Evaluating and Improving Neural Program-Smoothing-based Fuzzing

Mingyuan Wu†
Southern University of Science and Technology, Shenzhen, China and the University of Hong Kong, Hong Kong, China
11849319@mail.sustech.edu.cn

Ling Jiang, Jiahong Xiang,
Yuqun Zhang*
Southern University of Science and Technology, Shenzhen, China
{1171906,11812613,zhangyq}@mail.sustech.edu.cn

Huixin Ma, Sen Nie, Shi Wu
Tencent Security Keen Lab
Shanghai, China
{huixinma,snie,shiwu}@tencent.com

Heming Cui
The University of Hong Kong
Hong Kong, China
heming@cs.hku.hk

Lingming Zhang
University of Illinois Urbana-Champaign, USA
lingming@illinois.edu

ABSTRACT
Fuzzing nowadays has been commonly modeled as an optimization problem, e.g., maximizing code coverage under a given time budget via typical search-based solutions such as evolutionary algorithms. However, such solutions are widely argued to cause inefficient computing resource usage, i.e., inefficient mutations. To address this issue, two neural program-smoothing-based fuzzers, Neuzz and MTFuzz, have been recently proposed to approximate program branching behaviors via neural network models, which input byte sequences of a seed and output vectors representing program branching behaviors. Moreover, assuming that mutating the bytes with larger gradients can better explore branching behaviors, they develop strategies to mutate such bytes for generating new seeds as test cases. Meanwhile, although they have been shown to be effective in the original papers, they were only evaluated upon a limited dataset. In addition, it is still unclear how their key technical components and whether other factors can impact fuzzing performance. To further investigate neural program-smoothing-based fuzzing, we first construct a large-scale benchmark suite with a total of 28 popular open-source projects. Then, we extensively evaluate Neuzz and MTFuzz on such benchmarks. The evaluation results suggest that their edge coverage performance can be unstable. Moreover, neither neural network models nor mutation strategies can be consistently effective, and the power of their gradient-guidance mechanisms have been compromised. Inspired by such findings, we propose a simplistic technique, PreFuzz, which improves neural program-smoothing-based fuzzers with a resource-efficient edge selection mechanism to enhance their gradient guidance and a probabilistic byte selection mechanism to further boost mutation effectiveness. Our evaluation results indicate that PreFuzz can significantly increase the edge coverage of Neuzz/MTFuzz, and also reveal multiple practical guidelines to advance future research on neural program-smoothing-based fuzzing.

ACM Reference Format:

1 INTRODUCTION
Fuzzing [44] nowadays has been widely adopted to detect software bugs or vulnerabilities via feeding invalid, unexpected, or random data as inputs for executing programs under test. To date, many existing approaches model fuzzing as an optimization problem and attempt to solve it by augmenting code coverage via mutating program seed inputs under a given time budget. Such coverage-guided fuzzing tasks can be typically resolved by applying search-based optimization algorithms such as evolutionary algorithms [13, 15, 42, 49, 51]. Specifically, test inputs are iteratively filtered, mutated, and executed such that the test results can approach the optimal solutions to satisfy the fitness functions of the adopted evolutionary algorithms, which are usually designed to maximize code coverage. However, evolutionary fuzzers have been argued that they fail to “leverage the structure (i.e., gradients or higher-order derivatives) of the underlying optimization problem” [41]. To address such issue, neural program-smoothing-based techniques, e.g., Neuzz [41] and MTFuzz [40], have been recently proposed to exploit the usage of gradients for fuzzing via neural network models. Specifically, they first adopt a neural network which, given the byte sequence of a seed as input, outputs a vector representing its associated program branching behaviors. Next, they compute the
gradients of the collected output vectors with respect to the bytes of the given seed. Accordingly, they sort the resulting gradients and develop strategies to mutate the bytes with larger gradients more aggressively. Eventually, all the resulting mutants are used as test cases for fuzzing. Note that MTFuzz further attempts to outperform Neuzz by leveraging the power of multi-task learning and adopts a dynamic analysis module to augment the mutation strategy. In their original papers, Neuzz outperforms 10 existing coverage-guided fuzzers on 10 real-world projects by at least 3X more edge coverage over 24-hour runs and further detects 31 previously-unknown bugs. Compared to Neuzz, MTFuzz achieves 2X to 3X edge coverage upon all the benchmark projects and exposes 11 previously-unknown bugs which cannot be detected by the other fuzzers.

Despite the effectiveness shown in their original papers, the evaluation on Neuzz and MTFuzz can be potentially biased due to their limited benchmark suite with only 10 projects. Moreover, Neuzz and MTFuzz adopt a different edge coverage metric from many existing fuzzers [4, 9, 27, 31, 51] that can potentially bias the performance comparison. Furthermore, the investigation on the factors that can impact their edge coverage performance is rather limited, i.e., they only simply presented the overall effectiveness of the techniques without investigating the contributions made by individual components, e.g., the model structure, the gradient guidance mechanism, and the mutation strategy.

In this paper, to enhance the understanding of the effectiveness and efficiency of program-smoothing-based fuzzing, we first construct a large-scale benchmark by retaining all the projects adopted in the original Neuzz and MTFuzz papers (except one that we fail to run) and adding 19 additional open-source projects that were frequently adopted in recent fuzzing research work. We then conduct an extensive evaluation for Neuzz and MTFuzz accordingly. The evaluation result suggests while Neuzz and MTFuzz can outperform AFL on all the studied benchmark projects by 10.5% and 8.9% on average in terms of edge coverage respectively, MTFuzz does not always outperform Neuzz and both their edge coverage performances are highly program-dependent. We also find neither their mutation strategies nor neural network models can be consistently effective. Meanwhile, although the gradient guidance mechanisms can be promising, their strengths have not been fully leveraged.

Inspired by the findings of our study, we propose an improved technique, namely PreFuzz [38], upon neural program-smoothing-based fuzzing. In particular, we develop a resource-efficient edge selection mechanism to facilitate the exploration on unexplored edges rather than the already covered edges. Moreover, we also apply a probabilistic byte selection mechanism guided by gradient information to Neuzz and MTFuzz to further boost edge exploration. Our evaluation results suggest that PreFuzz can significantly outperform Neuzz and MTFuzz, i.e., 43.1% more than Neuzz and 45.2% more than MTFuzz averagely in terms of edge coverage.

To conclude, this paper makes the following contributions:

- **Dataset.** A dataset including 28 real-world projects that can be used as the benchmarks for future research on fuzzing.
- **Study.** An extensive study of neural program-smoothing-based fuzzers on the large-scale benchmark suite, with detailed inspection of both their strengths and limitations.
- **Technical improvement.** A technique improving neural program-smoothing-based fuzzers by combining a resource-efficient edge selection mechanism and a probabilistic byte selection mechanism.
- **Practical guidelines.** Multiple practical guidelines for advancing future program-smoothing-based fuzzing research.

2 BACKGROUND

2.1 Coverage-guided Fuzzers

Coverage-guided fuzzers nowadays widely adopt evolutionary algorithms [49] for mutation strategies since they can be advanced in discovering program vulnerabilities without prior program knowledge. In this section, we first introduce the basic framework for evolutionary algorithms, and then illustrate how a typical coverage-guided fuzzer AFL integrates evolutionary algorithms.

