Deep Learning for Coverage-Guided Fuzzing: How Far Are We?

Siqi Li, Xiaofei Xie, Yun Lin, Yuekang Li, Ruitao Feng, Xiaohong Li, Weimin Ge, and Jin Song Dong

Abstract—Fuzzing is a widely-used software vulnerability discovery technology, many of which are optimized using coverage-feedback. Recently, some techniques propose to train deep learning (DL) models to predict the branch coverage of an arbitrary input owing to its always-available gradients etc. as a guide. Those techniques have proved their success in improving coverage and discovering bugs under different experimental settings. However, DL models, usually as a magic black-box, are notoriously lack of explanation. Moreover, their performance can be sensitive to the collected runtime coverage information for training, indicating potentially unstable performance. In this work, we conduct a systematic empirical study on 4 types of DL models across 6 projects to (1) revisit the performance of DL models on predicting branch coverage (2) demystify what specific knowledge do the models exactly learn, (3) study the scenarios where the DL models can outperform and underperform the traditional fuzzers, and (4) gain insight into the challenges of applying DL models on fuzzing. Our empirical results reveal that existing DL-based fuzzers do not perform well as expected, which is largely affected by the dependencies between branches, unbalanced sample distribution, and the limited model expressiveness. In addition, the estimated gradient information tends to be less helpful in our experiments. Finally, we further pinpoint the research directions based on our summarized challenges.

Index Terms—Deep Learning, Testing, Fuzzing, Mutation, Coverage.

1 Introduction

Fuzzing is an indispensable key technology to locate vulnerabilities in binary programs, which has been used in many applications such as network communications [8], web development [10], and browser kernels [11]. Given a binary program, fuzzers feed it with randomly generated inputs. A potential vulnerability is located if any inputs can either cause it to crash [12] or trigger it to violate general specification (e.g., use after free, heap buffer overflow, memory leak, etc.) [13].

Most fuzzers pursue to achieve maximized branch coverage within limited budget, with the assumption that exploring more program branches can potentially lead to the discovery of more vulnerabilities. To this end, many fuzzers adopt various strategies, usually including SE (Symbolic Execution)-based fuzzing [14] and guided fuzzing [16], to cover more new branches. SE-based fuzzing considers the branch coverage as a constraint-solving problem. Given a branch, it transforms the branch conditions into a set of path conditions and leverage SMT solvers (e.g., Z3 [9], CVC4 [17], etc.) to solve the solution as an input. In contrast, guided fuzzing regards the branch coverage problem as an optimization problem. Given a branch and a seed input, it measures how promising the input can cover the branch. Based on the measurement, an evolutionary algorithm (e.g., genetic algorithm [18], hill-climbing algorithm [19], simulated annealing algorithm [20], etc.) is used to select and mutate a seed to optimize the branch coverage. Comparing to SE-based fuzzing which has stronger assumptions on (1) the feasibility of retrieving sophisticated path conditions and (2) the solvability of the path conditions fed to SMT solver, guided fuzzing only requires additional instrumentation to evaluate the measurement. Thus, it is adopted in almost all the fuzzers in practice [1], [21], [22], [23], [24], [34], [35].

As an optimization solution, the effectiveness of guided fuzzing largely depend on the estimated gradients (or feedback) in the fuzzing search space. However, it is challenging to ensure (1) the estimated feedback is available or accurate and (2) the search space is continuous and smooth. For example, for each covered branch, AFL (American Fuzzy Lop) [1] maintains a queue of inputs covering it. As for the feedback, AFL will allocate different computational resources to evolve a test input i by evaluating (1) how likely new the branches i discovers and (2) how efficient i can be executed. Such heuristic-based feedback can miss covering a lot of hard albeit important branches. A increasing number of research is proposed to improve the feedback mechanism during fuzzing (e.g., AFLFast [37], Steelix [25], Angora, Greyone [26]).

Recently, some researchers proposed to adopt the DL (Deep Learning) models to estimate the input-evolving gradients towards branch coverage [6], [32], [33]. Given a dataset of test inputs $\mathbf{X}$, labelled by its coverage information

- *Siqi Li, Xiaohong Li, and Weimin Ge are with College of Intelligence and Computing, Tianjin University, Tianjin, China. E-mail:siqili@tju.edu.cn, xiaohongli@tju.edu.cn, gewm@tju.edu.cn*
- *Yun Lin are with Shanghai Jiao Tong University and National University of Singapore.*
- *Jin Song Dong are with School of Computing, National University of Singapore. E-mail:dcsliny@nus.edu.sg, dcsdjs@nus.edu.sg*
- *Xiaofei Xie is with School of Computing and Information Systems, Singapore Management University, Singapore. E-mail:xfxie@smu.edu.sg*
- *Ruitao Feng and Yuekang Li are with School of Computer Science and Engineering, Nanyang Technological University. E-mail: rtfeng@ntu.edu.sg, yuekang.li@ntu.edu.sg*
- *Yun Lin (dcsliny@nus.edu.sg) and Ruitao Feng (rtfeng@ntu.edu.sg) are the corresponding authors.*

on the set of branches $\mathbf{B}$ in a subject program, a deep learning model is trained to learn a mapping function $\mathbf{f} : \mathbf{X} \rightarrow \mathbf{B}$. Given $b = \mathbf{f}(\mathbf{x})$ where $\mathbf{x} \in \mathbf{X}$, and b is a vector where each dimension $\mathbf{b}_i$ represents whether the branch is covered or not by the input $\mathbf{x}$. Given that neural networks are usually equipped with the property of *continuity and smoothness*, the partial derivative $\frac{\partial \mathbf{f}(b_i)}{\partial \mathbf{x}_i}$ can be naturally used as the feedback towards updating an arbitrary input $\mathbf{x}$ to cover any branches b_i. Neuzz is a representative work of learning the mapping relation from test input to branch coverage, which uses a fully connected neural network as the mapping relation. SampleFuzz [32] and GanFuzz [33] are different from the work of Neuzz and they use LSTM and GAN as the mapping relation, respectively.

More specifically, consider the example in Listing 1, where the rare branch (i.e., the one in line 5) can be successfully covered by Neuzz. In Listing 1, the input consists of two variables a and b. Given the effect of power function (line 1), there is a narrow range of a and b to sample (i.e., $a + b \in (0, \log_3 2)$) to exercise the true branch in line 5. By learning the branch coverage when $a + b$ is in other range (e.g., $(-\infty, 0)$ and $(1, +\infty)$), Neuzz can train a neural network $\mathbf{f}(.)$, extrapolating the relation between the input (i.e., a and b) and the branch coverage. Updating $a \leftarrow a + \eta \frac{\partial \mathbf{f}(\mathbf{b}_5)}{\partial a}$ and $b \leftarrow b + \eta \frac{\partial \mathbf{f}(\mathbf{b}_5)}{\partial b}$ allows us to calculate an input to cover the true branch in line 5, where $\mathbf{b}_5 = 1$ represents the true branch at line 5 is covered.

```
1  z = pow(3, a+b);
2  if(z<1){
3     return 1;
4  }
5  else if(z<2){
6     //vulnerability
7     return 2;
8  }
9  else if(z<4){
10    return 4;
```

Listing 1: An example which can be solved by Neuzz [6].

Novel as those approaches, they use off-the-shelf models (e.g., fully connected network, LSTM, and GAN) in computer vision or natural language processing community to estimate the semantic program runtime behaviors. Although DL-based fuzzing has achieved success [6], [7], it is still unclear how those models perform when fitting the behaviors of non-continuous program operation (e.g., mod, function, checking string-containing, etc.). Moreover, comparing to samples, such as images or sentences in a dataset, the types of program branches are far more diversified. For example, exercising one branch may only require satisfying one constraint (shallow branch) while exercising another can require satisfying over 10 constraints (deep branch). Thus, it is also unclear whether a unified deep learning model can be expressive enough to capture the semantics of all the branches. Last, many inherent issues of deep learning models (e.g., over-fitting, quality of training data, etc.) are still unexplored.

To better understand the effect of deep learning models for the coverage-guided fuzzing, in this work, we conduct a systematic empirical study on four types of deep learning models across six projects with the following research goals:

- *revisit* the performance of various DL models on predicting branch coverage on a large scale of subject programs,
- *demystify* what coverage information the DL models can exactly extrapolate,
- *study* the scenarios where the deep models can outperform and underperform the traditional fuzzers, and
- *gain* insight into the challenges of applying deep models on fuzzing.

