3.3.2).

3.3.1 Seed Model Generation. Existing DL framework testing methods usually (e.g. Muffin[14], LEMON[41]) use publicly available models as seed models, and generate new models by making small modifications to these seed models through a small number of mutations. The small number of mutations restricts the exploration space of model structures, resulting in generated models that are too similar to the seed models. In order to generate more diversified models, DLJSFuzzer directly generates relatively simple models as the seed model and applies a higher number of mutations to the same seed model. Next, we will introduce the model representation in DLJSFuzzer, and then use this model representation to describe the structure of the seed model.

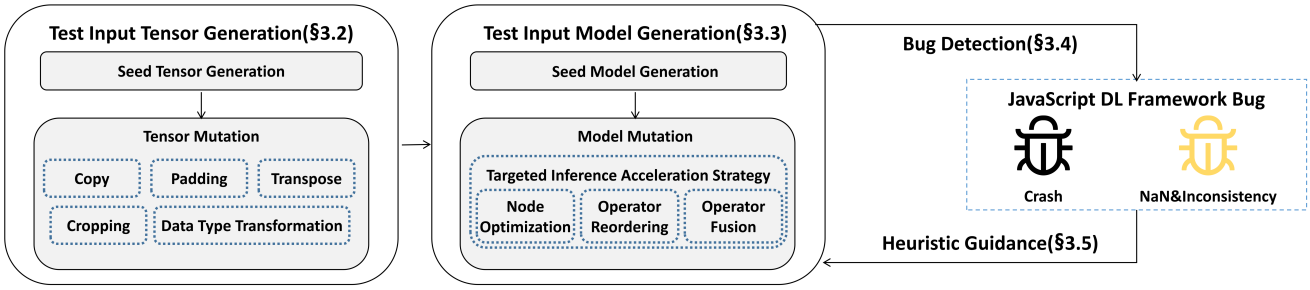

Figure 4: The Workflow of DLJSFuzzer

Table 1: Tensor Mutation Rules in DLJSFuzzer

ID	Mutation Rule	Description	Type
1	Weight Dimension Copy (WDC)	Copys the tensor and concatenates along the weight dimension to double tensor size	Tensor Copy
2	Height Dimension Copy (HDC)	Copys the tensor and concatenates along the height dimension to double tensor size	
3	Channel Dimension Copy (CDC)	Copys the tensor and concatenates along the channel dimension to double tensor size	
4	Batch Dimension Copy (BDC)	Copys the tensor and concatenates along the batch dimension to double tensor size	
5	Weight Dimension Padding (WDP)	Zero pads the tensor along the weight dimension	Tensor Padding
6	Height Dimension Padding (HDP)	Zero pads the tensor along the height dimension	
7	Channel Dimension Padding (CDP)	Zero pads the tensor along the channel dimension	
8	Batch Dimension Padding (BDP)	Zero pads the tensor along the batch dimension	
9	Height-Weight Dimension Transpose (HWDT)	Transposes the tensor in the height and weight dimension	Tensor Transpose
10	Random Cropping (RC)	Randomly crops the tensor to reduce its size	Tensor Cropping
11	Float Transformation (FT)	Converts the tensor to the data type $float$	Data Type Transformation
12	Double Transformation (DT)	Converts the tensor to the data type $double$	
13	BFloat Transformation (BFT)	Converts the tensor to the data type $bfloat16$	

Model Representation. DLJSFuzzer uses a Directed Acyclic Graph (DAG) to represent each model. The DAG consists of a vertex set V_G and an edge set E_G, where each vertex in V_G represents a tensor, and each edge in E_G represents an operator. Various types of edges in the DAG denote different types of operators. When multiple edges converge at the same vertex, the tensor represented by each edge's starting vertex is processed by the corresponding operator, and the output tensors are fused into the tensor represented by the shared endpoint vertex. DLJSFuzzer employs the operator $BatchNorm$ for tensor fusion. In order to maintain the integrity of the test input models, the DAG should adhere to two constraints. Firstly, as test input models should have only one input tensor and one output tensor, the DAG of the models should contain only one source vertex and one sink vertex. Secondly, to ensure the successful construction of generated models, the DAG of the model should be a connected graph.

Structure of the Initial Seed Model. The DAG of the initial seed model in DLJSFuzzer is an operator chain from the source vertex to the sink vertex, which is only composed of the operator $identity$. The length of this operator chain determines the size of the newly generated model, and it is predetermined before the start of the method. In addition, all newly generated models are regarded as seed models in the following rounds.

3.3.2 Mutation Rules. DLJSFuzzer generates new models by mutating the seed model generated in Section 3.3.1. To effectively detect quality issues in JavaScript DL frameworks caused by inference acceleration mechanisms, DLJSFuzzer designs eight model mutation rules targeting widely used inference acceleration strategies (e.g., Node Optimization, Operator Reordering, and Operator Fusion), as shown in Table 2. In this table, the first column represents the index, while the following three columns respectively represent the mutation rule, the corresponding description, and the inference optimization strategy the mutation rule targets. Next, we will introduce these mutation rules in detail.

