# A Study On Fuzzing Techniques and The Adaptability of Machine Learning Approach

**Dr. Deepa A, Associate professor, Department of MCA, Nehru college of Engineering and Research Centre, Thrissur, India, achatdeepajayan@gmail.com**

**Basil Jiji, Student Scholar MCA, Nehru College of Engineering and Research Centre, Thrissur, India, basilgg24@gmail.com**

**Abstract**: **Over the past several decades, fuzzing has significantly enhanced software development and testing. Applications of fuzzing have been the subject of recent research. ML provides practical techniques to solve difficulties in the fuzzing process. This review examines the most recent work on fuzzing and ML. This review addresses effective ML fuzzing applications, briefly examines difficulties encountered, and encourages more study to address fuzzing bottlenecks.**

***Keywords —Deep Neural Network, Fuzzing, Mutation Operator, Machine Learning, Testcase, Vulnerability***

## I. INTRODUCTION

Fuzzing is a technique used to test a programme for errors and vulnerabilities by feeding it a large number of generated inputs, both valid and invalid. Fuzzing is technique used inorder to find bugs and vulnerabilities. Fuzzing is a technique in which a large number of generated inputs both valid and invalid are fed into a programme. A large portion of this process is automated by fuzzers, which collect initial programme knowledge and report on any intriguing programme states found. Frequently, a human user analyses these output programme states in addition to providing initial programme knowledge. Historically, "interesting programme states" were software crashes that exposed vulnerabilities and flaws in the programme, but more sophisticated programme monitoring techniques now enable the detection of additional interesting states. A fuzzer's objective is to produce inputs that make the programme execute programme paths in order to find those that result in intriguing programme states. As a result, coverage the variety of programme paths explored is frequently used to evaluate fuzzers. A Naive fuzzer is one that creates input that is entirely random and feeds it to a software. Although naive fuzzers are relatively simple to create, they are not likely to quickly achieve interesting programme states. Modern fuzzers can be divided into three main categories: mutation-based, generation-based, and evolutionary. Blindly altering or changing the input given to the programme is what mutation-based fuzzers do. The majority of the time, mutation-based fuzzers are unable to make intelligent mutation selections because they are unaware of the expected input format or specifications. Peach is a fuzzer that has the ability to perform both generation-based and mutation-based fuzzing. The expected input format or protocol is obtained by generation-based fuzzers through specifications. On the basis of these requirements, generation-based fuzzers produce inputs. Peach and Sulley, a Python fuzzing framework, are examples of generation-based fuzzers that can produce inputs for file transfer protocols, network protocols, and file formats. The most recent kind of fuzzers, evolutionary fuzzers, improve on mutation-based fuzzers by favouring some inputs over others for mutation. In particular, evolutionary fuzzers try to assess what each input makes the programme do and modify their behaviour in response to that assessment. Modern evolutionary fuzzers actually rank inputs using a fitness function (often coverage) and pick the top-ranked inputs to mutate. Honggfuzz, AFL, and libFuzzer are a few examples of evolutionary fuzzers.

## II. LITERATURE SURVEY

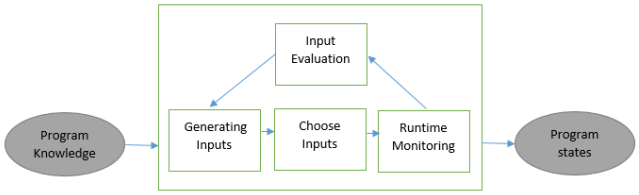
Please Over the past three decades, fuzzers, or automated tools to perform fuzzing, have been crucial in quality assurance, system administration, and vulnerability assessment [19, 20, 41, 23]. In this survey, this examine how some contemporary fuzzers incorporate various forms of machine learning. Modern fuzzers now incorporate techniques from other disciplines (ML). Due to their widespread use, this concentrates specifically on fuzzers used for vulnerability assessment. Computer models can be trained using machine learning to carry out specific tasks without having to be explicitly programmed. From image processing to sequence modelling, ML techniques are used to solve a variety of issues [25, 32, 22]. This study is concentrated on three main ML subtypes, each of which is best suited for a particular task.