The results of our large-scale empirical study reveal that the deep learning models is effective in limited scenarios, which are largely restrained by training data imbalance, dependant relation among branches, and the insufficient expressiveness of the state-of-the-art models. Moreover, we observe that carelessly applying the model gradients can sometimes cause a phenomenon of *contradicting feedback* during the fuzzing. Consequently, the estimated gradients by the models towards covering a branch can be incorrect in many scenarios. Based on our findings, we propose a general guideline of applying DL models on fuzzing, and further pinpoint some research directions based on our summarized challenges.

In summary, we make the following main contributions:

- **Large-scale study.** We conduct a large-scale empirical study on how the deep learning can improve the coverage-guided fuzzers. Specifically, we conduct a systematic comparison 1) among various DL-based fuzzers and 2) across deep-learning and non deep-learning fuzzers. We publish all the experimental settings and results in [47].
- **Findings.** We find that many limitations to apply the off-the-shelf DL models on guided fuzzing. Specifically, 1) Despite that the DL models may predict well on the coverage of those trivial branches (i.e., the branches covered by almost all the inputs), they are challenging to predict that of those non-trivial branches (i.e., the branches that are only covered by a few or not covered); 2) the feedback generated by the DL models can provide limited impact on fuzzing, i.e., the feedback information is not accurate in terms of the higher-ranked gradient bytes for mutating.
- **Summarized challenges.** Based on the results, we summarize the main challenges that cause the poor performance to predict the coverage of input including 1) imbalanced training dataset in terms of branch coverage (i.e., label), 2) dependency relation among branches (i.e., dependant label problem), and 3) the insufficient expressiveness of the state-of-the-art DL models. In addition, we also provide some preliminary strategies to mitigate some of the challenges.
- **Research directions.** We further pinpoint the future research directions on applying deep learning/machine learning approaches under the fuzzing scenarios, including designing a better embedding of labelled branch coverage, designing more expressive deep learning model structures, and developing a hybrid learning strategy combining DL-based fuzzing with symbolic execution and traditional guided-fuzzing strategies.

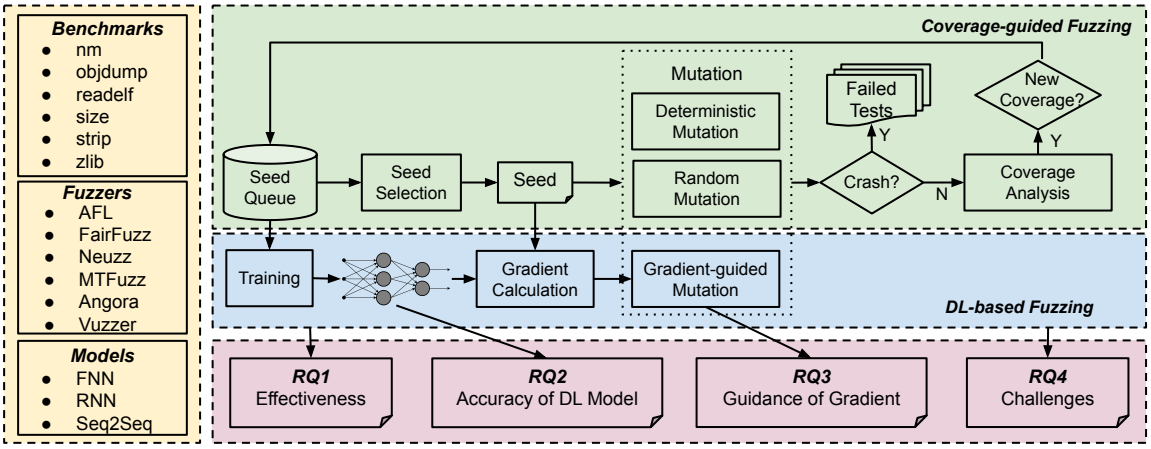


Fig. 1: Overview of our study. DL-based fuzzing can be regarded as a "plugin" into the coverage-guided fuzzing (gree region), which takes runtime coverage information as the dataset to train a deep learning model, and use the model gradients to guide mutation (blue region). In this study, we ask research questions (RQ1-4) revolving around different aspects of DL-based fuzzing (purple region), i.e., overall effectiveness on coverage (RQ1), the model prediction accuracy (RQ2), the effectiveness of gradient to provide feedback (RQ3), and the summarized challenges (RQ4).

2 Overview

Figure 1 shows an overview of (1) how DL-based fuzzing is integrated into general coverage-guided fuzzing, and (2) the rationale to design our study (through research questions) based on such a structure.

Structure: DL-based fuzzing as another coverage-guided fuzzing The general workflow of existing coverage-guided fuzzing framework is showed in green region. During the fuzzing process, a seed will be selected from a seed queue to mutate new test inputs. If any input can cause the program to crash, it will be kept as a failure-inducing case revealing the vulnerability. Otherwise, it will be added into the seed queue if discovering any new program branches. The whole process will continue until a user-defined budget runs out. Different fuzzers will define their own ways to mutate the seeds. For example, AFL fuzzer can use deterministic and random mutation, and FairFuzz can protect some bytes in some seeds to avoid being mutated.

DL-based fuzzing (shown in blue region) is integrated into such coverage-guided fuzzing by (1) taking the runtime coverage as the training data to train a deep learning model and (2) providing feedback to guide mutation via the calculated gradients from the model. As shown in the blue region, the DL-based fuzzing process consists of training data collection, model training, and gradient calculation as the feedback to mutate seed inputs. Different DL-based fuzzers can use different learning strategies with different types of deep learning models (e.g., fully connected network, RNN, etc.).

Empirical study design: Therefore, we design our empirical study by asking research questions revolving around different aspects of DL-based fuzzing. The aspect-question relations are shown by the links between the blue and purple region in Figure 1).

- **RQ1: The effectiveness of DL-based fuzzers.** Compared with traditional fuzzers, how effective are DL-based fuzzers in maximizing code coverage? How do different DL models impact the performance of the DL-based fuzzers?
- **RQ2: The accuracy of DL models.** How accurate are the DL models on approximating the target program's branching semantics, i.e., the accuracy of predicting the branch coverage of a given test input? And, under what scenarios can DL models achieve high coverage-prediction accuracy?
- **RQ3: The feedback used to guide fuzzing.** Since the gradient of the DL model is used as the feedback for guided fuzzing, whether the models with high coverage-prediction accuracy can really result in high coverage performance? And, what program-specific semantic information does the gradient (i.e., feedback) represent?
- **RQ4: The challenges of DL-based fuzzing.** What are the main challenges that limit the performance of the DL-based fuzzers? What are the potential remedies?

To answer the above questions, we choose 6 benchmarks, 5 fuzzers (3 for traditional fuzzing and 2 for DL-based fuzzing), and 3 types of deep learning models for this study, as showed in the yellow region in Figure 1). The details of fuzzers are shown in Table 1.

2.1 Subject benchmarks

To evaluate the effectiveness of DL-based fuzzers, we select 6 widely-used benchmarks (i.e., *nm*, *objdump*, *readelf*, *size*, *strip* and *zlib*) in Binutils (version 2.24). We select those benchmarks because: 1) they are widely used to evaluate both traditional fuzzing [1], [43] and the DL-based fuzzing [6], [7], [32] and 2) the inputs of these programs have specific syntax checking that is expected to be aware by the models in DL-based fuzzers. More details about the number of lines of code and the number of covered edges of initial seeds are shown in Table 2.

2.2 Selected fuzzers

Traditional fuzzers. We select 4 traditional fuzzers: AFL [1], Fairfuzz [43], Angora [22] and Vuzzer [30]. For AFL, we

TABLE 1: An overview of the fuzzers. The fuzzers with the asteroid * are the investigated fuzzers in this study.

