Evaluating and Improving Neural Program-Smoothing-based Fuzzing

Anonymous Author(s)

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 such issue, two neural program-smoothing-based fuzzers, *Neuzz* and *MTFuzz*, have been recently proposed by approximating program branching behaviors via a neural network model which inputs byte sequences of a seed and outputs 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. Although they have been shown to be effective in their 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 their performance. To further investigate neural program-smoothing-based fuzzing, we first construct a large-scale benchmark with a total of 28 influential open-source projects. Then, we extensively evaluate *Neuzz* and *MTFuzz* on such benchmark where the results suggest that they can incur quite inconsistent edge coverage performance. Moreover, neither neural network models or mutation strategies can be consistently effective, and the power of their gradient guidance mechanisms has been compromised so far. Inspired by such findings, we propose an improved technique, namely *RESuzz*, upon neural program-smoothing-based fuzzers by enhancing their adopted gradient guidance mechanisms along with appending the AFL havoc mechanism. Our evaluation results indicate that it can significantly enhance the edge coverage performance of *Neuzz* and *MTFuzz*. Furthermore, we also reveal multiple practical guidelines to advance future research.

1 INTRODUCTION

Fuzzing [50] nowadays has become widely adopted to detect software vulnerabilities by automatically generating invalid, unexpected, or random data as input for running programs to monitor possible exceptions, e.g., crashes, failing assertions, and memory leaks. 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 [54] [56][17][19][48]. Specifically, given initial seed inputs, such fuzzers iteratively mutate the existing seeds and filter the promising ones for further mutations such that they can approach the optimal solutions to satisfy the fitness functions (which are usually designed to maximize code coverage) of their adopted evolutionary algorithms.

Evolutionary fuzzers have been argued that they fail to “leverage the structure (i.e., gradients or higher-order derivatives) of the underlying optimization problem” [47]. To address such issue, neural program-smoothing-based techniques, i.e., *Neuzz* [47] and *MTFuzz* [46], have been recently proposed in S&P’19 and FSE’20 respectively to exploit the usage of gradients for fuzzing via neural network models. Specifically, they first adopt a neural network which inputs the byte sequence of a seed (e.g., a class file) and outputs a vector representing its associated program branching behaviors. Next, they compute the gradients as dividing the collected output vectors by the bytes of the given seed. Accordingly, they sort the resulting gradients and develop strategies to invest more computing resource to mutate the bytes corresponding to larger gradients. As a result, all the generated seeds after mutations are used as test cases for fuzzing. More specifically, *MTFuzz* leverages the power of multi-task learning and adopts a dynamic analysis module to augment the mutation strategy to enhance the code coverage performance of *Neuzz*. In their original papers, *Neuzz* outperforms 10 existing coverage-guided fuzzers on 10 real-world projects by achieving at least 3X more edge coverage over 24 hour runs and further explores 31 previously-unknown bugs. Compared to *Neuzz* and four other state-of-the-art fuzzers, *MTFuzz* achieves 2X to 3X edge coverage upon all the benchmark projects and exposes 11 previously-unknown bugs which cannot be found by either *Neuzz* or the other fuzzers.

Despite the effectiveness shown in their original papers, the evaluation on *Neuzz* and *MTFuzz* can be susceptible to bias due to their limited benchmark with 10 projects only. Moreover, *Neuzz* and *MTFuzz* adopt quite an inconsistent edge coverage measure with many existing fuzzers [56][4][32][36][10] to potentially compromise the fairness of their performance comparison with other fuzzers. Furthermore, the investigation on the factors which can impact their edge coverage performance is rather limited, i.e., they only simply present the overall testing effectiveness of the techniques while the contributions made by their individual components, e.g., the model structure, the gradient guidance mechanism, and the mutation strategy, were not investigated at all.

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 the majority of the projects adopted in the original *Neuzz* and *MTFuzz* papers and injecting 19 additional open-source projects which are frequently adopted in recent fuzzing research works. 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 11% and 9% on average in terms of edge coverage respectively, *MTFuzz* does not always outperform *Neuzz* and both their edge coverage performances are strongly program-dependent. We also find neither their mutation strategies or neural network models can be consistently effective.

Meanwhile, although their adopted gradient guidance mechanisms are generally effective, their strengths have not been fully leveraged.

Inspired by the findings of our study, we propose an improved technique, namely *RESuzz*, upon neural program-smoothing-based fuzzing. In particular, we develop a *resource-efficient edge selection mechanism* to guide the exploration towards the unexplored edges rather than the existing edges for enhancing the their adopted gradient guidance mechanisms. Moreover, we also append a random fuzzing strategy—the AFL havoc mechanism to *Neuzz* and *MTFuzz* for their iterative executions to further facilitate edge exploration. Our evaluation results suggest that *RESuzz* can significantly outperform *Neuzz* and *MTFuzz*, i.e., 36.2% more than *Neuzz* and 38.2% more than *MTFuzz* averagely in terms of edge coverage.

To conclude, this paper makes the following contributions:

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

2 BACKGROUND

2.1 Coverage-guided Fuzzers

Coverage-guided fuzzers nowadays widely adopt evolutionary algorithms [54] 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.

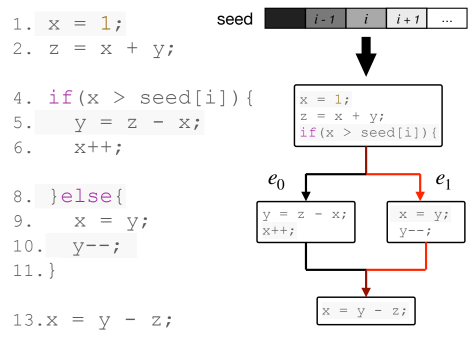
2.1.2 Integrating fuzzing with EA. The fitness functions of coverage-guided fuzzers are often defined as increased code coverage. Specifically, such fuzzers usually adopt edge coverage (where an *edge* refers to a basic-block-wise transition, e.g., a conditional jump in programs) to represent code coverage and allow only the seeds which increase edge coverage for further mutations. For instance, American Fuzzy Lop (AFL) [56], a typical coverage-guided fuzzer developed by Google, 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. As most of the coverage-guided fuzzers [10][32][36][4], when running a seed can increase edge coverage, AFL identifies such seed as an "interesting" seed and retains it for further mutations. Note that AFL enables two stages for mutations—the deterministic stage ($AFL_{Deterministic}$) and the havoc stage (AFL_{Havoc}). In particular, $AFL_{Deterministic}$ 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 $AFL_{Deterministic}$, all the collected "interesting" seeds are used to launch AFL_{Havoc} where random mutations, i.e., randomly chosen mutators, 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 [47]. 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* [47] and *MTFuzz* [46] are proposed to smooth programs via neural network models and guide mutations by yielding the power of their gradients for fuzzing. 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, i.e., a byte sequence, and its solution is a vector representing its associated branching behaviors. For instance, a solution vector $[1, 0, 1, ...]$ 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 facilitate the power of the neural network model because gradient can be only adopted on such function type.

