GAN-based Seed Generation for Efficient Fuzzing

Shyamili Toluchuri[1], Aishwarya Upadhyay[1], Smita Naval[1], Vijay Laxmi[1] and Manoj Singh Gaur[2]
[1]*Department of Computer Science and Engineering, Malaviya National Institute of Technology Jaipur, India*
[2]*Department of Computer Science and Engineering, Indian Institute of Information Technology Jammu, India*

Keywords: Software Testing, Vulnerability Assessment, Fuzzing, Gray-Box, GAN.

Abstract: Software vulnerabilities are a substantial concern in development, with testing being crucial for identifying mistakes. Fuzzing, a prevalent technique, involves modifying a seed input to discover software bugs. Selecting the right seed is pivotal, as indicated by recent research. In our study, we extensively analyze leading gray-box fuzzing tools, applying them to identify bugs across 22 open-source applications. An innovative addition to our approach is the integration of a Deep Learning Generative Model (DCGAN). This model offers a novel method for generating seed files by learning from crash files in previous experiments. Notably, it excels in generating images across various formats, enhancing flexibility in applications with consistent input formats. The system's primary advantages lie in its flexibility and improved fuzzing efficiency. It outperforms other applications in identifying vulnerabilities swiftly, marking a significant advancement in the current state of affairs.

1 INTRODUCTION

Fuzzing is an automated testing technique that excels at finding vulnerabilities in applications by exploring edge cases and maximizing code coverage (Oehlert, 2005). This surpasses manual testing and is crucial for vulnerability discovery (e.g., Microsoft Windows TIFF image vulnerability and Trend Micro zero-day). Traditional fuzzing, however, has limitations due to random data usage, hindering bug identification (Payer, 2019). Techniques like seed scheduling (Choi et al., 2023) and Machine Learning (ML) integration are being explored to improve coverage. Here, an ML model trained on crash data can generate targeted seeds, leading to more effective bug discovery. However, challenges remain, such as limited performance analysis of existing grey-box fuzzers and the lack of image-specific seed generation models.

Our contributions to this paper are:

i A comprehensive evaluation of four leading gray box fuzzers (AFL++, AFLfast, AFLgo, and Honggfuzz) across 22 diverse open-source applications, aiming to identify the most effective fuzzer in various software landscapes.

ii Beyond crash counts, a deeper assessment of fuzzer effectiveness based on crash discovery and achieved code coverage, offering a holistic understanding of vulnerability exposure.

iii Introduction of a novel seed generation model using Generative Adversarial Networks (GANs), crafting highly relevant input seeds to maximize code coverage and crash discovery.

iv Exploration of fuzzing frontiers by applying GAN-generated inputs to image-based applications, showcasing the effectiveness of our approach in enhancing the security of image processing software.

The paper is organized as follows: The next section contains a detailed survey of fuzzing techniques, ways to improve fuzzers, and all the recent research on using machine learning models to generate seeds and consequently improve results. Moving on, Section III contains information about the technical setup for the experiment, fuzzers, target applications, details of the DCGAN model used, and the evaluation metrics. In the end Section IV shows the results and the analysis of all the experiments performed and in Section V the work is concluded and its future aspects and direction are discussed.

2 RELATED WORKS

In recent years, there has been an increase in the number of methods for detecting vulnerabilities, and one

Toluchuri, S., Upadhyay, A., Naval, S., Laxmi, V. and Gaur, M.
GAN-based Seed Generation for Efficient Fuzzing.
DOI: 10.5220/0012761900003767

In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 686-691
ISBN: 978-989-758-709-2; ISSN: 2184-7711

of these methods is known as fuzzing. This has resulted in the growth of a large number of fuzzers that can be utilized for various targets, including the web, the network, the application, and the kernel. As a result, there is a requirement for the development of better methods for evaluating fuzzing.

