Machine Learning for Fuzzing: State of Art

Author: Pablo Barranca Fenollar
Tutor: Enric Hernández Jiménez

June 2, 2020

Universitat Oberta de Catalunya

Contents

1 Introduction

Machine learning has become more and more popular in recent years. This popularization has been stimulated by multiple factors: large and affordable computational power, new powerful algorithms, new tools that make it easy to use machine learning algorithms, availability of big data to train the models, etc. Many disciplines have experienced significant changes thanks to its adoption.

The fuzzing field has not been an exception. Many researchers have proposed applying machine learning algorithms to the various stages of the fuzzing process. Most studies seem to have brought improvements to the task, however it is not always clear at what cost. Moreover, the reasons behind the selection of one algorithm instead of another are not clear in much of the published literature.

This master thesis not only presents the benefits and disadvantages of using various machine learning algorithms in each fuzzing stage, but also identifies new promising paths that researchers should take.

1.1 Origins of fuzzing

Fuzzing is a technique to discover vulnerabilities that uses invalid data as input for identifying unexpected behaviors in software. The fuzzing method was proposed for the first time by Miller et al. [1990] and has evolved through the years, reaching better and better results.

In his work, Miller et al. [1990] analyze how some specific UNIX programs behave when receiving unusual input streams. The authors identify cases where the programs terminate abnormally or fall in infinite loops. They structure the fuzzing process in four stages:

1. Random characters generation

2. Interaction with the software application under test
3. Crash detection
4. Identification of the cause of the program crash

As a result of the tests, Miller et al. [1990] detect weaknesses in the implementation of several UNIX tools. Since their results were published, three strategies have been designed for improving the fuzzing process:

- Using dictionaries with test cases that have been successful in other tests
- Generating test cases following the specification a given technology
- Mutating valid samples

Classic approaches use heuristics to optimize this process, for example to define if it is worth testing a given element or if it is better to switch to the next one. Heuristics are still doing a great job nowadays, but the use of machine learning algorithms seems to be changing the rules of the game. Their capacity to learn from datasets and to adapt and solve complex problems makes machine learning algorithms the ideal solution to the challenges of fuzzing.

2 Fuzzing

Section 2.1 presents the types of fuzzing techniques. Section 2.2 describes the elements composing a fuzzing process. Section 2.3 talks about test-case generations strategies. Finally, Section 2.4 summarizes the fuzzing software used.

2.1 Fuzzing techniques

When performing fuzzing tests, three different approaches can be followed: white-box, black-box and grey-box. Choosing which of them has to be used will depend on different factors, for example: the availability of the source code, the protocol specification or the possibility of analyzing the outcome of each test. In this section all three are reviewed and their weaknesses and strengths described.

2.1.1 White-box

The white-box tests use the source code of the application under test. Adapting the fuzzing process to the requirements of a specific application is highly time-consuming, but results in higher coverage ratios. If performed properly, a white-box fuzzing tests should cover all the sections of software that can be reached through the defined inputs. This means that ideally, all the inputs will be tested and all the code branches will be covered. This two aspects can be used as part of the metrics that evaluate the quality of the process.

White-box fuzzing can be used as part of the development cycle, and so it can be adapted to fuzzing-specific software.

2.1.2 Black-box

The black-box technique consists in generating test cases for the targeted software without having knowledge of its structure or functionalities. Their strength resides on the speed. Additionally, black-box fuzzers do not require neither human intervention nor the analysis of the software or the protocol specification.

Google [2020a] describes the situations where this strategy should be used:

- *The target is large*: White-box and grey-box fuzzers may fail against large targets since the necessary time for covering all the elements can be unacceptable.

- *The target is not deterministic for the same input*: analyzing the software behavior will not be possible, grey-box fuzzing should be discarded in this case.
- *The target is slow*: running all the tests on slow targets may increase the time-costs in excess, random approaches are usually more effective in these cases.
- *The input format is complicated or highly structured (e.g. a programming language such as JavaScript)*: generating new test cases without providing its specification produces many non-valid test cases. When this happens, the generated test cases do not bring improvements if compared with a random approach.

2.1.3 Grey-box

Also known as coverage-guided fuzzers, grey-box fuzzers are capable of gathering and analyzing information from the tested software and adapt their own behavior. The approach differs from the black-box approach because in this strategy the specification of the protocol or the format are used for generating the test cases. By using this information, the amount of non-relevant samples drops dramatically, improving the speed of the fuzzing process.

Some grey-box fuzzers do not require much information about the target but include features for the fuzzer to learn about the target. Their strategy consists of investing an arbitrary amount of time on learning from the target for defining valid test-cases. This kind of fuzzers are the most common nowadays thanks to their efficiency.

2.2 The elements of the fuzzing process

Although some authors propose a less detailed classification of the elements that compose the fuzzing process, here a full list is provided. The proposed list is based on Wang et al. [2019], Miller et al. [1990]:

1. Interfaces identification: detect where inputs can be sent (e.g. command-line arguments that are processed by an application)

2. Input generation

    - Seed file generation: This is an essential component of the fuzzers, the capacity of producing good seeds affects significantly the results of the fuzzing process. While this generation can be done with a random generator, it is quite common that an algorithm drives this process. Classically heuristics have been used on this process and more recently machine learning algorithms have also been tested.

    - Test case generation: This task consists on using the seeds, generated by the previous component, for generating valid test cases. This can be done in two different ways:

        - Mutation based: the seed file for creating the test cases. If the source-code/specification of the tested element (application, protocol, etc.) are not available, this approach is the most feasible.

        - Generation-based: this approach requires understanding the protocol or the format expected by the application under test. When the source-code/specification is available, this strategy is feasible. As will be described in this thesis, understanding the specification from some valid samples is possible by training certain machine learning models.

- Test case filter: identify the test cases with more chances to trigger significant results. Commonly, fuzzers maintain a queue of useful test seeds, when a test case produce relevant results, this is included in the seeds list. The intervention of the test case filter consists on discarding those with low probabilities of producing exceptions or increase the path coverage. By doing this, the number number of tests can be reduced.
- Mutation operator selection: Test cases are mutated for triggering unexpected behaviors on the tested software. This is done by applying different operations to the bits which compose the test case. Some examples of these operations are: flipping bits, setting values to null, apply subtractions or additions to the values, cloning data in different places of the test case, etc.
- Fitness function: After all the previous steps have been executed a bunch of test cases are created. Although all the actions done, some of the generated elements will not produce relevant results in terms of crashes and code coverage. Fitness functions have the objective of detecting the cases that can be filtered out of the fuzzing tests. The fitness function uses the results of the tests once they have been completed and acts on the elements before they are served to the evaluation module. Since this activity has an important impact on the performance of the entire fuzzing process, usually the analysis by fitness functions is done only on a small percentage of the evaluated test cases. A common approach consists on estimating the quality of the final candidate case, this process indicates with a numerical value how close the values are to the optimal value, being '0' the desired result and increasing its value when the results are far from it Zeller Andreas et al. [2019].

3. Sending Inputs: provide the generated test-case to the analyzed piece of software. This can include simple like request-response interactions

but also more complex cases including multi-step tests or requiring adaptability to bypass security measures avoiding automated interactions.

4. Monitoring and evaluation: also known as Exploitability Analysis. This process should detect which of the outcomes are relevant. The analysis is done on the generated outputs (e.g responses, logs, dumps, etc.). For this to be done, the following aspects are considered:

    - Availability issues: non expected inputs may produce the tested software crashing or experiencing unusual delays for generating the responses (if there are).
    - Affectation of the file-system: mutations in the file-system of the target may reveal unexpected behaviors on the asset.

2.3 Test-case generation strategies

Fuzzing processes require generating data for testing the targeted software. Takanen et al. [2008] identifies four different ways of creating this data, also known as test cases. The strategies are *cyclic, random, or library*-based.

Table 1: Fuzzing strategies

Type	Description	Resources invested
Predefined cases	List of test cases defined for an specific software or protocol. Only the defined values are used.	Limited to the list length
Cyclic	Iterative testing, the values are modified on each iteration following a pattern.	Limited to the initial list and defined modifications
Random	Random testing, the values are randomly mutated, each variable is tested against all the values	Only limited by the time frame dedicated to the tests
Library	List of values that have been proven effective, each variable is tested against all the values	Limited to the product of the number of variables times the elements in the dictionary

The described types of strategies are usually mixed for obtaining greater results. The fuzzing process is limited by the resources boundaries, time and computing power, so finding shortcuts is part of this science. Many fuzzers include algorithms trying to improve the results (e.g. choosing the parameters with more chances to produce a relevant result, defining the order in which the data is provided).

As part of the techniques for improving the creation of test cases, machine learning has been used lately for this purpose. The use of this technology has brought great improvements on the fuzzing results and is the trend of the latest research. However using machine learning for this purpose implies solving new challenges. While the use of Machine Learning increases the success ratio (percentage of new paths or crashes per number of tests), it also adds computational costs, and consequently slows down the speed of the process. Almost all of the papers reviewed in the present document point to the challenge of balancing the costs of the learning process with its benefits compared to fully randomized approaches.

When using any of the strategies defined in the table 1 and regardless of whether machine learning is used or not, the generation of valid fuzzing test cases is always a challenge. Godefroid et al. [2017] exposes the complexity of learning how to generate a valid test case (well-formed input) but at the same time include elements breaking this structure (not well-formed inputs).

2.4 Fuzzing software

Most of the reviewed papers publications use the American Fuzzy Lop for their tests. This is because the great performance it provides and the the ease to integrate its approach with the different stages of the fuzzing process. Other two fuzzers have been also used, LibFuzzer and the Microsoft *test driver* included in the Microsoft Security Risk Detection testing service. Only one of them has implemented an ad hoc solution depending of another Microsoft tool (Intel's instrumentation tool Pin) for recording the execution sequences.

2.4.1 LibFuzzer

This fuzzer focuses on testing the software libraries. It is designed to be run against Linux targets but it is also possible using it on Windows under some functional limitations. It has some documented limitations including, for example issues when run against large targets (many inputs) or when the libraries have not been designed for supporting the fuzzer interface.