Mutation Rule 1-3. These mutation rules (including ZDT, IPT, and HWR) are proposed targeting at Node Optimization. In order to accelerate inference, the JavaScript DL framework replaces high computational resource-consuming nodes with equivalent nodes that have lower computational resource consumption. To effectively trigger these inference acceleration strategies, DLJSFuzzer designs mutation rules to insert high computational resource-consuming

Table 2: Model Mutation Rules in DLJSFuzzer

ID	Mutation Rule	Description	Targeted Inference Acceleration Strategy
1	Zero-Dimensional Tensorization (ZDT)	Inserts an operator containing zero-dimensional tensor	Node Optimization
2	In-Place Transpose (IPT)	Inserts the operator $Transpose$ without data movement	
3	Height-Weight ReduceMean (HWR)	Inserts the operator *ReduceMean* only on the height and weight dimensions	
4	Transpose-based Operator Reordering (TOR)	Inserts the operator combination based on the equivalence relation $A^T \times B^T = (B \times A)^T$	Operator Reordering
5	Maxpool-based Operator Reordering (MOR)	Inserts the operator combination ($MaxPool$, $Sigmoid$) or ($MaxPool$, $Softmax$)	
6	Equivalent Separable Convolution (ESC)	Inserts the operator combination consisting of two depth -wise convolutions equivalent to a separable convolution	Operator Fusion
7	Equivalent CBR Block (ECB)	Inserts the operator combination consisting of three operators: ($Convolution$, $BatchNorm$, $RELU$)	
8	Random Operator Replace (ROR)	Replaces an operator with a randomly selected operator	None

nodes in the model. Specifically, in **Rule 1** (ZDT), DLJSFuzzer inserts an operator containing a zero-dimensional tensor in the model to induce the JavaScript DL framework to replace the zero-dimensional tensor in this operator with a constant. In **Rule 2** (IPT), DLJSFuzzer inserts an operator $Transpose$ that does not move data to induce the JavaScript DL framework to replace this operator with an operator *Reshape* that provides the same computation results but executes faster. In **Rule 3** (HWR), DLJSFuzzer inserts an operator *ReduceMean* that is only on the height and weight dimensions to induce the JavaScript DL framework to replace this operator with an equivalent operator $AveragePool$.

Mutation Rule 4 & 5. These mutation rules (including TOR and MOR) are proposed targeting at Operator Reordering. As introduced in Section 2.2.2, many operator combinations satisfy equivalence relations (e.g., commutativity, associativity, and distributivity). To simplify computation and accelerate inference, the JavaScript DL framework rearranges operators in DL model based on these equivalence relations. Mutation 4 & 5 insert operator combinations that satisfy the above mentioned equivalence relations in the model in the order before reordering, thereby inducing the JavaScript DL framework to rearrange the operator combination. Specifically, **Rule 4** (TOR) targets the equivalence relation $A^T \times B^T \iff (B \times A)^T$, inserting the operator combination $A^T \times B^T$ into the model. **Rule 5** (MOR) targets the equivalence relation $(Maxpool, Sigmoid) \iff (SigmoidMaxpool)$. Since the former executes faster than the latter, DLJSFuzzer inserts the latter operator combination in the model to induce the JavaScript DL framework to reorder it into the former. This Mutation Rule also applies to the equivalence relation $(Maxpool, Softmax) \iff (Softmax, Maxpool)$.

Mutation Rule 6 & 7. These mutation rules (including ESC and ECB) are proposed targeting Operator Fusion. As introduced in Section 2.2.2, in addition to the equivalence relations between operator combinations mentioned above, many operators and operator combinations also satisfy equivalence relations. Because the inference speed of a single operator is usually faster than that of an equivalent operator combination, the JavaScript DL framework often fuses operator combinations that satisfy equivalence relations into a single operator. To effectively trigger Operator Fusion, DLJSFuzzer inserts operator combinations that satisfy equivalence relations into the model. Specifically, in **Rule 6** (ESC), since a separable convolution is equivalent to two depth-wise convolutions, DLJSFuzzer inserts two consecutive separable convolutions in the model to induce Operator Fusion. In **Rule 7** (ECB), since operator combination ($Convolution$, $BatchNorm$, $RELU$) is equivalent to a fused CBR (Conv_Batch_Relu) block, but the former consumes higher computational resources, DLJSFuzzer inserts the operator combination ($Convolution$, $BatchNorm$, $RELU$) into the model to induce Operator Fusion.

Mutation Rule 8. This mutation rule (ROR) aims at generating models with various operators. ROR randomly selects an operator in the model to be mutated and replaces the selected operator with another operator. It is worth noting that, the operator *None* is also a candidate operator. When the operator *None* in the model is replaced with another operator, it means inserting an operator into the model; when other operators in the model are replaced with the operator *None*, it means deleting the operator from the model.

3.4 Bug Detection

Based on the test inputs generated in Section 3.2 and 3.3, DLJSFuzzer employs differential testing between TensorFlow [1], PyTorch [34], and the most widely used JavaScript DL framework, TensorFlow.js [39]. To mitigate the impact of browser environment differences, DLJSFuzzer deploys TensorFlow.js in two widely used browsers, including Google Chrome and Microsoft Edge. When the same DL model produces different inference results in the two browsers, DLJSFuzzer will separately re-execute the model's inference process in both browsers until the inference results are equal. If the inference results remain different, the model will be discarded. DLJSFuzzer focuses on two categories of bugs: non-numerical bugs and numerical bugs. Specifically, non-numerical bugs involve crashes, which are detected by analyzing execution logs. Numerical bugs involve Not-A-Number (NaN) bugs and inconsistency bugs. Among them, NaN bugs are detected when the inference result in TensorFlow.js contains NaN while other frameworks' inference results do not. The detection of inconsistency bugs is based on the *inconsistency* between the inference results. The

calculation of *inconsistency* is as follows: ① Denote the inference result under each DL framework as $result_f$. The f in $result_f$ denotes the frameworks. ② Calculate the difference between all $result_f$ as $result\_diff_{ij}$. The i and j in $result\_diff_{ij}$ denote the frameworks. ③ Calculate the max scalar value in each $result\_diff_{ij}$ as $inconsistency_{ij}$. The i and j in $inconsistency_{ij}$ denote frameworks.