Seed selection plays a crucial role in fuzzing effectiveness (Herrera et al., 2021). Saha *et al.* propose that generating seed inputs beyond the fuzzer's algorithm can improve coverage of rare paths (Saha et al., 2022). Machine learning (Ramadan et al., 2022, Saavedra et al., 2019, Wang et al., 2020b) and deep learning (Miao et al., 2022, Li et al., 2022) techniques have shown promise in seed generation. Cheng *et al.* use an RNN-based generative model for PDF files (Cheng et al., 2019), while Godefroid et al. employ a sequence-to-sequence model (Godefroid et al., 2017). Wang *et al.*. leverage various deep neural network models for seed generation (Wang et al., 2020a, Wang et al., 2017). NeUFuzz (Wang et al., 2019) and MTFuzz (She et al., 2020) utilize deep learning for intelligent seed selection, while SmartSeed (Lyu et al., 2018) leverages a WGAN model for seed generation across multimedia formats. These advancements highlight the growing adoption of machine learning for improved seed generation and fuzzing effectiveness.

3 EXPERIMENTAL SETUP

3.1 Workflow

Our workflow utilizes AFL++ for grey-box fuzzing with a standard seed set (Figure 1). Crashes and novel code paths serve as training data for a DCGAN model. Images are pre-processed for model training, allowing pattern recognition. The model generates new images based on learned patterns. These images are then used to fuzz the applications again. Finally, crash triage analyzes crashes from both seed sets to identify exploitable vulnerabilities and pinpoint relevant code locations.

Note that the process can be a closed-loop. The machine learning model can make seed files that can be used by the fuzzing tools to find new crashes and paths. Then, we can improve our machine learning model's training set by putting in files that cause new crashes or paths.

In order to facilitate experimentation the fuzzers have been installed and executed in Kali Linux 2022 64bit Virtual Machines running on MacOS 13.3, Windows 11, and a Dell Inc. Desktop running Ubuntu 22.04.2 LTS.We have chosen four well-known gray-box and mutational fuzzers for our experiments and tested them on the target application which are AFLplusplus, AFLfast,AFLgo and Honggfuzz. All the fuzzers have a Command Line User Interface (CLI), they are installed and used from the terminal only. Table 1 provides a comprehensive analysis of each of the target applications. Our experiments are based on both instrumented and non-instrumented applications.

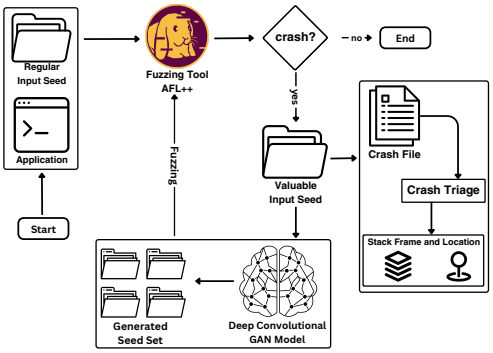

Figure 1: Workflow of the Experiment.

3.2 Deep Learning Model

Generative Adversarial Networks (GANs) (Goodfellow et al., 2020) are a type of machine learning algorithm capable of generating realistic data, including images, text, and music. GANs consist of two competing models: a generator and a discriminator. The generator creates new data, while the discriminator distinguishes between real and fake data (Jabbar et al., 2021). Through adversarial training, these models improve iteratively. Deep Convolutional GANs (DCGANs) (Radford et al., 2015) specifically utilize convolutional layers in both the generator and discriminator. They have demonstrated effectiveness in generating high-quality images of faces, objects, and scenes, as well as other types of data like text and music. Despite being still under development, GANs have the potential to revolutionize data generation and interaction.

Training Images: Low crash yield in image applications led to seed set upscaling (Figure 2). A Generative Adversarial Network (GAN) was trained on crashes and hangs from the initial fuzz run. Images were pre-processed to 32x32 pixels for training (93 total).

Generated Images: Some of the images that were created are displayed in Figure 3. After the GAN has been trained, it can be used to generate new images in the format and the same size as before, which is 32x32x3. There have been 500 of these images utilized.

Table 1: Application Details.