Fuzzers	Type	Configuration/Model Specification	Description
AFL*	Traditional	default	AFL [1] is fuzzer which uses genetic algorithm to keep generating inputs by mutating bytes on existing seed inputs. In this mode, AFL generate inputs by repetitively mutating through all the input bytes of an individual seed input.
		-d (i.e., remove deterministic mutation)	In this mode, AFL generate inputs by randomly mutating some bytes of an individual seed input.
FairFuzz*	Traditional	/	FairFuzz [43] can verify and learn the dynamic taint relation between certain positional bytes of an individual seed to cover a (rare) branch. The learned relation between input bytes and branch coverage can allow FairFuzz to avoid mutating relevant bytes when exploring the children branches of those branches.
Angora*	Traditional	/	Angora [22] is a mutation-based coverage guided fuzzer. The main goal of Angora is to increase branch coverage by solving path constraints without symbolic execution.
Vuzzer*	Traditional	/	In order to maximize coverage and explore deeper paths, Vuzzer [30] leverage control and dataflow features based on static and dynamic analysis to infer fundamental properties of the application.
Neuzz*	DL-based	3-layer fully connected network	Neuzz [6] takes as input as a vector of seed bytes and generate output as a vector of branch coverage. Each dimension in the output vector indicates whether a branch is covered or not. By selecting a few dimensions on the output vector, indicating the intent that we would like some branches to be covered (or not covered), Neuzz calculate the input bytes with their largest gradients to update.
MTFuzz*	DL-based	5-layer fully connected network	Different from Neuzz which uses seed bytes as model input, MTFuzz [7] first learns a "input embedding" based on diverse training samples for multiple related tasks (e.g., predicting different types of coverage, etc.). Then, the input embedding vector is used to predict the branch coverage, as what Neuzz does.
Sample-Fuzz	DL-based	Seq2seq	Samplefuzz [32] uses RNN to learn the same generative statistical input model: it can be used to generate new inputs based on the probability distribution of the learning model.
The Neural Augmented AFL	DL-based	RNN	The fuzzer [45] using RNN proposes a learning technique that uses neural networks to learn the patterns in the input file for fuzzy exploration in the past to guide future fuzzy exploration.
FuzzGuard	DL-based	CNN	FuzzGuard [46] uses a deep learning method to predict whether the seed file is reachable to the targeted test target before executing an input, and helps targeted gray box fuzzing to filter invalid seeds to improve the efficiency of fuzzing.

TABLE 2: Benchmark Information

Benchmark	nm	objdump	readelf	size	strip	zlib
#Lines	1678	3424	11463	611	2643	1432
# Edges	1326	2700	2321	1108	3092	584

use two configurations: the default setting including the deterministic mutation stage and the random setting without deterministic mutation (denoted as AFL-d).

Note that instead of comparing with all state-of-the-art fuzzing techniques (such as AFL-fast [37], AFLGo [38], Hawkeye [28] and LTL-Fuzzer [27], etc.), we choose AFL, Fairfuzz, Angora and Vuzzer as baselines in our experiments because their mechanisms are more suitable to be compared with the DL-based fuzzing techniques. The reason is that the selected baselines leverage taint information to guide the mutation process, which is similar to how the DL-based fuzzing techniques work. Therefore, comparing the DL-based fuzzing techniques against the selected baselines is more meaningful than comparing against general fuzzing techniques and the experiments can provide informative and conclusive evidence (see Section 3 for details).

DL-based fuzzers. For the DL-based fuzzers, we select Neuzz [6] and MTFuzz [7], which are the state-of-the-art DL based fuzzing approaches, and more importantly, publicly available. In addition, the deep neural networks in these two tools are used to capture the relationship between the input bytes and the internal branches of the program. For the study, we also select four popular deep learning models, i.e., the the feedforward neural network (FNN) with 3 layers [6], the FNN with 5 layers [7], the RNN [44], and the Seq2Seq [32].

2.3 Study Design

2.3.1 RQ1 (Effectiveness of DL-based fuzzers)

In this paper, we use the integrated tool [48] with the principle of AFL's afl-showmap to measure the edge coverage in the experiment. Edge coverage is a commonly used metric [15], [43] to measure the performance of fuzzing. We use edge coverage as the measurement for the same reason justified in [43]. Specifically, the program can be represented as a control flow graph (CFG), where each node of the graph represents a basic block and edges represent transitions between basic blocks. Existing fuzzing techniques often evolve (or mutate) test cases based on the feedback of the covered edges. Hence, edge coverage is a direct metric to measure the performance of fuzzing.

Moreover, to make the comparison fair, we equip all the fuzzers with the same initial seeds. We run each fuzzer for 12 hours and repeat the experiment for 5 times. We then calculate an average of the edge coverage for the comparison.

2.3.2 RQ2 (Accuracy of DL models)

Accuracy is an important indicator in deep learning for measuring the performance of the trained model. For the DL-based fuzzing, it is important to train an accurate model that can precisely capture the relationship between the input space (i.e., input bytes) and the program space (i.e., the program branches). Intuitively, the DL-based fuzzer can only be effective if the learned model is accurate. Hence, we investigate whether the relationship can be really captured by the DL models. We formally define the trained surrogate model as follows:

Definition 1 (Surrogate Model). A surrogate model is a deep neural network $\mathbf{f} : \mathbf{X} \rightarrow \mathbf{C}$, where $\mathbf{x} \in \mathbf{X}$ is a test input and $\mathbf{c} \in \mathbf{C}$ is a coverage vector that represents the coverage of each branch. If $\mathbf{c}_b$ is 1, the branch b is covered. Otherwise, b is not covered.

Given a branch b, and an input $\mathbf{x}$, we denote model's coverage prediction with $\mathbf{x}$ on b as $\mathbf{f}(\mathbf{x})[b]$.

Moreover, we use $\mathbf{p}(.)$ to denote the *ground truth* mapping from the test inputs to the branch coverage: $\mathbf{p} : \mathbf{X} \rightarrow \mathbf{C}$. Intuitively, $\mathbf{p}(.)$ represents the behavior of a target program $\mathbf{p}$. We use $\mathbf{B}_p$ to represent the set of branches, covered at least once by test inputs, in $\mathbf{p}$ Similarly, we denote groundtruth coverage of $\mathbf{x}$ on the branch $b \in \mathbf{B}_p$ as $\mathbf{p}(\mathbf{x})[b]$.

We equip Neuzz and MTFuzz with four trained models (FNN-3, FNN-5, RNN and Seq2Seq). We follow the setting described in [6], [7] to collect the training data. Specifically, we first run AFL for an hour to generate the test inputs. Then, we randomly split them into a training dataset (80%) and testing a dataset (20%). We measure the performance of each model by (1) the accuracy (denoted as Acc) and (2) the recall (denoted as Rec). We define both as follows:

Definition 2 (Acc). Given a test set $\mathbf{X}$ for the program $\mathbf{p}$, the test accuracy (Acc) of the DL model $\mathbf{f}$ is defined as:

$$Acc = \frac{\sum_{b \in \mathbf{B}_p} BAcc_b}{|\mathbf{B}_p|}$$

$$BAcc_b = \frac{|\{\mathbf{x} | \forall \mathbf{x} \in \mathbf{X}, \mathbf{f}(\mathbf{x})[b] == \mathbf{p}(\mathbf{x})[b]\}|}{|\mathbf{X}|}$$

where $BAcc_b$ defines the branch-wise accuracy of the model on the branch b.

For the trained model, the output of "covered" (i.e., 1) is treated as positive. We define the recall (Rec) to evaluate the performance on predicting the branches that are covered, i.e., whether the model really knows which branches can be executed for a given input.

Definition 3 (Rec). Given a test set $\mathbf{X}$ for the program $\mathbf{p}$, the overall recall (Rec) of the DL model $\mathbf{f}$ is defined as:

$$Rec = \frac{|\{(\mathbf{x}, b) | \forall \mathbf{x} \in \mathbf{X}, \forall b \in \mathbf{B}_p, \mathbf{f}(\mathbf{x})[b] == 1\}|}{|\{(\mathbf{x}, b) | \forall \mathbf{x} \in \mathbf{X}, \forall b \in \mathbf{B}_p, \mathbf{p}(\mathbf{x})[b] == 1\}|}$$

In short, the branch-wise accuracy measures how likely the model's prediction aligns with the ground-truth branch coverage, in contrast, the recall measures how accurate the model's prediction on the covered branches.