 DOI : 10.35291/2454-9150.2023.0052 

When training a model to determine the class label of a given data point, such as whether an image contains a particular object, supervised learning is used. Data sets with explicit labels for each data point are necessary for this kind of machine learning. Instead of labelling classes, unsupervised learning is used to train a model to look for patterns or similarities between data points. This type of ML is used when the data does not have explicit labels. Reinforcement learning is used to train a model, often referred to as an agent, to take an optimal set of actions in an environment. This type of ML rewards the agent for each action it takes in the environment. Similar to supervised learning, the rewards act as a label for the agent and provide an indication of the optimal actions to take. Thus, the agent can be trained to take a set of actions which lead to the highest reward. Each of these three types can also take on a particular form of ML called deep learning. Deep learning refers to a type of hierarchical learning that can be used to learn the underlying features and structure of a set of data points [25].

This survey looks at how machine learning (ML) has been used to answer key research questions in fuzzers used for programme vulnerability assessment. This study is representative rather than exhaustive; no attempt is made to cover all ML in fuzzer applications. This provides a brief summary of the fuzzing process in Section 3. This examines past and present ML-fuzzing research in Section 3 of this paper. This also talks about the challenges of using ML for fuzzing. This sums up how ML has been used to tackle fuzzing and how it might be used to tackle fuzzing in the future.

## III. WORKING OF FUZZING




### Phase 1: Program Knowledge

Typically, program knowledge is developed manually and provided by the human user. The following three stages of the fuzzing process, however, are automatically iterated over by fuzzers. A fuzzer will typically continue running until the user ends it. It will generate new inputs, down-select and order inputs to send to the programme, watch the programme for interesting programme states, and repeat. Next, examining of these phases.

### Phase 2: Testcase Generation

The fuzzer creates inputs to feed to the programme through the identified interfaces in the first stage of the fuzzing process using programme knowledge. This phase's objective is to produce inputs that will investigate novel and intriguing programme states. Only the most pertinent of the new inputs generated here will be sent on to the programme in the following stage of the fuzzing process. The fuzzer here, however, aims to produce the most pertinent inputs.

### Phase 3: Choose Inputs

The fuzzer chooses and arranges the inputs to be sent to the programme in the second stage of the fuzzing process. Remember that a fuzzer's objective is to rapidly test new programme paths. However, as stated by Böhme et al., the majority of inputs exercise the same few programme paths [7, 8]. A fuzzer must effectively utilise its input corpus and reduce the amount of computation required to find new programme paths in order to counter this. This propensity of inputs to explore specific programme paths can be thwarted by input test scheduling, also known as seed selection.

To determine which new inputs are most likely to result in novel and disparately interesting programme states, input test scheduling ranks and selects inputs and input order. Test scheduling for vulnerability assessment typically selects inputs to maximise the number of bugs found [11]. Finding the ideal scheduling approach for a specific programme and fuzzer remains a research challenge. Input test scheduling is essential to efficiently exploring large or infinite input search spaces. Fortunately, tools like FuzzSim enable quick comparison of input selection methods using data on input performance gathered over numerous process iterations [11].

### Phase 4: Runtime Monitoring

The third stage of fuzzing involves feeding the programme with selected inputs and monitoring the output to find interesting programme states. A particular programme behaviour is present in "interesting programme states" when they are present. A crash typically depicts the behaviour of interest (i.e., the programme fails unexpectedly).