S No.	Category	Apps	Version	Year	Input type
1		Xpdf	3.02	2007	pdf
2		Xpdf	3.03	2011	pdf
3		Libxml2	2.9.4	2012	xml
4	**Text**	Libxml2	2.9.5	2017	xml
5		Jsonparse	NA	NA	json
6		Mupdf	1.22.0	2023	pdf
7		Bloomy Sunday	NA	2016	txt
8		Libpng	1.6.39	2022	png
9		Libtiff	4.0.4	2012	tiff
10		Libtiff	4.0.5	2015	tiff
11		Libexif	0.6.14	2002	tiff, jpg
12	**Image**	Libexif	0.6.22	2020	tiff, jpg
13		imgp	2.8	2020	png, jpg
14		Flameshot	12.1.0	2022	png, jpg
15		ImageMagick	7.0.11	2021	png, tiff, jpg
16		GIMP	2.8.16	2020	png, jpg
17		VLC	3.0.7.1	2019	wmv
18	**Multimedia**	FFmpeg	N-110105-g9bf1848acf	2022	mov, mkv, mp4
19		mpv	0.35.0	2022	mpv
20		Wireshark	3.6.0	2021	pcap
21	**Network**	Crazy HTTP Server	NA	2020	pcap
22		tcpdump	4.9.2	2017	pcap

Figure 2: Images on which the GAN model is trained.

Figure 3: Generated Images from the GAN Model.

3.2.1 Evaluation Metrics

We have evaluated our experiments using the following metrics.

1. **No. of Crashes**: Crashes are the visible signs of a trigger, which is when the program stops running and can be easily discovered. This metric monitors these events. It always guesses fewer bugs than there are.
2. **No. of Hangs**:AFL++ tracks unique hangs, which indicate the fuzzer has encountered new execution paths in the target program. These new paths can expose potential bugs. More unique hangs suggest a more effective fuzzer in finding program bugs.
3. **Vulnerability Detection Speed (VDS):** This metric aims to estimate the speed for finding crashes computed as the ratio of the number of crashes found during a period of time (Equation 1).

$$V_b = \frac{No.\ of\ crashes}{No.\ of\ hours} \tag{1}$$

4. **Density:** Density (Equation 2) measures crashes per fuzzing cycle. A lower density suggests the fuzzer efficiently finds bugs with fewer inputs.

$$Density = \frac{No.\ of\ crashes}{No.\ of\ cycles} \quad (2)$$

5. **Edge Coverage:** Edge coverage reflects fuzzing's exploration depth for bugs. While high coverage doesn't guarantee finding all bugs, low coverage suggests limited bug-finding potential.
6. **Mutation Rate:** Mutation rate (Equation 3) refers to the portion of inputs mutated per fuzzing iteration. A higher rate generates more varied inputs, potentially finding more bugs, but also risks an increase in false positives.

$$M_r = \frac{\Delta Corpus\ count}{Initial\ corpus\ count} \quad (3)$$

4 ANALYSIS

After compiling all applications and defining input seed sets, gray-box fuzzers AFL++, AFLfast, AFLgo, and Honggfuzz were employed to fuzz the applications. AFL++ detected the highest number of crashes in Libxml2, Libtiff, and Xpdf compared to AFLgo and AFLfast (Figure 4). Additionally, AFL++ exhibited superior Edge Coverage compared to the other fuzzers (Figure 5).

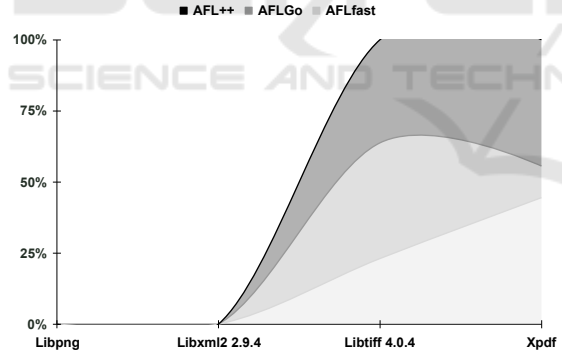

Figure 4: AFLplusplus finds the most files that will crash.