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 satisfy its access condition. Otherwise, one of its "sibling" edges (i.e., edges under one sharing prefix edge) can be accessed. For instance, in Figure 1, edge e_0 can be accessed when the value of $seed[i]$ satisfies the access condition for e_0, i.e., $seed[i] < 1$. Hence, mutating such $seed[i]$ can lead to exploring a new branching behavior, i.e., accessing e_0's "sibling" edge e_1 instead of e_0.

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., $seed[i]$ 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.


```
x = 1;
z = x + y;

if(x > seed[i]){
   y = z - x;
   x++;

}else{
   x = y;
   y--;
}

x = y - z;
```

Figure 1: An example of neural program-smoothing-based

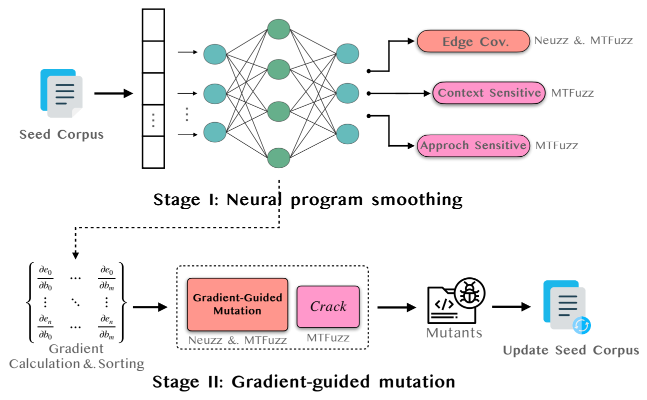

Seed Corpus
Edge Cov.
Neuzz & MTFuzz
Context Sensitive
MTFuzz
Approch Sensitive
MTFuzz
Stage I: Neural program smoothing
$\frac{\partial e_0}{\partial b_0}$ ... $\frac{\partial e_0}{\partial b_m}$
$\frac{\partial e_n}{\partial b_0}$ ... $\frac{\partial e_n}{\partial b_m}$
Gradient Calculation & Sorting
Gradient-Guided Mutation
Neuzz & MTFuzz
Crack
MTFuzz
Mutants
Update Seed Corpus
Stage II: Gradient-guided mutation

Figure 2: Framework of *Neuzz* and *MTFuzz*

2.2.1 Neural Program Smoothing. *Neuzz* and *MTFuzz* enable iterative training-and-mutation process. Under each iteration, they train their neural network models using the real-time collected "interesting" seed inputs (out of the "Seed Corpus" in Figure 2). Note that Figure 2 also reveals that *Neuzz* and *MTFuzz* adopt different neural network models which would be 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 by dividing the vectors of all the selected edges by all the bytes of the selected seed. 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 enable 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 included in the front segments. For any segment *seg*, all its bytes are simultaneously mutated, i.e., *(byte + 1) modulo 255*, for 255 times. As a result, *Neuzz* and *MTFuzz* can explore more mutation space of the front segments than the back ones, i.e., invest more computing resource in mutating the "promising" bytes, 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 form *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* enables 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 THE 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 [46, 47], such results can be possibly biased because they did not present any criteria to adopt "10 popular real-world programs" for evaluation. To be specific, it is unclear whether such 10 programs are sufficient or representative.

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 `Zlib` 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 the 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 [44]). 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; (2) the 28 collected benchmarks can cover 12 different file formats of seed inputs, e.g., ELF, XML, and JPEG; and (3) the LoC of each program which ranges from 1,886 to over 120K can represent a wide range of program size.

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. Moreover, we run our experiments for 5 times for each fuzzer and present the average results with close performance

Table 1: Statistics of the studied benchmarks

Benchmark	Class	LOC	Package
bison	LEX & YACC	18,701	3.7
xmlwf	XML	6,871	expat-2.2.9
mupdf	PDF	123,562	1.12.0
pngimage	PNG	11,373	libpng-1.6.36
pngfix	PNG	12,173	libpng-1.6.36
pngtest	PNG	11,323	libpng-1.6.36
tcpdump	PCAP	46,892	4.99.0
nasm	ASM	18,941	nasm-2.15.05
tiff2pdf	TIFF	17,272	libtiff-4.2.0
tiff2ps	TIFF	16,177	libtiff-4.2.0
tiffdump	TIFF	15,113	libtiff-4.2.0
tiffinfo	TIFF	15,014	libtiff-4.2.0
libxml	XML	73,239	2.9.7
listaction	SWF	6,278	libming-0.4.8
listaction_d	SWF	6,272	libming-0.4.8
libsass	SCSS	14,638	libsass-3.6.5
jhead	JPEG	1,886	3.04
readelf	ELF	72,111	Binutils 2.30
nm	ELF	55,212	Binutils 2.30
strip	ELF	65,683	Binutils 2.30
size	ELF	54,463	Binutils 2.30
objdump	ELF	74,710	Binutils 2.30
libjpeg	JPEG	8,856	9c
harfbuzz	TTF	9,853	1.7.6
base64	FILE	40,332	LAVA-M
md5sum	FILE	40,350	LAVA-M
uniq	FILE	40,286	LAVA-M
who	FILE	45,257	LAVA-M

under different runs. Note that we instrument all the benchmark projects with `afl-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 [36][4][3][55][33] and frequently upgraded for advancing its performance; and (2) *Neuzz* adopts multiple concepts originated from AFL for its implementation [45].