While it is possible to include personalized mutation operators, LibFuzzer comes with a set of mutation operators:

- PersAutoDict (AddWord From Persistent AutoDictionary Count)

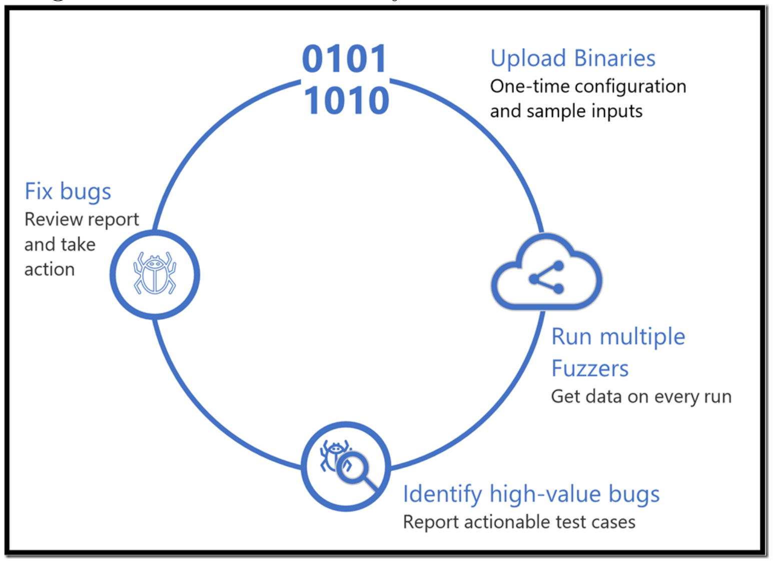
- CMP (AddWord From TORC Count)
- ChangeAsciiInt (Change ASCII Integer Count)
- ChangeBinInt (Change Binary Integer Count)
- ChangeBit (Change Bit Count)
- ChangeByte (Copy Part Count)
- CopyPart (Change Byte Count)
- CrossOver (Cross Over Count)
- CustomCrossOver (Custom CrossOver Count)
- CustomMutation (Custom Mutation Count)
- EraseBytes (Erase Bytes Count)
- InsertByte (Insert Byte Count)
- InsertRepeatedBytes (Insert Repeated Bytes Count)
- ShuffleBytes (Shuffle Bytes Count)

LibFuzzer is integrated in the Clang compiler, during the process, the mutation operators above-indicated are combined randomly for increasing the coverage of the tested software.

2.4.2 Microsoft Security Risk Detection

Although there is no documentation about the insights of this fuzzing technology, the descriptions offered by Microsoft indicate that multiple fuzzers are used in this process. As per now, the Microsof Security Risk Detection is not offered on premise and requires the developers uploading the software pieces to the Microsoft infrastructure.

Figure 1: Microsoft Security Risk Detection workflow


0101
1010
Upload Binaries
One-time configuration and sample inputs
Run multiple Fuzzers
Get data on every run
Identify high-value bugs
Report actionable test cases
Fix bugs
Review report and take action

Microsoft [2020]

It is very likely that this tool is grounded on the work presented in Godefroid et al. [2008b]. The reason behind this affirmation resides on the fact that is the only publication from Microsoft on whitebox fuzzing tests (like the Microsoft Security Risk Detection service).

> *Our approach records an actual run of the program under test on a well-formed input, symbolically evaluates the recorded trace, and gathers constraints on inputs capturing how the program uses these. The collected constraints are then negated one by one and solved with a constraint solver, producing new inputs that exercise different control paths in the program. This process is repeated with the help of a code-coverage maximizing heuristic designed to find defects as fast as possible.*
>
> A reader interested in diving into this thematic could also read Godefroid et al. [2012]

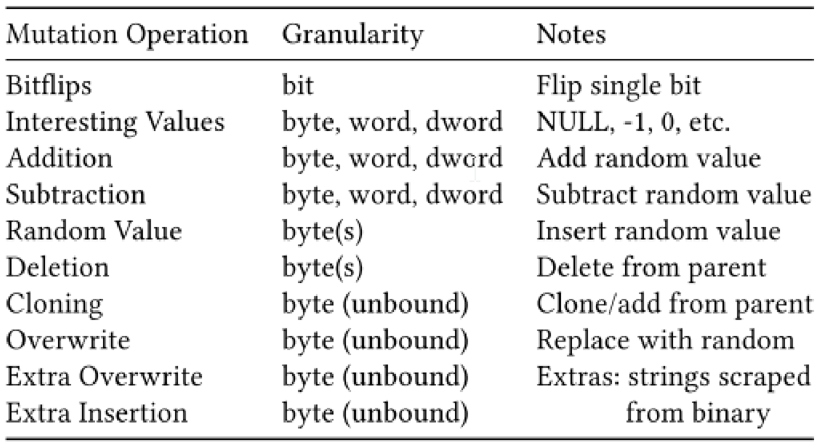
2.4.3 American Fuzzy Lop

AFL is the most popular fuzzing software. It is based on a genetic algorithm for understanding the file semantics necessary for generating test cases. This fuzzer integrates multiple features:

- crash explorer
- test case minimizer
- fault-triggering allocator
- syntax analyzer

It also comes with a set of default mutation operators:

Figure 2: Adaptive grey-box fuzz-testing with thompson sampling - List of AFL mutation operators



Mutation Operation	Granularity	Notes
Bitflips	bit	Flip single bit
Interesting Values	byte, word, dword	NULL, -1, 0, etc.
Addition	byte, word, dword	Add random value
Subtraction	byte, word, dword	Subtract random value
Random Value	byte(s)	Insert random value
Deletion	byte(s)	Delete from parent
Cloning	byte (unbound)	Clone/add from parent
Overwrite	byte (unbound)	Replace with random
Extra Overwrite	byte (unbound)	Extras: strings scraped
Extra Insertion	byte (unbound)	from binary

Karamcheti et al. [2018a]

The main reason why AFL has gained popularity among the researchers is the quality of the results. This has been possible thanks to the default features included in AFL but also due to the ease of including modifications in it and the integration with other tools, for example the ClusterFuzz project (https://google.github.io/clusterfuzz/) from Google integrates AFL with LibFuzzer for greater results.

Currently, AFL is presented in multiple flavors:

Table 2: AFL flavors

AFL Fuzzer	Linux applications
Win AFL	Windows cations
TriforceAFL	Linux applications, blackbox approach
AFLGo	Linux applications, targets only specific sections of the software.
Shellphish Fuzzer	Linux applications, Python interface for AFL

3 Machine Learning Algorithms

Throughout this work, various machine learning algorithms are referenced. This section provides a short description for each of them.

3.1 Recurrent Neural Networks

They are a class of artificial neural network. As described in Fan and Chang [2018], these neural networks are the attempt to solve the issues of the classical neural networks by keeping the status information among the loops. This allows the model to grow based on the results of the previous iterations.

Specifically, Recurrent Neural Networks generate results in base to the received input at the time *T* and the generated output at *T-1*. Since only the status of the previous step affects the result of the current cycle, the algorithm has difficulties on keeping the information over the cycles. This is related with the gradient descent algorithm Ruder [2016] that is used by the recurrent neural network.

This algorithm uses supervised learning. A dataset for training this model has to be composed by samples and evaluations of the samples. In fuzzing this usually means executing a test case and use this information together with the evaluation result for training the neural network.

Fan and Chang [2018] exposes that while in theory this algorithm can memorize long-term dependencies, *in practice, RNNs become unable to learn to connect the information in cases where the distance between the relevant information and the place that it is needed becomes very large.* This aspect is solved in the Long Short Term Memory algorithm which will be described later.

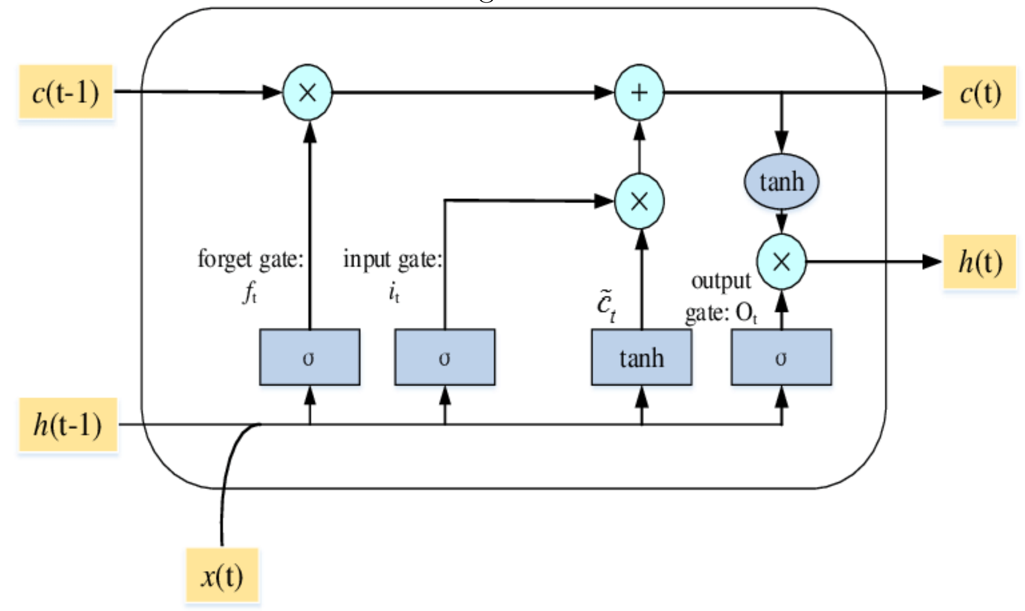
3.2 Long Short-Term Memory

LSTM is a type of Recurrent Neural Network algorithm. It is designed for avoiding the long-term dependency problem Fan and Chang [2018]. Differently from the Recurrent Neural Networks, the LSTM has a *state cell* where it is possible to add information that it is important to keep the information among the different cycles or remove undesired information.

As described in Yuan et al. [2020], the LSTM network is composed by three gates o controller functions:

- Forget gate: determines which information is not relevant, the result is used for updating the state cell
- Input/update gate: determines which information has to be kept, the result is used for updating the state cell
- Output gate: calculates the hidden state. Provides a filtered version of the state cell.