2.3.3 RQ3 (Effectiveness of gradient feedback)

For DL-based fuzzers, the trained model that captures the relationship between input bytes and program branches will be used to provide the feedback for the mutation, making it more likely to cover new branches. Therefore, we will further investigate whether the learned model can provide helpful feedback for fuzzing. Specifically, we consider the gradients that are used to provide the feedback in Neuzz and MTFuzz. We first select various models with different accuracy to evaluate their impact on the fuzzing effectiveness. Intuitively, the more accurate the model, the more useful the feedback it can provide. Then, we further study the impact of the ranking of input bytes based on the gradient feedback. We select bytes with different rankings to mutate the seed inputs and evaluate the fuzzing performance. The assumption is that the higher ranked bytes should achieve better results.

TABLE 3: Differences in the coverage of different models running different times

Projects	Fuzzers	Max	Min	Standard deviation
nm	AFL	4,146	4,162	6.52
	AFL-d	9,783	8,864	357.68
	FairFuzz	4,997	4,775	100.03
	Angora	6,237	5,973	23.17
	Vuzzer	4,301	4,018	37.42
	Neuzz	4,416	4,025	178.83
	MTFuzz	4,997	3,924	601.61
objdump	AFL	5,218	5,103	52.52
	AFL-d	8,545	8,291	121.77
	FairFuzz	5,672	5,483	85.10
	Angora	6,132	5,765	64.93
	Vuzzer	5,314	5,212	79.80
	Neuzz	5,335	4,967	170.10
	MTFuzz	6,829	5,487	758.62
readelf	AFL	6,066	6,043	13.74
	AFL-d	13,391	11,507	834.66
	FairFuzz	8,665	7,883	300.29
	Angora	7,812	7,433	43.92
	Vuzzer	7,935	7,677	78.39
	Neuzz	9,457	7,854	662.47
	MTFuzz	9,119	8,374	417.22
size	AFL	3,306	3,161	56.85
	AFL-d	4,839	4,675	72.24
	FairFuzz	4,045	3,583	195.02
	Angora	4,006	3,509	158.35
	Vuzzer	3,928	3,630	132.17
	Neuzz	4,242	3,811	183.44
	MTFuzz	4,515	3,851	490.80
strip	AFL	6007	5840	68.09
	AFL-d	9,708	9,567	62.16
	FairFuzz	7,444	6,952	184.67
	Angora	7,472	6,537	200.78
	Vuzzer	7,549	6,472	213.56
	Neuzz	7,431	5,942	797.01
	MTFuzz	7,132	4,916	991.55
zlib	AFL	1789	1663	57.26
	AFL-d	2,023	1,963	24.22
	FairFuzz	1,879	1,837	16.38
	Angora	1,853	1,793	20.17
	Vuzzer	1,809	1,620	88.71
	Neuzz	1,805	1,641	81.79
	MTFuzz	1,963	1,811	77.95

2.3.4 RQ4 (Challenges)

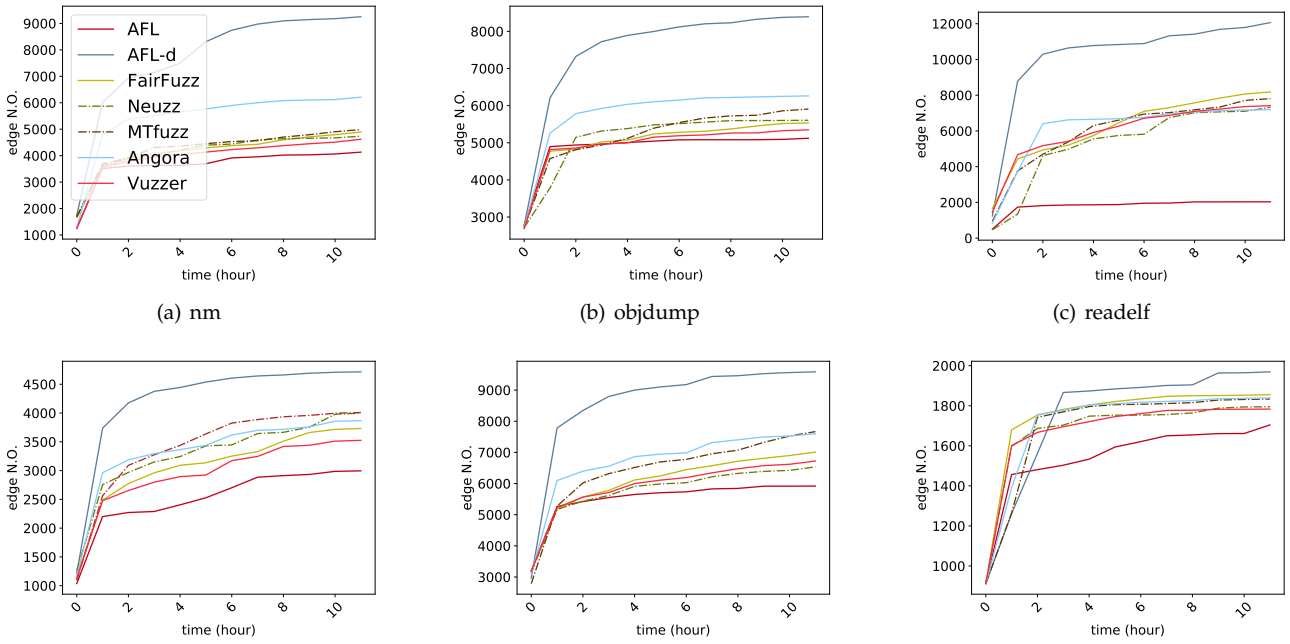
We qualitatively analyze the results where the DL models show limited effectiveness. Then, we simplify the programs and design a set of benchmarks to replicate those poor performance, which can be used to evaluate the coming DL-based fuzzers in the SE/Security community. Furthermore, we propose some preliminary strategies to mitigate some of these challenges.

3 Study Results

In this section, we will introduce the experimental results to answer the 4 research questions highlighted in Section 2.

3.1 RQ1: Effectiveness of DL-based Fuzzers

We evaluate the effectiveness of DL-based fuzzers by comparing the edge coverage with the traditional fuzzers. Figure 2 shows the average results of edge coverage in 12 hours. Based on the results, we have the following observation:



(a) nm (b) objdump (c) readelf

(d) size (e) strip (f) zlib

Fig. 2: The branch coverage of different fuzzers running for 12 hours

- **Large performance overhead caused by deterministic mutation strategy.** AFL-d (without deterministic mutation) significantly outperforms other fuzzers, while the default AFL (with deterministic mutation) underforms the other fuzzers. It indicates that deterministic mutation strategy takes a large overhead on the performance of coverage.
- **Limited improvement of DL models.** Deep learning does not introduce obvious improvement as expected. Specifically, MTFuzz and Neuzz only outperform AFL, which largely suffers from the negative impact of the deterministic mutation strategy. However, their performance is comparable with FairuFuzz and much worse than AFL-d. We will illustrate the reasons in Section 3.4.
- **MTFuzz trumps Neuzz.** Comparing two DL-based fuzzers, MTFuzz achieves better results, potentially indicating that the input embedding learned through multi-task learning by MTFuzz has positive impact on the performance of the coverage.
- **Angora slightly outperforms other traditional fuzzers.** Considering the results of traditional fuzzers, we are surprised that AFL-d significantly outperforms others, indicating that the random mutation (without deterministic mutation) is much efficient. In addition, Angora overall slightly outperforms other fuzzers, especially in nm and objdump.

We compared the results of multiple repeated experiments on the same programs. Table 3 shows the maximum, minimum, and standard deviation coverage comparison among the fuzzers for each program. We found that the results of AFL default mode are more stable, while others such as that of multiple AFL-d mode experiment are more changeable. The reason lies in that AFL-d and DL-based fuzzers adopt more randomness (e.g., through random mutation and model weights initialization) in their fuzzing process. Note that the results of Neuzz and MTFuzz are not exactly same as the results reported in the original paper because of two reasons: 1) the unavoidable randomness of the fuzzing process and 2) the methods for calculating edge coverage are different. Neuzz and MTFuzz calculate edge coverage by checking the trace_bits data structure in AFL, while we use afl-showmap to count the edge coverage in this paper. The results of afl-showmap are more precise since afl-showmap will perform extra calibration when counting the edge coverage. Directly obtaining the result from trace_bits could have redundancy. Thus, the edge coverage results reported in the Neuzz and MTFuzz papers are generally higher.