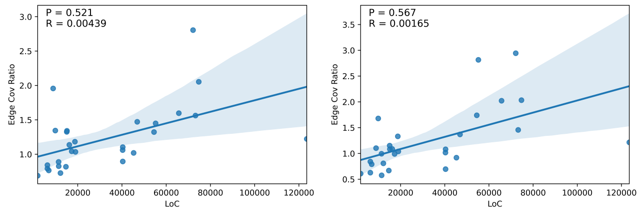
3.3 Research Questions

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.
- **RQ2:** How do the key components of *Neuzz* and *MTFuzz* impact on edge exploration? For this RQ, we attempt to investigate how exactly their adopted gradient guidance mechanisms, neural network models, and mutation strategies can impact on 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 particular, while the original *Neuzz* and *MTFuzz* papers measure edges by counting the byte number of the `trace_bits` structure implemented by AFL, such measures essentially are inconsistent with the results provided by the AFL built-in



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

Figure 3: Edge coverage advantage of the fuzzers over AFL

`afl-showmap` out of their own implementations. In this paper, following many existing coverage-guided fuzzers [56][4][32][36][10], we determine to adopt the number of the edges via `afl-showmap` to measure edge coverage.

Table 2 demonstrates the edge coverage results of our extensive study for *Neuzz* and *MTFuzz*. Overall, we can observe that *Neuzz* can significantly outperform AFL by 11% (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 severely degraded. We then investigate the performance difference among benchmark projects. Interestingly, we can observe that their performance advantage can be rather inconsistent, i.e., ranging from -31.2% to 180.5%. Moreover, *Neuzz* can only outperform AFL upon 10 out of 19 extended projects. Such results can suggest that *Neuzz* cannot always outperform AFL and the performance advantage of *Neuzz* over AFL can be strongly program-dependent.

We can also observe that *Neuzz* can outperform *MTFuzz* by 1.5% (22,395 vs. 22,070 explored edges) in terms of edge coverage on average upon all the adopted benchmarks. While in 11 benchmark projects, *MTFuzz* can outperform *Neuzz* by 19% on average, *Neuzz* can outperform *MTFuzz* by 15% in 17 benchmark projects. Furthermore, even AFL can outperform *MTFuzz* by 35% on average in 11 benchmark projects. Such results can indicate that similar to *Neuzz*, *MTFuzz* cannot incur consistent performance either.

We then attempt to reveal the characteristics of how the edge coverage performance varies among the studied benchmark 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 benchmark 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 benchmark project and the vertical axis denotes the ratio by 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 advantage of edge coverage over AFL upon larger benchmark projects than smaller ones. Such results clearly demonstrate that the program size can significantly impact the edge coverage performance of neural program-smoothing-based fuzzers.

We also evaluate AFL, *Neuzz*, and *MTFuzz* in terms of the originally adopted measure of edge coverage. In particular, *Neuzz* can outperform AFL by 35% (2,219 vs. 1,640 explored edges) and *Neuzz* can outperform *MTFuzz* by 1.8% (2,219 vs. 2,180 explored edges). In

Table 2: Edge coverage results of all the studied approaches

Benchmarks	**AFL**	***Neuzz***	***Neuzz***$_{ReverseGradient}$	***MTFuzz***	***MTFuzz***$_{ReverseGradient}$	***MTFuzz***$_{CrackOff}$	***Neuzz***$_{CNN}$	***Neuzz***$_{LSTM}$	***Neuzz***$_{Bi-LSTM}$
bison	10,374	12,260	12,218	**13,812**	12,799	12,801	12,375	12,592	12,444
xmlwf	**13,729**	10,499	9,192	10,853	10,532	10,752	10,872	11,465	11,469
mupdf	13,665	16,705	**17,664**	16,603	16,348	16,522	16,853	16,921	16,889
pngimage	**4,077**	3,369	2,522	2,347	2,172	2,373	2,946	2,553	3,011
pngfix	**7,134**	5,181	4,564	5,767	5,350	5,737	5,157	5,194	5,247
pngtest	3,185	2,828	2,933	3,166	2,719	3,016	3,074	**3,271**	3,103
tcpdump	12,434	18,293	18,566	17,026	15,097	17,463	18,091	18,910	**19,411**
nasm	33,633	34,788	33,838	34,958	34,451	33,907	**35,375**	34,528	35,009
tiff2pdf	45,183	47,109	42,519	44,765	38,449	44,230	46,934	44,617	**50,347**
tiff2ps	20,862	**23,705**	21,063	22,671	16,700	21,817	23,931	21,322	23,160
tiffdump	2,416	**3,239**	3,117	2,617	2,262	2,509	3,124	2,962	3,052
tiffinfo	11,964	**15,853**	15,742	13,785	10,249	12,394	14,698	13,239	15,569
libxml	20,064	31,340	**32,075**	29,236	29,162	27,902	31,421	31,774	31,731
listaction	**21,340**	17,945	14,969	13,382	12,257	12,356	17,743	17,562	17,073
listaction_d	**31,617**	25,006	18,643	26,629	21,644	23,619	25,869	28,436	23,622
libsass	**198,976**	162,717	158,800	132,972	132,491	132,644	154,793	160,318	163,492
jhead	**2,082**	1,433	1,566	1,268	1,215	1,273	1,327	1,560	1,502
readelf	14,329	40,186	34,994	42,173	35,178	40,005	42,889	**44,389**	32,796
nm	11,154	16,159	13,505	**31,402**	28,027	22,149	18,070	20,724	16,007
strip	20,536	32,791	31,604	**41,520**	35,649	32,072	33,549	33,816	33,348
size	10,730	14,197	12,414	**18,675**	16,525	12,623	14,488	14,254	12,284
objdump	15,492	31,808	28,617	31,507	27,227	30,074	31,176	33,165	**33,486**
libjpeg	8,197	16,037	13,460	9,038	8,446	8,255	16,576	16,859	**17,566**
harfbuzz	26,420	35,502	28,037	44,342	37,821	44,364	45,179	48,959	**50,911**
base64	**1,344**	1,202	987	935	912	819	1,247	1,226	1,243
md5sum	2,871	3,168	3,004	3,101	3,036	3,044	3,097	**3,241**	3,163
uniq	713	**756**	750	725	728	716	755	754	752
who	2,917	2,973	2,919	2,680	2,753	2,720	2,997	**3,031**	3,026
Average	20,265	22,395	20,724	22,070	20,007	20,648	22,665	**23,130**	22,883

general, the edge coverage advantage by our adopted measure can be bounded by the originally adopted measure.

Finding 1: Neuzz and MTFuzz cannot always be effective. Their strengths can significantly vary between programs, i.e., the larger the programs are, the larger edge coverage such neural program-smoothing-based fuzzers can lead to.