2.1.1 Evolutionary Algorithm. To solve an optimization problem, an evolutionary algorithm (EA) adopts operations such as mutating the existing solutions to generate new solutions. Among such generated solutions, an EA applies a fitness function to filter them based on their quality such that the remaining ones are retained as one population. Such process is iterated until hitting the preset time budget with the final population returned as the solutions for the optimization problem.

2.1.2 Integrating fuzzing with EA. Coverage-guided fuzzers often use increased code coverage as the fitness functions. Specifically, they usually adopt edge coverage (where an edge refers to a basic-block-wise transition, e.g., a conditional jump in programs) to represent code coverage and retain only the seeds that can trigger new edge coverage for further mutations. For instance, American Fuzzy Lop (AFL) [51], a widely-used coverage-guided fuzzer, is launched by instrumenting programs such that it can acquire and store the edge coverage of each program seed input at runtime. Subsequently, AFL iterates and mutates each seed input according to its adopted evolutionary algorithm. Like most coverage-guided fuzzers [4, 9, 27, 31], when running a seed increases edge coverage, AFL identifies such seed as an “interesting” seed and retains it for further mutations. Note that the mutations in AFL consist of two stages: the deterministic stage (AFL\textsubscript{Deterministic}) and the havoc stage (AFL\textsubscript{Havoc}). In particular, AFL\textsubscript{Deterministic} applies a fixed set of mutators, e.g., the bitflip, arithmetic, and interesting value mutators, for respectively mutating the bits of each existing “interesting” seed deterministically. After AFL\textsubscript{Deterministic}, all the collected “interesting” seeds are used to launch AFL\textsubscript{Havoc} where random mutations, i.e., randomly chosen mutants, are iteratively applied to the randomly selected bits of the seed inputs.

2.2 Neural Program-smoothing-based Fuzzers

Program smoothing refers to setting up a smooth (i.e., differentiable) surrogate function to approximate program branching behaviors with respect to program inputs [41]. While traditional program smoothing techniques [7, 8] can incur substantial performance overheads due to heavyweight symbolic analysis, integrating such concept with neural network models can be rather powerful since they can be used to cope with high-dimensional optimization tasks,
i.e., to resolve (approximate) complex and structured program behaviors. To this end, Neuzz [41] and MTFuzz [40] are proposed to smooth programs via neural network models and guide mutations by yielding the power of their gradients. Specifically, to formulate the optimization problem for fuzzing, the program branching behaviors are defined as a function \( F(x) \), where \( x \) represents a seed input in terms of byte sequence and the solution is a vector representing its associated branching behaviors. For instance, a solution vector \([1, 0, 1, \ldots]\) indicates that the first and the third edges have been accessed/explored while the second one has not. Since \( F(x) \) is typically discrete, smoothing programs, i.e., making \( F(x) \) differentiable, is essential to cope with the usage of gradients.

We then illustrate the rationale behind Neuzz and MTFuzz. Note that a program execution path, i.e., a sequence of edges, can be determined by the byte sequence of a seed input. Accordingly, an edge can be accessed/explored when the value of its corresponding bytes satisfies its access condition. Otherwise, one of its “sibling” edges (i.e., edges under one shared prefix edge) can be alternatively accessed. For instance, in Figure 1, edge \( e_0 \) can be accessed when the value of \( \text{seed}[i] \) satisfies the access condition for \( e_0 \), i.e., \( \text{seed}[i] \leq 1 \). Hence, mutating such \( \text{seed}[i] \) can lead to exploring a new branching behavior, i.e., accessing \( e_0 \)’s “sibling” edge \( e_1 \) instead of \( e_0 \).

**Figure 1: An example of neural program-smoothing rationale**

Neuzz and MTFuzz assume that neural network models can identify the “promising” byte(s) (i.e., the byte(s) corresponding to the access condition) for a previously explored edge. Specifically, the gradient of such byte(s) (e.g., \( \text{seed}[i] \) in Figure 1) to the explored edge is supposed to be larger than other bytes after training (illustrated in Section 2.2.1). Accordingly, mutating such byte(s) can indicate that the access condition of the corresponding edge may not be satisfied, i.e., potentially exploring new “sibling” edges. To summarize, Neuzz and MTFuzz learn to extract the existing branching behaviors to explore new edges rather than predicting “promising” bytes for unseen edges. In particular, their mechanisms commonly consist of two steps: neural program smoothing and gradient-guided mutations as shown in Figure 2.

2.2.1 Neural Program Smoothing. Neuzz and MTFuzz adopt an iterative training-and-mutation process. Under each iteration, they train neural network models using “interesting” seed inputs collected in real-time (out of the “Seed Corpus” in Figure 2). Note that Figure 2 also shows that Neuzz and MTFuzz adopt different neural network models which will be further illustrated in Section 2.2.3.

2.2.2 Gradient-guided Mutations. After obtaining the neural network models, Neuzz and MTFuzz randomly select a deterministic number of the “interesting” seeds and the explored edges. For each selected seed, they calculate the gradients of the selected edges vectors with respect to all the bytes. Furthermore, all such bytes are sorted according to their corresponding gradient rankings and then aggregated as one vector for further mutations. In particular, Neuzz and MTFuzz segment each selected seed such that the bytes in the front segments have larger gradients than the bytes in the back segments and the front segments include fewer bytes than the back segments. Accordingly, the “promising” bytes are expected to be located in the front segments. For any segment \( \text{seg} \), all its bytes are simultaneously mutated for 255 times. As a result, Neuzz and MTFuzz can explore more mutation space of the front segments than the back ones, i.e., mutating the more “promising” bytes more aggressively, for exploring new branching behaviors. Eventually, all the resulting seeds after the iterative training-and-mutation process are used as test cases for fuzzing.

2.2.3 MTFuzz vs. Neuzz. Figure 2 also demonstrates that MTFuzz differs from Neuzz by adopting multi-task learning technique and a dynamic analysis module to augment its mutation strategy. In addition to the widely-used edge coverage, MTFuzz adopts two additional tasks—the approach-sensitive edge coverage, i.e., how far off an unexplored edge is from getting triggered, and the context-sensitive edge coverage, i.e., the context for an explored edge, to construct the multi-task neural network model for smoothing programs and further guiding fuzzing. Moreover, MTFuzz adopts an independent module, namely Crack in its implementation, which uses dynamic program analysis to explore new edges without gradient information. Specifically, Crack iterates each byte of the seed input and mutates it to observe whether the variables associated with an unexplored branch can be also changed. If so, such byte is identified as a “promising” byte to be mutated for 255 times.

3 EXTENSIVE STUDY

3.1 Benchmarks

Although Neuzz and MTFuzz have been shown to outperform the existing fuzzers in terms of the edge coverage in the original papers [40, 41], such results can be possibly biased by the used subject
projects. For example, 10 popular real-world projects are the main experimental subjects for both Neuzz and MTFuzz; however, it is not clear how such 10 projects are selected and whether the experimental findings can generalize to other real-world projects.