> **Conclusion:** In this study, we observe that the performance of DL-based fuzzers do not significantly outperform other fuzzers. Overall, the AFL default mode does not perform as well as others and AFL-d outperforms the other approaches in covering more program branches.

3.2 RQ2: The accuracy of DL models

To further investigate the performance of DL-based fuzzers, we conduct a fine-grained analysis on the model accuracy (see definition in Section 2.3.2). Specifically, we use Neuzz and MTFuzz to train deep learning models (with 100 epochs) from the collected test inputs generated by AFL in one hour. Then, we evaluate their overall model accuracy (Definition 2) and overall recall (Definition 3).

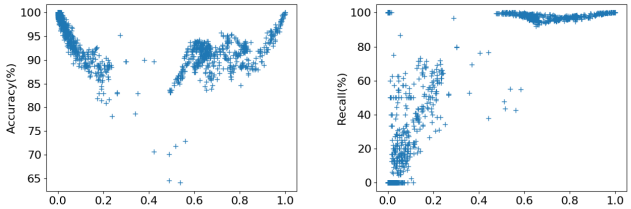
Table 5 shows the results. We can observe that both Neuzz and MTFuzz can achieve high model accuracy albeit low model recalls. Specifically, Neuzz achieves 92.47%,

TABLE 4: The Overall accuracy of different models on the projects with Neuzz.

	nm				objdump				readelf			
Model	FNN-3	FNN-5	Seq2seq	RNN	FNN-3	FNN-5	Seq2seq	RNN	FNN-3	FNN-5	Seq2seq	RNN
Acc	92.47	93.13	72.63	92.24	95.01	95.27	59.45	94.00	96.15	95.83	77.69	95.42
Rec	55.47	53.44	19.01	58.60	61.45	65.37	16.08	59.79	51.97	51.26	20.04	51.61
	strip				size				zlib			
Model	FNN-3	FNN-5	Seq2seq	RNN	FNN-3	FNN-5	Seq2seq	RNN	FNN-3	FNN-5	Seq2seq	RNN
Acc	88.07	89.33	59.98	90.01	89.87	88.97	72.00	88.34	84.77	83.83	73.06	83.61
Rec	61.36	63.15	14.15	62.38	63.37	57.55	18.83	56.71	41.00	39.50	12.43	40.23

TABLE 5: Results of accuracy on different projects (%)

		nm	objdump	readelf	strip	size	zlib
Neuzz	Acc	92.47	95.01	96.15	88.07	89.87	84.77
	Rec	55.47	61.45	51.97	61.36	63.37	41.00
MTFuzz	Acc	91.23	94.65	95.49	89.79	89.32	87.33
	Rec	57.06	59.84	50.82	62.76	56.97	36.72



(a) Accuracy (b) Recall

Fig. 3: The impact of training data distribution on accuracy/recall of each branch. Each point indicate a branch. In terms of overall model accuracy, the overall branch-wise accuracy is very high when a branch is either rarely covered (e.g., between 0 and 0.2) or almost always covered (e.g., between 0.8 and 1). In contrast, the branches with coverage frequency between 0.2 and 0.8 have much lower accuracy. In terms of recall, the branches with high recall are usually because they are "easy" branches. For the branches that are less covered by training data, the recall is low.

95.01% and 96.15% accuracy on *nm*, *objdump* and *readelf*, respectively. In contrast, Neuzz and MTFuzz can only achieve 63.37% and 62.76% respectively, as their best recall.

High overall accuracy and effect of majority votes. We observe that the model tends to predict 0 (i.e., the branch is not covered) for most branches, just because most of branches are less covered or uncovered. Thus, the trained model is more likely to predict "uncover" label for any branches, which significantly suppress those branches can be covered by some test inputs. As a result, the model suffers from low recall, i.e., with limited performance on predicting the covered branches. The above investigation indicates that the statistical measurement used in the loss function of deep learning model may miss important causal program relation. As a result, the model can lack the capability of predicting the most interested branches during fuzzing.

Imbalanced training dataset. Moreover, we further found that the training dataset are very imbalanced regarding the labels (i.e., branches). Figure 3 shows the results of Neuzz on the project *readelf* (More results could be found in our website [47]). Each dot in the figure represents one branch in the program. The X axis shows the frequency of the branch to be covered. For example, 0.0 means that the branch is never covered while 1.0 means that the branch is covered in all training data. The Y axis shows the accuracy of predicting each branch. We can see that the training data is highly biased, most branches are either frequently covered or very difficult to cover (e.g., most dots are located in the upper left or upper right corner). During fuzzing, some shallow branches can be easily covered, which occupy the majority of the training/testing set. In contrast, some deep branches (probably the most interested branches) occupy the minority of the training/testing set. For the latter, the model has very poor performance of predicting their coverage.

Model performance of different neural networks. We also evaluate the performance of different neural networks on the same project. We replace the model architecture of Neuzz with other architectures and measure the accuracy. As shown in Table 4, Seq2Seq achieves the worst results while other models have similar performance. Seq2seq is a model for input and output variable-length tasks, and it is precisely because of this feature that it has a wide range of application scenarios, such as neural machine translation, text summarization, speech recognition, text generation, etc. However, the deep learning task studied in this paper is a multi-label problem that performs multi-branch coverage prediction for input sequences of indeterminate lengths, so Seq2seq performs very poorly on such research problems.

Conclusion: We observe that DL-based fuzzers can often suffer from the problem of imbalanced training data, which makes the model predict branches as "uncovered" blindly, simply based on statistical evidence.

3.3 RQ3: Feedback from neural networks

Furthermore, we study the effectiveness of the neural network to provide the guidance for the mutation. Specifically, we conduct two experiments to evaluate 1) whether the model with higher/lower overall prediction accuracy can achieve better/worse coverage performance? and 2) whether the model with higher overall prediction accuracy can provide correct feedback to guide mutation?

3.3.1 *Coverage with the models with different accuracy.*

For each project, we first train a Neuzz model for 100 epochs until it achieves a high accuracy on the testing dataset (the accuracy of 95.0% in this study), denoted as $Model_{95}$. We choose Neuzz as it is simple to configure comparing to MTFuzz while two fuzzers have comparable performance. Then, we adopt the mutation testing technique [5] on $Model_{95}$ to randomly generate a set of models with different

TABLE 6: The number of edges covered by mutating different bytes after 12 hours

Bytes	nm	objdump	readelf	size	strip	zlib
Top-64	4330	5504	9250	3142	6042	1724
Second-64	4025	5609	9286	3605	6098	1664
Bottom-64	4092	5433	8474	3361	6013	1675

accuracy on the testing dataset. We sample the models with accuracy of 45%, 60%, 75%, denoted as $Model_{45}$, $Model_{60}$, and $Model_{75}$. In addition, we create a dummy model, which adopt a *random* strategy which mutate some bytes randomly on seeds. We equip Neuzz with those models as a Neuzz variant. Each run of a Neuzz variant takes for 5 hours, and we conduct 5 runs for each variant.

Figure 4 shows the results of different Neuzz variants on *readelf* program. Readers can see [47] for the results of more programs, which are similar to the results in Figure 4. Overall, we observe little correlation between the model accuracy and the coverage performance. $Model_{75}$ can outperform $Model_{95}$, and $Model_{45}$ is no worse than $Model_{60}$. Even the dummy network can have comparable results with $Model_{95}$ after fuzzing for three hours. The results indicate that the model provides limited or less significant feedback even if it achieves high model prediction accuracy.

3.3.2 Correctness of the feedback provided by the neural network

DL-based fuzzers adopt the gradient to identify key bytes to mutate. The feedback is provided in the form of reported bytes to mutate on a given seed, and each byte is ranked by the gradient score. In order to fully evaluate the quality of feedback, we further modified Neuzz so that it can only use the feedback generated by the model to mutate[1]. We design multiple experiments to let Neuzz use feedback generated from the top K most important bytes (Top-K), the second K most important bytes (Second-K) and K least important bytes (Bottom-K). In this study, we let K be 64.