We also investigate the efficiency in exploring edges of all the studied fuzzers. To this end, we establish a metric, namely *Edge Discovery Rate* (EDR), to represent the number of the edges per second. Specifically for AFL, we also investigate its deterministic stage $AFL_{Deterministic}$ and the havoc stage AFL_{Havoc}. Note that they are executed independently under their respective execution time. To summarize, we determine to measure the EDR of *Neuzz*, *MTFuzz*, AFL, $AFL_{Deterministic}$ and AFL_{Havoc}.

Figure 4 presents our EDR results of all the studied fuzzers. We can observe that overall, *Neuzz* and *MTFuzz* can outperform AFL by 25% and 28% respectively. Interestingly, AFL_{Havoc} achieves the highest EDR, i.e., 22X larger than $AFL_{Deterministic}$, 14X larger than *Neuzz*, and 14X larger than *MTFuzz* averagely under all the benchmarks. Accordingly, we can derive that AFL_{Havoc} can significantly augment edge exploration, i.e., promptly explore edges based on the limited seed inputs provided by $AFL_{Deterministic}$. Such result can be enlightening that applying the "havoc" mechanism to the fuzzer-provided seed inputs can potentially augment edge exploration.

Finding 2: AFL_{Havoc} can dominate the efficiency of edge exploration, indicating that it can be promising to augment edge exploration by applying the "havoc" mechanism to the seed inputs provided by fuzzers.

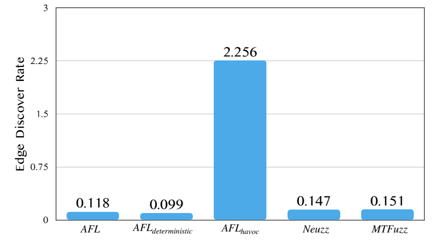


Figure 4: EDR of the studied approaches

3.4.2 RQ2: Effectiveness of the key components.

Gradient guidance. The inconsistencies between our finding and the declared results in the original papers (i.e., finding 1) inspire us to further investigate the performance impact of the adopted mechanisms of *Neuzz* and *MTFuzz*. To this end, we determine to first investigate the effectiveness of their dominating mechanism, i.e., the gradient guidance mechanism. In particular, since such mechanism is proposed to guide the computing resource spent on mutating the "promising" bytes for edge exploration via gradient computation, our purpose is to investigate whether their derived gradients can locate such bytes. More specifically, we propose an intuitive gradient guidance mechanism—instead of allocating more computation resource to mutate the bytes with larger gradients in the original *Neuzz* and *MTFuzz*, we allocate more computation resource to mutate the bytes with smaller gradients. Such mechanism is injected to *Neuzz* and *MTFuzz* to form their variants $Neuzz_{ReverseGradient}$ and $MTFuzz_{ReverseGradient}$. We thus evaluate $Neuzz_{ReverseGradient}$ and $MTFuzz_{ReverseGradient}$ and observe their performance difference from *Neuzz* and *MTFuzz* to investigate the effect of the gradient guidance mechanisms.

We can observe from Table 2 that *Neuzz* can explore 8% (1,671) more edges than $Neuzz_{ReverseGradient}$ and *MTFuzz* can explore 10% (2,063) more edges than $MTFuzz_{ReverseGradient}$ on average. Such consistent results suggest that larger gradients can be a better indicator to promising bytes, i.e., the derived gradients can reflect promising bytes.

Interestingly, $Neuzz_{ReverseGradient}$ can outperform *Neuzz* on 5 out of 28 projects, i.e., `libxml`, `mupdf`, `jhead`, `tcpdump`, and `pngtest`. Meanwhile, $MTFuzz_{ReverseGradient}$ can outperform *MTFuzz* under `uniq` and `who`. Such results can also indicate that the power of the gradient guidance in *Neuzz* and *MTFuzz* has not been completely leveraged.

Finding 3: Although the gradient guidance mechanisms adopted by Neuzz and MTFuzz can be somewhat effective to reflect the promising bytes, they still can be improved.

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 can impact edge exploration. Specifically, since compared to *Neuzz*, *MTFuzz* enables the independent dynamic analysis module *Crack* to augment their mutation strategy, we determine to turn it off and form its variant $MTFuzz_{CrackOff}$, 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) [31] model and two commonly-used Recursive Neural Network (RNN) [18] models, i.e., LSTM [29] and Bi-LSTM [27], and adopt them in the original *Neuzz* to form its variants $Neuzz_{CNN}$, $Neuzz_{LSTM}$, and $Neuzz_{Bi-LSTM}$. 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 [13] (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.

We can observe from Table 2 that overall, all our studied approaches can incur quite similar edge coverage. Specifically, *Neuzz* slightly outperforms $MTFuzz_{CrackOff}$ by 8% (22,395 vs 20,648 explored edges), underperforms $Neuzz_{CNN}$ by 1.2% (22,395 vs. 22,665 explored edges), $Neuzz_{LSTM}$ by 3.3% (22,395 vs. 23,130 explored edges) and $Neuzz_{Bi-LSTM}$ by 2.2% (22,395 vs. 22,883 explored edges). Meanwhile, we can also observe that none of the studied approaches can dominate their performance advantage on top of all the studied benchmarks, i.e., *Neuzz* dominates 7, $MTFuzz_{CrackOff}$ dominates 2, $Neuzz_{CNN}$ dominates 4, $Neuzz_{LSTM}$ dominates 9, and $Neuzz_{Bi-LSTM}$ dominates 6. Therefore, we can derive that upgrading neural network models cannot significantly impact the performance of edge exploration.

Finding 4: The neural network model adopted by MTFuzz does not really work; and altering neural network models can enable limited effectiveness for enhancing fuzzing effectiveness.

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 $MTFuzz_{CrackOff}$. Table 2 demonstrates that overall, *MTFuzz* can outperform $MTFuzz_{CrackOff}$ by 7% (22,070 vs. 20,648 explored edges). However, such advantage can be rather inconsistent, ranging from -2.5% to 48% upon individual benchmark project. On the other hand, applying *Crack* can be potentially cost-inefficient 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 effect of the dynamic analysis module Crack adopted by MTFuzz can be limited.

3.5 Discussion

Our findings throughout answering RQ1 and RQ2 indicate that *Neuzz* and *MTFuzz* cannot always be effective since their mechanisms are somewhat defective. Specifically, while the gradient guidance mechanism can be potentially enhanced, the neural network models and the mutation strategy using dynamic analysis might be essentially ineffective/cost-inefficient under diverse programs. In this section, we attempt to illustrate their rationale.