Figure 3:



Yuan et al. [2020]

The algorithm uses multiple networks like the one described above. The state cell is shared among all the nodes and this gives the network the capacity of keeping the information among the cycles. By training the cells forget and update it is possible to determine which information is important to be kept and which is not.

The use of the state cell ensures that long samples do not influence the model to the detriment of short ones. Using this algorithm is it useful in fuzzing processes because test cases usually have variable length.

3.3 Generative Adversarial Networks

Generative Adversarial Networks are a type of Neural Networks that are used for unsupervised learning. They capture variations among the samples of a given dataset. From those variations they are able copy those for generating new elements.

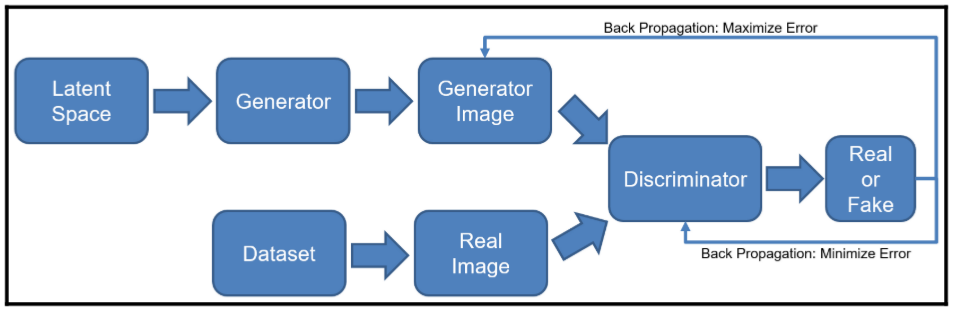
They were introduced by Goodfellow et al. [2014] with the objective of reducing the computational costs of previous proposals including the Markov decision chain Bellman [1977].

Generative Adversarial Networks are composed by two main elements: a generator (of samples) and a discriminator (a samples analyzer) who evaluates the quality of the generated samples. Both elements are neural networks.

This algorithm is useful when there is not enough data to train a model (e.g. fuzzing processes with limited samples amount).

Kalin [2019] presents an example of these networks for generating images. While the generator produces image samples, the discriminator evaluates if its a true image or not by comparing it with a dateset of real images. While the generator tries to generate completely new images, the discriminator ensures that those look like real ones.

Figure 4: Architecture diagram updated to show the backpropagation step in training the GAN model



Kalin [2019]

The competition between bot elements makes possible generating new and valid samples from a very small dataset of original samples.

3.4 Logistical regression

Also known as logit model it is a nonlinear regression model that can be used, among other possible uses, to classify observations according to their characteristics. The model is able to divide the classifications in two categories.

This method is taken from the classic statistics methods. It allows not only classifying data but also identify if the samples are useful for the predictions (Wald test). For doing this, the variable's effect on the binary outcome is calculated and discarded if it is not statistically different from zero. It is a supervised learning algorithm, this means that the learning process is done by providing tagged data to the model.

This algorithm is not useful in fuzzing stages where generating new data is necessary but it successfully improves classifying test cases as shown in Karamcheti et al. [2018b] and it is very likely that can improve the mutation operators selection stage.

3.5 Reinforced Learning

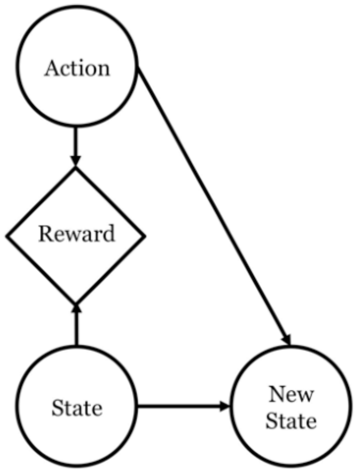
There are three paradigms in automated learning: supervised, non-supervised and reinforced learning. The reinforced learning is a technique that uses multiple and autonomous agents which are capable of choosing actions by interacting with the environment.

For building Reinforced Learning models, Markov Decision processes are used. This process is composed by the following elements:

- Environment
- Agent
- States
- Actions

- Rewards

Figure 5: Markov Decision Process (MDP)



Reid et al. [2014]

The reinforced learning algorithm consists on selecting actions from a given state, which produce changes in the environment and bring it to a new state. The model learns by rewarding the agent when those actions produce the desired final state of the environment. The learning process is then the product of a *trial-and-error interactions with a dynamic environment* Kaelbling et al. [1996].

Drozd and Wagner [2018] exposes the benefits obtained when selecting mutation operators in fuzzing processes. The rewards are a key factor for the success of this algorithm. It may be interesting to use it in for exploitability analysis and test case filter in fuzzing processes.

3.6 Naive Bayes

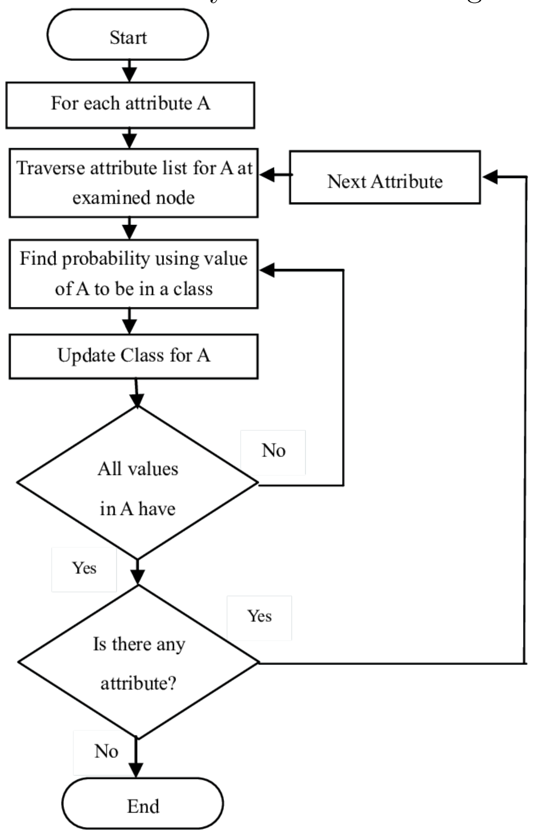
A probabilistic classifier based on the Bayes theorem, a multivariable statistical method. It is an supervised-learning algorithm used for classification.

It is capable to process large samples without having impacts on the results quality. The model is trained with labeled data with the objective of identifying characteristics in the samples that can be used for classifying them. When the training process has been complete, the model is able to assign the probability of given classification criteria inside each evaluated sample.

The principal factor to be taken in consideration when using Naive Bayes lies on the high amounts of samples that are needed to train the model. In other words, the predictions may not be accurate when we have very little labeled data.

The classification is done through the decision tree described in the next diagram:

Figure 6: Naïve Bayes decision tree algorithm.



Karim and Rahman [2013]

While the standard use of this algorithm consists on training the model with labeled data and then use it for classification, it is possible to improve the accuracy if the classification results can be evaluated.

This algorithm has been proven effective in fuzzing processes for classifying samples in the exploitability analysis stage. It is very likely that other stages, specially the test case filter, can be improved with this technique.

4 Literature Review

This section contains the analysis of several papers that use machine learning in fuzzing. These papers have been grouped into six subsections, each section representing the stage of the fuzzing process they seek to improve. Apart from summarizing the relevant papers, each subsection also includes observations about the strategies followed by the researchers. These observations include identifying the strengths and weaknesses of the proposed method. Additionally, for each stage, gaps in the literature are identified and proposals for future research are included.

4.1 Seed generation

Seed generation is one of the main features that most of the fuzzers include. The ability on generating high quality seeds has a big impact on the fuzzing results. Identifying the quality of these seeds is a duty where machine learning techniques have a significant impact.

There are mainly two approaches to this problem: the first consists of discarding seeds with low chances of producing significant results, the second one on supporting the process by generating new seeds of greater quality. These strategies can also be combined Cheng et al. [2019].

When creating machine-learning-based solutions the memory of the network becomes a critical factor for seed generation. Using classic Recurrent Neural Networks (RNN) for generating seeds brings some difficulties when generating seeds, these are related with the memory of the network impact over each of the predictions. While Cheng et al. [2019] solves this problem by adding simplification functions, using LSTM solves these issues Nichols et al. [2017].

Although neural networks perform well, the results obtained with Generative Adversarial Networks Nichols et al. [2017] are of higher quality. However, these results should be taken cautiously since Nichols et al. [2017]only test one binary.

Personally, I consider that LSTM neural networks with different tunes can be the a valid choice for future research.

4.1.1 Optimizing seed inputs in fuzzing with machine learning

The research by Cheng et al. [2019] presents a framework for generating seeds that improve the coverage of the targeted software. The generation is supported by a Recurrent Neural Networks (RNN) machine learning algorithm. The tests are performed on the American Fuzzy Lop fuzzer.

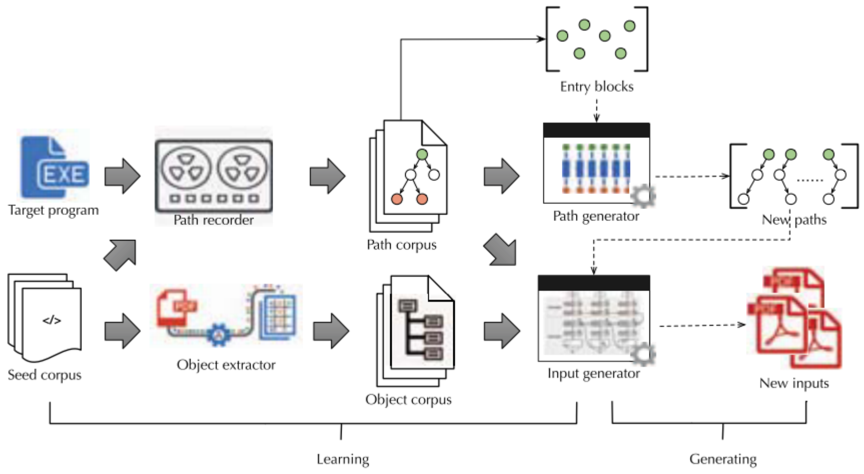
Cheng et al. [2019] address the problem of generating many similar seeds unable to increase the coverage of the fuzzing process. For solving this issue the authors propose using RNN for identifying the seeds, thus having more chances of covering new sections of the fuzzed software. Redundant results are eliminated by the RNN increasing the ratio of significant results per number of tests.

