FUZZING SOFTWARE WITH DEEP LEARNING

MARTIN SABLOTNY

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

MARCH 2023

Abstract

Generation based fuzz testing can uncover various bug classes and security vulnerabilities. However, compared to mutation based fuzz testing it takes a great amount of time to develop a well balanced generator that generates good test cases and decides were to break the underlying structure to exercise new code paths.

This thesis provides an evaluation of generative deep learning algorithms to generate HTML test cases to fuzz test a browser's HTML rendering engine. The experiments highlight that various deep learning algorithm are performing well in this setting. However, there are large differences in the stability of the training and code coverage performance. The best performing in terms of code coverage as well as training stability is a Temporal Convolutional Network (TCN).

The TCN model is then also used to learn from real world HTML data to generate novel test cases withouth the need of a generative fuzzer in the first place. The results show that the approach is able to discover new code areas that were neither discovered by the underlying fuzzer nor the prior models. Furthermore, this highlights how an existing fuzzer can be augmented with the help of a deep learning model and publicly available training data.

Finally, reinforcement learning is used to further improve the existing fuzzer by utilizing the code coverage data from the browser under test. The designed DDQN agent is able to guide the test case creation of a TCN to even outperform the underlying baseline test case generator.

Acknowledgements

First, I would like to thank Dr. Bjørn Jensen and Dr. Jeremy Singer for their guidance throughout the PhD journey. Furthermore, I also want to thank Professor Chris Johnson for encouraging me to start my PhD studies and for his support during the initial phase of the studies.

Secondly, I would like to thank my wife, Hannah, for her love and unconditional support in continuing and finalizing the PhD studies.

Thirdly, I would like to thank my parents for their support in laying the foundations necessary to pursue this degree in the first place.

Finally, I like to thank NVIDIA and especially Dr. Chris Schneider for sponsoring the GPUs used during the research as well as providing additional feedback throughout the study.

Even a fool learns something once it hits him. – Homer, Iliad

Table of Contents

1 Introduction **1**

1.1 Thesis Statement 4

1.2 Research Questions 4

1.3 Research Contributions 5

1.4 Thesis Structure 5

2 Background and Related-Work **7**

2.1 Software Fuzz Testing 7

2.1.1 History 8

2.1.2 Behaviour Analysis 8

2.1.3 Test case creation 9

2.1.4 Summary Fuzz Testing 12

2.2 Machine Learning 12

2.2.1 History 14

2.2.2 Recurrent Neural Networks 15

2.2.3 Convolutional Neural Networks 17

2.2.4 Model Architectures 18

2.2.5 Other approaches 21

2.2.6 Reinforcement Learning 22

2.2.7 Optimisation 24

2.2.8 Summary of Machine Learning 25

2.3 Machine Learning and Fuzz Testing 25

2.4 Summary 28

3 Experimental Environment **30**

3.1 Environmental Setup 30

3.2 Dataset Creation 31

3.3 Data Collection 33

3.4 Baseline Results 36

3.5 Summary 37

4 Effects of different RNN based architectures on the Code Coverage Performance **39**

4.1 Stacked Recurrent Neural Network 40

4.1.1 Model Architecture 40

4.1.2 Model Training 42

4.1.3 HTML-tag generation 43

4.1.4 Results 44

4.2 Sequence-to-Sequence 54

4.2.1 Model Architecture 54

4.2.2 Model Training 55

4.2.3 HTML-tag generation 56

4.2.4 Results 57

4.3 Discussion 63

4.4 Summary 64

5 Effects of different TCN based architectures on the Code Coverage Performance **67**

5.1 Default Temporal Convolutional Networks 68

5.1.1 Model Architecture 68

5.1.2 Model Training 69

5.1.3 HTML-tag Generation 70

5.1.4 Additional Comparison 71

5.1.5 Results 72

5.2 Sequence-to-Sequence Temporal Convolutional Network 77

5.2.1 Model Architecture 77

5.2.2 Model Training . . . 78
5.2.3 HTML-tag Generation . . . 78
5.2.4 Results . . . 79
5.3 Discussion . . . 83
5.4 Summary . . . 84

6 Effects of using real world training data on the code coverage performance 87
6.1 Real World Dataset . . . 88
6.2 Model Training . . . 89
6.3 HTML generation . . . 89
6.4 Results . . . 89
6.4.1 Minimal Preprocessing . . . 89
6.4.2 Additional Preprocessing . . . 93
6.4.3 Adjusting the sampling strategy . . . 94
6.5 Discussion . . . 96
6.6 Summary . . . 97

7 The Application of Reinforcement Learning on Generating HTML 100
7.1 Extended Environment . . . 100
7.2 DDQN Architecture . . . 102
7.3 DDQN Training . . . 103
7.4 Experiments . . . 104
7.5 Results . . . 106
7.6 Discussion . . . 107
7.7 Summary . . . 108

8 Conclusions and Future Work 110
8.1 Thesis Statement revisited . . . 110
8.2 Summary of the Key Results . . . 110
8.3 Summary of Contributions . . . 112
8.4 Future Work and Final Remarks . . . 112

A Appendix **114**

A.1 HTML-tag Distribution 114

A.2 Hello World Example 118

Bibliography **120**

List of Tables

3.1 Basic blocks discovered by the different datasets 37

3.2 Mutation set performance 37

4.1 Summary of the training results of the stacked LSTM and GRU models. It provides the training and validation minimum loss and highest accuracy, the average number of training steps, the average training duration in minutes, and the number of trainable parameters. The internal unit size of the models was set to 256 in all training runs. 44

4.2 Overlap in percent of discovered basic blocks of the best performing LSTM models. The best performing models are compared against the dataset and the randomly mutated test runs introduced in Section 3.4. 49

4.3 Overlap in percent of discovered basic blocks of the best performing GRU models. The best performing models are compared against the dataset and the randomly mutated test runs introduced in Section 3.4. 52

4.4 Trainable parameters by model depth for GRU based architectures 58

5.1 The different configurations used during training. The kernel size k is fixed over all convolutional layers (max. 7) and the dilation rate d_i is adjusted to cover the whole input sequence (max. 200 characters). Furthermore $dense_i$ provides the number of internal units used. 70

5.2 Summary of the training results of the TCN models. It provides the training and validation minimum loss, the average number of training steps, the average training time in minutes, and the number of trainable parameters. . 73

5.3 Best performing default TCN model's overlap with the best performing dataset and the differently mutated sets. 75

5.4 Comparison between the TCN models' performance with the dataset and AFL++ runs in total basic blocks. 76

5.5 Trainable parameters by model depth for Seq2Seq TCN based architectures 80

5.6 Best performing seq2seq TCN model's overlap with the best performing dataset and the differently mutated sets. 83

6.1 Best performing TCN model's overlap with the best performing dataset baseline and the differently mutated sets. 92

6.2 Direct comparison of the config-05 default TCN models trained on the default dataset and the preprocessed dataset in terms of triggered basic blocks with 12KB per test case. The last column shows the arithmetic mean. . . . 94

6.3 Direct comparison of the config-05 default TCN models trained on the default dataset and the preprocessed dataset in terms of triggered basic blocks with 24KB per test case. The last column shows the arithmetic mean. . . . 95

6.4 Direct comparison of the configuration five models using random tag insertion with a probability of 40% and 75% in terms of triggered basic blocks with 12KB per test case. 95

6.5 Direct comparison of the configuration five models using random tag insertion with a probability of 40% and 75% in terms of triggered basic blocks with 24KB per test case. 96

7.1 The configurations used for the convolutional layers in Figure 7.2. 102

7.2 Evaluated hyper-parameter configurations for the DDQN agent. The parameters k_i, s_i, f_i provide the kernel size, stride and filter dimensions of the CNN layer i respectively. d_i provides the number of internal units of the dense layer i. 104

A.1 Absolute distribution of HTML-tags in the fuzzer created dataset and the real world dataset. 117

List of Figures

1.1 Workflow during fuzz testing 2

3.1 General code coverage collection process 33

4.1 Example model architecture with two layers. The input x_0 to x_n is sequence of positve integers representing a character each. The output module provides a probability distribution to predict the character at x_{n+1}. 40

4.2 Average validation loss of the different LSTM based models (30MB data set) with error-bars indicating the standard deviation across the individual models. 45

4.3 Error rate per HTML-tag in regards to model depth for stacked RNN models with LSTM cells 47

4.4 LSTM based models: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 48

4.5 LSTM based models: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 49

4.6 Average validation loss of the different GRU models with error-bars indicating the standard deviation across the individual models. 50

4.7 Error rate per HTML-tag in regards to model depth for stacked RNN models with GRU cells 51

4.8 GRU-based models: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 52

4.9 GRU-based models: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 53

4.10 The seq2seq architecture used as HTML-fuzzer. The encoder consisted of two GRU layers and the decoder consisted of three layers. 54
4.11 Average validation loss of the seq2seq models with error-bars indicating the standard deviation across the individual models 58
4.12 Error rate per HTML-tag in regards to the internal size of the seq2seq model. 58
4.13 Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 60
4.14 Code coverage performance with 256 HTML-tags per test case: (left) absolute number of basic blocks discovered by model; (right) basic blocks not discovered by the test set 61
4.15 The impact of increasing the standard deviation during the sampling on the HTML error rate per tag. 61
4.16 Code coverage performance of the models with 256 internal units with varying standard deviation in the setting with 128 HTML-tags per case. 62
4.17 Code coverage performance of the models with 256 internal units with varying standard deviation in the setting with 256 HTML-tags per case. 63
4.18 Comparison of the best performing models with 128 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset 64
4.19 Comparison of the best performing models with 256 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset 65

5.1 General architecture of the default TCN models 68
5.2 Average validation loss of the default TCN models with error-bars indicating the standard deviation across the individual models 72
5.3 Error rate per HTML-tag in regards to the model configuration in terms of kernel size and dilation rate 74
5.4 Default TCN: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 75
5.5 Default TCN: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 76
5.6 Seq2Seq TCN architecture overview 77

5.7 Average validation loss of the default TCN models with error-bars indicating the standard deviation across the individual models 80

5.8 Error rate per HTML-tag in regards to the model configuration in terms of kernel size and dilation rate 81

5.9 Seq2Seq TCN: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 81

5.10 Seq2Seq TCN: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 82

5.11 Comparison of the best performing models with 128 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset 84

5.12 Comparison of the best performing models with 256 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset 85

6.1 Average validation loss of the default TCN models with error-bars indicating the standard deviation across the individual models 91

6.2 Error rate per HTML-tag in regards to TCN models' configuration. 92

6.3 Code coverage performance with 12KB of HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 93

6.4 Code coverage performance with 24KB of HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set 94

6.5 Comparison of the best performing models with 128 HTM-tags per file (12KB for real-world (RW) models). (left) total number of basic blocks achieved, (right) difference to the testset 97

6.6 Comparison of the best performing models with 256 HTM-tags per file (24KB for real-world (RW) models). (left) total number of basic blocks achieved, (right) difference to the testset 98

7.1 The overall setup for the reinforcement learning experiments. 101

7.2 The deep neural networks used to approximate the optimal Q-function. Here, x_1 to x_n is the input integer sequence representing the HTML string and c_1 to c_a the Q values of the available actions. 101

7.3 Code coverage performance results of the models trained during hyper-parameter search. Per configuration 5 different learning rates were tested. The violin plots per configuration are ordered by decreasing learning rate. 105

7.4 (left) DDQN agents' performance compared to the fuzzer baseline; (right) DDQN agents uniquely discovered basic blocks compared to the fuzzer baseline 105

7.5 (left) DDQN agents' uniquely discovered basic blocks compared to the best performing GRU model; (right) DDQN agents uniquely discovered basic blocks compared to the best performing TCN model 106

7.6 Policy similarity of the 15 trained DDQN agents per configuration. A lighter shade indicates a smaller Kullback Leibler Divergence [84] between the policies. 107

7.7 Comparison of the best-performing models with 128 HTM-tags per file (12KB for real-world (RW) and DDQN models). (left) total number of basic blocks achieved, (right) difference to the testset 109

Summary of Notation

- x : scalar real value
- $\mathbf{x}$: Vector (lowercase bold letter)
- $\mathbf{x} \cdot \mathbf{y}$: dot product of two vectors
- $\hat{\mathbf{y}}$: prediction output vector of a neural network
- $[\mathbf{x}, \mathbf{y}]$: Concatenation of vectors
- A : Matrix (uppercase letter)
- $A^\intercal$: Transpose of matrix A
- $A_{a,b}$: Value at position (a, b)
- $M_{i \times j}(\mathbb{N})$: The set of matrices with dimension $i \times j$ over the natural numbers
- $\{0, 1\}$: A set including 0 and 1
- $|\{0, 1\}|$: The cardinality of a set
- $[0, 1]$: An interval including the endpoints
- $\odot$: Element-wise multiplication

Chapter 1

Introduction

Web browsers are used to display web pages on digital devices. They also are capable of executing software programs locally, for example, JavaScript code. However, the focus here lies on rendering web pages written in the Hypertext Markup Language (HTML). HTML is a complex markup language. It consists of over 100 keywords declaring HTML-tags and enables the specialization of those tags with additional attributes, like styles or captions. In addition, it is even possible to define animation solely in HTML. The HTML5 standard [1] describes on over 1200 pages HTML5 and all its components in natural language. The available HTML tags and their attributes are described on over 540 pages. So, it provides the necessary documentation to implement HTML in a web browser, but it also provides a huge space for programming and specification errors due to its overall complexity. For example, it is possible to define many attributes into an HTML tag and nest it into other HTML tags, which also can be nested. This implies the construction of huge tree-like structures. The HTML document as a whole needs to be parsed, and the attributes need to be applied to the tags. Finally, the combination of all the tags and attributes needs to be displayed on the screen. The daily contact with web browsers and HTML highlights the importance of eliminating those programming errors.

To avoid serious flaws from being introduced, for example, through developer mistakes or unaccounted side effects, a plethora of software testing methods can be applied during different stages of the software development lifecycle, each with its advantages and disadvantages. It is out of the scope of this study to look into all those testing methods in great detail. However, the following two examples highlight the mentioned advantages and disadvantages.

First, the developer could write test cases to test the functionality of the written code. This approach is called unit testing. The advantage is that the developer should know the specification and the implementation in great detail. Therefore, they can write highly detailed test cases. On the other hand, it is important to strictly define the test boundaries and purpose as highlighted by Daka et al.[2]; otherwise, valuable development time is wasted by designing

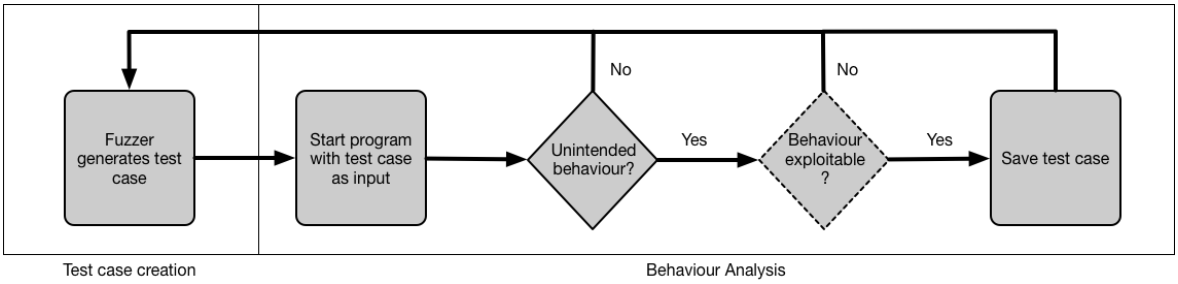

Figure 1.1: Workflow during fuzz testing

and implementing unnecessary test cases. In addition, critical areas, like user input handling sections, with serious flaws might be untested. In addition, unit testing introduces an additional workload for all developers by designing, implementing, and maintaining test cases (including checking for obsolete test cases and disabling them). It might also lead to a situation where newly implemented features are not tested because the developer relies on old test cases.

Secondly, static code analysis relies on analyzer programs that search for known bug patterns in the source code. The main advantage is that this approach can be applied directly to the source code and does not need the software under test to be executed. However, the problem here is to reduce the rate of false positives as described by Bessey et al.[3]. They highlighted how a high false-positives rate destroys the trust of the developer in the analyzer and eventually leads to the developer ignoring the results altogether. Furthermore, false negatives are also a problem in a twofold way after they are discovered during later development stages or even after the release. First, they further destroy the developers' confidence in the analyzer and, secondly, lead to the earlier mentioned possible loss of reputation and revenue.

An alternative approach to the methods mentioned above, which relied on either developer knowledge or the detection rate of predefined patterns is to detect bugs during execution. This has two advantages; first, it provides an example of how to trigger the bug, and second, it certainly is a bug. So, a developer can act on the test results and fix the cause of the bug. A prominent approach to finding bugs by generating test cases and observing the software's behavior is called fuzz testing (or fuzzing). The basic concept of fuzz testing is to present the software under a test with an unexpected (i.e., the developer has not strictly defined the test case) input and observe the software's behavior.

The general workflow during fuzz testing software is shown in Figure 1.1. It is subdivided into test case generation and behavior analysis. The whole process starts with generating test cases in the first phase. The software under test is then executed with this test case as an input (e.g., a document file for a word processor or a sequence of network frames for a FTP server). During the process execution, the process state is monitored, for example, with an attached debugger. Suppose the process state is different from any predefined expected

value. In that case, the test case is further examined to determine how likely exploitation of this unintended behavior is (see Section 2.1.2) to pre-categories the test case. In the case that those two are not successful, the process is restarted from the beginning otherwise the test case is saved for further examination before the process is restarted.

The main scope of this thesis lies in the test case generation and ways to improve the code coverage achieved by the generated test cases. In general, the approaches used to generate test cases fall into two categories [4], [5] (see Section 2.1.3):

1. mutation-based fuzzing
2. generation-based fuzzing

Mutation-based fuzzing needs a basic set of valid inputs, which are then modified by a given mutation function, for example replacing random positions with random values or flipping bits at random places. This technique can be implemented quickly if the necessary basic set is available. However, it creates a lot of malformed inputs, which cannot pass the input parser. For example, take a JPG image file. If the magic header value of this file is changed, the JPG rendering library will reject that file without even looking further into it[1]. In conclusion, many test cases might be rejected upfront; therefore, much time is wasted by executing the same code paths repeatedly.

In contrast, generation-based fuzzers create the test cases from a given rule set or grammar. This allows for fine-tuning the actual compliance with the underlying protocol. Furthermore, it enables the resulting test case to discover code paths that lie deeper in the program. However, the main problems during generator development are the effort needed to understand the underlying structure, which is amplified if there is no documentation available, and developing the generator based on that structure. A well-known grammar-based fuzzer is the CSmith compiler fuzzer [6]. CSmith uses a defined grammar to create C99 code and is 40,000+ lines of code C++ program. It highlights the complexity of valid code generation, i.e., code that is not rejected by the compiler but also highlights the necessity of fuzz testing. Yang et al. [6] reported 325 previously unknown bugs in the three years they worked on CSmith.

In addition, the generator properties (e.g., how many malformed bytes are introduced) have to be fine-tuned to increase the possible depth of penetration into the software. The depth is important to find bugs that are hidden deep in the execution graph.

This thesis introduces novel approaches to improve the test case generation during fuzz testing by applying *generative deep learning models* to the problem. First, it shows how an existing fuzzer can be improved by using its test cases as a base set for training. Secondly,

[1] if the header value is checked

an approach to select a well-performing trained model without knowing the code coverage performance is provided. This avoids time-consuming code coverage collection for under-performing models. Thirdly, the generative deep learning models are trained on a real-world data set, highlighting the possibilities arising through readily available input examples on the internet. Using real-world data can significantly shorten the test case generator development but also requires careful tuning to achieve good results. Finally, this study delivers a novel architecture to improve the code coverage of a trained generative deep learning model with little knowledge about the input structure and the means of reinforcement learning.

Overall, the goal is to show how fuzz testing can be improved by applying different deep-learning methods. It highlights the opportunities deep learning brings to fuzz testing and delivers the knowledge and first steps to a fully automated development of test case generators for fuzz testing scenarios.

1.1 Thesis Statement

Machine learning algorithms can be used to improve fuzz testing by reducing the time needed to develop test case generators. Additionally, they can help to guide the generator's output to trigger new code paths of the software under test. Those algorithms can learn from input examples and produce novel outputs, which can then be used to test software products with a higher degree of code coverage. In addition, performance data collected during the tests can be utilized to improve the generator's output further.

1.2 Research Questions

1.) Which machine learning models are suitable as HTML test case generators? (Section 5.4)

   a) What difference in code coverage does the choice of model and architecture make?

   b) What kind of model is adequate to learn the input structure of a program?

   c) How can a model's performance be predicted after the training process (i.e., How to choose a model)?

2.) How can the real-world HTML data be utilized to build HTML test case generators? (Section 6.6)

   a) How does the model perform with increasingly complex (e.g., overall grammar and vocabulary size) input data?

b) Which preprocessing methods are necessary, and how do they impact the performance?

3.) How can feedback from the web browser be utilized? (Section 7.7)

a) What data representation can be used?

b) Does code coverage provide a suitable feedback signal?

1.3 Research Contributions

This thesis provides novel ways to utilize deep learning models in a challenging fuzz testing scenario.

First, it provides a method for using a generative deep-learning model as a test case generator and estimating a trained model's performance in code coverage. This provides an approach to selecting a good-performing model before observing the code coverage, which is computationally intensive. In addition, it shows how an already existing fuzzer can benefit from training a model with its output as the training set.

Secondly, it highlights ways to use real-world data to shorten the initial development time for a test case generator. This is achieved by providing an example study where readily available real-world data is used to train a generative deep learning model as a test case generator. In addition, it highlights how the performance of those models is impacted by additional dataset preprocessing and the application of prior knowledge about the dataset.

Finally, it renders and evaluates an approach to utilize feedback in the form of code coverage to guide deep neural network-based test case generators to higher performance. Here, a framework is introduced and evaluated that uses reinforcement learning to improve the performance of a trained generative deep learning test case generator.

1.4 Thesis Structure

Chapter 2 surveys the general concepts of fuzzing and machine learning, which concludes with a review of research results in the area of combining machine learning and fuzz testing in Section 2.3.

Chapter 3 describes the common environment properties during the experiments. It introduces the software and hardware used for collecting the code coverage data and training the machine learning models. In addition, information about the used dataset in Chapters 4 and

5 is provided, including how it was generated and what modifications were necessary. It concludes by showing the results generated by a baseline test set and a naive mutation algorithm on that test set.

Chapters 4 and 5 are focussing on the first research question and evaluate different model architectures to provide answers.

In Chapter 4, the focus is on analyzing the effects of different recurrent neural network cells and architectures on the resulting HTML and, in conclusion, on the code coverage. It provides a detailed evaluation on how the architecture and overall complexity impact the performance and introduce an approach to estimate a model's performance before starting to collect code coverage data, which helps to select a model for test case generation.

Chapter 5 changes the underlying network to a non-recurrent neural network and analyzes the two former architectures in the new setting. This shows how the performance of overall less complex network architecture compares to the more complex recurrent approach. The contribution of this chapter is a direct comparison and evaluation of the results. Overall, these chapters further improve the model selection process and model design by providing additional inside in terms of training performance and resulting code coverage performance.

Chapter 6 changes the underlying data set from an artificially created and specially modified dataset to a real-world dataset. Therefore, it focuses on finding an answer to the second research question. The chapter focuses on solving the dependency on an already existing test case generator. It shows how the use of real-world data can speed up the development process of a fuzzer but also highlights the complications that arise from a real-world dataset in terms of preparation and output generated by the models.

By utilizing a well-performing generator model from the previous chapters, Chapter 7 evaluates an approach to improve the overall fuzz test results with the help of a feedback signal. It provides a framework to improve the results of a pre-trained generator using feedback from the running program. The chapter is dedicated to answering the third research question.

The thesis concludes in Chapter 8 by revisiting the research questions, which is followed by a summary of the key results and contributions. The last section provides an outlook into possible areas of future work and ends with the final remarks.

Chapter 2

Background and Related-Work

This chapter provides the necessary background information and introduces the relevant research advancements in the areas of fuzz testing, machine learning, and the application of machine learning during software testing. Those research advancements are summarized and critically evaluated.

First, Section 2.1 starts by introducing fuzz testing through its historical context. Then the two main parts of fuzz testing, namely the behavior analysis and test case creation, are introduced and research advancements are highlighted in Section 2.1.2 and Section 2.1.3 respectively.

Secondly, machine learning is introduced by a brief historical summary, followed by an introduction of the basic relevant architectures in terms of supervised learning. Section 2.2.4 provides a critical summary of research advancement regarding the model. In Section 2.2.5, additional approaches in terms of model design and use case are provided. A different approach to machine learning followed this, namely, reinforcement learning in Section 2.2.6. The machine learning section concludes with a brief overview of optimization techniques and a summary of the main techniques used during the following experiments.

Finally, the recent research developments in applying machine learning for software testing are introduced. Those developments are critically evaluated, and the existing gaps are highlighted to provide the foundation for the following chapters.

2.1 Software Fuzz Testing

This Section starts with introducing the history of fuzz testing. Following the categorization introduced in Chapter 1, which was also shown in Figure 1.1 first the advancements in test case creation techniques are summarized, and secondly, an introduction to behavior analysis is provided, followed by recent development in that area.

Overall it is essential to notice that fuzz testing is only one way of finding bugs in software. During software development stages, many different approaches are taken to find and fix bugs, for example, applying unit tests or formal verification methods. However, this research is about the combination of fuzz testing and deep learning, which implies a focus on fuzz testing.

2.1.1 History

In 1989 Miller et al.[7] created a program called *fuzz*, which marked the birth of fuzz testing, particularly generation-based fuzzing. They made the observation that during bad weather, the noise introduced to a modem connection was provoking UNIX tools to behave strangely or even crash. This observation was used in order to test a large palette of UNIX tools on different versions. The program mentioned above *fuzz* creates a random string of characters, which is used as input to a UNIX tool. Then the behavior of the tool under test is monitored to detect any strange behavior or crashes. They tested a total of 88 UNIX tools, from which 24% to 33% showed unintended behavior. Miller et al. defined the basic concept of fuzz testing. In addition, their paper also shows how a simple observation (here, random characters introduced by bad weather) can lead to an empirical technical study and, finally, to a widely adapted technique.

2.1.2 Behaviour Analysis

The security-related vulnerabilities can be classified by their usefulness for an attacker, for example:

1. a Denial of Service vulnerability enables an attacker to block legitimate users from actually using a software, this could be achieved by a specially crafted input that is not filtered correctly.

2. an Information Disclosure vulnerability enables an attacker to access sensitive information like personal information or password, this could be done via an SQL injection where an input is not escaped correctly

3. a Remote Code Execution vulnerability enables an attacker to execute arbitrary code on the target machine, which basically means taking over control; this could be done via a Buffer Overflow

In those three examples above, Remote Code Execution is the most useful vulnerability for an attacker. It grants total control of the targeted machine and, therefore, includes the capability to execute other attacks. Therefore it is the most severe class of vulnerability.

However, the main problem is to decide whether the unintended behavior during the execution of a test case leads to such a vulnerability. For example, take a test case that crashes a process due to an out-of-bounds exception, caused by memory access trying to overwrite a non-writeable region of memory. This particular exception might lead to an exploitable state only if other requirements are met, such as location (e.g., stack or heap), impact on branching decisions, or use during process execution in general. Nonetheless, since there is the possibility of an exploitable bug, the test case should be classified as exploitable and analyzed further during a manual test case evaluation.
Because of the non-deterministic relation between the test case and a possible vulnerability as highlighted by the above example, this approach can produce a lot of false positives. Microsoft [8] released a debugger plugin to rate the exploitability of crashing test cases. The classification itself still generates a lot of false positive test cases (and even false negatives[1]), but it's main contribution lies in providing a technique to identify differences in the state of crashing test case during execution in order to avoid duplicates. Advanced approaches like Yan et al. [9] try to predict the exploitability from prior knowledge. In particular, they used Bayesian reasoning in order to predict whether a program state is vulnerable to a certain exploiting technique.
Although the main focus above was on analyzing a test case crashing a program, the classification problem arising during fuzz testing is not a constraint to this particular kind of software testing. It is present during fuzz testing of all different kinds of applications, like web applications or industrial systems. It is always necessary to be able to make an observation in order to determine changes in behavior. For example, during fuzz testing a web application it can be essential to notice whether a user-provided input is triggering an error page or the default reply over the course of multiple test cases. Because it can make the difference between an exploitable behavior (e.g., a blind SQL-injection Halfond et al. [10]) or an non-exploitable one.

2.1.3 Test case creation

As mentioned in Chapter 1, fuzz testing or fuzzing is a software testing approach mainly used to find security-related bugs. Fuzz testing itself can be classified by the type of test case generation as Oehlert [5] described in 2005. He classified fuzzers into two categories:

1. generation-based fuzzing
2. mutation-based fuzzing

[1]The plugin is out of date and not maintained, resulting in newer exploiting techniques not being taken into account.

The difference between the two classes is that during generation-based fuzzing, the test cases are generated from scratch (e.g., through a provided grammar), whereas mutation-based test cases are derived from a basic set of mutated test cases (e.g., by replacing values with random values at certain positions). Developing a generation-based fuzzer requires deep knowledge of the file format or protocol specification of a software's input. Acquiring this knowledge and developing the fuzzer is a time-consuming task. In contrast, a mutation-based fuzzer can be developed quickly, and in many cases, a basic set can be acquired from the internet, for example, for JPG picture fuzzing (see Chapter 1. However, the mutation-based fuzzing approach produces a lot of test cases that are rejected in the early execution stages of the software under test. The main reason for that is simple mutation-based fuzzers are not aware of the underlying format specification (e.g., specific fixed values that must be set or otherwise a file is not loaded at all), which leads to the problem that simple rules (for example in the input parser) are able to detect the malformed input. The second problem directly relates to the chosen mutation basic set since this influences the overall performance of the fuzzer, e.g., if the basic set consists of very similar cases in terms of code coverage, the fuzzing process takes more time to discover new areas of the program compared to an already varying basic set. Generation-based fuzzers are able to penetrate faster into deeper layers of the software but in order to find bugs, a balance between introduced randomness to the test cases and complying with the rules have to be found.

