Revisiting Neural Program Smoothing for Fuzzing

Maria-Irina Nicolae
Irina.Nicolae@bosch.com
Robert Bosch GmbH
Bosch Center for AI
Stuttgart, Germany

Max Eisele
MaxCamillo.Eisele@bosch.com
Robert Bosch GmbH
Stuttgart, Germany
Saarland University
Saarbrücken, Germany

Andreas Zeller
zeller@cispa.de
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany

ABSTRACT

Testing with randomly generated inputs (fuzzing) has gained significant traction due to its capacity to expose program vulnerabilities automatically. Fuzz testing campaigns generate large amounts of data, making them ideal for the application of machine learning (ML). *Neural program smoothing*, a specific family of ML-guided fuzzers, aims to use a neural network as a smooth approximation of the program target for new test case generation.

In this paper, we conduct the most extensive *evaluation* of neural program smoothing (NPS) fuzzers against standard gray-box fuzzers (>11 CPU years and >5.5 GPU years), and make the following contributions: (1) We find that the original performance claims for NPS fuzzers *do not hold;* a gap we relate to fundamental, implementation, and experimental limitations of prior works. (2) We contribute the first *in-depth analysis* of the contribution of machine learning and gradient-based mutations in NPS. (3) We implement Neuzz++, which shows that addressing the practical limitations of NPS fuzzers improves performance, but that *standard gray-box fuzzers almost always surpass NPS-based fuzzers.* (4) As a consequence, we propose *new guidelines* targeted at benchmarking fuzzing based on machine learning, and present MLFuzz, a platform with GPU access for easy and reproducible evaluation of ML-based fuzzers. Neuzz++, MLFuzz, and all our data are public.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; • **Software and its engineering** → *Software testing and debugging*; • **Computing methodologies** → *Neural networks.*

KEYWORDS

fuzzing, machine learning, neural networks, neural program smoothing

ACM Reference Format:
Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. 2023. Revisiting Neural Program Smoothing for Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616308

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616308

1 INTRODUCTION

In recent years, fuzzing—testing programs with millions of random, automatically generated inputs—has become one of the preferred methods for finding bugs and vulnerabilities in software, mainly due to its speed, low setup efforts, and successful application in the industry. Google's OSSFuzz initiative [23], for instance, has revealed thousands of bugs in open-source software.

Fueled by success stories of practical fuzzing, researchers are constantly seeking ways to make fuzzers more efficient [28]. The most popular approach is still *coverage-guided fuzzing:* generate new test cases from prior ones using an evolutionary search that optimizes code coverage through a fitness function. Techniques used to enhance fuzzers include concolic execution [40, 48], or static analysis [47]. Along them, machine learning methods have increasingly been applied to different parts of the fuzzing loop in academic research [7, 12, 16, 19, 35].

Fuzz testing generates significant amounts of data which make a welcome input for machine learning. Moreover, obtaining labels through feedback from the fuzzer or the program is most often fast and cheap. Constructing a dataset for training machine learning models is thus relatively straightforward in fuzzing. However, despite their increased traction in the research community in the past decade, ML-based fuzzers are not widely used in practice [33].

Recently, *neural program smoothing* [38, 39, 46] has been proposed to approximate the tested program with a neural network. The trained model learns to predict coverage from test cases, being additionally smooth and differentiable. These properties allow computing gradients, which cannot readily be done on programs directly. Test cases are mutated into new ones based on the predictions of the neural network using gradient descent. The use of gradients allows to steer the mutations in the most relevant directions, which have higher chances of reaching new coverage. Despite promising significant performance gains, both in terms of code coverage and number of bugs found, these methods are not currently used by practitioners for testing real software.

Motivated by the applicability of neural program smoothing to real-world fuzzing, we provide a *systematic and thorough analysis of NPS-guided fuzzing methods* with the following contributions:

(1) We provide a critical analysis of NPS-guided fuzzing, uncovering fundamental, conceptual and practical limitations that were previously ignored. We show that *neural network performance does not translate to improved coverage,* as the model fails to capture rare edge coverage.
(2) We compare multiple NPS-guided fuzzers in an extensive benchmark against AFL, AFL++, and the recent $\text{Havoc}_{\text{MAB}}$ on 23 target programs. NPS-guided fuzzers underperform regarding code coverage and bug finding, which is *at odds with*

the results from the original papers. We explain this performance gap by outdated or incorrect experimental practices in prior work.

(3) We reimplement Neuzz as a custom mutator for AFL++ and show that fixing practical limitations of NPS significantly improves fuzzing performance. Nevertheless, we find that neural program smoothing methods *are outperformed by state-of-the-art gray-box fuzzers,* despite their use of additional computation resources.

(4) Based on our findings, we propose *better-suited guidelines* for evaluating ML-enhanced fuzzing, and present *MLFuzz,* the first fuzzing benchmarking framework with GPU support dedicated to ML-based fuzzing. MLFuzz allows for easy, reproducible evaluation of fuzzers with or without machine learning, similar to standard practices used by FuzzBench [33].

The remainder of the paper is structured as follows. Section 2 introduces prior work on coverage guided fuzzing and neural program smoothing, before tackling our main analysis on limitations of neural program smoothing in Section 3. Section 4 presents our implementation of NPS fuzzing and the benchmarking platform. Section 5 covers experiments, followed by new experimental guidelines in Section 6. We conclude this work in Section 7. All our results and code are publicly available (Section 8).

2 BACKGROUND

Coverage-guided fuzzing. Coverage-guided fuzzers explore the input space of a program starting from a few sample inputs called seeds. They mutate the seeds into new test cases based on a *fitness criterion,* which rewards reaching new code coverage obtained by gray-box access through binary instrumentation. Test cases that increase coverage are kept in the corpus to be evolved further. Over time, the input corpus and the total code coverage grow. During execution, the fuzzer checks the target program for unwanted behavior, notably crashes and hangs. Popular coverage-guided fuzzers are American Fuzzy Lop (AFL) [49], its successor AFL++ [18], and libFuzzer [30]. Alongside basic mutations, most gray-box fuzzers use the *havoc* mutation strategy, where a fixed number of randomly chosen atomic mutations are chained to a more complex mutation [18]. Motivated by the success of havoc in modern fuzzers, $\text{Havoc}_{\text{MAB}}$ [45] was designed to implement the havoc strategy as a two-layer multi-armed bandit [4]. Despite the trivial reward function used by the bandit, $\text{Havoc}_{\text{MAB}}$ claims to significantly improve code coverage over random havoc in extensive benchmarks.

Fuzzing with machine learning. ML has been applied to various tasks in the fuzzing loop. Neural byte sieve [35] experiments with multiple types of recurrent neural networks that learn to predict optimal locations in the input files to perform mutations. Angora [12] uses byte-level taint tracking and gradient descent to mutate test cases towards new coverage. FuzzerGym [16] and Böttinger *et al.* [7] formulate fuzzing as a reinforcement learning problem that optimizes coverage. In parallel to mutation generation, machine learning is naturally fit for generating test cases directly. Skyfire [42] learns probabilistic grammars for seed generation. Learn&Fuzz [19] uses a sequence-to-sequence model [41] to implicitly learn a grammar to produce new test cases. GANFuzz [25] uses generative adversarial networks (GANs) [20] to do the same for protocols. DeepFuzz [29] learns to generate valid C programs based on a sequence-to-sequence model for compiler fuzz testing. The application of ML to fuzzing is covered more extensively in [36, 44].

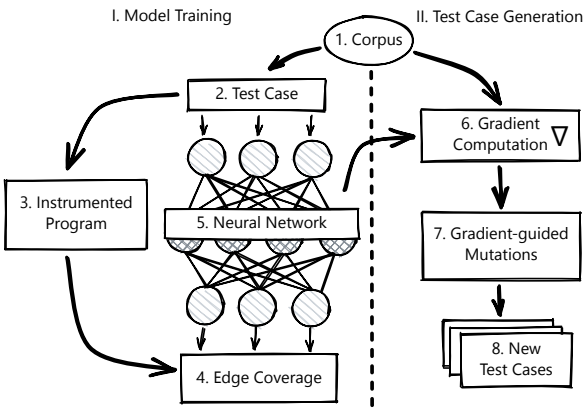

Figure 1: Neural program smoothing for fuzzing.