However, it is possible to use any behaviour that can be observed through programme instrumentation to pinpoint intriguing programme states. For instance, Valgrind can spot memory corruption even if it doesn't result in a crash [16]. Heelan also employs fuzzing to find potential programme memory allocators [28]. Interesting behaviours are observable behaviours that are associated with potential flaws or vulnerabilities for vulnerability assessment. A fuzzer needs programme knowledge about how to instrument the programme and what makes for an interesting programme state, as was previously mentioned. This data is frequently provided by a human user, but it is still difficult to define what constitutes an interesting

 DOI : 10.35291/2454-9150.2023.0052 

programme state. What observable behaviours, in the context of vulnerability assessment, are most important for locating flaws or vulnerabilities?

**Phase 5: Input Evaluation**

The fuzzer assesses the performance of inputs in the fourth stage of the fuzzing process. Code coverage is a common fuzzing technique that is used to assess an input's usefulness. If an input prompts the execution of a new section of code, usually a new basic block, it has increased coverage and is rated highly. An input receives a high rating from the libFuzzer tool's data coverage metric if new data values appear at a previously examined comparison in the code. Some fuzzers rate inputs that result in a crash highly in order to find bugs.

**Phase 6: Program states**

The user examines intriguing programme states that the fuzzer outputs after the fuzzing process. This process frequently involves a lot of manual labour and gains from research being done in software engineering-related fields. The user examines each output state to identify the underlying cause of any intriguing behaviour present in that state. Frequently, the user will manually watch as the programme processes the associated input in an effort to isolate the problem. After that, the user decides if the root cause indicates a fresh flaw or vulnerability. Fuzzers frequently produce an excessive amount of intriguing programme states. To determine which intriguing programme states warrant further investigation, the user must perform triage. Output programme states, unfortunately, frequently have the same underlying cause. Even worse, states with the same underlying cause may manifest very differently. For instance, because the effect of the vulnerability is not immediately noticed, a memory corruption vulnerability can cause crashes in numerous different parts of the programme, with wildly different memory images. Triage and root cause analysis continue to be difficult problems for research. Some automated tools try to deduplicate fuzzer outputs [28] or root causes, but these are frequently flawed.

## IV. METHODOLOGY

Machine learning gains new knowledge or abilities by learning from available sample data or experience and automatically optimizing the performance of computer systems themselves. Tasks of machine learning can be categorised into Traditional Machine Learning, Deep Learning and Reinforcement Learning. Traditional Machine Learning is divided into Monitored Learning, Unsupervised Learning, and Semi-Monitored Learning, depending on whether or not input data are labelled and how many data are labelled. Deep Learning is a multi-non-linear artificial neural data display learning network that represents a deeper expansion of the machine-learning algorithms. The original "Feature Engineering" will be replaced on the basis of the method of representative learning and the machine will be able to automatically extract beneficial features from input data. Learning to reinforce is a machine learning branch that describes the problem that agents can maximize feedback or achieve specific goals by learning strategies in interacting with the environment. Contrary to learning from a wide range of input examples, reinforcement training is essentially an automated decision making process. Current techniques of machine learning have been used in the fields of statistical education, Data Mining, Computer Vision and the processing of natural language.

This section, looks at research that fuzzes data using machine learning (ML). ML has been used to create new inputs for the fuzzing process. With fuzzing tools like AFL [34] integrating genetic algorithms (GAs) into the input generation process, unsupervised learning has seen the most successful applications to input generation. Additionally, reinforcement learning (RL) and supervised learning (SL) have both recently been applied to input generation [4, 5, 9, 38]. Additionally, symbolic execution has been used with all three types of ML [10, 40, 42, 6], primarily to speed up constraint equation solve times. For crash triage and root cause categorization, both supervised and unsupervised learning have been used in post-fuzzing processes [15, 33, 39].

A test case can be created by mutating the seed file that is constructed on the basis of known input formats. As a last input the contents of a test case will directly influence whether an error is triggered or not. Therefore, creating a test case with a high level of code coverage or susceptibility-oriented testing can effectively increase the efficiency of detecting vulnerabilities in Fuzzer. The generate inputs stage is where machine learning is most effectively applied to fuzzing. This study goes over the different ways that ML has been used for input generation in this section. This demonstrates how deep learning (DL) and generalised algorithms (GAs) for unsupervised learning have successfully been applied to the generation of input. Although this research tends to be more exploratory, this also covers how RL and supervised learning are applied to various forms of input generation.

