Regression Fuzzing for Deep Learning Systems

Hanmo You
College of Intelligence and Computing
Tianjin University
Tianjin, China
youhanmo@tju.edu.cn

Zan Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
wangzan@tju.edu.cn

Junjie Chen†
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Shuang Liu
College of Intelligence and Computing
Tianjin University
Tianjin, China
shuang.liu@tju.edu.cn

Shuochuan Li
College of Intelligence and Computing
Tianjin University
Tianjin, China
lishuochuan@tju.edu.cn

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00019

***Abstract*—Deep learning (DL) Systems have been widely used in various domains. Similar to traditional software, DL system evolution may also incur regression faults. To find the regression faults between versions of a DL system, we propose a novel regression fuzzing technique called DRFuzz, which facilitates generating inputs that trigger diverse regression faults and have high fidelity. To enhance the diversity of the found regression faults, DRFuzz proposes a diversity-oriented test criterion to explore as many faulty behaviors as possible. Then, DRFuzz incorporates the GAN model to guarantee the fidelity of generated test inputs. We conduct an extensive study on four subjects in four regression scenarios of DL systems. The experimental results demonstrate the superiority of DRFuzz over the two compared state-of-the-art approaches, with an average improvement of 1,177% and 539% in terms of the number of detected regression faults.**

***Index Terms*—Regression, Fuzzing, Deep Learning**

I. Introduction

Deep learning (DL) systems have been widely used in various domains, including safety-critical domains, such as autonomous driving [1], medical diagnosis [2], and face recognition [3]. Like traditional software [4]–[6], DL systems also evolve to satisfy new requirements or fix known faults. For instance, Apple Face-ID has been updated to a new version in iOS 14.5 so as to allow face recognition with masks on, which brings huge convenience in the COVID-19 pandemic [7]. However, DL system evolution may also incur regression faults in the process. In July 2021, after the Face Unlock function of Galaxy S10 5G was updated to a new version, users failed to authenticate themselves regardless of the lighting condition. This is a DL regression fault, which affects Samsung's reputation and causes economic losses [8]. Such faults are harmful but are easy to be concealed by the overall performance improvement of the DL system. After fixing known faults in the prior version of a DL model, developers usually run test data to check whether the overall performance of the new DL model is improved, but overlook the risks incurred by the evolution, e.g., originally passed test inputs fail in the new version [9], which hinders the detection of regression faults in DL systems.

Detecting regression faults refers to finding test inputs that make the current version produce wrong predictions but make the prior version produce correct predictions. That is, regression faults are caused by (minor) evolution, and detecting them should carefully mine the (minor) difference between versions. Several regression fuzzing techniques have been proposed for traditional software, which can clearly locate code changes in software evolution [10] and utilize them to guide the regression fuzzing process. However, they are not applicable to DL systems, which consist of a large number of neurons connected with weighted links and do not have explicit logical structures. The differences brought by evolution are significant for DL systems if we measure neuron changes (e.g., changes of weights between neurons), but the new version tends to behave similarly to the prior version if we measure the overall performance of the DL system, e.g., precision and recall for classification models. Unlike code changes that tend to affect only a small portion of code (relevant to the changed code), neuron changes usually affect a large portion of neuron links and thus cannot precisely reflect the behavior difference between versions of DL systems. The significant difference between traditional software and DL systems indicates that regression fuzzing techniques specific to DL systems are definitely desired.

In recent years, some fuzzing techniques, such as DeepHunter [11], [12], have been proposed to detect faults in DL systems. However, these techniques aim to detect faults in a specific version of the DL system rather than regression faults. DiffChaser [13] is the most related technique, which aims to detect disagreements between the original model and its quantized model. It just considers whether a test input is generated near the decision boundary of one version for disagreement detection but ignores the differences between versions. As claimed by the existing work [14], [15], many fault-triggering inputs actually fool the DL model with very high confidence. That results in the poor effectiveness of

†Junjie Chen is the corresponding author.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00019

DiffChaser in detecting DL regression faults (confirmed by our study in Section IV-B1). Moreover, Diffchaser ignores the diversity and fidelity of generated input, further limiting its practical use. In particular, it was just proposed for the quantization scenario without exploring general regression scenarios. To sum up, it is necessary to elaborately design a regression fuzzing technique by capturing the (minor) difference due to evolution, which can complement existing fuzzing techniques (e.g., DeepHunter and DiffChaser) for sufficient quality assurance of DL systems.

In this work, we propose a novel regression fuzzing technique for DL systems called **DRFuzz**. Ideally, such a technique is required to satisfy three criteria, which also correspond to three technical challenges:

- *Fault-triggering*: The generated test inputs by DRFuzz should reveal as many regression faults as possible, and this is the core criterion. To achieve this goal, DRFuzz utilizes the prediction difference between versions, rather than neuron changes or the decision boundary of one version, to capture the influence brought by evolution. Such information is more direct in revealing regression faults and does not rely on white-box information, making it more efficient. By incorporating the feedback from existing generated inputs and designing a series of mutation rules, DRFuzz guides the process of generating new test inputs toward the direction of amplifying the prediction difference between versions.
- *Diversity*: Revealing diverse regression faults is more helpful in improving the quality of DL systems. DRFuzz measures the diversity of regression faults from two dimensions: 1) the initial seeds used for generating fault-triggering test inputs and 2) the faulty behaviors produced by the fault-triggering test inputs (to be defined in Section III-A). To improve the diversity of detected regression faults, DRFuzz designs an adaptive seed maintenance strategy, which dynamically adjusts the selection probabilities of seeds once a fault-triggering test input is generated.
- *Fidelity*: It is important to ensure the fidelity of each fault-triggering test input (i.e., preserving original semantics and looking natural to humans) since such test inputs are more concerned by developers. Indeed, test inputs with low fidelity are easy to fool DL systems, but they are not considered within the scope of the DL systems. To obtain test inputs with high fidelity, DRFuzz designs a GAN-based fidelity assurance method, which first learns a discriminator via GAN to discriminate natural inputs with synthetic inputs precisely and then utilizes it to ensure that the generated test inputs resemble natural inputs in the fuzzing process.

We conducted an extensive study on four subjects under four different regression scenarios (i.e., supplementary training, adversarial training, model fixing, and model pruning, which will be introduced in Section IV-A2). We compared DRFuzz with two state-of-the-art techniques, and the experimental results show that DRFuzz outperforms them with an average improvement of 539% on DeepHunter [11] and 1,177% on DiffChaser [13] in terms of the number of detected regression faults. Also, DRFuzz shows consistently stable performance across different subjects and regression scenarios. Moreover, fine-tuning with regression-fault-triggering inputs generated by DRFuzz can averagely fix 82.90%, 71.72%, and 71.48% regression faults triggered by inputs generated by DRFuzz, DeepHunter, and DiffChaser, respectively, and outperforms compared techniques. The results further demonstrate the value of DRFuzz in improving model quality.

In summary, we make the following major contributions:

- We take the first step in formalizing the concepts for regression testing of DL systems.
- We propose a heuristic-based fuzzing framework that facilitates detecting regression faults with high diversity.
- We propose a GAN-based fidelity assurance technique, which assists in generating test inputs with high fidelity.
- We have conducted an extensive study on four subjects under four regression scenarios, and the results demonstrate the effectiveness of our proposed approach.

II. Definition

The regression process of Deep Learning Systems can be formally defined as below:

Definition 1. *Regression. Given a Deep Learning (DL) model $\mathcal{M}_1$. The regression process on $\mathcal{M}_1$ is defined as improving $\mathcal{M}_1$ through re-training/fine-tuning the model or directly changing the neuron weights to obtain $\mathcal{M}_2$, which performs better than $\mathcal{M}_1$ on some measurements (such as accuracy, robustness, or efficiency). We refer to $\mathcal{M}_1$ as the prior version model and $\mathcal{M}_2$ as the regression model of $\mathcal{M}_1$.*

Given an N-class classification model $\mathcal{M}$ and an input set $\mathcal{X}$, the prediction process of $\mathcal{M}$ maps an input $x \in \mathcal{X}$ to an N-dimensional vector $\tilde{C}_{\mathcal{M}}[x]$, containing the confidence on each category. The category with the highest confidence in $\tilde{C}_{\mathcal{M}}[x]$ is denoted as $c_{\mathcal{M}}[x]$ and is considered as the prediction result, which is formally defined in formula (1).

$$c_{\mathcal{M}}[x] = \arg\max \tilde{C}_{\mathcal{M}}[x] \tag{1}$$

Definition 2. *Regression fault. Given a prior version model $\mathcal{M}_1$, its regression model $\mathcal{M}_2$, a test input x, and its ground truth label y, x is called to trigger a regression fault if x is correctly predicted by $\mathcal{M}_1$, yet wrongly predicted by $\mathcal{M}_2$, i.e., $c_{\mathcal{M}_1}[x] = y \wedge c_{\mathcal{M}_2}[x] \neq y$. In addition, we denote x as the regression fault triggering input and the faulty behavior triggered by x as $(c_{\mathcal{M}_1}[x] \rightarrow c_{\mathcal{M}_2}[x])$.*