To reduce such threat, we extend the benchmark for evaluating Neuzz and MTFuzz. In particular, in addition to retaining the adopted 9 projects in the original papers (we could not successfully run project 21lb out of the 10 original projects), we also adopt additional 19 projects for our extended evaluations. More specifically, to extend our benchmark projects, we first investigate all the fuzzing papers published in ICSE, ISSTA, FSE, ASE, S&P, CCS, USENIX Security, and NDSS in year 2020 and collect all their benchmark projects. Next, we sort the collected benchmark projects in terms of their usage in all the collected papers (presented in [38]). We then collect the top 30 most used benchmark projects and successfully run 19 of them which are eventually included in our extended benchmarks (the failed executions are mainly caused by environmental inconsistencies and unavailable dependencies). Table 1 presents the statistics of our adopted benchmarks. Specifically, we consider our benchmark to be sufficient and representative due to following reasons: (1) to the best of our knowledge, this is a rather large-scale benchmark suite compared with prior work; (2) the 28 collected benchmarks cover 12 different file formats for seed inputs, e.g., ELF, XML, and JPEG; and (3) the LoC of each program, ranging from 1,886 to over 120K, represents a wide range of program sizes.

3.2 Evaluation Setups
We conduct all our evaluations on Linux version 4.15.0-76-generic Ubuntu18.04 with RTX 2080ti. Following the evaluation setups of Neuzz and MTFuzz, for each selected benchmark project, we first run AFL-2.57b on a single CPU core for 1 hour to initialize our seed collection and then run Neuzz, MTFuzz and all their variants (introduced in later sections) upon the collected seeds with a time budget of 24 hours. Note that all the edges within the 1-hour initial seed collection are excluded from the evaluation results in the remaining sessions. Moreover, we run our experiments for 5 times for each fuzzer and present the average results with close performance under different runs. Note that we instrument all the benchmark projects with afl1-gcc to acquire runtime edge coverage.

In addition to studying Neuzz and MTFuzz, we also include AFL as a baseline technique throughout our extensive evaluations because (1) AFL is widely adopted as baseline by many fuzzing approaches [3, 4, 28, 31, 50] and frequently upgraded for improving its performance; and (2) Neuzz adopts multiple concepts originated from AFL for its implementation [39].

3.3 Research Questions
We investigate the following research questions to extensively study neural program-smoothing-based fuzzing.

- **RQ1:** How do Neuzz and MTFuzz perform on a large-scale dataset? For this RQ, we investigate their effectiveness and efficiency of edge exploration under our large-scale benchmark suite.
- **RQ2:** How do the key components of Neuzz and MTFuzz affect edge exploration? For this RQ, we attempt to investigate how exactly their adopted gradient guidance mechanisms, neural network models, and mutation strategies can affect edge exploration.

3.4 Results and Analysis

3.4.1 RQ1: performance of Neuzz and MTFuzz on a large-scale dataset.
We first investigate the edge coverage performance of all the studied fuzzers. In this paper, following many existing coverage-guided fuzzers [4, 9, 27, 31, 51], we derive the number of the edges via afl1-showmap as our default edge metric. Moreover, note that the edge metric of the original Neuzz and MTFuzz papers can be potentially biased since it counts the byte number of the traces structure implemented by AFL and thus is inconsistent with the results provided by the guidance function (i.e., defining "interesting" seeds mentioned in Section 2.1.2) in their implementations. Nevertheless, as a comprehensive study, we also evaluate all the studied fuzzers in terms of the edge metric of the original Neuzz and MTFuzz papers.

Table 2 presents the edge coverage results of our extensive study for Neuzz and MTFuzz under both adopted metrics. For instance, for AFL under bison, 10.374 corresponds to our default edge metric and 308 corresponds to the original metric in the Neuzz/MTFuzz papers. For our default edge metric, we can observe that Neuzz significantly outperforms AFL by 10.5% (22,395 vs. 20,265 explored edges) in terms of edge coverage on average. Compared with the performance advantage claimed in its original paper (i.e., 2.7X), it is clearly degraded. We then investigate the performance difference among benchmark projects. Interestingly, we can observe that their performance advantage is rather inconsistent, i.e., ranging from -31.2% to 180.5%. Moreover, Neuzz only outperforms AFL upon 10 out of 19 extended projects. Such results suggest that Neuzz cannot
Table 2: Edge coverage results of all the studied approaches

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>AFL</th>
<th>Neuzz</th>
<th>MTFuzz</th>
<th>Neuzz/AFL</th>
<th>MTFuzz/AFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>harfbuzz</td>
<td>26,420</td>
<td>34,528</td>
<td>9,038</td>
<td>3.39</td>
<td>0.27</td>
</tr>
<tr>
<td>objdump</td>
<td>752</td>
<td>2,082</td>
<td>1,215</td>
<td>1.5%</td>
<td>0.6%</td>
</tr>
<tr>
<td>readelf</td>
<td>14,329</td>
<td>34,994</td>
<td>32,791</td>
<td>2.35</td>
<td>2.04</td>
</tr>
<tr>
<td>nm</td>
<td>11,154</td>
<td>34,994</td>
<td>32,791</td>
<td>2.35</td>
<td>2.04</td>
</tr>
<tr>
<td>strip</td>
<td>20,536</td>
<td>32,791</td>
<td>30,074</td>
<td>1.95</td>
<td>1.02</td>
</tr>
<tr>
<td>size</td>
<td>10,730</td>
<td>34,994</td>
<td>32,791</td>
<td>2.35</td>
<td>2.04</td>
</tr>
<tr>
<td>ulibcmp</td>
<td>15,492</td>
<td>34,994</td>
<td>32,791</td>
<td>2.35</td>
<td>2.04</td>
</tr>
<tr>
<td>librpng</td>
<td>8,197</td>
<td>32,791</td>
<td>30,074</td>
<td>1.95</td>
<td>1.02</td>
</tr>
<tr>
<td>harfbuzz</td>
<td>26,420</td>
<td>34,994</td>
<td>32,791</td>
<td>2.35</td>
<td>2.04</td>
</tr>
<tr>
<td>baseetc</td>
<td>13,444</td>
<td>34,994</td>
<td>32,791</td>
<td>2.35</td>
<td>2.04</td>
</tr>
<tr>
<td>mdssum</td>
<td>2,871</td>
<td>34,994</td>
<td>32,791</td>
<td>1.95</td>
<td>1.02</td>
</tr>
<tr>
<td>uniq</td>
<td>713</td>
<td>34,994</td>
<td>32,791</td>
<td>1.95</td>
<td>1.02</td>
</tr>
<tr>
<td>who</td>
<td>2,917</td>
<td>34,994</td>
<td>32,791</td>
<td>1.95</td>
<td>1.02</td>
</tr>
<tr>
<td>Average</td>
<td>20,385</td>
<td>34,994</td>
<td>32,791</td>
<td>1.95</td>
<td>1.02</td>
</tr>
</tbody>
</table>

(a) Neuzz/AFL     (b) MTFuzz/AFL

Figure 3: Edge coverage advantage of the fuzzers over AFL

always outperform AFL and the performance advantage of Neuzz over AFL can be program-dependent.

We also observe that Neuzz outperforms MTFuzz by 1.5% (22,395 vs. 22,070 explored edges) averagely in terms of edge coverage on all benchmark projects. While on 11 of 28 total projects, MTFuzz outperforms Neuzz by 20.8% averagely, Neuzz outperforms MTFuzz by 17.7% on the other 17 projects. Furthermore, even AFL outperforms MTFuzz by 33.6% averagely on a total of 11 projects. Such results indicate that similar to Neuzz, MTFuzz cannot perform consistently either.