Neural program smoothing. Program smoothing [10, 11] was initially introduced as a way to facilitate program analysis and overcome the challenges introduced by program discontinuities. Among the uses of machine learning in fuzzing, neural program smoothing is one of the most recent and popular methods, due to its great performance in the original studies. Neuzz [39] trains a neural network to serve as a smooth approximation of the original program in terms of code coverage (Figure 1). First, all test cases (2) from the corpus (1) are executed on the instrumented program (3) to obtain their individual code coverage (4), i.e. edge coverage from `afl-showmap`. The respective pairs of test case and coverage are then used to train a neural network (5), which learns to predict the coverage for each test case. Being smooth and differentiable, the neural network can be used for computing *gradients*, the values of derivatives of the program w.r.t. its inputs. These indicate the direction and rate of fastest increase in the function value and can be used to flip specific edges in the bitmap from zero to one (6). Each gradient corresponds to one byte in the input. The locations with the highest gradient values are mutated (7) to propose new test cases (8) that should reach the targeted regions of the code. This idea is inspired by adversarial examples, more precisely FGSM [21], where a change in the input in the direction of the sign of the gradient is sufficient to change the model outcome.

MTFuzz [38] extends Neuzz with multitask learning [8]: the neural network is trained against three types of code coverage instead of only edge coverage. *Context-sensitive coverage* [12, 43] distinguishes between distinct caller locations for the same covered edge, while *approach-sensitive coverage* [2] introduces a third possible value in the coverage bitmap reflecting when an edge was nearly covered because the execution has reached a neighboring edge. The three types of coverage help learn a joint embedding that is used to determine interesting bytes for mutation in the test case. The bytes are ranked using a saliency score, which is computed as the sum of gradients for that byte in the learned embedding space. Each "hot byte" is mutated by trying out all possible values, without further relying on the gradients.

PreFuzz [46] attempts to solve some limitations of Neuzz and MTFuzz by extending Neuzz in two ways. The program instrumentation is changed to include all neighboring edges of covered ones in the bitmap. This information is used to probabilistically choose which edge to target next for coverage, with the end goal of encouraging diversity in edge exploration. Additionally, the success of havoc mutations [18] is leveraged: after the standard Neuzz mutation, havoc is applied probabilistically to pre-defined segments of bytes in the test case, according to their gradient value.

3 ANALYZING NEURAL PROGRAM SMOOTHING

In this section, we provide our main analysis of neural program smoothing, covering both the concepts behind NPS, as well as existing fuzzer implementations. We tackle three orthogonal perspectives: (i) conceptual or fundamental, (ii) implementation and usability, and (iii) experimental considerations.

3.1 Conceptual Limitations

(C1) Approximation errors of the neural network. Being an empirical process, neural network training can suffer from errors introduced in the training process by, e.g., limited training data and training time, or sensitivity to hyperparameters. Even in the ideal case, *being a smooth approximation, the NPS model will always differ from the actual program exactly at the most interesting points*, i.e., discontinuities, branches, and jumps. This approximation error is intrinsic to a smoothing approach and, at the same time, what allows NPS methods to use gradients and numeric optimization towards producing new inputs.

(C2) Capacity to reach targeted edges. Arguably, the most salient research question to elucidate about neural program smoothing is whether the gradient-guided mutation can indeed reach the targeted edges. As NPS is based on multiple components (Figure 1), the overall performance of the fuzzer critically depends on the effectiveness of its individual components:

(1) The prediction accuracy of the neural network (5);
(2) The capacity of the gradient-based mutations (7) to achieve the expected new coverage on the target program.

The experiments we perform later in the paper show that the machine learning component as used by neural program smoothing has impaired performance. To the best of our knowledge, prior NPS studies have not assessed what the model was learning and whether it was reaching its objective.

(C3) Incomplete coverage bitmaps. Another central limitation of neural program smoothing that we uncover relates to the incompleteness of the coverage bitmaps that the neural network receives. All NPS fuzzers retrieve covered edges through `afl-showmap`, which only reports the edge IDs that are reached. When the coverage information from all seeds is put together for the overall bitmap used for training the neural network, it thus only contains edges that were reached at least once by any of the seeds. As such, unseen edges are not part of the bitmap and cannot be explicitly targeted and discovered by the model. In practice, if the neural network does discover new edges, it is rather inadvertently due to randomness. While having access to only an incomplete coverage bitmap is a conceptual limitation, it can be addressed on an implementation level. It is sufficient to change the instrumentation of the program to include uncovered edges to overcome this issue. Among existing NPS fuzzers, PreFuzz is the only one that considers information about neighbors of reached edges in the coverage bitmap, albeit not motivated by the limitation we uncover. Their goal is rather to be able to choose the next edge to target in a probabilistic fashion, depending on the degree of coverage of each edge and its neighbors.

The fundamental limitations uncovered in this section, while some easier to solve than others, are what we see as main obstacle in the adoption of NPS-based fuzzing in practice. As will be confirmed in Section 5, the experiments are consistent with these limitations.

3.2 Implementation and Usability Limitations

We now turn to practical aspects that make existing approaches to neural program smoothing inconvenient to use, such that an independent evaluation requires major effort and code rewriting.

(I1) Use of outdated components. Existing implementations of neural program smoothing [38, 39, 46], along with $\text{Havoc}_{\text{MAB}}$ [45] are implemented as extensions of AFL instead of using the more recent, more performant AFL++ as base. Moreover, their dependency on outdated Python, TensorFlow and PyTorch versions impacts usability. For the purpose of experiments, we have patched the code and updated the dependencies of all these fuzzers, as even for the most recent ones, some of their used libraries were already not available at the time of their publication.

(I2) Difficulty in building targets. Prior NPS studies provided the binaries used in their own research, ensuring reproducibility. However, for a fuzzer to be practical, it is advisable to rather provide instructions on how to build new programs for its use. This is especially important when the fuzzer uses custom target instrumentation. MTFuzz [38], for instance, compiles a target program in five different ways due to the introduction of three additional types of instrumentation. For this reason, we exclude MTFuzz from our empirical study as not being practical for real-world fuzzing. Moreover, we argue that the three types of coverage used by MTFuzz are to a large extent redundant (conceptual limitation) and could be grouped into a unified coverage, thus reducing the build effort for this fuzzer.

(I3) Use of magic numbers. The magic numbers programming antipattern [31] is frequently encountered in the implementations of neural program smoothing-based fuzzers. These values and other algorithmic changes are not mentioned in the original papers where each NPS fuzzer is introduced. It is thus difficult to establish whether the performance of each method is strictly linked to its proposed algorithm or rather to the implementation tweaks. E.g., the maximum number of mutation guiding gradients per seed is set to 500; this value is not a parameter of the algorithm presented in the paper.

Our findings above show that the effort to set up existing NPS fuzzers and build targets for them is significantly higher than for standard gray-box fuzzers, such as AFL and its variants, or libFuzzer.

3.3 Evaluation Limitations

In this section, we highlight flaws and limitations of previous experimental evaluations of NPS fuzzers and $\text{Havoc}_{\text{MAB}}$, which have led to unrealistic performance claims.

(E1) Experimental protocol. The more recent NPS publications [38, 46] *lack of comparisons with recent gray-box fuzzers,* such as AFL++ and libFuzzer—fuzzers that were available and confirmed as state-of-the-art long before their publication. Havoc$_{MAB}$ [45] has included Neuzz and MTFuzz in their evaluation alongside AFL++. However, we find that they use the same binary target for both AFL and AFL++, instead of building the program separately for AFL++. AFL++ runs on AFL instrumented binaries, but not efficiently. Moreover, the size of the coverage bitmap is usually larger for AFL++ than with AFL instrumentation; hence, code coverage as measured by the fuzzers is not directly comparable. This makes the conclusions in the Havoc$_{MAB}$ evaluation [45] questionable.

(E2) Fuzzer configuration for speed. We note that prior studies benchmarking NPS methods compile their targets using `afl-gcc`, which results in slower targets and thus impacts fuzzing speed. The AFL++ documentation recommends using preferably `afl-clang-fast` or `afl-clang-lto` [17]. Additionally, AFL-based fuzzers have multiple options for transferring fuzz data to the program. The most basic is to have AFL write test cases to file, and the target program executed with command line options to process the file as input. The more sophisticated and recommended *persistent* mode uses a fuzzing harness that repeatedly fetches fuzz data from AFL via shared memory and executes the function with the test data as input without restarting the whole program. "*All professional fuzzing uses this mode*", according to the AFL++ manual [5]. Depending on the target, the persistent mode can increase the throughput by 2–20× [18]. Previous neural smoothing papers seem to run all experiments by feeding inputs via files, which should considerably slow down all fuzzers. This is consistent with their results, where the more modern AFL++ consistently performs worse than AFL in the Havoc$_{MAB}$ study [45], and the targets are printed with command line arguments in the original Neuzz paper [39]. We conjecture that this tips the scale in favor of ML-based fuzzers, which are themselves orders of magnitude slower than modern fuzzers [16]. This statement is validated experimentally in Section 5.7.