An inconsistency bug is detected when the *inconsistency* between TensorFlow and TensorFlow.js, and between PyTorch and TensorFlow.js, both exceed the pre-specified threshold ϵ. To ensure fairness and minimize errors from computational randomness, DLMOSA sets the threshold ϵ to the maximum among all existing methods (e.g., Gandalf [29], Muffin [14], etc), which is 0.15.

3.5 Heuristic Guidance

To continuously improve the effectiveness of framework testing, DLJSFuzzer designs the heuristic guidance to control the test input model generation in the following rounds based on the bug detection performance of newly generated models. In this section, we will introduce the designed heuristic indicator (Section 3.5.1), and its application in seed model selection (Section 3.5.2) and model mutation rule selection (Section 3.5.3).

3.5.1 Heuristic Indicator. DLJSFuzzer designs the heuristic indicator $fitness$ to quantitatively measure the bug detection effectiveness. The $fitness$ of each newly generated model is separately calculated and recorded, which is used to reflect the bug detection performance. Depending on the categories of bugs detected, DLJSFuzzer supports different $fitness$ calculations. Specifically, when detecting crashes and NaN bugs, $fitness$ is m_d. m_d is 0 when no crashes and NaN bugs are triggered, and m_d is the mean value of the test input tensor when these bugs are triggered. When detecting inconsistency bugs, $fitness$ is *inconsistency* (see Section 3.4).

3.5.2 Heuristic Guidance in Seed Model Selection. To further enhance the effectiveness of the DL framework testing, DLJSFuzzer uses a tournament algorithm [10] to select the model with the best bug detection performance from the initial seed model and all models generated in previous rounds as the seed model for the next round of mutation (the first round directly selects the initial seed model). The bug detection performance of the model is measured by the heuristic indicator $fitness$ (introduced in Section 3.5.1).

3.5.3 Heuristic Guidance in Model Mutation Rules Selection. Heuristically selecting mutation rules helps to continuously enhance the effectiveness of DLJSFuzzer in detecting bugs in DL frameworks. DLJSFuzzer applies the heuristic indicator $fitness$ to control the probability of each mutation rule being selected. Based on $fitness$, the probability p of each mutation rule being selected is calculated as follows. Firstly, DLJSFuzzer calculates the total contribution c of each mutation rule:

$$c_1 = c_0 + \Delta fitness$$

where c_1 and c_0 is the total contribution c after and before mutation, respectively. $\Delta fitness = fitness_{new} - fitness_{old}$, where $fitness_{new}$ is the $fitness$ of the newly generated model and $fitness_{old}$ is the $fitness$ of the model before mutation. Based on c, the probability p of each mutation rule being selected is calculated as follows:

$$p = \frac{c}{\sum_{i=1}^{n} c_i}$$

where n is the number of mutation rules. It is worth noting that, in order to generate models with various operators, when calculating n, DLJSFuzzer treats the replacement of each operator with **Rule 8** (ROR) (see Section 3.3) as a mutation rule.

4 EVALUATION

To evaluate the bug detection effectiveness of DLJSFuzzer, we conduct an experiment to compare DLJSFuzzer with three state-of-the-art methods, including LEMON[41], Muffin[14], and Gandalf[29]. All the code and experimental results have been open-sourced in our Github repository.

4.1 Experimental Setup

Existing DL framework testing methods do not support detecting JavaScript DL frameworks (e.g., TensorFlow.js [39]). For the fairness of our experiment, all methods in our evaluation are configured to use the same parameter and code to call the API of the JavaScript DL framework. We separately download the source code of all baselines and run them to generate test input tensors and test input models. After that, we execute the test inputs generated by each method in the same experimental environment. Each method is executed for three hours respectively. The initial seeds are adopted as specified in each method's paper. The workstation's operating system is Ubuntu 22.04, equipped with an Intel Core Processor (Skylake) (16 cores, 2.0GHz). The framework being detected is TensorFlow.js, and other frameworks assisting in the differential testing include TensorFlow [1] and PyTorch [34]. The versions of each framework are: TensorFlow.js 3.19.0, TensorFlow 2.9.0, PyTorch 1.12.0. TensorFlow.js is deployed on two widely-used browsers, Microsoft Edge and Google Chrome. The browser versions are: Microsoft Edge 125.0.2535.67 (64 bit) and Google Chrome 125.0.6422.112 (64 bit). The threshold value ϵ for detecting NaN & Inconsistency bugs is set to 0.15 (see Section 3.4). DLJSFuzzer not only supports randomly generating test input tensors but also loading test input tensors from open-source datasets. DLJSFuzzer supports all datasets used by baselines, including: MNIST, Fashion-MNIST, CIFAR-10, ImageNet, Sine-Wave, and Stock-Price.

4.2 Research Questions

In our evaluation, we conduct the experiment to answer the following four research questions.

- **RQ1:** How does DLJSFuzzer perform in detecting JavaScript DL framework bugs?
- **RQ2:** What is the root cause of detected JavaScript DL framework bugs?
- **RQ3:** How does DLJSFuzzer perform in efficiency and the diversity of generated test input models?
- **RQ4:** To what extent do the proposed test input tensor and model generation method contribute to bug detection?