III. APPROACH

We propose a novel regression fuzzing technique called DRFuzz, and Figure 1 depicts the overview of DRFuzz. DRFuzz selects a seed input each time to conduct mutation, and the mutated inputs are passed into the GAN-based Fidelity Assurance model to identify inputs with high fidelity, which are then executed by both the original model and the regression model. If the two models provide different prediction results, then a regression-fault-triggering input is identified. The inputs

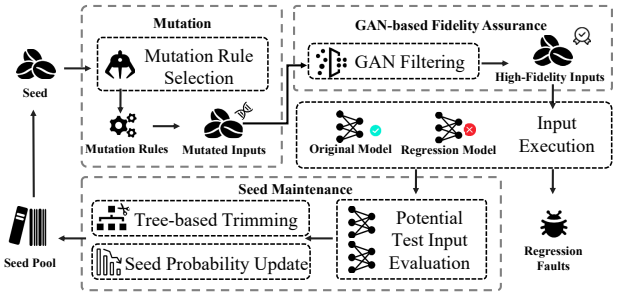


Fig. 1. The framework of DRFuzz

that do not trigger regression faults will be maintained with carefully designed seed maintenance strategies in order to improve the diversity of the inputs.

In this Section, we provide detailed descriptions of how to tackle the three challenges presented in Section I. In particular, we discuss the diversity measurement in Section III-A, the GAN-based *fidelity* assurance technique in Section III-B, and lastly, introduce the heuristic-based regression fuzzing process, which facilities diverse *fault-triggering*, in Section III-C.

A. Diversity Measurement

It is important to generate as many fault-triggering test inputs as possible. However, if the generated inputs trigger duplicate faults, it would limit the capability of improving the quality of DL systems. Therefore, it is crucial to reveal diverse regression faults. However, it is hard to determine the root cause of each regression fault, and thus inspired by the existing work [11], [16], DRFuzz measures the diversity of regression faults from two aspects, i.e., the initial seeds used for generating fault-triggering test inputs and the faulty behaviors produced by the fault-triggering test inputs. Please note that initial seeds refer to those seeds added to the corpus before the fuzzing process starts.

The first aspect considers static information of the fuzzing process, i.e., the initial seeds. The intuition is that if two fault-triggering inputs are obtained from different initial seeds, then they are more likely to trigger different regression faults [11], [17], [18]. Formally, suppose two fault-triggering test inputs ta_i and tb_j are generated by conducting a series of mutations on initial seeds ta_0 and tb_0, i.e., $(ta_0 \rightarrow \ldots \rightarrow ta_{i\text{-}1} \rightarrow ta_i)$ and $(tb_0 \rightarrow \ldots \rightarrow tb_{j\text{-}1} \rightarrow tb_j)$, respectively. If ta_0 and tb_0 are different, then ta_i and tb_j are more likely to trigger different regression faults.

The second aspect reflects the dynamic behavior of a regression-fault-triggering test input, which complements the static initial seed information to assist in better diversification of faults. The intuition is that if two fault-triggering inputs show different faulty behaviors, then they are more likely to trigger different regression faults [16], [19]. We denote the predicted class for a test input ta_i by the prior version $\mathcal{M}_1$ as $c_{\mathcal{M}_1}[ta_i]$ and the predicted class for ta_i by the current version $\mathcal{M}_2$ as $c_{\mathcal{M}_2}[ta_i]$. Suppose that ta_i and ta_j are two regression-fault-triggering inputs mutated from the same initial seed ta_0, $c_{\mathcal{M}_1}[ta_i] = c_{\mathcal{M}_1}[ta_0] = c_{\mathcal{M}_1}[ta_j]$ and $c_{\mathcal{M}_2}[ta_i] \neq c_{\mathcal{M}_1}[ta_0] \neq c_{\mathcal{M}_2}[ta_j]$, it indicates that ta_i and ta_j displays different faulty behaviors $(c_{\mathcal{M}_1}[ta_i] \rightarrow c_{\mathcal{M}_2}[ta_i])$ and $(c_{\mathcal{M}_1}[ta_j] \rightarrow c_{\mathcal{M}_2}[ta_j])$, then they are likely to trigger different regression faults.

Based on the static initial seeds and the dynamic faulty behaviors for fault-triggering test inputs, DRFuzz adopts the tuple $[ta_0, (c_{\mathcal{M}_1}[ta_i] \rightarrow c_{\mathcal{M}_2}[ta_i])]$ to approximately distinguish the fault triggering inputs and uses the tuple diversity to represent the diversity of the fault-triggering inputs.

B. GAN-based Fidelity Assurance

It is also important to ensure the fidelity of fault-triggering test inputs since it is easy for test inputs with low fidelity to fool a DL system but are not concerned by developers due to being out of the scope of the DL system [20]. Therefore, test inputs with high fidelity are more meaningful for regression fuzzing in practice. However, conducting *iterative* mutations on an initial seed is likely to produce test inputs with low fidelity, and thus it is important for DRFuzz to reserve test inputs with high fidelity from a number of mutated test inputs.

Some methods, e.g., SSIM [21] and Euclidean Distance [22], have been proposed to measure the fidelity of test inputs. SSIM uses the contrast and brightness values derived from pixels to measure the structural similarity of images, while Euclidean Distance directly uses pixel values to measure image similarity. Both of them highly rely on the pixel-wise difference between images and are not applicable to the test inputs generated with some mutation operators, e.g., image scaling, that intensively change pixels but retain nearly all semantic information. Therefore, those measurements may discard a large number of test inputs with high fidelity and thus limit the capability of detecting test inputs triggering different faulty behaviors.

To mitigate the limitations of existing methods, we adopt Generative Adversarial Network (GAN) for fidelity assurance. GAN is composed of a Generator network and a Discriminator network. During the training process, the Generator network synthesizes fake images based on randomly initialized vectors. It sends generated images to the Discriminator network to distinguish the given fake inputs from their corresponding original inputs in the training set. GAN trains Discriminator and Generator in alternating periods. The purpose is to improve the performance of both models until synthesized images are indistinguishable and the Discriminator can filter out most out-of-scope synthesized images. The Discriminator will be used to assure the fidelity of test inputs generated by DRFuzz, as it considers semantic similarity [1] rather than strict pixel-wise difference.

In DRFuzz, we adopt the DCGAN model [23] for the following reasons: 1) It improves classic GAN on model structure, in which deep convolution structure facilitates better feature representation and enables the Generator to generate fake inputs with high fidelity. [23], [24]. Thus, the Discriminator is able to distinguish high-fidelity fake inputs. 2) It has been demonstrated to be efficient in prediction and

TABLE I
MUTATION RULES

Pixel-Level Mutation Rule	Description	Image-Level Mutation Rule	Description
Pixel Adding Gaussian Noise	Adds noise on pixels following Gaussian distribution	Image Scaling	Amplifies or shrinks the size of the image
Pixel Adding Salt & Pepper Noise	Converts the color of certain pixels into black or white	Image Translation	Moves the pixels of the image for a certain distance
Pixel Adding Multiplicative Noise	Multiplies noise on pixels following Gaussian distribution	Image Shearing	Moves the pixels horizontally while fixing vertical coordinates
Patch Coloring Black	Blocks the effect of a patch by coloring it black	Image Rotating	Rotates the image for a certain angle
Patch Coloring White	Blocks the effect of a patch by coloring it white	Image Brightness Adjustment	Adjusts the brightness to simulates the illumination changes
Patch Color Reverse	Reverses the color of a patch for max opposite influence	Image Contrast Adjustment	Modifies contrast value, the inter-pixel brightness of the image
Patch Shuffling	Shuffles the pixels to break interrelations of adjacent pixels	Image Blurring	Smoothies the color transition in the image
		Image Dilation	Expanding certain regions to change contents shape
		Image Erosion	Removing the edgy pixels to shrink the size of certain region

training processes compared with more complex GAN-based models. That is, it brings less overhead to the overall fuzzing process [25].

Specifically, DRFuzz first pre-processes the training set to normalize each pixel value into the range of $[0, 1]$, which accelerates the convergence process and improves training performance. We use the training set of model $\mathcal{M}_1$ as the training set of DCGAN. During the training process, the Generator will synthesize an image from a random initialized vector. The Discriminator takes the synthesized image, which is labeled 0 (indicating fake), and the image in the training set, which is labeled 1 (indicating real), as input. For each given input, the well-trained Discriminator can output a score in the range of $[0, 1]$, where a higher score indicates the given input resembles the training set with higher fidelity. During the fuzzing process, we feed an initial seed into the Discriminator to get a fidelity score, which serves as the fidelity threshold for filtering out the low-fidelity inputs generated from this initial seed.