The resultant seeds are used by the *test case generation module.* The targeted software requires receiving valid PDF files. Generating valid complex-format files like PDF is not trivial, so the author uses a different machine learning algorithm, seq2seq for inserting the appropriate sequences and generate those files.

RNN in the seeds generation process provides, according to the author, better results *when learning long sequences of discrete tokens.* For supporting this affirmation, the author confronts RNN against other two machine learning algorithms: AutoRegressive Integrated Moving Average (ARIMA) and Convolutional Neural Networks (CNN).

Generating seeds able to increase the coverage of the fuzzing process requires training the RNN model7. First the author uses an Intel tool for recording the paths status after each cycle. Then the paths are shortened by *replacing short sequences of basic blocks shared by multiple execution paths with super-*

Figure 7: A framework for improving seed inputs in fuzzing



blocks. After this process, the paths are inserted in the RNN model. The authors then use the char-rnn1 implementation by Karpathy [2015]which allows a *two layer structure of standard char-RNNs.* In turn, this two layer allow training the model for both, identifying *how basic blocks form functions* and *how functions form complete execution paths.* Finally, the result of this process is processed to generate a valid PDF file. Only seeds able to reach uncovered paths are considered valid, and this decision is taken by a seq2seq machine learning model Britz et al. [2017].

The effectiveness of the approach is evaluated on three different software: MuPDF, libpng and freetype. Each one of the targetted binaries deals with different filetypes: PDF, PNG and TFF. The results on a 24 hours limited test sow great improvements on both the crashes and the covered paths.

4.1.2 Faster Fuzzing: Reinitialization with Deep Neural Models

The paper by Nichols et al. [2017] presents a framework for improving the results of the American Fuzzy Lop (AFL) fuzzer by altering the original randomness of the seeds generation.

The study compares the results of three different strategies when generating the seed for the fuzzing process:

- Standard AFL seed
- Seed generation based on Generative Adversarial Networks (GAN)
- Seed generation based on Long short-term memory (LSTM)

As a first step, the researcher runs AFL against the tested binary, ethkey, during an arbitrary time period. AFL generates seeds that are used for running the fuzzing process but also for training the machine learning model. However, not all these seeds are used. The author proposes discarding both identical and same-size seeds.

The GAN and the LSTM are then trained with the selected seeds. The trained models are executed to generate the same amount of new samples. As a final step, AFL is executed using the newly-generated samples.

Table 3: Faster Fuzzing: results

Strategy (C)	L(C)	Novel Rate
Random	778	1.000
GAN	555	0.705
LSTM	837	1.062

L(C) code paths with unique length per second, Novel Rate rate of code paths not found in the training set.

Nichols et al. [2017]

The results show that determinate deep learning models can be used in the process of seed generation. The quality of the results differ depending of the chosen algorithm, in this sense Generative Adversarial Networks have delivered better results.

4.2 Test case generation

In test case generation there is a predominance of using the sequence to sequence model with neural networks. The reviewed literature emphasizes the importance of applying the mutations in the right positions. Fan and Chang [2018] and Godefroid et al. [2017] expose how the selection of the insert point affects the results, they define three possible choices:

1. *NoSample*: each character is generated based on a given prefix. This strategy chooses the character with highest probability identified by the model. The outcome of this strategy are valid test cases with low or without entropy. The generated cases simply re-create the samples used for training the model.

2. *Sample*: each character is chosen from a group of characters from the combination of the different patterns learned by the model. This strategy does not choose always the character with higher probability for the given prefix, but takes one of the possible characters (The random function for this selection is not described in the paper). The result adds entropy to the test cases which is interesting for the fuzzing process but produces less valid cases than other strategies.

3. *SampleSpace*: the strategy defined in the *NoSample* is applied for generating each character until there is a white-space; when this happens the *Sample* strategy is applied and so on until a end-of-file is generated. As a result, this strategy provides a balance between the previously described ones by reducing the entropy added by the *Sample* strategy.

The two papers reviewed are almost equal in their approach; both use seq2seq, but Godefroid et al. [2017] uses it with RNN while Fan and Chang [2018] uses LSTM. The strategies described by Godefroid et al. [2017] (*NoSample, Sample* and *SampleSpace*) are used by Fan and Chang [2018] with different names (*Max at Each Step*, *Sample at Each Step* and *Sample on Spaces*) although no reference to each other appear in the papers.

The benefits of using sequence to sequence models for test case generation appear limited. The results obtained in the two reviewed paper using *SampleSpaces* do not differ much from a random approach, however it is possible that the models improve with reinforcement, although this requires investing more time and it is uncertain whether the results will be greater in terms of crashes per time unit.

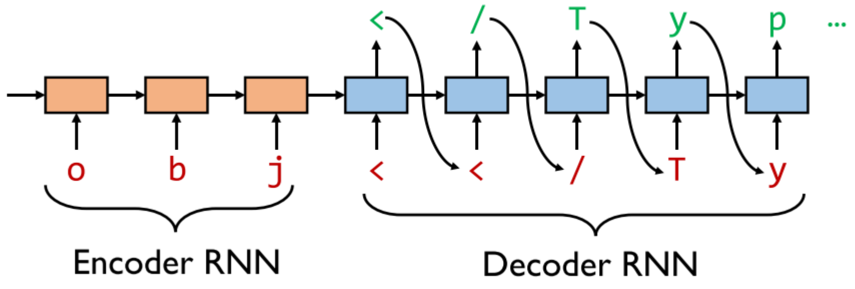
4.2.1 Learn&Fuzz: Machine learning for input fuzzing

The research by Godefroid et al. [2017] makes a proposal for improving the test case generation in grammar-based fuzzers. For this purpose the author uses a Sequence-to-sequence (seq2seq) containing two Recurrent Neural Networks (RNN) using unsupervised learning. The generated test cases should be aligned with the PDF format.

Defining the grammar boundaries increases the chances of generating valid cases. Non-grammar-based fuzzers will fail against complex formats like the targeted in this research. The paper focuses on fuzzing *non-binary data objects* that can be fuzzed by the already-existent techniques based on white-box and black-box strategies.

The proposed solution is based on a seq2seq model trained for generating valid PDF test cases.

Figure 8: Learn&Fuzz: Machine learning for input fuzzing: model diagram



Godefroid et al. [2017]

The seq2seq model 8is composed by two RNNs; an encoder and a decoder. The purpose of the encoder consists in normalizing the inputs with different lengths to a *fixed dimensional representation.* The decoder then takes the results of the encoder to generate *variable dimensional output sequences.*

The process takes the corpus section from all the PDF files and then splits the result in blocks with the same size. As a result the model generates test cases following the PDF specification. All the generated test cases start with a valid prefix, then a variable section filled with the data generated by the trained model which ends wherever the model generates an end-of-object suffix (in this case *endobj*).

After training the machine-learning model, this can be used for generating the test cases. To ensure that the generated cases are valid PDF objects, but at the same time keeping chances to trigger exceptions in the tested software, the author defines the following strategy: generate the next character from a given prefix, then, when the current prefix is a white space, sample the distribution. This allows avoiding choosing always the *top predicted character* and, in consequence generating test cases without significant variations.

The results obtained by the author bring coverage improvements when running the fuzzing process using the *test-driver* tool by Microsoft. The author indicates that this results could be improved with reinforced learning.

4.2.2 Machine Learning for Black-Box Fuzzing of Network Protocols

Fan and Chang [2018] propose using machine learning to improve the generation of test cases so that they are capable to perform black-box fuzzing on network protocols.

The paper uses Sequence to Sequence with long short-term memory neural networks. The trained model has to be capable of fuzzing a network protocol without knowing its specification or the code implementation.

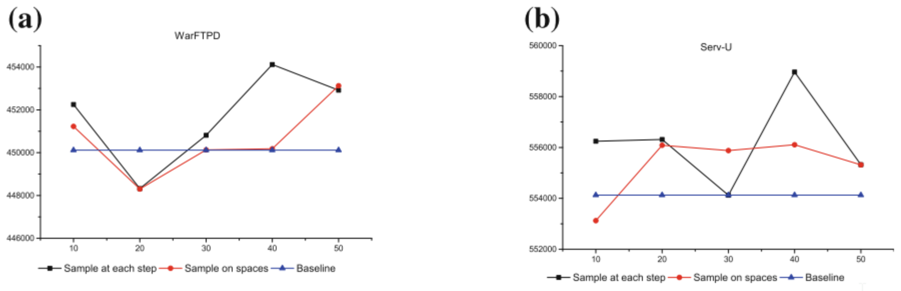
The raw data used is a dump of network traffic containing only packets of the targeted protocol. From this raw data, a dataset is composed by (unique) characters sorted by the number of occurrences. The characters and their position in the list are stored in a dictionary. This dictionary is then used to calculate optimizations related with the used technology (these will not be reviewed because it does not add value to the present review).

The test cases generated by this process are then used against a server running the targeted software. A specifically created listener handles the responses. The results are analyzed using the Microsoft AppVerifier Microsoft [2017].

The coverage results are presented using as a baseline the normal use of the applications, which is the same network traffic used for training the models. The baseline is not related with a fuzzing process and therefore does not allow assessing the real performance of the seed strategy. When comparing the results with sampling at each step vs. at each white space the differences in performance do not seem significant and more substantial and statistically sound evidence is missing.

The fuzzing speed of the proposal is compared with other well-known fuzzing tools. In this aspect the fuzzing process is significantly slower than the other solutions.

Figure 9: Coverage for WarFTPD and Serv-U from 10 to 50 epochs.



4.3 Test case filter

The reviewed literature related with test case filtering indicates that machine learning techniques can be used for filtering test cases.