We then attempt to reveal the characteristics of how the edge coverage performance varies among the studied projects. To this end, we delineate the correlation between the edge coverage advantage of the studied fuzzers compared with AFL and the size of their studied projects via the Pearson Correlation Coefficient analysis [1]. Figure 3 presents such results of Neuzz and MTFuzz. In each subfigure, the horizontal axis denotes the LoC of each project and the vertical axis denotes the ratio as dividing the edge coverage result of each studied approach by the edge coverage result of AFL. We can observe that overall, the correlation is rather strong (at the significance level of 0.05), i.e., all the studied approaches can result in larger edge coverage improvement over AFL upon larger projects than smaller ones. Such results clearly demonstrate that program size can significantly impact the edge coverage performance of neural program-smoothing-based fuzzers.

We observe similar data trends in terms of the edge metric in the original Neuzz/MTFuzz papers. In particular, Neuzz can outperform AFL by 35.3% (2,219 vs. 1,640 explored edges) and can outperform MTFuzz by 1.8% (2,219 vs. 2,180 explored edges). Note that under such measure, for certain projects, e.g., base64, Neuzz and MTFuzz explore zero edges after excluding the edges from 1-hour initial seed collection. Such results could be misleading that the studied fuzzers perform equally poor in base64, while such performance gaps can be clearly presented by our default edge metric.

Finding 1: The performances of Neuzz and MTFuzz can be largely program-dependent. Interestingly, such program-smoothing-based fuzzers tend to perform better on larger programs.

Note that randomness is injected to many existing fuzzers [4, 27, 31, 50] for selecting bytes to guide mutations, e.g., AFLRand. However, Neuzz and MTFuzz utilize only deterministic mutation strategies, i.e., adopting no randomness for selecting bytes which can be deterministically identified based on their corresponding gradient ranking. Therefore, we further investigate the edge exploration efficiency of random byte selection to infer whether including them in Neuzz and MTFuzz can be potentially beneficial. Specifically, we explore AFL in a fine-grained manner, i.e., its deterministic stage AFL-deterministic and the havoc stage AFL-havoc (i.e., essentially the random byte selection mechanism) both of which enable non-deterministic execution time, for performance comparison with Neuzz and MTFuzz.

Figure 4 presents our evaluation results in terms of the explored edge number per second, namely Edge Discovery Rate (EDR) in this...
paper, of Neuzz, MTFuzz, AFL, AFL-Deterministic and AFL-Hauc. We can observe that overall, Neuzz and MTFuzz can outperform AFL by 10.2% and 8.5% respectively. Interestingly, AFL-Hauc achieves the highest EDR, i.e., 21.8X larger than AFL-Deterministic. 7.7X larger than Neuzz, and 7.8X larger than MTFuzz averagely on all the benchmarks. Accordingly, we can derive that AFL-Hauc can significantly augment edge exploration, i.e., it promptly explores edges upon the limited seed inputs provided by AFL-Deterministic. Such result is enlightening that applying random byte selection mechanism to neural program-smoothing fuzzers can potentially boost edge exploration.

**Finding 2:** AFL-Hauc dominates the efficiency of edge exploration, indicating that it is promising to augment edge exploration by adopting random byte selection mechanism.

**Finding 3:** Although the gradient guidance mechanisms adopted by Neuzz and MTFuzz are overall effective for identifying the promising bytes, their performance can be rather unstable on some programs.

**DNN models.** Now that the gradients derived by Neuzz and MTFuzz can be proven to be effective in reflecting promising bytes for mutations, we further investigate how their corresponding neural network models impact edge exploration. Specifically, since compared to Neuzz, MTFuzz enables the independent dynamic analysis module Crack to augment their mutation strategy, we turn it off and form its variant MTFuzzOff, i.e., applying the mutation strategy of Neuzz in MTFuzz, such that they only differ in the adopted neural network models. Moreover, we also include the Convolutional Neural Network (CNN) [26] model and two commonly-used Recursive Neural Network (RNN) [14] models, i.e., LSTM [24] and Bi-LSTM [22], and adopt them in the original Neuzz to form its variants NeuzzCNN, NeuzzRN, and NeuzzBRN. Note that we investigate more RNN-based models since they are typically used in learning the distribution over a sequence to predict the future symbol sequence [10] (e.g., for speech recognition) and expected to better match the program input features than CNN-based models. Eventually, we determine to evaluate Neuzz and all the variant techniques to detect how multiple neural network models impact the edge exploration of program-smoothing-based fuzzers. Note that their hyper-parameter setups are introduced in our GitHub page [38].

We can observe from Table 2 that overall, all our studied approaches perform similarly in terms of edge coverage. Specifically, Neuzz slightly outperforms MTFuzzOff by 8.5% (22,395 vs 20,648 explored edges), underperforms NeuzzCNN by 1.2% (22,395 vs. 22,665 explored edges), NeuzzRN by 3.3% (22,395 vs. 23,130 explored edges) and NeuzzBRN by 2.2% (22,395 vs. 22,883 explored edges). Meanwhile, we can also observe that none of the studied approaches can dominate on top of all the studied projects, i.e., Neuzz dominates 7, MTFuzzOff dominates 2, NeuzzCNN dominates 4, NeuzzRN dominates 9, and NeuzzBRN dominates 6. Therefore, we derive that upgrading neural network models cannot significantly impact the performance of edge exploration.

**Finding 4:** Different neural network models have limited impact on the effectiveness of program-smoothing-based fuzzing.

**Mutation Strategies.** We then investigate the impact from the mutation strategy of the neural program-smoothing-based fuzzers. Specifically, since MTFuzz differs from Neuzz mainly by enabling Crack for mutations and their respective neural network models do not significantly impact the edge exploration (reflected by Finding 4), we concentrate our investigation on the impact from Crack. To this end, we evaluate MTFuzz and MTFuzzOff. Table 2 demonstrates that overall, MTFuzz can outperform MTFuzzOff by 6.9% (22,070 vs. 20,648 explored edges). However, such advantage can be rather inconsistent, ranging from -2.5% to 47.9% upon individual projects. On the other hand, applying Crack can be potentially cost-ineffective since it is quite heavyweight. Therefore, it is essential...
to consider whether it is worthwhile in applying such technique for neural program-smoothing-based fuzzing.

**Finding 5:** The dynamic analysis module Crack adopted by MTFuzz can be cost-ineffective.

### 3.5 Discussion

We first discuss why neural network models do not significantly impact the edge coverage performance. To this end, we ought to understand the effect of the adopted neural network models of Neuzz and MTFuzz. In particular, note that neural networks are usually used for data prediction, i.e., learning and generalizing historical data to predict unseen data. Accordingly, researchers have developed many neural network models to strengthen their generalization and prediction capabilities. Therefore, one may misunderstand that Neuzz and MTFuzz attempt to use neural network models to predict the bytes corresponding to unexplored edges. Instead, as a matter of fact, Neuzz and MTFuzz leverage neural network models which compute the gradients to reflect the relations between explored edges and seed inputs, i.e., mutating the byte corresponding to a larger gradient can be more likely to explore a new edge other than the existing edge under one shared prefix edge. As a result, any neural network model can be applied as long as it can successfully deliver gradients to reflect such explored edge—seed input relations, i.e., how its generalization or prediction capability does not quite matter under such scenarios. Therefore, it is quite likely that a simplistic model (e.g., feed-forwarded network model adopted by Neuzz) can perform similarly as fine-grained models (e.g., multi-task learning model adopted by MTFuzz and the RNN models adopted by the studied Neuzz variants).