C. Heuristic-based Regression Fuzzing Process

Since the input space for a DL system is enormous, and regression faults are more concealed, it is quite inefficient to find fault-triggering test inputs by randomly exploring the entire space. To improve the regression fuzzing process, we expect that in each iteration, the new test input generated through mutation can produce a larger prediction difference (measured by the prediction probability difference in each class) between versions than that produced by the test input before mutation. When the prediction difference is accumulated to a large value, models of different versions could make different predictions, which indicates that a regression fault is triggered by the generated test input. With this expectation, we design a search strategy in DRFuzz to guide the generation of test inputs toward the direction of amplifying the prediction difference between versions. In this way, fault-triggering test inputs can be generated more efficiently. More specifically, the heuristic-based regression fuzzing process in DRFuzz contains two major steps: 1) mutation rule selection (to be presented in Section III-C1) and 2) maintenance of the seed pool (to be presented in Section III-C2). We follow the standard high-level fuzzing process [11], [26], but carefully design specific steps in DRFuzz to cater to the DL regression fuzzing task.

1) Mutation: Following the existing work [11], [27], DRFuzz conducts mutation on existing seed inputs to generate new test inputs.

Mutation Rules. In DRFuzz, we adopt all the mutation rules for images designed by the existing work [11], [27], [28], as these mutation rules change a given test input from different perspectives and can increase the possibility of generating test inputs triggering diverse regression faults. In total, we implemented 16 mutation rules in DRFuzz. Besides generating diverse fault-triggering test inputs, another criterion for mutation rules is to preserve the semantics of a test input after mutation. That is, we should carefully design the usage of each mutation rule in DRFuzz. We observe that some mutation rules preserve image semantics by changing only a small set of pixels in an image, while some other mutation rules have to be applied to the whole image to avoid the violation of image semantics. According to the usage of each mutation rule, we classify them into two categories, i.e., *Pixel-Level Mutation* and *Image-Level Mutation*.

Pixel-Level Mutation aims to change a small set of pixels in an image to generate a new test input. It achieves the goal of preserving the semantics of a test input by making slight pixel-wise changes on the whole image. DRFuzz randomly selects a small set of pixels (i.e., 0.5% total pixels) for mutation. Image-Level Mutation aims to change the whole image to generate a new test input. Intuitively, it is more likely to preserve the semantics of a test input by changing it as slightly as possible, but this category of mutation rules can make a test input become unrealistic if changing only a part of it. For example, scaling/rotating only a part of an image can produce an image with low fidelity than directly scaling/rotating the whole image. That is, this category of mutation rules can simulate real-world scenarios more effectively when applying them to the whole image. The detailed mutation rules are listed in Table I.

Mutation Rule Selection. Different mutation rules change the test input by different degrees and thus induce the seed towards triggering different faulty behaviors. Intuitively, if a mutation rule can frequently generate test inputs with high fidelity and amplify the prediction difference towards becoming a regression fault, it should be selected more frequently in the fuzzing process. Therefore, we design a reward function to calculate the priority score for each mutation rule, which is formally defined in Formula (2). *#DiffTriggerInputs* refers to the number of generated inputs that trigger prediction difference between the two versions of models, and *#FidelInputs* refer

to the number of generated high-fidelity inputs. The reward function is designed with two goals, i.e., the probability of a mutation rule to bring a large prediction difference between versions and the probability of a mutation rule to generate test inputs with high fidelity. Then, DRFuzz ranks mutation rules based on their rewards in descending order.

$$Reward = \frac{\#DiffTriggerInputs}{\#TotalSelect} \times \frac{\#FidelInputs}{\#TotalSelect} \quad (2)$$

Please note that it is not tenable to only select mutation rules with the highest rewards since it may lead to generating test inputs with similar behaviors. Hence, all mutation rules should have a certain probability of being selected, while mutation rules with higher priority in the ranking should have larger chances of being selected. That is, the current mutation rule selection is affected by the historical behaviors of mutation rules, which is a typical Markov Chain [29]. Thus, we can model the mutation rule selection process as a Markov Chain Monte Carlo (MCMC) problem [30], which is used to sample from a probability distribution by constructing a Markov chain converging to the desired distribution. Specifically, DRFuzz adopts the Metropolis-Hastings (MH) algorithm [31] to guide mutation rule selection. Based on the proposal distribution, it selects a new mutation rule $\mathcal{R}_A$ according to the current mutation rule $\mathcal{R}_B$. Suppose all mutation rules can be ranked by the score defined in formula (2), the possibility of selecting mutation rule $\mathcal{R}_A$ according to a given mutation rule $\mathcal{R}_B$ is calculated by the ranks, which are obtained according to the reward of $\mathcal{R}_A$ and $\mathcal{R}_B$. This is formally defined in formula (3), where p is the probability of being successful in one Bernoulli trial in geometric distribution used for approximating the desired distribution following the existing MCMC framework [18], [32], k_a and k_b are ranks of $\mathcal{R}_A$ and $\mathcal{R}_B$, respectively. Due to the space limit, more details on MCMC can refer to the existing work [18], [32].

$$P(\mathcal{R}_A|\mathcal{R}_B) = min(1, (1-p)^{k_a - k_b}) \quad (3)$$

2) Seed Maintenance: The initial seed pool is the set of test inputs used for testing the prior version of the DL system. Since our goal is to detect regression faults, DRFuzz conducts pre-processing to filter out the test inputs that make the prior version produce wrong predictions. To facilitate the effectiveness of generating diverse regression faults, it is crucial to maintain a high-quality seed pool containing two aspects, i.e., exploring more diverse initial seeds and triggering more diverse faulty behaviors, according to the diversity measurement defined in Section III-A. Therefore, we propose three techniques to facilitate seed pool maintenance. In particular, we 1) conduct potential input evaluations to preserve those inputs that contribute to fault-triggering; 2) propose seed probability update to select inputs that are more likely to trigger regression faults; and 3) propose tree-based seed pool trimming to remove redundant seeds.

Potential Test Input Evaluation. We carefully evaluate the potential of generated inputs based on the diversity measurement presented in Section III-A. In general, we want to maximize the number of different tuples in the seed pool. Therefore, we filter the mutated inputs in each generation and put those that change intensively towards non-ground-truth categories in the seed pool. Specifically, given the i_{th} generation of mutated test inputs and a regression model, if the prediction probability on a non-ground-truth category of an input changes intensively compared with the $(i-1)_{th}$ generation, it means this input is prone to trigger new faulty behaviors, thus should be put into the seed pool for high-order mutation.

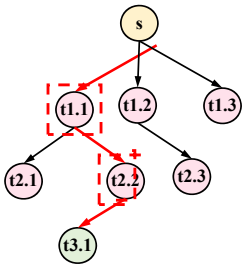


Fig. 2. Model Mutation Process as a Tree

Seed Probability Update. Intuitively, the mutated inputs, which are more likely to change the predicted category compared with their previous generations, should have more chances of being selected for high-order mutation. To achieve this goal, we propose a seed probability update strategy (shown in (4a)) to lower the probability of a test input being selected as its time of selection increases (controlled by N_s) so that the newly generated inputs are more likely to be selected. To improve the diversity of initial seeds, which is an important aspect of our diversity measurement, we also need to consider the generation of each input to limit the chance of inputs with the younger generation being selected. This is controlled by hyper-parameter N_g in formula (4a).

Specifically, We adopt the exponential decay function, a widely used decay strategy, shown in equation (4a), to adjust the selection possibility for each new seed. N_g is the generation of the input, i.e., the number of mutations from the initial seed to the current input, and N_s is the number of times the seed is selected for mutation. The weights w_1 and w_2 are used to balance the influence of N_g and N_s, and we empirically set w_1 to be 1 and w_2 to be 0.5. P_{init} is the maximum possible rate, and P_{finish} is the minimal possible rate after decaying max_times times. max_times is the maximum number of times a seed can be mutated, which can limit the number of similar seeds and avoid significant changes causing semantic deviations from the initial seed to some degree. α is called the exponential decay constant, which can affect the speed of the decay process; it is calculated according to the formula (4b).

$$P = e^{-\alpha(w_1 \times N_g + w_2 \times N_s)} P_{init} \quad (4a)$$

$$\alpha = \frac{1}{max\_times} ln(\frac{P_{init}}{P_{finish}}) \quad (4b)$$

Tree-based Seed Pool Trimming. The Trimming process aims to trigger more diverse faulty behaviors by removing redundant seeds, which potentially generate redundant faulty-behavior-triggering inputs, from the seed pool. The mutation process can be naturally modeled as a tree since, in each generation, multiple mutated inputs are obtained based on different mutation rules. As illustrated in Figure 2, the root is the initial seed selected from the seed pool. Each node of the tree represents a high-fidelity test input generated through fuzzing, and each branch represents a mutation process conducted on the input represented by its parent node. The brother nodes are inputs mutated from the same seed with different first-order mutation rules.