In RQ1, we evaluate the effectiveness of DLJSFuzzer when detecting Crashes and NaN & Inconsistency bugs. In RQ2, we analyze the root cause of all detected crashes. In RQ3, we evaluate the method efficiency and the diversity of test input models (representing the test sufficiency). In RQ4, we conduct an ablation study to investigate the contribution of the proposed test input tensor mutation and test input model mutation.

4.3 RQ1: Effectiveness in Detecting Bugs

DLJSFuzzer detects both numerical bugs (including crashes) and non-numerical bugs (including NaN & Inconsistency bugs) in JavaScript DL frameworks. As mentioned in Section 3.4, crashes are confirmed by analyzing the model's runtime logs. The NaN & Inconsistency bugs are confirmed based on the inference results under different frameworks. Specifically, if there is NaN in the inference result under TensorFlow.js, but no NaN under TensorFlow or PyTorch, an NAN bug is successfully detected. If the max difference between the inference results exceeds the specified threshold ϵ, an inconsistency bug is successfully detected. To avoid redundancy, DLJSFuzzer implements the following post-processing measures: For crash detection, two researchers independently analyze the model's runtime logs to eliminate identical crashes. For NaN & Inconsistency bug detection, DLJSFuzzer records the model structure that triggers the bug. If the model structures that trigger the bug are identical, the two bugs are considered the same. The number of unique bugs detected by DLJSFuzzer in TensorFlow.js in all datasets is shown in Table 3. The first column in the table represents the datasets used. The last two columns represent the number of crashes detected and the number of NaN & Inconsistency bugs detected, respectively. The last row in the table represents the total number of bugs detected by DLJSFuzzer in all datasets after removing redundancy. This table shows that DLJSFuzzer successfully detects 21 unique crashes and 126 unique NaN & Inconsistency bugs. All crashes have been reported to the open-source community, with 12 of them already confirmed by developers, and the rest are still being processed.

Table 3: Number of Unique Detected Bugs in Each Dataset

Dataset	Crash	NaN & Inconsistency
Random	18	69
MNIST	11	1
Fashion-MNIST	10	13
CIFAR-10	12	0
ImageNet	19	88
Sine-Wave	8	2
Stock-Price	8	3
Total	21	126

To further evaluate the effectiveness of DLJSFuzzer, we compare the number of unique bugs detected by DLJSFuzzer and all baselines (including LEMON, Muffin, and Gandalf). The experimental results are shown in Table 4. The first column represents the method name, and the following two columns represent the number of detected crashes and NaN & Inconsistency bugs, respectively. The experimental result shows that DLJSFuzzer successfully detects 21 crashes and 126 NaN & Inconsistency bugs, which is the most compared to all baselines. LEMON only detects four NaN & Inconsistency bugs and does not detect any crashes, which is limited by the low diversity of generated models. Muffin detects 12 crashes and 2 NaN & Inconsistency bugs. The latter number is limited due to the lack of effective test input tensor mutation. The models generated by Gandalf are too simple, thus failing to trigger any bugs successfully. In addition, we analyze the containment relationship and the root causes of the bugs detected by these methods (see Section 4.4 for more introduction about these root causes). We find that the root cause of all crashes detected by Muffin is cache reuse, which is completely covered by our method. The root cause of all NaN & Inconsistency bugs detected by LEMON and Muffin is random and precision, also completely covered by our method.

Table 4: Comparison in the Number of Unique Detected Bugs

Method	Crash	NaN & Inconsistency
DLJSFuzzer	21	126
LEMON	0	4
Muffin	11	2
Gandalf	0	0

Answer to RQ1: DLJSFuzzer successfully detects 21 unique crashes and 126 unique NaN & inconsistency bugs in TensorFlow.js, which is the most among existing baselines and completely covers all bugs detected by existing baselines. The experimental results fully demonstrate the superior effectiveness of DLJSFuzzer in detecting bugs in JavaScript DL frameworks.

4.4 RQ2: Root Cause

As shown in Table 3, DLJSFuzzer successfully detects 21 unique crashes and 126 NaN & Inconsistency bugs. In this section, we will further analyze the root cause of these bugs as follows.

4.4.1 Root Cause of Crashes. By analyzing the execution logs of detected crashes, DLJSFuzzer categorizes the root causes of detected crashes into four types: cache reuse, implementation bug, inference acceleration, and others. The number of crashes in each category is illustrated in Figure 5.

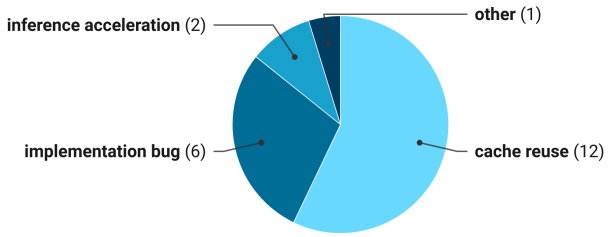

Figure 5: Root Cause of Detected Crashes

Cache reuse (12/21, 57.14%). Cache reuse is the most common root cause found in the detected crashes. This type of bug primarily stems from inconsistencies in tensor dimensions. More specifically, flaws in the cache reuse mechanism result in the loss of certain parameters within the tensor, consequently triggering alterations in the tensor dimensions. The bug occurs when the actual tensor dimensions received by the operator do not align with the declared dimensions.