Nine out of 22 applications were fuzzed without instrumentation (a separate fuzzer plug-in). This mode produced more hangs than crashes. These files can be used as seeds for further fuzzing (refer to Figure 6 for application-specific hangs). Notably, Bloomy Sunday and VLC media player exhibited the most hangs.

Focusing on the most efficient fuzzer, AFL++, we analyzed its impact on edge coverage, a key metric for bug detection. AFL++ effectively explores new code paths (edges) by mutating input seeds.

Figure 7 (a) shows that both Libtiff versions achieved higher edge coverage and mutation rates

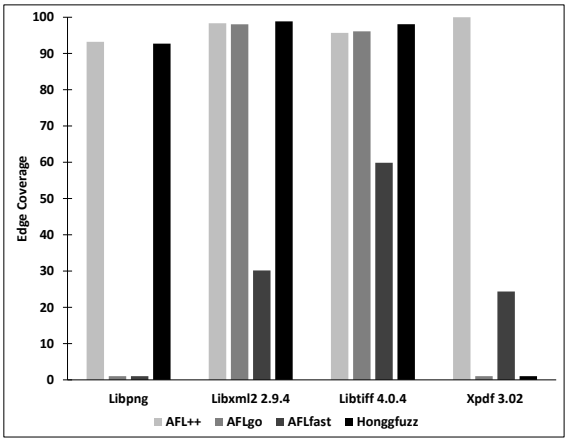

Figure 5: Edge Coverage.

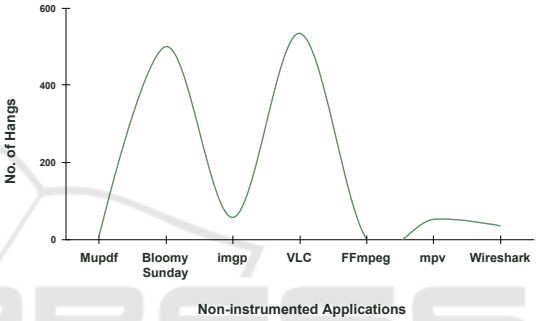

Figure 6: Trend of No. of Hangs in Non-instrumented Applications.

compared to other applications using standard seeds. Interestingly, image processing applications generally outperformed text-based ones in edge coverage. Fuzzing time is another important factor. Our experiments (Figure 7 (b)) revealed a correlation between fuzzing time and crashes - more time resulted in more crashes. However, hangs were not significantly impacted by fuzzing time. The complex nature of image processing applications made them more susceptible to vulnerabilities, with faster vulnerability detection rates compared to other categories (Figure 8 (a)). Mutation rates remained consistent across categories (Figure 8 (b)).

AFL++ excelled in edge coverage, mutation rate, and vulnerability detection for image processing applications like Libtiff. We further enhanced these metrics using crash data-based seeds. This new seed set significantly improved crashes, hangs, and detection speed for Libtiff (Table 2) and Libpng (Table 3), while Imgp showed benefits in hangs. Limited improvements in other image applications suggest factors like insufficient fuzzing time, seed inefficiency, or fuzzer limitations might be at play.