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* enable neural network models which compute the gradients to reflect the *explored edge—existing seed* relations, 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 sharing 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—existing seed* relations, i.e., how its generalization or prediction capability does not matter under such scenarios. Therefore, it is quite likely that a simplistic model (e.g., feed-forwarded network model adopted by *Neuzz*) can incur similar performance as fine-grained models (e.g., multi-task learning model adopted by *MTFuzz* and the RNN models adopted by the *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 to compute gradient (illustrated in Section 2.2.2). As a result, *Neuzz* and *MTFuzz* can still incur inefficient resource usage, i.e., investing computing resource on exploring existing edges other than new edges. 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 the "promising" bytes of such an edge for exploring new edges.

4 THE TECHNICAL IMPROVEMENT—*RESUZZ*

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 *RESuzz* (**R**esource-efficient **E**dge selection for neural program-**S**moothing-based F**uzz**ing). Note that *RESuzz* also appends AFL_{Havoc} to faclitate edge exploration. Figure 5 presents the workflow of *RESuzz*. *RESuzz* first trains a neural network model by applying all the existing seeds as the training set. Next, *RESuzz* 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. Finally, *RESuzz* executes AFL_{Havoc} to facilitate mutations.

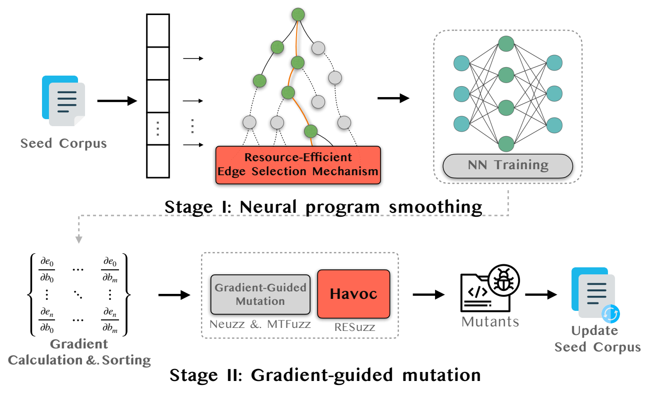

Seed Corpus
Resource-Efficient Edge Selection Mechanism
NN Training
Stage I: Neural program smoothing
Gradient Calculation & Sorting
Gradient-Guided Mutation
Havoc
Neuzz & MTFuzz
RESuzz
Mutants
Update Seed Corpus
Stage II: Gradient-guided mutation

Figure 5: Framework of *RESuzz*

4.1 The Details

4.1.1 Resource-Efficient Edge Selection Mechanism. The purpose of the *resource-efficient edge selection mechanism* is to prevent investing massive computation resource on 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), 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.

We now introduce our *resource-efficient edge selection mechanism* as follows. Algorithm 1 presents the details of the entire process. First, it is quite essential to acquire the runtime information of the

Algorithm 1 Candidate Edge Set Construction

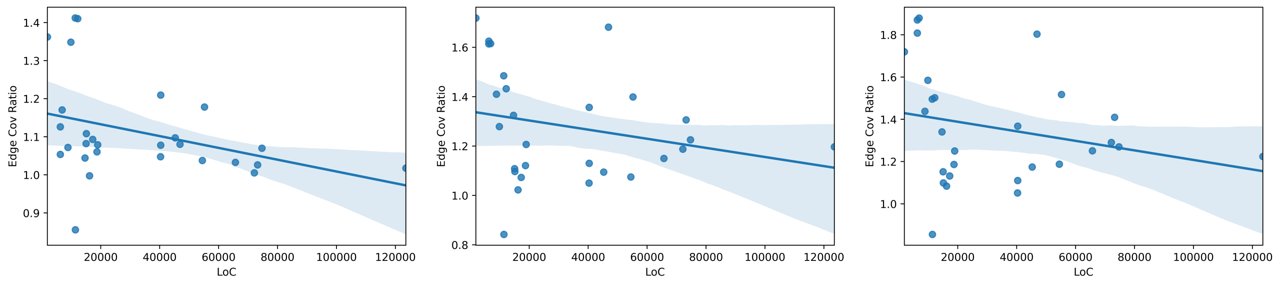
```
    Input : threshold, exploredEdge
    Output:selectedEdges
 1: function CONSTRUCT_CANDIDATE_EDGE_SET
 2:     candidate ← set()
 3:     correspRelation ← getEdgeRelation()
 4:     for edge in exploredEdge do
 5:         explored ← 0
 6:         siblings ← |correspRelation[edge]|
 7:         for 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:     selectedEdges ← randomlySelectFromSet(candidate)
13:     return selectedEdges
```

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 result in the exploration of its "sibling" edge e_1. While *Neuzz* and *MTFuzz* are designed to facilitate 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) to compromise the effectiveness of the gradient guidance mechanism. 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.

4.1.2 Appending AFL_{Havoc}. For *Neuzz* and *MTFuzz*, we append AFL_{Havoc} to mutate each "interesting" seed (as defined in AFL) identified from the gradient guidance. If the generated seeds after mutations are also "interesting", they are retained for further gradient computation and AFL_{Havoc}. Such process is iterated until hitting the time budget.

4.2 Performance Evaluation

We attempt to evaluate the performance of *RESuzz* and its technical components respectively. To evaluate the usage of *resource-efficient edge selection mechanism* and AFL_{Havoc}, we form two *Neuzz* variants, i.e., $Neuzz_{EdgeSelection}$ which injects *resource-efficient edge selection mechanism* to *Neuzz* and $Neuzz_{havoc}$ which appends AFL_{Havoc} to *Neuzz*. Note that we retain *Neuzz*, *MTFuzz*, and AFL as our baselines for performance comparison. The experimental setups in this



(a) ***Neuzz*$_{EdgeSelection}$*/Neuzz*** **(b)** ***Neuzz*$_{havoc}$*/Neuzz*** **(c)** ***RESuzz/Neuzz***

Figure 6: Edge coverage ratio upon *Neuzz*

Table 3: Edge coverage results of *Neuzz*, *MTFuzz*, *Neuzz*$_{EdgeSelection}$ and *RESuzz*