If a leaf node, e.g., *t3.1*, triggers a regression fault and can be represented by $[s, (c_{\mathcal{M}_1}[t3.1] \rightarrow c_{\mathcal{M}_2}[t3.1])]$, the trimming algorithm traverses the mutation trace from the root node to the leaf node, i.e., $s \rightarrow$ *t1.1* $\rightarrow$ *t2.2* $\rightarrow$ *t3.1*. Then DRFuzz carefully inspects if each seed represented by the corresponding node along the trace has the potential to trigger a new faulty behavior. Specifically, we estimate the fault-triggering potential of the seed by the second highest confidence in the prediction vector, as the category with the highest confidence for *t1.1* and *t2.2* must be different from that for *t3.1*. If the second highest confidence class in the predication vectors of *t1.1* and *t2.2* on the regression model is equal to $c_{\mathcal{M}_2}[t3.1]$, it means that *t1.1* and *t2.2* are more likely to trigger the same faulty behavior as that triggered by *t3.1*, and thus the seeds on the trace should be removed from the seed pool. For efficiency considerations, we trim the tree on the trace level.

Please note that if all kinds of faulty behaviors that an initial seed could potentially trigger have been explored, then all seeds originating from that initial seed should be removed from the seed pool. If we are targeting an N-class classification task, then a seed could potentially trigger $N-1$ faulty behaviors, in which the prediction results on the seed input deviate from its ground truth label.

IV. Evaluation

We address the following research questions (RQs).

- **RQ1**: How does DRFuzz perform in detecting regression faults?
- **RQ2**: Does each main component contribute to the overall effectiveness of DRFuzz?
- **RQ3**: Can DRFuzz help improve the quality of DL systems in regression scenarios?

A. Experimental Setup

1) Subjects: To sufficiently evaluate the effectiveness of DRFuzz, we used four pairs of datasets and DNN models as subjects, i.e., LeNet-5 on MNIST, VGG16 on CIFAR10, AlexNet on Fashion-MNIST (FM), and ResNet18 on SVHN, which have been widely used in the existing studies on DL testing [17], [33]–[36]. Specifically, MNIST is a 10-class handwritten digit dataset [37]. CIFAR10 is a 10-class ubiquitous object recognition dataset [38]. Fashion-MNIST (FM) is a 10-class dataset of Zalando's article images [39]. SVHN is a 10-class street view house number dataset [40]. Note that all the models adopted in our subjects are different, and they involve different degrees of model complexity (e.g., LeNet-5 consists of 7 layers with 236 neurons, while ResNet18 consists of 54 layers with 13,066 neurons).

TABLE II
Experiment of DRFuzz.

Scenario	$\mathcal{M}_1$ Acc.	$\mathcal{M}_2$ Acc.	Project
SUPPLY	85.87%	97.83%	MNIST-LeNet5
SUPPLY	87.67%	87.88%	CIFAR10-VGG16
SUPPLY	89.33%	90.34%	FM-AlexNet
SUPPLY	88.85%	91.93%	SVHN-ResNet18
ADV:BIM	98.07%	97.50%	MNIST-LeNet5
ADV:BIM	87.92%	87.51%	CIFAR10-VGG16
ADV:BIM	91.70%	90.96%	FM-AlexNet
ADV:BIM	92.05%	91.90%	SVHN-ResNet18
ADV:CW	98.07%	98.30%	MNIST-LeNet5
ADV:CW	87.92%	88.00%	CIFAR10-VGG16
ADV:CW	91.70%	91.87%	FM-AlexNet
ADV:CW	92.05%	92.01%	SVHN-ResNet18
FIXING	98.07%	98.12%	MNIST-LeNet5
FIXING	87.92%	88.40%	CIFAR10-VGG16
FIXING	91.70%	92.90%	FM-AlexNet
FIXING	92.05%	92.10%	SVHN-ResNet18
PRUNE	98.07%	98.12%	MNIST-LeNet5
PRUNE	87.92%	76.27%	CIFAR10-VGG16
PRUNE	91.70%	91.54%	FM-AlexNet
PRUNE	92.05%	91.00%	SVHN-ResNet18

* Please note that in SUPPLY scenario, we use 80% of the training set to train $\mathcal{M}_1$, and for the rest of the regression scenarios, we use the entire training set to train $\mathcal{M}_1$, so that the accuracy of $\mathcal{M}_1$ of SUPPLY scenario is a bit lower than that of other scenarios.

2) Regression Scenarios: According to the definition of Regression for DL systems in Section II. we studied four typical regression scenarios, i.e., supplementary training, adversarial training, white-box model fixing, and model pruning, to evaluate the effectiveness of DRFuzz. Among the four regression scenarios, the first two belong to the evolution method of fine-tuning the prior version of the DL model, while the remaining two belong to the method of directly changing the neuron weights of the prior version. Table II presents the basic information of the models of regression scenarios. Column 1 presents the regression scenarios, and Columns 2-3 present the accuracy of the prior version model and the accuracy of the regression model, respectively. Column 4 presents the dataset-model pair of each subject.

Supplementary Training (denoted as SUPPLY), which fine-tunes the model by incorporating more new training data and then producing a new version of the model, is a widely-used black-box means of improving the model accuracy in practice. In our study, for each subject, we simulate this scenario by sampling 80% of the training data in the subject to train the prior version of the model and then using the remaining 20% of the training data to fine-tune the prior version and obtain the new version of the model. Specifically, we fine-tune the prior version for 20 epochs and use the version with the highest accuracy.

Adversarial Training (denoted as ADV) is proposed to improve the robustness of a model against adversarial attacks, which fine-tunes the model by incorporating adversarial inputs and then produces a new version of the model. In our study, for each subject, we simulate this scenario by sampling 5,000 training inputs in the subject to generate the same amount of adversarial inputs with an existing adversarial input generation method. Then, we use these adversarial inputs to fine-tune the prior version and obtain the new version of the model. In particular, we adopted two widely-used adversarial input generation methods, i.e., BIM (Basic Iterative Method) [41] and C&W (Carlini & Wagner) [42], in our study. Hence, this scenario can be further divided into Adversarial Training – BIM (denoted as ADV:BIM) and Adversarial Training – C&W (denoted as ADV:CW). Specifically, we conduct adversarial training for 20 epochs and use the version with the highest robustness and minimum accuracy decrease.

White-Box Model Fixing (denoted as FIXING) is a white-box way of improving the accuracy of a DNN model. Unlike black-box supplementary training, it improves the model accuracy by directly modifying the weights of the model. A self-maintained state-of-the-art white-box model fixing method is APRICOT [43], which divides the training set into several subsets, each of which is used to build a sub-model, and then the weights of the model are adjusted based on the weights of these sub-models. In this way, a new version of the model with greater accuracy can be produced without the fine-tuning process. In our study, we adopted APRICOT to construct the white-box model fixing scenario for evaluation.

Model Pruning (denoted as PRUNE) aims to shrink the volume of the model and thus improve the prediction efficiency through pruning unimportant parts in the DL model. There are several model pruning tools in the literature [44]–[46]. In this study, we adopted one of the most widely-used tools, i.e., KerasSurgeon [46], to construct this scenario for evaluation. Specifically, it prunes unimportant channels of the last convolution layer in the model, producing a new version of the model.

Please note that the quantization scenario targeted by DiffChaser is not a regression scenario since quantization is conducted once-for-all without the characteristics of continuous evolution for regression. Hence, we do not consider it in our study.

3) Compared Approaches: Since our work targets the problem of detecting regression faults, we selected the compared approaches according to three criteria: 1) The approach has been officially published; 2) The approach was proposed or evaluated to detect differences between two versions of models, e.g., a DL model and its quantized version; 3) The approach has been open-sourced. Based on the first two criteria, we identified DiffChaser [13], DeepEvolution [47], DeepHunter [11], and Tensorfuzz [48] as candidates. However, DeepEvolution is not open-sourced, and Tensorfuzz has been demonstrated to underperform DeepHunter [11], and thus we finally selected DiffChaser and DeepHunter as the compared approaches.

TABLE III
OVERALL EFFECTIVENESS OF DRFUZZ.

Approach	Average				Improvement (%)			
	#RFI	#RF	#Seed	#GF	#RFI	#RF	#Seed	#GF
DiffChaser	19,750	1,235	911	27,391	204	1,177	638	722
DeepHunter	4,830	2,468	1,601	25,367	1,130	539	320	787
DRFuzz	**59,420**	**15,763**	**6,725**	**225,098**	-	-	-	-

* Columns 2-5 present the number of averages results on all the subject on each metric, and Columns 6-9 present the average improvement of DRFuzz over each compared approach on each metric.