The above-mentioned classification is still valid, although authors tried to introduce their approach as a new class of fuzzing, for example, Godefroid et al. [11]. They used symbolic execution and a constraint solver to mutate a valid test case to discover new code paths, which describes a subclass of mutation-based fuzzing. However, the results they provide are inconclusive because of the small sample size (as also mentioned by them) of seven different applications with ten-hour-long tests each. Nevertheless, the main results were that all of the found bugs were shallow in the code path and a well-formed input file seed discovered more bugs than bogus files.

Most research articles released focus on either very specialized applications for fuzzing or demonstrate how to transfer a fuzzing algorithm to a different application. As highlighted by Shapiro et al. [12] focused on fuzz testing SCADA application with little to no knowledge about the protocol itself by intercepting and mutating network packets before they reach the SCADA device calling this approach LZFuzz. The issue with that particular paper is that it is all about SCADA devices and claims the high effectiveness of their LZFuzz approach, but all the experimental results were gathered during testing with default network software, and they just claim to have good results with SCADA devices without any proof. In addition, they claimed to eliminate the need for a debugger attached to the target but did not provide any data about either false positives or false negatives.

A different research article by Coppit and Lian [13] introduced yagg (yet another generator

generator). This tool uses a grammar-based approach to generate the whole available input space. Sadly the efficiency is very low since generating the entire input space is a very time-consuming task and they also did not deliver any test data. In addition, they did not introduce any randomness to trigger edge cases, which are more likely to discover bugs in software.

The first article highlights that providing sufficient data for your claims and a sensible baseline to compare the introduced approach is essential. Furthermore, the second one shows the importance of breaking the underlying rules to discover code paths the developers have not thought about to trigger unindented behavior.

In addition, to papers introducing specialized fuzzers there are also broader application areas introduced. A very noteworthy one by Höschele and Zeller [14] provides a method to obtain input grammar from a program by observing the program during execution. This grammar can be used during generation-based fuzzing to create new test cases. This is an important step during fuzzer development because the development of an input grammar is a very time-consuming task for known protocols. In addition, it gets even more important for unknown protocols since reverse engineering a protocol is even more time-consuming then implementing it from available protocol specifications. The proposed AUTOGRAM approach provides a way to automate input grammar creation, but the process is still time and resource-consuming. Additionally, they only delivered results for simple input structures like URLs. Bastani et al. [15] followed a related approach by utilizing a set of valid input seeds and black-box access to the program to derive a grammar.

A mutation-based fuzzing approach developed by Rawat et al.[16] called VUzzer, uses an input set to generate a baseline of code coverage and tries to maximize the penetration depth of mutated inputs by utilizing the control flow graph in order to guide the mutation process. They use a genetic approach guiding the mutation by calculating the fitness of a particular test case. Whereby the fitness is influenced by the depth of penetration and purpose of the discovered basic blocks, e.g., a basic block responsible for error handling reduces the fitness, whereas a basic block in a task-related function increases it. The error handling basic blocks are identified heuristically with the help of the input set, which is assumed to not trigger any error handling. The main result of this paper is that the proposed approach is able to trigger more bugs in a less amount of test cases compared to a fuzzer not able to utilize such guidance. However, results and comparisons which only take triggered bugs into account have to be read carefully since there are many open questions regarding the configuration of the baseline fuzzers (e.g., used seeds or mutation chance). Those choices can have a huge impact on the fuzzer's performance.

In addition to academic research, there is a large number of different fuzzing tools with different approaches publicly available. One of the most famous tools is the Peach Fuzzing Framework [17], which can be configured via an XML file that describes the protocol or file

format that should be fuzzed. This particular tool is mutation based but uses the knowledge of the underlying format in order to reduce rejected test cases and therefore improve overall performance. The framework also includes a tool that determines the code coverage of the provided base set, which helps to reduce overlapping base cases. However, this tool lacks a code coverage-guided mutation approach.

Another widely used tool is American Fuzzy Lop (afl) [18]. This tool combines mutation-based fuzzing with feedback about code coverage. It uses a genetic algorithm in order to improve the code coverage of new mutated test cases. Still, this approach is based on a good base set since the algorithm is not able to recreate complex input grammars. The code coverage data is collected through instructions inserted during compile time. This makes it necessary to have the source code of the software, which is not always the case during security testing.

Another technique that can be applied to fuzz test structured data is "Probalisitc Grammar Fuzzing" [19]. This approach uses a grammar expansions combined with probabilities to generate new test cases. Adjustment of the probabilities can be used to influence the program areas that are tested. Overall, for this approach, a grammar needs to be available, and the probabilities need to be adjusted manually.

2.1.4 Summary Fuzz Testing

This section introduced the different parts of Fuzz Testing. The first part is the behavior analysis, which aims to determine the severity of found unintended behavior and classify it to take further action. The second part is the Test Case Creation, the two main classes of generators, namely mutation and generation-based ones. Further, it provided an overview of recently proposed methods to create test cases and a summary of available tools. The main issue highlighted with primarily the generation-based methods is deriving the grammar of the input structure in an efficient[2] way to develop a well-working[3] test case generator.

2.2 Machine Learning

According to Samuel [20] Machine Learning is the

> "field of study that gives computers the ability to learn without being explicitly programmed."

[2] time-saving and development labor-saving
[3] in terms of code coverage and found bugs

Here, this ability is going to be the creation of test cases without explicitly programmed grammar. A subclass of Machine Learning is Deep Learning which is linked to a certain set of algorithms focused on artificial neural networks. Since this research depends on the use of such algorithms, the next section briefly introduces Deep Learning from a historical point of view. Followed by Recurrent Neural Networks, because of their ability to memorize past input sequences and use this in the actual output calculation. This is in contrast to default feed-forward networks, which do not have states carried over time to memorize past inputs. In general, they compute

$$\hat{\mathbf{y}} = f(W \cdot \mathbf{x} + \mathbf{b}) \tag{2.1}$$

with $\hat{\mathbf{y}}$, f, W, $\mathbf{x}$, $\mathbf{b}$ being the prediction vector, a non-linear function, a weight matrix, an input vector, and a bias vector respectively. In a fuzzing environment, f in combination with learned weights W and bias b would be part of the test case generator creating a position in the test case $\hat{\mathbf{y}}$ while $\mathbf{x}$ represented the prior steps created by the model. The input $\mathbf{x}$ is a vector that provides the mathematical representation of an object or data point that is used to do the computation. For example, an image, a sequence of text, or a single data point of a time series providing weather information.

However, the Equation (2.1) shows that the prediction only depends on the actual input and the trained weights and biases. This leads to a problem while generating sequential data, which has dependencies on past inputs because no information from those past inputs is stored dynamically. Therefore they are not suitable for generating test cases where the outputs depend on past inputs. For example, network protocol traffic, where multiple packets could provide information about a device's capabilities, which are then used during test case generation or a closing tag in HTML depending on the fact that it was opened in the past. Nonetheless, the Equation (2.1) provides the foundation for all the concepts introduced in the following sections.

This chapter starts with a historical introduction to machine learning and deep learning, followed by supervised learning concepts. Hereby, supervised means that the algorithms are trained on data, which is labeled with the ground truth. The supervised concepts introduced are based on two different based architectures Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) in subsection 2.2.2 and subsection 2.2.3. Those two architectures extend the concept introduced by Equation 2.1. In subsection 2.2.4 different models are introduced and their advantages and disadvantages are highlighted. This is followed by a description of reinforcement learning, which is a different subset of machine learning algorithms than the aforementioned supervised learning concepts. The machine learning part is concluded with a brief introduction to some of the most used optimization algorithms.

Overall the goal of this section is to provide the necessary foundations and an overview of

the field. Especially the foundations are laid out to provide the background knowledge for the experiments conducted in later parts, starting in Chapter 4.

2.2.1 History

The first mathematical model of the nervous system and simultaneously of a neural network was given by Warren and Walter [21] in 1943. This mathematical model simplifies the procedures happening inside the nervous system. The neurone cells modelled are binary, which means the output of a cell can be either 0 or 1. Each cell can have multiple inputs and a threshold. The threshold determines how many inputs have to be active to output a 1. This model marks the birth of artificial neural networks.
In 1949 Donald O. Hebb [22] published his approach of formalising learning inside a neural network. Today it is known as Hebbian Theory, which says:

> ”When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.”

For artificial neural networks this is the oldest and simplest learning rule, which is interpreted as a weight change between two neurones.
After a first success using neural networks in a real-world application in 1959. Namely ADALINE, which removed echoes from analogue phone lines in real-time and was developed by the University of Stanford. The downfall came in 1969 with Minsky et al.[23] publishing their two assumptions, which have been rendered responsible for ceasing research funding. These stated that it is not possible to compute exclusive-or functions with neural networks and that there is not enough processing power for large neural networks.
A revival of interest came during the mid-1980s as Rummelhart et al.[24] described how the backpropagation algorithm could be used to train a multilayer perceptron in an effective way. The multilayer perceptron is a multilayer feedforward network which uses a nonlinear activation function. They demonstrated its ability to learn the XOR function, but besides some success using neural networks, for example, by AT&T Bell Labs for recognising handwritten digits, it was not possible to scale neural networks in order to solve larger problems mainly because of the lack of data, computing power or both.
The real breakthrough and hype of Deep Learning started in the later stage of the 21st century’s first decade as computing power increased more and large amounts of data became available. This enabled more and more areas in which Deep Learning could be used, like Natural Language Processing, Language Translation Tasks or Image Classification, to name only a few.

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are used to model sequential data over time as described in [25] and [26]. They use a hidden state as short-term memory which carries information between time steps. The conventional RNN with input $\mathbf{x_t}$ is defined through a hidden state $\mathbf{h_t}$ and an output $\hat{\mathbf{y}}_\mathbf{t}$ at time step t as follows

$$\mathbf{h_t} = f_h(\mathbf{x_t}, \mathbf{h_{t-1}}) \tag{2.2}$$

$$\hat{\mathbf{y}}_\mathbf{t} = f_o(\mathbf{h_t}), \tag{2.3}$$

with f_h and f_o being the hidden transformation and output function, respectively.
As described by Hochreiter [27] and later by Bengio et al. [28] those default RNNs suffer from the vanishing gradient (or exploding gradient) problem. This means that the weight updates become infinitesimal small during training which consumes a lot of time but does not lead to a better-trained network.
Hochreiter and Schmidhuber introduced the concept of Long-Short Term Memory (LSTM) cells [29]. RNNs using those cells do not suffer from the vanishing (exploding) gradient problem. LSTM cells use a hidden state $\mathbf{h_t}$, a candidate value $\mathbf{c_t}$ and three gates namely a forget gate $\mathbf{f_t}$, an input gate $\mathbf{i_t}$ and an output gate $\mathbf{o_t}$. The forget gate controls how much information from the former candidate value flows into $\mathbf{c_t}$, and the input gate controls the amount of information coming from the input into $\mathbf{c_t}$ so that

$$\mathbf{c_t} = \mathbf{f_t} \odot \mathbf{c_{t-1}} + \mathbf{i_t} \odot \tilde{\mathbf{c}}_\mathbf{t}. \tag{2.4}$$

In a similar way the output gate $\mathbf{o_t}$ controls information flowing from $\mathbf{c_t}$ into $\mathbf{h_t}$ with

$$\mathbf{h_t} = \mathbf{o_t} \odot tanh(\mathbf{c_t}). \tag{2.5}$$

Those gates are neural networks itself all having their weight matrices W and biases $\mathbf{b}$. The input for those networks is the concatenation of the former hidden state $\mathbf{h_{t-1}}$ with the actual input $\mathbf{x_t}$. The activation function used is the sigmoid function defined as

$$sigm(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

and the gates are defined as

$$\mathbf{f_t} = sigm(W_f \cdot [\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_f}), \tag{2.7}$$

$$\mathbf{i_t} = sigm(W_i \cdot [\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_i}), \tag{2.8}$$

$$\mathbf{o_t} = sigm(W_o \cdot [\mathbf{h_{t-1}}, \mathbf{x_t}] + \mathbf{b_o}). \tag{2.9}$$

Those LSTM cells are widely used and available for use in the major deep learning frameworks. They are also used in many papers for comparison reasons because of their effectiveness.

There are many applications for RNNs and they have been proven useful and performant for those. Sutskever et al. [25] experimented with RNNs and introduced the multiplicative RNN, which is an RNN cell with just two gates and not relying on the sigmoid function. They used the RNN as a generative model, which means they sampled new output from a learned corpus. For example, they used a sequence of characters extracted from Wikipedia stripped from XML as a training set. Since there is no real comparison baseline for that kind of generative model it is not possible to compare or score such a model. Nonetheless, the authors should have used other approaches like LSTM to determine whether there are differences in the quality of the resulting language model.
Another type of RNN cell was introduced by Cho et al. [30], called the Gated Recurrent Unit (GRU), which was motivated by the LSTM cell. It only uses two gates, an update gate $\mathbf{z_t}$ and a reset gate $\mathbf{r_t}$ defined as:

$$\mathbf{r_t} = sigm(W_r \cdot \mathbf{x_t} + U_r \cdot \mathbf{h_{t-1}}), \tag{2.10}$$

$$\mathbf{z_t} = sigm(W_z \cdot \mathbf{x_t} + U_z \cdot \mathbf{h_{t-1}}), \tag{2.11}$$

$$\mathbf{h_t} = \mathbf{z_t} \cdot \mathbf{h_{t-1}} + (\mathbf{1} - \mathbf{z_t}) \cdot tanh(W \cdot \mathbf{x_t} + U \cdot (\mathbf{r_t} \odot \mathbf{h_{t-1}}) \tag{2.12}$$

The advantage of the GRU cell is the lower amount of trainable parameters, but still being able to model sequential data and omitting the exploding gradient problem.

Chung et al.[31] proposed a the Gated Feedback RNN, which extends stacked RNN approaches by enabling the sharing of the hidden states of all layers with all layers during the transition from time step $t-1$ to t. This makes it possible to share memory over different layers. The influence of each layer is controlled by reset gates, which are comparable to the forget gates in LSTM.
The experiments conducted show that this model outperforms default Gated Recurrent Units (GRU) and LSTM in the task of language modeling and python evaluation, even with the same size of overall parameters. Overall this paper provides a detailed model and methodology description.
The greatest downside of all RNN cells is their sequential nature. Since every time step depends on the prior's final result in the form of the hidden state, RNN architecture takes a longer time to train and does not profit as much from being computed on a graphics card as other strongly parallelizable cells. In addition, all upcoming newly developed cells and models need to be studied carefully even when the publications suggest that those new meth-

ods outperform the old ones because many publications do not provide an evaluation of how hyperparameters were selected and tuned for the different models.

2.2.3 Convolutional Neural Networks

LeCun et al. [32] introduced the concept of Convolutional Neural Networks (CNNs) for classifying handwritten digits. They applied different concepts to build their network design. The main motivation was that default feed-forward neural networks were not applicable to image classification because they need a large number of trainable parameters (weights), which grow with the image size. In contrast, the convolutional architecture applies weight sharing in order to keep the number of trainable parameters comparably small. In addition, CNNs are able to detect specific features at different positions in images, whereas a default feed-forward neural network would need to learn those features for every position of an image.

The basic operation of a CNN is the multiplication of an input window with a kernel (weights). The size of the kernel can be chosen freely and determines the dimensions of the resulting feature map. This operation is then repeated until the end of the input sequence is reached with a stride parameter defining how many positions the input window is moved for each multiplication. In order to introduce non-linearity, the feature map is used as input for the $relu$ function introduced by Hahnloser et al. [33] defined as:

$$relu(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{otherwise} \end{cases} \tag{2.13}$$

After applying the $relu$ function to the feature map, the next steps depend on the general network design. For example, the feature map can be used in an additional convolutional layer or a max pooling layer follows, to reduce the dimensionality further. Max pooling also uses a sizable window from the input sequence and outputs the largest value in this window and therefore reduces the dimensionality of the input. After applying several stages of convolutional and max pooling layers, one or more fully connected are following with a final output layer. CNNs have shown great performance in, for example, image classification tasks (Szegedy et al.[34]) and facial recognition (Taigman et al.[35]). However, researchers have demonstrated that CNNs can also perform well in time series prediction tasks with performance close to the aforementioned RNNs, for example, Han et al. [36] demonstrated the use of CNNS for highway traffic flow prediction.

2.2.4 Model Architectures

This section introduces different model architectures based on the RNN and CNN architecture. The general structure of those designs is described, the provided research results are evaluated, and the general quality and the relevance for the following experiments is analyzed.

Sequence to Sequence Models

Cho et al. [30] proposed the RNN Encoder-Decoder model, which became known as the sequence to sequence model (seq2seq). The main idea is to take a variable length input sequence into an RNN, which calculates a fixed length vector representation of that input (encoder step). This vector is then used as input for the second RNN (decoder), which calculates a corresponding sequence to that vector. They used the formerly mentioned GRU cell for their experiments with the seq2seq approach. Those experiments showed that this model performs well in translating phrases from English to French. But again, there is no comparison with a network using LSTM cells, only with a default neural network without recurrent connections. Nonetheless, Cho et al. delivered a detailed description of the model and the experiments.

Bahdanau et al.[37] extended and improved the model proposed by [30]. They identified the single fixed-length vector, which is the encoder's output, as a bottleneck during performance improvement. To tackle this problem, they propose to extend the model by first using an RNN as an encoder that provides a combined hidden state calculated on the input sequence in forward and reverse order. So these hidden states include information of previous and following words. The decoder computes the context vector as a weighted sum of the hidden states. The output is a probability vector that gives insights into which words need to be used in the translation. To build the translation, a search algorithm is necessary. The results of the conducted experiments are an improvement over the prior approach, especially with longer sentences.

Another example of a seq2seq machine translation model was provided by Sutskever et al. [38]. Their proposed seq2seq model consists of two separate LSTM RNNs. Again one is used as an Encoder computing the input till a special stop character instead of a variable length sequence. The encoder's output is fed into the Decoder generating the output sequence again till a special stop character but this time, the character has to be predicted by the decoder. Besides outperforming several competing approaches, including the formerly mentioned approach by Cho et al. [30], they made some interesting observations. Firstly, deep LSTM networks outperform shallow ones, and more importantly, reversing the input order greatly improves translation performance.

The former two papers were published in a time frame of 3 months; this emphasises the fast pace and pressure to publish in the deep learning community, especially in areas with high commercial potential, like machine translation.

Generative Adversarial Networks

The basic approach of Generative Adversarial Networks (GANs) is to train a generator (G) in a two a player min-max game where a discriminator (D) decides who wins each round. It was introduced by Goodfellow et al. [39]. The discriminator evaluates whether the provided input is generated by G or from the real data/training set, and the generator's goal is to be misclassified as real data. This leads to following optimization problem:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[log(D(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[log(1-G(\mathbf{z}))] \tag{2.14}$$

with $\mathbf{x}$ being from the real data and $\mathbf{z}$ a vector with random values used as generator input. This training method has shown some impressive results, especially in generating pictures of all different kinds, for example, hotel rooms [40] or faces [41]. However, it is important to mention that GANs are difficult to train and that the whole training process is very unstable. The min-max approach can lead to conditions where either the generator or discriminator is too good and always wins. This brings the training to a standstill. Therefore it is essential to fine-tune the training process and decide when to train which part of the model. To ease this problem, Arjovsky et al. [42] introduced a concept called Wasserstein GANs (WGANs). Instead of trying to solve the problem described by Equation (2.14), the goal in WGANs is

$$\min_G \max_{D\in\mathbb{D}} \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[D(\mathbf{G}(\mathbf{z}))]. \tag{2.15}$$

This change results in eliminating balancing the training steps between generator and discriminator carefully. Furthermore, it allows always to train the discriminator optimally. The discriminator also provides a metric about the quality generated by the generator function, which is not the case for default GANs. This was further improved by Gulranji et al.[43] by introducing a gradient penalty to stabilize the training process, which avoids undesired behavior that can occur using weight clipping.

Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs), as defined by Bai et al. [44], were specifically designed for generative tasks on sequences. They modified the default convolutional layer (see Section 2.2.3) to combine it with additional concepts into one architecture. Namely, a TCN combines the concepts 1-dimensional fully convolutional network (1D-FCN) [45],

causal convolution, dilated convolution [46] and residual blocks [47]. Firstly, 1D-FCN uses padding to ensure that the hidden layers have the same length as the input layer, resulting in the output layer, also having the same length as the input layer. Secondly, the causal convolution ensures that the computation for a time step t only can access information i with $0 \leq i \leq t-1$ and therefore prevents utilizing knowledge from the future.

$$dConv(X) = (\mathbf{x} *_d f) = \sum_{i=0}^{k-1} f(i) \cdot \mathbf{x}_{s-d \cdot i} \tag{2.16}$$

$$norm(\mathbf{w}_{flat}) = \frac{g}{\|\mathbf{w}_{flat}\|} \cdot \mathbf{w}_{flat} \tag{2.17}$$

Thirdly, dilated convolution provides control over the past by setting a gap in the past steps, which are used for the actual computing step. This allows the further past to be taken into account at an advanced position. This effect is amplified by stacking several dilated convolutional layers with increasing dilation rates onto each other. Causal and dilated convolution are combined into one step shown in Equation (2.16), with X being the input sequence as a matrix, $\mathbf{x}$ a column vector from X, k the kernel size and f a filter. Fourthly, weight normalization is applied to the dilated causal convolution's output. Weight normalization is defined in Equation (2.17) with $\mathbf{w}_{flat}$ being the vector that results from concatenating all columns of the weight matrix W into a single vector. Weight normalization improves training stability and increases convergence speed, as demonstrated by Salimans et al. [48]. Fifthly, dropout Srivastava et al. [49] is applied to the weight normalization output. Dropout reduces overfitting during training by randomly dropping connections between neurons. This results in the following combination of functions for half a residual block:

$$half\_block(\mathbf{x}) = (dropout \circ relu \circ norm \circ dConv)(\mathbf{x}) \tag{2.18}$$

To form a residual block Equation (2.18) is applied twice to the input. Finally, the residual connection adds the block's input sequence to the block's output. This is very similar to the use of the gated connections as in Seq2Seq models Section 2.2.4, which connects the cell's input to the hidden state. Equation (2.19) shows the final residual block.

$$res\_block(\mathbf{x}) = \mathbf{x} + (half\_block \circ half\_block)(\mathbf{x}) \tag{2.19}$$

The main advantage of using a TCN is the ability to highly parallelize the training process and reduce the memory requirements. This is in comparison to using recurrent networks where the input must be processed sequentially because of interdependencies. Nonetheless, the TCN architecture also has downfalls, especially during the generation process. For ex-

ample, it is necessary to re-feed the whole sequence in each step while generating a sequence with this architecture because of the lack of a hidden state where an RNN can store and retrieve information about past inputs. This can become a bottleneck for very large sequences because those might not fit into memory, whereas RNNs only need one prior time step during sampling.

Memory Networks

Even though the formerly presented approaches can store and forget information in a hidden state, this might not be sufficient enough to capture dependency over a long period of steps. For example, in complex HTML documents, the script definitions might be separated from the HTML sections using the scripts by tens of kilobytes. Memory Networks introduced by [50] tackle this problem by introducing a long-term memory, which is consulted in order to create the output. They evaluate their concept with experimentation in a question-answering (QA) setting, where the network is trained on input-output examples to create the dedicated response. The experiments show that memory networks are able to outperform conventional and LSTM-based RNNs. Especially in question settings regarding the past (in the form of older sentences) and in combinatory ways. Nonetheless, there are some open issues about that approach regarding the memory replacement/update function. Especially how to decide to replace something. Sukhbaatar et al.[51] introduced End-to-End Memory Networks (MemN2N), which improved the above-mentioned memory networks to provide the possibility of end-to-end learning from input-output examples without supervision in the steps between. They achieve this goal by using a smooth function for the output calculations. This enables back-propagation through the network because the gradients can be easily computed. In the QA setting, this approach outperforms LSTM RNNs but is not able to achieve a better error rate than the above-mentioned Memory Networks. Nonetheless, research results in the area of memory networks do not provide an evaluation of how these methods perform in a generative setting. Furthermore, the problems during design regarding memory implementation and the functions necessary to use the memory are significant hurdles. The choice of memory access functions also decides whether the memory network can be trained in parallel or sequentially, for example, RNNs.

2.2.5 Other approaches

A different area of application is built around the problem of automated development. For example, finding a program for a given input-output definition (also known as Inductive Program Synthesis), which was done by Balog et al. [52]. First, they designed a simple Domain-Specific Language (DSL) that was loosely inspired by query languages. Secondly, they used

two simple feed-forward neural networks, one as an encoder, which takes input-output examples as input to provide a vector representing those examples. The other network takes this vector and predicts the probability with which each function of the DSL appears inside the source code corresponding to the input-output example. Finally, the given probabilities guide a search algorithm to find the source code representing the input-output example. This is quite a different and interesting approach compared to generating the source code from a given input-output example. Still, their results demonstrated that synthesizing programs for the simplest problems is only possible.
Zoph presented a related and interesting approach and Le [53]. They trained an RNN to generate neural network architecture to solve a task (e.g., image classification). Basically, the RNN generates the hyperparameters for a neural network till a fixed maximum size. The generated neural network is then trained and the achieved performance is used as feedback for the RNN to evaluate its error and train it. They describe the approach in great detail, which is an interesting research area. It is also highly related to the above problem of program synthesis since both approaches try to find a solution for a given problem in an automatic and generalized way.

2.2.6 Reinforcement Learning

The basic concept behind reinforcement learning differs from the former approach of using labeled data to train an algorithm. In reinforcement learning, an agent is trained to choose an action that provides the highest long-term reward. Hereby, the agent interacts with an environment with a defined set of action A and receives information about the environment states and a scalar reward value r. The environment is influenced by the agents action and provides the state information to the agent. For example, in the setting of fuzz testing, the feedback provided by a program's execution state (like code coverage and uncaught exception) could be used to compute a reward value. This value is then used to train an agent to predict the next action to maximize the accumulative reward.

Watkins et al. [54] introduced the iterative Q-learning algorithm, which builds the basis for many modern approaches to reinforcement learning. It provides an agent with the capability to learn how to maximize the reward in a finite Markov Decision Process (MDP). The Q-function takes a state, action tuple (s, a) as input and returns the expected discounted reward for executing action a in state s:

$$Q(s, a) = max_{\pi}\mathbb{E}[R_t|s_t = s, a_t = a, \pi] \tag{2.20}$$

with π being the policy followed for future steps. The objective for Q-learning is now to

approximate the Q-function for an optimal policy. The algorithm follows a basic pattern of

1. Select the action which maximizes the Q-value in the current state s_t: $a_t = max_{a \in A} Q_t(s_t, a)$
2. Get feedback included the new state s_{t+1} and the reward r_t earned
3. Update the value for Q_t using a learning factor α and a discount factor γ with $0 < \alpha, \gamma < 1$

Mnih et al. [55] presented an approach to using deep neural networks in order to approximate the Q-function described above. It follows the same steps described earlier but uses Experience Replay introduced by Long-Ji [56]. Experience replay basically builds up a memory of past transition, including the information of the previous state, the action taken, the new state, and the reward gained. Those data points are selected randomly during the training of the deep neural networks to smooth out the training distribution and remove the temporal component. So, the training can be formalized by minimizing the loss function

$$\mathcal{L}_t(\Theta_i) = \mathbb{E}_{s,a \sim p(\cdot)}[(y_t - Q(s, a; \Theta_t))^2], \tag{2.21}$$

with y_t being the expected discounted reward using a former set of network parameters Θ_i if s_{t+1} is not a terminal state otherwise y_t equals r_t resulting in

$$y_t = \begin{cases} \mathbb{E}_{s_{t+1} \sim \mathcal{E}}[r_t + \gamma \max_{a'}(s_{t+1}, a'; \Theta_i)] \text{ if } s_{t+1} \text{ is a non-terminal state,} \\ r_t \text{ else,} \end{cases} \tag{2.22}$$

with $\mathcal{E}$ being the experience memory. This means the approach actually uses two deep neural networks, one called the estimator network used to determine the actual action to take in a state and one called the target network used during training. After a set number of time steps, the network parameters of the estimator network are copied to the target network and training continues. They used this approach to train a network to play six Atari games. The resulting was able to surpass human performance in three of those. Mnih et al. extended their work to 49 Atari games in [57], where they also provided more information about the implementation of the algorithm and introduced the term Deep Q-Network (DQN) for their approach.

The main problem with DQNs is the tendency to overestimate the action values, as Hasselt et al. [58] highlighted. In order to tackle this problem, they transfer a prior approach called double Q-learning introduced by Hasselt [59] from a tabular setting into a deep learning setting.

2.2.7 Optimisation

In general, learning takes place during the optimization process. This is when the model is trained by minimizing a loss function via an optimization algorithm. The optimization algorithm determines how a model's trainable parameters (i.e., weights) have to be changed to minimize the given loss function, which leads to an approximation of the real function from domain A to the target domain B.