We then attempt to illustrate why Neuzz and MTFuzz cannot always be effective. Note that even though Neuzz and MTFuzz enable gradient guidance mechanisms to explore new edges, their iterative training-and-mutation strategy via randomly selecting edges and seeds in the beginning can nevertheless select existing edges other than unexplored edges to compute gradients (illustrated in Section 2.2.2), i.e., they still allow inefficient mutations. Specifically for the smaller programs where Neuzz and MTFuzz cannot outperform AFL, their edge exploration converges faster than larger programs due to the limited number of edges, i.e., they have a higher chance to select an existing edge whose “sibling” edges have already been explored by other seeds for gradient computation. Thus, it can be difficult to mutate its “promising” bytes for exploring new edges.

### 4 Prefuzz

Our findings reveal that we can possibly leverage the power of the gradient guidance mechanism to enhance the edge exploration of neural program-smoothing-based fuzzers. To this end, we propose Prefuzz (Probabilistic resource-efficient program-smoothing-based Fuzzing). Figure 5 presents the workflow of Prefuzz. Prefuzz first trains a neural network model by applying all the existing seeds as the training set. Next, Prefuzz adopts a resource-efficient edge selection mechanism to select edges for gradient computation. Then, the gradient information is utilized to generate mutants for fuzzing. Note that a mutant which explores new edges can be used as a seed for further edge exploration. Meanwhile, Prefuzz adopts probabilistic byte selection mechanism (PBS in Figure 5) to facilitate mutations.

### 4.1 The Details

#### 4.1.1 Resource-Efficient Edge Selection Mechanism.

The purpose of the resource-efficient edge selection mechanism is to prevent exploring the existing branching behaviors (i.e., edges). To this end, our mechanism is designed to identify the edge worthy being explored for later selecting and mutating its corresponding byte. Intuitively, when one edge can identify the number of its “sibling” edges (as defined in Section 2.2.2), such edge number can be a potential indicator whether the given edge should be included for further gradient computation. More specifically, the more “sibling” edges have been explored, the less likely new “sibling” edges can be explored via the gradient computation for the given edge.

Algorithm 1 presents the details of the resource-efficient edge selection mechanism. First, it is quite essential to acquire the runtime edge exploration states, e.g., the number of “sibling” edges of a given edge and how many have been explored (lines 2 to 3). To this end, we decompile the assembly-level programs, parse them to the instructions via AFL-specific instrumentation, and construct the edge exploration states via statically analyzing the parsed instructions. Next, given one edge, we derive the ratio of its explored “sibling” edge number over its total “sibling” number (lines 5 to 9). If such ratio is lower than a preset threshold, we retain the given edge and stores it in a Candidate Edge Set where we later randomly select such edges for further gradient computation (lines 10 to 12). We use Figure 1 to further illustrate such algorithm. Assuming that $e_0$ can be explored given the “seed” in Figure 1, mutating the byte of the given seed corresponding to the access condition of $e_0$ can explore its “sibling” edge $e_1$. While Neuzz and MTFuzz are designed to perform such mutation for edge exploration, $e_1$ could have nevertheless been explored already due to the randomness injected to their mechanisms (illustrated in Section 2.2.2). Thus, the effectiveness of the gradient guidance mechanism may be compromised. However, our resource-efficient edge selection mechanism can collect the exploration information of the “sibling” edge of $e_0$, i.e., $e_1$, before computing the gradient for $e_0$. If it finds out that $e_1$ has already been explored, it would not select $e_0$ for gradient computation in the first place to save the computing resource.
Algorithm 1 Candidate Edge Set Construction

Input: threshold, exploredEdge
Output: selectedEdges
1: function CONSTRUCT_CANDIDATE_EDGE_SET
2: 
3:  
4: for edge in exploredEdge do
5:   explored ← 0
6:   siblings ← [correspRelation[|edge|]]
7:   for neighbour in correspRelation[edge] do
8:     if neighbour in exploredEdge then
9:       explored ← explored + 1
10:   if explored/siblings < threshold then
11:     candidate.add(edge)
12: return selectedEdges