DiffChaser [13] is a search-based testing framework aiming to find disagreements in the predictions between a DL model and its quantized version. Following the configurations of DiffChaser, we select the fitness function to find inputs close to the boundaries of the top two classes to facilitate finding disagreements in our experimentation. DeepHunter [11] is a fuzzing framework for a DL model guided by structural coverage [49], such as neuron coverage. It is initially designed for fuzzing a specific version of the model, but it has also been evaluated to find disagreements between a DL model and its quantized version. We use KMNC (K-Multisection Neuron Coverage) [50] with k = 1,000 following the configuration in DeepHunter.

4) Measurements: To measure the effectiveness of a regression fuzzing approach, we adopt *the number of regression-fault-triggering test inputs* (denoted as #RFI) and *the number of regression faults detected by fault-triggering test inputs* (denoted as #RF), similar to the existing work [11], [13], [16]. As defined in Section III-A, RF refers to the number of [initial seed, faulty behavior] tuples corresponding to the regression-fault-triggering test inputs. Following the existing work [11], [51], we also measure *the number of initial seeds for fault-triggering test inputs* (denoted as #Seed) as another effectiveness metric.

Indeed, detecting regression faults is the core goal of a regression fuzzing approach, but the compared approach, DeepHunter, is proposed for fuzzing one version. Hence, it is also interesting to compare them in terms of *the number of detected general faults* (denoted as #GF) during the fuzzing process. Specifically, a detected general fault refers to a fault in the current version of a model, regardless of triggering a fault in the prior version or not.

Please note that in our study, we ran each approach on each subject in each scenario for 24 hours and then measured the effectiveness of each approach in terms of the above metrics. The same testing period can ensure a fair comparison among those approaches.

5) Implementations and Experiments: We implement DRFuzz on Keras 2.3.1 and Tensorflow 1.15.0 and adopt the existing implementations of compared approaches released in their work [11], [13]. Regarding mutation rules, we follow the configurations proposed in their original work [11], [27]. For pixel level mutation, we select a small portion of pixels, i.e., 0.5% of the total number of pixels, to mutate and patch size of 2×2 to preserve the semantic. Also, for the seed probability update process, we set P_{init} to be 1 and P_{finish} to be 0.05,

TABLE IV
EFFECTIVENESS ON DIFFERENT REGRESSION SCENARIOS

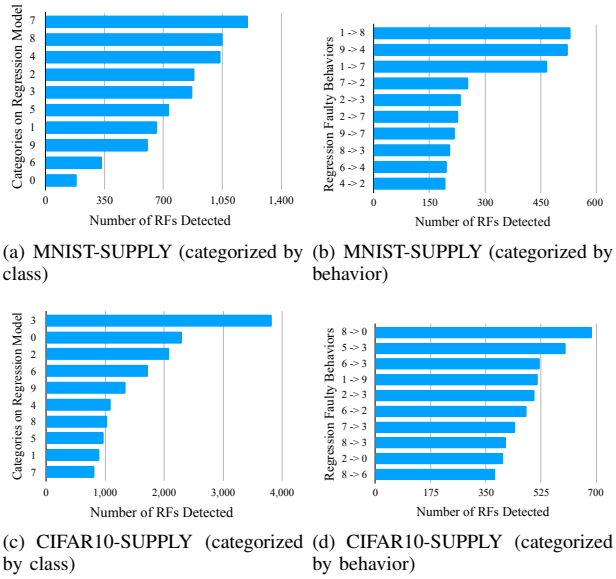
Regression Scenario	Approach	#RFI	#RF	#Seed	#GF
SUPPLY	DiffChaser	12,489	991	846	18,529
	DeepHunter	3,450	1,832	1,402	26,854
	DRFuzz	**43,265**	**13,391**	**6,272**	**207,917**
ADV	DiffChaser	7,543	514	417	15,366
	DeepHunter	4,319	2,196	1,422	25,290
	DRFuzz	**45,620**	**13,545**	**6,198**	**252,035**
FIXING	DiffChaser	14,066	1,172	859	20,036
	DeepHunter	3,850	2,362	1,608	19,202
	DRFuzz	**76,555**	**19,359**	**7,267**	**228,039**
PRUNE	DiffChaser	56,211	2,983	2,015	67,656
	DeepHunter	8,210	3,752	2,152	30,200
	DRFuzz	**86,040**	**18,975**	**7,690**	**185,464**

max_times to be 10, w_1 to be 1 and w_2 to be 0.5. Besides, for each fuzzing iteration, we set the number of test case generation $Batch\_Size$ to be 10 according to the preliminary study on a small dataset, and observe that the above settings are generally effective.

We conduct our experiment on a machine with Intel (R) Xeon (R) CPU E5-2640 v4, 40 cores, 2.40 GHz, 125Gb RAM, and running on Ubuntu 18.04. The code and extra experimental results will be found on the project homepage: https://github.com/youhanmo/DRFuzz.

B. Experiment Results

1) Effectiveness of DRFuzz: **Overall Effectiveness**: The overall effectiveness results of DRFuzz are presented in Table III. Due to the limited space and large amounts of subjects used in the study, we put detailed results on each subject on our project homepage. From Table III we can observe that DRFuzz performs the best on average across all subjects and scenarios in terms of all metrics, with significant superiority over DiffChaser and DeepHunter. The average number of RFI and RF detected by DRFuzz outperforms DiffChaser with an average improvement of 204% and 1,177%, respectively, and outperforms DeepHunter with an average improvement of 1,130% and 539%, respectively. Moreover, considering the number of initial seeds for fault-triggering inputs (Seed) explored by each approach, DRFuzz covers 6,725 initial seeds, while DiffChaser and DeepHunter only cover 911 and 1,601 initial seeds, respectively. DRFuzz achieves an average improvement on covered initial seeds by 638% and 320% compared with them. The results demonstrate the effectiveness of DRFuzz. Please note that DeepHunter is designed to detect fault-triggering inputs in the current version (i.e., GF), but DRFuzz still outperforms DeepHunter with an improvement of 787%. The possible reason lies in that DeepHunter utilizes neuron coverage to guide the fuzzing process, but the existing studies have demonstrated neuron coverage is not strongly correlated with fault detection capability [52]–[54]. However, DRFuzz utilizes the prediction results to guide fault detection, which is more explicit and straightforward for fault detection.
Effectiveness on Different Scenarios Table IV shows the effectiveness of DRFuzz on different regression scenarios.



(a) MNIST-SUPPLY (categorized by class) (b) MNIST-SUPPLY (categorized by behavior)

(c) CIFAR10-SUPPLY (categorized by class) (d) CIFAR10-SUPPLY (categorized by behavior)

Fig. 3. Per Class/Behavior #RF of DRFuzz.

We can observe that DRFuzz outperforms the compared approaches stably on all the regression scenarios in terms of various metrics. In particular, the regression faults triggered (#RF) by DRFuzz have a 1,468.6% improvement over DiffChaser and a 568.3% improvement over DeepHunter, across all the scenarios. The improvements on explored initial seeds (#Seed) are 763.8% and 323.1%, respectively. The results further demonstrate the stable effectiveness of DRFuzz in various regression scenarios.
Regression Explication based on DRFuzz We further focus on utilizing regression faults (#RF) to explain the bias and risks induced in the regression process. For clarity of illustration, we select and report the results on subjects of MNIST-LeNet5 (Figure 3(a) and Figure 3(b)) and CIFAR10-VGG16 (Figure 3(c) and Figure 3(d)) on the supplementary training scenario. From Figure 3(a), we can observe that Classes 7, 8, and 4 contain the most regression faults, which means that the regression process may induce overfitting in these classes. Figure 3(b) shows that inputs of class 1 are wrongly predicted as class 8, inputs of class 9 are wrongly predicted as class 4, and inputs of classes 1, 2, and 9 tend to be predicted as 7 due to the regression process.

The results of CIFAR10-VGG16 on the supplementary training scenario (Figure 3(c) and Figure 3(d)) show a similar trend, where class 3 (cat), 0 (airplane), and 2 (automobile) are the top-3 classes showing regression faults. Figure 3(d) shows that class 8 (ship) is likely to be wrongly predicted as 0 (airplane), and animal-related classes such as 5 (dog), 6 (frog), and 2 (bird) are easily misclassified as 3 (cat). The prediction results and the faulty behaviors facilitate explaining the bias (such as overfitting) in each class to judge if the regression process contributes to data relevance.

The effectiveness results show that DRFuzz outperforms

the state-of-the-art approaches consistently across different datasets, model structures, and regression scenarios. The results produced by DRFuzz facilitate explaining the regression behaviors, assisting DL developers to better understand the risks and bias induced through regression.