Implementation bug (6/21, 28.57%). This type of bug primarily stems from certain operators in TensorFlow.js lacking support for specific data types or parameter settings. For instance, the operator *Batch_normalization* in TensorFlow.js does not support the data

type $bfloat16$. The bug occurs when a tensor of data type $bfloat16$ is passed to this operator.

Inference acceleration (2/21, 9.52%). This type of bug primarily stems from parameter setting errors in TensorFlow.js during inference acceleration. Specifically, TensorFlow.js incorrectly sets parameters for optimized operators (or operator combinations) while applying strategies such as node optimization and operator fusion, leading to framework bugs.

Other (1/21, 4.76%). There are still some other root causes of detected bugs. For example, due to the weakly-typed nature of JavaScript, occasional data type conflicts may arise when passing parameters within TensorFlow.js, leading to framework bugs.

4.4.2 Root Cause of NaN & Inconsistency Bugs. By reproducing and locating detected NaN & Inconsistency bugs, DLJSFuzzer categorizes root causes of detected NaN & Inconsistency bugs into four types: precision, environment, cache reuse, and random. Figure 6 shows the number of NaN & Inconsistency bugs in each category.

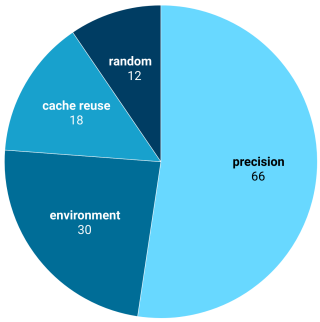

Figure 6: Root Cause of Detected NaN & Inconsistency Bugs

Precision (66/126, 52.38%). Precision loss is the most common root cause of NaN & Inconsistency bugs detected by DLJSFuzzer. On the one hand, large-scale numerical calculation inevitably comes with precision loss. On the other hand, the weakly-typed nature of JavaScript exacerbates precision loss. Precision loss accumulates during model inference, leading to framework bugs.

Environment (30/126, 23.81%). This type of bug primarily stems from the fluctuation of the framework deployment environment (especially browser environment). Compared to the local environment, the browser environment in which TensorFlow.js is deployed is more unstable and susceptible to factors such as network and operational context, leading to framework bugs.

Cache Reuse (18/126, 14.29%). The flawed cache reuse mechanism leads to framework bugs. Specifically, the flawed cache reuse occasionally causes parameters to be incorrectly overwritten or modified, resulting in framework bugs.

Random (12/126, 9.52%). The introduction of randomness by some operators under test also leads to some framework bugs, for example, the operator *Dropout*.

Answer to RQ2: In DLJSFuzzer, the root causes of detected crashes include cache reuse, implementation bug, inference acceleration, and other. The root causes of detected NaN & Inconsistency bugs include precision, environment, cache reuse, and random.

4.5 RQ3: Efficiency & Model Diversity

In this section, we evaluate the efficiency of DLJSFuzzer and the diversity of generated test input models. Model diversity is a widely used measurement to reflect the sufficiency of framework testing. The higher the model diversity, the more comprehensive the framework testing. Model diversity is calculated by the indicator *med* (mean edit distance). *med* is defined as the average of the edit distances between all pairs of models. The edit distance between two models is the minimum value of operations needed to transform one model into another. The comparison of efficiency is shown in Table 5. The first column represents the method name, and the second column represents the average time consumption for each round of framework testing. The last column represents the average time consumption for detecting each framework bug. Due to Gandalf not detecting any bugs, it is unable to calculate the average time Gandalf takes to detect each bug, so we mark it as NaN in the table. From this table, we can conclude that compared to all baselines, DLJSFuzzer has the shortest testing time per round (19.60s). Compared to the best performance in the baselines (36.99s per round), the model generation efficiency improvement is over 47%. Besides, DLJSFuzzer has the shortest time to detect each bug (73.47s). Compared to the best performance in the baselines (830.77s per bug), the bug detection efficiency improvement is over 91%. The improvement in the efficiency of DLJSFuzzer is attributed to the models generated by DLJSFuzzer effectively triggering the inference acceleration mechanism.

Table 5: Comparison in Efficiency

Method	Time Per Round (s)	Time Per Bug (s)
DLJSFuzzer	19.60	73.47
LEMON	63.91	2,700
Muffin	47.79	830.77
Gandalf	36.99	NaN

To calculate the model diversity, we save all models generated by each method in three hours and compute their *med*. The experimental results are shown in Figure 7. This figure uses different colors to represent different methods, with the numbers in the figure indicating the *med* of the models generated by each method. The figure shows that the *med* of the models generated by DLJSFuzzer is 6.46, higher than 5.73 in Muffin, 2.82 in Gandalf, and 0.13 in LEMON. We can conclude that the model mutation in DLJSFuzzer also contributes to increasing model diversity.

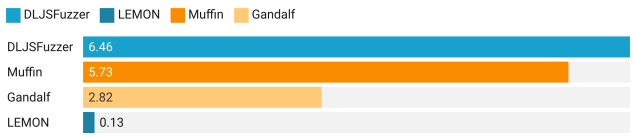

Figure 7: Comparison in Model Diversity

Answer to RQ3: Compared to three baselines, DLJSFuzzer has grown by over 47% in model generation efficiency and over 91% in bug detection efficiency. In addition, the models generated by DLJSFuzzer have the highest model diversity, reflecting the framework testing in DLJSFuzzer is more sufficient.