Table 6 shows the result on the covered branches by different Neuzz variants for 12 hours. We can see that the provided feedback is overall effective, as the feedback of Top-64 outperforms Second-64 and Bottom-64 in general. Nevertheless, the improvement is limited in project like `size`. Through experimental data we further found that the top-ranked bytes in the gradient computed by the neural network do not always align with the really important bytes, which we will illustrate in the next subsection.

Conclusion: In the current DL-based fuzzers, the neural network models provides effective feedbacks to guide the mutations. Nevertheless, the improvement sometimes is not significant in some projects.

3.4 RQ4: Technical challenges for DL-based fuzzing

To further understand the challenges of DL-based fuzzers, in this section, we conduct a more detailed qualitative analysis by demonstrating several examples simplified from our study, where neural network models are challenging

1. Neuzz combines gradient and random strategy to mutate [6].

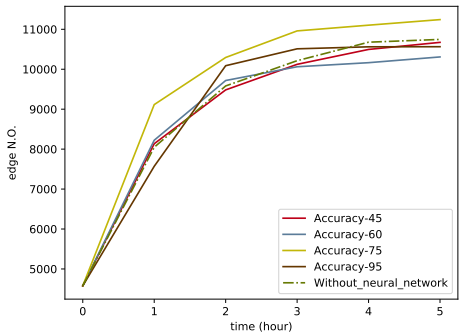


Fig. 4: The influence of different accuracy neural networks on the Neuzz

to predict the coverage of a test input. The challenges include dependant labels, model over-generalization, and model inexpressiveness. For each challenge, we describe its impact and propose some of its preliminary remedy. Further potential systematic solutions will be discussed in Section 3.5.

3.4.1 Dependent labels

Given a conventional deep learning model $\mathbf{f} : \mathbf{X} \rightarrow \mathbf{Y}$ where $\mathbf{X}$ represents the input space where $\mathbf{Y}$ represents the label space, the dimensions of any $\mathbf{y} \in \mathbf{Y}$ are assumed to be independent. Nevertheless, such an assumption can hardly be held when branch coverage is used as the label. The reason lies in that the coverage of a non-root branch always depends on the coverage of its parent branch. Such dependant labels can cause (1) unbalanced the training dataset and (2) contradicting gradients (or feedback).

```
1  int status = 0;
2  else if (str[10] == 'F') {
3     status = 1;
4     if (str[11] == 'u') {
5        status = 2;
6        if (str[12] == 'z')
7           status = 3;
8        else
9           status = 4;
10    }
11 }
```

Listing 2: Example for data imbalance and unreliable gradients

Listing 2 shows a simplified example where dependant labels happen. The input is a string, which will be transformed into a model input vector where each dimension represents a character. There are 6 branches in Listing 2, and we use $b_{l,p}$ to denote the p (true or false) branch defined on line l. For example, $b_{2,true}$ represents the true branch defined in line 2, i.e., from line 2 to line 3. Thus, we have output vector of length of 6 in this example, where four of them are involved in the control dependency relation. Figure 5 shows the details.

We have counted this type of label-dependent situation in the real world. Taking the Benchmark we use as an example, we make the following definition:

Definition 4 (Dependent branch). If a branch has sibling branches or sub-branches, the dependent branches of the

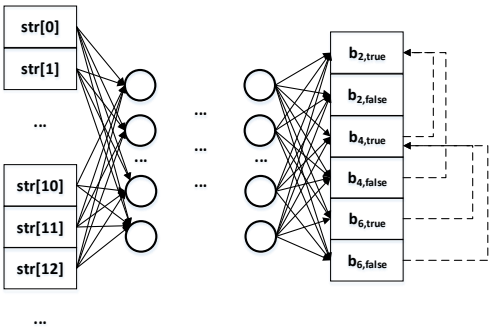


Fig. 5: A deep learning model applied to predict the coverage of an input string in Listing 2. The solid lines represent the feedforward relation of the network, and the dashed lines represent the dependant relation between the dimensions of the output layers.

TABLE 7: The proportion of dependent branches in the actual project

	nm	objdump	readelf	size	strip	zlib
Number of Labels	1,064	1,742	2,211	814	1,705	582
Number of Dependent Labels	425	823	1,237	251	959	141
Ratio (%)	39.94	47.24	55.95	30.83	56.25	24.23

branch include sub-branches, sibling branches, and sub-branches of sibling branches.

Since Neuzz and MTFuzz use the data after running AFL for a period of time as the training set, we calculated the ratio of dependent branches (labels with dependencies) in the seeds used for training to all training branches (all labels) on each benchmark. The data in Table 7 shows that there are dependencies among many labels in the training set data used by Neuzz and MTFuzz.

Imbalanced data. Here, we have the label relations such as $b_{6,true} = 1 \rightarrow b_{4,true} = 1$ as $b_{6,true}$ is control-dependant on $b_{4,true}$. Moreover, such dependency is transitive. As a result, the inputs with label $b_{6,true} = 1$ will always be with label $b_{4,true} = 1$ and $b_{2,true} = 1$. Such an asymmetric relation causes the number of inputs with label $b_{6,true} = 1$ is smaller than that with label $b_{4,true} = 1$ and $b_{2,true} = 1$. The deeper the branch, the larger the gap.

Sampling arbitrary string with a traditional guided-fuzzing can cause huge training data imbalance on labels $b_{6,true} = 1$, $b_{4,true} = 1$, and $b_{2,true} = 1$. Thus, we control the "balance level" of the training inputs to exercise above labels and observe their impact on the model prediction accuracy on testing inputs. Given a set of branches $B = \{b_1, b_2, ..., b_n\}$ and a set of training inputs I, let the set of training inputs exercising b_i be $I(b_i) \subset I$, we let the balance level of I on B be $|I(b_1)| : |I(b_2)| : ... : |I(b_n)|$. Here, we compare the branch-wise prediction accuracy when sampling inputs exercising $b_{2,true}$, $b_{4,true}$, and $b_{6,true}$ with different balance level from (8:4:1 to 1:1:1). Table 8 preliminarily shows that more balanced training data leads to more accurate model prediction accuracy. As the distribution trend of training samples is gradually balanced, the prediction accuracy of the included labels is improved to a certain extent. However, in the real program environment, the relationship between program branches is complex, and the situations between program branches are full of tricks. Adjusting the training samples of two unbalanced labels often leads to an unbalanced proportion of some other branch labels, for example, for two branch labels with an inclusion relationship, if the number of positive samples of the inner label is increased, the number of positive samples of the outer label will also increase.

TABLE 8: Model prediction accuracy with changed balance level

Balance level	$b_{2,true}$	$b_{4,true}$	$b_{6,true}$
8:4:1	91.2%	56.2%	53.2%
6:3:1	90.8%	58.4%	52.9%
3:2:1	90.2%	62.1%	53.7%
2:2:1	91.7%	63.0%	55.8%
1:1:1	94.6%	66.7%	61.3%

We compare the precision of the feedback from these two models as follows: given a test input "...FAB..." that covers $b_{2,true}$, we aim to cover the branch $b_{4,true}$. Then we use the gradient to identify the key bytes (should be 'A') that should be more important for covering $b_{4,true}$. With the original model (i.e., 3:2:1), we found that the character 'A' has small gradient ($5.7 * 10^{-28}$) and is ranked as the $13th$ important byte. With the improved model (i.e., 1:1:1), the gradient of 'A' is increased (0.096) and is the $3rd$ important byte, indicating that high-quality model is helpful.

Contradicting gradients. Given a branch (i.e., label) b_i and an input position $\mathbf{x}_j$, DL-based fuzzing can update $\mathbf{x}_j$ to cover b_i by updating $\mathbf{x}_j = \mathbf{x}_j + \alpha \frac{b_i}{\mathbf{x}_j}$ where α is the learning rate. In practice, covering a target branch requires covering a sequence of its parent branches (as the path condition), denoted by $b_{i1}, b_{i2}, ..., b_{ik}$. Here k is the number of parent branches. Therefore, we can have k gradients to update an input position. Our observation is that those k gradients can contradict with each other if we are to cover an uncovered branch in the training dataset.