Genetic Algorithms, GAs is the most frequently used ML input generation technique [17, 18, 21, 34, 37]. The fundamental input generation algorithms in evolutionary fuzzers are frequently provided by GAs, a class of unsupervised ML inspired by biological evolution. Utilizing a GA involves these 3 main steps: First, create a small base population of inputs, then transform those inputs, and finally, assess how well the transformed inputs

 DOI : 10.35291/2454-9150.2023.0052 

performed. On the inputs that perform the best based on a selected metric, this process is repeated. A set of seed programme inputs make up the base population of inputs for fuzzers. In order to explore the code space and find new paths in the code, the GA will modify these seed inputs. Due to their capacity to expand upon previously successful inputs, GAs have proven effective for input generation. As mentioned in Section 2 and further explained in Section 2, evolutionary fuzzers rank inputs for selection and mutation using a fitness function. The performance of the fuzzer, its capacity to spot specific kinds of bugs, and its propensity to get stuck in local minima can all be significantly impacted by the fitness function that is selected. As a result, selecting a fitness function requires special consideration. While traditional code coverage is frequently used as a fitness function metric, more sophisticated heuristics, like the Dynamic Markov Model (DMM) heuristic, have also been employed [26]. This work uses a Markov process to represent the programme control graph, which is a stochastic process with transition probabilities for each control graph edge. This Markov model and the transition probabilities are used by the DMM heuristic to produce a fitness function. This setup, as opposed to code coverage, enables more precise control of the fuzzer, directing it towards particular sections of the code that might have a bug or flaw. A promising area of research may be the creation of new fitness functions that induce desired fuzzer behaviours because the fitness function that is selected is a significant determinant of fuzzer behaviour. For particular fuzzing objectives, fitness functions that enable finer-grained control of the fuzzer might be more advantageous.

Neural networks and deep learning. In order to enhance generation-based and mutation-based fuzzers, deep learning (DL) and neural networks (NNs) have been applied to input generation. The most popular DL technique used for fuzzing is recurrent neural networks (RNN) [36]. For use in input generation, researchers have focused on Long-Short Term Memory (LSTM) [36] networks, a variation of the RNN.

To create input grammar for PDF files in generation-based fuzzers, Godefroid et al. used LSTMs in one study [24]. LSTMs have demonstrated promise in a variety of tasks involving sequence generation [27]. Godefroid et al. discovered, however, that the objectives of fuzzing and LSTM input grammar learning frequently clash: LSTMs are biassed towards producing well-formed inputs, whereas fuzzers aim to produce model inputs that investigate novel programme states. A sampling strategy was used to select inputs produced by the LSTM in order to get around this paradox, resulting in an input grammar with a healthy mix of well-formed and improperly-formed inputs. The use of ML for input grammar generation is a promising method for boosting code coverage, according to experimental results.

Rajpal et al. integrated DL into AFL in a different study to broaden the coverage of the fuzzer by picking which input bytes to mutate [5]. A heat map indicating the predicted likelihood of increasing code coverage when any specific byte is mutated was created using a NN. The following four sequence learning architectures were compared to produce the heatmap. The four most common LSTMs are the standard LSTM, a bidirectional LSTM [36], Seq2Seq [14], which transforms a sequence into another sequence, and a variant of Seq2Seq [3] that uses an attention mechanism to concentrate on the most crucial portions of an input. Each model improved code coverage in some circumstances, but the standard LSTM model performed slightly better than the others overall.