Although unsupervised machine learning algorithms may be useful for many tasks, this fuzzing stage seems to be suitable for using regression algorithms (supervised learning). The reason is because in this stage we do not need generating new data but just selecting the best samples arriving to the filter, then we have an automated evaluator that indicates the quality of the result. Despite the computational time overrun added by training the model, the reviewed literature has revealed that the benefits overcome the costs.

A test case filter should be able to deal with different-length samples, in Rajpal et al. [2017] the Long short-term memory architecture is chosen because of their capability on running variable length inputs. However this model has not brought very impressive results to the author. It could be (but this is a personal conclusion since there is not enough information in the paper) that the architecture complexity adds processing costs that penalize the final results of the process.

Future works could compare the results of the logistic regression with the Decision Tree algorithm but also with Random Forest and Neural Network. All three require supervised learning but the last two may require more time for training the models and this can reduce the effectiveness for fast fuzzing processes.

4.3.1 Improving Grey-Box Fuzzing by Modeling Program Behavior

Karamcheti et al. [2018b] exposes how machine learning can improve the efficiency of a popular fuzzer, American Fuzzy Lop (AFL), by altering its behavior.

While generating seeds does not require high computational costs, evaluating them against the tested software does. Most of the fuzzers follow a common approach for filtering test cases: they maintain a set of useful inputs, these are used for testing the targeted software, and if the input covers a new path (not covered previously) this input is included in the set and used in further iterations. The author proposes optimizing the number of test cases by discarding those with less chances to produce successful results.

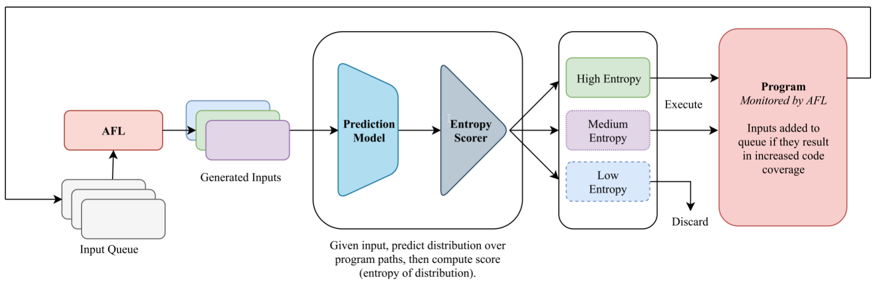
The followed strategy consists on taking the test cases generated by AFL and use them and the obtained result to train a Logistic Regression (LR) machine learning model. This model is trained with the execution cycles, the more they are, the more the filtering precision is.

They describe the steps of the process as follows:

1. *Use AFL to generate some number of possible children inputs,*
2. *Feed these inputs through our model to predict distributions over execution paths,*
3. *Rank these generated inputs by the confidence in the predictions,*
4. *Execute some fraction of those ranked inputs that we are the least confident about, and*
5. *Use the executed inputs to retrain our path prediction model.*

(Karamcheti et al. [2018b])

Figure 10: Proposed workflow


AFL
Input Queue
Generated Inputs
Prediction Model
Entropy Scorer
Given input, predict distribution over program paths, then compute score (entropy of distribution).
High Entropy
Medium Entropy
Low Entropy
Execute
Discard
Program
Monitored by AFL
Inputs added to queue if they result in increased code coverage

Karamcheti et al. [2018b]

The results of this research are compared with other approaches. This is used to measure the performance of the proposal. Four baselines are considered:

1. AFL, enabling the setting for fast fuzzing: *FidgetyAFL* Bohme et al. [2019a] Bohme et al. [2019b]

2. AFL, using the batched version *Batched FidgetyAFL* Bohme et al. [2019a]

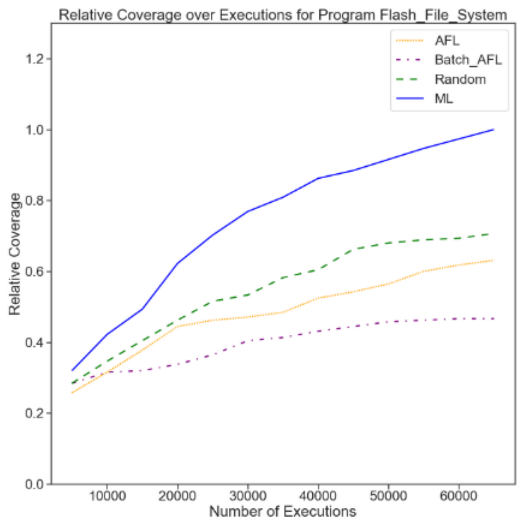
3. AFL, using the Random Batched FidgetyAFL Bohme et al. [2019a]

4. Logistic regression (the proposed approach)

The paper focuses on the benefits obtained for fast fuzzing processes. The tests have been done on 24 different binaries. The results show coverage improvements.

As with linear regression models, the precision of predictions improve when the model is fed with more samples. This improvement is confirmed with the performed tests.

Figure 11: Results comparison



Karamcheti et al. [2018b]

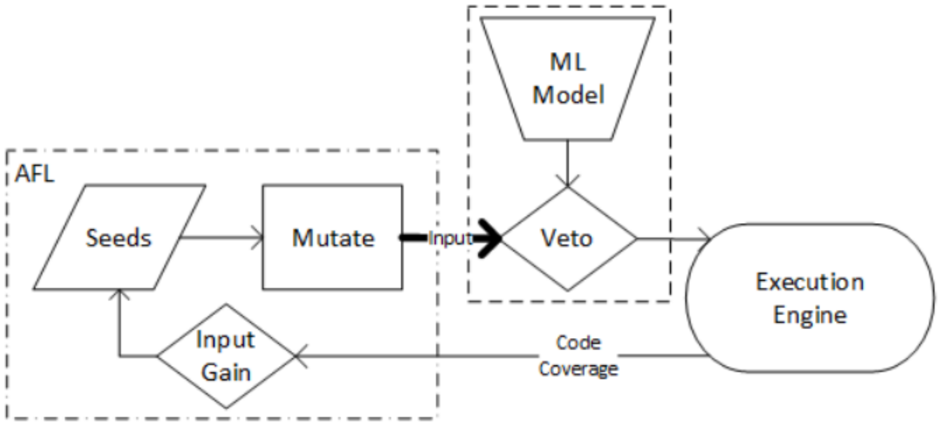
4.3.2 Not all bytes are equal: Neural byte sieve for fuzzing

In Rajpal et al. [2017], the author uses a sequence to sequence (seq2seq) model with a Long short-term memory (LSTM) recurrent neural network (RNN) for modifying the behavior of the American Fuzzy Lop (AFL) in a grey-box approach. The final objective consists of increasing the coverage of the fuzzing process for a given set of tested software pieces.

The author focuses on distinguishing the test case locations where the mutations have more chances to generate a interesting result (e.g. crash or covering a new path). First of all the author runs AFL for a limited time. The inputs covering new paths are compared to detect the locations where the mutations produce changes in the coverage. The cases with few interesting locations are discarded, so they do not influence the model training process. Finally the cases and the locations identified during this comparison are marked and used for training the model.

The paper adds an interesting strategy to the above-described process and it is relevant to be considered in future works; the author faces the challenge of distinguishing whether a given mutation (from the AFL fuzzing techniques) produces on the tested software a behavior that has not been observed before. The purpose: remove cases with high scores containing the cases where a test case gives equivalent results for different mutations. The result of this analysis is used to feed the reward function for tuning the model.

Figure 12: Not all bytes are equal Neural byte sieve for fuzzing - workflow



Rajpal et al. [2017]

AFL keeps a queue of inputs with high chances to cover new paths, however, many of them will not be successful in this sense. This paper proposes refining this process avoiding some of the inputs with low chances of discovering paths. The proposal is evaluated by testing ELF, PNG, PDF, and XML data processors.

The results reported by the author reflect significant benefits of following this strategy. However the author indicates that will be necessary adjusting the model for some file formats (PDF, and XML).

4.4 Mutation operator selection

The reviewed research on the mutation operator selection have revealed that fuzzing processes can be improved by including machine learning in this stage. The impact on the results seems to be greater in this stage than in others reviewed in this thesis (e.g. Test Case filter or Fitness function).

The analysis of the papers Drozd and Wagner [2018] and Karamcheti et al. [2018a] has revealed the following important points that should be considered in future research:

- Finding the balance between the time invested on learning and the time invested on learning is a complex problem. The Multi-armed Bandit Problem is a problem that researchers should understand and take in consideration when trying to improve the fuzzing process.

- Fuzzers often add multiple (and different different) mutation operators in each test case. While including many mutations gives the benefit of testing different operators in a single test, identifying the success of a specific operator can be done better when the number of different ones is low.

- The effectiveness of any proposal to improve the fuzzing process has a strong dependence on the process performance degradation. This is specially important for choosing mutation operators.

- Re-running the tests many times helps on tuning the process for greater results.

- When using machine learning algorithms in the mutation operators selection, reinforcement on the models can be used for directing the tests to cover more paths or discovering more crashes depending on how we define the rewards.

- Testing software from multiple domains (different purpose) is important for evaluating the performance of the fuzzing process, but the improvements added by the proposals can also be measured by the benefits added on a determinate purpose software.

Future research could evaluate the effectiveness of other algorithms with classification and regression capabilities. Two valid candidates are Random Forest or Neural Networks. Since the impact on the process performance may be noticeable, minimizing it can be a challenge.

Another possible research path can be a review of Drozd and Wagner [2018] for identifying the reason behind the differences among the different tested software.

4.4.1 Adaptive grey-box fuzz-testing with Thompson Sampling

Karamcheti et al. [2018a] presents an approach to choose the mutation operators with high probability of producing relevant results from the point of view of the fuzzing process. The proposal uses the American Fuzzy Lop as a base and then tries to alter its behavior for improving the results.

The paper follows the approach of the Multiarmed Bandit Problem by Herbert [1952]and combines it with the Thompson Sampling algorithm by Agrawal and Goyal [2012] to balance the two keys of the Multiarmed Bandit Problem: exploration and exploitation.