4 IMPLEMENTING NEUZZ++ AND MLFUZZ

In this section, we introduce *Neuzz++,* our implementation of neural program smoothing that aims to solve some limitations identified in Section 3, as well as the new experimental platform for evaluating ML-based fuzzers.

Neuzz++. We implement a variation of Neuzz as a custom mutator for AFL++, which we name Neuzz++ (see Figure 2). This allows our method to leverage most AFL++ features, like its standard mutations and power schedule. More importantly, it allows for machine learning-produced test cases and randomly mutated ones to evolve from each other. We choose AFL++ as base for our implementation for its state-of-the-art performance, thus addressing Issue I1. Being a custom mutator, Neuzz++ is modular, easy to build, and integrated with a default AFL++ installation.

In practice, Neuzz++ consists of two parts: the main AFL++ process with the custom mutator implemented in C, and a Python extension that is called for machine learning operations. The two processes communicate using named pipes. We set a minimum requirement of T test cases in the corpus for the custom mutator to

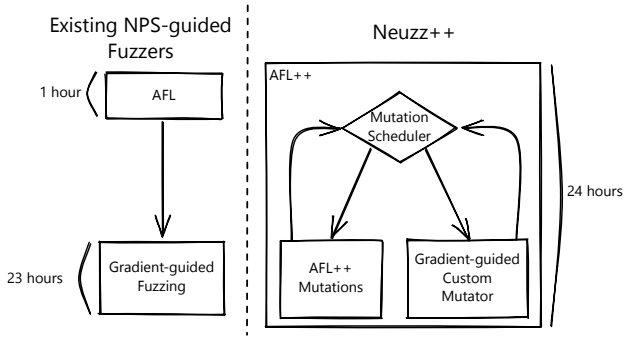

Figure 2: Operation mode of previous NPS-guided fuzzers and our Neuzz++.

run. These are used to train the neural network for the first time; the model is retrained at most every hour if at least ten new test cases have been added to the corpus[1]. This allows to refine the model over time with new coverage information from recent test cases. In practice, we use $T = 200$; this value is tuned experimentally and aims to strike the balance across all targets between fuzzing with machine learning as early as possible, while waiting for enough data to be available for model training. Intuitively, a larger dataset produces a better performing model. `afl-showmap` is used to extract the coverage bitmap. We introduce a coverage caching mechanism for model retraining which ensures that coverage is computed only for new test cases that were produced since last model training. Each time the C custom mutator is called by AFL++, it waits for the Python component to compute and send the gradients of the test case. Based on these, the mutations are computed by the C mutator and returned to AFL++. In contrast to Neuzz, the gradients are not precomputed per test case, they are not saved to disk, the neural network is kept in memory, and the gradients are computed only on demand. These optimizations minimize the time spent on ML-related computations, keeping more time for fuzzing.

The neural network is a multi-layer perceptron (MLP) with the same structure as Neuzz (one hidden layer, 4096 neurons). As shown in the PreFuzz paper [46], we also found that different neural network architectures do not improve fuzzing performance. In contrast to NPS fuzzers, we keep 10% of the test cases as validation set for evaluating the performance of the model. We use the Adam optimizer [26], a learning rate of 10^{-4}, and cosine decay with restarts.

It is easy to parallelize model training and the main AFL++ routine for improved fuzzing effectiveness when testing real targets. However, for experimental evaluation, we choose to have AFL++ wait for the neural network to train, similarly to previous implementations of neural program smoothing fuzzers. This allows for fair experimental comparison and computation resource allocation.

The original Neuzz implementation applies four different mutation patterns on each byte selected according to the highest ranking gradients: incrementing the byte value until 255, decrementing the byte value down to 0, inserting a randomly sized chunk at the byte location, and deleting a randomly sized chunk starting at the given byte location. We apply the same mutation pattern for Neuzz++.

[1]Neuzz and PreFuzz solve this issue by running AFL for the first hour of fuzzing, then use the collected data for model training (Figure 2).

MLFuzz. MLFuzz serves as a benchmarking framework for building test targets, running fuzzing trials in an isolated environment, and analyzing the findings. Its main features are:

- Test targets from Google Fuzzer Test Suite [22] are compiled with the recommended and most recent compiler of the appropriate fuzzer; the build scripts are made available (addressing Issue I2 and issue E2).
- Targets are compiled with AddressSanitizer [37] to detect memory errors.
- Six fuzzers are currently included in MLFuzz: AFL v2.57b, AFL++ v3.15a, $\text{Havoc}_{\text{MAB}}$, Neuzz, PreFuzz and our Neuzz++.
- The implementation is containerized via Docker [32]. Python dependency specification is handled via virtual environments and Poetry [15].
- Each fuzzing trial runs on one dedicated CPU and optionally one GPU for fuzzers that support it.
- All supported fuzzers have been modified to accept seeding of their random number generator for reproducible results.
- For all fuzzers, coverage is measured by replaying the corpus at the end of a run. We use binaries instrumented with AFL to ensure we do not disadvantage the AFL-based fuzzers, and `afl-showmap` from AFL++, since it has a larger bitmap with less hash collisions.
- Test cases are transmitted to fuzzers via shared memory, with the option to switch to slow transmission of test cases via the file system (addresses Issue E2).

5 EXPERIMENTS

This section introduces our experiments and practical analysis, complementing the main findings from previous sections. After presenting our setup (Section 5.1), we assess the performance of the components of NPS-based fuzzers in Section 5.2. We compare our Neuzz++ to prior neural program smoothing fuzzers and standard gray-box fuzzers in an extensive benchmark in Section 5.3. Sections 5.4 to 5.6 explore the added benefit of machine learning to NPS fuzzers, while Section 5.7 sheds light on experimental protocol differences with previous NPS publications and their impact on fuzzing results. Finally, we report bugs found in Section 5.8.

5.1 Experimental Setup

All experiments are performed on a server running Ubuntu 20.04 with four Nvidia Titan Xp GPUs. Our study includes the six fuzzers from MLFuzz: AFL and AFL++ as standard gray-box fuzzers, $\text{Havoc}_{\text{MAB}}$ as recent fuzzer claiming state-of-the-art performance, and NPS fuzzers Neuzz, PreFuzz, and our own Neuzz++. We use the original implementation and parameters provided by the authors for all baselines, except when stated otherwise. We patch the code of Neuzz and PreFuzz to port them to Python 3.8.1, CUDA 11.5, TensorFlow 2.9.1 [1] and PyTorch 1.4 [34], as the original implementations are based on outdated libraries that are not available anymore or incompatible with our hardware.

We choose Google Fuzzer Test Suite [22] and FuzzBench [33] as standard, extensive benchmarks for our experimental evaluation. We make use of 23 targets, summarized in Table 1. These are selected for being accessible, having dependencies available on Ubuntu 20.04, and being non-trivial to cover through fuzz testing. Note that we only include targets from FuzzBench if they are not already included in Fuzzer Test Suite. All results are reported for 24 hours of fuzzing. We repeat each experiment 30 times to account for randomness, unless stated otherwise. Each standard gray-box fuzzer is bound to one CPU core, while NPS fuzzers are allotted one CPU and one GPU per trial. The main metrics used for evaluation are code coverage and number of bugs found. For code coverage, we use edge coverage as defined by the AFL family of fuzzers. However, we emphasize that AFL and AFL++ compute edge coverage differently. In order to avoid the measuring errors introduced when ignoring this aspect, we count coverage by replaying the corpus using `afl-showmap` from AFL++ on the same binary, independently of which fuzzer was used in the experiment. The setup we use fixes all experimental limitations we highlighted in Section 3.3 (Issues E1 and E2).

Table 1: Target programs from Google Fuzzer Test Suite [22] and FuzzBench [33].