2) Contribution of the main component of DRFuzz: DRFuzz comprises three major components, i.e., mutation, GAN-based fidelity assurance, and seed maintenance. Therefore, we conduct ablation experiments to measure the contribution of each component. In particular, we selected four subject-scenario pairs, i.e., MNIST-SUPPLY, CIFAR10-ADV:CW, FM-FIXING, and SVHN-PRUNE, which considers representative dataset-model pairs and regression scenarios combinations. We compare DRFuzz with three variants of DRFuzz, i.e., DRFuzz_r, DRFuzz_NG, and DRFuzz_NSM, representing DRFuzz with the random mutation rule selection strategy (which replaces the MCMC-guided mutation rule selection strategy), DRFuzz without GAN-based fidelity assurance, and DRFuzz without seed maintenance technique. We ran DRFuzz and each variant for 24 hours to calculate the results.

As shown in Table V, DRFuzz_r detects 24.3% and 13.1% fewer RFIs and RFs compared with DRFuzz. The results demonstrate that DRFuzz can generate more diverse regression faults than DRFuzz_r across all the used subjects. The reason is that DRFuzz_r can not guide amplifying the confidence difference in prediction results and thus ignores the test inputs that have the potential to become regression faults. The results confirm the contribution of the MCMC-guided mutation rule selection strategy. DRFuzz_NSM detects 47.3% and 56.8% fewer RFIs and RFs compared with DRFuzz. The results show that the Seed Maintenance technique effectively contributes to detecting more regression faults. DRFuzz_NG can detect an average of 18.5% and 27.8% more RFIs and RFs compared with DRFuzz. Recall that we propose the GAN-based Fidelity Assurance technique to filter out inputs with low fidelity, which are out-of-scope but can trigger regression faults. We further manually evaluate the fidelity of inputs obtained with and without the fidelity assurance mechanism. Specifically, we sample 100 inputs generated by DRFuzz_NG and DRFuzz, respectively, and two authors manually evaluate their fidelity individually. The Cohen's Kappa coefficient between their evaluation is 0.67. They label 86% of inputs generated by DRFuzz as high-fidelity and only 57.5% of inputs generated by DRFuzz_NG as high-fidelity. This indicates that the GAN-based Fidelity Assurance technique can filter out more than 20% of fault-triggering inputs with low fidelity. We find that the low-fidelity inputs generated by DRFuzz_NG can be categorized into three categories, i.e., blurry inputs, noisy inputs, and over-changed inputs (presented in Figure 4). That is, DRFuzz drops some fault-triggering inputs compared with DRFuzz_NG to achieve a balance between fault detection effectiveness and input fidelity.

3) Model Quality Improvement based on DRFuzz: In RQ3, we investigate the value of the regression faults detected by DRFuzz in improving model quality. That is, we try to fix the regression faults induced through regression by fine-tuning the current regression model $\mathcal{M}_2$ to obtain an improved model $\mathcal{M}_2'$. Please note that, like regression testing in traditional software, we only consider fixing the regression faults induced from the original model (i.e., $\mathcal{M}_1$) to the current regression model (i.e., $\mathcal{M}_2$). Following experimental settings in the existing work [55], [56], to avoid data leakage, we randomly split test inputs into two parts following the ratio of 1:9, then we fine-tune the $\mathcal{M}_2$ model with regression-fault-triggering inputs generated from 10% seeds on one approach for ten epochs and evaluate its effectiveness on regression-fault-triggering inputs generated from 90% seeds by all compared approaches. During fine-tuning, we ensure the accuracy of models should not decline intensively and evaluate whether $\mathcal{M}_2'$ can fix regression faults triggered by inputs generated by each approach.

TABLE V
ABLATION EXPERIMENT RESULTS

	#RFI	#RF	#seed	#GF
DRFuzz	70,093	16,464	6,942	231,675
DRFuzz_r	53,037	14,309	6,523	185,354
DRFuzz_NG	83,042	21,044	7,748	279,544
DRFuzz_NSM	36,936	7,109	3,239	136,723

(a) Blurry (b) Noisy (c) Over-changed

Fig. 4. Inputs generated by DRFuzz (on the left of each figure) and DRFuzz_NG (on the right of each figure)

Table VI shows the results of quality improvement on different regression scenarios. Overall, fine-tuning on regression-fault-triggering inputs generated by DRFuzz can fix 77.72%∼87.03% of regression faults generated by DRFuzz, 52.26%∼80.68% by DiffChaser and 66.63%∼79.88% by DeepHunter. Fine-tuning on DeepHunter can only fix 53.99%∼64.12% of regression faults generated by DRFuzz, and 55.13%∼71.84% by DiffChaser. Fine-tuning on Diffchaser can only fix 48.52%∼58.84% of regression faults generated by DRFuzz, and 49.62%∼60.39% by DeepHunter. Besides, the largest decrement in the accuracy of the regression model by fine-tuning on DRFuzz is just 0.12%, while those by fine-tuning on DiffChaser and DeepHunter are 2.30% and 1.38%, respectively. That further demonstrates the stable value of DRFuzz in fixing regression faults. Overall, the regression faults detected by DRFuzz can effectively fix the regression faults induced in the regression model on different scenarios with the training accuracy preserved.

In addition, fine-tuning on regression faults generated by DRFuzz can not only fix regression faults generated by DRFuzz (i.e., fixing an average of 82.90% of regression faults generated by DRFuzz) but also fix most of the regression faults generated by compared approaches. Fine-tuning on DRFuzz does not outperform DiffChaser and DeepHunter in some

TABLE VI
RETRAINING ACCURACY ON DIFFERENT REGRESSION SCENARIOS

Scenario	Train\Test	DiffChaser	DeepHunter	DRFuzz	$\Uparrow_{Acc}$(%)
SUPPLY	DiffChaser	67.11%	49.62%	53.35%	-0.97%
	DeepHunter	61.97%	72.83%	60.13%	-0.06%
	DRFuzz	**73.25%**	**74.09%**	**84.98%**	**0.34%**
ADV:CW	DiffChaser	72.96%	60.39%	58.84%	0.39%
	DeepHunter	71.84%	75.25%	64.12%	0.66%
	DRFuzz	**80.68%**	**79.88%**	**87.03%**	**0.81%**
ADV:BIM	DiffChaser	**77.47%**	50.39%	55.70%	-0.25%
	DeepHunter	64.13%	**68.43%**	58.50%	**0.04%**
	DRFuzz	76.87%	67.64%	**83.23%**	-0.04%
FIXING	DiffChaser	**64.25%**	50.70%	48.52%	-2.30%
	DeepHunter	55.13%	65.02%	53.99%	-1.38%
	DRFuzz	52.26%	**66.63%**	**77.72%**	**-0.12%**
PRUNE	DiffChaser	**75.61%**	55.55%	53.46%	3.66%
	DeepHunter	63.84%	**76.10%**	59.74%	3.95%
	DRFuzz	74.35%	70.37%	**81.53%**	**4.04%**

* $\Uparrow_{Acc}$ shows the average Accuracy improvement from the regression model to the model fine-tuned by regression fault-triggering inputs.

cases where the training set and test set for the fine-tuning process are generated by the same technique. This is because such training and test data are more similar, thus making the used technique perform better. The overall results further demonstrate that regression faults generated by DRFuzz are more diverse and can subsume a large portion of regression faults generated by compared approaches.

V. DISCUSSION

A. Generality of DRFuzz

Our study has demonstrated the effectiveness of DRFuzz in the image domain. Howbeit, DRFuzz can actually be applied to more domains (e.g., text, volume) since the three main components of DRFuzz are all extendable. The mutation process of DRFuzz can be extended to a new domain as long as the mutation rules of a new domain are well-designed. As presented in Section III-C, designing mutation rules shares the same high-level idea, i.e., slightly changing the input data using widely-used mutation operators for generating adversarial examples in the corresponding domains. It is easy to extend DRFuzz to a new domain since we can easily find adversarial attack methods for the corresponding domain (e.g., Word replacement Attack for the text domain [57], Noise Attack for the audio domain [58]). The GAN-based Fidelity Assurance technique, which requires training a Discriminator using the training set to evaluate the fidelity of test inputs, can also be extended to more domains since it is easy to acquire the training set and widely-used GAN structures suitable for the domain [59], [60]. Besides, the seed maintenance strategies are also adaptable as they are not related to the input format. Therefore, our approach can be applied to domains other than image classification models.

B. Diversity Analysis

To further analyze the diversity of regression-fault-triggering test inputs generated by DRFuzz, we sample a portion from the inputs generated by DRFuzz and conduct manual analysis to check if the diversity measurement used in DRFuzz (defined in Section III-A) actually facilitates triggering diverse faulty behaviors. To assist the visual analysis, we use the Gradient-weighted Class Activation Map (heatmap) [61] to visualize the importance of each feature in the prediction process. Figure 5 presents heatmaps of four regression faults mutated from the same initial seed (with ground truth label 1) and are recognized as 8 and 9, respectively, by the regression model. The important features of Figure 5(a) and Figure 5(b) for faulty behavior $(1 \rightarrow 8)$ lie in the center of the images. The important features of Figure 5(c) and Figure 5(d) for faulty behavior $(1 \rightarrow 9)$ lie on the left of the images. The phenomena above show that different faulty behaviors may be caused by different features, further indicating that diversity criteria based on faulty behaviors (proposed in Section III-A) can reflect the diversity of fault-triggering test inputs to some extent.