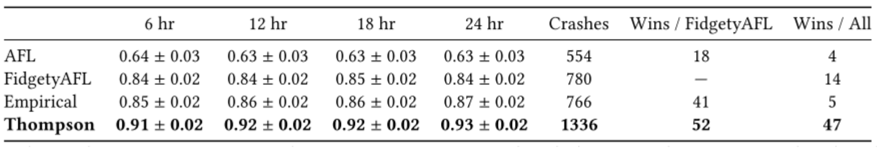
As most of the fuzzers, AFL comes with a library of possible mutation operators like multi-length bitflips, bit additions, subtractions, etc. It is expected that not all of them will have the same success rate when fuzzing a specific software. Prioritizing the ones with more chances of producing useful results (e.g. crashes, covering new paths) will then help on reducing the number of tests to be performed by the execution engine for finding a single crash.

First the author considers the mutation operators included in AFL as elements (bandits) integrating the Multi-armed Bandit Problem. Then AFL is run and each of the mutation operators is evaluated with the outcome of the process: the operators that have caused a crash or increased the coverage will be rewarded while the others will not. Following the Thompson Sampling algorithm, in next iterations the elements which have been successful will be used then more often.

AFL can run multiple mutations at the same time and the author is aware about the noise that this can add to the results (mutation operators being rewarded wrongly) for solving this, the author runs multiple times the model making AFL use a different number of mutations. The results show that using a small amount of mutations (exactly 4) and accepting the error rate added to the model gives promising results.

The proposal is evaluated with binaries from the DARPA CGC and LAVA-M data-set during 24 hours.

Figure 13: Test results



	6 hr	12 hr	18 hr	24 hr	Crashes	Wins / FidgetyAFL	Wins / All
AFL	0.64 ± 0.03	0.63 ± 0.03	0.63 ± 0.03	0.63 ± 0.03	554	18	4
FidgetyAFL	0.84 ± 0.02	0.84 ± 0.02	0.85 ± 0.02	0.84 ± 0.02	780	–	14
Empirical	0.85 ± 0.02	0.86 ± 0.02	0.86 ± 0.02	0.87 ± 0.02	766	41	5
Thompson	**0.91 ± 0.02**	**0.92 ± 0.02**	**0.92 ± 0.02**	**0.93 ± 0.02**	**1336**	**52**	**47**

Karamcheti et al. [2018a]

Although the final average is still positive, as shown in 13, the success of the tests do differ among the binaries, with some of them not showing any improvement while others do. According to the author, the cause of these differences is related with the decision of reducing the number of mutations that AFL can perform in a single sample. In his own words, "*we are in fact limiting the expressive power of our fuzzer*".

While increasing the number of mutations may seem a reasonable solution, the results provided in the paper expose a conflict when trying to find a better choice. High values on mutations per sample do not bring great results.

The reviewed publication shows that fuzzing some binaries can be faster with this strategy than with the standard AFL one, but its use can cause loosing coverage capabilities.

4.4.2 FuzzerGym: A Competitive Framework for Fuzzing and Learning

Drozd and Wagner [2018] present the research results of using machine learning for improving the coverage of fuzzing tests through the selection of the best mutation operators. The triage is done by a Reinforced Learning (RL) machine learning algorithm that bases its decisions on the structure of the input data. The paper proposes using a Markov decision process tuned for reducing the performance degradation.

The author builds a layer of improvement on the LibFuzzer tool. The samples used for trainig the RL model have the same structure used by LibFuzzer. As many other proposals, crashes and coverage improvements reported by the fuzzer are used as rewards that are translated into higher decision probabilities for future cycles.

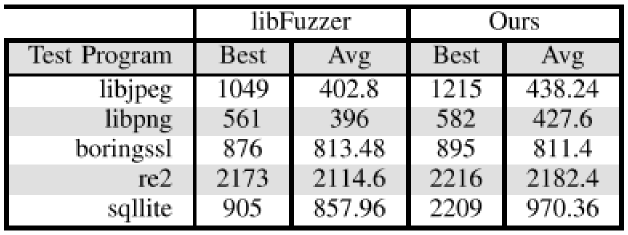
There is an interesting topic related with the the way the model is rewarded, the author points that these rewards can be used for adapting the objectives of the fuzzing process *The flexibility of the RL reward function means that it is possible to create fuzzers with various specific needs such as rapid bug finding or an increased reward for targeting specific code paths.* This adaptability is something that might seem obvious but not many research consider this in their approaches.

While evaluating the model, the author notices that evaluating all the results can cause a significant performance degradation. Executing samples on the targeted software is faster than analyzing the responses with the RL algorithm. The proposed solution consists in analyzing a subset of all the samples tested by the libFuzz fuzzer asynchronously. This is done by buffering a limited number of test-result pairs and allowing new insertions as soon as one of them is processed.

The efficiency of the proposal is evaluated against five targets that process different data nature. The results are compared with the official ones from LibFuzzer. The tested processes are:

1. Decompressing a JPEG
2. Parsing a PNG
3. Encoding a Private SSL Key
4. Evaluating a Regular Expression
5. Executing a SQL Query

Figure 14: FuzzerGym: A Competitive Framework for Fuzzing and Learning-results



	libFuzzer		Ours	
Test Program	Best	Avg	Best	Avg
libjpeg	1049	402.8	1215	438.24
libpng	561	396	582	427.6
boringssl	876	813.48	895	811.4
re2	2173	2114.6	2216	2182.4
sqllite	905	857.96	2209	970.36

Drozd and Wagner [2018]

Using reinforced learning for choosing the right mutation operators seems to bring improvements to standard executions of LibFuzzer. Still, the author express doubts about the results on other software and reminds the importance of tuning the model by changing the rewards strategy.

For future research, the author proposes studying the reason behind the improvement differences among the different tested software and extending the tests on more software domains.

4.5 Fitness function

The literature review about fitness function research has revealed that refining the results of popular fuzzers (e.g. AFL) with machine learning is possible.

The better results have been obtained by using a training dataset which is composed by binaries written in the same programming language than the one to be fuzzed.

AFL supports C, C++ and Objective C, training a machine learning model with only C binaries for targeting C binaries could be the reason behind the improvements observed in this section.

Research indicates also that limiting the capabilities of AFL for exploring new paths and produce more crashes results on more unique crashes. This could be positive for fast fuzzing processes but further research is necessary to ensure that this does not penalize exhaustive fuzzing processes. It is very likely that these conclusions can be extended to most of the fuzzing stages.

Fitness functions are used to evaluate the quality of the test cases after having completed all the previous stages. The reviewed literature selects the most useful test cases for increasing the coverage, instead future research could focus on understanding how to use the final test cases for tuning the strategies applied on the previous stages. This may allow balancing the different strategies adopted on the other stages instead of focusing only in one of them.

4.5.1 Machine Learning Augmented Fuzzing

Joffe [2018] proposes using neural networks to predict crashes. The prediction is based on a model trained with execution traces generated by the AFL fuzzer.

The author analyses all binaries contained in the Codeflaws program repository. The training dataset is adapted for improving the results of the targeted software (the analyzed binaries allow few parameters and AFL is configured to limit these).

The experiment starts with the execution of AFL over the binaries. The information of the executions is classified using the Valgrind's Callgrind tool that labels the results indicating if the sample has produced a crash or not. Then, using the Keras framework a neural network is trained for predicting crashes for each given sample. The author do not specify whether Convolutional neural networks (CNN) or Recurrent neural net (RNN) are used at this point.

After training the model, the behavior of AFL is modified for prioritizing the analysis of samples where the neural network indicates high probability of crash. Elements with intermediate ratings are not proposed to be run by the execution engine but used as seeds for future cycles (the author indicates this strategy has brought good results).

The author do not show results in this paper but continues in Joffe and Clark [2019] that will also be reviewed in the current thesis.

4.5.2 Directing a Search Towards Execution Properties with a Learned Fitness Function

Joffe and Clark [2019] shows the results from implementing the approach proposed in Joffe [2018]. The purpose of the fitness functions is to measure the quality of the samples generated by the fuzzed and help generating more useful samples in the future. In Joffe [2018], the author exposes how to use machine learning for identifying characteristics on generated samples that can be indicators of greater coverage.

The author trains a model running AFL with a subset of the *Codeflaws program repository*; 200 binaries are used. The outcome of the AFL execution is used for training a neural network (the publication do not specify which is used) is trained. The trained model is used for classifying then samples generated by AFL for fuzzing the targeted binaries.

Three binaries are analyzed: VLC media player, libjpeg library, and mpg321 library. The run experiments require limiting rewards on AFL to the strategies covering new paths in order to focus all the process on finding crashes. The author defines 8 different configurations of AFL and test them. The best results are obtained by limiting the reward for new paths discovery to a fixed amount (instead of allowing dynamic assignations) and merging this AFL valuations with the ones produced by the trained neural network.

The paper demonstrates that increasing the number of crashes with AFL is possible by modifying its behavior and adjusting the configuration of the process. While the results show big differences in the number of crashes, it is not the case with the unique crashes; although the introduction of neural networks for evaluating the results brings improvements, these are small. The paper focuses on analyzing three binaries, testing a larger dataset could bring consistence to the results.

The influence of the dataset used for training the model seems to be relevant for future research. In the research, binaries written in C language are used for both training and analysis. Experimenting this approach with other languages could help identifying the weaknesses and strengths of this proposal

4.6 Exploitability analysis

While generating working exploits still requires human intervention, analyzing how likely is that determinate results can materialize in a vulnerability is possible.

Fuzzing results usually contain plenty of crashes. All of them are related with a weakness in the software but only some will open the possibility for attackers to take advantage of them.

The reviewed literature focuses the investigation on the fields of software assurance. Yan et al. [2017] presents a tool that can estimate the level of exploitability of a given software piece. The results of these evaluations can be useful also for other purposes, for example, determining where the developers of those software need to put efforts for solving the weaknesses.

Using the results from fuzzing processes and identifying the nature of the vulnerability that could have caused the crashes has been proven possible. There are different classification-oriented machine learning algorithms that can be used for this purpose. In Yan et al. [2017] Naive Bayes is used providing acceptable results.

Future research could consider following a similar approach using neural networks based algorithms like LSTM. Testing different datasets for training the models can bring more accuracy to the results, probably using binaries based on the same programming language will help but also using samples from the same domain (e.g. PDF parsing, PNG edit, etc.) should be considered.