4.6 RQ4: Ablation Study

The test input model mutation and test input tensor mutation are two key components of DLJSFuzzer. In this section, we conduct an ablation study to explore the contribution of these two components to bug detection in JavaScript DL frameworks. Specifically, we set up three baselines named $DLJSFuzzer_m$, $DLJSFuzzer_t$, and $DLJSFuzzer_n$. In $DLJSFuzzer_m$, test input tensor mutation is disabled, and all test input tensors are generated randomly. In $DLJSFuzzer_t$, test input model mutation is disabled, and all models are initial seed models (introduced in Section 3.3.1). In $DLJSFuzzer_n$, both test input tensor mutation and test input model mutation are disabled. The number of bugs detected by DLJSFuzzer and all baselines are shown in Table 6. The meanings of each column in Table 6 are the same as those in Table 4. From the table, it can be seen that compared to other baselines, DLJSFuzzer detects the most crashes (21) and NaN & Inconsistency bugs (126). Compared to $DLJSFuzzer_n$, $DLJSFuzzer_m$ detects more crashes (12) and NaN & inconsistency bugs (44), demonstrating the effectiveness of test input model mutation. In particular, due to the overly simple model structure, $DLJSFuzzer_t$ and $DLJSFuzzer_n$ do not successfully trigger any framework bugs. However, when test input tensor mutation and test input model mutation act together, the test input tensor mutation will exert its effect. Specifically, DLJSFuzzer outperforms $DLJSFuzzer_m$, indicating that test input tensor mutation can further improve the bug detection effectiveness of the method based on test input model mutation.

Table 6: Number of Unique Detected Bugs in Ablation Study

Method	Crash	NaN & Inconsistency
$DLJSFuzzer$	21	126
$DLJSFuzzer_m$	12	44
$DLJSFuzzer_t$	0	0
$DLJSFuzzer_n$	0	0

Answer to RQ4: In the ablation study, DLJSFuzzer outperforms $DLJSFuzzer_m$, and $DLJSFuzzer_m$ outperforms $DLJSFuzzer_n$. This result shows our test input model mutation is effective, and the test input tensor mutation can further improve the bug detection effectiveness together with test input model mutation.

5 VALIDITY & THREAT

In this section, we will analyze the threats to validity in this paper, including internal threats, external threats, and construct threats.

The internal threats lie in the experimental configuration. On one hand, to reduce the impact of experimental randomness, DLJSFuzzer generates over 100 models for each dataset respectively. On the other hand, in order to investigate whether the design of tensor generation and model generation in DLJSFuzzer contributes to the improvement of framework testing effectiveness, we conducted an ablation experiment to explore the effects of each component of DLJSFuzzer (see Section 4.6).

The external threats lie in the choice of browsers, and the tested JavaScript DL framework and optimization strategies. Firstly, DLJSFuzzer tests one JavaScript DL framework, TensorFlow.js. Although there are other JavaScript DL frameworks available (e.g., WebDNN [18], etc.), to our knowledge, TensorFlow.js is currently the only popular and actively maintained JavaScript DL framework. The core idea of DLJSFuzzer is also applicable to testing other JavaScript DL frameworks. Secondly, the choice of browsers may threaten the validity of our method. To mitigate the impact of browser differences on experimental results, we repeatedly executed the model under two widely used browsers, Microsoft Edge and Google Chrome, until the inference results were consistent. Thirdly, tested optimization strategies may threaten the validity of our method. DLJSFuzzer can also be generalized to other optimizations (e.g., Dead Code Elimination) by applying more tensor and model mutation rules.

The construct threats lie in the root cause analysis and the selection of baselines. In the root cause analysis, DLJSFuzzer identifies crashes and analyzes their root causes by manually reviewing execution logs. To minimize the impact of subjective factors on the experimental results, two researchers independently review the execution logs. If the analyses of the root cause of a crash by the two researchers are inconsistent, a third expert will determine the final root cause. In addition, DLJSFuzzer further analyzes the root cause of all confirmed bugs (see our Github for more details). In the selection of baselines, as a model-based DL framework testing method, DLJSFuzzer adopts state-of-the-art model-based baselines (e.g., Muffin [14], Gandalf [29], etc.), and does not adopt interface-based baselines (e.g., FreeFuzz [42], DeepREL [8], etc.). In fact, existing interface-based baselines do not consider the unique optimization mechanisms in JavaScript DL frameworks, and therefore cannot effectively detect JavaScript DL framework bugs.

6 RELATED WORKS

6.1 Testing DL Frameworks & Compilers

DL frameworks and compilers serve as the underlying support for DL applications. Their quality has attracted widespread attention from researchers. Testing methods for various DL frameworks and compilers have been proposed. Some methods aim to detect quality issues in DL compilers such as TVM [5], with representative works like Nnsmith [30], TVMfuzz [38], MTDLComp [43], Neuri [31]. They only target conventional optimization mechanism in general DL compilers (e.g., TVM) and do not include JavaScipt-specific inference acceleration strategies or their unique implementations. Therefore, they cannot effectively detect bugs in TensorFlow.js. Some other methods focus on detecting quality issues in common DL frameworks such as TensorFlow and PyTorch. Among them, Audee [16] is one of the early representative works that introduce differential testing into DL framework testing. Predoo [45] is the first to test precision bugs in DL frameworks (e.g., NaN & Inconsistency). FreeFuzz [42] improves the test oracle of differential testing. DeepREL [8] is the first to reveal the equivalence relationship between operators and operator combinations in DL models. Cradle [35] proposes using DL models as test inputs. Graph-Fuzz [32]