Benchmarks	AFL	*Neuzz*	*MTFuzz*	*Neuzz*$_{EdgeSelection}$	*Neuzz*$_{havoc}$	*RESuzz*
bison	10,374	12,260	13,812	13,003	13,744	**14,543**
xmlwf	13,729	10,499	10,853	12,290	16,960	**19,731**
mupdf	13,665	16,705	16,603	17,002	19,995	**20,447**
pngimage	**4,077**	3,369	2,347	2,883	2,838	2,882
pngfix	7,134	5,181	5,767	7,307	7,422	**7,783**
pngtest	3,185	2,828	3,166	3,993	4,199	**4,229**
tcpdump	12,434	18,293	17,026	19,764	30,767	**32,983**
nasm	33,633	34,788	34,958	37,534	41,973	**43,493**
tiff2pdf	45,183	47,109	44,765	51,506	50,555	**53,319**
tiff2ps	20,862	23,705	22,671	23,649	24,247	**25,692**
tiffdump	2,416	3,239	2,617	**3,590**	3,554	3,560
tiffinfo	11,964	15,853	13,785	17,157	17,572	**18,259**
libxml	20,064	31,340	29,236	32,161	40,935	**44,171**
listaction	21,340	17,945	13,382	20,208	29,161	**32,447**
listaction_d	31,617	25,006	26,629	26,351	40,355	**46,762**
libsass	198,976	162,717	132,972	169,936	215,510	**218,130**
jhead	2,082	1,433	1,268	1,952	2,463	**2,464**
readelf	14,329	40,186	42,173	40,396	47,727	**51,855**
nm	11,154	16,159	**31,402**	19,040	22,605	24,519
strip	20,536	32,791	**41,520**	33,864	37,705	41,015
size	10,730	14,197	**18,675**	14,734	15,261	16,863
objdump	15,492	31,808	31,507	34,036	38,983	**40,387**
libjpeg	8,197	16,037	9,038	17,192	22,615	**23,057**
harfbuzz	26,420	35,502	44,342	47,877	45,412	**56,249**
base64	1,344	1,202	935	1,454	1,631	**1,644**
md5sum	2,871	3,168	3,101	3,415	**3,580**	3,518
uniq	713	756	725	792	794	**795**
who	2,917	2,973	2,680	3,262	3,255	**3,491**
Average	20,265	22,395	22,070	24,155	28,636	**30,510**

section follow the same settings in Section 3.2. The `threshold` for Algorithm 1 is set to 0.4 (more threshold setups would be evaluated later in Section 4.2.4).

4.2.1 Edge exploration effectiveness. Table 3 presents the experimental results of edge exploration effectiveness. We can find that overall, *RESuzz* can outperform all the existing baselines in terms of edge coverage averagely, e.g., *RESuzz* can outperform AFL by 51% (30,510 vs. 20,265 explored edges) and *Neuzz* by 36% (30,510 vs. 22,395 explored edges). Note that under the originally adopted measure of edge coverage, *RESuzz* can be also more advanced than *Neuzz* and *MTFuzz* by 19% and 22%. Such results suggest that combining *resource-efficient edge selection mechanism* and *AFL*$_{Havoc}$ for *Neuzz* can be rather powerful. Moreover, *Neuzz*$_{EdgeSelection}$ can outperform *Neuzz* by 8% (24,155 vs. 22,395 explored edges) and *MTFuzz* by 9% (24,155 vs. 22,070 explored edges). Specifically, *Neuzz* obtains 271 more edges averagely than *Neuzz*$_{EdgeSelection}$ in 2 projects while *Neuzz*$_{EdgeSelection}$ obtains 1,917 more edges averagely than *Neuzz* in the rest 26 projects. Such results indicate that the *resource-efficient edge selection mechanism* can enhance the overall effectiveness of the original *Neuzz*. In addition, *Neuzz*$_{havoc}$ also outperforms *Neuzz* by 28% (28,636 vs. 22,395 explored edges) and *MTFuzz* by 30% (28,636 vs. 22,070 explored edges). Such results demonstrate that appending *AFL*$_{Havoc}$ can significantly enhance the edge coverage of the neural program-smoothing-based fuzzers.

Figure 6 presents the correlation between the edge coverage advantage of *Neuzz*$_{EdgeSelection}$, *Neuzz*$_{havoc}$, *RESuzz* 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 (at the significance level of 0.05), i.e., the edge coverage advantage is not affected by the program size. Moreover, such advantage can also be rather consistent. Specifically, *RESuzz*, *Neuzz*$_{EdgeSelection}$, and *Neuzz*$_{havoc}$ can enable 36%, 8%, and 28% larger edge coverage than *Neuzz* on average with the standard deviation of 0.27, 0.13, and 0.22. Mingyuan:[Since Coefficient of Variation (CV) [5] is used to measure the dispersion of a probability distribution [41, 43, 53], we enable CV to measure the consistency of the performance for our improvements. The corresponding CV for such ratios are 19.6% (*RESuzz*/*Neuzz*), 11.5% (*Neuzz*$_{EdgeSelection}$/*Neuzz*) and 17.4% (*Neuzz*$_{havoc}$/*Neuzz*), which is significantly reduced compared with the CV of *Neuzz*/AFL (37.6%).] Therefore, we can summarize that our technical improvements can significantly and consistently enhance the neural program-smoothing-based fuzzers. Note that we can find under the edge coverage measure adopted in the original *Neuzz*/*MTFuzz* paper, *RESuzz* can enable the performance gain over *Neuzz* as 19.4% (2,651 vs. 2,219 explored edges) which is also rather significant.

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

4.2.2 For Neuzz$_{EdgeSelection}$*.* We further investigate *Neuzz*$_{EdgeSelection}$ in terms of the explored edges. Specifically, we can observe that overall, 24% edges do not need to be explored by applying *Neuzz*$_{EdgeSelection}$ under each iteration averagely (1,230 vs. 935 explored edges). Such result can indicate that applying *resource-efficient edge selection mechanism* can significantly save the effort on exploring the edges which cannot contribute to enhancing edge coverage.