Although the classifying granularity is one of the objectives, reducing this to the barely minimum (as less vulnerability types as possible) could ease the task of the classifier and reduce the false positives.

4.6.1 ExploitMeter: Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability

Yan et al. [2017] presents a framework for evaluating the level of exploitability of a given software. The framework uses a Naive Bayes for both classifying the software based on previous analysis (results obtained from Basic Fuzzing Framework and Ofuzz) and classifying the errors per type of vulnerability based on the CERT triage tools.

First the classified is trained for understanding the vulnerabilities that can affect software. The classification process relies on the information provided by *!exploitable* (Windbg implementation on windows, and CERT triage tools). Four kinds of data are used for training the classifier: Hexdump features, Objdump features, Libraries features, and Relocation features.

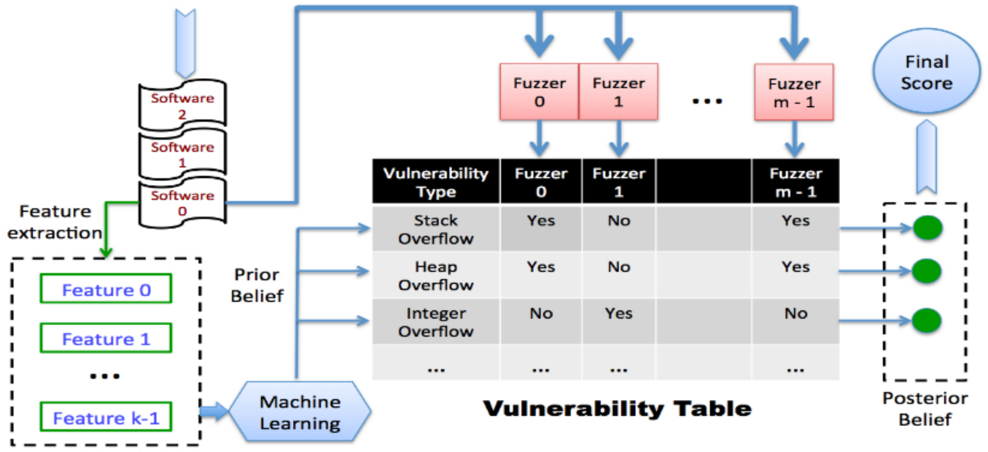
The lack of detection is not considered as lack of issues but a low confidence that they exist. This is, as the author indicates, because we cannot trust that a software is free from issues only because nobody has detected them.

Once the model is trained, it is capable to estimate the probability that a given software application suffers from a specific vulnerability. Then each program under test is analyzed by multiple fuzzers. The resulting crashes are classified by the model to indicate which type of vulnerability can be related with a determinate crash.

The final step consists of assigning a score to each software. The author defines an arbitrary value to each vulnerability type, and this is used for calculating the final score.

> The exploitability of a software program depends upon the vulnerability types it contains as well as how likely each vulnerability type can be turned into a software exploit.
>
> Yan et al. [2017]

Figure 15: Exploitmeter architecture

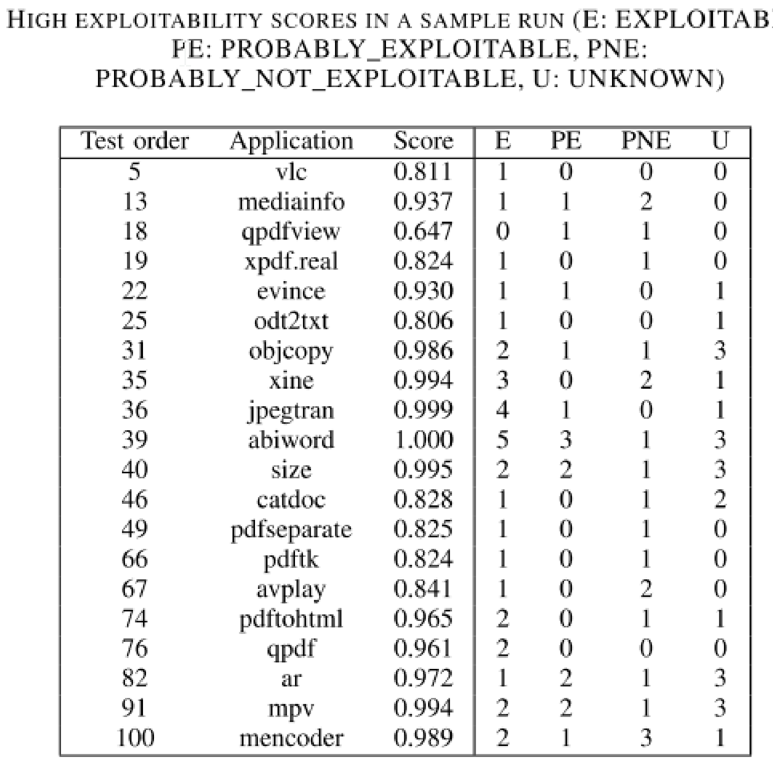


Vulnerability Type	Fuzzer 0	Fuzzer 1		Fuzzer m - 1
Stack Overflow	Yes	No		Yes
Heap Overflow	Yes	No		Yes
Integer Overflow	No	Yes		No
...	...	...		...

Yan et al. [2017]

The effectiveness of the proposal is evaluated by analyzing 100 different Linux applications. From the analyzed software, the tool identifies some with high probability of being exploitable:

Figure 16: Scores



HIGH EXPLOITABILITY SCORES IN A SAMPLE RUN (E: EXPLOITABLE, PE: PROBABLY_EXPLOITABLE, PNE: PROBABLY_NOT_EXPLOITABLE, U: UNKNOWN)

Test order	Application	Score	E	PE	PNE	U
5	vlc	0.811	1	0	0	0
13	mediainfo	0.937	1	1	2	0
18	qpdfview	0.647	0	1	1	0
19	xpdf.real	0.824	1	0	1	0
22	evince	0.930	1	1	0	1
25	odt2txt	0.806	1	0	0	1
31	objcopy	0.986	2	1	1	3
35	xine	0.994	3	0	2	1
36	jpegtran	0.999	4	1	0	1
39	abiword	1.000	5	3	1	3
40	size	0.995	2	2	1	3
46	catdoc	0.828	1	0	1	2
49	pdfseparate	0.825	1	0	1	0
66	pdftk	0.824	1	0	1	0
67	avplay	0.841	1	0	2	0
74	pdftohtml	0.965	2	0	1	1
76	qpdf	0.961	2	0	0	0
82	ar	0.972	1	2	1	3
91	mpv	0.994	2	2	1	3
100	mencoder	0.989	2	1	3	1

Yan et al. [2017]

The author of the paper indicates the need of performing more tests and tuning in order to increase the accuracy of the predictions, this is because during the tests many low probability occurrences happened.

Reading this paper rises two observations:

1. The number of vulnerability definitions can be a critical factor for this and other similar proposals. While having few definitions could mean

missing relevant issues, having many of them could lead to errors in their classification.

2. The use of ExploitMeter (or other similar solutions) for Software assurance can be interesting as an additional layer for checking the risk level of using a specific software, unfortunately while the revision can detect some codding vulnerabilities all the errors introduced through the application's business logic will not be detected.

5 Discussion and Conclusions

This thesis reviews the main strands of literature on the application of machine learning algorithms in the fuzzing process. In doing this, it shows the strengths and weaknesses of different machine learning algorithms when they are applied to fuzzing tests. It also determines which algorithms are more suitable for each stage of the fuzzing process, and which should be used in future research. These observations are a novelty of this thesis, and the conclusions presented for each of the fuzzing steps can inspire new applications since many possibilities have not yet been investigated.

Seed generation is a stage where the improvements brought by machine learning have a large impact. There are two ways of improving the seed generation:

- Generating high quality seeds: when few valid samples are available, algorithms like Generative Adversarial Networks can populate the fuzzer with valid seeds.
- Discarding low quality seeds: the results of the fuzzer's evaluation engine can train models capable of identifying seeds with low chances of producing crashes or covering new paths of the code. In this stage, Recurrent Neural Networks have been used with success. This thesis concludes that the Long Short-Term Memory algorithm is likely to bring even better results.

Discarding samples at this stage means not processing them in the following stages. Errors in this process could cause a lack of coverage if the wrong seeds are discarded.

In the test case generation stage, the objective is to transform the seeds for generating new cases. Selecting the right position where to apply the transformation affects the efficiency of the fuzzing test. The reviewed literature implements both Recurrent Neural Networks and the Long Short-Term Memory algorithm for improving this stage. However, the cost of training the networks partially offset the benefits obtained with this design. The observed improvements are consequently moderate in this stage.

After generating the test cases, the test case filter stage identifies the most valuable samples. This stage is necessary because some of the samples may produce redundant results. In particular, two papers have demonstrated to improve this process using machine learning algorithms: one uses logistic regression and the other one uses neural networks (both RNN and LSTM). The results have shown moderate benefits of using any of those algorithms. In general, algorithms which have low impact on performance seem to be more suitable for this task, even if they are less accurate.

The mutation operator selection stage alters the test cases to produce unexpected behaviors in the fuzzed software. The literature uses Reinforced Learning and Thompson Sampling to improve the selection of the operators. The results indicate that implementing such algorithms largely affects the speed of the fuzzing process. It is possible that the accuracy of the operators selection can improve with the use of algorithms like Random Forest or Recurrent Neural Networks, but these algorithms imply extra computational costs which may degrade the performance.

Once the test cases have been created, a fitness function stage analyses them for discarding those with low chances of producing relevant results. For example, using machine learning it is possible to detect which seeds have more chances of covering a new path or producing a crash. The reviewed papers use neural networks for executing this analysis. The models are then able to predict when a given sample has high or low chances of producing a relevant result. Although the publications describe improvements in the fuzzing process, the impact is lower than the one obtained when altering other stages like the mutation operator selection or the seed generation.

The last stage composing the fuzzing process is the exploitability analysis. The quality of the results obtained in this stage determines crucially the value that the fuzzing processes can add. Classifying correctly the exceptions triggered by the fuzzer is the major challenge that researchers face in this stage. This classification can be done with machine learning, but it depends heavily on the availability of datasets for training the model and on the granularity with which the crashes must be classified.