According to experimental results, standard AFL performed better on the PNG format than DL-augmented AFL did on the ELF, XML, and PDF formats. Modelling programme behaviour is an alternative method of generating inputs, and the most promising inputs are then chosen using the model. The behaviour of a programme was modelled using the NEUZZ method as a smooth continuous function using a shallow NN [38]. Based on a seed input, the NN was trained to predict the program's branching behaviour. It was discovered experimentally that the gradients generated during training, particularly the larger gradients, were helpful in pinpointing which input bytes control branching behaviour. This knowledge allowed NEUZZ to experimentally outperform common fuzzers like AFL and Angora on some programmes by directing fuzzer mutations. The Angora fuzzing tool was used by Chen et al. to model programme behaviour as well [12]. To depict the route from a program's starting point to a given branch constraint, Angora used a discrete function. The function representation was then used to implement gradient descent over brief discrete intervals in order to identify a set of inputs that satisfy the constraint and advance the programme to that particular branch. The ability to efficiently resolve branch constraints and experimentally outperform AFL and symbolic execution on some programmes was demonstrated by this method.

Cheng et al. [13] implemented another alternative strategy for input generation using DL. In this method, new programme paths were predicted using RNNs. Following that, these paths were fed into a Seq2Seq model, which created fresh seed inputs for the fuzzer that was intended to follow the predicted paths. The generated corpus increased fuzzer code coverage for the PDF, PNG, and TFF formats, according to this preliminary study's experimental results.

Although DL and NN applications for input generation

 DOI : 10.35291/2454-9150.2023.0052 

show promise, there are still obstacles in the way of more widespread use. First off, since DL models need a lot of computation time to train, it is unlikely that training can be practically incorporated into the fuzzing process. Training individual models for individual programmes could be an alternative, but this is also probably not practical. Many of these studies made an effort to address these problems. For instance, Rajpal et al. reduce training costs by training the NN on a limited set of inputs prior to the start of fuzzing [5]. She et al., on the other hand, only used retraining with the most beneficial data points, which resulted in a more limited form of model re-training throughout the fuzzing process [38]. Both of these techniques shorten training times, but it is unclear how they will ultimately affect fuzzing performance. The development of techniques to transfer previously trained models between programmes could be an alternative to reducing the size of the data. . Although it is unclear how these methods may affect performance, they may eliminate or drastically reduce the amount of model training necessary for fuzzing previously unexplored programmes. Performance consistency across file formats is a second hindrance for DL applications to fuzzing. Numerous of these studies showed that while other file formats could not compete with state-of-the-art methods, DL consistently improved fuzzing performance for some file formats, such as PDF. Future studies may be helpful to comprehend the suitability of DL and NNs for particular file formats.

Reinforcement learning (RL) has been used by two groups for input generation [29], [2]. By altering network packets sent to a host, Becker et al. used the SARSA algorithm [29] to fuzz the IPv6 protocol [4]. The SARSA algorithm determines the agent's optimal behaviour by taking into account both its current state and its actions. In order to learn a grammar describing inputs for generation-based fuzzers on the PDF format [9], Bottinger et al. used a deep Q-learning network [42]. To translate states into actions, the deep Q-learning network used a deep neural network. These studies provide crucial information for RL input generation applications. They first demonstrate the importance of programme representation in the training of successful RL agents. The IPv6 protocol was represented by Becker et al. using a finite state machine, where each state represented the host's current response to a specific packet and transitions between states represented potential packet mutations. The Markov decision process was another type of finite state machine that Bottinger et al. used to represent problems. Stochastic transitions between states are a characteristic of Markov Decision Processes. In this study, the transitions represented probabilistic rewrite rules for the seed input at each state, while the states represented a specific seed input for the fuzzer. Becker and Bottinger, therefore, Second, they shed light on how to define an agent's reward function, which is frequently the most difficult part of RL. Becker and colleagues developed a multi-part reward function using the following standards: amount of programme functions that are called from a single input, the existence of an error, and the potential for message corruption or delay in the program's response [4]. The agent received the clearest indication that it had entered an intriguing region of the code space when the error appeared. Even in the absence of the error signal, the programme response was used to direct the agent. As a result, each component of the reward function had a specific function in directing the agent, and omitting any one of these criteria might have led to a less efficient agent. Bottinger et al. experimented with three different reward functions: one that used execution time, another that used code coverage, and a third that combined the two. The reward function affected the fuzzer's investigation of the input space in both studies. For instance, the agent learned inputs that made the programme terminate quickly when Bottinger used execution time as a reward. The research by Becker and Bottinger suggests that reward functions should be carefully defined, taking into account the type of software programme, the kinds of bugs that need to be found, the fuzzing metrics at hand, and the ultimate fuzzing goals. Traditional fuzzers don't yet support RL, and applications are still hypothetical. A deeper comprehension of reward functions is an important first step in this direction. More specifically, it is not yet clear whether there are reward functions that are consistently effective or how programme dependent a reward function should be.