Target	Format	Seeds[a]	LOC[b]
Source: Fuzzer Test Suite			
boringssl-2016-02-12	SSL private key	107	102793
freetype2-2017	TTF, OTF, WOFF	2	95576c
guetzli-2017-3-30	JPEG	2	6045
harfbuzz-1.3.2	TTF, OTF, TTC	58	21413
json-2017-02-12	JSON	1	23328
lcms-2017-03-21	ICC profile	1	33920
libarchive-2017-01-04	archive formats	1	141563
libjpeg-turbo-07-2017	JPEG	1	35922
libpng-1.2.56	PNG	1	24621
libxml2-v2.9.2	XML	0	203166
openssl-1.0.2d	DER certificate	0	262547
pcre2-10.00	PERL regex	0	67333
proj4-2017-08-14	custom	44	6156
re2-2014-12-09	custom	0	21398
sqlite-2016-11-14	custom	0	122271
vorbis-2017-12-11	OGG	1	17584
woff2-2016-05-06	WOFF	62	2948
Source: FuzzBench			
bloaty	ELF, Mach-O, etc.	94	690642
curl	comms. formats	41	153882
libpcap	PCAP	1287	56663
openh264	H.264	174	97352
stb	image formats	467	71707
zlib	zlib compressed	1	30860

[a]Targets that do not have seeds use the default from Fuzzbench.
[b]Retrieved with `cloc` [14].

5.2 Performance of Machine Learning Models

We now investigate the quality of coverage predictions by the neural network and gradient-based mutations, in relation to concerns about the fundamental principle of neural program smoothing (Section 3.1). We tackle the following questions:

- Can the neural network learn to predict edge coverage?
- Can gradient-based mutations reach targeted edges?

To this end, we propose quantitative and qualitative analyses of the performance of the neural network in neural program smoothing fuzzers. Without loss of generality, we investigate these based on Neuzz++ as a proxy for all neural program smoothing fuzzers

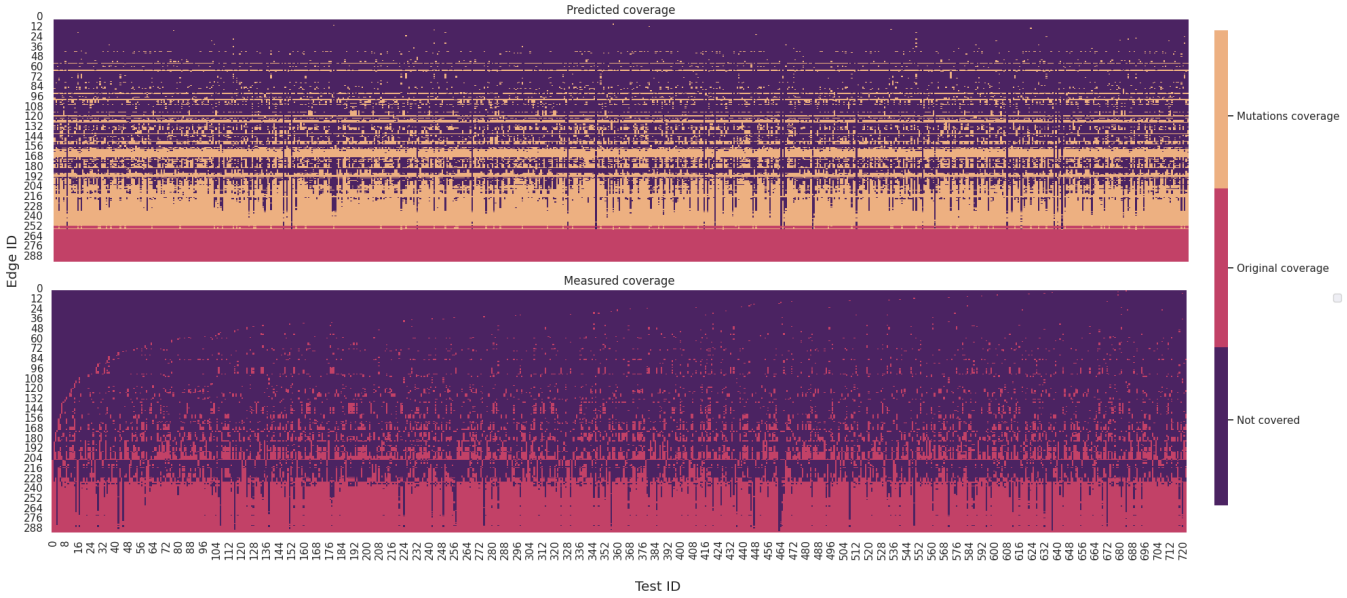

Figure 3: Predicted and actual edge coverage on *libpng* for the entire corpus. Top: ML-predicted coverage (pink) is trivial and almost constant over test cases. When each edge is targeted by mutations, predicted coverage (orange) increases for certain edges, but many code edges remain unattainable. Bottom: Coverage extracted with `afl-showmap` shows that all edges present have been covered at least once by the corpus.

Table 2: Dataset properties and neural network evaluation.

Target	%covered edges	Acc	Prec	Recall	F1	PR-AUC
bloaty	17.1%	0.53	0.17	0.18	0.17	0.15
boringssl	19.3%	0.90	0.18	0.17	0.17	0.20
curl	15.2%	0.89	0.15	0.15	0.15	0.23
freetype2	8.6%	0.89	0.09	0.09	0.09	0.10
guetzli	18.5%	0.84	0.18	0.18	0.18	0.19
harfbuzz	6.9%	0.93	0.07	0.07	0.07	0.07
json	12.7%	0.88	0.11	0.08	0.09	0.10
lcms	20.9%	0.84	0.19	0.19	0.19	0.21
libarchive	6.9%	0.94	0.07	0.06	0.06	0.07
libjpeg	17.8%	0.84	0.17	0.09	0.17	0.18
libpcap	6.4%	0.92	0.06	0.06	0.06	0.07
libpng	28.8%	0.86	0.28	0.27	0.27	0.29
libxml2	10.5%	0.92	0.10	0.09	0.09	0.11
openh264	21.4%	0.81	0.22	0.30	0.21	0.22
openssl	31.2%	0.79	0.30	0.30	0.29	0.31
pcre2	4.3%	0.96	0.04	0.03	0.03	0.04
proj4	8.2%	0.95	0.08	0.07	0.07	0.08
re2	16.2%	0.87	0.15	0.13	0.13	0.16
sqlite	16.3%	0.91	0.12	0.12	0.12	0.17
stb	6.0%	0.92	0.06	0.05	0.05	0.06
vorbis	29.6%	0.81	0.30	0.30	0.30	0.30
woff2	22.8%	0.85	0.22	0.22	0.21	0.13
zlib	16.1%	0.85	0.14	0.10	0.11	0.16

included in our study. As all these methods use the same neural network architecture, loss function, method of training, etc., it is to be expected that their models will achieve the same performance when trained on the same dataset. The results of the analyses can be summarized as follows and are detailed subsequently:

- Table 2 quantifies the model performance for all targets in terms of standard machine learning metrics;
- Figure 3 provides a qualitative analysis of model predictions for a given target, opposing them to correct labels.
- Lastly, Figure 3 also assesses the capacity of the neural network to reach edges through gradient-based mutations.

ML performance metrics. To assess one factor of difficulty of the machine learning task, we evaluate dataset imbalance for the training corpus. This measures the percentage of the positive class (covered edges, in our case the minority) in the coverage bitmap of the training set. Recall that the bitmap is produced by `afl-showmap` and accounts for the coverage obtained by the corpus before training; the coverage was not necessarily achieved based on a neural network, but rather by AFL++ mutations. Note that this value is averaged across test cases and edges; rare edges might have much smaller coverage ratios, resulting in more difficulty in training an accurate model for those edges. When facing class imbalance, the model tends to prefer the majority class, thus making wrong predictions. For this reason, the performance of the neural network is assessed using precision, recall, F1-score, and precision-recall (PR) trade-off as performance metrics for the neural network. Accuracy is also computed for completeness, but keep in mind that this metric is misleading for imbalanced datasets[2]. We measure the area-under-the-curve (AUC) of the PR metric to evaluate all the operational points of the neural network. Similar to accuracy, PR-AUC saturates at one, but is more sensitive to wrong predictions in the positive class. The learning setup of neural program smoothing is a multi-label binary classification task, i.e., for each

[2]One can trivially predict all-zeros (no coverage) and obtain very high accuracy.

test case, multiple binary predictions are made, one per edge; in consequence, the metrics are computed for each edge in the bitmap independently, then averaged over all edges, and finally averaged over trial repetitions.

Table 2 reports the model performance metrics, along with the percentage of the positive class in the dataset as imbalance metric. All model metrics are computed on a 10% holdout set of test cases that were not used for training. As Neuzz++ retrains the model multiple times, all measurements are performed on the last trained neural network using the state of the corpus at that time. The precision, recall, F1-score, and PR-AUC values in Table 2 indicate that the neural network has low performance. These metrics are particularly low when the class imbalance is stronger, i.e., for small values of "%covered edges". The dataset imbalance is quite extreme for seven targets, where the positive class represents less than 10% of the dataset, making predictions particularly difficult.