There are several different algorithms used to minimize the loss function, for example, Stochastic Gradient Descent (SGD), Adaptive Gradient algorithm (AdaGrad) [60] or Root Mean Square Propagation (RMSProp) [61]. In this thesis, the Adaptive Moment Estimation (ADAM) [62] optimization algorithm is used. It is a gradient-based algorithm, which means the updated decision regarding the next step is made based on the computed gradient with the parameter update rule:

$$\Theta_t = \Theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.23}$$

First, Θ_t describes the new model parameters and Θ_{t-1} the parameters of the previous step. Secondly, α is the pre-set learning rate controlling how large the parameter update steps are. Thirdly, $\hat{m}_t$ is the bias-corrected first moment estimated defined as:

$$\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)} \tag{2.24}$$

where β_1^t is the exponential decay rate for the first moment estimates at time step t and m_t the biased first-moment estimate defined as:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t, \text{with} m_0 = 0. \tag{2.25}$$

Finally, $\hat{v}_t$ is the bias-corrected second raw moment estimate which is defined as:

$$\hat{v}_t = \frac{v_t}{(1 - \beta_2^t)} \tag{2.26}$$

here β_2^t is the exponential decay rate for the second raw moment estimate, and v_t is the biased second raw moment defined as:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2, \text{with} v_0 = 0. \tag{2.27}$$

In Equation (2.25) and Equation (2.27) g_t is the gradient at time step t based on the parameters Θ_{t-1}.

2.2.8 Summary of Machine Learning

In Section 2.2 the background information for machine learning was provided and followed up by recent research advancements. In general, two categories of machine learning algorithms were introduced, namely supervised and reinforcement learning.

First, during the supervised learning part, the focus was on research advancements regarding generative models. Those models are important for the first set of experiments utilizing supervised generative models to train them as a HTML test case generator starting in Chapter 4.

Secondly, while introducing reinforcement learning, the focus was on algorithms that are capable of performing well in environments with large state and action spaces. This was necessary because software programs can be very complex and therefore have many different code paths that can be executed. Eventually, it depends on the design and implementation of reinforcement learning environment and agent to provide and utilize the available information respectively. The experiments utilizing reinforcement learning are described in Chapter 7.

The following chapter will introduce the basic environment of the experiments, including the performance metrics applied and datasets used later on. Furthermore, the design choices made are described, and justifications are provided.

2.3 Machine Learning and Fuzz Testing

The first evaluation of the possible use of deep learning algorithms for fuzz testing was presented by Godefroid et al. [63]. They used a training set containing $63,000$ PDF objects to train an LSTM based stacked RNN to generate PDF objects. Those PDF objects were then inserted in three different host files and executed with Edge[4]'s PDF rendering engine. The execution was instrumented to collect the uniquely executed instructions. The collected coverage data was then compared to a baseline, which was generated by executing $1,000$ randomly selected PDF objects in those three hosts. There are a few flaws in this paper. First, they highlight the complexity of the PDF format, and in the next section, they explain how they stripped away almost all complexity by restricting it to non-binary PDF objects. Furthermore, they claim that those kinds of objects are responsible for the bulk of the $1,300$ pages long PDF standard without providing any evidence. Secondly, they claim that the epochs of training are one of the most important parameters, but without providing any metric from the training itself. It is impossible to judge whether or not the used model ever reached convergence or even overfitted. Thirdly, adding to the possibility of overfitting,

[4]successor of the Microsoft Internet Explorer

there is no information about the use of validation data. Lastly, the lack of information provided regarding the model architecture and the parameters makes it impossible to recreate the model and therefore reproduce the research. Nonetheless, they identified the research potential in combining learning fuzzing in order to ”explore how to learn, as automatically as possible, the higher-level hierarchical structure of PDF documents [...]”. In addition, they also mentioned the possible use of reinforcement to improve the learning process with feedback from the application as an area of interest and also introduced the SampleFuzz algorithm. It uses the smallest value of a probability distribution if and only if the highest value is greater than a provided threshold and a random coin toss value equals one. This was necessary because of an observed tension between learning the structure and breaking it for fuzz testing, but as mentioned earlier, it is not possible to judge if this tension is only observed due to overfitting.

Lee et al. [64] introduced Montage, a JavaScript fuzzer based on a trained LSTM. They propose an interesting approach of using slices of an Abstract Syntax Tree (AST) as input to the LSTM and predict the next slice. The LSTM uses a hidden state of 32 but is not further detailed in the paper. Furthermore, there is no information about the size of the dataset only that the authors collected it from JavaScript regression tests. The papers main claim is that Montage is able to generate more unique crashes than state-of-the-art fuzzers. However, this only holds for the single JavaScript engine they analyzed and most importantly only on the debug build. Sadly, the authors miss the opportunity to provide details to why this is behind the assumption of additional asserts. So, it can well be that those crashes are purely caused by assert statements due to the JavaScript code quality and not because of the test cases themselves. In addition, they don’t provide any information about the code coverage and only provided a single model architecture. Finally, the authors did not publish the used code and therefore it is not possible to reproduce the results.

A way to improve mutation-based fuzzing was described by Böttinger et al.[65]. They utilized deep Q-learning to determine the best mutation actions to maximize a reward. The rewards analyzed were code coverage execution time and a combination of both. The set of available actions is not clearly defined. Therefore, it is difficult to judge the quality of the baseline chosen during evaluation, which is defined by evaluating the reward value of randomly chosen actions where each action has the same probability at any state. The additionally provided baseline are only comparing the models with themselves (e.g., use of replay memory or activation function). This results in no independent baseline, and results are only briefly provided in the form of a percentage describing reward gain. A better-suited baseline might have been their seed input used to analyze whether the reinforcement learning approach is able to provide any improvement. Overall the results are not detailed, and the final claim that their learning approach learns the grammar of the underlying input structure cannot be reconstructed with the reasoning provided. Especially the policy π cannot be seen

as the input grammar with so many information missing about the available actions in the defined action classes, for example using the depth of the code coverage as a reward might as well lead to replacing everything with valid tokens from the provided dictionary instead of learning a grammar.

Fan et al. [66] proposed a seq2seq model based on LSTM cells to learn proprietary network protocol messages from captures. However, they only used a single model type and did not provide a detailed explanation of their training methods. Furthermore, they did not provide information about the training, so graphs for the error rates and training progress were provided. This makes it difficult to reproduce any results. During the evaluation, essential details, like the definition of their code coverage baseline, are missing.

Cummins et al.[67] introduced a method using a generative deep learning model to fuzz test OpenCL-based compilers. Their approach was able to improve fuzz testing compilers and uncover previously unknown bugs. The generative model was trained on a corpus of open-source OpenCL programs that were preprocessed, removing user-defined variable names and other non-functional naming decisions to make training easier. The generative model itself was an LSTM (Section 2.2.2) based network. In order to generate valid code, the model is guided by a synthesizer part that takes care of adhering to syntactical rules like the matching amount of brackets. A further example of compiler fuzzing was presented by Liu et al.[68]. They use an LSTM seq2seq-based model to generate C source code. The best compiler pass rate of the generated source code is achieved by always taking the maximum from the probability distribution outputted by the model. This leads to the problem that the output directly depends on the input. So, the same seed input always leads to the same output (the same source code). Furthermore, the authors did not mention any results with regard to model training, like losses or training time. The is also no information about the size of the dataset in terms of examples. They only mention that 10,000 well-formed C programs from the GCC test suite were used. The comparison with Csmith [5] is interesting but also lacks details. For example, they compare 10,000 programs with each other where their model uses a standard sampling method. It would have been interesting to see how many programs were sorted out because they failed compilation completely. This would have provided additional insights into the performance and practicality of the proposed approach. Xu et al. [69] proposed TSmith an LSTM-based encoder-decoder approach that uses incomplete ASTs as input and predicts the complete AST. The authors provide a detailed analysis of the generated test cases including code coverage and pass rate of the test cases. However, some important points are missing. Firstly, a description of the data used for training and the source. Secondly, the strategy naming is implicit so it is not clear which strategy number refers to which strategy. Finally, the authors did not provide any comparison to other approaches like Csmith.

[5] state of the art C compiler fuzzer [6]

Another approach is to derive the grammar of an input by using active learning. It was introduced by Höschele et al. [70]. Active learning means that the used machine learning algorithm needs an input from outside to retrieve the correct label for example, with statistical models as Cohn et al. [71] demonstrated. The learned grammar could be used in different settings like reverse engineering or indeed fuzzing. However, the research focussed on how to derive as complete as possible grammar and not the utilization of that grammar.

Wang et al. [72] provided a systemic review of fuzzing techniques utilizing machine learning. They classified the techniques into five categories:

1. Seed file generation
2. Testcase generation
3. Testcase filtering
4. Mutation operator selection
5. Exploitability analysis

The closest categories to this thesis are test case generation and mutation operator selection. Furthermore, the thesis focuses on file formats, and the papers discussed earlier in this section also overlap with this categorization. Finally, the review provides insight into the different approaches and categories. The closely related publications found in the review were also discussed earlier in this section.

2.4 Summary

This chapter provided an introduction and overview for the areas of fuzz testing and machine learning in Section 2.1 and Section 2.2 respectively. The main problem while developing a test case generator is the effort needed to implement the ruleset for a given input specification, which can be further complicated if this specification is not publicly available or not documented at all. In addition, finding the balance between adhering to the rules and breaking them is very important to discover bugs, which are located deep in the execution path.

Machine learning has a proven track record of being able to detect underlying structures in input data in order to reproduce those as was highlighted in Section 2.2. The goal of the experiments conducted starting in Chapter 4 is designed to show the transferability of the generative machine learning abilities to the field of fuzz testing without repeating the shortcomings of prior research in this area identified in Section 2.3.

In order to tackle the balancing problem, Chapter 7 applies the introduced methods of reinforcement learning (see Section 2.2.6) to a fuzz testing environment. The applied approach utilized the feedback in form of code coverage in order to guide the training process to discover more basic blocks inside the software.

Chapter 3

Experimental Environment

This chapter introduces the common parts of the environment used during the following chapters in this thesis. First the hardware and software used for training the learning algorithms and collecting the code coverage data are described in Section 3.1. In particular the basic setup is introduced and the design choices made are explained.

Secondly, the dataset used for training the models during the experiments in Chapters 4 and 5 is introduced in Section 3.2. This section describes how the dataset was created and what steps were necessary to use it during the training of the learning algorithms.

Thirdly, the data collection process is described in Section 3.3. After introducing the collection process and the software used, this section describes how the three baselines for evaluating the trained models' performance during the experiments were established.

Fourthly, the code coverage results from the baselines are introduced to provide the necessary details. Those details serve to put the following chapters' results into context.

Finally, this chapter concludes by summarizing the information provided and links to the following chapters.

3.1 Environmental Setup

All deep learning models were trained on physical systems with 64-bit versions[1] of Ubuntu Linux. Those systems were equipped with different graphic cards, which were utilized during the training process. The parallel processing power provided by a graphics card speeds up the training process of deep learning models. All models were implemented in Python with the use of Google's TensorFlow framework [73], which provides all the necessary components to build deep learning models, including different types of neural network cells,

[1] 18.04 and 20.4 LTS

optimization algorithms, and loss functions. Besides providing a very low-level interface for defining computational graphs, TensorFlow also provides a higher-level interface for adding commonly used machine learning features, like feed-forward neural networks or RNNs, and it automatically utilizes available GPUs. In addition, models created on machines utilizing a GPU for computation are still transferable and, more importantly, usable on machines with no general-purpose GPU computation capabilities.

The code coverage collection occurred on Virtual Machines (VMs) running Ubuntu Linux 18.04 64bit and Firefox 57.0.1. This version of Firefox introduced the so-called headless mode. The mode does not display the graphical user interface (GUI) and therefore needs fewer resources and starts quicker. The VMs utilized 8 GB of RAM and ran on VMWares' ESXi hypervisor software. One advantage of running VMs on a dedicated hypervisor compared to a consumer operating system (OS) like Windows or Linux is omitting all additional introduced overhead from the consumer OS. The choice for a VM itself allowed the quick duplication (cloning) of the VM, which enables the parallel collection of code coverage data on the same host computer and multiple ones. The data collection itself was performed by DynamoRIO's drcov tool [74]. It collects the triggered basic blocks during program execution and saves this information in a log file for each process involved in the program execution (see Section 3.3 for more details). An alternative to DynamoRIO's drcov is Intel's PIN tool [75] however, a series of upfront tests have shown a lower performance in terms the execution speed.

3.2 Dataset Creation

The basic dataset used for training, validation, and comparison was generated by PyFuzz2 [76], a multi-purpose fuzzer, that can be extended with new test case generators and already included generators for HTML and JavaScript. This enabled the creation of a dataset with a configurable size and influenced the structure and complexity of the dataset. It allowed conducting experiments with fixed complexity and to establish baselines that were directly comparable with the trained models' results.

The PyFuzz2 HTML-fuzzer component was configured to create a single 36MB large file containing a single complete HTML-tag per line, which includes an opening HTML tag, attributes with their corresponding values, an inner value and the closing HTML-tag corresponding to the opening one (see Listing 3.1). The dataset file contained $409,000$ HTML tags in total and was 36MB in size. In addition, the fuzzer was configured to avoid the use of nested tags[2], which would add complexity to the problem. Since a learning algorithm would have to keep track of all opened and closed HTML tags to create correct HTML.

[2]HTML-tags encapsulated by another HTML-tags

Furthermore, it enabled the creation of test cases on a per HTML-tag basis. This made supported a fair comparison based on the number of HTML-tags per test case. However, this also means that not all available HTML tags were used during training. For example, ”td” or ”th” were excluded since they must be encapsulated by a ”table” HTML tag. In addition to the two already mentioned HTML tags, a further 15 tags were excluded, including ”script” and ”noscript”. This was necessary to avoid additional languages (like JavaScript) in the training set or because of the functional restrictions of the fuzzer. Additional programming languages like JavaScript would have added more complexity to the problem and the model size. Furthermore, compared to HTML engines JavaScript engines are stricter in terms of syntax enforcement that a single wrong character can prevent the execution of a whole script. Therefore, a dedicated JavaScript generator would be more appropriate especially combined with the advancements in program synthesis, for example, highlighted in Austin et al. [77].

The occurrence of included HTML tags was distributed evenly over all those tags (see Appendix A.1 for a full overview). This provided a smooth distribution of HTML tags for training the generative models by configuring the fuzzer to select a random HTML tag where each HTML tag had a probability of $1/n$ to be chosen, with n being the total number of HTML tags. A biased training data distribution also leads to a biased output which was observed in Chapter 6 and added the need for additional preprocessing steps.

The final step regarding the dataset preparation was to convert the characters into positive integers, including zero. This was necessary to use them as input values during model training because machine learning models need a mathematical representation of the text-based input. The values for the mapping were set on a first come first served basis regarding the occurrence of the character in the dataset, so the first occurrence of a character set the integer value used for the character in the whole dataset. For example, see Listing 3.1 the first character is $<$ which is converted to 0, the second to 1, and so on. When a character is encountered again the value of the character is looked up and put into the sequence. This resulted in a vocabulary size of 107 for the fuzzer-created dataset. The resulting integer sequence was saved and reused during training to avoid repeating the whole conversion process for every model training.

```
1 <textarea id="id18" lang="uz" style="style" accesskey=":" class="
    style_class_0"> -7500000000</textarea>
```

Listing 3.1: Example HTML-tag from the dataset

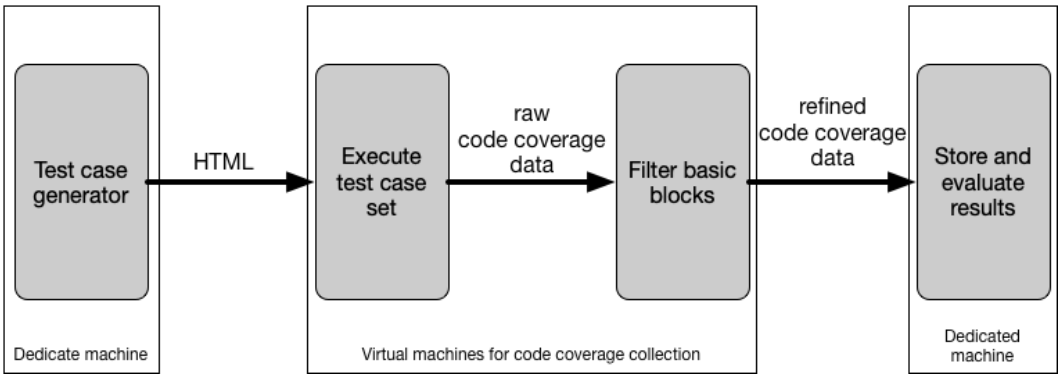

Figure 3.1: General code coverage collection process

3.3 Data Collection

In Figure 3.1 a general overview of the data collection process is provided. It highlights the separation between generating the test cases, executing a set of test cases as well as storing and evaluating the results. The code coverage data was collected with DynomaRIO's drcov tool at a basic block level on a VM as described in section 3.1. The tool was used to dynamically instrument Firefox during execution. Dynamical instrumentation means the process of inserting additional logic into a binary executable file by locating the appropriate location defined in the instrumentation tool and inserting the opcodes for the corresponding assembler instructions. As a result, drcov generated files containing information about all executed basic blocks in all processes spawned by Firefox. Those files contained an overview of the memory mappings of the different program parts, as shown in Listing 3.2. The memory mappings overview contained a unique assigned number to the related part, the start of the memory region, its size, and a filename corresponding to the mapping. This helped to identify the interesting basic blocks (i.e. which were in a targeted library or area of the program). Listing 3.3 show an example for the basic blocks area of the generated drcov file. It includes an offset into the library, which can be used as a unique identifier of that particular basic block. This identifier is also still valid if Address Space Layout Randomization (ASLR) is active. ASLR loads program parts at different memory locations between program executions, but it loads each part as a block into memory. This means the offset into a library keeps stable for the same library version.

The generated code coverage files were then parsed in order to filter out all basic blocks which are not part of Firefox's libxul.so. This library contains the whole web engine responsible for rendering HTML, CSS, and executing JavaScript. It also contains various other functionality (e.g., update routines and embedded search functionalities), and first experiments have shown that the number of executed basic can vary greatly between executions of the same test case. This indicated the necessity of a zero line for a sound comparison of the different trained models.

```
1  DRCOV VERSION: 2
2  DRCOV FLAVOR: drcov-64
3  Module Table: version 2, count 168
4  Columns: id, base, end, entry, path
5  0, 0x0000000000400000, 0x0000000000421000, 0x0000000000405220, /home/
       user/firefox/bin/firefox
6  1, 0x0000000072000000, 0x0000000072003000, 0x0000000000000000, /home/
       user/dynamoRIO/tools/lib64/release/libdrcov.so
7  2, 0x0000563d7ebac000, 0x0000563d7ed87000, 0x0000563d7ecc3ec6, /home/
       user/dynamoRIO/lib64/release/libdynamorio.so
8  3, 0x00007feaa1b48000, 0x00007feaa1d70000, 0x00007feaa1b48c30, /lib/
       x86_64-linux-gnu/ld-2.23.so
9  ...
10 90, 0x00007fea9223e000, 0x00007fea92465000, 0x00007fea92247bb0, /usr/
       lib/x86_64-linux-gnu/libdbus-glib-1.so.2.3.3
11 91, 0x00007fea9248a000, 0x00007fea96fc4000, 0x00007fea9248a000, /home
       /user/firefox/toolkit/library/libxul.so
12 92, 0x00007fea90900000, 0x00007fea90b04000, 0x00007fea90900700, /usr/
       lib/x86_64-linux-gnu/gconv/UTF-16.so
13 ...
```

Listing 3.2: Excerpt from drcov memory mapping section

```
1  module id, start, size:
2  ...
3  module[  3]: 0x0000000000000c30,    8
4  module[ 89]: 0x0000000000000660,    2
5  module[ 90]: 0x0000000000008628,   16
6  module[ 90]: 0x000000000000863d,    5
7  module[ 90]: 0x0000000000009ca0,   13
8  module[ 90]: 0x0000000000009cad,    5
9  module[ 90]: 0x0000000000009c10,   40
10 module[ 90]: 0x0000000000009c50,    2
11 module[ 91]: 0x00000000009b2880,   16
12 ...
```

Listing 3.3: Excerpt from a drcov basic blocks section

The zero line was created by first executing the HTML template shown in Listing 3.4 directly as a file from the local disc for as long as new basic blocks were triggered in a period of 128 executions. Secondly, the same template was executed again, but this time loaded via HTTP from a locally running web server until no new basic blocks were triggered over a period of 128 executions.

The second and main baseline for comparison was created by executing fuzzer-generated cases. The fuzzer was used to generate an additional six times 16384 HTML-tags. Each of this six additional HTML-tag collections was used to build two datasets, the first one containing 128 files with 128 HTML-tags each and the second one containing 64 files with 256 HTML-tags each. This approach provided additional certainty in the comparison and avoided falling for a very low or high outlier result of the dataset. Additionally, each dataset contained the same amount of HTML tags as the generated sets used later to ensure fairness in terms of absolute numbers and were approximately 12KB and 24KB per file in size for the 128 and 256 HTML-tags settings, respectively.

Finally, a third baseline for comparison was established by using a naive mutation strategy. This mutation strategy replaced characters in the dataset's test cases with randomly chosen characters from all appearing characters overall. The replacement probability varied incrementally from $2^0/10\ \%\ (0.1\%)$ up to $2^9/10\ \%(51.2\%)$. This comparison line was introduced to analyze whether a trained model is able to perform better than a quick-to-implement naive mutation strategy.

The trained models (described in the following sections) were used to generate 16384 HTML tags each, which were also used to create two test sets differing only in the amount tags inserted into the template as for the main baseline.

The collected code coverage data was used to generate a set of triggered basic blocks (identified by their offset into the libxul.so library) per dataset, for example, a single fuzzer-generated baseline contained one set of basic blocks for the 128 HTML-tags setting and a second one for the 256 HTML-tags setting. Those individual sets of basic blocks were

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title> Learned HTML </title>
5  <meta http-equiv="Cache-control" content="no-cache"></meta>
6  </head>
7
8  <body id="doc_body">
9  HTML_BODY
10 </body>
11 </html>
```

Listing 3.4: HTML-template used as host document

created by computing the union of all basic blocks in one of the settings.

All code coverage collections were controlled by a Python script. The script took the test case directory and targeted software as input parameters. The test case directory was scanned for input files, and those were then used as input for the targeted software. In order to reduce the time necessary for executing a single test case, the collection took place in a separate process, and the main script process regularly checked that the code coverage collection process is still using the CPU and enforcing a maximum of three minutes of execution time. It also took care of cleaning up so-called zombie processes[3], which appeared due to the non-conventional termination of the processes involved, here non-conventional means the processes were terminated by sending a ”KILL” signal and therefore forcing the termination instead of triggering the designated exit procedures of Firefox.

In order to allow multiple Firefox instances in parallel, it was also necessary to provide a unique profile for each launched instance. Those profiles were adjusted to disable functionality that would change the code coverage result or prevent the automatic collection. In particular, the various functions for updating Firefox and its plugins and the recovery methods that would, for example, force it to start Firefox in Safe Mode, which breaks the automatic data collection. The aforementioned termination of Firefox also led to compromised profiles. Therefore as soon as such a profile was detected, it was necessary to replace it with a backed-up version in order to continue the code coverage collection. The main observation made to identify defunct profiles was a steep drop in triggered basic blocks, i.e., a functional profile triggers multiple tens of thousands of basic blocks, whereas a malfunctioning one only triggers up to five thousand basic blocks. Furthermore, additional command line parameters were necessary. First, the ”no-remote” flag implied that a new browser instance was created for each program start, and no details were reused. Secondly, the aforementioned ”headless” flag (see Section 3.1) to avoid more resources being used and problems with multiple GUIs being displayed at the same time, which could corrupt the code coverage results and reduce the number of parallel running instances by wasting resources.

3.4 Baseline Results

In total, there are $1,885,784$ basic blocks in Firefox’s libxul. After running the empty test case on all VMs, the set union of those test runs resulted in $458,923$ basic blocks, which are not related to different inputs. For example, those program areas take care of reading the input file to memory or establishing the necessary network connections to receive the input in the first place.

[3] abandoned child processes that are still running

Dataset:	1	2	3	4	5	6	Avg.
#bb 128:	48,907	50,360	48,694	48,767	49,324	53,822	49,979
#bb 256:	47,066	50,512	48,170	48,846	49,329	54,263	49,698

Table 3.1: Basic blocks discovered by the different datasets

Mutation %:	1.6	3.2	6.4	12.8	25.6	51.2
Basic Blocks 128 HTML-tags:	53,105	52,793	50,126	39,998	20,665	11,779
Basic Blocks 256 HTML-tags:	52,929	53,103	52,044	33,403	27,328	9,683

Table 3.2: Mutation set performance

The fuzzer-created dataset cases achieved a maximum of $53,822$ and $54,263$ basic blocks for the settings with 128 and 256 HTML tags, respectively, as shown in Table 3.1. The minimum was $48,767$ in the 128 HTML-tags setting and $47,066$ in the 256 HTML-tags setting. Therefore, those values form the dataset coverage area, which is the target area the models should reach or exceed during the experiments. The dataset coverage area forms the first baseline.

The values for the second baselines are provided in Table 3.2. The overall best-performing dataset achieved those basic blocks (see model number one in Table 3.1). The idea behind the mutation is to swap characters for random ones with a fixed mutation rate. Six different mutation rates were evaluated as shown in Table 3.2. For example, if the mutation rate is 25.6%, there is a 25.6% chance for each character in the sequence to be replaced by a random character.

It is essential to notice how steep the decline in basic blocks is after passing a mutation chance of 10% (see Table 3.2). In addition, the overlap between the underlying dataset one and the mutation sets with a chance of 1.6% was 87.6% during the 128 HTML-tags setting and 86.2% in the 256 HTML-tags setting. The overlap was still $\approx 86\%$ in both settings for the 3.2% mutation chance and also above 80% for the 6.4% chance also in both settings. This highlights how close the code coverage was even with random characters introduced and the inability of the mutation-based approaches to improve the overall code coverage in this setting.

3.5 Summary

In this chapter, the hardware and software setup was introduced, which was used during the following experiments. The main points are that GPUs were used to accelerate model training and that a VM was created to parallelize the collection of code coverage data.

Furthermore, a detailed description of how the basic dataset was created with the help of a

fuzzer to reduce the initial complexity for the initial experiments with different architectures in Chapters 4 and 5. Chapter 6 used a separate dataset to evaluate the transferability to a real-world-based dataset, which will be introduced there.

Section 3.3 introduced how the data was collected and described the three baselines used to evaluate the trained generative models used during all following experiments. In addition, the ideas behind the collection scripts were introduced. It is important to remember that it was necessary to use a version of Firefox supporting headless mode and multiple profiles to enable parallel data collection since a single test case took up to three minutes to finish.

Finally, the results in terms of code coverage for the three baselines were provided. This is important for the following experiments to have an intuition about the result values supplied during the experiments.

Chapter 4

Effects of different RNN based architectures on the Code Coverage Performance

This chapter introduces and analyzes model architectures, which utilize recurrent cells. It provides the initial data to identify suitable machine learning algorithms to generate HTML based test case (see the first research question in Section 1.2). Arguably for the first time, this chapter shows the use of RNN based model architectures for generating HTML test cases in a fuzz testing scenario. In addition, it provides first insights into the correlation between the machine learning based performance metric of validation loss and the resulting code coverage performance of the generated test cases (see Section 1.2). Beyond that, it also utilizes a seq2seq architecture for the first time as a test case generator in fuzz testing. In contrast to prior work introduced in Section 2.3 provides a detailed evaluation of the training process and resulting code coverage performance. The chapter is divided into two sections, each considering distinct model architectures.

First, Section 4.1 provides details about the stacked RNN architecture utilizing the two different types of recurrent cells namely LSTM and GRU introduced in Section 2.2.2. In Section 4.1.2 the model training procedures are outlined, and the parameter choices are described in detail. This is followed by Section 4.1.3, where the HTML-tag generation method is introduced highlighting how the models were utilized to generate HTML-tags. In Section 4.1.4 the different performance values (see Section 3.3) are provided to compare the different models subdivided by cell type with each other and to the results gathered from the baseline (Section 3.4). Furthermore, the results provide an insight in what kind of cell type is better suited to be used for HTML modeling. The results were also published as a conference paper at ICISC conference [78].

Secondly, Section 4.2 describes the proposed seq2seq architecture introduced in Section 2.2.4.

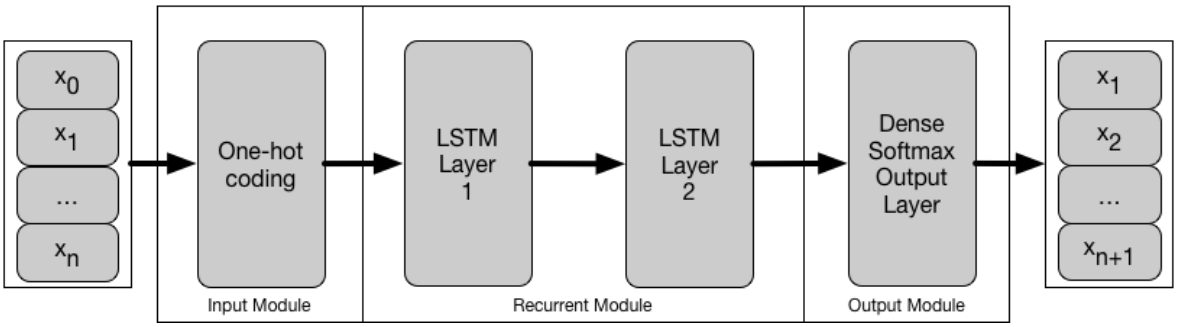

Figure 4.1: Example model architecture with two layers. The input x_0 to x_n is sequence of positve integers representing a character each. The output module provides a probability distribution to predict the character at x_{n+1}.

This is also followed by details regarding the training set and differences to the former training process. In Section 4.2.3 describes the algorithm used for generating HTML-tags in the seq2seq case. Lastly, the results are delivered comparing the approaches with each other and highlighting the performance in comparison to the dataset.

The chapter concludes with a summary providing an evaluation of the presented results and their implication on the rest of the thesis.

4.1 Stacked Recurrent Neural Network

This section introduces the proposed model architecture. It includes the general layout, the types of recurrent cells used, as well as the applied activation functions.

4.1.1 Model Architecture

The model proposed in this section consisted of three modules as highlighted in Figure 4.1 by an example architecture using LSTM recurrent cells. Namely, those modules were

1. An input module,
2. A recurrent module,
3. An output module.

First, the recurrent module takes an integer sequence representing the single characters occurring in the input set. Those integers are then converted into a so-called one-hot vector representation. The one-hot coded vectors have a dimension equal to the amount of different characters in the input set. In addition, those vectors are zero at every position except the one position for the character they are representing, which is set to one. This leads to following

definition: Let X be the input sequence of integers so that $X = \{x_1, x_2, x_3, ..., x_N\}$ with $x_i \in \mathbb{N}_0 \mid 1 \leq i \leq N$ and N denotes the sequence length. Then $\hat{\mathbf{x}}_\mathbf{i} = one\_hot(x_i)$, where $\hat{\mathbf{x}}_\mathbf{i} \in \mathbb{R}^V$ with $V = max(X) + 1$ (vocabulary size), where one is added to account for zeros in the input sequence. The resulting vector $\hat{\mathbf{x}}_\mathbf{i} = (x_1, x_2, x_3, ..., x_V)^\intercal$ with

$$x_j = 0 \,\forall\, 1 \leq j \leq V \;:\; j \neq x_i \;\wedge\; x_j = 1 \;\Leftrightarrow\; j = x_i \tag{4.1}$$

is the converted representation of a single input character. For example, set the input character to "h", which is represented by the integer 2 and let the total vocabulary size be 5 then $one\_hot("h") = (0.0, 0.0, 1.0, 0.0, 0.0)^\intercal$. In general, the one-hot transformation adds dimensionality to the one dimensional input sequence, which improves the overall performance of the neural network. This mainly is because it is easier to learn the relation between a single value set to one and all others set to zero than learning the same relation from a single integer value.

Secondly, the recurrent module takes the one-hot-coded sequence as input and computes $\mathbf{h}_\mathbf{t}^\mathbf{l} = recurrent\_module(\hat{\mathbf{x}}_\mathbf{t})$ with $cell$ being the respective RNN cell used in the particular model (GRU or LSTM) and l the number of recurrent layers. In cases where the recurrent module contains multiple layers (i.e. $l > 1$) the $recurrent\_module$ function is a composite function, as for example shown in Figure 4.1. The figure shows a recurrent module with two LSTM layers, which results in $recurrent\_module(\hat{\mathbf{x}}_\mathbf{t}) = cell_2 \circ cell_1(\hat{\mathbf{x}}_\mathbf{t})$ with $cell$ being the function defined in Equation (2.9). So, in this example $recurrent_module$ uses the one hot coded vector $\hat{\mathbf{x}}_\mathbf{t}$ as input for the first layer ($cell_1$) and the resulting value $\mathbf{h}_\mathbf{t}^\mathbf{1}$ is used as input for the second layer ($cell_2$).

Lastly, the result of the recurrent module is used as input for the output layer to compute the dense layer's output

$$\mathbf{d}_\mathbf{t} = W_{dense} \cdot \mathbf{h}_t^l + \mathbf{b}_{dense}, \tag{4.2}$$

with the result used in $\mathbf{y}_\mathbf{t} = softmax(\mathbf{d}_\mathbf{t})$ with

$$softmax(\mathbf{d})_a = \frac{e^{d_a}}{\sum_{j=1}^{I} e^{d_j}}. \tag{4.3}$$

The $softmax$ function is used to get a valid probability distribution, which means that the sum of the output vector's elements equals one. Therefore, the vector $\mathbf{y}_\mathbf{t}$ provides a probability distribution, which is used to predict the next character in the sequence. In order to generate longer sequences the resulting character is re-fed as input for the next time step $t + 1$ until a maximum sequence length is reached or an end marker value (e.g. a newline character) is generated.

4.1.2 Model Training

All models utilized during the experiments were trained to predict the following character in the sequence. For example, the input was ”$< butto$” then the models were trained to predict ”$button$” for a sequence length of 6. During the model training, the sequence length was set to 150 characters. The vocabulary size (see Section 3.2), and therefore the dimension of the one-hot-coded layer, was 107. The input to the model was an integer sequence where each number represented a unique character. For example, the HTML sequence in Listing 4.1 was converted into the sequence of integers shown in Listing 4.2 that was then used as input for the one-hot-coded layer.

The dataset introduced in Section 3.2 was randomly split into five different training and validation sets, with the validation sets set to a size of 1 MB and being nonoverlapping between splits. Those splits were fixed for all training runs to ensure comparable and reproducible results. The validation set was set aside and only used for computing the validation loss. For each layer and cell type configuration, 15 different models were trained to achieve reliable and reproducible results. Those 15 models consisted of three runs on each of the five splits introduced earlier. Overall, this resulted in 180 differently trained models to collect data from and evaluate the results.

All model configurations were trained for 50 epochs because initial tests indicated that the models had converged more than 20 epochs earlier. The convergence was ensured by regularly monitoring the corresponding graphs showing the development of validation and training loss over time. Furthermore, the monitoring provided a comparison baseline, which indicates the mathematical training quality. The training script saved the model’s internal state to a file every time a new minimum validation loss was achieved.

As mentioned, different configurations of the models were trained in order to study the effects on the generated HTML’s quality and code coverage performance. However, all models used a constant size for the recurrent cells of 256. The models varied in the number of layers. They ranged from one to six layers to follow the results of Cho et al. [30] highlighting that deeper models outperform shallow ones. Furthermore, two types of recurrent cells were used, namely GRU or LSTM (see Section 2.2.2).

All different configured models were initialized with weights drawn from the Glorot uniform