(a) (1->8)(1) (b) (1->8)(2) (c) (1->9)(1) (d) (1->9)(2)

Fig. 5. An example from MNIST:SUPPLY for Qualitative Analysis

C. Threats to Validity

Internal Threats to Validity lie in the implementation of DRFuzz, the compared approaches, and the experiment scripts. To mitigate the threats, we adopt widely-used libraries to implement DRFuzz and carefully check the code and experimental scripts. For compared approaches, we use the implementation released by authors and carefully follow their parameter settings.

External Threats to Validity lie in the subjects studied in the evaluation. To reduce the threat of subject selection, we select plenty of models, datasets, and various regression scenarios in our evaluation. Although we mainly focus on classification tasks, we provide detailed analysis in Section V-A to prove that DRFuzz can be easily generalized to other tasks, e.g., regression and recommendation.

Construct Threats to Validity lies in the measurement used in our study and randomness incurred in experiments. We follow the existing work [11] to measure the diversity of seeds and extend the diversity metric to regression faults (section III-A). For the randomness that may incur, we conduct a preliminary study on MNIST-LeNet5, where we repeat the experiment four times, and the Coefficient of variation in terms of each metric (i.e., #RFI, #RF, #Seed, and #GF) range from 0.31%∼0.67% for DiffChaser, 2.19%∼9.65% for DeepHunter and 3.78%∼4.84% for DRFuzz. All the Coefficient of variation in terms of each metric is lower than 10%, which proves the stability of each approach. Moreover, we run the experiment on each configuration setting for 24 hours, which is sufficiently long to eliminate the effect of randomness.

VI. RELATED WORK

A. Deep Learning Testing

The statistical nature of deep learning systems makes it hard to be sufficiently tested [62], [63]. Testing for DL systems has been explored from code [64], [65], model [49], [50], [66]–[68], and library [69], [70] perspectives. Nowadays, many researchers have been working on proposing testing frameworks to expose the vulnerabilities of deep learning systems. Guo et al. [71] proposed DLFuzz, which considers confidence changing and neuron coverage to guide the fuzzing process. Wang et al. [72] proposed RobOT, which automatically generates test cases to improve model robustness. Sun et al. [73] proposed DeepConcolic, leveraging the execution of concrete inputs and symbolic analysis to synthesize new test inputs. Ma et al. [74] proposed a mutation-based framework DeepMutation, which mutates the model from the source-code level and the model level to further explore the weakness in the model. Ma et al. [75] proposed DeepCT based on the insight of combinatorial testing to build an LP-constraint-solving-based test generator. Du et al. proposed DeepCruiser [76] and DeepStellar [77], which model RNN as Markov Decision Process to generate semantic-preserved test cases. These works were designed and evaluated specifically for a single version of a model, while DRFuzz considers the difference between the original model and regression models to facilitate generating regression-fault-triggering test inputs.

B. Regression Fuzzing

Fuzzing is a technique for detecting bugs through automatically generated test inputs. Fuzzing is widely used to test a variety of different programs, such as JVM [78], [79], compilers [80], [81] and smart contracts [82]. For traditional software, fuzzing is used to generate inputs that can trigger unexpected behaviors. The fuzzing process is usually guided using structural coverage to measure the exploration of programs. The insights of fuzzing traditional software are also inherited to test deep learning systems, which include fuzzing for DNNs [11], [12], deep learning libraries [83] and deep learning compilers [80]. Fuzzing is proved to be effective for input generation and defect localization.

Regression testing ensures that changes made through software evolution do not impact the previously working functionality [5]. It usually involves optimization techniques such as test case prioritization [6], [84], selection [85], and reduction [86] to reduce the cost, effort, and time taken to perform regression testing. Different from the above works, regression fuzzing focuses on fuzzing on the changes in software systems and amplifying the impact of changes to detect regression bugs induced through each regression process [87]. Multiple regression fuzzing approaches on traditional software have been proposed to boost the effectiveness and efficiency of fuzzing, such as differential analysis [88] and symbolic execution [89]. DRFuzz proposes a novel approach to tackle the problem of regression fuzzing of deep learning systems and conduct large-scale experiments with diverse regression scenarios and test subjects.

VII. CONCLUSION

In this work, we propose a regression fuzzing technique, DRFuzz, to generate regression-fault-triggering inputs with high fidelity and diversity. It adopts the MCMC strategy to select mutation rules that are prone to generating fault-triggering test inputs and proposes a GAN-based fidelity assurance method to ensure the fidelity of the generated inputs. Also, DRFuzz incorporates tree-based seed pool trimming and seed probability update to maintain the quality of the seed pool and thus increase the diversity of the generated regression faults. We compared DRFuzz with two state-of-the-art approaches to four subjects in four regression scenarios, and the results show that DRFuzz outperforms the compared approaches across all regression scenarios. Moreover, fine-tuning on regression-fault-triggering inputs generated by DRFuzz can fix more regression faults than fine-tuning on compared approaches, which proves that regression faults triggered by inputs generated by DRFuzz can be used to improve the quality of the models effectively. In our future work, we will focus on investigating the influence of both in-distribution and out-of-distribution regression-fault-triggering test inputs generation and further exploring approaches to fix regression faults without inducing new regression faults.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of China Nos. 61872263, 62232001, and 62002256.

REFERENCES

[1] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *ASE*. ACM, 2018, pp. 132–142.

[2] Y. Zhang, J. M. Górriz, and Z. Dong, "Deep learning in medical image analysis," *J. Imaging*, vol. 7, no. 4, p. 74, 2021.

[3] G. Hu, Y. Yang, D. Yi, J. Kittler, W. J. Christmas, S. Z. Li, and T. M. Hospedales, "When face recognition meets with deep learning: An evaluation of convolutional neural networks for face recognition," in *ICCV Workshops*. IEEE Computer Society, 2015, pp. 384–392.

[4] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *ISSTA*. ACM, 2014, pp. 105–115.

[5] Y. Lou, J. Chen, L. Zhang, and D. Hao, "Chapter one - A survey on regression test-case prioritization," *Adv. Comput.*, vol. 113, pp. 1–46, 2019.

[6] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang, "Exploring better black-box test case prioritization via log analysis," *ACM Trans. Softw. Eng. Methodol.*, 2022.

[7] J. Krol, "News," Accessed: 2022. [Online]. Available: https://www.cnn.com/2021/04/27/cnn-underscored/ios-14-5-iphone-update-face-id-mask-apple/index.html

[8] J. Torres, "News," Accessed: 2022. [Online]. Available: https://www.slashgear.com/galaxy-s10-5g-update-reportedly-breaks-face-recognition-26684006/

[9] S. Ma, Y. Liu, W. Lee, X. Zhang, and A. Grama, "MODE: automated neural network model debugging via state differential analysis and input selection," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 175–186.

[10] M. Wen, R. Wu, and S. Cheung, "Locus: locating bugs from software changes," in *ASE*. ACM, 2016, pp. 262–273.

[11] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *ISSTA*. ACM, 2019, pp. 146–157.

[12] S. Demir, H. F. Eniser, and A. Sen, "Deepsmartfuzzer: Reward guided test generation for deep learning," in *AISafety@IJCAI*, ser. CEUR Workshop Proceedings, vol. 2640. CEUR-WS.org, 2020.

[13] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," in *IJCAI*. ijcai.org, 2019, pp. 5772–5778.

[14] A. M. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *CVPR*. IEEE Computer Society, 2015, pp. 427–436.

[15] D. Jakubovitz and R. Giryes, "Improving DNN robustness to adversarial attacks using jacobian regularization," in *ECCV (12)*, ser. Lecture Notes in Computer Science, vol. 11216. Springer, 2018, pp. 525–541.

[16] X. Gao, Y. Feng, Y. Yin, Z. Liu, Z. Chen, and B. Xu, "Adaptive test selection for deep neural networks," in *ICSE*. ACM, 2022, pp. 73–85.

[17] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 30:1–30:35, 2020.

[18] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 788–799.

[19] V. Riccio and P. Tonella, "Model-based exploration of the frontier of behaviours for deep learning system testing," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 876–888.

[20] S. Dola, M. B. Dwyer, and M. L. Soffa, "Distribution-aware testing of neural networks using generative models," in *ICSE*. IEEE, 2021, pp. 226–237.

[21] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, 2004.