To provide an intuition into what the neural network learns, we design a qualitative evaluation of its predicted coverage. This experiment uses the target *libpng* and the test cases generated in a 24-hours run of Neuzz++. Figure 3 shows two coverage plots for this target for the entire corpus, where each "column" in the plot represents one test case, while each "row" is a program edge. We compare the coverage predicted by a trained ML model for the same test cases and edges (Figure 3 top) to the true coverage extracted with `afl-showmap` (bottom). The bottom plot is the coverage bitmap extracted with `afl-showmap` for the corpus and used for model training by Neuzz, PreFuzz, and Neuzz++. A reduction (deduplication) operation is applied to it, which for *libpng* reduces the number of edges from 900 to the 293 present in the plot; this operation also explains any visual artifacts present in the image, as the edges are reordered. The pink areas of the two plots differ significantly, with the model predictions being almost constant over all test cases: the model only predicts trivial coverage and fails to capture rare edges. While this is a consequence of the difficulty of the machine learning tasks (small dataset, class imbalance, too few samples w.r.t. the size of the test cases and bitmaps, see Table 2), it results in large approximation errors in the neural network, as outlined in Issue C1. Moreover, recall that Neuzz, PreFuzz and Neuzz++ use the same ML model type and structure, with minor differences in the training procedure and similar model performance. Our findings thus extend to all NPS methods.

Finally, we investigate the effectiveness of gradient-based mutations as essential component of NPS fuzzers. In the same setup on *libpng* from the previous section, we apply Neuzz++ mutations to the corpus generated by a 24-hours fuzzing run as follows. For each edge in the bitmap, we consider the case when it is explicitly targeted and generate all mutations with a maximum number of iterations in the mutation strategy. Figure 3 (top) plots the predicted coverage for each test case and edge before the mutations, as well as the increment of coverage after mutation. Each edge (row) is considered covered by one test case (column) if at least one of the few thousand mutations generated to target it reaches the code location. The results represent coverage estimated by the ML model, not run on the program. However, the coverage the model predicts is an optimistic estimate of the one actually achieved on the target, as the model dictated the mutations. Note that the mutations are generated in the same way for Neuzz, PreFuzz and Neuzz++; our analysis thus applies to all methods and targets.

Figure 3 (top) indicates that some locations are more readily reachable through mutations. The harder to reach edges overall match the rarer edges of the corpus, as measured by `afl-showmap` in the bottom plot. Most importantly, *none of the edges targeted or covered by the mutations in the top plot represent new coverage.* Recall that, by NPS methods' design, a code edge is only present in the bitmap only if it has already been covered by the initial corpus used for training (Issue C3). This becomes evident in the bottom plot of Figure 3: all edges have been covered by at least one test case. As will be shown later, this fundamental flaw of NPS methods translates to a limited practical capacity of reaching new coverage.

> *The model predicts trivial edge coverage (Issue C1), and gradient mutations cannot target new edges (Issue C3).*

5.3 Comparing Code Coverage

We present the main experiment comparing the achieved code coverage of available neural program smoothing approaches to AFL, AFL++ and the recent $\text{Havoc}_{\text{MAB}}$ in Table 3 (average coverage) and Figure 4 (coverage over time). This experiment alone requires a total computation time of over 11 CPU years and 5.5 GPU years.

Overall, AFL++ obtains the best performance on ten targets, followed by $\text{Havoc}_{\text{MAB}}$ with eight targets, and Neuzz++ on par with AFL, winning two targets each. In view of AFL++ performance w.r.t. AFL, it is clear that not including AFL++ as a baseline in all prior neural program smoothing works leads to overly optimistic conclusions about their capacities. After AFL++, $\text{Havoc}_{\text{MAB}}$ is the second most performant fuzzer in terms of code coverage. However, we find that it does not reach the expected ranking advertised in the $\text{Havoc}_{\text{MAB}}$ paper [45].

We observe that Neuzz and PreFuzz are never in the top two fuzzers. Moreover, although they were designed to improve AFL performance, their coverage is in most cases lower than that of AFL. AFL wins on 20 out of 23 targets over Neuzz, and 18 out of 23 over PreFuzz. PreFuzz outperforms Neuzz on most targets, however this difference is significant only on six targets (see confidence intervals in Figure 4). This finding is also at odds with original PreFuzz results [46], where the performance gap is significantly wider. Section 5.7 is dedicated to further explaining the difference in performance with the initial papers. Neuzz++ obtains higher coverage than Neuzz and PreFuzz on 21 programs, proving that our improvements over these methods are effective.

Targets *libarchive, libxml2, proj4,* and *woff2* exhibit the most variability among fuzzers. Neuzz and PreFuzz exhibit large standard deviation on *woff2,* where coverage varies depending if the fuzzers reach plateau or not. For the other targets, it seems AFL-based fuzzers do not perform as well as AFL++-based ones.

> *Overall, AFL++ achieves the highest code coverage. Among NPS fuzzers, Neuzz++ achieves the highest code coverage.*

Table 3: Average edge coverage and standard deviation over 30 runs. Best value in bold, second best underlined.

Target	AFL	AFL++	Havoc$_{MAB}$	Neuzz	PreFuzz	**Neuzz++**
bloaty	14220±49	**15607**±100	15240±194	12518±790	12936±319	15296±196
boringssl	2936±34	2940±34	**2956**±1	2863±14	2867±21	2930±32
curl	13002±1103	13398±1376	**14121**±324	8999±211	9048±218	11260±1401
freetype2	10722±126	11090±104	**11408**±95	8569±281	8870±386	10960±138
guetzli	7306±21	6772±9	**7398**±29	7099±26	7141±45	6702±7
harfbuzz	**12056**±137	11887±137	11953±146	10672±110	10875±102	11654±65
json	2033±5	2018±11	**2036**±0	1974±79	1970±86	2032±12
lcms	**2483**±198	1904±441	2423±277	1593±372	1876±433	1809±455
libarchive	3708±383	**5281**±207	4970±153	3729±289	3718±225	5246±204
libjpeg	2685±82	**3058**±161	2980±192	2647±5	2664±60	2892±189
libpcap	2203±219	3733±115	2833±243	1859±358	1875±373	3529±155
libpng	1234±7	1235±3	1240±2	1220±6	1219±7	**1241**±3
libxml2	4857±286	**9155**±989	5416±273	4895±403	4853±271	7306±1191
openh264	13381±341	**15234**±15	14902±109	14135±241	14537±142	15126±80
openssl	1891±6	**1899**±1	1894±4	1878±7	1884±8	1886±8
pcre2	7797±142	7960±142	**8076**±98	7555±76	7575±77	7763±67
proj4	1837±1621	**5585**±101	4190±741	1526±514	1849±392	4550±78
re2	6680±52	6717±7	**6777**±26	6497±162	6547±110	6731±30
sqlite	2004±154	2123±12	2121±0	1982±158	2025±146	**2125**±16
stb	3305±109	3390±11	**3413**±20	3286±18	3315±14	3380±14
vorbis	2317±6	**2348**±4	2342±19	2186±29	2181±40	2311±38
woff2	3305±3	**3472**±34	3418±36	2080±667	1650±893	3062±530
zlib	615±12	**623**±6	620±5	592±10	595±12	620±3

Table 4: Average edge coverage of ML component over 30 runs.