Still with the example in Figure 5, assuming we have the training inputs for all the labels except $b_{6,true} = 1$. In other words, the training inputs cover all the branches except for the branch $b_{6,true}$. Given the input position $\mathbf{x}[10]$ ='F', $\mathbf{x}[11]$ ='u', $\mathbf{x}[12]$ ='z', we have label $b_{6,false} = 0$, $b_{4,true} = 1$, and $b_{2,true} = 1$. To switch $b_{6,false} = 0$ to $b_{6,false} = 1$ (i.e., uncover the true-branch on line 6) while preserving $b_{4,true} = 1$, and $b_{2,true} = 1$, the model will provide two conflicted solutions: (1) to switch $b_{6,false} = 0$, the model suggests to change $\mathbf{x}[10]$ to the character other than 'F'; in the meantime, (2) to preserve $b_{4,true} = 1$, the model suggests to preserve $\mathbf{x}[10]$.

Here is the observation. Our experiment shows that given an input $\mathbf{x}[10]$ ='F', $\mathbf{x}[11]$ ='u', $\mathbf{x}[11]$ ='z' the gradient $\frac{\partial b_{6,false}}{\partial \mathbf{x}[10]}$ = 0.72, indicating to change the value of $\mathbf{x}[10]$ to a larger character like 'G' or 'H' (according to the order of ASCII code), In contrast, the gradient $\frac{\partial b_{2,true}}{\partial \mathbf{x}[10]}$ = $\frac{\partial b_{4,true}}{\partial \mathbf{x}[10]}$ = 0, indicating to preserve the value of $\mathbf{x}[10]$. The same phenomenon happens to $\mathbf{x}[11]$. We observe that this is a side effect caused by miss-considering the label dependency, which is non-trivial in DL-based fuzzing techniques.

3.4.2 Over-generalization

Deep learning model can extrapolate the label to the neighbours of a training sample. For example, given an input $\mathbf{x}$ =[1, 1, 1] with label 1, a well learned model can predict an input $\mathbf{x}'$ =[1.02, 0.99, 1] with label 1 as $\mathbf{x}'$ is the neighbour of $\mathbf{x}$, However, branch conditions sometimes require exact match, where the model can makes incorrect prediction. Listing 3 shows an example where the model over-generalize the coverage prediction. The model will predict the branch $b_{2,true} = 1$ with an input $\mathbf{x}$ with $\mathbf{x}[10]$ of value 'D' and 'E'. It is because 'D' and 'E' have similar ASCII code with the correct value 'F'.

A remedy for this situation is to apply data augmentation. More specifically, we feed inputs by perturbing original training samples so that $\mathbf{x}[10]$ can be of value 'D' and 'E'. By data augmentation, we improve the model test accuracy (i.e., branch-wise accuracy) for predicting $b_{2,true}$ in Listing 3 from 80.2% to 88.7%.

```
1  int status = 0;
2  if (str[10] == 'F') {
3     status = 1;
4  }
5  else {
6     status = 2;
7  }
```

Listing 3: An example of over-generalized model

3.4.3 Limited model expressiveness

Finally, we observe that the model can hardly fit the semantics such as *substring*, *contains*, *prefix*, etc. Those operations require the model make the prediction independent of the input position, as shown in Listing 4. In contrast, the existing deep learning model structures (e.g., DNN, CNN, Transformer, etc.) are highly dependent on the positional information. The model accuracy to predict branch $b_{2,true}$ is only 47.7%. Moreover, the longer the training input, the worse the branch-wise prediction accuracy.

```
1  int status = 0;
2  else if (str.find('ABC') == 'True') {
3     status = 1;
4  }
5  else {
6     status = 2;
7  }
```

Listing 4: An example of limited expressive model

3.5 Practice and Research Direction

Based on the qualitative and quantitative analysis, the state-of-the-art DL-based fuzzing trains and uses the deep learning model, often with the model assumptions violated. Therefore, in this section, we first propose a practice to apply deep learning model for coverage-guided fuzzing, with maximum conformance to the model training/application assumption. Then, we pinpoint a few research directions in the future.

3.5.1 Practice

As shown in Figure 6, we design the practice mainly for addressing the problem of imbalanced training dataset(see discussion in Section 3.4.1). In Figure 6, the white boxes (i.e., processes) and ellipses (i.e., artifacts) are inherent in DL-based fuzzing. We enhance the traditional workflow by introducing the process of data balancing, trusthworthiness checking, and an optional active learning (the dashed line from the training process to runtime seed coverage dataset).

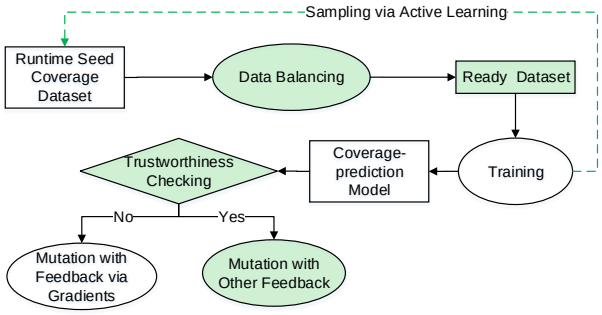


Fig. 6: Practice to apply deep learning in fuzzing. The green entities are the suggested enhanced steps for DL-based fuzzing.

Data balancing. When training the deep learning models, we can reweight the training samples. For example, we can increase the number by mutating the seeds covering a branch with less training samples (i.e., *data augmentation*). A balancing measurement can be proposed to guide such processes. While a perfect balance may not be achieved, a relative balance can still improve the model performance.

Trustworthiness checking. The goal is to distinguish when we can believe in the decision of neural network model. A measurement of trustworthiness is required to avoid the model to make unreliable predication on some branches. For example, a neural network usually requires a considerable number of samples to achieve a good performance, which means rare-branches (i.e., the branches covered by fewer training inputs) cannot benefit well from the model. The measurement can be defined with heuristics or some other indicators, e.g., the degree of contradicting gradients (see Section 3.4.1). Once an untrustful decision is detected, we can switch to other more reliable or traditional feedback to apply mutation.

Active learning. Last, to avoid over-generalization (see Section 3.4.2), model learning process can keep refining, by sampling or generating *informative* inputs to fine-tune the model. For neural network model, we can achieve the classification decision boundaries via many methodologies [2], [3], [4]. Thus, we can at least force the fuzzer to generate training inputs lies on the model decision boundaries, so that retraining the model can be more effective.

3.5.2 Research Directions

In this section, we pinpoint three technical challenges to address.

Independent output dimensions. First, dependant output dimensions need to be addressed, either through (1) building several models each of which is with independent outputs, or (2) apply dimension reduction techniques (e.g., SVD, LDI, etc.) to convert a dependant output vector to an independent (or less dependant) vector.

More expressive model structure. Second, the semantic of program behaviors is far more complicated and less continuous in the input space, comparing to images and sentences in the AI community. In addition, the existing models (DNN, CNN, Transformer) are position-dependant, which cannot well address the position-dependant branch condition such as the containment and substring relation for string. Thus, a more sophisticated model structure may be required in the future.

Hybrid learning for DL-based fuzzing. Finally, the limits of DL-based fuzzing should be well explored. We can be aware of the limit either through proactive analysis (e.g., defining heuristics) or runtime detection (verifying the effectiveness of feedback during the fuzzing process). Thus, exploring an effective hybrid fuzzing strategy of symbolic execution, random, and DL-based fuzzing is also required.

4 Threat to Validity

The selected subjects could be a threat that may cause a bias when evaluating the effectiveness of fuzzers in RQ1. We selected 6 projects that are widely used in the other work including traditional fuzzing and DL-based fuzzing. The randomness in the fuzzing and the model training is a threat. To mitigate this factor, we repeat the evaluation for multiple times to calculate the average. Another threat is the test cases collected for training the model, which may affect the results of the model accuracy (RQ2) and the feedback (RQ3). We follow the setting of Neuzz and collect the test cases that are generated by AFL. Nevertheless, we believe that our experimental results have clearly shown the limitation of the existing DL-based fuzzers, which provide pinpoint the research directions for the future research.