```
1 <datalist id="id1" spellcheck="false" style="style" contenteditable="
    true" class="style_class_0"> -4400000000</datalist>
```

Listing 4.1: Example HTML training input

initializer [79], which means weights were drawn from a uniform distribution in the interval

$$(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}},\ \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}) \tag{4.4}$$

with n_l being the internal size of layer l. Afterward, the models were trained to minimize the cross-entropy loss function

$$\mathcal{L}(\Theta) = -\frac{1}{N}\sum_{i=1}^{N} \mathbf{y_i} \log(\hat{\mathbf{y}}_\mathbf{i}) + (1 - \mathbf{y_i}) \log(1 - \hat{\mathbf{y}}_\mathbf{i}), \tag{4.5}$$

by utilizing the ADAM optimization algorithm (see subsection 2.2.7) using a batch size of 512 training set elements per gradient update and setting the learning rate for the gradient updates to 0.001. The objective function in Equation (4.5) in combination with the input sequence and labels, made the model predict the next character in the sequence. During the training, the goal was not to improve code coverage but to produce HTML.

4.1.3 HTML-tag generation

The saved model states (i.e. weights and trainable parameters) were restored for the corresponding model in order to generate HTML-tags. Algorithm 1 describes how those HTML-tags were generated. It takes the restored model, a seed character and a stop character as input. During the sampling procedure a single character "<" (HTML-tag opening symbol) was used as seed character and the initial hidden state was set to zero. Then the model was used to generate a probability distribution for the next character in the sequence $p(\mathbf{x}|seed, hidden, model)$. This distribution was used to sample the next character in the sequence, which was concatenated with the seed. For example, let $p(\mathbf{x}|seed, hidden, model) = (0.9, 0.02, 0.08)^\intercal$ for a setting with three different character classes, then the first class will be chosen in 90% of the cases, the second class in 2% and the third in 8%.

The while loop (line 6) then repeated the prediction, sampling and concatenation steps until the provided stop character was predicted. Finally, the completed HTML-tag was returned. The structure of the used training set allowed the use of the newline character (i.e.
n) as stop character, because as described in section 3.2 each line of the training set contained a single HTML-tag. In order to accelerate the HTML-tag generation process, it was possible to add an additional input dimension. So, instead of using an one dimensional input character a two dimensional list of input characters provided. Therefore, it was also necessary to have an additional hidden state for each item in the list of input characters. This approach enabled a parallel generation of HTML-tags, which were concatenated to one larger string and then returned.

Algorithm 1 Generate a single HTML-tag with a stacked RNN

```
procedure GENERATEHTMLTAG(model, seed, stopCharacter)
    hidden ← 0
    distribution, hidden ← model.predict(seed, hidden)
    nextChar ← sample(distribution)
    tag ← seed + nextChar                          ▷ String concatenation
    while nextChar ≠ stopCharacter do
        distribution, hidden ← model.predict(nextChar, hidden)
        nextChar ← sample(distribution)
        tag ← tag + nextChar                       ▷ String concatenation
    end while
    return tag
end procedure
```

Cell	Layers	Training		Validation		Training steps	Duration (minutes)	Parameters
		loss	acc.	loss	acc.			
LSTM	1	0.4389	0.8405	0.5344	0.8196	19120	33	400,235
	2	0.4171	0.8449	0.5335	0.8212	19132	67	925,547
	3	0.3842	0.8510	0.5202	0.8259	19068	99	1,450,859
	4	0.3947	0.8487	0.5367	0.8211	19120	132	1,976,171
	5	0.3939	0.8499	0.5892	0.8136	19132	164	2,501,483
	6	0.4095	0.8452	0.5690	0.8024	16714	198	3,026,795
GRU	1	0.4301	0.8410	0.5266	0.8214	19118	29	307,051
	2	0.4330	0.8402	0.5470	0.8191	14130	53	701,035
	3	0.3962	0.8512	0.5501	0.8195	11512	77	1,095,019
	4	0.3906	0.8516	0.5483	0.8197	13960	102	1,489,003
	5	0.3821	0.8546	0.5383	0.8221	13940	126	1,882,987
	6	0.3824	0.8547	0.5380	0.8200	13878	152	2,276,971

Table 4.1: Summary of the training results of the stacked LSTM and GRU models. It provides the training and validation minimum loss and highest accuracy, the average number of training steps, the average training duration in minutes, and the number of trainable parameters. The internal unit size of the models was set to 256 in all training runs.

4.1.4 Results

The trained models generated two test case sets each, with 128 HTML-tags and 256 HTML-tags, respectively. DrCov instrumentalized Firefox while executing the test cases to gather all basic blocks triggered by the test case (see Section 3.3). The code coverage results were then compared with the fuzzer baseline. Table 4.1 provides a summary of all trained models. It already highlights that the GRU models achieved a comparable loss and accuracy during training and validation. Furthermore, this was accomplished with less training time and fewer trainable parameters. The following sections start with reporting the detailed results from the training phase followed by the code coverage reports.

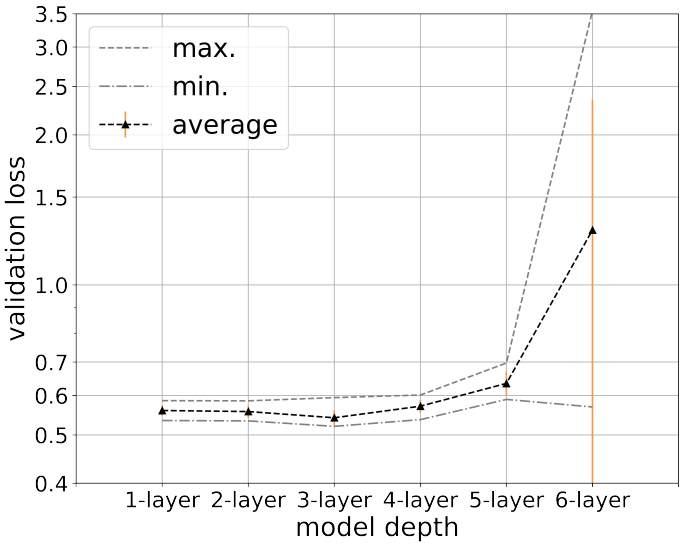

Figure 4.2: Average validation loss of the different LSTM based models (30MB data set) with error-bars indicating the standard deviation across the individual models.

LSTM-based stacked RNN

The training phase has shown that the average validation loss decreases with adding layers to the architecture, as shown in Figure 4.2, reaching a minimum average validation loss of 0.54 for the 3-layer models. The 1-layer models achieved the lowest standard deviation with a value of 0.0133. Furthermore, the figure highlights how the average validation and standard deviation increased in model architectures with more than 3-layers. Especially the models with 6-layers showed the highest standard deviation with 1.0645 and the highest average validation loss with 1.288 compared to the other models. This indicates that the optimization process with the provided training set in combination with the high amount of trainable parameters (see Table 4.1) was very unstable, and also that the stacked RNN architecture reached its performance limits in regards to this specific training set.

Overall, the small differences in loss (≤ 0.02) observed when comparing model archtictures with 1-5 layers did not reflect the difference in the generated HTML's quality. Listing 4.3 shows an example from 1-layer model with a close to average validation loss. The generated HTML-tag is recognizable as HTML, but with a non-existent opening tag and several misspellings in the attribute names. In contrast to this Listing 4.4 shows two HTML-tags generated by an on average performing 3-layer model. Those two generated HTML-tags utilize existent HTML-tags and the attribute names contain fewer errors. However, the generated closing tag for the second HTML-tag is not the correct corresponding one (i.e., $< /head >$). The next two examples in Listing 4.5 highlight the visible difference of a higher validation loss (i.e. ≥ 0.6). The two generated HTML-tags in the prior example contain more mistakes than the former examples together and are not recognizable as HTML at all.

```
1 |    0,  5, 18, 10, 18, 12,  4,  9, 10,  3,  4,  5,  6,  7,  4,  5,
      23,
2 |    7,  3,  9, 14, 13, 12, 12, 15,  1, 13, 15, 16,  6,  7, 17, 18,
      12,
3 |    9, 13,  7,  3,  9, 10, 11, 12, 13,  6,  7,  9, 10, 11, 12, 13,
      7,
4 |    3, 15, 30, 22, 10, 13, 22, 10, 13,  5,  4, 10, 18, 31, 12, 13,
      6,
5 |    7, 10, 19, 32, 13,  7,  3, 15, 12, 18,  9,  9,  6,  7,  9, 10,
      11,
6 |   12, 13, 33, 15, 12, 18,  9,  9, 33,  8,  7, 27,  3, 34, 35, 35,
     8,
7 |    8,  8,  8,  8,  8,  8,  8,  0, 28,  5, 18, 10, 18, 12,  4,  9,
      10,
8 |   27
```

Listing 4.2: Example integer input sequence

```
1 | <war id="id55804" scellcheck="false" tpalleaeck="false" class="
    style_class_0" title="50000000"> null</sab>
```

Listing 4.3: Example HTML-tag from a 1-layer model

```
1 | <p id="id38564" lang="mk"> BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</
    p>
2 | <head id="id240801" sang="al" style="style" class="style_class_0"
    dir="rtl"> 7500000000</pre>
```

Listing 4.4: Example HTML-tag form a 3-layer model

```
1 | <mfoled="id276705" paceeskey="L" wpanslaee="yo" aaie="htly:
    //1270.0.01:8000" dir="utto ctals="style_class_0" dil="atlo"
    tibindex="4500000000"> BABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB//
    outtn>
2 | <sbddotimu id="id74509" ateten="ssfl%" cpnelnheck="true" siass="
    style_class_0" dir="atlo"> p</cj>
```

Listing 4.5: Example HTML-tag form a 6-layer model

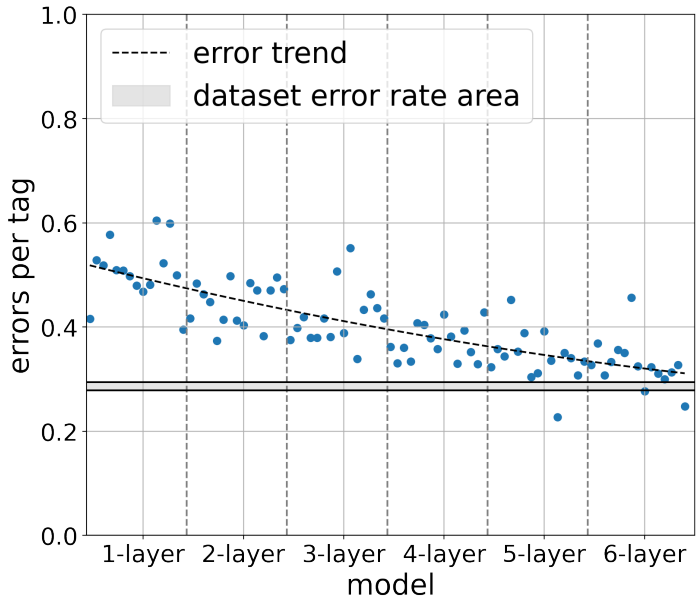

Figure 4.3: Error rate per HTML-tag in regards to model depth for stacked RNN models with LSTM cells

The former highlighted visible differences among the models are further supported by Figure 4.3. It shows the HTML error rate per tag is following the trend of the average validation loss and also shows that the 4-layer models reached the lowest average error rate. The high standard deviation in validation loss of the 6-layer models is also reflected in the error rate. The 6-layer models had the highest difference between the minimum and maximum error rates and distances to the average value. Overall, the error rates approached the datasets' rate area, and models with 4- and 5-layers were able to achieve error rates in or below the datasets' area.

The code coverage in terms of uniquely discovered basic blocks followed the same already observed general trend of the validation loss. The amount of uniquely discovered basic blocks increased with a lower validation loss and error rate as shown in Figure 4.4 (left) and Figure 4.5 (left). On average the test cases with 128 HTML-tags were able to trigger more basic blocks than the test cases with 256 HTML-tags each. The larger test cases introduced errors by doubling the amount of HTML-tags per file. The larger amount of errors led to more test cases being at least partially rejected by the rendering engine. In terms of total discovered basic blocks in the 128 HTML-tags setting, the four layer models were able to achieve coverage performance in the datasets' coverage area, with the best performing four layer model achieving an absolute difference of 691 basic blocks to the best performing dataset. In the 256 HTML-tags per file setting, none of the models were able to reach the datasets' coverage area.

At the same time, in both settings all models were able to discover basic blocks, which were not triggered by the best performing dataset (see Figure 4.4 (right) and Figure 4.5 (right)).

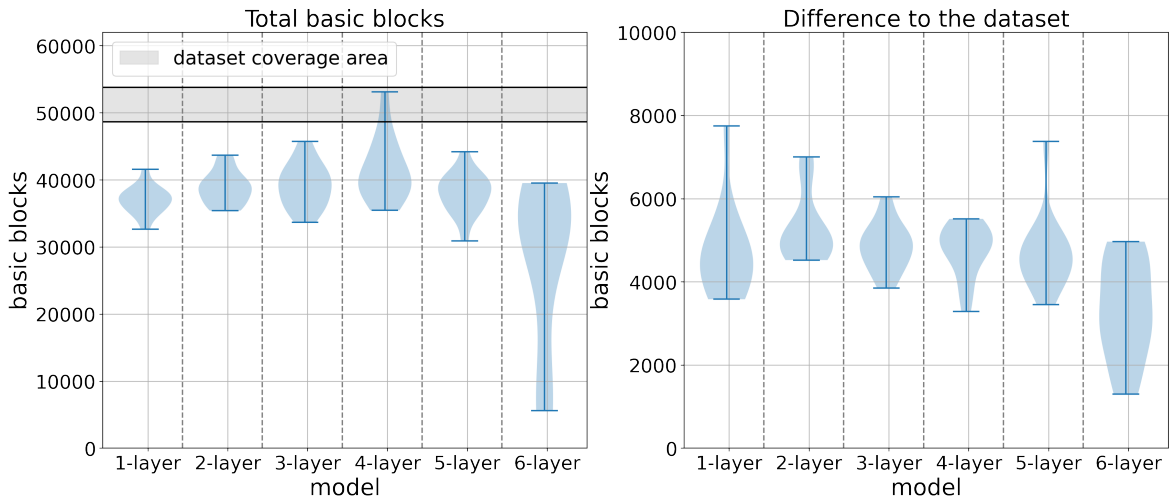

Figure 4.4: LSTM based models: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

Furthermore, the same trend regarding the connection between average validation loss and error rate could be seen in the ability to discover new basic blocks with the average trend following the same path.

In both cases the 4-layer models discovered the most basic blocks in terms of absolute numbers. The best performing 4-layer model came very close to the best performing dataset with an overlap of 88.6% and still able to discover more than $5,400$ new basic blocks during the run with 128 HTML-tags. On the other end of the spectrum the 6-layer models performed worst, with three models discovering less than $10,000$ basic blocks. In addition, the results for the 6-layer models were distributed over a large area, which is not unexpected especially when taking the former observed standard deviation into account. Combined with the observed standard deviation and the error rate this further supports that the stacked RNN with 6-layers using LSTM cells reached its performance limit with regards to the provided training set.

In comparison to the mutation set the models achieved a maximum overlap of 90% with the lowest mutation chance 1.6%. The sets with a mutation chance of 1.6% also had a high overlap with the best performing dataset of 87.6% and 86.5% for the dataset with 128 and 256 HTML-tags respectively. Table 3.2 shows the absolute number of basic blocks discovered by the different mutation sets. Especially the sets with the lowest mutation chance were still able to outperform the models in absolute terms. Since the low mutation chances kept the HTML structure mostly intact and therefore were also able to perform close to the base dataset. The Table 4.2 highlights the development of the overlap between the best performing LSTM based models, dataset and differently mutated dataset. It shows that the overlap decreased steeply as soon as the mutation chance exceeded 10%. This also emphasizes the ability of

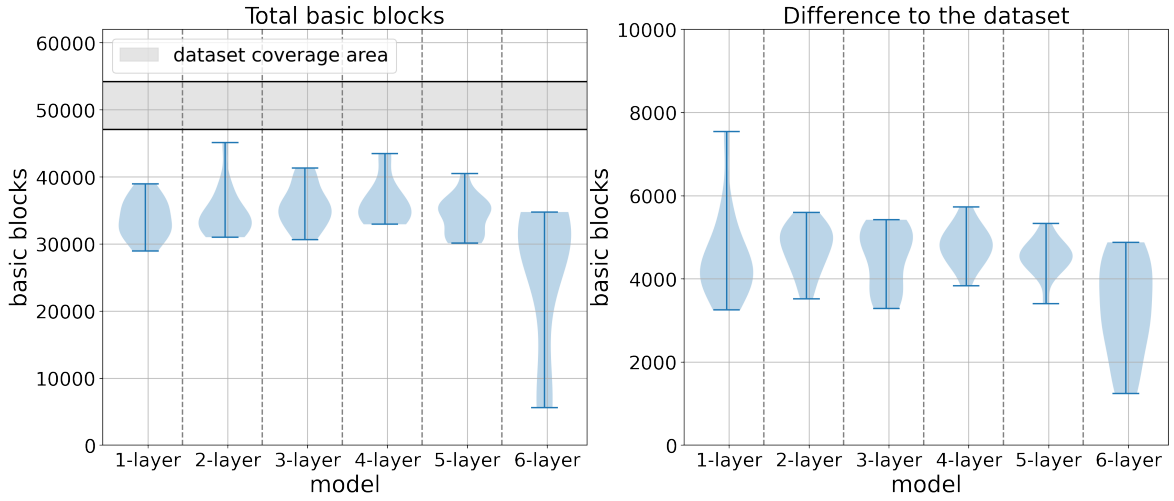

Figure 4.5: LSTM based models: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

Test set:	dataset	1.6%	3.2%	6.4%	12.8%	25.6%	51.2%
Overlap 128:	88.6%	86.2%	86.5%	84.1%	67.4%	35.8%	20.0%
Overlap 256:	70.6%	68.8%	68.9%	67.4%	57.9%	47.3%	16.3%

Table 4.2: Overlap in percent of discovered basic blocks of the best performing LSTM models. The best performing models are compared against the dataset and the randomly mutated test runs introduced in Section 3.4.

the best performing models to discover code paths neither discovered by the dataset nor the mutated sets.

GRU-based stacked RNN

The training phase of the GRU cell based stacked RNN showed a different development in average validation loss and standard deviation compared to the LSTM based models, as highlighted in Figure 4.6. The figure shows how the average validation loss increased up to three layers, while the standard deviation decreased. Adding more than three layers led to a decrease in average validation but an increase in standard deviation. In general, the differences between the models' average validation losses were all ≤ 0.02 with reaching a minimum of 0.545 in the 1-layer setup. At the same time this setup achieved the highest standard deviation with 0.0108, which was less than the minimum standard deviation achieved by the LSTM based models. The smallest standard deviation was achieved by the 3-layer models with a value of 0.00659. The overall smaller average standard deviation over all models indicates a more stable training process in GRU based setting. It is also important to notice that the GRU based models with a comparable amount of parameters were able to achieve a better training performance than the LSTM based models, for example the $1,882,987$ and

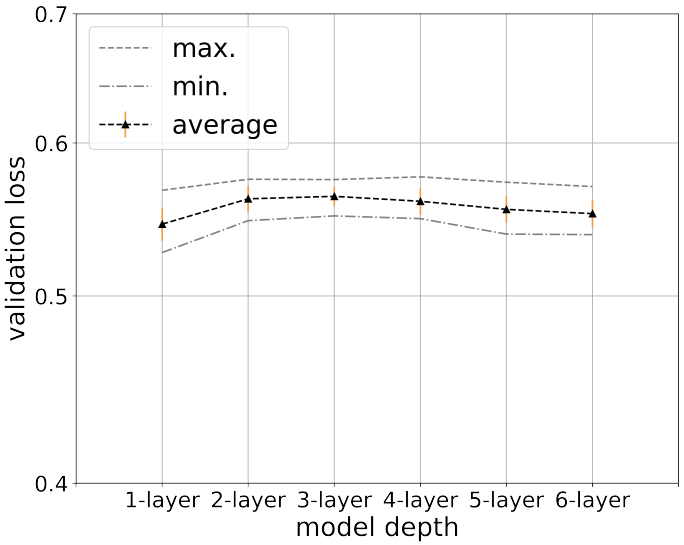

Figure 4.6: Average validation loss of the different GRU models with error-bars indicating the standard deviation across the individual models.