Neuzz	PreFuzz	**Neuzz++**
292±206	**666**±593	261±166
0±2	0±0	**11**±19
141±182	153±88	534±245
635±225	**865**±255	429±130
28±11	**69**±35	32±13
161±52	**377**±145	345±163
2±3	21±46	**216**±86
61±210	**344**±415	8±23
105±97	101±124	**1136**±206
4±5	6±6	**103**±35
43±27	59±41	**1608**±289
2±3	1±2	**105**±31
52±46	74±32	**950**±215
1846±290	**2248**±284	148±67
6±7	6±6	**91**±70
0±2	13±9	**1396**±160
186±242	145±204	**197**±82
25±93	24±75	**660**±209
0±0	0±0	**94**±129
122±77	**150**±82	77±13
186±21	**186**±42	16±11
5±16	14±49	**39**±21
9±9	10±8	**24**±14

5.4 Code Coverage from Machine Learning

After presenting total coverage for 24-hour runs in Table 3, we now measure how much of the total coverage can be attributed to the machine learning component for each NPS fuzzer. On one hand, the goal is to discount the coverage produced strictly by AFL in the first hour of Neuzz and PreFuzz runs (recall that they use AFL for data collection, see Figure 2) and only measure the NPS fuzzers' contribution. On the other hand, we wish to do the same for Neuzz++, and separate its contribution from that of the base fuzzer AFL++. As Neuzz++ is a custom mutator for AFL++, its seeds usually alternate with regular AFL++ seeds. To this end, we measure edge coverage by corpus replaying, this time only taking into account the seeds obtained by Neuzz, PreFuzz and Neuzz++, respectively. For Neuzz and PreFuzz, this is equivalent to excluding the first hour of coverage, as done by the original authors. In practice, this will include ML-based mutations, but also other hard-coded mutations that the methods apply, such as havoc in the case of PreFuzz. Table 4 summarizes the comparison of edge coverage obtained by the ML components of Neuzz, PreFuzz, and Neuzz++. Program names are aligned with Table 3.

Neuzz++ obtains the highest coverage in 14 over 23 targets, with values at least one order of magnitude higher than Neuzz and PreFuzz. Nevertheless, even on targets where Neuzz++ does not obtain the highest ML coverage (e.g., *freetype2, harfbuzz*), the overall Neuzz++ edge coverage (Table 3) is higher than that of Neuzz and PreFuzz, with the latter two obtaining lower coverage than their base fuzzer AFL. The added value of Neuzz and PreFuzz is low in nine, respectively five targets, with coverage close to zero. In these cases, Neuzz and PreFuzz do not achieve (almost) any coverage past the first hour of fuzzing with AFL (see also Figure 4). This reinforces our previous conclusion that the time spent using Neuzz and PreFuzz might be better spent applying the AFL or AFL++ mutation strategy. Moreover, the Neuzz++ results suggest that it might benefit from the alternation between ML-guided mutations and standard AFL++ ones. We explore this last point with additional analyses in Section 5.5.

> *For most programs, the time budget spent on Neuzz or PreFuzz is better spent on standard gray-box fuzzing.*

5.5 Quality of Machine Learning Test Cases

We now aim to assess the quality of the test cases found by the machine learning component of Neuzz++. We do so with two analyses: we investigate (i) the inclusion of ML-generated inputs in the AFL++ power schedule for further mutation, and (ii) the rarity of code edges found through machine learning-based mutations.

First, Table 5 presents statistics regarding ML-produced test cases for each target averaged over all trials. The column "%ML seeds" shows the overall percentage of inputs produced through ML mutations. Out of these, "%MLcov+" discover new coverage (relative percentage). Finally, "%derived" is the total percentage of the corpus produced by direct mutations of ML-based inputs. We find that the ratio of machine learning inputs varies significantly across targets, representing up to a third of the corpus. ML test cases seem to be most impactful for finding new coverage on programs where they represent a low percentage of the corpus. On average, each ML test case is mutated at least once successfully, generating new test cases that are kept by Neuzz++ in the corpus.

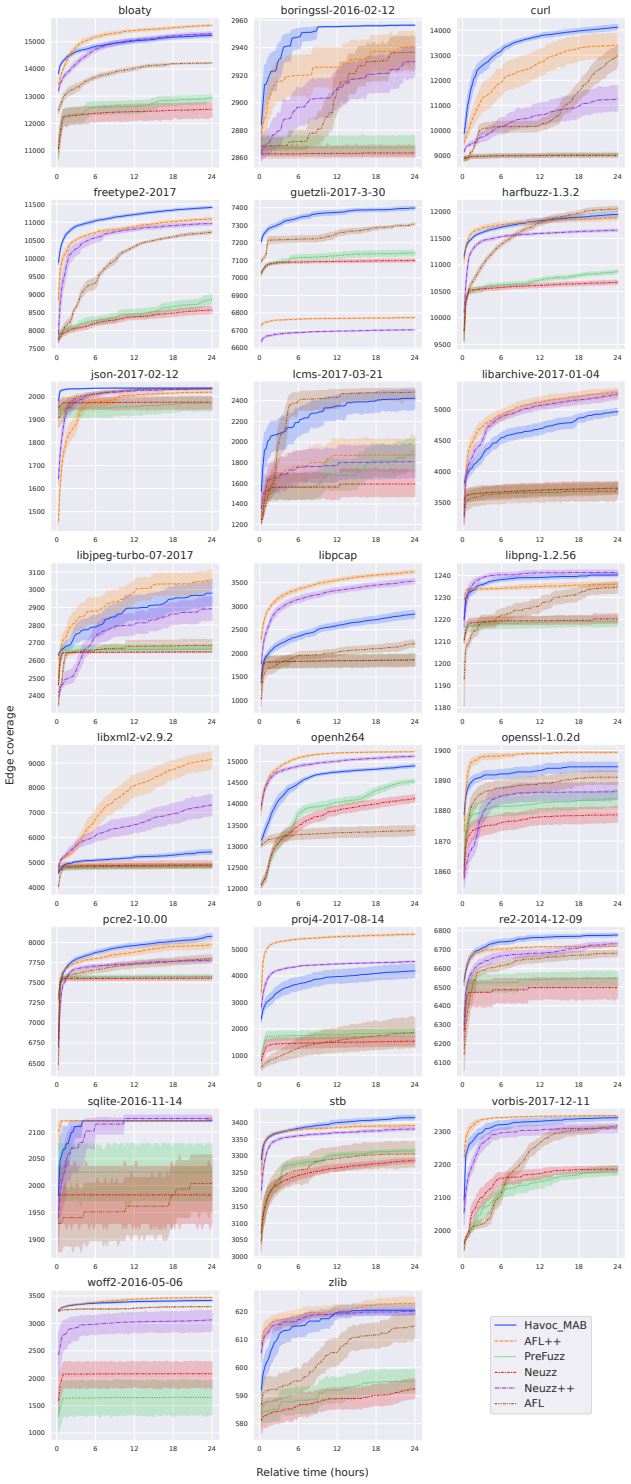

Figure 4: Average edge coverage over time with 95% confidence interval.

Table 5: Statistics for ML-generated test cases of Neuzz++. "%ML seeds" and "%derived" are computed over the total size of the corpus. "%MLcov+" is relative to "%ML seeds".

Target	%ML seeds	%MLcov+	%derived
bloaty	4.72%	28.3%	8.78%
boringssl	27.7%	3.3%	27.5%
curl	18.6%	26.1%	33.1%
freetype2	2.2%	31.9%	3.8%
guetzli	9.9%	9.8%	13.8%
harfbuzz	6.6%	30.2%	15.2%
json	13.7%	37.3%	25.4%
lcms	1.6%	57.1%	1.1%
libarchive	18.3%	30.2%	34.9%
libjpeg	11.8%	10.7%	15.9%
libpcap	13.8%	40.0%	20.8%
libpng	19.6%	13.8%	41.0%
libxml2	15.1%	23.9%	30.1%
openh264	10.2%	8.0%	8.5%
openssl	30.6%	5.6%	28.7%
pcre2	18.3%	17.4%	29.88%
proj4	5.5%	49.7%	7.4%
re2	23.3%	22.8%	34.7%
sqlite	8.1%	20.2%	6.2%
stb	14.8%	15.3%	19.9%
vorbis	6.0%	8.3%	7.9%
woff2	3.8%	19.8%	5.2%
zlib	16.9%	20.7%	18.1%

The second analysis studies whether NPS fuzzers explore code areas that are harder to reach by standard fuzzers. In that case, neural program smoothing fuzzers could be used in an ensemble of diverse fuzzers, opening the path for all fuzzers to rare parts of the code [13]. To measure the rarity of edges reached by Neuzz++, we compare the edge IDs that Neuzz++ and AFL++ reach on each program, all trials joint. The edge IDs are obtained by replaying all the test cases with `afl-showmap`.

We summarize the results in Table 6 as follows: Neuzz++ (denoted N+) reveals less than 0.5% additional edges that AFL++ (denoted A+) in 16 out of 23 targets. Neuzz++ does not find any such exclusive edges for eight programs; it is most successful on *lcms*, with 8.2% exclusive edges. On the other hand, AFL++ finds up to 16.4% exclusive edges, lacking exclusive edges on only two programs (*json* and *sqlite*). We can therefore conclude that NPS-guided fuzzers explore essentially the same code areas as traditional fuzzers.

NPS fuzzers find less rare edges than gray-box fuzzers.