5 Related Work

Fuzzing Technique In recent years, many state-of-the-art technologies have been proposed. AFL [1] is one of the representative CGF fuzzers, which provides guidance for other fuzzers. For example, Gan [21] et al. reduce path conflicts by providing more accurate coverage information while still maintaining low instrument overhead. Böhme [37] et al. propose AFLFast to use Markov model to facilitate fuzzing. It selects the seed of the low-frequency execution path, and then mutates it to cover more code to find errors. Another variant of AFL is AFLGo [38], which selects seeds whose execution paths are closer to the target path and mutates them to trigger target errors. Many people also improve AFL from both dynamic and static analysis perspectives. Chen [29] et al. use dynamic techniques such as colorful taint analysis to find bugs. Rawat [30] et al. use both static and dynamic analysis techniques to obtain control flow and data flow information to improve the effectiveness of mutations. In addition, there are some fuzzers that use other novel technologies. MOPT [41] uses a customized particle swarm optimization algorithm (PSO) to find the optimal selection probability distribution of the operator from the perspective of fuzzy validity. You [42] et al. propose a fuzzing technology called ProFuzzer, which automatically discovers input fields and their semantics through a lightweight random fuzzing process called "probing" to guide the online evolution of seed mutations. We choose AFL and the fuzzer guided by the input-branch correlation: FairFuzz as the controlled group for the research.

Application of Learning in Fuzzing. Recently, researchers start to adopt machine learning techniques in fuzzing. For example, Skyfire [31] proposed by Wang et al. uses a data-driven seed generation method to automatically extract semantic information through PCFG (Probabilistic Context-Related Grammar, which contains semantic rules and grammatical features). The Samplefuzz [32] proposed by Godefroid et al. uses neural network-based statistical learning technology to automatically generate input grammar from sample inputs. It proposes and evaluates a PDF object automatic learning generation model based on seq2seq recurrent neural network. The GANFuzz [33] studied by Hu et al. estimates the basic distribution function of industrial network protocol messages by training the generated model in the generative confrontation network, thereby learning the protocol grammar. Paduraru [36] et al. used different file formats to cluster the corpus. By treating the corpus of the input file as a series of characters, the generative model of each cluster can be learned through seq2seq. DeepFuzz [39] learned the correct C program syntax from the original GCC test suite through the seq2seq model. The model continuously generates grammatically correct C programs according to the learned grammar. LEFT [40] built a model based on reinforcement learning to fuzz test the LTE functions in Android phones. We mainly investigated fuzzers using models which learn input and branch coverage correlation, and conducted fine-grained research.

6 Conclusion

In this paper, we conducted an empirical study towards understanding the effect of deep learning in coverage-guided fuzzing. Our results show that deep learning has not brought improvements in fuzzing as we expected. The in-depth study shows that deep neural networks cannot learn the behavior of programs well, making the feedback on the mutation inaccurate during the fuzzing. We further conducted fine-grained case studies to analyze the challenges and propose some preliminary strategies to mitigate these challenges. Finally, we pinpoint the future research directions on improving DL-based fuzzing.

7 Acknowledgement

This research is supported in part by the Minister of Education, Singapore (21-SIS-SMU-033, T1-251RES1901, T2EP20120-0019, MOET32020-0004), the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme (Award No. NRF-NCR_TAU_2021-0002) and A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002)

References

[1] M. Zalewski, "American fuzzy lop: a security-oriented fuzzer," *URl: http://lcamtuf.coredump.cx/afl/*, 2010.

[2] W. He, B. Li, and D. Song, "Decision boundary analysis of adversarial examples," in *International Conference on Learning Representations*, 2018.

[3] A. Mandelbaum and D. Weinshall, "Distance-based confidence score for neural network classifiers," *arXiv preprint arXiv:1709.09844*, 2017.

[4] R. Yousefzadeh and D. P. O'Leary, "Investigating decision boundaries of trained neural networks," *arXiv preprint arXiv:1908.02802*, 2019.

[5] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[6] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[7] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.

[8] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Conference on Security and Privacy in Communication Networks (SECURECOMM)*, 2015.

[9] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

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

[11] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[12] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 227–242.

[13] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International journal on software tools for technology transfer*, vol. 11, no. 4, pp. 339–353, 2009.

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

[15] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection." *In Proceedings of the 2019 13th Joint Meeting on Foundations of Software Engineering*, 2019, pp. 533–544.

[16] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[18] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[19] M. De La Maza and D. Yuret, "Dynamic hill climbing," *AI expert*, vol. 9, no. 26, p. 26, 1994.

[20] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[21] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[22] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[23] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," *arXiv preprint arXiv:1901.01142*, 2019.

[24] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," *School of Computer Science Carnegie Mellon University*, 2012.

[25] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 627–637.

[26] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "{GREYONE}: Data flow sensitive fuzzing," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2577–2594.

[27] Meng R, Dong Z, Li J, Beschastnikh I, Roychoudhury A. "Linear-time Temporal Logic guided Greybox Fuzzing." in *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[28] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[29] K. Chen, D. Feng, P. Su, and Y. Zhang, "Black-box testing based on colorful taint analysis," *Science China Information Sciences*, vol. 55, no. 1, pp. 171–183, 2012.

[30] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[31] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[32] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[33] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: a gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 138–145.

[34] Y. Lin, Y. Ong, J. Sun, G. Fraser, and J.S. Dong, "Graph-based seed object synthesis for search-based unit testing" in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[35] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J.S. Dong, "Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing" in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[36] C. Paduraru and M.-C. Melemciuc, "An automatic test data generation tool using machine learning." in *ICSOFT*, 2018, pp. 506–515.

[37] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[38] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[39] K. Böttinger and C. Eckert, "Deepfuzz: Triggering vulnerabilities deeply hidden in binaries," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 25–34.

[40] K. Fang and G. Yan, "Emulation-instrumented fuzz testing of 4g/lte android mobile devices guided by reinforcement learning," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 20–40.

[41] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1949–1966.

[42] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 769–786.

[43] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[44] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," *arXiv preprint arXiv:1906.11133*, 2019.

[45] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[46] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2255–2269.

[47] "More results," https://sites.google.com/view/ai4fuzz/.

[48] "Fuzzer data collector," https://github.com/ThePatrickStar/fuzzer-data-collector.

Siqi Li received the bachelor's degree from Hangzhou Dianzi University, Hangzhou, China. He is pursuing a master's degree at Tianjin University, Tianjin, China. Currently, he is conducting research on software vulnerability analysis, fuzzing and other related aspects in the Institute of Software and Information Security Engineering of Tianjin University.

Xiaofei Xie received his Ph.D, M.E. and B.E. from Tianjin University. He is currently an assistant professor in Singapore Management University, Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance analysis of artificial intelligence. He has published some top tier conference/journal papers relevant to software analysis in ICSE, ISSTA, FSE, ASE, TDSC, TIFS, TSE, IJCAI, NeurIPS, ICML and CCS. In particular, he won two ACM SIGSOFT Distinguished Paper Awards in FSE'16 and ASE'19.

Yun Lin received his Ph.D from Fudan University. He is currently an research assistant professor in National University of Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance analysis of artificial intelligence. He has published some top tier conference/journal papers relevant to software analysis in ICSE, ISSTA, FSE, ASE, TIFS, TSE, AAAI, and USENIX Security. In particular, he won ACM SIGSOFT Distinguished Paper Awards in ICSE'18.

Yuekang Li received his B.S. degree from the Nanyang Technological University in 2015 and his Ph.D. degree from the Nanyang Technological University in 2020. He is now a research assistant professor in Nanyang Technological University since 2020. His research interests include fuzzing and other software vulnerability detection techniques.

Ruitao Feng received his B.S. degree from the Tianjin University in 2014 and his Ph.D. degree from the Nanyang Technological University in 2021. He is now a research fellow in Nanyang Technological University since 2021. Previously, he was a research assistant in Nanyang Technological University from 2014 to 2020. His research interests include discovering and solving security problems on mobile platform, IoT system and AI-based cybersecurity system.

Xiaohong Li received the PhD degree from School of Computer Science and Technology, Tianjin University, China, in 2005.She is the director of the Institute of Software and Information Security Engineering, Tianjin University. She is mainly engaged in computer science and computer application, software engineering and security software engineering, high-assurance software and network security and other information security fields.

Weimin Ge received the PhD degree from School of Computer Science and Technology, Tianjin University, China, in 2008. His current research interests include computer networking, mobile computing and information systems development and applications.

Jin Song Dong received his PhD degree from the University of Queensland, Australia. He is a professor with the School of Computing, National University of Singapore. His research interests include software engineering, program analysis, formal verification, and model checking.