Machine learning used for symbolic execution can aid in producing useful new inputs for fuzzers, but computational cost and path explosion continue to be major obstacles. The viability of enhancing constraint solving, which could support symbolic execution for fuzzer input generation, is the subject of numerous ML research projects. Unfortunately, current attempts to use ML are merely feasibility studies and do not match state-of-the-art methods using graph algorithms [31].

Supervised Learning is used to resolve equations involving constraints. To determine whether constraint equations had valid solutions or not, graph neural networks were employed in one study [10]. In a different study, Wu combined Monte Carlo techniques with logistic regression to find the initial values that increased the likelihood of discovering a workable solution to a constraint equation. Using these initial, promising values, the constraint equation was solved using the Monte Carlo methods, and the validity of the solution was shown by logistic regression. The Minisat solver ran faster after incorporating these new initial values [6]. Another study

 DOI : 10.35291/2454-9150.2023.0052 

[42] trained LSTMs (i.e., DL) to resolve constraint equations. The LSTMs were able to solve constraint equations from domains they had not been trained on, showing that the DL models can generalise even though they could not outperform cutting-edge constraint solvers. Another technique created by Shiqi et al. represented constraint equations using NNs [30]. Gradient descent was then used to find the answers to these constraint equations. In general, these studies each present distinct approaches to solving constraint equations. Despite not being state-of-the-art, they still provide a solid foundation for using supervised learning to reduce the amount of time needed to solve constraint equations.

RL was used by Mairy et al. to enhance neighbourhood-specific search techniques [40]. Local neighbourhood search techniques find answers to constraint equations iteratively by determining answers to different subsets of the constraint equation and combining the subsets to create the final solution. These local neighbourhood search methods need to carefully consider the space of potential subsets in order to find useful subsets. In an effort to speed up the process of finding a valid solution, the RL agent was directed to select subsets that were more likely to do so.

Instead of directly solving constraint equations, one research project used machine learning to condense the size of the solution search space. Li et al. attempted to decrease the number of infeasible paths, i.e., paths that can never be reached because of conflicting constraints, by redefining the typical collection of path constraints as an optimization problem [35]. They solve the optimization problems using the ML technique RACOS [1], an optimization technique that scales well to high dimensional problems; however, only very small programmes up to 335 lines were analysed. Initial experiments with ML to enhance symbolic execution appear promising, but it is still unclear how useful ML will actually be in this situation. Modern research is typically limited to minor issues that don't challenge cutting-edge methods. Additionally, the combined fuzzer and symbolic execution techniques described in Section 2 have not yet had this research put to use. If ML can help improve these combined techniques is an open question.