5.6 NPS-based Fuzzing without GPUs

Due to their increased performance for linear algebra and data throughput, GPUs are the *de facto* standard for machine learning. All NPS methods studied in this paper leverage GPU access to train machine learning models and compute gradients for test case mutations. In practice, this means that they use more computational resources than state-of-the-art gray-box fuzzers, and that practitioners are required to invest in additional hardware. In this section, we wish to assess the performance of NPS methods in the absence of GPUs. Model training with only CPU access should be slower,

Table 6: Reached edges for AFL++ (A+) and Neuzz++ (N+).

Target	A+	N+	A+ ∪ N+	A+ \ N+	N+ \ A+
bloaty	5131	4926	5139	213	8
boringssl	1210	1208	1210	2	0
curl	6862	6656	6899	243	37
freetype2	6555	6292	6575	283	
guetzli	2645	2624	2655	31	10
harfbuzz	5438	5081	5440	359	2
json	2036	2036	2036	0	0
lcms	935	1010	1019	9	84
libarchive	3622	3223	3649	426	27
libjpeg-turbo	1565	1539	1565	26	0
libpcap	2869	2628	2901	273	32
libpng	621	616	621	5	0
libxml2	5401	4856	5410	554	9
openh264	5849	5840	5851	11	2
openssl	812	811	812	1	0
pcre2	5548	5355	5577	222	29
proj4	2802	2343	2803	460	1
re2	2391	2426	2440	14	49
sqlite	950	950	950	0	0
stb	2012	2014	2021	7	9
vorbis	1142	1100	1142	42	0
woff2	1254	1268	1270	2	16
zlib	337	333	337	4	0

Table 7: Average edge coverage of NPS fuzzers with and without GPU access (10 runs).

	Neuzz		PreFuzz		**Neuzz++**	
Target	CPU	GPU	CPU	GPU	CPU	GPU
harfbuzz	10607	**10677**	10675	**10897**	11631	**11664**
libjpeg	2618	**2647**	2635	**2688**	**2998**	2892
sqlite	**2017**	1993	2049	**2089**	2121	**2127**
woff2	**1919**	1816	**1702**	954	3288	**3389**

but it should not impact the performance of the trained model. As such, any loss in fuzzing performance comes from spending more time training and less fuzzing. For this small experiment, we select four targets that operate on a varied range of input formats for diversity. We perform ten trials of all NPS fuzzers with and without GPU access (Table 7).

Nine of twelve experiments obtain more code coverage when training the model on GPU, which is to be expected. The exception is PreFuzz on woff2, which is however aligned with this fuzzer's tendency of sometimes becoming stuck on this program (Table 3). Overall, the fuzzing performance on GPU is marginally better, as training times for NPS models are relatively short. The gap between CPU and GPU seems tighter for Neuzz++, which we attribute to an already optimized and short training procedure, which cannot be much further improved by GPUs.

Using GPUs usually results in better coverage for NPS fuzzers.

5.7 Impact of Test Case Transmission Method

In Issue E2, we underlined that NPS-guided fuzzers use files to transfer test cases to the target program. We now show that test case transmission has a major impact on fuzzing performance for the methods in [38, 39, 45, 46]. We note that AFL++ does not reach the performance of its predecessor AFL by a margin in the $\text{Havoc}_{\text{MAB}}$ work [45]. This is inconsistent with several other large benchmarks [3, 33], where AFL++ ranks among the top fuzzers. While not using the persistence mode slows down all fuzzers, we expect state-of-the-art gray-box fuzzers to be affected the most, i.e., they would lose their competitive advantage of speed. This experiment uses the same targets and setup as the previous section.

Table 8: Relative degradation of edge coverage not using persistent mode (10 runs).

Target	AFL	AFL++	$\text{Havoc}_{\text{MAB}}$	Neuzz	PreFuzz	**Neuzz++**
harfbuzz	-8.9%	-60.2%	-2.9%	-3.4%	-2.6%	-2.9%
libjpeg	-2.8%	-55.7%	-5.9%	-8.9%	-7.7%	-14.6%
sqlite	0.1%	-57.4%	-0.3%	-4.9%	-7.9%	-1.7%
woff2	-34.5%	-84.3%	-40.4%	-43.8%	-46.8%	-0.6%

Table 8 presents the performance difference when the persistence mode is not used. This setup reproduces both the protocol and results from $\text{Havoc}_{\text{MAB}}$ [45], the only paper that compares NPS-guided fuzzers against AFL++. As expected, coverage decreases when passing inputs through files and restarting the program for each test case. Most interestingly, AFL++ shows the largest slowdown with a consistent coverage loss over 50%, while the AFL-based fuzzers mainly show single-digit percentage degradation. Consequently, not using the recommended persistence mode can distort the ranking of fuzzers in a benchmark. In our opinion, this setting does not yield a fair or practically relevant comparison. Worth mentioning here is that Neuzz++ can compensate the performance loss of its base fuzzer AFL++, obtaining more coverage in absolute values. As conjectured, results indicate that NPS-guided fuzzers suffer less under slow operation than other fuzzers. Despite that, we are still not able to reproduce the performance of Neuzz and PreFuzz against AFL reported in the original papers.

AFL++ is most slowed down when not using the persistent mode.

5.8 Bugs Found

The main goal of fuzzing is to find as many unique bugs as possible. The default coverage-based crash identification mechanism of AFL and AFL++ tends to overcount unique bugs [27]. To improve this behavior, we apply a more precise stack trace-based deduplication algorithm. We therefore execute each reported crashing input on the target within GNU debugger (GDB) and retrieve all stack frame addresses when the error occurs. This list of addresses then serves as a unique identifier of the triggered bug. Note that deduplication based on stack traces is ineffective when stack overflow errors occur, because the stack frames are then corrupted.

Table 9 contains the number of crashes with unique stack trace signatures across all trials for each target that reported any crashes. Neuzz and PreFuzz find the lowest number of crashing inputs (none for most targets), followed by AFL, their base fuzzer; $\text{Havoc}_{\text{MAB}}$ significantly improves over AFL. AFL++ is most successful in revealing crashes, with most bugs found and all targets covered. In summary, all NPS-based fuzzers find fewer crashing inputs than the fuzzer they are based upon.

Table 9: Bugs found after stack trace deduplication.

Target	AFL	AFL++	$\text{Havoc}_{\text{MAB}}$	Neuzz	PreFuzz	**Neuzz++**
bloaty	1	1	2	0	0	1
guetzli	8	264	5	0	0	170
harfbuzz	0	355	1	0	0	12
json	20	11	22	18	16	10
lcms	0	16	0	0	0	8
libarchive	0	1	0	0	0	0
libxml2	0	648	1	0	0	289
openssl	138	1324	409	37	40	721
pcre2	87	4174	262	40	35	1371
re2	0	172	1	0	0	2
vorbis	0	2	1	0	0	0
woff2	20	361	671	1	1	172

NPS-guided fuzzers find fewer bugs than standard fuzzers.

6 BENCHMARKING ML-BASED FUZZERS

Fuzzer evaluation is an open research topic abundently studied in recent works [3, 6, 27, 33]. A common guideline is that each fuzzer must be tested on multiple programs, using multiple repetitions to account for randomness. The recommended number of repetitions revolves around 10–20 trials. Besides the average performance, indicators of variability (i.e., confidence intervals, statistical tests) are necessary to assess the significance of the results. The main goal of fuzzers is to find bugs, which suggests that unique bugs found in fixed time should be the evaluation metric. However, since bugs are rather rare, the performance of fuzzers is often measured in code coverage over time. This may be justified by observations that more code coverage correlates with more bugs found [6]. To complement these principles, we propose the following practices when evaluating novel machine learning-based fuzzing methods:

(1) **Analyze each new component in the fuzzing loop.** Both performance evaluations and ablation studies of ML models are critical. Metrics specific to the task solved should be used (e.g., accuracy, or precision and recall for classification, mean absolute error or mean squared error for regression, etc.). These complement the view on the overall system performance, i.e., coverage or bugs found in the case of fuzzing. ML evaluation should employ a validation set distinct from the training data to avoid an overly optimistic estimates [24].

(2) **Use state-of-the-art fuzzers and configurations as baselines.** Lacking strong baselines prevents one from claiming novel state-of-the-art accomplishments in terms of code coverage and bugs found. All fuzzers in an experiment should be configured for performance (e.g., appropriate compiler, compilation options, harness, input feeding mode). We also recommend introducing new scientific or technical contributions based on recent fuzzers and evaluation platforms, as opposed to their older counterparts.