```
<address id="id391211" lang="wo" accesskey="K" title="\0">
    BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</pirer>
```

Listing 4.6: Example HTML-tag from a 1-layer GRU based model

$1,976,171$ trainable parameters for the 5-layer GRU and 4-layer LSTM based models respectively (see Table 4.1). The comparison of the average loss values of the 5-layer GRU models (0.5544) and the 4-layer LSTM models (0.5710) shows that the GRU based model are able to outperform the LSTM ones. Furthermore, comparing the standard deviations with values 0.00861 and 0.0165 for the 5-layer GRU models and 4-layer LSTM models respectively, emphasizes the more stable and therefore also more reliable training process of the GRU based models.

Listing 4.6 and Listing 4.6 show examples for generated HTML-tags by a 1-layer and 5-layer GRU model respectively. The HTML-tag generated by the 1-layer GRU model is recognizable as HTML besides using a non existent closing tag, especially the comparison with Listing 4.3 highlights a visible performance difference. It is also noteworthy that the 1-layer GRU model has approximately $100,000$ parameters less than the 1-layer LSTM model[1] and is able to reproduce the HTML structure with higher accuracy. The excerpt in Listing 4.7 shows an example generated by one of the best performing models. It reproduced the structure correctly and even closed the opened HTML-tag with the correct closing tag, however this was still not always the case.

In spite of achieving the lowest average validation loss, the 1-layer models are not able to achieve the lowest average HTML error rate as highlighted in Figure 4.7. It also shows how

[1] $100,000$ parameters are a quarter of the 1-layer LSTM model's total parameters

```
<ins id="id88418" style="style" tabindex="Number.MIN_VALUE"
    contenteditable="true" class="style_class_0" title="10000">
    4400000000</ins>
```

Listing 4.7: Example HTML-tag from a 5-layer GRU based model

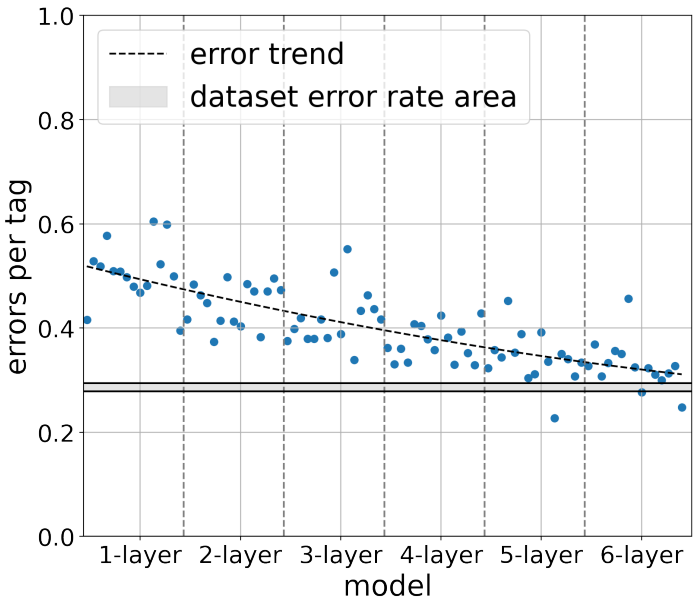

Figure 4.7: Error rate per HTML-tag in regards to model depth for stacked RNN models with GRU cells

the average HTML error rate further decreases even at point, where the trainable parameters already exceed the amount of the overall best performing 4-layer LSTM based model. It is important to notice, that the GRU cells are able to achieve a lower HTML error rate with less trainable parameters in direct comparison with the LSTM cells using the same overall architecture (i.e. stacked layers).

The code coverage performance of the GRU based approach shows that these models are able to achieve an higher absolute performance than the best performing LSTM based model and the dataset in the 128 HTML-tag case. The 5-layer GRU model is able to achieve the highest absolute number of discovered basic blocks in comparison with the datasets and LSTM based models. It is important to notice, that in this case the average value also lies well within the datasets' coverage area, as shown in Figure 4.8. Those 5-layer models are also able to achieve the on average highest difference to the datasets, which means they discovered overall more new unique basic blocks than all the other models. The best performing 5-layer model is able to achieve an overlap of 91.7% with the best performing dataset, while still able to discover more than $7,000$ basic blocks not triggered by the dataset. In total, four of the 5-layer GRU models were able to discover more basic blocks than the datasets. Table 4.3 second row provides the overlap data for the best performing GRU based model compared

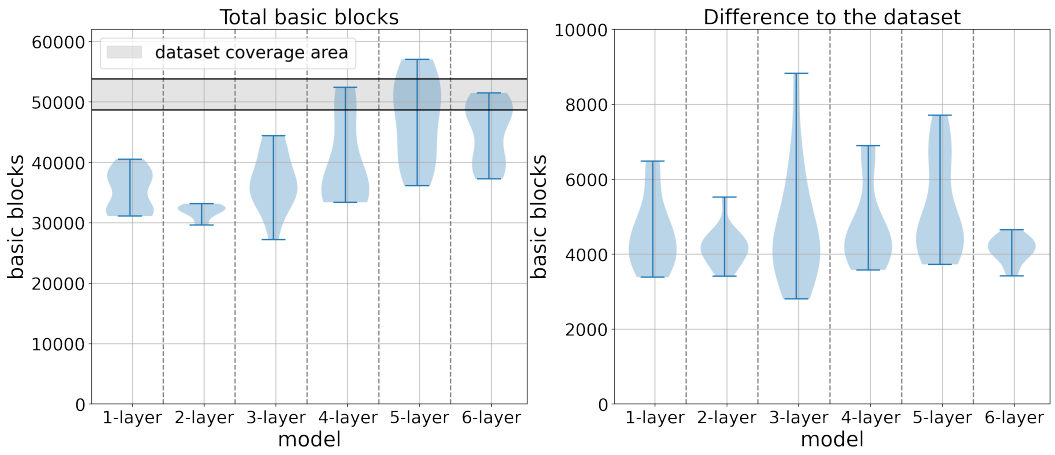

Figure 4.8: GRU-based models: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

Test set:	dataset	1.6%	3.2%	6.4%	12.8%	25.6%	51.2%
Overlap 128:	91.7%	84.1%	83.8%	81.5%	65.4%	33.9%	19.0%
Overlap 256:	86.9%	95.4%	94.9%	92.8%	61.1%	49.6%	17.2%

Table 4.3: Overlap in percent of discovered basic blocks of the best performing GRU models. The best performing models are compared against the dataset and the randomly mutated test runs introduced in Section 3.4.

to the differently mutated data set in the 128 HTML-tags per case scenario. It highlights the decrease in overlap with an increasing mutation chance. Furthermore it shows that the naive mutation strategy was not able to provide the same coverage as the model. The overlap has step drop for mutation chances over 10%, which is not surprising because the model was able to reproduce the structure of HTML and the random mutation is breaking it.

The GRU based test cases with 256 HTML-tags per case also show a higher code coverage performance than the LSTM based approach (see Figure 4.9), with one 4-layer model to achieve a performance in the datasets' coverage area and the 5- and 6-layer models being able to reach that area for 10 out of 30 times. However, the amount of new unique basic blocks discovered decreased, when using 256 HTML-tags. The overall effect of adding more HTML- tags per file is decreasing the number of basic blocks discovered by the models in absolute values and in difference to the best performing dataset. This is basically the same observation made in the LSTM based model, nonetheless the results indicate that the GRU cells are better suited to be model HTML based data for fuzz testing. The best performing model in the 256 HTML-tags scenario is a 6-layer model. It achieved an 86.9% overlap with the best performing dataset and discovers more than $4,500$ basic blocks not included in the dataset, as shown in the last row of Table 4.3. The table also shows the overlaps with the differently mutated data sets. It is surprising to see the increased overlap comparing

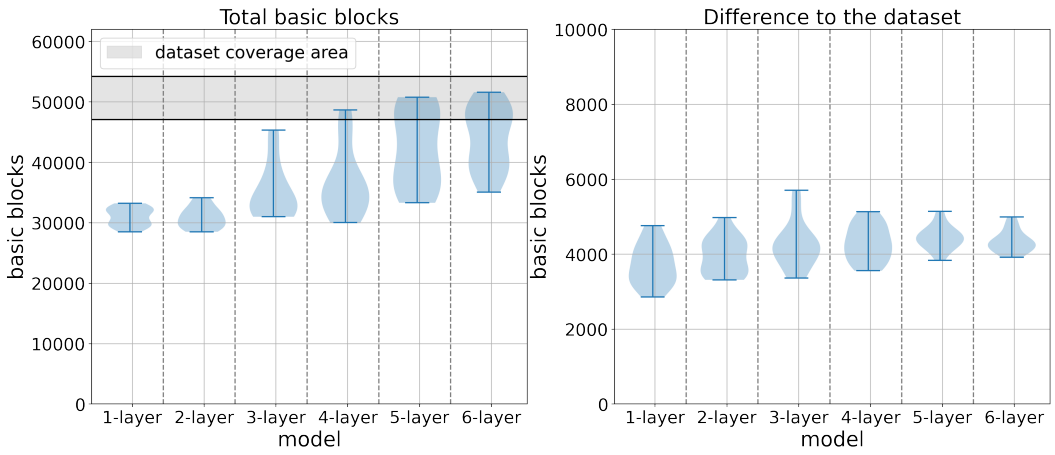

Figure 4.9: GRU-based models: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

the dataset and mutation chances smaller than 10%, because it is not following the trend observed before. Nonetheless, the overlap still decreased strongly as soon as the mutation chance was set to values greater than 10% with a decrease of one third during the step from a mutation chance of 6.4% to 12.8%.

Summary

The results show that GRU based models are better suited to learn the HTML modeling task, regarding the used training set and model architecture. They outperformed the LSTM based models in all metrics. Firstly, the training phase showed a more stable average loss and standard deviation with the best performing models able to achieve a lower validation loss. Secondly, GRU based models were able to outperform the LSTM based ones in terms of average HTML error rate per tag even with less trainable parameters. In addition, the results highlight that the combination of average validation loss and HTML-error rate is a good indicator of the performance a model is able to achieve. Since both RNN cell types best performing models in terms of code coverage also achieved a lower HTML-error rate compared to the others. Lastly, comparing the best performing models in terms of code coverage the GRU based models outperformed the LSTM ones by $4,000$ basic blocks with less trainable parameters. The performance data of the 5- and 6-layer GRU based models also show an on average more reliable and reproducible performance compared to the LSTM based models.

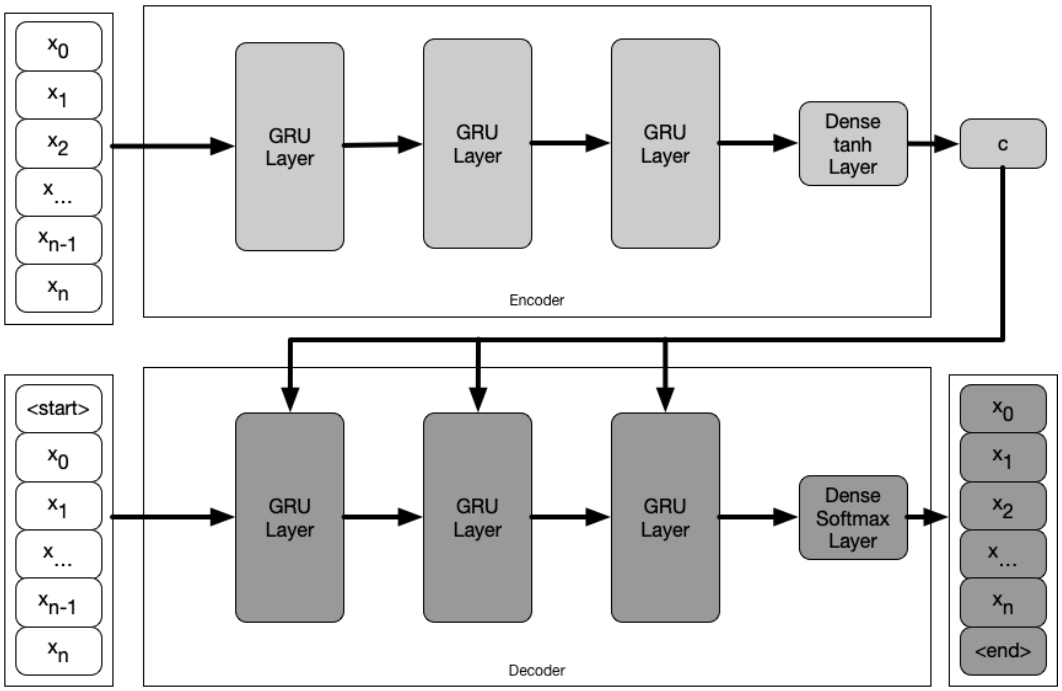

Figure 4.10: The seq2seq architecture used as HTML-fuzzer. The encoder consisted of two GRU layers and the decoder consisted of three layers.

4.2 Sequence-to-Sequence

The prior section analyzed how stacked RNN models with LSTM or GRU cells (see Section 2.2.2) perform in terms of HTML validity and code coverage as test case generators. This section analyzes the performance of Sequence-to-Sequence (seq2seq) models (see Section 2.2.4).

4.2.1 Model Architecture

The model architecture proposed in this section uses the basic concepts described in Section 2.2.4. The cells used inside the recurrent layers are also GRU (see Section 2.2.2) because the results from the stacked RNN approach in subsection 4.1.4 showed that those are better suited to learn and generate HTML.

The seq2seq model used consists of an input model model, an encoder and a decoder as shown in Figure 4.10. The input model computes $\hat{\mathbf{x}}_\mathbf{i} = one_hot(x_i)$ for each $x_i \in X = \{x_1, x_2, x_3, ..., x_N\}$ as described in subsection 4.1.1. The encoder consists of two GRU based layers computing

$$\mathbf{h_t} = enc(\hat{\mathbf{x}}_\mathbf{t}) = gru_2 \circ gru_1(\hat{\mathbf{x}}_\mathbf{t}) \tag{4.6}$$

and a final dense layer used to compute the candidate value $\mathbf{c}$ with

$$\mathbf{c} = tanh(W_{dense} * \mathbf{h_t} + \mathbf{b}_{dense}) \tag{4.7}$$

The decoder takes the candidate value $\mathbf{c}$ and uses it as input for another dense layer with $tanh$ as activation function with $\mathbf{c}_{dec}$ being the result. Then $\mathbf{c}_{dec}$ is used as the initial hidden state for all three layers. In order to implement the attention mechanism the gates and the final output function for each GRU cell is changed:

$$\mathbf{r'_t} = sigm(W_r \cdot \mathbf{x_t} + U_r \cdot \mathbf{h_{t-1}} + C_r \cdot \mathbf{c}_{enc}), \tag{4.8}$$

$$\mathbf{z'_t} = sigm(W_z \cdot \mathbf{x_t} + U_z \cdot \mathbf{h_{t-1}} + C_z \cdot \mathbf{c}_{enc}), \tag{4.9}$$

$$\mathbf{h'_t} = \mathbf{z_t} \cdot \mathbf{h_{t-1}} + (\mathbf{1} - \mathbf{z_t}) \cdot tanh(W \cdot \mathbf{x_t} + \mathbf{r_t} \odot (U \cdot \mathbf{h_{t-1}} + C \cdot \mathbf{c_{enc}}) \tag{4.10}$$

Those modified cells gru_{mod} were then used to build the decoder:

$$\hat{\mathbf{y}}_\mathbf{t} = dec(\mathbf{c}, \mathbf{x}_t) = gru_{mod-3} \circ gru_{mod-2} \circ gru_{mod-1}(\mathbf{x}_t). \tag{4.11}$$

The decoder's input $\mathbf{x_0}$ at $t = 0$ is a special $< start >$ symbol. The decoder's output is fed into a final dense layer with a $softmax$ activation function as already used in the stacked RNN approach (see subsection 4.1.1). This final layer yields a probability distribution over the next character in the sequence. After sampling the next character from the distribution, it is fed into the decoder again until the decoder predicts a special $< end >$ symbol.

4.2.2 Model Training

The training set for the seq2seq models differed in terms of structure from the training set proposed during Section 4.1. The maximum sequence length was set to 250 characters, which is longer than during the stacked RNN training. The increased sequence length was necessary to be able to use even the longest input examples. Each training example consisted of a whole tag instead of a continuing sequence from the dataset. Each of those examples was converted into an integer sequence and used in three different ways. First, it was used as the input for the encoder. Secondly, the $< start >$ symbol was prepended to the tag and it was used as the decoder's input and lastly the $< end >$ symbol was appended to the tag and to be the prediction label. All those sequences were padded with $< pad >$ symbols to the maximum sequence length. This was done to allow the use of the dynamic RNN function in TensorFlow, which allows to unroll the used RNNs over the time steps and therefore accelerates the computing. In order to avoid a negative impact on the loss function or even train the resulting model on predicting $< pad >$ symbols a dynamic mask

was created during training, that omitted the padded positions during the loss computation. All models were trained to generate the decoder's input sequence with an appended $< end >$ symbol.

All cells were initialized by the Glorot uniform initializer (see Equation 4.4). The internal size of the cells used in the decoder and encoder was set to 256, 512 and 1024 and again 3 runs with 5 different splits were trained, resulting in 45 models total. The internal sizes were chosen so that the smallest trained model and the best performing stacked GRU based RNN had a comparable number of trainable parameters. All models were trained using the ADAM optimizer, described in subsection 2.2.7 with batch sizes of 512, 256 and 128 for the models with 256, 512 and 1024 internal units respectively. It was necessary to use different batch sizes, since the larger batch sizes did not fit into the GPU's memory when the number of internal units increased and using the same smaller batch size for the different internal sizes would have resulted into not utilizing the available computational resources optimally. Additionally, the training was stopped if the there was no improvement in validation loss for 5 epochs in order to avoid continue using computational resources on models that already converged to the lowest validation loss.

During each training run a checkpoint was created for the lowest achieved validation loss. This checkpoint enabled to restore the model in order to generate the HTML-tags. In addition, the candidate values for this lowest validation loss were saved. The training phase resulted in providing 45 seq2seq models, which were used during the experiments.

4.2.3 HTML-tag generation

The algorithm used for HTML-tag generation differed slightly from the one described in subsection 4.1.3. In the seq2seq model, the former algorithm takes place in the model's computational graph without returning from the model until the "$< end >$" symbol is reached. Algorithm 2 describes the basic steps of generating HTML tags with a seq2seq model. It takes the encoder, a list of candidate values, the end symbol, and a value used as standard deviation as input. It is crucial to notice that the saved candidate values were not used directly during the generation of new HTML tags. Instead, a random value from the list of candidates was selected (line 2). Then, the selected candidate value (initC) was used to sample a new candidate (c) from the normal distribution with the mean set to the candidate value (initC) and the standard deviation (std) set to 0.0001 (line 3). Lastly, the new candidate value (c) and the end symbol (endSymbol) were passed to the decoder to create a new HTML tag (line 4). The adjustment to create new candidate values was necessary because it became apparent after sampling from the first seq2seq models that a fixed candidate value led to repeating tags, where the opening and closing tags were close to constant, and only a few attributes differed between sampling runs.

Algorithm 2 Generate a single HTML-tag with a seq2seq model

1: **procedure** SEQ2SEQHTMLTAGGENERATION($encoder, candidates, endSymbol, std$)
2: $initC \leftarrow random.choice(candidates)$
3: $c \leftarrow normalDistribution(initC, std)$
4: $tag \leftarrow encoder.predict(c, endSymbol)$
5: **return** tag
6: **end procedure**

The former described approach is fundamentally different from the traditional way of using seq2seq-based networks. For example, an input is provided to the encoder during translation tasks, and the result is taken directly from the decoder without modifying the candidate value. The way seq2seq models are used in this work is more focused on the generative task instead of approximating a function between two domains (e.g., languages). It follows that the candidate values are used as seeds for the fuzzing process. The results provided in the next section (see Section 4.2.4) indicate a very stable performance for the models with 256 internal units. This stable performance made it necessary to evaluate the behavior of these models for different values of standard deviation in the normal distribution used to generate a new candidate value (line 3 in Algorithm 2). The additional values used for the evaluation were 0.005, 0.01, 0.05, and 0.1. An additional $16,384$ HTML tags per model were generated and used to create test cases for each value.

4.2.4 Results

The seq2seq training phase has shown that models with an internal size of 256 units have the lowest average validation loss, but achieve a higher standard deviation than the other model types with values of 0.2425 and 0.0654 respectively. However, the standard deviation is decreased by doubling the internal size to 512 units with a value of 0.0152, but the average validation loss is slightly increased to 0.2999. Extending the internal size to 1024 units leads to another increase in standard deviation to 0.0267 and also to an increase in average validation to 0.3499, which is approximately the same difference as between the 256 and 512 units models. The low increase is surprising after observing a steep increase in validation loss during the LSTM-based stacked RNN approach after 4-layers and an increasing number of trainable parameters. It should have been expected that even with GRU cells in the seq2seq architecture a similar steep increase happens after introducing more than $13\times$ trainable parameters in direct comparison between the 256 units and 1024 units models (see Table 4.4) with $1,987,356$ compared to $26,822,940$ respectively.

The similarity between the models in terms of generated HTML-tags was higher than anticipated by the differences in average validation loss and prior experience with the stacked RNN models (see subsection 4.1.4). Listing 4.8 shows a HTML-tag generated by a seq2seq

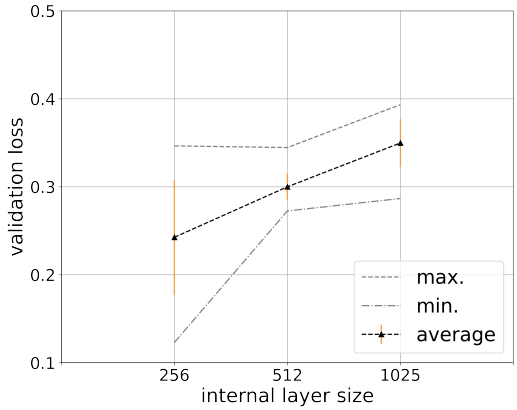

Figure 4.11: Average validation loss of the seq2seq models with error-bars indicating the standard deviation across the individual models

Units:	256	512	1024
Parameters:	1,987,356	7,120,156	26,822,940

Table 4.4: Trainable parameters by model depth for GRU based architectures

model with an internal size 256 units. It is well formed and the attributes are spelled correctly. The HTML-tag shown in Listing 4.9 is also well formed and uses correctly spelled attributes. In contrast to both of those tags Listing 4.10 shows a HTML-tag from a seq2seq model with 1024 units. It still has a well formed structure, but the attributes are spelled incorrectly. The emphasize in this case lies on the quality of the reproduced structure. The model generated the right corresponding closing HTML-tag and all attribute follow the right principle of $attributename = "value"$.

The described well formed structure is the reason for Figure 4.11 showing a low HTML-error rate for all three models. This is not surprising, since the validator checks the HTML input for the overall structure and adherence to the syntactical rules of HTML. It is also

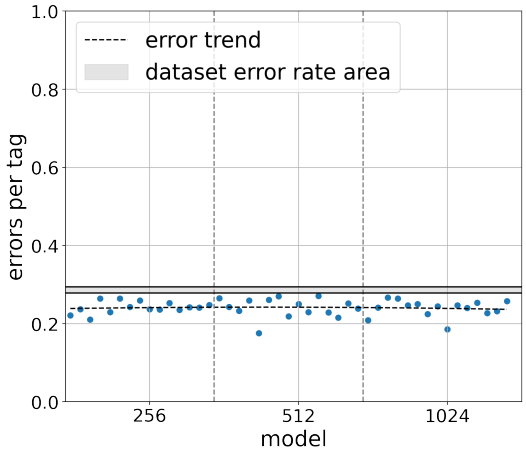

Figure 4.12: Error rate per HTML-tag in regards to the internal size of the seq2seq model.

remarkable that the average HTML-error rate stays below the all the datasets error rates, which highlights the capability of the seq2seq approach to learn and reproduce the structure of HTML.

```
<sup id="id11901" lang="dz" title="(-00" translate="yes"
    contenteditable="false" tabindex="0">
    BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</sup>
```

Listing 4.8: Example HTML-tag from a GRU based seq2seq model with an internal size 256

```
<span id="id276570" lang="ps" tabindex="-4400000000" spellcheck="
    false" dir="ltr" title="-10000000"> -7500000000</span>
```

Listing 4.9: Example HTML-tag from a GRU based seq2seq model with an internal size 512