4.2.3 For AFL$_{Havoc}$*.* We further investigate the efficacy of *AFL*$_{Havoc}$ in terms of the *Edge Discovery Rate*. To this end, we also include

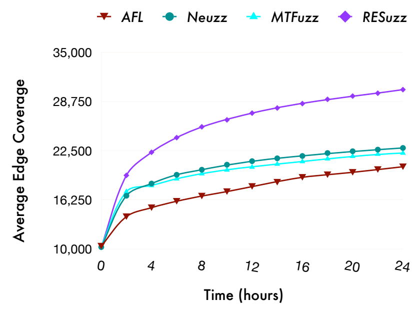


Figure 7: Edge coverage of *RESuzz* in terms of time

AFL_{Havoc} and the havoc mechnism adopted by *RESuzz* and the gradient guidance stage of $Neuzz_{EdgeSelection}$ and *RESuzz* (represented as $RESuzz_{havoc}$, $Neuzz_{EdgeSelectiongradient}$, and $RESuzz_{gradient}$ respectively) for performance comparison. Figure 8 presents our evaluation results. We can observe that overall, $RESuzz_{havoc}$ can significantly outperform all the other studied approaches on top of all the studied benchmarks, e.g., $RESuzz_{havoc}$ can be 2X more efficient than AFL_{Havoc} (8.32 v.s. 2.256). Accordingly, we can infer that the gradient guidance adopted by *RESuzz* can provide more "high-quality" seeds for launching its havoc mechanism to explore more new edges than $AFL_{Deterministic}$. Furthermore, we can observe that the EDR of $RESuzz_{gradient}$ can also outperform the original $Neuzz_{EdgeSelectiongradient}$ by 67%. Therefore, we also infer that $RESuzz_{havoc}$ can advance the edge exploration efficiency of $RESuzz_{gradient}$. To summarize, combining the two technical improvements can mutually advance their efficiency of edge exploration.

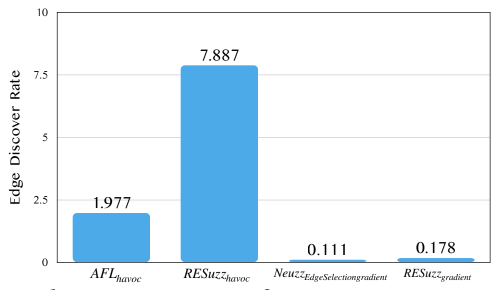


Figure 8: *Edge Discovery Rate* of AFL, $Neuzz_{EdgeSelection}$ and *RESuzz*

4.2.4 Impact from parameter settings. We attempt to investigate the performance impact from different `thresholds` of the *resource-efficient edge selection mechanism.* Specifically, Figure 9 presents the average edge coverage results among all the benchmark projects where x in $RESuzz_x$ represents the studied threshold for *RESuzz*. Overall, we can observe that while setting `threshold` to 0.4 for *RESuzz* can lead to the optimal performance, other threshold setups for *RESuzz* can incur close performance. Such consistent results can suggest that our *resource-efficient edge selection mechanism* can be quite robust and resilient to diverse threshold setups.

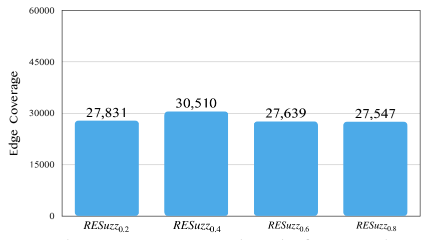


Figure 9: Edge coverage under different thresholds

4.3 Implications

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

Do not waste time on improving neural network models. Our study results reveal that the edge coverage performance can be essentially impacted by how the resulting gradients of the adopted neural network models can reflect the *explored edge—existing seed* relations rather than their generalization or prediction capabilities. Therefore, we recommend not to take just simply improving neural network models as a direction of technical improvement.

Think twice on applying program 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 [16][6][24]. Therefore, we recommend to think carefully before adopting program analysis techniques to enhance neural program-smoothing-based fuzzing—users need to consider the program features (e.g., size) and weigh the cost.

Edge selection? Yes! Gradient computation? Unnecessary. Our evaluations reveal that selecting "promising" edges for mutations can be quite effective in enhancing the edge coverage performance on top of varying-sized programs. Furthermore, we ask an insightful question: 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 "misused", i.e., their generalization and prediction capabilities are unused, such question can then be transformed as—is it necessary to use neural networks only for computing gradients to represent the *explored edge—existing seed* relations? To answer such question, it is worthwhile to attempt for lightweight alternatives to represent the *explored edge—existing seed* relations which can be potential research directions.

No doubt to adopt random search strategy. Our study results indicate that the edge coverage performance of the neural program-smoothing-based fuzzers can be significantly enhanced by appending the random search strategy AFL_{Havoc}. Intuitively, we suggest the users to append such havoc stage to any of their adopted fuzzers when possible. Accordingly, one possible research direction can be how to integrate such random 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* where we implemented *RESuzz* accordingly. Meanwhile, to implement AFL_{Havoc}, we also reused such code from the original AFL for the *RESuzz* implementation. Moreover, all the authors including two faculty members and senior engineers from industry manually reviewed *RESuzz* 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, 33, 36, 55] published recently.

Threats to construct validity. The threat to construct validity mainly lies in the metrics used. While the edge coverage measure 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 measure, the performance advantages of *Neuzz* and *MTFuzz* are significantly reduced, our proposed approach *RESuzz* can incur quite strong performance gain under both measures.

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

We divide our discussions on the existing fuzzing approaches into two groups—general-purpose fuzzing and domain-specific fuzzing.

General-purpose fuzzing. Böhme et al. [4] proposed *AFLFast* which designs a seed selection strategy to weigh seeds via Markov Chain on top of the original AFL. They also proposed *AFLGo* [3] to take advantages in weighting seeds based on edge structures to explore the target point specified by users. Rawat et al. [42] 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. [38] presented a technique to learn the token of program by tainting to fuzz more efficiently. Lemieux et al. [32] proposed *fairfuzz* to increase graybox fuzz testing coverage by focusing on fuzzing rare branches of program. Fietkau et al. [20] introduced *KleeFL* which focuses on the paths that can be hardly explored by AFL and uses Klee to generate seeds via symbolic execution to reach such paths. Chen et al. [10] introduced *Angora*, a mutation-based fuzzer that solves path constraint without symbolic execution by taint checking and searching. Manès et al. [37] proposed *Ankou*, a grey-box fuzzing solution based on different combinations of execution information. Fioraldi et al. [21] introduced *WEIZZ* to automatically generate and mutate inputs for unknown chunk-based binary formats.