(3) **Use comparable metrics for fuzzing performance.** As not all fuzzers measure the same type of coverage, we encourage the use of one common evaluation metric between multiple fuzzers. In practice, this is easiest done by replaying the corpus at the end of a fuzzing trial, as implemented by FuzzBench [3, 33] and MLFuzz.

(4) **Repeat trials often enough to account for variance.** We propose to use 30 trials for fuzzing evaluation, resulting in tight confidence intervals. This sample size is commonly used in statistics and deemed sufficient for the central limit theorem [9] to hold. As shown in Figure 4, ML-based fuzzers can have higher coverage variability than gray-box fuzzers, thus requiring more trials for stable baselining.

(5) **Ensure reproducible results by fixing and serializing parameters.** While it is difficult to control all sources of randomness when training ML models on GPUs, it remains a good practice in both machine learning and software testing to control possible sources of randomness by seeding random number generators and reusing the same seeds. Experimental configurations and, in the case of ML, hyperparameters should be documented for reproducibility.

(6) **Ensure usability of proposed fuzzers.** It should be possible to run a newly proposed fuzzer on programs outside the original publication study. Providing a containerized environment can sustainably decrease setup efforts. We also support integration of new fuzzers with existing benchmarking platforms, such as FuzzBench and now MLFuzz.

7 CONCLUSION AND CONSEQUENCES

Neural program smoothing for fuzzing neither reaches its advertised performance, nor does it surpass older fuzzing techniques that are still state-of-the-art. In our in-depth analysis of NPS fuzzers, we analyzed conceptual limitations of previously published approaches, as well as implementation and evaluation issues. Our comprehensive benchmark showed that NPS-guided fuzzers were by far unable to reach their stated performance. Addressing the implementation issues did not suffice to outperform state-of-the-art gray-box fuzzers. The reason for the limited fuzzing performance lies in the difficulty of the machine learning task, which yields trivial models on the data available during fuzzing.

To guide future fuzzing research and practical validation, we developed improved experimental guidelines targeting fuzzing with machine learning. Our MLFuzz framework for ML-based fuzzers includes patched and containerized versions of the investigated fuzzers to help with additional benchmarking. We encourage researchers to perform ablation studies and provide deeper insights into the components they introduce in fuzzing.

While we highlight fundamental limitations of neural program smoothing, whether and how much this technique can enhance fuzzing remains an open topic for future research. We hope that this work contributes to fair and comprehensive evaluations of future fuzzers, be they ML-based or not.

8 DATA AVAILABILITY

The open-source implementation of Neuzz++ and MLFuzz, the evaluation setup, and raw results are available at

https://github.com/boschresearch/mlfuzz
https://github.com/boschresearch/neuzzplusplus.

ACKNOWLEDGEMENTS

Thanks are due to Josselin Feist and the anonymous reviewers for valuable discussions that have led to paper improvements. This work was supported by the German Federal Ministry of Education and Research (BMBF, project CPSec – 16KIS1565 and 16KIS1564K).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org. Software available from tensorflow.org.

[2] Andrea Arcuri. 2010. It Does Matter How You Normalise the Branch Distance in Search Based Software Testing. In *International Conference on Software Testing, Verification and Validation*. 205–214. https://doi.org/10.1109/ICST.2010.17

[3] Dario Asprone, Jonathan Metzman, Abhishek Arya, Giovani Guizzo, and Federica Sarro. 2022. Comparing Fuzzers on a Level Playing Field with FuzzBench. In *IEEE International Conference on Software Testing, Verification and Validation - Industry (ICST Industry)*. https://doi.org/10.1109/ICST53961.2022.00039

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2 (2002), 235–256. https://link.springer.com/article/10.1023/A:1013689704352

[5] Jana Aydinbas. 2022. AFLplusplus Persistence Mode README. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md. Accessed: 2023-05-10.

[6] Marcel Böhme, Laszlo Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *International Conference on Software Engineering (ICSE)*. http://seclab.cs.sunysb.edu/lszekeres/Papers/ICSE22.pdf

[7] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. In *IEEE Security and Privacy Workshops (SPW)*. 116–122. https://arxiv.org/abs/1801.04589

[8] Rich Caruana. 1997. Multitask Learning. *Machine Learning* 28 (1997), 41–75. https://doi.org/10.1023/A:1007379606734

[9] Horng-Jinh Chang, Kuo-Chung Huang, and Chao-Hsien Wu. 2006. Determination of sample size in using central limit theorem for Weibull distribution. *International journal of information and management sciences* 17 (2006), 31–46.

[10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity Analysis of Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 57–70. https://doi.org/10.1145/1706299.1706308

[11] Swarat Chaudhuri and Armando Solar-Lezama. 2011. Smoothing a Program Soundly and Robustly. In *Computer Aided Verification (CAV)*. https://www.cs.utexas.edu/~swarat/pubs/cav11.pdf

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (SP)*. https://web.cs.ucdavis.edu/~hchen/paper/chen2018angora.pdf

[13] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium (USENIX Security)*. https://www.usenix.org/system/files/sec19-chen-yuanliang.pdf

[14] Albert Danial. 2021. *cloc: v1.92*. https://doi.org/10.5281/zenodo.5760077

[15] Poetry developers. 2018. Python Poetry. https://python-poetry.org. Accessed: 2022-10-20.

[16] William Drozd and Michael D. Wagner. 2018. FuzzerGym: A Competitive Framework for Fuzzing and Learning. *CoRR* (2018). arXiv:1807.07490 http://arxiv.org/abs/1807.07490

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. [n. d.]. AFL++ best practices. https://aflplus.plus/docs/fuzzing_in_depth. Accessed: 2022-10-20.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[19] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE.2017.8115618

[20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems (NIPS)*, Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[21] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1412.6572

[22] Google. 2017. Fuzzer test suite. https://github.com/google/fuzzer-test-suite. Accessed: 2022-10-20.

[23] Google. 2022. OSS-Fuzz. https://google.github.io/oss-fuzz/. Accessed: 2022-10-20.

[24] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference and prediction* (2 ed.). Springer. http://www-stat.stanford.edu/~tibs/ElemStatLearn/

[25] Zhicheng Hu, Jianqi Shi, YanHong Huang, Jiawen Xiong, and Xiangxing Bu. 2018. GANFuzz: A GAN-Based Industrial Network Protocol Fuzzing Framework. In *ACM International Conference on Computing Frontiers*. 138–145. https://doi.org/10.1145/3203217.3203241

[26] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1412.6980

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138.

[28] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 1–13.

[29] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing, In AAAI Conference on Artificial Intelligence (AAAI). *AAAI Conference on Artificial Intelligence* 33, 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[30] LLVM. 2022. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2022-10-20.

[31] Robert C. Martin and James O. Coplien. 2009. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall. https://archive.org/details/cleancodehandboo00mart_843

[32] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[33] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[35] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR* (2017). https://arxiv.org/abs/1711.04596

[36] Gary J. Saavedra, Kathryn N. Rodhouse, Daniel M. Dunlavy, and Philip W. Kegelmeyer. 2019. A Review of Machine Learning Applications in Fuzzing. *CoRR* (2019). https://arxiv.org/abs/1906.11133

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Conference on Annual Technical Conference (USENIX ATC)*. https://dl.acm.org/doi/10.5555/2342821.2342849

[38] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *ACM Joint European Software Engineering Conference and Symposiumon the Foundations of Software Engineering (ESEC/FSE)*. https://dl.acm.org/doi/pdf/10.1145/3368089.3409723

[39] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: efficient fuzzing with neural program smoothing. In *IEEE Symposium on Security and Privacy (S&P)*. https://arxiv.org/abs/1807.05620

[40] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[41] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[43] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, 1–15.

[44] Yan Wanga, Peng Jiaa, Luping Liub, and Jiayong Liu. 2019. A systematic review of fuzzing based on machine learning techniques. *CoRR* (2019). https://arxiv.org/abs/1908.01262

[45] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, New York, NY, USA, 1634–1645. https://doi.org/10.1145/3510003.3510174

[46] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and

improving neural program-smoothing-based fuzzing. In *International Conference on Software Engineering (ICSE)*. http://zhangyuqun.com/publications/icse2022a.pdf

[47] Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 789–800. https://doi.org/10.1145/3377811.3380388

[48] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium (USENIX Security)*. USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[49] Michal Zalewski. 2017. American fuzzy lop. https://github.com/google/AFL.

Received 2023-02-02; accepted 2023-07-27