Table 3: Edge coverage results of PreFuzz

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>AFL</th>
<th>Neuzz</th>
<th>MTFuzz</th>
<th>NeuzzEdgeSelection</th>
<th>NeuzzProb</th>
<th>PreFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>binsh</td>
<td>10,974</td>
<td>12,260</td>
<td>18,812</td>
<td>13,003</td>
<td>13,744</td>
<td>15,078</td>
</tr>
<tr>
<td>xmilibf</td>
<td>13,729</td>
<td>10,499</td>
<td>10,853</td>
<td>12,290</td>
<td>16,960</td>
<td>21,009</td>
</tr>
<tr>
<td>muspdfd</td>
<td>13,665</td>
<td>16,705</td>
<td>16,403</td>
<td>17,002</td>
<td>19,995</td>
<td>21,203</td>
</tr>
<tr>
<td>pngimagex</td>
<td>4,077</td>
<td>3,369</td>
<td>2,437</td>
<td>2,885</td>
<td>2,838</td>
<td>4,876</td>
</tr>
<tr>
<td>pugfix</td>
<td>7,134</td>
<td>5,181</td>
<td>5,767</td>
<td>7,307</td>
<td>7,422</td>
<td>9,390</td>
</tr>
<tr>
<td>pgtest</td>
<td>3,185</td>
<td>2,828</td>
<td>3,166</td>
<td>3,993</td>
<td>4,199</td>
<td>4,703</td>
</tr>
<tr>
<td>tcpdump</td>
<td>12,484</td>
<td>18,295</td>
<td>17,026</td>
<td>18,764</td>
<td>30,767</td>
<td>34,947</td>
</tr>
<tr>
<td>nasm</td>
<td>33,633</td>
<td>34,788</td>
<td>34,958</td>
<td>37,534</td>
<td>41,973</td>
<td>43,628</td>
</tr>
<tr>
<td>tiff2pdf</td>
<td>45,183</td>
<td>47,109</td>
<td>44,765</td>
<td>51,506</td>
<td>50,555</td>
<td>57,172</td>
</tr>
<tr>
<td>tiff2ps</td>
<td>20,862</td>
<td>23,705</td>
<td>21,671</td>
<td>23,649</td>
<td>24,247</td>
<td>29,332</td>
</tr>
<tr>
<td>tiffdump</td>
<td>2,416</td>
<td>3,219</td>
<td>2,617</td>
<td>3,590</td>
<td>3,554</td>
<td>3,888</td>
</tr>
<tr>
<td>tiffinfo</td>
<td>11,964</td>
<td>15,853</td>
<td>13,785</td>
<td>17,157</td>
<td>17,572</td>
<td>21,839</td>
</tr>
<tr>
<td>libxml</td>
<td>20,064</td>
<td>31,340</td>
<td>29,236</td>
<td>32,161</td>
<td>40,955</td>
<td>47,689</td>
</tr>
<tr>
<td>lpbuzz</td>
<td>21,340</td>
<td>17,945</td>
<td>13,382</td>
<td>20,208</td>
<td>29,161</td>
<td>32,447</td>
</tr>
<tr>
<td>lpbuzz_d</td>
<td>31,617</td>
<td>25,006</td>
<td>26,629</td>
<td>26,351</td>
<td>40,355</td>
<td>46,762</td>
</tr>
<tr>
<td>libass</td>
<td>198,976</td>
<td>162,717</td>
<td>132,972</td>
<td>159,936</td>
<td>215,510</td>
<td>218,130</td>
</tr>
<tr>
<td>jhead</td>
<td>2,082</td>
<td>1,433</td>
<td>1,268</td>
<td>1,952</td>
<td>2,463</td>
<td>2,464</td>
</tr>
<tr>
<td>readelf</td>
<td>14,329</td>
<td>40,186</td>
<td>42,173</td>
<td>40,396</td>
<td>47,227</td>
<td>53,859</td>
</tr>
<tr>
<td>nm</td>
<td>11,154</td>
<td>16,159</td>
<td>31,402</td>
<td>19,840</td>
<td>22,605</td>
<td>30,709</td>
</tr>
<tr>
<td>strip</td>
<td>20,536</td>
<td>32,791</td>
<td>41,520</td>
<td>33,846</td>
<td>37,795</td>
<td>42,943</td>
</tr>
<tr>
<td>size</td>
<td>10,730</td>
<td>14,197</td>
<td>18,675</td>
<td>14,734</td>
<td>15,261</td>
<td>19,231</td>
</tr>
<tr>
<td>objdump</td>
<td>15,492</td>
<td>31,808</td>
<td>31,507</td>
<td>34,036</td>
<td>38,983</td>
<td>43,195</td>
</tr>
<tr>
<td>llibjpeg</td>
<td>8,197</td>
<td>16,037</td>
<td>9,058</td>
<td>17,192</td>
<td>22,615</td>
<td>24,365</td>
</tr>
<tr>
<td>harbuzz</td>
<td>26,420</td>
<td>33,502</td>
<td>44,342</td>
<td>47,877</td>
<td>45,412</td>
<td>60,333</td>
</tr>
<tr>
<td>base64</td>
<td>1,344</td>
<td>1,202</td>
<td>935</td>
<td>1,454</td>
<td>1,631</td>
<td>1,644</td>
</tr>
<tr>
<td>md5sum</td>
<td>2,871</td>
<td>3,168</td>
<td>3,101</td>
<td>3,415</td>
<td>3,580</td>
<td>3,535</td>
</tr>
<tr>
<td>uniq</td>
<td>713</td>
<td>796</td>
<td>728</td>
<td>792</td>
<td>794</td>
<td>795</td>
</tr>
<tr>
<td>who</td>
<td>2,917</td>
<td>2,973</td>
<td>2,680</td>
<td>3,262</td>
<td>3,255</td>
<td>3,491</td>
</tr>
<tr>
<td>Average</td>
<td>20,285</td>
<td>22,395</td>
<td>22,070</td>
<td>24,155</td>
<td>28,636</td>
<td>32,042</td>
</tr>
</tbody>
</table>

4.1.2 Probabilistic Byte Selection Mechanism. Inspired by Finding 2, we further inject an additional nondeterministic stage to neural program-smoothing fuzzers. To this end, we develop a Probabilistic Byte Selection Mechanism and append it to Neuzz to expand edge exploration. Note that the probabilistic byte selection mechanism utilizes the gradient information generated by the resource-efficient edge selection mechanism, and gets activated after the mutation stage inherited from Neuzz. This stage contains three steps: (1) dividing each seed input into segments, (2) selecting segments by gradient-based probability distribution, and (3) randomly selecting bytes from the selected segment for mutation via AFLHavoc mutators.

Unlike AFLHavoc which randomly selects bytes from the whole seed, we first divide a seed into a constant number (8 by default in our paper) of equal-length segments. We then select seed segments based on their probabilities. Note that while intuitively leveraging byte-wise probability distribution for byte selection is more natural, this is essentially deterministic and excludes the benefits of randomness (as in Finding 2). Therefore, our probability distribution is established upon seed segments rather than individual bytes so as to leverage the power of randomness and AFLHavoc.

Next, we calculate the fitness score for each segment, presented in Equation 1, where \( \sum_{i=1}^{seg} grad_i \) denotes the gradient sum for all the bytes within a given segment \( seg_i \), \( length(seg_i) \) denotes its byte number, and the fitness score for a given segment \( seg_i \) is computed as the average gradient of all the bytes within \( seg_i \).

\[
fitness_{seg_i} = \sum_{j=1}^{seg_i} grad_j / length(seg_i) \tag{1}
\]

Accordingly, the probability \( Prob_{seg_i} \) for selecting a segment \( seg_i \) for mutation is presented in Equation 2, i.e., the ratio of the fitness score of \( seg_i \) over the total fitness scores of all the segments.

\[
Prob_{seg_i} = \frac{fitness_{seg_i}}{\sum_{j=1}^{total} fitness_{seg_j}} \tag{2}
\]

Finally, we apply AFLHavoc to mutate the selected segments. In particular, AFLHavoc randomly selects a byte from the segment for mutation based on its mechanism. Note that if the mutants are also “interesting”, they are retained for further gradient computation and the probabilistic byte selection mechanism. Such process is iterated until hitting the time budget.

4.2 Performance Evaluation

We attempt to evaluate the performance of PreFuzz and its technical components respectively. To evaluate the usage of the resource-efficient edge selection mechanism and the probabilistic byte selection mechanism, we form two Neuzz variants, i.e., NeuzzEdgeSelection
which injects resource-efficient edge selection mechanism to Neuzz and Neuzz$_{prob}$ which appends the probabilistic byte selection mechanism to Neuzz. Note that we retain Neuzz, MTFuzz, and AFL as our baselines for performance comparison. The experimental setups in this section follow the same settings in Section 3.2. The threshold for Algorithm 1 is set to 0.4$^1$.

4.2.1 Edge exploration effectiveness. Table 3 presents the experimental results of edge exploration effectiveness. We can find that overall, PreFuzz outperforms all the existing baselines in terms of edge coverage averagely, e.g., PreFuzz can outperform AFL by 58.1% (32,042 vs. 20,265 explored edges) and Neuzz by 43.1% (32,042 vs. 22,395 explored edges). Note that under the originally adopted metric of edge coverage, PreFuzz also outperforms Neuzz and MTFuzz by 34.3% and 36.7%. Such results suggest that combining the resource-efficient edge selection mechanism and the probabilistic byte selection mechanism for Neuzz can be rather powerful. Moreover, Neuzz$_{EdgeSelection}$ outperforms Neuzz by 7.9% (24,155 vs. 22,395 explored edges) and MTFuzz by 9.4% (24,155 vs. 22,070 explored edges). Specifically, Neuzz obtains 271 more edges averagely than Neuzz$_{EdgeSelection}$ on 2 projects while Neuzz$_{EdgeSelection}$ obtains 1,917 more edges averagely than Neuzz on the rest 26 projects. Such results indicate that the resource-efficient edge selection mechanism can enhance the overall effectiveness of Neuzz. In addition, Neuzz$_{prob}$ also outperforms Neuzz by 27.9% (28,636 vs. 22,395 explored edges) and MTFuzz by 29.8% (28,636 vs. 22,070 explored edges). Such results demonstrate that introducing randomness can also significantly increase the edge coverage of the neural program-smoothing-based fuzzers.