Domain-specific fuzzing. Chen et al. [11] proposed *Classming* which manipulates the control flow within a class file to fuzz the deeper stages of JVM systems. Chen et al. [9] introduced a new grey-box fuzzing technique named *MUZZ* that hunts for bugs in multithreaded programs. Chen et al. [12] proposed *POLYGLOT*, a generic fuzzing framework that generates high-quality test cases for exploring processors of different programming languages. Choi et al. [15] presented a practical static binary analyzer *NTFUZZ* which is a type-aware Windows kernel fuzzing framework that automatically infers system call types on Windows at scale. Padhye et al. [39] automatically guided QuickCheck-like random input generators to semantically analyze test programs for generating test-oriented Java bytecode. Park et al. [40] proposed an aspect-preserving mutation strategy to preserve desirable properties in the seeds for mutation to fuzz JavaScript engines. Wu et al. [51] proposed *Simulee* to parse constraints of input from a given GPU kernel function and mutated the input to fuzz CUDA programs. Wen et al. [49] proposed a memory usage guided fuzzing technique to generate the excessive memory consumption inputs and trigger uncontrolled memory consumption bugs. Xie et al. [52] introduced *DeepHunter* to fuzz deep neural networks by a semantically preserved mutation strategy.

6.2 Deep Learning on Fuzzing

In addition to *Neuzz* and *MTFuzz*, Lyu et al. [35] introduced *SmartSeed* which used Generative Adversarial Networks [26] to generate seeds from learning the patterns of valuable existing seeds. Zong et al. [57] proposed *fuzzguard*, a deep-learning-based approach to predict the reachability of inputs before executing the program. Liu et al. [34] proposed *DeepFuzz* to automatically and continuously generate C programs by a generative Sequence-to-Sequence model [14]. Godefroid et al. [25] divided fuzzing tasks into two categories, i.e., learning input format to fuzz deeper and breaking input format to trigger defects. In this paper, we propose *RESuzz* with *resource-efficient edge selection mechanism* and AFL_{Havoc} 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. [30] provided guidelines on evaluating the effectiveness of fuzzers by assessing the experimental evaluations carried out by different fuzzers. Gavrilov et al. [22] 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. [23] performed an empirical study on multiple artificial vulnerability benchmarks to understand how close these benchmarks reflect reality. Herrera et al. [28] investigated and evaluated how seed selection affects a fuzzer's ability to find bugs in real-world software. In this paper, we conduct an empirical study to investigate the power and the limitation for neural program-smoothing-based fuzzing and reveal various findings/guidelines for future fuzzing research.

7 CONCLUSION

In this paper, we investigate the strengths and limitations of neural program-smoothing-based fuzzing approaches, e.g., *MTFuzz* and *Neuzz*. We first extend our benchmark by injecting the projects which are widely adopted in the existing fuzzing evaluations. Next, we evaluate *Neuzz* and *MTFuzz* on the extensive benchmark to study their effectiveness and efficiency. Inspired by our study findings, we propose *RESuzz* combining two technical improvements, i.e., the *resource-efficient edge selection mechanism* and the AFL havoc mechnism. The evaluation results demonstrate that *RESuzz* can significantly outperform *Neuzz* and *MTFuzz* in terms of the average edge coverage. Furthermore, our results also reveal various findings/guidelines for advancing future fuzzing research.

REFERENCES

[1] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 1–4.

[2] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and Reflections. *IEEE Software* (2020).

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[5] Charles E Brown. 1998. Coefficient of variation. In *Applied multivariate statistics in geohydrology and related sciences*. Springer, 155–157.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[7] Swarat Chaudhuri and Armando Solar-lezama. 2010. Smooth interpretation. In *In PLDI*.

[8] Swarat Chaudhuri and Armando Solar-Lezama. 2011. Smoothing a Program Soundly and Robustly. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 277–292. https://doi.org/10.1007/978-3-642-22110-1_22

[9] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[10] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[11] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[12] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*.

[13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[15] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. (2021).

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[17] Jared DeMott, Richard Enbody, and William F Punch. 2007. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. *BlackHat and Defcon* (2007).

[18] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990), 179–211.

[19] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. 2006. Sidewinder: An Evolutionary Guidance System for Malicious Input Crafting. *Black Hat USA* (2006).

[20] Julian Fietkau, Bhargava Shastry, and JP Seifert. 2017. KleeFL-seeding fuzzers with symbolic execution. In *Poster presented at USENIX Security Symposium*.

[21] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3395363.3397372

[22] M. Gavrilov, K. Dewey, A. Groce, D. Zamanzadeh, and B. Hardekopf. 2020. A Practical, Principled Measure of Fuzzer Appeal: A Preliminary Study. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 510–517. https://doi.org/10.1109/QRS51102.2020.00071

[23] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. 2020. An empirical study on benchmarks of artificial software vulnerabilities. *arXiv preprint arXiv:2003.09561* (2020).

[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[25] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

[26] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial networks. *arXiv preprint arXiv:1406.2661* (2014).

[27] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks* 18, 5-6 (2005), 602–610.

[28] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Tony Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2021)*.

[29] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[31] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[32] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[33] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. 2019. V-fuzz: Vulnerability-oriented evolutionary fuzzing. *arXiv preprint arXiv:1901.01142* (2019).

[34] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

[35] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. 2018. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606* (2018).

[36] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.

[37] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1024–1036. https://doi.org/10.1145/3377811.3380421

[38] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–37.

[39] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[40] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642.

[41] Rahul Potharaju and Navendu Jain. 2013. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 conference on Internet measurement conference*. 9–22.

[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[43] Yanyan Ren, Shriram Krishnamurthi, and Kathi Fisler. 2019. What Help Do Students Seek in TA Office Hours. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 41–49.

[44] Github Repository. 2021. Program smoothing fuzzing. https://github.com/PoShaung/program-smoothing-fuzzing.

[45] Dongdong She. 2020. neuzz repository. https://github.com/Dongdongshe/neuzz.

[46] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 737–749.

[47] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.

[48] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance*. Artech House.

[49] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*.

Association for Computing Machinery, New York, NY, USA, 765–777. https://doi.org/10.1145/3377811.3380396

[50] Wikipedia. 2020. Fuzzing. en.wikipedia.org/wiki/Fuzzing. Online; accessed 27-Jan-2020.

[51] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: Detecting cuda synchronization bugs via memory-access modeling. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 937–948.

[52] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[53] Yalong Yang, Bernhard Jenny, Tim Dwyer, Kim Marriott, Haohui Chen, and Maxime Cordeil. 2018. Maps and globes in virtual reality. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 427–438.

[54] Xin Yao, Yong Liu, and Guangming Lin. 1999. Evolutionary programming made faster. *IEEE Transactions on Evolutionary computation* 3, 2 (1999), 82–102.

[55] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761.

[56] Michał Zalewski. 2020. American Fuzz Lop. https://github.com/google/AFL.

[57] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2255–2269.