designs the overall workflow of a generation-based model generation method. LEMON [41] designs the overall workflow of a mutation-based model generation method. Based on Graph-Fuzz and LEMON, researchers have designed various model generation methods to improve the diversity of test input models. Among them, Muffin [14] further refines the generation-based DL framework testing, expanding the model search space from simple model structures to model parameters. Gandalf [29] inherits and improves the mutation-based model generation method, introducing various RL techniques (e.g., Q-learning [7], DQN [17], DDQN [40]) to make the generated models more diversified. Ramos [47] designs a hierarchical heuristic model mutation method. COMET [28] introduces heuristic guidance in mutation-based methods, improving the diversity of generated models. However, the above methods fail to consider the resource scarcity in JavaScript environment and corresponding optimization mechanisms, so it is not suitable for detecting JavaScript DL frameworks. Our approach follows the overall workflow of mutation-based DL framework testing, and designs tensor mutation rules and model mutation rules targeting the optimization mechanisms in JavaScript DL frameworks. In conclusion, our method expands the applicability of mutation-based DL framework testing methods to the JavaScript environment.

6.2 Bugs in DL Frameworks

As a significant threat to the quality of DL applications, bugs in DL frameworks have received extensive attention from researchers in recent years. Zhang [46] conducts an empirical study on user coding errors in TensorFlow-based applications. Based on Zhang's work, M.J. [21, 22] studies five popular DL frameworks focusing on bugs such as incorrect parameters, invalid structures, and API abuse. Hummatova [20] focuses on the DL framework bugs related to the training process. Zhong [24, 25] studies the internal bugs of TensorFlow. Du [9] explores triggering conditions to classify DL framework bugs, and then analyzes frequency distribution and evolution characteristics of different bug types. Furthermore, Feng [44] summarizes the classification, symptoms, root causes, fix methods, and fix costs of DL framework bugs at the source code level, then divides DL framework bugs into 11 categories, symptoms into five categories, and root causes into 13 categories.

In recent years, JavaScript DL applications in web browsers have been increasingly popular [3, 13, 19]. Despite there are few studies on bugs in JavaScript DL frameworks, the quality issues of JavaScript DL frameworks have attracted widespread attention from researchers. Among them, Guo [15] conducts an empirical study to explore the prediction accuracy and performance of trained models when they are transferred from PCs to web browsers. Ma [33] identifies the performance gaps between DL in browsers and on native platforms by comparing the performance of TensorFlow.js and TensorFlow in Python. Quan [36] collects 700 real-world faults from JavaScript-based DL systems and analyzes their fault symptoms, root causes, and fix patterns, respectively. The above study summarizes the characteristics and root causes of bugs in JavaScript DL frameworks, providing inspiration for our design of JavaScript DL framework testing methods.

7 CONCLUSION

In this work, we propose DLJSFuzzer, which is the first to target JavaScript DL optimization mechanisms in framework testing. To comprehensively test components associated with cache reuse in JavaScript DL frameworks, DLJSFuzzer designs 13 tensor mutation rules to generate new test input tensors with flexible shapes and data types. Additionally, to generate new models that effectively trigger the inference acceleration mechanism, DLJSFuzzer designs eight model mutation rules targeting three inference acceleration strategies, including Node Optimization, Operator Reordering, and Operator Fusion. We evaluate DLJSFuzzer to test the most widely used JavaScript DL framework, TensorFlow.js. Experimental results show that DLJSFuzzer successfully detects 21 unique crashes and 126 unique NaN & Inconsistency bugs. Furthermore, DLJSFuzzer's model generation and bug detection efficiency have increased by more than 47% and 91%, respectively. In the future, we will design testing methods for DL frameworks targeting more optimization mechanisms in the JavaScript environment.

ACKNOWLEGEMENT

We would like to thank anonymous reviewers for their insightful comments. This work is supported partially by the National Natural Science Foundation of China (61932012, 62372228), and Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 265–283.

[2] Bileschi, S., Nielsen, E., and Cai, S. *Deep Learning with JavaScript: Neural Networks in TensorFlow. js.* Simon and Schuster, 2020.

[3] Bileschi, S., Nielsen, E., and Cai, S. *Deep Learning with JavaScript: Neural networks in TensorFlow. js.* Simon and Schuster, 2020.

[4] Chen, L., Li, S., Bai, Q., Yang, J., Jiang, S., and Miao, Y. Review of image classification algorithms based on convolutional neural networks. *Remote Sensing 13*, 22 (2021), 4712.

[5] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), pp. 578–594.

[6] Chowdhary, K. Natural language processing. *Fundamentals of Artificial Intelligence* (2020), 603–649.

[7] Clifton, J., and Laber, E. *Q-learning: Theory and Applications.* Annual Reviews, 2020.

[8] Deng, Y., Yang, C., Wei, A., and Zhang, L. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 44–56.

[9] Du, X., Sui, Y., Liu, Z., and Ai, J. An empirical study of fault triggers in deep learning frameworks. *IEEE Transactions on Dependable and Secure Computing* (2022).

[10] Fang, Y., and Li, J. A review of tournament selection in genetic programming. In *International Symposium on Intelligence Computation and Applications* (2010), Springer, pp. 181–192.

[11] Flanagan, D. *JavaScript: The Definitive Guide: Activate Your Web Pages.* " O'Reilly Media, Inc.", 2011.