Figure 6 presents the correlation between the edge coverage advantage of Neuzz$_{edgeSelection}$, Neuzz$_{prob}$, PreFuzz over Neuzz by dividing their corresponding edge coverage results and the LoC of the studied benchmark projects. Interestingly, we can observe that the correlation is rather weak, i.e., all presented $p$ values (0.0688, 0.2211 and 0.1602) fail to reach the significance level of 0.05. It indicates that the edge coverage advantage over the original Neuzz is not affected by the program size. Moreover, such advantage is rather consistent. Specifically, we determine to use Coefficient of Variation (CV) [5], a widely-used metric for measuring the dispersion of a probability distribution [35, 37, 48], to measure the consistency of their performance improvement. Note that a lower CV indicates a more consistent performance improvement. As a result, PreFuzz, Neuzz$_{EdgeSelection}$, and Neuzz$_{prob}$ can achieve 19.6%, 11.5%, and 17.4% of CV for their performance improvement over Neuzz, which are all significantly reduced compared with the CV of Neuzz (37.6%) for its improvement over AFL. Therefore, we summarize that our proposed mechanisms can significantly and consistently strengthen the neural program-smoothing-based fuzzers. Note that we find under the edge coverage metric adopted in the original Neuzz/MTFuzz papers, the performance gain of PreFuzz over Neuzz is 34.3% (2,981 vs. 2,219 explored edges) which is also rather significant.

Figure 7 presents the average time trend of edge coverage within 24 hours for AFL, MTFuzz, Neuzz and PreFuzz among all the benchmark projects. We can observe that at any time being, PreFuzz can outperform other fuzzers significantly in terms of edge coverage.

4.2.2 In-depth Ablation Study. In this section, we further perform in-depth ablation studies to investigate the efficacy of our resource-efficient edge selection mechanism and probabilistic byte selection mechanism respectively. Specifically for the resource-efficient edge selection mechanism, we find that overall, 24.0% edges do not need to be explored by applying Neuzz$_{EdgeSelection}$ under each iteration averagely (1,230 vs. 935 edges). Moreover, the probabilistic byte selection mechanism in PreFuzz is more efficient when combining with the resource-efficient edge selection mechanism since PreFuzz explores averagely 11.9% more edges than Neuzz$_{prob}$ (32,042 vs. 28,636 explored edges in Table 3). Such results indicate that applying the resource-efficient edge selection mechanism can significantly save the effort on exploring the edges which cannot contribute to increasing edge coverage.

We further investigate the probabilistic byte selection mechanism in terms of Edge Discovery Rate. To this end, we also include AFL$_{Havoc}$, Neuzz$_{EdgeSelection}$, the gradient-guided mutation stage of PreFuzz, and the probabilistic byte selection stage of PreFuzz (represented as PreFuzz$_{Gradient}$ and PreFuzz$_{prob}$, respectively) for performance comparison. Note that PreFuzz$_{Gradient}$ and PreFuzz$_{prob}$ results are extracted from the two stages of a complete PreFuzz run, e.g., PreFuzz$_{prob}$ utilizes the resource-efficient edge selection mechanism to select edges for computing their gradients while Neuzz$_{prob}$ randomly selects explored edges for gradient computation. Figure 8 presents our evaluation results. We can observe that overall, PreFuzz$_{prob}$ can significantly outperform all the other studied approaches on top of all the studied benchmarks, e.g., PreFuzz$_{prob}$ can be 62.0% more efficient than AFL$_{Havoc}$ (3.656 vs. 2.256 EDR). Accordingly, we can infer that the gradient guidance adopted by PreFuzz can provide more “high-quality” seeds and more efficient guidance (i.e., gradients) for launching its probabilistic byte selection mechanism to explore more new edges than AFL$_{Havoc}$. Furthermore, we can observe that the EDR of PreFuzz$_{Gradient}$ can also outperform the original Neuzz$_{edgeSelection}$ by 91.4%. Therefore, we also infer that PreFuzz$_{Gradient}$ can advance the edge exploration efficiency of PreFuzz$_{Gradient}$. To summarize, combining the two improvements can mutually advance their edge exploration.

Figure 7: Edge coverage of PreFuzz in terms of time

4.2.3 Crashes. Table 4 presents the unique crashes exposed by Neuzz, MTFuzz and PreFuzz in the studied benchmarks. Overall, PreFuzz explores the most unique crashes by outperforming Neuzz by 62% (149 vs. 92), and MTFuzz by 80% (149 vs. 83). In addition, PreFuzz dominates the number of the exposed unique crashes in each benchmark. Furthermore, the crashes exposed by Neuzz and

$^1$We also evaluate more threshold setups and present the results in our GitHub link [38] which indicate that changing threshold setups incurs limited performance impact.
Table 4: Unique crashes found by Neuzz, MTFuzz and PreFuzz

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Neuzz</th>
<th>MTFuzz</th>
<th>PreFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>5</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>readdelf</td>
<td>15</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>libjpeg</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>objdump</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>who</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>bison</td>
<td>15</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>jhead</td>
<td>8</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>listaction</td>
<td>25</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>listaction_d</td>
<td>7</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>nm</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>strip</td>
<td>10</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>92</td>
<td>83</td>
<td>149</td>
</tr>
</tbody>
</table>

![Figure 8: Edge Discovery Rate of different PreFuzz stages](image)

MTFuzz are also detected by PreFuzz in our evaluation. Such results suggest that PreFuzz can also be more effective than Neuzz and MTFuzz in terms of exposing potential vulnerabilities.

4.3 Implications

Based on our findings in this paper, we propose the following implications for advancing the future research on fuzzing.

Several neural network models may suffice. Our study results reveal that the edge coverage performance can be essentially impacted by how the resulting gradients of the adopted neural network models reflect the relations between explored edges and seed inputs rather than their generalization or prediction capabilities. That said, simplistic neural network models may already suffice for program-smoothing-based fuzzing.

Think twice before applying dynamic analysis. Our evaluations indicate that the dynamic analysis module adopted in MTFuzz can be quite effective on large programs. However, executing such module can be rather heavyweight, similar as many other program analysis techniques [6, 12, 19]. Therefore, we recommend to think carefully before adopting dynamic analysis techniques to enhance neural program-smoothing-based fuzzing.

Edge selection? Yes! Gradient computation? Maybe. Our evaluations reveal that selecting “promising” edges for mutations can be quite effective in increasing the edge coverage performance on programs of varying sizes. Meanwhile, one question can be asked: is it necessary to bind such powerful mechanism with gradient guidance? Especially when we realize that the power of neural networks can be argued to be “underused” (i.e., their generalization and prediction capabilities are underused), such question can then be transformed as — is it necessary to use neural networks for computing gradients to represent the relations between explored edges and seed inputs? To answer such question, it is worthwhile to attempt other lightweight alternatives to represent such relations as potential future research directions.