## V. CONCLUSION

This study talks about fuzzing applications of machine learning (ML) in this survey. Supervised learning is rarely used to support the fuzzing process because unsupervised and reinforcement learning techniques are more naturally suited to fuzzing problems. The Generate Inputs phase of the fuzzing process is where ML is most frequently used to support it. As a result of tools like AFL incorporating unsupervised algorithms into their workflow, unsupervised methods currently offer the greatest advantages. Deep learning and reinforcement learning, on the other hand, are not yet widely used but are being researched for potential advancements. Additionally, ML has been used to post-fuzz Interesting Program States, which aids in triaging crashes and supports root cause analysis. The Select Inputs stage of the fuzzing process has not, however, been subject to the application of ML, perhaps because it is not a significant bottleneck. Additionally, post-fuzzing reproducibility of crashes has not been evaluated using ML. Lack of training data, difficulty transferring models between programmes, and computationally intractable training times are just a few of the reasons why ML is likely absent from some parts of the fuzzing process.

Although ML has been crucial to the operation of fuzzing systems, there are still many unresolved research issues. Recently, a large number of researchers have started allocating funds to tackle some of these problems. Future vulnerability assessments of programmes will still place a high priority on fuzzing. This research anticipate that ML will continue to be used to address bottlenecks in the fuzzing process as this field of study develops.

## REFERENCES

[1] Yang Yu, Hong Qian, and Yi-Qi Hu. 2016. Derivative-Free Optimization via Classification (Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence).

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A Brief Survey of Deep Reinforcement Learning. CoRR abs/1708.05866 (2017). arXiv:1708.05866 https: //arxiv.org/abs/1708.05866

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. CoRR abs/1409.0473 (2014). arXiv:1409.0473 http://arxiv.org/abs/1409.0473

[4] Sheila Becker, Humberto Abdelnur, Radu State, and Thomas Engel. 2010. An Autonomic Testing Framework for IPv6 Configuration Protocols. In Mechanisms for Autonomous Management of Networks and Services, Burkhard Stiller and Filip De Turck (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 65–76.

[5] William Blum, Mohit Rajpal, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. (November 2017). https://www.microsoft.com/en-us/research/publication/ not-all-bytes-are-equal-neural-byte-sieve-for-fuzzing/

[6] Haoze Wu. 2017. Improve SAT-solving with Machine Learning. CoRR abs/1710.11204 (2017). arXiv:1710.11204 http://arxiv.org/abs/1710. 11204

[7] 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 (CCS '17). ACM, New York, NY, USA, 2329–2344. https: //doi.org/10.1145/3133956.3134020

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In 23rd ACM Conference on Computer and Communications Security (CCS).

[9] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. [n. d.]. Deep Reinforcement Fuzzing. ([n. d.]). arXiv:1801.04589 http://arxiv. org/abs/1801.04589

[10] Benedikt Bünz and Matthew Lamm. 2017. Graph Neural Networks and Boolean Satisfiability. CoRR abs/1702.03592 (2017). arXiv:1702.03592 http://arxiv.org/abs/1702.03592

 DOI : 10.35291/2454-9150.2023.0052 

[11] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS '13). ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/ 2508859.2516736

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In 2018 IEEE Symposium on Security and Privacy (SP). 711–725. https://doi.org/10.1109/SP.2018.00046

[13] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. 2019. Optimizing seed inputs in fuzzing with machine learning. CoRR abs/1902.02538 (2019). arXiv:1902.02538 http://arxiv. org/abs/1902.02538

[14] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. CoRR abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[15] Yingnong Dang, Rongxin wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. (06 2012), 1084–1093.

[16] Valgrind. 2017. Valgrind tool. http://valgrind.org

[17] Jared D. DeMott, Richard J. Enbody, and William F. Punch. 2007. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. (2007).