```
<input id="id17876" max="-21" tyte="clon" formuecestyle="style"
    spellcheck.t="21" asckEpso="polofia"> 2e100</input>
```

Listing 4.10: Example HTML-tag from a GRU based seq2seq model with an internal size 1024

In terms of code coverage the overall performance of the models in the case of 128 HTML-tags per model is shown in Figure 4.13. Especially, the performance of the 256 units models is noteworthy. Those models achieve a very a low distribution and are all in the coverage area of the dataset. The overall best performing model (256 units) achieves a maximum code coverage of $52,100$ basic blocks and a difference with the best performing dataset of $4,315$ basic blocks. In all three cases the overall performance of the models is closer distributed with no visible outliers compared to the stacked RNN approach (see Figure 4.8). This is not a surprise taking into account the close to linear HTML-error rate and the development of average validation loss. The models with 512 units also achieve an average performance inside of the dataset code coverage area and a maximum of $51,460$ basic blocks. In contrast, the models with 1024 are only able to achieve the lower bound of the dataset area with a maximum of $48,656$ basic blocks. The difference between the dataset and the models is lower than in the stacked RNN scenario, but still averages at approximately $3,500$ basic blocks. The overlap between the best performing 256 units model and the dataset is 88.8%. The average overlap between the dataset and the best performing models is 88.76%.

The code coverage in the 256 HTML-tags setting shows a very similar picture compared to the 128 HTML-tags one. Figure 4.14 (left) in comparison to Figure 4.13 (left) highlights this similarity. The models with 256 units are again able to achieve a performance completely inside the dataset area. It is remarkable that there is no drop in performance between the 128- and 256-HTML tags per case settings, which was observed during the stacked RNN approach for both cell types (see subsection 4.1.4). The difference between the best performing dataset and the models also shows little difference to the 128 HTML-tags per case setting,

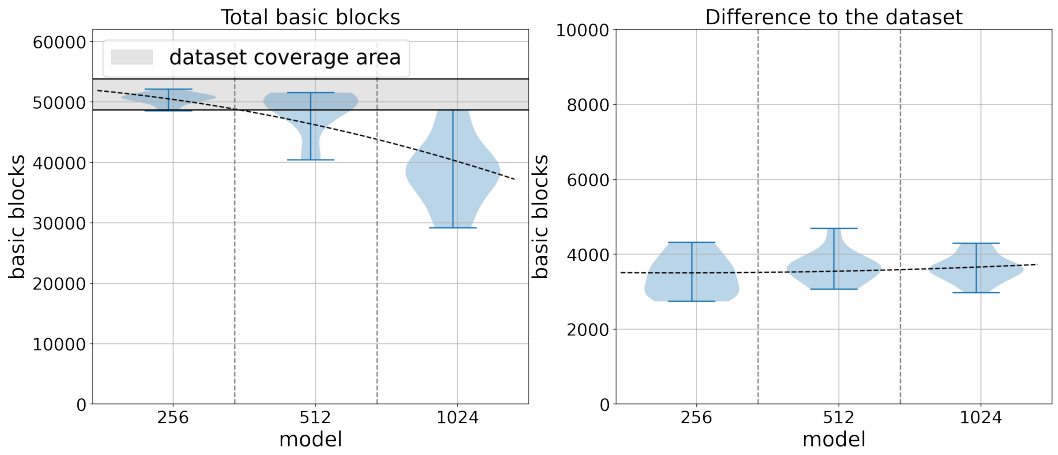

Figure 4.13: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

as highlighted by Figure 4.14 (right) the average over all models is even slightly higher with approximately $3,800$ basic blocks.

Compared to the mutation set the mutation set the models achieved an average overlap of 88.3%. If the lowest three mutation chances (1.6%, 3.2% and 6.4%) are left out the average overlap drops to 66.15%, which is not a surprise since the discovered basic blocks are less and the higher mutation chance destroys the structure of the HTML, whereas all three kinds of seq2seq are producing a well formed structure.

Overall the results show that the seq2seq architecture is able to outperform the stacked RNN approaches regarding the average performance reliably. However, this approach did not achieve an absolute performance above the dataset coverage area as the 5-layer GRU based model did in the 128 HTML-tags case. Nonetheless, the seq2seq architecture also proved its capability in performing well in both settings (128 and 256 HTML-tags per case). In addition, this performance is achieved with a comparable amount of trainable parameters for the best performing models. The results also show that a stable model's performance in terms of code coverage is transferable between the 128 and 256 HTML-tags per case settings and therefore achieve in both cases a similar performance.

Results for varying standard deviation

The results for the varying standard deviation while sampling a new candidate value (see Section 4.2.3) also showed a very stable picture. During the HTML-validation step it already became apparent that sampling candidate values in a large area (i.e. std ≤ 0.01) around the best performing values for the model still achieved the low HTML-error rate (see Figure 4.15) already seen in the prior experiment. This was the case for all standard deviation

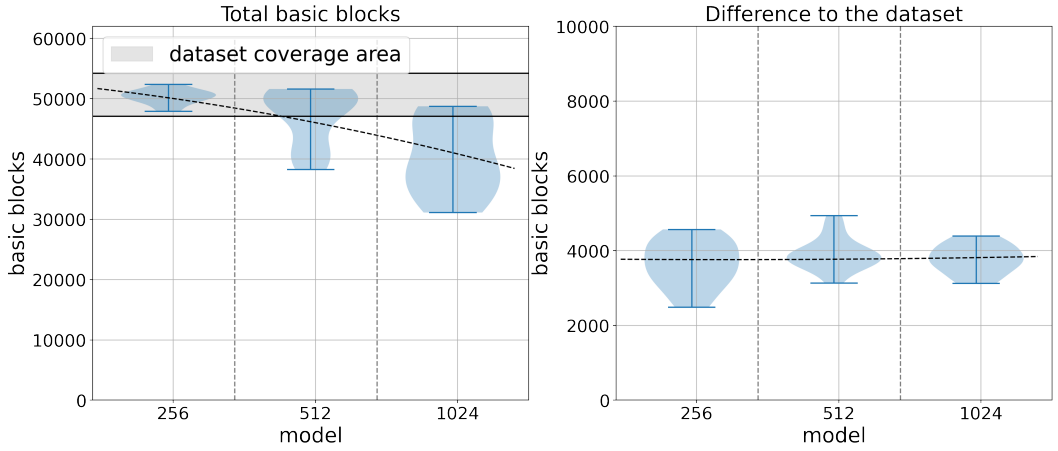

Figure 4.14: Code coverage performance with 256 HTML-tags per test case: (left) absolute number of basic blocks discovered by model; (right) basic blocks not discovered by the test set

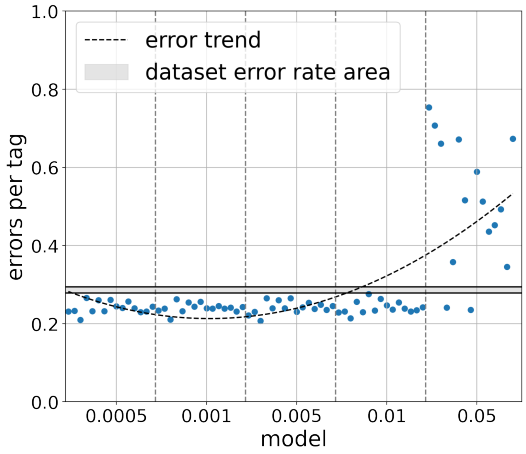

Figure 4.15: The impact of increasing the standard deviation during the sampling on the HTML error rate per tag.

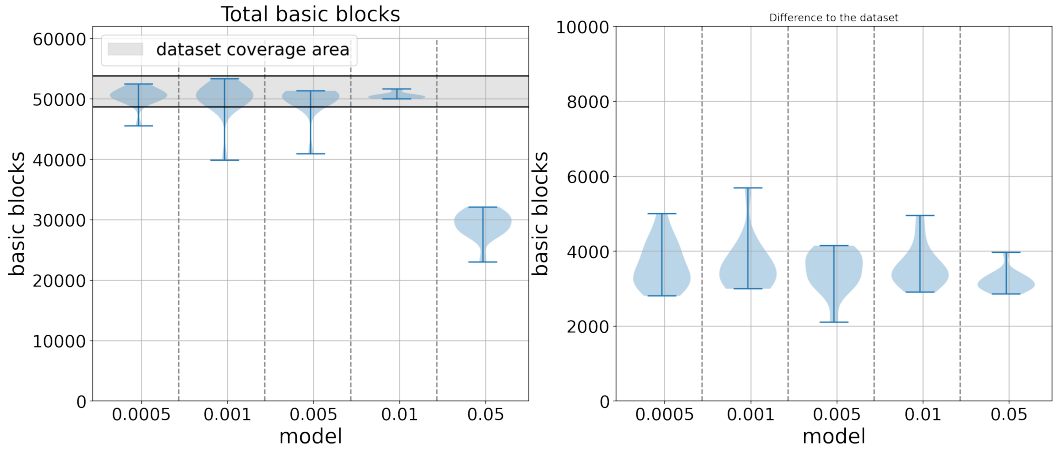

Figure 4.16: Code coverage performance of the models with 256 internal units with varying standard deviation in the setting with 128 HTML-tags per case.

values less than 0.05 with a minimum HTML error rate of 0.21 and a maximum of 0.27. However, setting the standard deviation to 0.05 lead to a wide spread HTML error rate with an average of 0.58, a minimum of 0.24 and a maximum of 0.75 error per HTML-tag. Especially, the maximum indicates that three out four HTML-tags contained errors and as seen in Section 4.1 a high HTML error rate is a good indicator for a low code coverage performance although not conclusive on its own.

In general the code coverage performance in terms of absolute values and difference to the best performing dataset was kept in the dataset area for all standard deviation values less than 0.05 with exception of single outliers, as highlighted in Figure 4.16 and Figure 4.17. The best performing model in the 128 HTML-tags setting used a standard deviation of 0.001 and achieved an absolute number of $53,329$ basic blocks, which is higher than the achieved value with the default settings. The difference of that model to the best performing dataset was $5,694$ basic blocks. This was also a better performance than the standard setting for the 256 units seq2seq models. Nonetheless, it has to be taken into account that this was a single outstanding performance. The average performance was $\approx 50,000$ basic blocks with another outlier achieving only $39,843$ basic blocks. This is not a surprise, because the result itself is highly dependent on what initial candidate value is chosen. Therefore, it relied on the position of that initial value in the distribution of values. For example if that value already was at the border of the distribution then there was a high chance to sample values outside of that distribution leading to not as good HTML as values inside of the distribution. The effect can be observed in a very strong way for the test run with a standard deviation of 0.05, which almost certainly lead to values outside of the distribution of valid values.

For the test cases with 256 HTML-tags the results reflect the prior ones (see Figure 4.17). Especially, the runs with stable results in the former setting also show similar values in this one. It showed again that a standard deviation of 0.05 is too large to perform well on

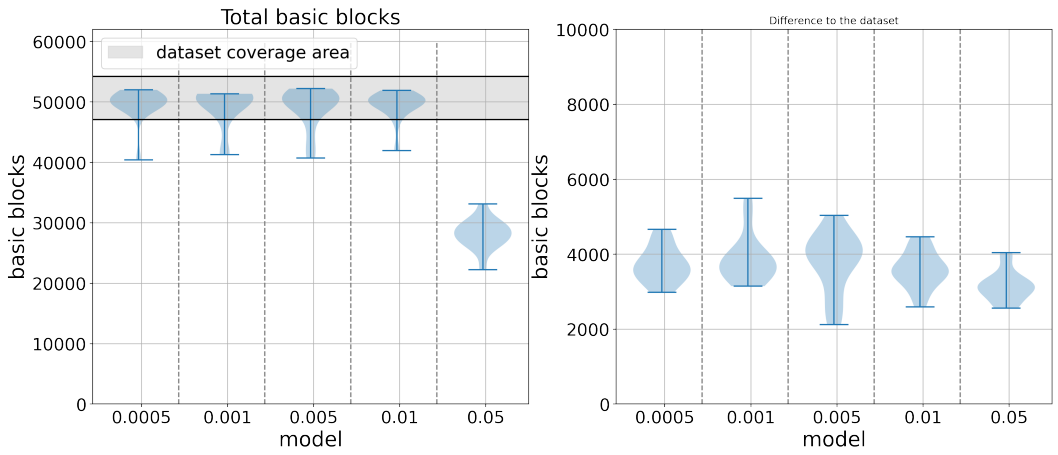

Figure 4.17: Code coverage performance of the models with 256 internal units with varying standard deviation in the setting with 256 HTML-tags per case.

the provided distributions and therefore did not lead to generating well performing HTML as already hinted by the HTML error rate. The test run with a standard deviation set to 0.001 again achieved the best overall performance with $51,289$ basic blocks in total and a difference of $5,023$ basic blocks compared to the best performing dataset.

4.3 Discussion

The results presented in Section 4.1.4 have shown that the LSTM-based networks were outperformed by the GRU-based ones in terms of achieved code coverage. This indicates that the LSTM-based networks were not able to cover the underlying structure, this could be caused by the complexity added into the LSTM-cell by the additional gating mechanism (see Section 2.2.2). The additional gating adds more trainable parameters to the LSTM-cell and therefore makes it more difficult to train the overall network.

In addition, using the sequence-to-sequence models that add additional parameters during training by adding a second RNN helped to improve the code coverage further. It also introduced more continuity and stability into the training process. These effects indicate that the sequence-to-sequence based model captures the underlying structure better than the stacked RNNs. It also introduced more variety in the resulting HTML-tags, which might be due to the combination of sampling from the Encoder's output vector space and the bias of the stacked RNN models during sampling. Here, the bias comes from the statistical distribution of first letters in HTML-tags, because they are not evenly distributed between all possibilities.

After combining the resulting code coverage, HTML error rate and validation loss it became clear that validation loss on its own can only be a small performance indicator, since even models with a very low validation loss can perform very badly in terms of code coverage.

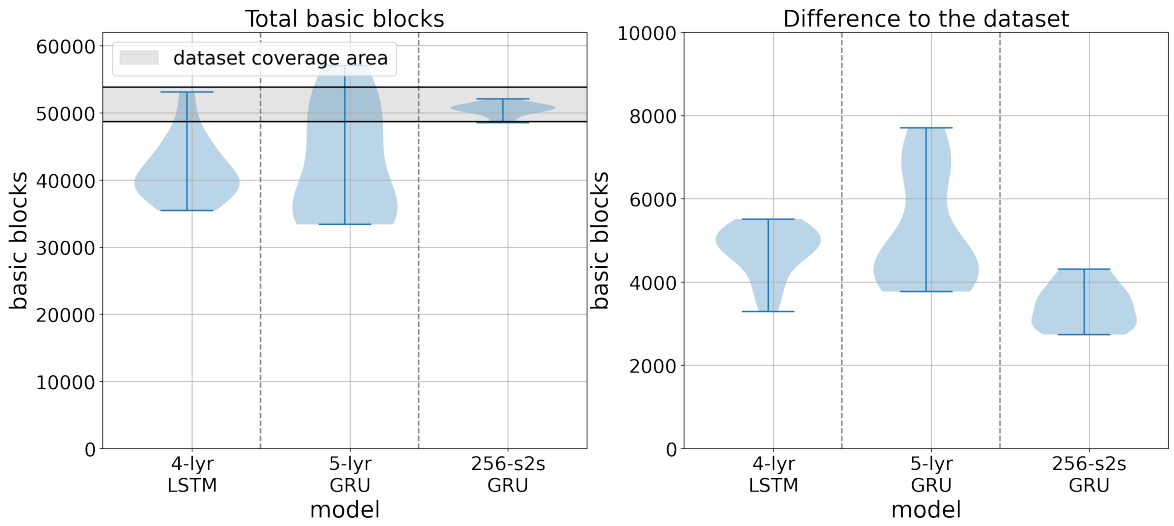

Figure 4.18: Comparison of the best performing models with 128 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset

This indicates that the validation shouldn't be the only information used during model choice. The combination of validation loss and HTML error rate provided a better information source for code coverage performance. This demonstrates the need for a second verification source that is informed about the underlying structure compared to the validation loss.

4.4 Summary

This chapter provided a performance analysis of stacked RNN architectures with two different cell types, which was followed by the evaluation of a seq2seq architecture utilizing the former best performing cell. The first part of the chapter provided a detailed comparison between the performance of stacked RNNs with LSTM and GRU cells (see subsection 4.1.4). The GRU based approach was able to outperform the LSTM in the average HTML-error rate on a per tag basis and in terms of code coverage. The GRU achieved this besides a slightly higher average validation error in direct comparison of the best performing models.

The second part analyzed the performance difference achieved by using a seq2seq based architecture with GRU cells in order to generate HTML-tags. It introduced a new method to use seq2seq models in a fuzz test scenario by sampling from the best performing candidate values (see subsection 4.2.3). This was followed by the performance metric results, which showed that the seq2seq approach with 256 internal units provides a more stable and reliable way of HTML generation. The results also highlighted that on average the seq2seq approach is able to beat both former evaluated model architectures in all provided metrics.

The combined results show that all models in both architectures were able to discover basic blocks which were not discovered by the datasets. Figure 4.18 and Figure 4.19 highlight

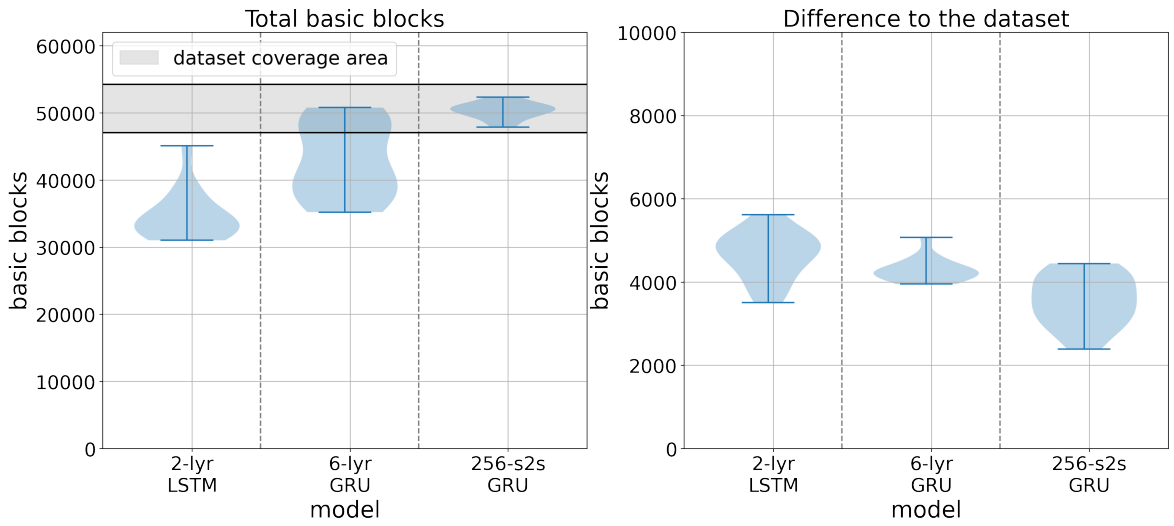

Figure 4.19: Comparison of the best performing models with 256 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset

the performance of the best performing models in this chapter in the 128 HTML-tags and 256 HTML-tags setting respectively. The graphs emphasize that the five layer stacked GRU based approach was able to outperform the seq2seq approach in the 128 HTML-tags setting. This was achieved with 6% less trainable parameters compared to the best performing seq2seq model. However, the seq2seq architecture with 256 GRU units per layer was able to achieve a very stable result during training, HTML-validation, and finally, most importantly during, the execution inside of Firefox in terms of basic blocks.

The introduced methods for training and generating test cases are still limited in several ways:

1. The training depends on the availability of an existing generation-based fuzzer to provide a training set (see in Chapter 6).

2. The modesl are not able to utilize the programs feedback in order to discover new areas of the program under test (see Chapter 7).

3. The training process and, therefore, the resulting code coverage performance are unstable for both variants of stacked models, which leads to unpredictable results (see Chapter 5).

Overall, this chapter has shown that different model architectures can perform well as HTML test case generators. Furthermore, unsurprisingly the choice of complexity and architecture has a strong impact on the performance and reliability of a test case generator during training and evaluation. This chapter also has shown that the stacked GRU based approach already achieves good performance compared to the baseline and therefor provided some initial intuition in terms of model suitability and performance expectations (**RQ1** Section 1.2).

The next chapter evaluates the performance of a different model architecture to improve the unstable training, reduce the number of parameters needed and increase code coverage performance. It proposes Temporal Convolutional Networks (see Section 2.2.4) instead of recurrent ones in order to compare their performance in comparison with the models used in this chapter and further to see whether the results provided by Bai et al. [44] are transferable to this setting.

Chapter 5

Effects of different TCN based architectures on the Code Coverage Performance

In this chapter the focus switches away from the RNN based models evaluated in chapter 4 to model architectures based on Temporal Convolutional Networks (TCNs), which were designed for generative tasks (Section 2.2.4). The chapter provides further data points to evaluate the suitability of different generative deep learning algorithms for HTML test case creation and how to chose a well performing model (**RQ1** Section 1.2).

The chapter follows the structure of the last chapter and starts in subsection 5.2.1 with a detailed description of the TCN architecture used during the experiments, which was briefly introduced in section 2.2.4. This is followed by an overview of the models' training procedure and a summary of how the trained models were used for HTML-tag generation. The first part of the chapter concludes in subsection 5.1.5 with the evaluation of the results accumulated during training, testing and execution.

The second part of the chapter starts with subsection 5.2.1 providing the description of a TCN architecture, which is closely related to the seq2seq architecture introduced in section 2.2.4 and utilized in during section 4.2. This architecture combines the advantages of the seq2seq architecture with the ability to be executed in parallel without sequential dependencies. Then the training procedure for this kind of architecture is introduced followed by the procedure for generating HTML-tags with the introduced architecture. The last part in subsection 5.2.4 provides the results in form of the defined metrics and highlights how this approach performs in comparison to the other introduced architectures.

Lastly, in section 5.4 provides a summary of the results and highlights their impact in comparison to the models evaluated in chapter 4. Furthermore, an outlook is provided for the

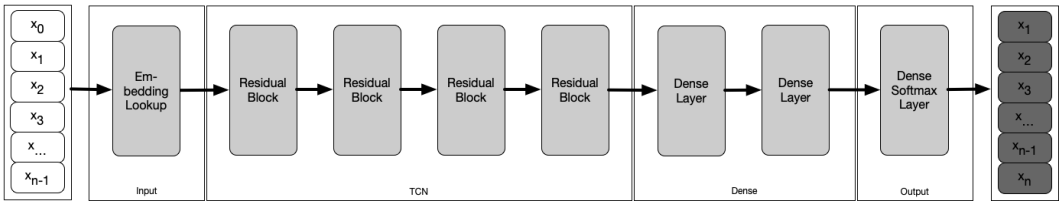

Figure 5.1: General architecture of the default TCN models

following experiments.

5.1 Default Temporal Convolutional Networks

5.1.1 Model Architecture

The architecture used for the default TCN approach followed closely the information provided by [44] introduced in Section 2.2.4. The basic model architecture is shown in Figure 5.1. This architecture consists of four modules, namely the input, the TCN, the dense and the output module. In contrast to the former models, an embedding layer was used in the input module instead of the sole one-hot coding during Chapter 4. This enabled the models to learn a dense representation of the input integer value and therefore learn to encode additional information about the input compared to a one-hot coded vector with all positions set to zero except for one.

The embedding layer translates an input integer $x_i \in \mathbb{N}$ to a vector v by

$$\mathbf{v} = (one\_hot(x_i)^\intercal * W)^\intercal, \tag{5.1}$$

with $W \in M(\mathbb{R})_{ab}$, where a is the total of number of occurring integers and b the size of the embedding and therefore of the resulting output vector v. The lookup matrix W is learned during the training process and the initial values were drawn from a normal distribution set to the interval $[-1.0, 1.0)$[1]. After applying the lookup to the whole integer input sequence and concatenating the vectors v the result was a matrix $V \in (\mathbb{R})_{bn}$ with n being the sequence length.

The matrix V was then used as input for the TCN module and were a configurable number of residual blocks were applied to it. Those residual blocks had a different dilatation rate from layer to layer in order to expand the temporal field to cover information from as many time steps as possible. However, the filter dimensions and kernel sizes used in the different layers

[1] Including -1.0 and excluding 1.0

of residual blocks was constant, resulting in

$$R_1 = res\_block_1(V) \text{parameterized by} d_1, k, f, \tag{5.2}$$

$$R_l = res\_block_{l-1}(R_{l-1}) \text{parameterized by} d_l - 1, k, f, \tag{5.3}$$

with res_block as defined in Equation (2.19), l being the number of layers, d_i the dilation rate at layer i (controlling the number of past steps taken into account), k the set kernel size and f the filter dimensions. The detailed configurations for all experiment runs are provided in Table 5.1. The causal convolutional layers (see Section 2.2.4) inside the residual block applied $relu$ (Equation (2.13)) as activation function and no activation was applied onto the residual connections.

The TCN layers were followed by two dense layers and the final output layer (as introduced in Section 4.1.1). The two dense layers also used $relu$ as an activation function, whereas the final output layer applied the $softmax$ function Equation (4.3) to ensure a valid probability distribution as output. Overall, the implemented architecture was freely configurable in the number of layers used in the TCN and dense module as well as the hyper-parameters which defined the modules (e.g., filter dimension or number of units in the dense layers).

5.1.2 Model Training

The default TCN models were trained on the same training set used during the stacked RNN approach with the same splits for validation. This makes the gathered data directly comparable with each other. In addition, the same loss function was applied to train the TCN models. During the training process, early stopping was applied when the validation loss did not improve for five consecutive epochs. In total, 120 models were trained. The sequence length was set to 200 because the TCN solely relies on the input to determine the retrieved information from the past and predict the output it was necessary to provide a long enough lookup to determine the critical features.

The parameters and number of layers were chosen to make sure that the models were able to capture the whole input sequence. This enabled the last character prediction in the output sequence to be influenced by the first character in the input sequence. In addition, the goal was to keep the number of trainable parameters in an area that is either below or close to the best-performing models introduced in Chapter 4. In total, 8 different configurations were trained. The embedding size was set to 128 dimensions as was the filter dimension for all configurations. Each TCN configuration was tested with two different configurations for the dense layers. First, the dense layers were set to 512 and 256, and second to 1024 and 512 units.

config	k	d_1	d_2	d_3	d_4	d_5	d_6	d_7	$dense_1$	$dense_2$
01	3	1	2	4	8	16	32	64	512	256
02	3	1	2	4	8	16	32	64	1024	512
03	5	1	2	4	8	16	32	–	512	256
04	5	1	2	4	8	16	32	–	1024	512
05	9	1	2	4	8	16	–	–	512	256
06	9	1	2	4	8	16	–	–	1024	512
07	18	1	2	4	8	–	–	–	512	256
08	18	1	2	4	8	–	–	–	1024	512

Table 5.1: The different configurations used during training. The kernel size k is fixed over all convolutional layers (max. 7) and the dilation rate d_i is adjusted to cover the whole input sequence (max. 200 characters). Furthermore $dense_i$ provides the number of internal units used.

The configuration for the residual blocks varied from 7 to 4 layers. In all configurations, the settings in terms of kernel size and corresponding dilation rate were chosen to ensure that at the maximum sequence length, the final output still receives information from the first input in the sequence. This resulted in a starting kernel size of three as recommended for the character-based approach in [44] and ended with 18. Since the kernel size and dilation rate together the window into the past of the sequence, an increasing kernel size leads to a lower dilation rate as seen in Table 5.1. Therefore, the kernel size was adjusted accordingly and with the number of layers applying dilated causal convolutions to ensure that the whole input sequence is considered. Leading to a final kernel size of 18 with only four layers of residual blocks. In contrast to default multi-dimensional CNNs (see Section 2.2.3) the number of parameters does not grow exponentially. The dilation rates were increased exponentially, as suggested by Bai et al. [44], which also increases the time window taken into account. The time window taken into account by a layer is defined as $(k-1)d$.

5.1.3 HTML-tag Generation

The saved model checkpoints were restored and used for HTML-tag generation. The basic approach with the default TCN architecture was the same as described in subsection 4.1.3, here basic approach means that the model was used to provide the probability distribution for the next character in the sequence. The procedure described in Algorithm 3 describes how HTML-tags were generated by the default TCN models. The lack of the hidden state makes it necessary to grow the models input sequence after each sampling step. In practice, a maximum sequence length was set at 250 characters after reaching this maximum the input sequence x was cut down to the last 200 sampled characters. This avoided a continuously increasing computation time, which would end in a memory exhaustion.

As in chapter 4 each of the trained models were used to generate $16,384$ HTML-tags for the

Algorithm 3 Generate a single HTML-tag with a default TCN model

```
1: procedure DEFAULTTCNHTMLTAGGENERATION(model, maximum)
2:     x ← " < "
3:     n ← 0                                   ▷ Number of sampled tags
4:     while n ≤ maximum do
5:         y ← model.predict(x)
6:         x ← x + y                           ▷ String concatenation
7:         n ← count(" \ n", x)                ▷ Count the newline characters
8:     end while
9:     return x
10: end procedure
```

use during the code coverage collection.

5.1.4 Additional Comparison

This chapter also utilizes an extended experimental setup. Libxml2 ([80]) is introduced as a second target to gather code coverage from to demonstrate the transferability of the results. It was chosen because it contains an HTML parser and can be invoked through a simple program harness. Furthermore, it is a relevant library since it is used in the GNOME desktop manager, besides various other use cases.

The already sampled test cases can be reused to gather code coverage from the program. In addition to introducing another target for the test, this chapter also introduces a second baseline. Namely, AFL++ [81] is used to generate a comparison baseline. AFL++ is the continuation and improvement of AFL [18] and the state-of-the-art mutation-based fuzzer.

AFL++ was set into FRIDA mode which enables code coverage collection based on basic blocks in DrCov format. Therefore it allows a direct comparison with the existing code coverage collection method using DrCov. The first ten test cases from the test set (see Section 3.4) were used as seed examples for AFL++. This was done to provide the same base values for AFL++ and the models. AFL++ was run three times on the target.

The first run was until 15,000 executions (AFL++ 15,000) were reached to provide a lower-end baseline. This lower-end baseline still has more than seven times the number of test cases compared to the total number of test cases generated by all TCN models combined. The second (AFL++ 24h) and third (AFL++ 48h) runs were timed for 24 hours and 48 hours, respectively, to highlight the development of the code coverage over time. Further, it is important to note that AFL++ is a highly optimized parallel fuzzing framework that has been actively developed for over a decade. In contrast to the model-based test case generation approach proposed in this thesis.

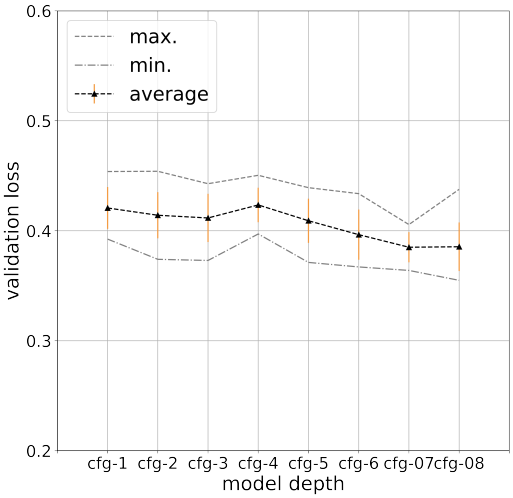

Figure 5.2: Average validation loss of the default TCN models with error-bars indicating the standard deviation across the individual models

5.1.5 Results

The training phase has shown a very stable validation loss over the different configurations as highlighted in Figure 5.2. The general trend over the different configurations shows that a larger kernel size and less layers leads to lower average validation loss. Furthermore, the smaller number of internal units in the dense layers lead to no conclusive result, but the trend in three out of four comparisons shows that more units result in a higher standard deviation. The configuration number 7 has achieved the lowest average validation loss and standard deviation with values of 0.385 and 0.0137 respectively. Since the default TCN approach and the stacked RNN one used the same training set with the same validation splits, those results are directly comparable and show that the default TCN approach outperforms the stacked RNN. In particular the lowest average validation losses show a difference of 0.15 comparing the default TCN with both stacked RNN approaches presented in Chapter 4. In terms of standard deviation the GRU based stacked RNNs achieved lower values with 0.00659 being the smallest achieved standard deviation. Table 5.2 provides a summary of the training results highlighting the minimum error rates for the TCN models. Furthermore, the provided training times highlight that the TCN models are trained quicker on average, even though they have more parameters (see Table 4.1). This is a direct consequence of having no recurrent connections in the TCN models.

The quality of the HTML-tags generated by the default TCN models is highlighted by the Listings 5.1 and 5.2, providing examples from a configuration 1 and 7 model respectively. This shows that there is no remarkable difference in the quality of the generated HTML-tags even though that the difference in average validation loss is 0.04, where a difference of 0.02 already lead to visible quality difference in the LSTM-based stacked RNN (see Sec-

Config	Training loss	Validation loss	Training steps	Duration (minutes)	Parameters
cfg-1	0.3411	0.3924	12267	33	700,011
cfg-2	0.3104	0.3740	14050	49	1,186,923
cfg-3	0.3122	0.3730	14617	40	830,699
cfg-4	0.3103	0.3971	11133	39	1,317,611
cfg-5	0.3124	0.3712	13900	37	1,059,691
cfg-6	0.3043	0.3671	15150	49	1,546,603
cfg-7	0.3063	0.3639	13467	34	1,485,291
cfg-8	0.3001	0.3549	12883	44	1,972,203

Table 5.2: Summary of the training results of the TCN models. It provides the training and validation minimum loss, the average number of training steps, the average training time in minutes, and the number of trainable parameters.