Post-fuzzing, crash triage is crucial to identify ex-

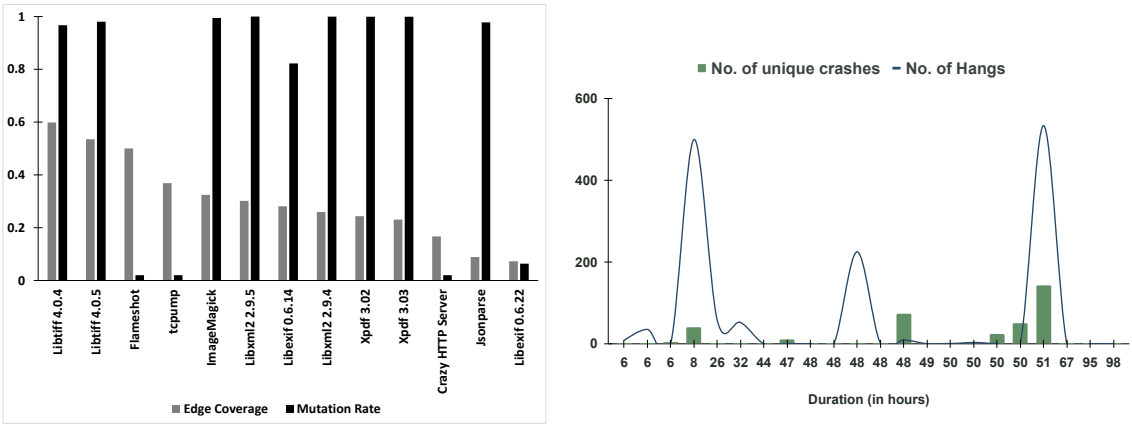

Figure 7: (a) Comparison of Edge Coverage and Mutation Rate in Applications (b) Shows the relation between unique crashes and hangs with respect to the hours spent fuzzing the application.

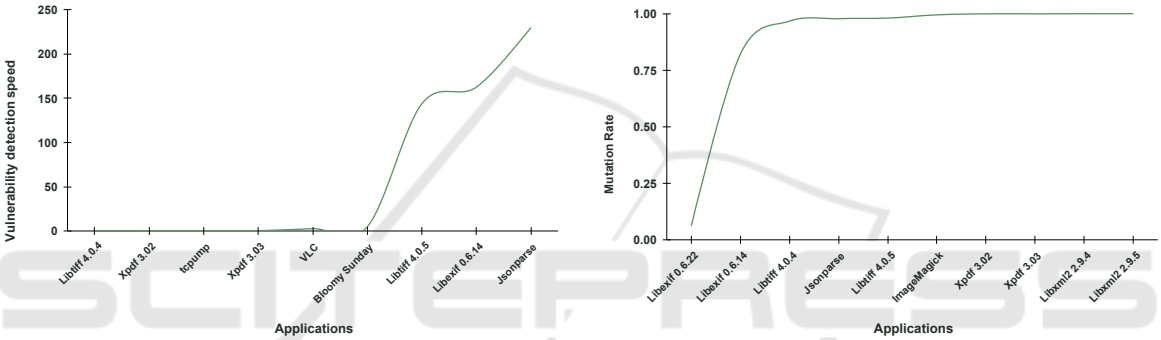

Figure 8: (a) Shows the VDS (V_b) in the applications (b) Shows the Mutation Rate (M_r) of the Applications.

Table 2: Results of fuzzing Libtiff with the new input seed set.

	With regular seed input		With the generated input seed	
	Libtiff 4.0.4	**Libtiff 4.0.5**	**Libtiff 4.0.4**	**Libtiff 4.0.5**
No. of Crashes	2	74	479	490
No. of Hangs	0	9	76	14
VDS (V_b)	0.04	143.89	8860	1943.75
Density	0	0	12305.55	8481.81

Table 3: Results of fuzzing Libpng with the new input seed set.

	With the regular seed input	With the generated seed input
	Libpng	**Libpng**
No. of Hangs	0	1
Edge Coverage	0%	6%
Mutation Rate	0%	29%

ploitable crashes. Not all crashes lead to vulnerabilities. Our analysis revealed two main crash types in the standard seed set: segmentation faults and buffer overflows (Table 4). Segmentation faults often indicate memory access issues and can stem from leaks, out-of-bounds access, or uninitialized pointers. Crash triage is focused on these crashes, with the remainder discarded. Libtiff 4.0.5 crashes with the standard seed set revealed no exploitable vulnerabilities. These crashes stemmed from a known heap buffer overflow in tif_print.c (Lhee and Chapin, 2003). The new seed set, however, identified vulnerabilities in both Libtiff 4.0.5 and 4.0.4 (Table 5), demonstrating the model's effectiveness in enhancing fuzzing.

Table 4: Crash triage results with the regular seed input.