Probabilistic search helps. Our study results indicate that the edge coverage performance of the neural program-smoothing-based fuzzers can be significantly enhanced by appending the probabilistic byte selection mechanism. Intuitively, we suggest the users to design such probabilistic search strategy with more guidance to any of their adopted fuzzers when possible. Accordingly, one possible research direction can be how to integrate such probabilistic search strategy with diverse fuzzers for optimizing the edge coverage performance.

5 THREATS TO VALIDITY

Threats to internal validity. The threat to internal validity lies in the implementation of the studied fuzzing approaches in the experimental study. To reduce this threat, we reused the source code of Neuzz and MTFuzz when we implemented PreFuzz. Meanwhile, to implement the probabilistic byte selection mechanism, we also reused such code from the original AFL for the PreFuzz implementation. Moreover, all the student authors manually reviewed PreFuzz code carefully to ensure its correctness and consistency.

Threats to external validity. The threat to external validity mainly lies in the benchmarks used. To reduce this threat, we adopt the original benchmarks used by Neuzz and MTFuzz, and add 19 more projects widely used for the evaluations in many popular fuzzers [3, 4, 28, 31, 50] published recently.

Threats to construct validity. The threat to construct validity mainly lies in the metrics used. While the edge coverage metrics adopted by Neuzz and MTFuzz are not widely used by the existing fuzzers and can be arguably limited to reflect edge coverage, to reduce this threat, we determine to follow the majority by using the AFL built-in tool afl-showmap for measuring edge coverage while also presenting partial results in the original measure as well. Notably while under our metric, the performance advantages of Neuzz and MTFuzz are reduced, our PreFuzz can incur quite strong performance gain under both metrics.

6 RELATED WORK

As this work mainly studies deep learning-based fuzzing approaches, we are going to discuss closely related work in the following three dimensions: the existing fuzzing approaches (Section 6.1), the deep learning-based fuzzing techniques (Section 6.2), and the existing studies on fuzzing (Section 6.3).

6.1 Fuzzing

To date, various fuzzing techniques have adopted evolutionary algorithms to improve the performance of fuzz testing. Böhme et al. [4] proposed AFLFast which designs a seed selection strategy to weigh seeds via Markov Chain on top of the original AFL [51]. They also proposed AFLGo [3] to take advantages in weighting seeds based on edge structures to explore the target point specified by users. Lemieux et al. [27] proposed FairFuzz to increase greybox fuzz testing coverage by fuzzing rare branches of program. Manès et al. [32] proposed Ankou, a grey-box fuzzing solution based on different combinations of execution information. Fioraldi et al. [16] introduced WEIZZ to automatically generate and mutate inputs.
for unknown chunk-based binary formats. Similar to their PreFuzz, many works also utilized light-weight program analysis to facilitate fuzzing efficacy. Rawat et al. [36] proposed VUzzer to leverage control- and data-flow features based on static and dynamic analysis to infer fundamental properties of the application without any prior knowledge or input format. Mathis et al. [33] presented a technique to learn program tokens by tainting for fuzzing. Padiyhe et al. [34] automatically guided QuickCheck-like random input generators to semantically analyze test programs for generating test-oriented Java bytecode. Chen et al. [9] introduced Angora, a mutation-based fuzzer that solves path constraint without symbolic execution by taint checking and searching. Furthermore, new guidance other than code coverage are proposed to fuzz specific software systems. Wu et al. [46] proposed Simulee to parse constraints of inputs from a given GPU kernel function and mutate the inputs guided by memory access conflict to fuzz CUDA programs. Accordingly, they further proposed AuCS [47] to repair the detected synchronization bugs. Wen et al. [43] proposed a memory-usage-guided fuzzer to generate excessive memory consumption inputs and trigger uncontrolled memory consumption bugs. Zhao et al. [53] synthesized programs for testing JVMs based on the ingredients extracted from JVM historical bug-revealing tests.

6.2 Deep Learning on Fuzzing
She et al. proposed Neuzz [41], the first neural program-smoothing-based fuzzer using neural network models to discover “promising” bytes for a previously explored edge. They [40] also proposed MTFuzz to fuzz a system more efficiently via a multi-task neural network. Meanwhile, deep learning is also used in evolution-based fuzzing. Zong et al. [55] proposed FuzzGuard, a deep-learning-based approach to help evolution-based fuzzers predict the reachability of inputs before executing programs. Moreover, researchers have also utilized deep learning to learn how to generate valid inputs for deeply fuzzing a system. Lyu et al. [30] introduced SmartSeed which used Generative Adversarial Networks [21] to generate seeds from learning the patterns of valuable existing seeds. Liu et al. [29] proposed DeepFuzz to automatically and continuously generate C programs by a generative Sequence-to-Sequence model [11]. Godefroid et al. [20] divided fuzzing tasks into two categories, i.e., learning input format to fuzz deeper and breaking input format to trigger defects. Zhang et al. [52] proposed DeepRoad to automatically generate driving scenes to fuzz image-based autonomous driving systems. Zhou et al. [54] further generated realistic and continuous physical-world images to fuzz such systems. In this paper, we propose Prefuzz with the resource-efficient edge selection mechanism and the probabilistic byte selection mechanism to improve the performance of neural program-smoothing-based fuzzers.

6.3 Studies on Fuzzing
The empirical studies on fuzzing give many implications for further research. Klees et al. [25] provided guidelines on evaluating the effectiveness of fuzzers by assessing the experimental evaluations carried out by different fuzzers. Gavrilov et al. [17] proposed a new metric consistently with bug-based metrics by conducting a program behavior study during fuzzing. Böhme et al. [2] summarized the challenges and opportunities for fuzzing by studying existing popular fuzzers. Geng et al. [18] performed an empirical study on multiple artificial vulnerability benchmarks to understand how close these benchmarks reflect reality. Herrera et al. [23] investigated and evaluated how seed selection affects a fuzzer’s ability to find bugs in real-world software. Wu et al. [45] studied the features of the havoc mechanism adopted by many fuzzers including AFL, and found it is already a powerful fuzzer which outperforms many existing ones. In this paper, we conduct an empirical study to investigate the power and limitation of neural program-smoothing-based fuzzing and reveal various findings/guidelines for future learning-based fuzzing research.

7 CONCLUSION
In this paper, we investigated the strengths and limitations of neural program-smoothing-based fuzzing approaches, e.g., MTFuzz and Neuzz. We first extended our benchmark suite by including additional projects that were widely adopted in the existing fuzzing evaluations. Next, we evaluated Neuzz and MTFuzz on the extensive benchmark suite to study their effectiveness and efficiency. Inspired by our study findings, we proposed Prefuzz combining two technical improvements, i.e., the resource-efficient edge selection mechanism and the probabilistic byte selection mechanism. The evaluation results demonstrate that Prefuzz can significantly outperform Neuzz and MTFuzz in terms of edge coverage. Furthermore, our results also reveal various findings/guidelines for advancing future fuzzing research.

8 ACKNOWLEDGEMENT
This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Shenzhen Peacock Plan (Grant No. QTVD201611254355531). This work is also partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group.

REFERENCES