```
1   <var id="id138064" accesskey="3" translate="yes" class="
        style_class_0" dir="ltr">
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</var>
```

Listing 5.1: Example HTML-tag from a config-01 model

tion 4.1.4).

The good quality of the generated HTML-tags is further supported by the results of the HTML-validation shown in Figure 5.3. It highlights that all configuration were able to learn the underlying structure of the training set and in addition are able to generate HTML-tags with a lower error than the underlying fuzzer. It is also noteworthy that the default TCN models did not create HTML-tags comparable to the 1024 units seq2seq case, which were syntactically well formed but used non-existent attribute names and values. Overall. the HTML-validity over the different configurations is more similar to the seq2seq approach than the stacked RNN approach, with the smallest configuration already being able to generate HTML-tags reliably regarding the error rate. This is achieved with $1,287,345$ less trainable parameters than the seq2seq model with 256 units (compare Table 5.2 and Table 4.4) and $1,182,976$ parameters difference compared to the 5-layer stacked RNN approach.

The code coverage performance in the scenario with 128 HTML-tags per test case showed a very close performance to the underlying dataset, as shown in Figure 5.4. The different configuration tested showed a very similar performance in terms of absolute number of triggered basic blocks. In comparison to the previous provided results there is no single model clearly outperforming the others and additional there are only four trained models in total that are performing below average with close to $40,000$ basic blocks. Figure 5.4 also shows that the reduction of layers and simultaneous increase of the kernel size leads to a more stable performance. The configurations utilizing the larger dense layers (see Table 5.1) were able to achieve on average a higher performance. The single best performing default TCN

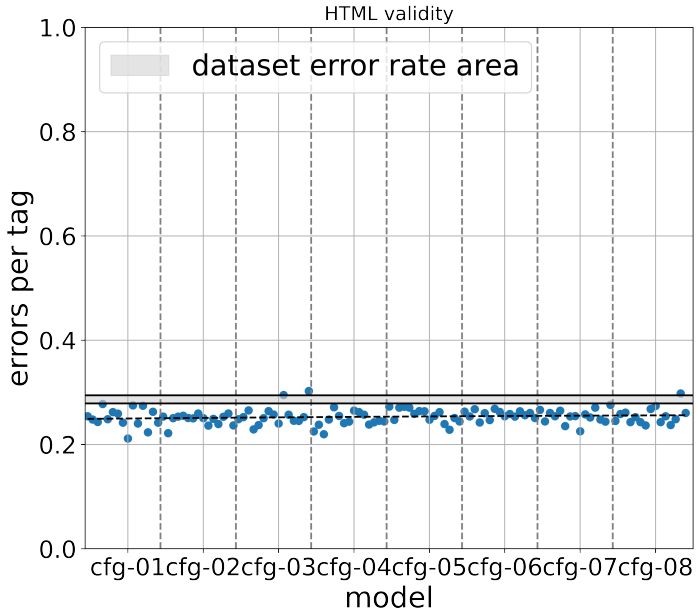

Figure 5.3: Error rate per HTML-tag in regards to the model configuration in terms of kernel size and dilation rate

```
1 <textarea id="id61134" lang="lo" autofocus placeholder="{}" max="
      2200000000" rows="-1e6" form="id6810"> 5e6</textarea>
2 <meter id="id61344" spellcheck="true" low="1" value="4400000000">
      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</meter>
```

Listing 5.2: Example HTML-tag from a config-07 model

model used configuration six and achieved to trigger $53,580$ unique basic blocks, which is only 242 basic blocks short of the best performing dataset. In terms of difference to the best performing dataset all models were able to discover new unique basic blocks not triggered by the dataset. The highest difference was achieved by the best performing default TCN model with $5,915$ newly discovered basic blocks.

The test sets with 256 HTML-tags per case performed very similar, however in this setting the performance was more distributed than before, as highlighted by Figure 4.14. The figure also shows that on average the models were able to increase the code coverage. Furthermore, the on average performance of all models was again inside the datasets' coverage area. Similarly to the 128 HTML-tags per case setting there is no single model clearly outperforming the others. The best performing model utilizing configuration five achieved to trigger $53,969$ unique basic blocks compared to $54,221$ achieved by the best performing dataset. The difference between the models and the best performing dataset averages at approximately $4,000$ basic blocks. The highest difference to the best performing dataset was $5,907$ basic blocks achieved by a configuration one model. Furthermore, the overall best performing configuration five was $5,661$ basic blocks.

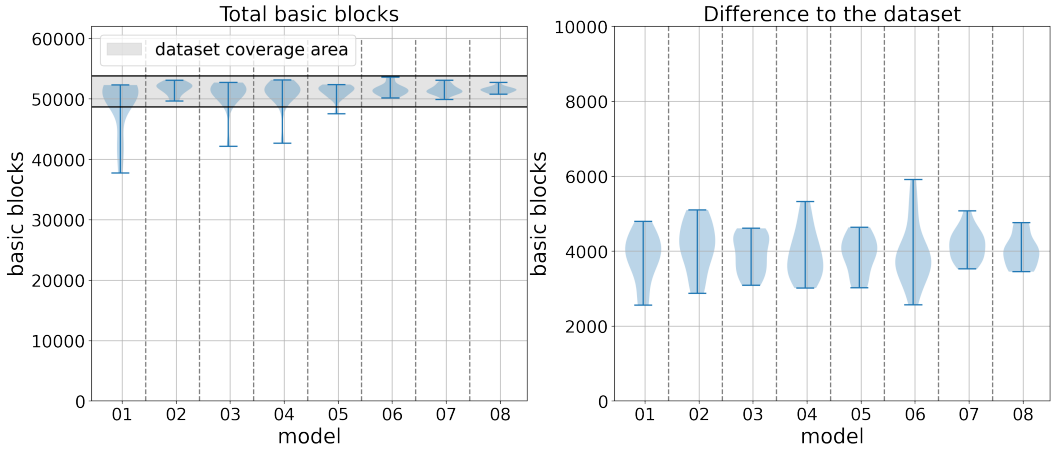

Figure 5.4: Default TCN: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

Test set:	dataset	1.6%	3.2%	6.4%	12.8%	25.6%	51.2%
Overlap 128:	88.6%	92.2%	91.1%	86.5%	69.5%	35.7%	20.3%
Overlap 256:	89.0%	92.4%	92.5%	89.6%	58.2%	47.1%	16.0%

Table 5.3: Best performing default TCN model's overlap with the best performing dataset and the differently mutated sets.

Compared to the mutation set the best performing model in the 128 HTML-tags per case the overlap varied from 92.2% to 86.5% for mutation chances less than 10% the steep decline in overlap already observed in Section 4.1.4 and Section 4.2.4 was also observed here, as highlighted in Table 5.3. In the 256 HTML-tags per case setting the same observation were made with a larger decrease in the step to mutation chances larger than 10%. For both settings this emphasizes the capability of the models to trigger unique basic blocks not covered by the dataset or the differently mutated datasets.

Overall the results highlight the stable performance achieved by the default TCN models. In comparison with the seq2seq models (see Section 4.2.4) this performance is achieved with 25% less trainable parameters and an on average higher performance at the same time. It highlights the capability of the default TCN architecture to learn the underlying structure and reproduce it in a reliable way. Furthermore the results reinforce the former made observations that the performance of reliably performing models is transferable between the 128 HTML-tags per case scenario to the 256 HTML-tags one.

Libxml2 results

The results highlighted in Table 5.4 show that AFL++ achieved the highest amount of unique basic blocks. It is important to keep in mind that AFL++ was able to execute on average

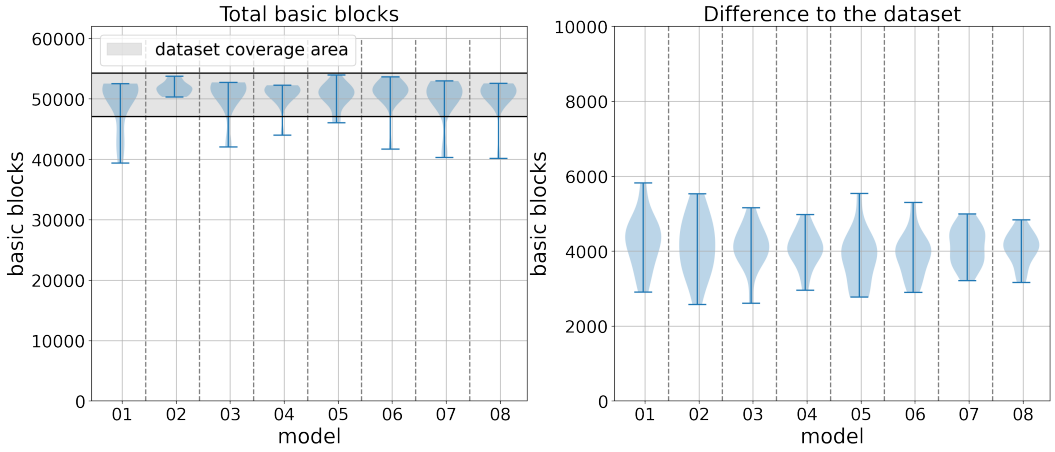

Figure 5.5: Default TCN: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

Model:	**1**	**2**	**3**	**4**	**5**
Total:	3,953	3,836	3,842	3,903	3,825
Model:	**6**	**7**	**8**	**9**	**10**
Total:	3,886	3,560	3,494	3,503	3,532
Model:	**11**	**12**	**13**	**14**	**15**
Total:	3,512	3,487	3,551	3,478	3,484
Model:	**Combined TCN**	**Dataset**	**AFL++ 15,000**	**AFL++ 24h**	**AFL++ 48h**
Total:	4,048	3,225	3,624	4,531	4,744

Table 5.4: Comparison between the TCN models' performance with the dataset and AFL++ runs in total basic blocks.

between 80-120 test cases per second. So after two seconds, AFL++ has already executed nearly twice the number of test cases a single model has created. The model-generated test cases were able to achieve on average 3,656 basic blocks which is slightly above the AFL++ 15,000 performance with 3,624 basic blocks. Combining all the models' test cases results for 1,920 test cases and 4,048 basic blocks which is higher than the AFL++ 15,000 baseline but below the other baselines of AFL++ 24h and AFL++ 48h with 4,531 and 4,744 basic blocks respectively. The best single model performance peaked at 3,953 basic blocks with just 128 test cases. This is in comparison to approximately 15.5 million and 87.2 million executions performed by AFL++ 24h and AFL++ 48h. Those numbers highlight the large difference in the number of test cases executed. In total, 1,920 model-based test cases were executed compared to 15,000 in the shortest AFL++ run.

In comparison to the dataset with 3,225 basic blocks, all models were able to outperform the dataset results. This highlights that the earlier gathered results in the Firefox case can be transferred to the libxml2 setting.

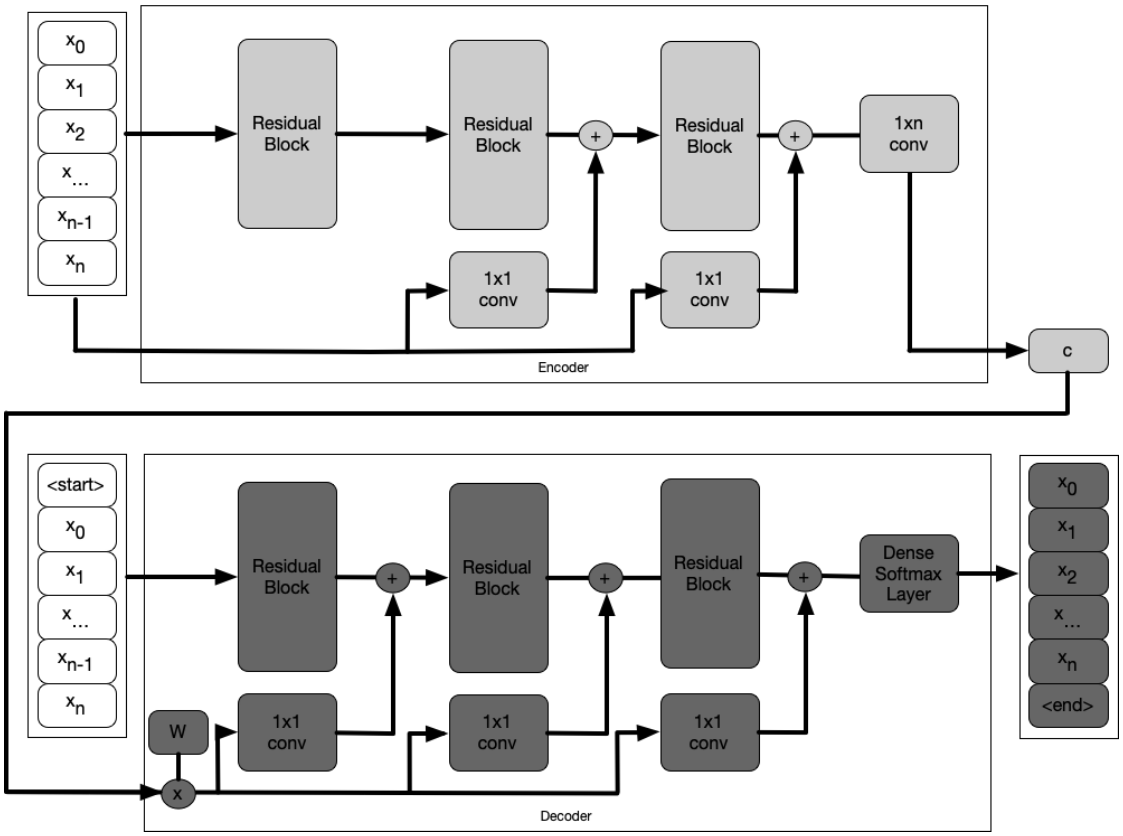

Figure 5.6: Seq2Seq TCN architecture overview

5.2 Sequence-to-Sequence Temporal Convolutional Network

5.2.1 Model Architecture

The architecture used and evaluated in the following sections is called Sequence-to-Sequence Temporal Convolutional Network (Seq2Seq TCN). It is a model architecture using residual blocks with dilated convolution as described in Section 2.2.4, direct connection between layers comparable to those inside a residual block and the concepts of classical Seq2Seq models introduced in Section 2.2.4 and utilized in Section 4.2.

First, an embedding layer (see Equation (5.1)) was applied to the integer input sequence. The resulting vectors were again concatenated and used as input for the residual blocks. After the second layer of residual blocks the embedding input was added to the residual blocks result in order to provide a direct connection from the input to a layer's result. The result of the encoder's residual blocks was used to compute a vector c via $1 \times n$ convolutional layer. Therefore, c represented the embedding of the whole input sequence into a latent space.

Secondly, the decoder used the a starting symbol and the vector c to predict the next character in the sequence. This character was concatenate with the decoder's input and used as

new input together with the same vector c until an end symbol was predicted. In order to strengthen the influence of c for the resulting prediction c was scaled back to the sequence length by multiplying it with a learned matrix W. The result of that multiplication was then fed into a 1×1 convolutional layer to scale the dimension accordingly to the corresponding residual block's output dimension. The final output layer was a dense layer with the number of units set to the number of different prediction classes and the $softmax$ Equation (4.3) function used as activation.

5.2.2 Model Training

The training process of the Seq2Seq-TCN is very similar to the GRU-based seq2seq training approach described in subsection 4.2.2. In particular, the same training set representation, loss function (including the application of masks) and optimizer were used during the training. The main difference is the direct application of learning rate decay with a rate of 0.96 every $15,000$ steps, which follows the results mentioned in Section 2.2.4.

The configurations used during training for the encoder and decoder are the same as shown in Table 5.1. So, the Seq2Seq TCN approach used two full TCNs instead of one with the added residual connection mentioned in Section 5.2.1. Configurations one to four used only a filter size of 64 to make the models fit into the memory. Further, configurations five to eight used a filter size of 128 again. The maximum sequence length had to be adjusted to 250 because it depended on the training set. The configurations were chosen to ensure a large enough window into the past sequence and to analyze the effects of varying model depth. The time steps taken into account by the model for the output at time t depend on the kernel size and dilation rate combination.

For each configuration shown, three runs on five different splits were trained. This resulted in a total of 120 models, which were evaluated in terms of code coverage performance.

5.2.3 HTML-tag Generation

The HTML-tag generation followed the algorithm introduced in Section 4.2.3 especially with regards to the candidate value handling. However, it was necessary to adjust it to the different model architecture and the lack of a hidden state providing information about the past. In order to be able to generate the maximum sequence length the input seed "$<$" was padded with $<$ PAD $>$ symbols to the maximum sequence length. The padding allowed to sample HTML-tags in batches, since all HTML-tags had the same length over all sampling steps even when one HTML-tag already contained an $<$ END $>$ symbol.

During the sampling process the first occurrence of a padding symbol in the input sequence was replaced by character sampled from the probability distribution provided by the model's output for that particular position. This was done until either the model predicted an $<$ END $>$ symbol or the maximum sequence length was reached. One problem encountered during sampling was that the seed value "<" led the model to predict a "/" character with such a confidence close to 1 and therefore predicted a following closing tag and an $<$ END $>$ symbol. In order to prevent this the generation procedure checked whether the predicted character is second character in the sequence and modified the resulting probability distribution by setting the probability of the "/" character to 0 and distribute its original value evenly over all other classes.

In addition, the prior observed necessity (see Section 4.2.3) to manipulate the candidate value by sampling a new value from an area around the provided candidate values in order to avoid sampling repeatedly similar HTML-tags (especially the same opening and closing HTML-tag) was not observed while sampling from the TCN based seq2seq models.

5.2.4 Results

The lowest average validation loss was achieved by the models with configuration four. The general observation that removing layers and instead increasing the kernel size leads to lower validation loss (see Section 5.1.5) could not be confirmed with the seq2seq TCN architecture as is highlighted in Figure 5.7. The number of units inside the final two dense layers impacted the training positively when increased for configurations one to four. Overall all configurations are very close together regarding the average validation loss with a maximum difference of 0.08, which is less than for the GRU-based seq2seq models. It is also noteworthy that configuration seven is able to achieve a comparable average validation loss to configuration four even though the configurations in between had an increased loss value. Furthermore, the average loss of configuration eight models is below the losses for configurations five and six. The number of parameters in configurations five and six is comparable to configurations seven and eight, respectively (see Table 5.5). The average loss results for those models demonstrate that removing a layer at that parameter range has a positive impact on training performance. One model with configuration seven achieved the overall lowest validation loss. The standard deviation of the models varied widely, with a maximum of 0.078 for the models using configuration one and the lowest value achieved by the models using configuration three with 0.021.

Listings 5.3 and 5.4 show excerpts from models trained with configuration four and one respectively. All HTML-tags are well formed especially they are opened and closed correctly. In addition, there is no visible difference in quality as for the GRU based seq2seq models. This also highlights the capability of the seq2seq TCN models to generate the underlying

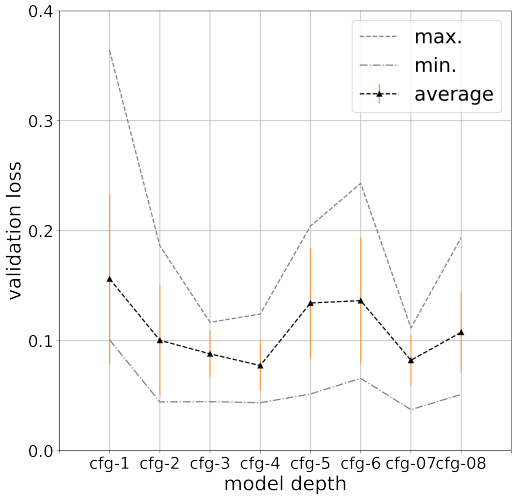

Figure 5.7: Average validation loss of the default TCN models with error-bars indicating the standard deviation across the individual models

Config:	config-01	config-02	config-03	config-04
Parameters:	1,553,390	1,980,142	1,622,510	2,049,262
Config:	config-05	config-06	config-07	config-08
Parameters:	3,353,902	3,813,422	3,590,446	4,049,966

Table 5.5: Trainable parameters by model depth for Seq2Seq TCN based architectures

structure correctly compared to the stacked RNN with LSTM cells where a small difference in average validation loss had a large impact on the resulting HTML (see Section 4.1.4) or also the GRU based seq2seq models (see Section 4.2.4). The HTML-validation confirms the overall good quality of the generated HTML-tags with an average performance of all models below the datasets' HTML error rate as highlighted in Figure 5.8. Furthermore, there was no model generating HTML-tags comparable to the GRU based seq2seq models with 1024 units (see Listing 4.10), which kept the structure but generated non-existent HTML-tags. Furthermore, this was achieved with a comparable number of trainable parameters in regards to the GRU based seq2seq models with 256 units and less parameters than in the 512 units models.

In terms of code coverage performance the collected coverage data overall shows that it is comparable to the stacked RNN approach (see Section 4.1.4). The generated test cases were able to trigger an amount of basic blocks close to the dataset. However, the performance is much more distributed between the different training runs than in default TCN setting, as

```
1  <nav id="id181820" style="style" tabindex="1" spellcheck="false"
       title="uneval(n1)"> false</nav>
2  <progress id="id181853" max="25e6" value="-4500000000" value="5">
       BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</progress>
```

Listing 5.3: Example HTML-tag from a config-04 Seq2Seq TCN model

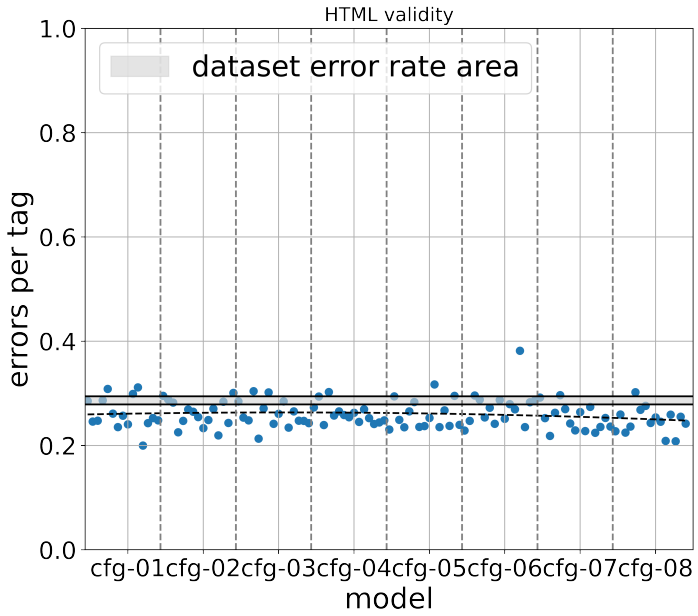

Figure 5.8: Error rate per HTML-tag in regards to the model configuration in terms of kernel size and dilation rate

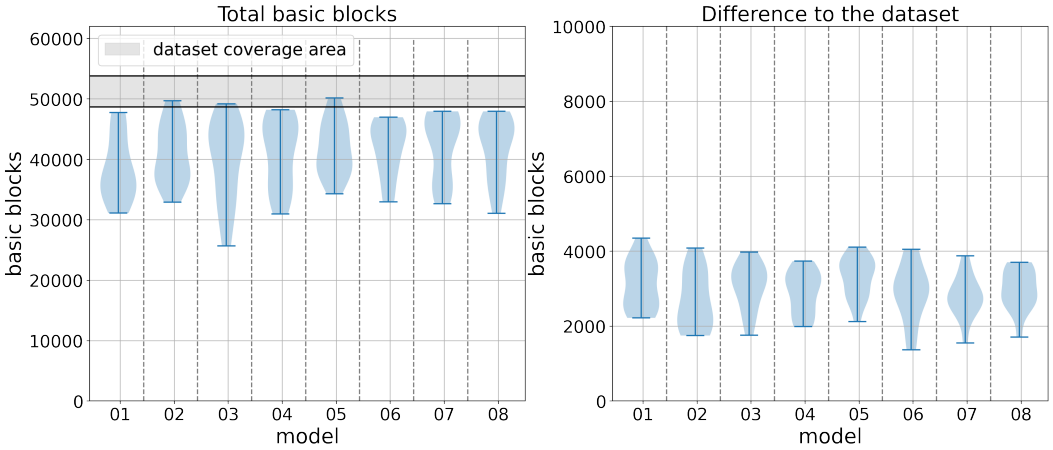

Figure 5.9: Seq2Seq TCN: Code coverage performance with 128 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

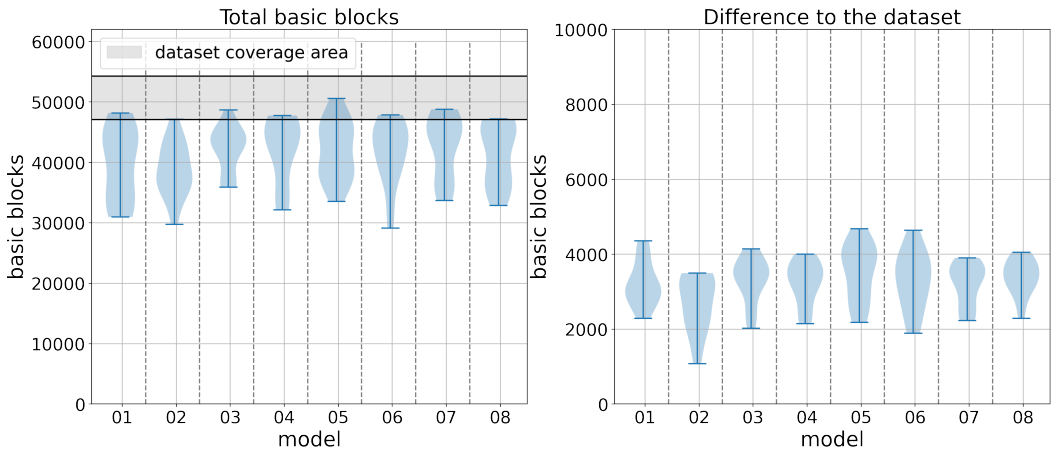

Figure 5.10: Seq2Seq TCN: Code coverage performance with 256 HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

shown in Figure 5.9. The test cases with 128 HTML-tags per case were able to reach the lower boundary of the datasets' coverage area and most configurations triggered on average a number of basic blocks between $40,000$ and $50,000$ except for configuration one. The overall highest number of basic blocks was discovered by a configuration five model with $50,186$ basic blocks, which was inside the datasets' coverage area. All model configurations were able to execute basic blocks not discovered by the best performing dataset. The average difference between the models and the dataset was approximately 3000 basic blocks. The models with configuration one were able to discover the largest difference with the best performing model triggering 4346 basic blocks not included in the best performing dataset. The performance of the models during the 256 HTML-tags per case setting was comparable to the former setting, as highlighted by Figure 5.10. The average performance over all models was still in the area between $40,000$ and $50,000$ basic blocks. The best performing configuration five was able to improve the performance slightly in terms of absolute discovered basic blocks to $50,555$. The difference between the models and the best performing dataset kept the average at approximately $2,500$ basic blocks with configuration six achieving a $4,742$ basic blocks maximum difference closely followed by configuration five $4,721$ basic blocks.

The overlap between the best performing model and the dataset is close to the values observed earlier with 86.8% and 87.1% for the 128 and 256 HTML-tags setting respectively. Compared to the mutation set the observation made during the other test runs (see Sec-

```
1   <select id="id224382" lang="se" accesskey="Ü" translate="no"
        lang="te" class="style_class_0">
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</select>
2   <ul id="id223537" translate="yes" dir="ltr"> -4400000000</ul>
```

Listing 5.4: Example HTML-tag from a config-01 Seq2Seq TCN model

Test set:	dataset	1.6%	3.2%	6.4%	12.8%	25.6%	51.2%
Overlap 128:	86.8%	95.5%	94.2%	90.3%	72.31%	36.0%	20.3%
Overlap 256:	87.1%	96.5%	96.2%	93.2%	60.3%	49.3%	16.9%

Table 5.6: Best performing seq2seq TCN model's overlap with the best performing dataset and the differently mutated sets.

tion 5.1.5 and Chapter 4) regarding the steep drop in overlap after exceeding 10% mutation chance was also made for both settings in seq2seq TCN models case, as shown in Table 5.6. The overlap with the data set is at approximately 87% for both cases. The high overlap with the 1.6% mutation chance still left a difference of $2,258$ and $1,754$ basic blocks for the 128 HTML-tags and 256 HTML-tags per case respectively.

Overall the seq2seq TCN models did not perform as well as the default TCN (see Section 5.1.5) or the GRU based seq2seq models (see Section 4.2.4). They were still able to discover code paths not included in the dataset, but the overall performance was comparable to the stacked RNN approach, with a wider performance distribution over the configurations than the former better performing approaches. However, those results reinforce the observation that stable training in combination with a low HTML-error rate leads to a transferable performance between the settings with 128 HTML-tags and 256 HTML-tags per case.

5.3 Discussion

The results indicate that the stacked TCN approach is able to outperform both of the RNN based architecture evaluated in Chapter 4. This might be due to the fact that the RNN based architectures rely on encoding the past information into the hidden state and therefore also loss parts of the input information. Whereas the stacked TCN approach has access to all past values at every step and can directly use that information to guide the output. This is also supported by the lower performance of the TCN based seq2seq architecture which has a lower code coverage performance than the stacked TCN and RNN seq2seq architectures.

The stacked TCN approach demonstrated a higher suitability for the task of creating HTML-tags than the other models tested, which is also achieved with a considerably lower amount of parameters. It indicates that the convolution operation inside the residual blocks of the TCN capture the input structure more reliably than the GRU or LSTM cells with their fully connected gates. In addition, the gates storing and removing information from the hidden state might not reflect the real value of the information compared the impact on the validation loss. The performance of the seq2seq TCN based models compared to the default TCN approach indicates that the seq2seq architecture is not well suited for the TCN modules. The reason might be the lack of gates controlling the flow of information into the candidate value.

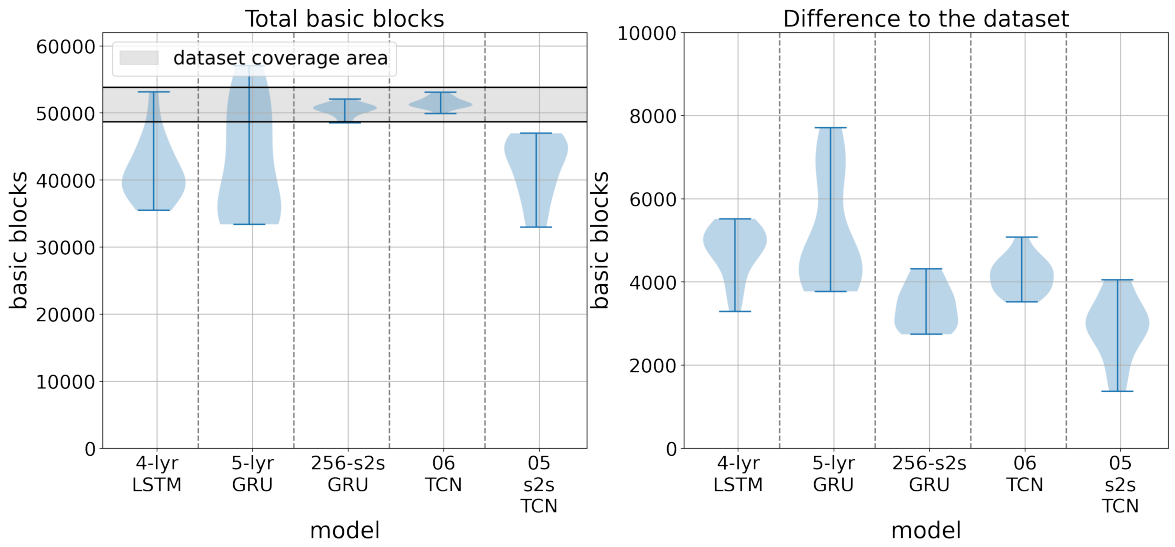

Figure 5.11: Comparison of the best performing models with 128 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset

Overall, the results further demonstrate the connection between a stable training performance with a low validation loss, a low HTML error rate and a stable code coverage performance, which provides more evidence that the validation loss together with a low HTML error rate can be used as a good indicator of code coverage performance.

5.4 Summary

This chapter focused on providing additional data on the suitability of generative deep learning architectures to generate HTML test cases (**RQ1** Section 1.2).

The results provided in Section 5.1.5 highlight the capability of a default TCN to generate HTML in a reliable way. It is important to notice that if the kernel size and dilation rate are chosen accordingly to receipt a large enough window into the past the quality of the resulting HTML and the code coverage performance only varies slightly. In contrast to the seq2seq models with 1024 units used in Section 4.2 the default TCN models do not only capture the underlying structure but also still are able to generate executable HTML and the correct structure.

In Section 5.2 a model was introduced and evaluated which tried to utilize the seq2seq architecture in a non-recurrent model design. The results have shown that this mixture model is trainable but neither able to perform as well as the default TCN nor the seq2seq models. The TCN seq2seq models were not able to deliver a stable code coverage performance and triggered on average less basic blocks than the datasets. The main takeaway from those results was the reinforcement transferability of code coverage between the 128 HTML-tags and 256

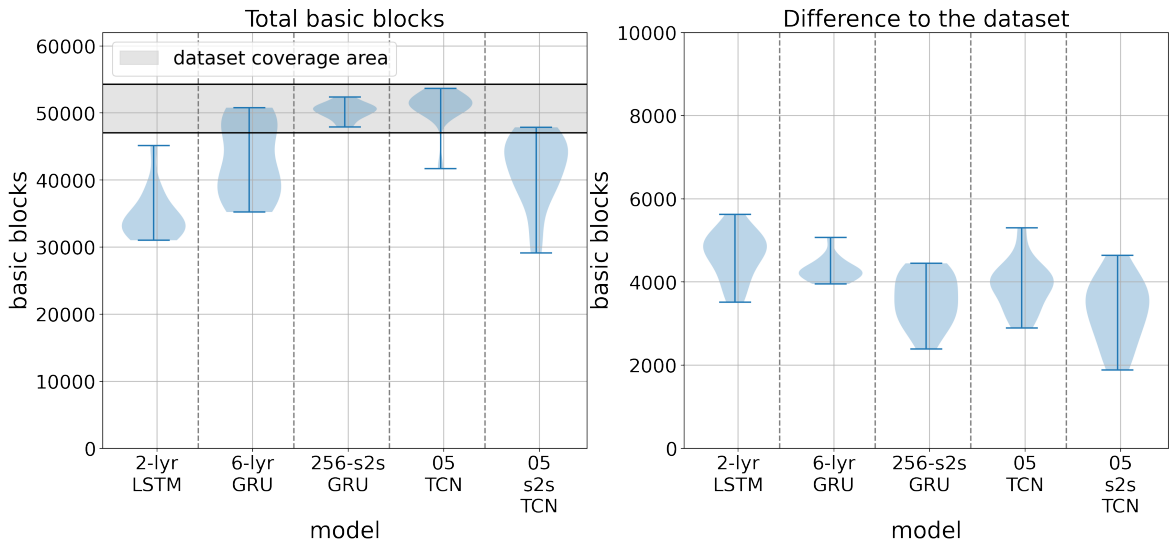

Figure 5.12: Comparison of the best performing models with 256 HTM-tags per file. (left) total number of basic blocks achieved, (right) difference to the testset

HTML-tags per case settings.

Figure 5.11 and Figure 5.12 provide an overview of the best performing models so far. The graphs emphasize the stable performance of the default TCN based approach. Furthermore, it shows that the five layer stacked GRU models were able to achieve a higher performance but with the downside of an unstable performance compared to the default TCN. The both seq2seq approaches are neither competative in terms of stability nor number of trainable parameters with the TCN seq2seq approach also not being competative in terms of basic blocks. The default 05-TCN model achieved this performance with 1,059,691 trainable parameters this is 47% and 68% less parameters than in the recurrent seq2seq (Table 4.4) and TCN based seq2seq (Table 5.5) respectively.

The approaches evaluated in this chapter also have shown that although the average validation loss and HTML-error rate is an indicator for the code coverage performance. It is difficult to make a model choice purely on those indicators without testing a range of model configurations. However, the architecture introduced and evaluated in Section 5.1 has clearly shown that a stable training performance leads to a stable code coverage performance. Therefore, it solves the the problem of unstable training and code coverage performance all the other models suffered from. So, two problems are still remaining, namely the dependency on an existing fuzzer to provide a training set for the models and that the collected feedback form the program is not utilized to discover new program areas. Overall, this chapter demonstrated that the default TCN model can be trained reliably and provides a stable performance in terms of code coverage. Furthermore, a low validation error already provides a strong signal that the code coverage performance will be stable. Therefore the default TCN has been shown to be a suitable model for HTML test case creation (**RQ 1** Section 1.2).

The next chapter will analyze the performance of the default TCN models using a real world data with a varying degree of preprocessing applied in order to evaluate how readily available real world data can be applied and utilized for the training of test case generators. This means the next chapter provides a way to avoid the dependency on an existing fuzzer.

Chapter 6

Effects of using real world training data on the code coverage performance

This chapter evaluates how real world training data affects the code coverage performance of the TCN-based generator models. Throughout this chapter default TCN models (see Section 5.1) are used to generate HTML. They have shown a stable combined training, HTML validity and code coverage performance, as described in Section 5.1.5. First the real world dataset is introduced in Section 6.1. The source of the data is described and the modifications made to use it as a training set are explained. Secondly, the differences during the training phase are highlighted and the reasons for those are provided in Section 6.2. In addition, the new configuration settings are explained to provide an overview of the expanded training process. Thirdly, the HTML generation process is explained and the differences to prior approaches are highlighted in Section 6.3. Fourthly, the results are provided in a threefold way in Section 6.4. The first part Section 6.4.1 describes the performance of models trained on minimal preprocessed data (see Section 6.1) and the second part evaluates (Section 6.4.2) the effects a further preprocessed training set has on the final performance of the models. The third and final result part (Section 6.4.3) highlights the influence of the sampling strategy on the code coverage performance. Fithly, the results are discussed in Section 6.5 focussing on providing an explanation for the additionally triggered basic blocks and the influence of the sampling strategy. Finally, the results are summarized and the lessons learned are highlighted in Section 6.6

6.1 Real World Dataset

The dataset consisted of real world web data obtained in September 2018 from the Common Crawl project [82]. This project provides web crawl data, i.e. downloaded websites including HTTP-headers, which can then be used in projects. From the project a total of 8.3GB of data containing websites was downloaded, which equates to one archive. This data was converted into a dataset by first removing any protocol data related to the crawling and HTTP protocol, like WARC-format headers and HTTP request. Secondly, the files were scanned for websites from those websites the unnecessary parts including the JavaScript, header fields, links, meta information and empty lines were removed. Lastly, the final data was merged into a single file.