[12] Goel, A., Ruamviboonsuk, V., Netravali, R., and Madhyastha, H. V. Rethinking client-side caching for the mobile web. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications* (2021), Association for Computing Machinery, p. 112–118.

[13] Goh, H.-A., Ho, C.-K., and Abas, F. S. Front-end deep learning web apps development and deployment: A review. In *Applied Intelligence* (2023), Springer, pp. 15923–15945.

[14] Gu, J., Luo, X., Zhou, Y., and Wang, X. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference*

on Software Engineering (2022), pp. 1418–1430.

[15] Guo, Q., Chen, S., Xie, X., Ma, L., Hu, Q., Liu, H., Liu, Y., Zhao, J., and Li, X. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering* (2019), IEEE, pp. 810–822.

[16] Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y., Li, X., and Shen, C. Audee: Automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), IEEE, pp. 486–498.

[17] Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., et al. Deep q-learning from demonstrations. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2018).

[18] Hidaka, M., Kikura, Y., Ushiku, Y., and Harada, T. Webdnn: Fastest dnn execution framework on web browser. In *Proceedings of the 25th ACM International Conference on Multimedia* (2017), ACM, pp. 1213–1216.

[19] Hidaka, M., Miura, K., and Harada, T. Development of javascript-based deep learning platform and application to distributed training, 2017.

[20] Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., and Tonella, P. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 1110–1121.

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

[22] Islam, M. J., Pan, R., Nguyen, G., and Rajan, H. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering* (2020), IEEE, pp. 1135–1146.

[23] Jia, F., Jiang, S., Cao, T., Cui, W., Xia, T., Cao, X., Li, Y., Zhang, D., Ren, J., Liu, Y., et al. Accelerating in-browser deep learning inference on diverse edge clients through just-in-time kernel optimizations. *arXiv preprint arXiv:2309.08978* (2023).

[24] Jia, L., Zhong, H., Wang, X., Huang, L., and Lu, X. An empirical study on bugs inside tensorflow. In *International Conference on Database Systems for Advanced Applications* (2020), Springer, pp. 604–620.

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

[26] Ko, H., Lee, S., Park, Y., and Choi, A. A survey of recommendation systems: Recommendation models, techniques, and application fields. *Electronics 11*, 1 (2022), 141.

[27] Laborde, G. *Learning TensorFlow. js.* " O'Reilly Media, Inc.", 2021.

[28] Li, M., Cao, J., Tian, Y., Li, T. O., Wen, M., and Cheung, S.-C. Comet: Coverage-guided model generation for deep learning library testing. *ACM Transactions on Software Engineering and Methodology 32*, 5 (2023), 1–34.

[29] Liu, J., Huang, Y., Wang, Z., Ma, L., Fang, C., Gu, M., Zhang, X., and Chen, Z. Generation-based differential fuzzing for deep learning libraries. *ACM Transactions on Software Engineering and Methodology 33*, 2 (2023), 1–28.

[30] Liu, J., Lin, J., Ruffy, F., Tan, C., Li, J., Panda, A., and Zhang, L. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023), pp. 530–543.

[31] Liu, J., Peng, J., Wang, Y., and Zhang, L. Neuri: Diversifying dnn generation via inductive rule inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023).

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

[33] Ma, Y., Xiang, D., Zheng, S., Tian, D., and Liu, X. Moving deep learning into web browser: How far can we go? In *The World Wide Web Conference* (2019), WWW, p. 1234–1244.

[34] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems 32* (2019).

[35] Pham, H. V., Lutellier, T., Qi, W., and Tan, L. Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering* (2019), IEEE, pp. 1027–1038.

[36] Quan, L., Guo, Q., Xie, X., Chen, S., Li, X., and Liu, Y. Towards understanding the faults of javascript-based deep learning systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2023), IEEE.

[37] Rivera, J. D. D. S. *Practical TensorFlow. js: Deep Learning in Web App Development.* Springer, 2020.

[38] Shen, Q., Ma, H., Chen, J., Tian, Y., Cheung, S.-C., and Chen, X. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), ACM, p. 968–980.

[39] Smilkov, D., Thorat, N., Assogba, Y., Nicholson, C., Kreeger, N., Yu, P., Cai, S., Nielsen, E., Soegel, D., Bileschi, S., et al. Tensorflow. js: Machine learning for the web and beyond. In *Proceedings of Machine Learning and Systems* (2019), pp. 309–321.

[40] Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2016).

[41] Wang, Z., Yan, M., Chen, J., Liu, S., and Zhang, D. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 788–799.

[42] Wei, A., Deng, Y., Yang, C., and Zhang, L. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 995–1007.

[43] Xiao, D., LIU, Z., Yuan, Y., Pang, Q., and Wang, S. Metamorphic testing of deep learning compilers. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2022), ACM, pp. 1–28.

[44] Yang, Y., He, T., Xia, Z., and Feng, Y. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology 151* (2022), 107004.

[45] Zhang, X., Sun, N., Fang, C., Liu, J., Liu, J., Chai, D., Wang, J., and Chen, Z. Predoo: Precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021), pp. 400–412.

[46] Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., and Zhang, L. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), pp. 129–140.

[47] Zou, Y., Sun, H., Fang, C., Liu, J., and Zhang, Z. Deep learning framework testing via hierarchical and heuristic model generation. *Journal of Systems and Software 201* (2023).

[48] Zou, Z., Chen, K., Shi, Z., Guo, Y., and Ye, J. Object detection in 20 years: A survey. *Proceedings of the IEEE 111*, 3 (2023), 257–276.