[18] Fabien Duchene. 2013. Fuzz in the Dark: Genetic Algorithm for BlackBox Fuzzing. In Black Hat 2013. Black Hat, Sao Paulo, Brazil. https: //hal.inria.fr/hal-00978844

[19] Joe W. Duran and Simeon Ntafos. 1981. A Report on Random Testing. In Proceedings of the 5th International Conference on Software Engineering (ICSE '81). IEEE Press, Piscataway, NJ, USA, 179–183. http://dl.acm. org/citation.cfm?id=800078.802530

[20] Joe W. Duran and Simeon C. Ntafos. 1984. An Evaluation of Random Testing. IEEE Trans. Softw. Eng. 10, 4 (July 1984), 438–444. https: //doi.org/10.1109/TSE.1984.5010257

[21] James Fell. 2017. A Review of Fuzzing Tools and Methods. Technical Report. https://dl.packetstormsecurity.net/papers/general/a-review-offuzzing-tools-and-methods.pdf

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In Advances in Neural Information Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112. http://papers.nips.cc/paper/ 5346-sequence-to-sequence-learning-with-neural-networks.pdf

[23] Ari Takanen, Jared Demott, and Charlie Miller. 2008. Fuzzing for Software Security Testing and Quality Assurance. Artech House, Inc. http://index-of.co.uk/Reversing-Exploiting/Fuzzing.pdf.

[24] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn & Fuzz: Machine Learning for Input Fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 50–59. http://dl.acm.org/ citation.cfm?id=3155562.3155573

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. Deep Learning. MIT Press. http://www.deeplearningbook.org.

[26] Sherri Sparks, Shawn Embleton, Ryan K Cunningham, and Cliff Changchun Zou. 2007. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). 477–486. https://doi.org/10.1109/ACSAC.2007.27

[27] Alex Graves. 2013. Generating Sequences With Recurrent Neural Networks. CoRR abs/1308.0850 (2013). arXiv:1308.0850 http://arxiv. org/abs/1308.0850

[28] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/ presentation/heelan

[29] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. CoRR abs/cs/9605103 (1996). arXiv:cs/9605103 https://arxiv.org/abs/cs/9605103

[30] Shen Shiqi, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In NDSS Symposium 2019. https://doi.org/10.14722/ndss.2019.23530

[31] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. 2010. Tackling the Path Explosion Problem in Symbolic ExecutionDriven Test Generation for Programs. In Proceedings of the 2010 19th IEEE Asian Test Symposium (ATS '10). IEEE Computer Society, Washington, DC, USA, 59–64. https://doi.org/10.1109/ATS.2010.19

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/ 4824-imagenet-classification-with-deep-convolutional-neural-networks. Pdf

[33] H. Lal and G. Pahwa. 2017. Root cause analysis of software bugs using machine learning techniques. In 2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence. 105–111.

[34] lcamtuf. 2014. afl-fuzz: crash exploration mode. https://lcamtuf. blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html blog

[35] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016). ACM, New York, NY, USA, 554–559. https://doi.org/10.1145/2970276.2970364

[36] Zachary C. Lipton, John Berkowitz, and Charles Elkan. 2015. A Critical Review of Recurrent Neural Networks for Sequence Learning. CoRR abs/1506.00019 (2015). arXiv:1506.00019 http://arxiv.org/abs/1506. 00019

[37] Guang-Hong Liu, Gang Wu, Zheng Tao, Jian-Mei Shuai, and ZhuoChun Tang. 2008. Vulnerability Analysis for X86 Executables Using Genetic Algorithm and Fuzzing. In 2008 Third International Conference on Convergence and Hybrid Information Technology, Vol. 2. 491–497. https://doi.org/10.1109/ICCIT.2008.9

[38] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2018. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. CoRR abs/1807.05620 (2018). arXiv:1807.05620 http://arxiv.org/abs/1807.05620

[39] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. SIGPLAN Not. 51, 1 (Jan. 2016), 298–312. https: //doi.org/10.1145/2914770.2837617

[40] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. 2011. Reinforced Adaptive Large Neighborhood Search. Doctoral Program at the Interational Conference on Principles and Practice of Constraint Programming, 55–60.

[41] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. Commun. ACM 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[42] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2018. Learning a SAT Solver from SingleBit Supervision. CoRR abs/1802.03685 (2018). arXiv:1802.03685 http: //arxiv.org/abs/1802.03685

 DOI : 10.35291/2454-9150.2023.0052 