Application	Error	Method Used
Xpdf 3.03	Segmentation Fault	AFLTriage
Xpdf 3.02	Segmentation Fault	AFLTriage
Libtiff 4.0.5	None	AFLTriage
Libtiff 4.0.4	None	AFLTriage
Libexif 0.6.14	Segmentation Fault, Signal Abort	AFLTriage
tcpdump	Heap Buffer Overflow	Used the crash file as an input
Libttiff 4.0.4	Heap Buffer Overflow	Used the crash file as an input

Table 5: Crash triage results with the generated input seed.

Application	Error	Method Used
Libtiff 4.0.5	Heap Buffer Overflow	AFLTriage + Used the crash file as an input
Libtiff 4.0.4	Heap Buffer Overflow	AFLTriage

Crash analysis revealed a heap buffer overflow in the tif_print.c function's fprintf() call. fprintf() was allocated one byte, but attempted to read a second character, causing the overflow. The format string %s likely expected a null-terminated string, but potentially encountered an unterminated pointer.

5 FUTURE WORK AND CONCLUSIONS

This paper offers a thorough analysis of fuzzing, identifying AFL++ as the most effective gray-box fuzzer compared to AFLfast, AFLgo, and Honggfuzz. It covers fuzzing history, techniques, application instrumentation, and crash triage. AFL++ yields positive results across various applications, with image format crashes used to train a Deep Convolutional Generative Adversarial Network for generating a new seed set. This new seed set enhances fuzzing metrics and uncovers critical vulnerabilities. Future research could involve designing a seed generation model compatible with additional fuzzers and incorporating all relevant file formats.

REFERENCES

Cheng, L., Zhang, Y., Zhang, Y., Wu, C., Li, Z., Fu, Y., and Li, H. (2019). Optimizing seed inputs in fuzzing with machine learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 244–245. IEEE.

Choi, G., Jeon, S., Cho, J., and Moon, J. (2023). A seed scheduling method with a reinforcement learning for a coverage guided fuzzing. *IEEE Access*, 11:2048–2057.

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

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM*, 63(11):139–144.

Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., and Hosking, A. L. (2021). Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 230–243.

Jabbar, A., Li, X., and Omar, B. (2021). A survey on generative adversarial networks: Variants, applications, and training. *ACM Computing Surveys (CSUR)*, 54(8):1–49.

Lhee, K.-S. and Chapin, S. J. (2003). Buffer overflow and format string overflow vulnerabilities. *Software: practice and experience*, 33(5):423–460.

Li, S., Xie, X., Lin, Y., Li, Y., Feng, R., Li, X., Ge, W., and Dong, J. S. (2022). Deep learning for coverage-guided fuzzing: How far are we? *IEEE Transactions on Dependable and Secure Computing*.

Lyu, C., Ji, S., Li, Y., Zhou, J., Chen, J., and Chen, J. (2018). Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606*.

Miao, S., Wang, J., Zhang, C., Lin, Z., Gong, J., Zhang, X., et al. (2022). Deep learning in fuzzing: A literature survey. In *2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information (ICETCI)*, pages 220–223. IEEE.

Oehlert, P. (2005). Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62.

Payer, M. (2019). The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security & Privacy*, 17(1):78–82.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Ramadan, A. E.-R. K. E.-D., Bahaa, A., and Ghoneim, A. (2022). A systematic literature review on software vulnerability detection using machine learning approaches. *FCI-H Informatics Bulletin*, 4(1):1–9.

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

Saha, S., Sarker, L., Shafiuzzaman, M., Shou, C., Li, A., Sankaran, G., and Bultan, T. (2022). Rare-seed generation for fuzzing. *arXiv preprint arXiv:2212.09004*.

She, D., Krishna, R., Yan, L., Jana, S., and Ray, B. (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*, pages 737–749.

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

Wang, X., Hu, C., Ma, R., Li, B., and Wang, X. (2020a). Lafuzz: neural network for efficient fuzzing. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 603–611. IEEE.

Wang, Y., Jia, P., Liu, L., Huang, C., and Liu, Z. (2020b). A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749.

Wang, Y., Wu, Z., Wei, Q., and Wang, Q. (2019). Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352.