[22] K. Elmore and M. Richman, "Euclidean distance as a similarity metric for principal component analysis," *Monthly Weather Review - MON WEATHER REV*, vol. 129, 03 2001.

[23] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," in *ICLR (Poster)*, 2016.

[24] W. Fang, F. Zhang, V. S. Sheng, and Y. Ding, "A method for improving cnn-based image recognition using dcgan," *Computers, Materials and Continua*, vol. 57, no. 1, pp. 167–178, 2018.

[25] H. Yin, Y. Wei, H. Liu, S. Liu, C. Liu, and Y. Gao, "Deep convolutional generative adversarial network and convolutional neural network for smoke detection," *Complex.*, vol. 2020, pp. 6 843 869:1–6 843 869:12, 2020.

[26] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Validity fuzzing and parametric generators for effective random testing," in *ICSE (Companion Volume)*. IEEE / ACM, 2019, pp. 266–267.

[27] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: automated testing of deep-neural-network-driven autonomous cars," in *ICSE*. ACM, 2018, pp. 303–314.

[28] S. B. Tambe, D. Kulhare, M. Nirmal, and G. Prajapati, "Image processing (ip) through erosion and dilation methods," 2013.

[29] C. J. Geyer, "Practical markov chain monte carlo," *Statistical science*, pp. 473–483, 1992.

[30] R. E. Kass, B. P. Carlin, A. Gelman, and R. M. Neal, "Markov chain monte carlo in practice: a roundtable discussion," *The American Statistician*, vol. 52, no. 2, pp. 93–100, 1998.

[31] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[32] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *PLDI*. ACM, 2016, pp. 85–99.

[33] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *ICSE*. IEEE, 2021, pp. 397–409.

[34] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 13:1–13:22, 2021.

[35] Q. Hu, Y. Guo, M. Cordy, X. Xie, L. Ma, M. Papadakis, and Y. L. Traon, "An empirical study on data distribution-aware test selection for deep learning enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 78:1–78:30, 2022.

[36] Y. Zhang, Z. Wang, J. Jiang, H. You, and J. Chen, "Toward improving the robustness of deep learning models via model transformation," in *ASE*. ACM, 2022, pp. 104:1–104:13.

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[38] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[39] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[40] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng, "Reading digits in natural images with unsupervised feature learning," *NIPS*, 01 2011.

[41] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *ICLR (Workshop)*. OpenReview.net, 2017.

[42] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 39–57.

[43] H. Zhang and W. K. Chan, "Apricot: A weight-adaptation approach to fixing deep learning models," in *ASE*. IEEE, 2019, pp. 376–387.

[44] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *ICCV*. IEEE Computer Society, 2017, pp. 2755–2763.

[45] "Tensorflow model optimization toolkit," https://github.com/tensorflow/model-optimization, Accessed:2022.

[46] B. Whetton and SvenWarnke., "keras-surgeon," https://github.com/BenWhetton/keras-surgeon, 2017.

[47] H. B. Braiek and F. Khomh, "Deepevolution: A search-based testing approach for deep neural networks," in *ICSME*. IEEE, 2019, pp. 454–458.

[48] A. Odena, C. Olsson, D. G. Andersen, and I. J. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *ICML*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 4901–4911.

[49] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *SOSP*. ACM, 2017, pp. 1–18.

[50] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: multi-granularity testing criteria for deep learning systems," in *ASE*. ACM, 2018, pp. 120–131.

[51] Q. Shen, J. Chen, J. M. Zhang, H. Wang, S. Liu, and M. Tian, "Natural test generation for precise testing of question answering software," in *ASE*. ACM, 2022, pp. 71:1–71:12.

[52] Z. Li, X. Ma, C. Xu, and C. Cao, "Structural coverage criteria for neural networks could be misleading," in *ICSE (NIER)*. IEEE / ACM, 2019, pp. 89–92.

[53] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim, "Is neuron coverage a meaningful measure for testing deep neural networks?" in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 851–862.

[54] Y. Dong, P. Zhang, J. Wang, S. Liu, J. Sun, J. Hao, X. Wang, L. Wang, J. S. Dong, and D. Ting, "There is limited correlation between coverage and robustness for deep neural networks," *CoRR*, vol. abs/1911.05904, 2019.

[55] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai, "White-box fairness testing through adversarial sampling," in *ICSE*. ACM, 2020, pp. 949–960.

[56] P. Zhang, J. Wang, J. Sun, X. Wang, G. Dong, X. Wang, T. Dai, and J. S. Dong, "Automatic fairness testing of neural classifiers through adversarial sampling," *IEEE Trans. Software Eng.*, vol. 48, no. 9, pp. 3593–3612, 2022.

[57] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *ICSE*. ACM, 2020, pp. 974–985.

[58] R. Taori, A. Kamsetty, B. Chu, and N. Vemuri, "Targeted adversarial examples for black box audio systems," in *IEEE Symposium on Security and Privacy Workshops*. IEEE, 2019, pp. 15–20.

[59] M. Cha, Y. Gwon, and H. T. Kung, "Adversarial nets with perceptual losses for text-to-image synthesis," in *MLSP*. IEEE, 2017, pp. 1–6.

[60] J. B. Harvill, D. Issa, M. Hasegawa-Johnson, and C. D. Yoo, "Synthesis of new words for improved dysarthric speech recognition on an expanded vocabulary," in *ICASSP*. IEEE, 2021, pp. 6428–6432.

[61] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *ICCV*. IEEE Computer Society, 2017, pp. 618–626.

[62] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 1–36, 2022.

[63] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: a systematic mapping," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5193–5254, 2020.
[64] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*. ACM, 2020, pp. 1110–1121.
[65] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: Fault localization for deep neural networks," in *ICSE*. IEEE, 2021, pp. 251–262.
[66] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. IEEE / ACM, 2019, pp. 1039–1049.
[67] H. Converse, A. Filieri, D. Gopinath, and C. S. Pasareanu, "Probabilistic symbolic analysis of neural networks," in *ISSRE*. IEEE, 2020, pp. 148–159.
[68] S. Kang, R. Feldt, and S. Yoo, "SINVAD: search-based image space navigation for DNN image classifier test input generation," in *ICSE (Workshops)*. ACM, 2020, pp. 521–528.
[69] M. Yan, J. Chen, X. Zhang, L. Tan, G. Wang, and Z. Wang, "Exposing numerical bugs in deep learning via gradient back-propagation," in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 627–638.
[70] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen, "Predoo: precision testing of deep learning operators," in *ISSTA*. ACM, 2021, pp. 400–412.
[71] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: differential fuzzing testing of deep learning systems," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 739–743.
[72] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "Robot: Robustness-oriented testing for deep learning systems," in *ICSE*. IEEE, 2021, pp. 300–311.
[73] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Deepconcolic: testing and debugging deep neural networks," in *ICSE (Companion Volume)*. IEEE / ACM, 2019, pp. 111–114.
[74] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," in *ISSRE*. IEEE Computer Society, 2018, pp. 100–111.
[75] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *SANER*. IEEE, 2019, pp. 614–618.
[76] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: model-based quantitative analysis of stateful deep learning systems," in *ESEC/SIGSOFT FSE*. ACM, 2019, pp. 477–487.
[77] X. Du, X. Xie, Y. Li, L. Ma, J. Zhao, and Y. Liu, "Deepcruiser: Automated guided testing for stateful deep learning systems," *CoRR*, vol. abs/1812.05339, 2018.
[78] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. L. Traon, "Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities," in *QRS*. IEEE, 2021, pp. 586–597.
[79] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *ICSE*. ACM, 2022, pp. 1133–1144.
[80] Q. Shen, H. Ma, J. Chen, Y. Tian, S. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 968–980.
[81] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "Fuzzjit: Oracle-enhanced fuzzing for javascript engine jit compiler."
[82] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: an efficient adaptive fuzzer for solidity smart contracts," in *ICSE*. ACM, 2020, pp. 778–788.
[83] X. Zhang, J. Liu, N. Sun, C. Fang, J. Liu, J. Wang, D. Chai, and Z. Chen, "Duo: Differential fuzzing for deep learning operators," *IEEE Trans. Reliab.*, vol. 70, no. 4, pp. 1671–1685, 2021.
[84] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *ICST*. IEEE Computer Society, 2016, pp. 266–277.
[85] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 187:1–187:29, 2019.
[86] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How do assertions impact coverage-based test-suite reduction?" in *ICST*. IEEE Computer Society, 2017, pp. 418–423.
[87] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *CCS*. ACM, 2021, pp. 2169–2182.
[88] Y. Noller, C. S. Pasareanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "Hydiff: hybrid differential software analysis," in *ICSE*. ACM, 2020, pp. 1273–1285.
[89] Y. Noller, "Differential program analysis with fuzzing and symbolic execution," in *ASE*. ACM, 2018, pp. 944–947.