As a general remark, the typical approach of the current literature is: to include a machine learning algorithm in one of the stages of a given fuzzer, then to execute tests for measuring the performance, and finally to adjust the model and to compare the results with the standard behavior of the fuzzer.

Although comparing the results within a single stage, in isolation, make machine learning look promising in most of the papers, none of the reviewed studies considers how the specific technique behaves in combination with others. A more systemic approach would be beneficial. Combining the improvements brought by the machine learning algorithms can be a tough challenge. The reasons are multiple:

- Process performance: finding the best balance between investing time in learning and performing tests. This becomes a difficult decision when it has to be taken for multiple processes.
- Chained processes: the improvements on a determinate stage may reduce the effectiveness of the algorithms used in later steps. This effect can be caused by overfitted or underfitted models.
- Lack of standard benchmarks: the type of input data affects both the performance of the fuzzing process and the optimal machine learning algorithm to be used, making comparisons difficult. Ideally, there should be a standard collection of datasets divided by the protocol or file-format used by the software subject of being tested.

Future researchers should try to use multiple combinations of machine learning algorithms and find the best ones. While some combinations are potentially under-performing or point to inaccurate results, others could bring disruptive improvements.

List of Figures

Bibliography

Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. *Journal of Machine Learning Research*, 23: 1–26, 2012. ISSN 15337928. URL `http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf`.

Richard Bellman. Markovian decision processes, 1977. ISSN 00765392. URL `https://www.iumj.indiana.edu/IUMJ/FTDLOAD/1957/6/56038/pdf`.

Marcel Bohme, Van Thuan Pham, and Abhik Roychoudhury. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019a. ISSN 19393520. doi: 10.1109/TSE.2017.2785841. URL `https://mboehme.github.io/paper/CCS16.pdf`.

Marcel Bohme, Van Thuan Pham, and Abhik Roychoudhury. aflfast, 2019b. URL `https://github.com/mboehme/aflfast`.

Denny Britz, Anna Goldie, Minh Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures, 2017. URL `https://github.com/farizrahman4u/seq2seq`.

Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. *Proceedings of the 28th USENIX Security Symposium*, pages 1967–1983, 2019. URL `https://www.usenix.org/system/files/sec19-chen-yuanliang.pdf`.

Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. Optimizing seed inputs in fuzzing with machine learning. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019*, pages 244–245, 2019. doi: 10.1109/ICSE-Companion.2019.00096.

William Drozd and Michael D. Wagner. FuzzerGym: A Competitive Framework for Fuzzing and Learning. 2018. URL `http://arxiv.org/abs/1807.07490`.

Rong Fan and Yaoyao Chang. Machine learning for black-box fuzzing of network protocols. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10631 LNCS, pages 621–632, 2018. ISBN 9783319894997. doi: 10.1007/978-3-319-89500-0_53.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices*, 43(6):206–215, 2008a. ISSN 15232867. doi: 10.1145/1379022.1375607. URL `https://people.csail.mit.edu/akiezun/pldi-kiezun.pdf`.

Patrice Godefroid, Michael Y. Levin, and David a. Molnar. Automated Whitebox Fuzz Testing. *Network and Distributed System Security Symposium (NDSS)*, 9(July):pdf, 2008b. ISSN 1064-3745. doi: 10.1007/978-3-642-02652-2_1. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.151.9430{&}rep=rep1{&}type=pdf`.

Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. ISSN 00010782. doi: 10.1145/2093548.2093564. URL `https://dl.acm.org/doi/pdf/10.1145/2090147.2094081`.

Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine learning for input fuzzing. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59, 2017. ISBN 9781538626849. doi: 10.1109/ASE.2017.8115618. URL `https://ieeexplore.ieee.org/abstract/document/8115618/`.

Weiwei Gong, Gen Zhang, and Xu Zhou. Learn to accelerate identifying new test cases in fuzzing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10656 LNCS, pages 298–307, 2017. ISBN 9783319723884. doi: 10.1007/978-3-319-72389-1_24.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in Neural Information Processing Systems*, 3(January):2672–2680, 2014. ISSN 10495258. doi: 10.3156/jsoft.29.5_177_2. URL `https://arxiv.org/pdf/1406.2661.pdf`.

Google. ClusterFuzz, 2020a. URL `https://google.github.io/clusterfuzz/`.

Google. Seq2Seq - Introduction, 2020b. URL `https://google.github.io/seq2seq/`.

Robbins Herbert. Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952. ISSN 02730979. doi: 10.1090/S0002-9904-1952-09620-8. URL `https://www.ams.org/journals/bull/1952-58-05/S0002-9904-1952-09620-8/S0002-9904-1952-09620-8.pdf`.

Leonid Joffe. Machine Learning Augmented Fuzzing. In *Proceedings - 29th IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2018*, number Ml, pages 178–183. IEEE, 2018. ISBN 9781538694435. doi: 10.1109/ISSREW.2018.000-1.

Leonid Joffe and David Clark. Directing a search towards execution properties with a learned fitness function. *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pages 206–216, 2019. doi: 10.1109/ICST.2019.00029.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. ISSN 10769757. doi: 10.1613/jair.301. URL `https://arxiv.org/pdf/cs/9605103.pdf`.

Josh Kalin. *Generative adversarial networks cookbook : over 100 recipes to build generative models using Python, TensorFlow, and Keras.* 2019. ISBN 9781789139907. URL `https://www.packtpub.com/big-data-and-business-intelligence/generative-adversarial-networks-cookbook`.

Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 37–47, 2018a. ISSN 15437221. doi: 10.1145/3270101.3270108. URL `https://arxiv.org/pdf/1808.08256.pdf`.

Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Improving Grey-Box Fuzzing by Modeling Program Behavior. 2018b. URL `http://arxiv.org/abs/1811.08973`.

Masud Karim and Rashedur M. Rahman. Decision Tree and Naïve Bayes Algorithm for Classification and Generation of Actionable Knowledge for Direct Marketing. *Journal of Software Engineering and Applications*, 06 (04):196–206, 2013. ISSN 1945-3116. doi: 10.4236/jsea.2013.64025.

Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, 2015. URL `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. 2014. URL `http://arxiv.org/abs/1404.5997`.

Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. ISSN 14764687. doi: 10.1038/nature14539.

Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey, 2018. ISSN 2523-3246.

Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing. pages 1–16, 2019. URL `http://arxiv.org/abs/1901.01142`.

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, and Wuwei Shen. Fuzzing : State of the Art. 67(3):1199–1218, 2018.

Jian Wei Liao, Tsung Ta Tsai, Chia Kang He, and Chin Wei Tien. SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing. *2019 6th International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019*, pages 458–465, 2019. doi: 10.1109/IOTSMS48152.2019.8939256.

Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. *Others*, 2019. URL `www.aaai.org`.

Microsoft. Application Verifier, 2017. URL `https://docs.microsoft.com/en-us/security-risk-detection/concepts/application-verifier`.

Microsoft. Microsoft Security Risk Detection, 2020. ISSN 02637863. URL `https://devblogs.microsoft.com/premier-developer/microsoft-security-risk-detection/`.

Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12): 32–44, 1990. ISSN 15577317. doi: 10.1145/96267.96279. URL `https://dl.acm.org/doi/abs/10.1145/96267.96279`.

Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster Fuzzing: Reinitialization with Deep Neural Models. (2008), 2017. URL `http://arxiv.org/abs/1711.02807`.

Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. pages 1–10, 2017. URL `http://arxiv.org/abs/1711.04596`.

Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. *Proceedings of the 23rd USENIX Security Symposium*, pages 861–875, 2014.

Tyler Reid, Todd Walter, Per Enge, and Ananda Fowler. Crowdsourcing arctic navigation using multispectral ice classification &GNSS. *27th International Technical Meeting of the Satellite Division of the Institute of Navigation, ION GNSS 2014*, 1:707–721, 2014.

Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. URL `http://arxiv.org/abs/1609.04747`.

Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A tutorial on Thompson sampling. *Foundations and Trends in Machine Learning*, 11(1):1–96, 2018. ISSN 19358245. doi: 10.1561/2200000070. URL `https://web.stanford.edu/{~}bvr/pubs/TS{_}Tutorial.pdf`.

Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A Review of Machine Learning Applications in Fuzzing. 2019. URL `http://arxiv.org/abs/1906.11133`.

Ramani Sagar, Rutvij Jhaveri, and Carlos Borrego. Applications in security and evasions in machine learning: A survey. *Electronics (Switzerland)*, 9 (1):1–42, 2020. ISSN 20799292. doi: 10.3390/electronics9010097.

Ari. Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. 2008. ISBN 9781596932142.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. *Proceedings - IEEE Symposium on Security and Privacy*, pages 579–594, 2017. ISSN 10816011. doi: 10.1109/SP.2017.23.

Yan Wang, Peng Jia, Luping Liu, and Jiayong Liu. A systematic review of fuzzing based on machine learning techniques. aug 2019. URL http://arxiv.org/abs/1908.01262.

Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. ExploitMeter: Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability. *Proceedings - 2017 IEEE Symposium on Privacy-Aware Computing, PAC 2017*, 2017-Janua:164–175, 2017. doi: 10.1109/PAC.2017.10. URL https://www.cs.binghamton.edu/{~}ghyan/papers/pac17.pdf.

Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, Xiaofeng Wang, and Bin Liang. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2019-May, pages 769–786, 2019. ISBN 9781538666609. doi: 10.1109/SP.2019.00057. URL https://www.cs.purdue.edu/homes/ma229/papers/SP19.pdf.

Xiaofeng Yuan, Lin Li, and Yalin Wang. Nonlinear Dynamic Soft Sensor Modeling with Supervised Long Short-Term Memory Network. *IEEE Transactions on Industrial Informatics*, 16(5):3168–3176, 2020. ISSN 19410050. doi: 10.1109/TII.2019.2902129.

Zeller Andreas, Gopinath Rahul, Böhme Marcel, Fraser Gordon, and Holler Christian. *The Fuzzing Book*. 2019. URL https://www.fuzzingbook.org/{#}citation.