```
1 <strong>Download the free report now </strong>
```

Listing 6.1: Example HTML from the Common Crawl project before additional preprocessing was applied.

The resulting data was also converted to an integer sequence, this resulted in a vocabulary size of 256, because the dataset contained unicode characters (i.e. Chinese characters), whereas the generated dataset only contained a subset of ASCII characters (see Section 3.2).

The resulting code coverage performance is described in Section 6.4.1. In combination with the large vocabulary size and the mix of different languages the results lead to the assumption that more preprocessing is necessary to improve the performance. In addition, to the above mentioned steps the dataset was parsed for text sequences that are not part of a HTML-tag (including attributes). Listing 6.1 shows HTML from the dataset as it was used during the first set of experiments and Listing 6.2 shows the same part after the additional preprocessing. The text enclosed by $< strong >$ has been replaced by "AAAA". This also removed characters, which are not in the Latin alphabet, i.e. Chinese and Korean symbols. The additional steps taken decreased the vocabulary size to 199 from 255 characters and the corresponding results are described in Section 6.4.2.

Overall, the first training dataset with minimal preprocessing contained 272MB of HTML and the second one with additional preprocessing contained 368MB.

```
1 <strong>AAAA</strong>
```

Listing 6.2: Example HTML from the Common Crawl project after additional preprocessing was applied.

6.2 Model Training

The first batch of models was trained using the configurations from Table 5.1, because those have shown a stable performance over all the configurations using the first dataset with less preprocessing. The sequence length was modified from 200 to 250 to cover also longer sequences especially in the dataset with less preprocessing. All models were again trained for three runs on five different randomly chosen non overlapping splits which yielded 120 models. In addition, the models were not trained on the whole dataset, instead a subset of 50MB was used. This was necessary because initial training runs showed that the varying languages in the dataset lead to models generating ”mixed language” text and very close to none HTML. For the second training cycle only the on average best performing model was taken into account and trained three times on each of the five splits resulting in 15 additional models.

6.3 HTML generation

The strategy to generate HTML differed from the strategies described earlier (see Algorithm 3). This was mainly because of the different structures of the underlying datasets, where the fuzzer generated datasets had a fixed structure (i.e. one HTML-tag per line), the real world based dataset had no underlying structure except for the assumptions, that it adheres to the rules of HTML.

Algorithm 4 shows how HTML was generated from the trained models a $<$ character was used as input seed and the models output to sample the next character in the sequence. The concatenated string of characters is then used as the next steps input until the maximum sequence length the model has been trained on was reached. The new input sequence for the model was then basically the upper (newer) half of the old input sequence. This process was repeated until the maximum test case size of 24KB was reached, which resulted in a single test case that was equivalent in size to earlier test cases with 256 HTML-tags. The test cases were then split in half to get 12KB large test cases which were used as the equivalent of test cases with 128 HTML-tags.

6.4 Results

6.4.1 Minimal Preprocessing

The average validation loss and standard deviation were higher than during the prior experiments using default TCN models (see Section 5.1.5). However, those values are not directly

Algorithm 4 Generate a single HTML-tag with a real world default TCN model

procedure REALWORLDTCNHTMLTAGGENERATION($model$)
    $result \leftarrow$ ” $<$ ” ▷ Final result
    $x \leftarrow$ ” $<$ ”
    $n \leftarrow 0$ ▷ Number of sampled characters
    while $n < 24,576$ **do**
        $y \leftarrow model.predict(x)$
        $x \leftarrow x + y$ ▷ String concatenation
        $result \leftarrow result + y$
        $n \leftarrow length(x)$ ▷ Count the characters
        if $n \geq maxLength$ **then**
            $x \leftarrow$ ”$x[-maxLength + 50 :]$” ▷ Move the input window to drop the 50 oldest character
        end if
    end while
    return $result$
end procedure

comparable since different training sets were used, but still provide an indication about the increased training difficulty. The average validation loss values of all trained models were close together with a maximum difference of 0.04 as highlighted by Figure 6.1. The maximum average validation loss was achieved by the configuration one models with 1.196 and the lowest by the configuration four models with 1.145. In terms of the standard deviation the minimum was achieved by the configuration eight models and the maximum by the configuration three ones with 0.145 and 1.71 respectively. Furthermore, Figure 6.1 highlights that a larger unit size in the final dense layers[1] improves the average validation loss and reduces the standard deviation.

```
<a title="Pittericary on manos with tective works of got verypers
    the scada in a notigrium a murament. Bankaku Sonster or
    Kontazioni likomen di eboas. </span></a>
```

Listing 6.3: Example HTML-tag from a configuration one default TCN model

```
<table border="0" value="SearchPortal Dame-JOrg zeg Stockuna Xp7</
    a> [NON PRINTABLE CHARACTER]</td>
```

Listing 6.4: Example HTML-tag from a configuration five default TCN model

```
  <div class="cont_messages2_datinfo">[SEQUENCE OF RUSSIAN
      CHARACTERS]</h3>
  <td align="right" width=390 style="text-align: celler:
      bottom:0pt;">
```

Listing 6.5: Example HTML-tag from a configuration eight default TCN model

[1] models with even numbers utilize more units in the final dense layers

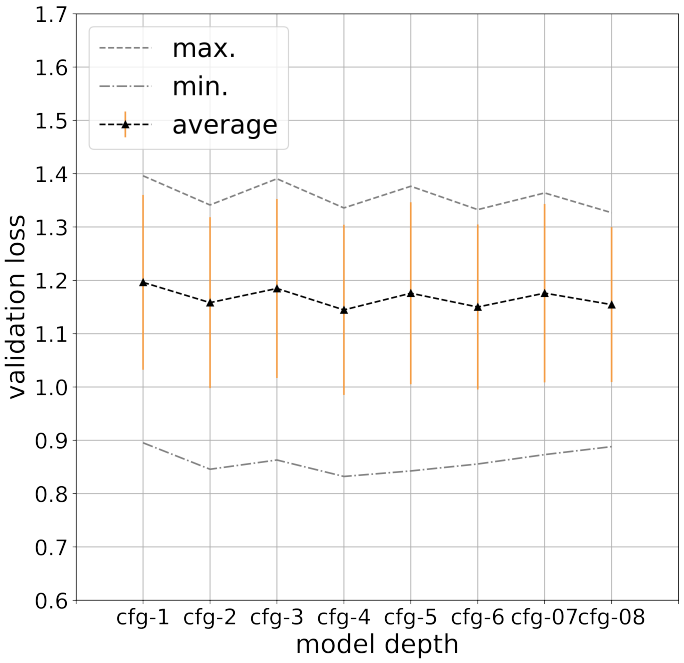

Figure 6.1: Average validation loss of the default TCN models with error-bars indicating the standard deviation across the individual models

The three Listings 6.3, 6.4 and 6.5 show examples of HTML-tags generated by models with different configurations. All examples highlight that those models not only learned and generated HTML but also tried to learn and generate prose in different languages[2]. However, the models were able to replicate the general structure of HTML in regards to using existing HTML-tags and the syntax of attribute declarations, but the HTML-tags were not always closed correctly, nested were not matched and attribute declarations were not always ended by a double quote character. Overall, the HTML-error rate for all models was lower than the fuzzer produced datasets, as shown in Figure 6.2. This was no surprise, because the underlying training data for the models was derived from real world data, which was used as live websites on the internet and the dataset was created by a fuzzer, which introduced errors on purpose to the test cases. In addition, there were no similar examples found to Listing 4.10 where the syntax was correct but the keywords were wrong.

In terms of code coverage performance none of the models was able to reach the absolute number of basic blocks in the dataset coverage area as highlighted in Figure 6.3 and Figure 6.4 for the 12KB and 24KB setting respectively. However, the difference in uniquely discovered basic blocks between the dataset and models is very high compared to all other results with an average of $14,000$ basic blocks, which is approximately half of the total discovered basic blocks per model. This is very surprising, since both the dataset and the models use HTML. In the 12KB HTML per case setting the best performing model in the absolute

[2]Besides the above shown examples the training set included many more languages, for example also Chinese and Korean

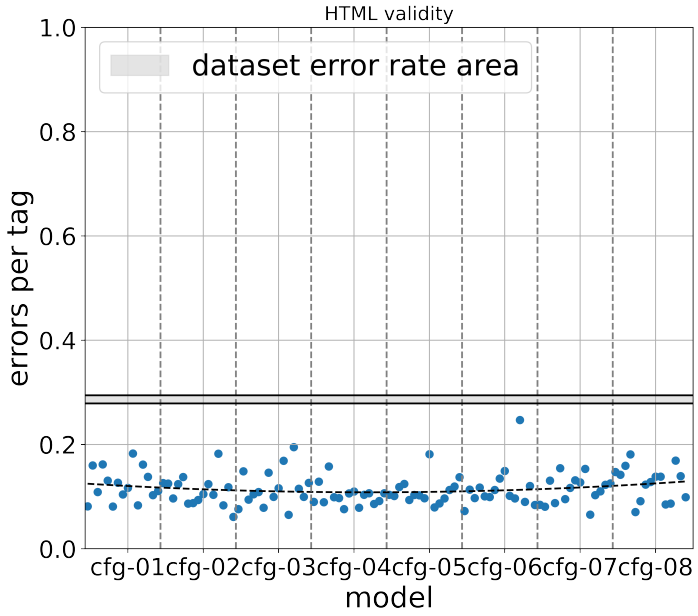

Figure 6.2: Error rate per HTML-tag in regards to TCN models' configuration.

	Baseline	**Mutation Sets**					
		1.6%	3.2%	6.4%	12.8%	25.6%	51.2%
Overlap 12KB:	42.0%	64.5%	62.7%	60.3%	44.5%	36.8%	21.9%
Overlap 24KB:	43.9%	65.4%	65.3%	65.3%	40.5%	43.4%	21.0%

Table 6.1: Best performing TCN model's overlap with the best performing dataset baseline and the differently mutated sets.

number of basic blocks and difference to the best performing dataset was a configuration five model. The model achieved $40,089$ basic blocks in total and a difference of $17,462$ basic blocks. The best performing model configuration in the 24KB HTML setting with regards to absolute performance was a configuration two models. It achieved $39,756$ basic blocks in total and difference to the best performing dataset of $15,969$ basic blocks.

The overlap with the mutation sets were relatively low. This is not surprising after seeing the big difference in basic blocks before, which automatically leads to a smaller overlap. The afore observed trend of a dropping overlap after exceeding a 10% mutation chance can still be observed in the real world data application, as shown in Table 6.1. In both the 12KB and 24KB HTML per case settings the maximum overlap was achieved with the 1.6% mutation chance test set and the minimum unsurprisingly with the 51.2% as it already was in the settings analyzed during Chapters 4 and 5. Furthermore, this highlights the capability of the trained models to trigger a larger number of basic blocks not seen before.

Overall the results have shown that real world HTML websites can be used as training data for a generative model. However, the performance of the models in terms of total basic blocks was lower than during the earlier experiments, here the assumption lies closely that

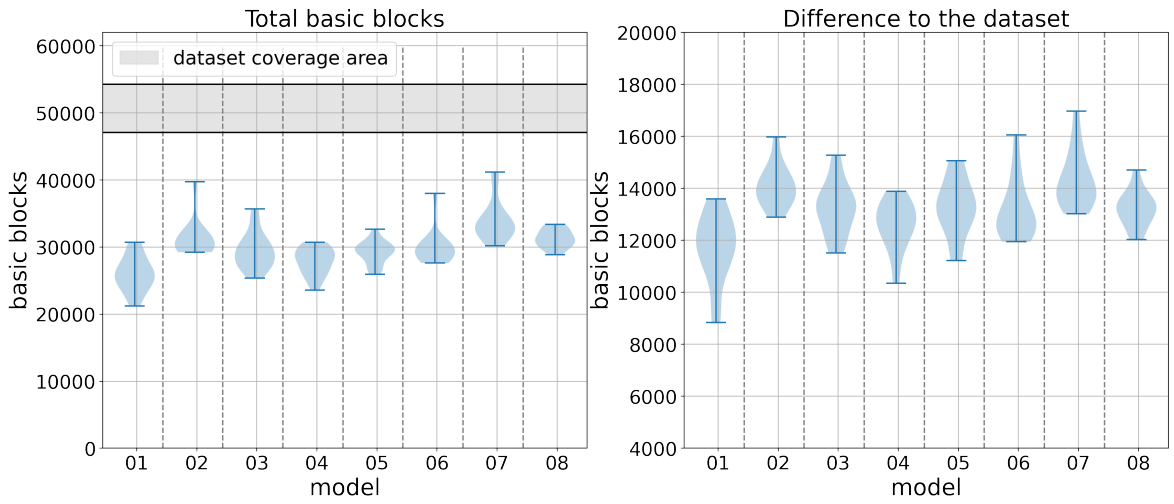

Figure 6.3: Code coverage performance with 12KB of HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

the different languages (HTML, CSS and real world languages) have an negative impact on the overall model's performances, which led to evaluating the performance of a model with a further preprocessed dataset in the following Section.

6.4.2 Additional Preprocessing

All the former results have shown that a stable training leads to a close performance in terms of code coverage, therefore only one additional default TCN model was trained three times on five different splits. Listing 6.6 shows example HTML created by one of those models. It highlights that the model correctly generated opening and closing tags as well as correctly closing the "input" tag. This observation could also be made with multiple models showing that the preprocssing improved the learning of the structure of HTML.

```
1    <td><input type="button" value="Bestationtip" value="Sarperivary"
         size="30" /></td>
```

Listing 6.6: Example HTML-tag from a model trained on the dataset with additional preprocessing.

The results for the 12KB per test case are provided in Table 6.2. It compares the number of discovered basic blocks of the default dataset (see Section 6.4.1) with the dataset using additional preprocessing to remove human languages. The results show both models are close together and on average the additional preprocessing only helped to improve the results marginally. In the 24KB test case size setting the results are comparable the 12KB setting. The difference in average code coverage is slightly higher in favor of the additional preprocessing with a value of $1,673$ basic blocks.

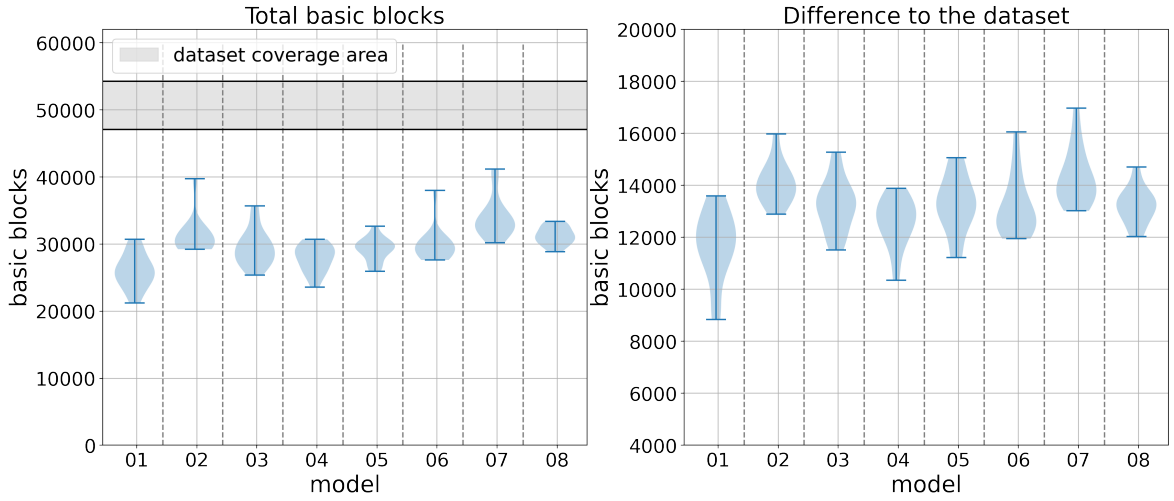

Figure 6.4: Code coverage performance with 24KB of HTML-tags per test case: (left) absolute amount of basic blocks discovered by model; (right) basic blocks not discovered by the test set

Model:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Default:	28,890	28,441	28,531	30,711	30,050	28,991	29,051	32,623
Preproc.:	31,142	31,170	31,064	29,808	25,140	31,542	28,125	33,255
Model:	**9**	**10**	**11**	**12**	**13**	**14**	**15**	$\overline{x}$
Default:	40,089	30,074	30,576	29,808	30,825	28,791	32,383	30,685
Preproc.:	30,736	29,814	30,606	29,578	30,774	38,440	0	30,799

Table 6.2: Direct comparison of the config-05 default TCN models trained on the default dataset and the preprocessed dataset in terms of triggered basic blocks with 12KB per test case. The last column shows the arithmetic mean.

Overall, the additional preprocessing makes a small difference in the performance. So, the question is still why is there such a large gap between the fuzzer generated baseline and the models trained with real world data. Here, the overall distribution of tags might provide an insight for the fuzzer generated baseline the tags are randomly chosen from the set of valid HTML-tags, resulting in evenly distributed HTML tags for the training set described in Section 3.2. In contrast the real world dataset had a distribution of HTML-tags based on the usage habits and necessities of the website developers. This resulted in the trained models being strongly biased to generate those frequently used HTML-tags and basically never generating the rarely used HTML-tags because the probability of sampling the characters needed was just close to 0.

6.4.3 Adjusting the sampling strategy

As consequence of the prior described results it was necessary to tweak the HTML sampling process in order to allow more variety in the generated takes. Therefore, it was necessary

Model:	1	2	3	4	5	6	7	8
Default:	27,011	26,573	25,921	29,597	29,441	27,868	29,221	32,689
Preproc.:	30,531	29,039	29,514	29,037	28,672	29,895	27,845	30,609
Model:	**9**	**10**	**11**	**12**	**13**	**14**	**15**	$\overline{x}$
Default:	30,562	29,726	30,415	28,986	30,741	29,565	29,652	28,198
Preproc.:	30,411	29,105	29,755	28,108	30,900	38,021	27,634	29,871

Table 6.3: Direct comparison of the config-05 default TCN models trained on the default dataset and the preprocessed dataset in terms of triggered basic blocks with 24KB per test case. The last column shows the arithmetic mean.

Model:	1	2	3	4	5	6	7	8
40%:	35,632	33,057	41,338	33,338	34,581	36,659	34,270	32,487
75%:	33,336	34,500	30,583	34,211	31,790	33,438	32,386	32,151
Model:	**9**	**10**	**11**	**12**	**13**	**14**	**15**	$\overline{x}$
40%:	34,209	33,931	32,559	42,010	33,400	34,544	33,258	35,018
75%:	32,651	31,999	33,461	31,919	32,571	31,634	32,086	32,581

Table 6.4: Direct comparison of the configuration five models using random tag insertion with a probability of 40% and 75% in terms of triggered basic blocks with 12KB per test case.

to identify the start of a new HTML-tag. This was achieved by monitoring the sampled characters for the "$<$ /" sequence signaling a closing HTML-tag. After that the appearance of "$<$" in the sequence was used to insert a randomly chosen HTML-tag from the set of valid HTML-tags in order to introduce the desired variety. This was only done in 40% of the cases were a new opening HTML-tag was generated to still generate HTML-tags with model itself. The set of valid HTML-tags was collected with a regular expression matching the sequence starting with a "$<$" character followed by any number of characters from "a-z" and ended by a space character, which nonetheless requires knowledge about the underlying protocol. A second set of test cases was generated with a chance of 75% for inserting a random HTML-tag. The models used for the adjusted sampling strategy were the configuration five models trained on the additionally preprocessed data described in Section 6.4.2.

The results for both created sets is shown in Tables 6.4 and 6.5 for the 12KB and 24KB scenario respectively. The runs with a chance of 40% have lead to an average increase of discovered basic blocks of $4,333$ compared to the default training method and $4,219$ basic blocks compared to the additional preprocessed dataset. The best performing model achieved a total number of $42,010$ basic blocks. It is important to emphasize that training of the models used in the actual setting still took place on the dataset with additional preprocessing. The picture for the 24KB large test cases and 40% chance of randomly choosing an opening HTML-tag was similar. On average this sampling strategy discovered $5,760$ and $4,087$ more basic blocks compared to the default dataset and additional preprocessing models re-

Model:	1	2	3	4	5	6	7	8
40%:	33,544	32,324	40,159	31,800	33,978	35,722	32,995	32,467
75%:	31,995	33,017	28,871	30,012	29,535	33,370	31,311	32,912
Model:	**9**	**10**	**11**	**12**	**13**	**14**	**15**	$\overline{x}$
40%:	32,896	33,019	30,184	41,184	31,759	34,911	32,433	33,958
75%:	31,051	30,291	31,007	30,996	32,533	30,137	31,734	31,251

Table 6.5: Direct comparison of the configuration five models using random tag insertion with a probability of 40% and 75% in terms of triggered basic blocks with 24KB per test case.

spectively. Here, the best performing model was the same as in the 12KB setting with a total number $41,184$ basic blocks triggered. In addition, this also means that the gap to the best performing data set was also narrowed down by that margin while still obtaining the large difference in basic blocks compared to the best performing dataset and the mutated ones.

The higher chance of 75% to insert a random tag was not improving the average absolute code coverage as shown in Tables 6.4 and 6.5. The average performance in the 12KB setting was $2,437$ basic blocks lower than with the 40% chance, but it was still higher than in the scenario without random tag insertion with $1,896$ more basic blocks than the performance on the default training set and $1,782$ more than the preprocessed training set. The 24KB test cases also achieved a lower average performance than the set with 40% randomly inserted tags with a difference of $2,707$ basic blocks. In comparison with the two training sets the performance of the set with 75% randomly inserted tags was $3,053$ and $1,380$ basic blocks higher with regards to the default and preprocessed training set respectively.

6.5 Discussion

All the models that were trained on the real world datasets triggered unique code paths, one explanation for some of those additional basic blocks are Cascading Style Sheet[3] (CSS) styles, which are include as attributes in the training set. Listing 6.5 line 2 shows such an example with correct CSS style attribute. However, collecting code coverage data on a modified fuzzer generated data set with CSS styles reduces the difference in basic blocks by $\approx 4,000$ and can therefore CSS can only be accountable for some of those additional basic blocks. The characters from the different non Latin alphabet based languages might also contribute to the unique basic blocks, since those character were not used in the datasets before.

In Section 6.4.3 the use of 40% randomly selected HTML-tags led to a significant improvement compared to the initial real world dataset trained TCN models. This indicates that the

[3]used to modify the appearance properties of HTML

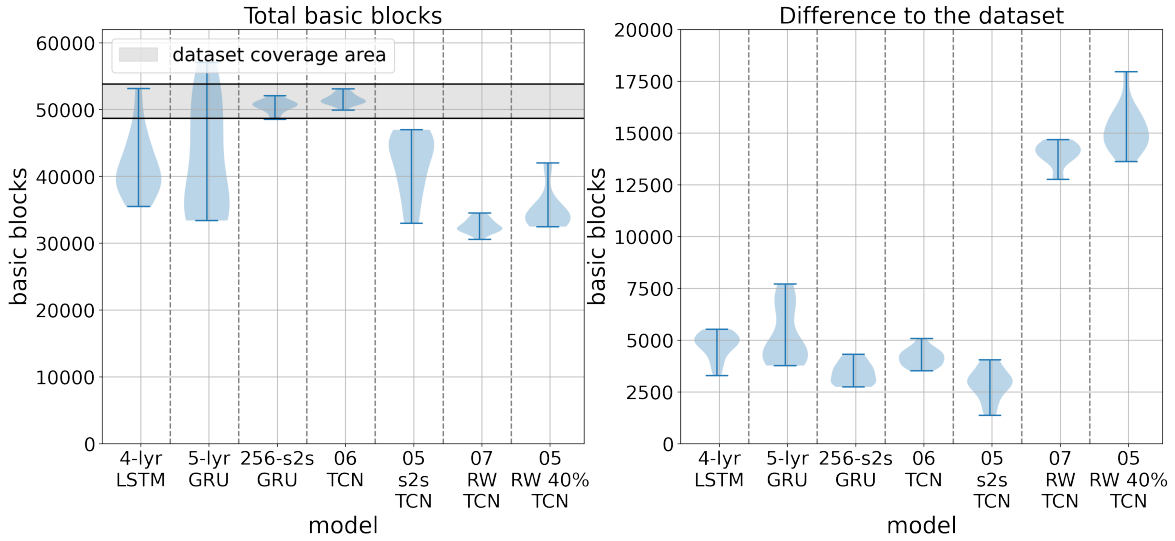

Figure 6.5: Comparison of the best performing models with 128 HTM-tags per file (12KB for real-world (RW) models). (left) total number of basic blocks achieved, (right) difference to the testset

distribution of HTML-tags created is important to the overall performance. It also shows that different HTML-tags use different code paths depending on their function. Some HTML-tags like ”input” need be evaluated further, while others like ”strong” only need to be applied and displayed. Interestingly, the performance decreased with a 70% randomly inserted opening tags which indicates that those additionally inserted HTML-tags might break other structures. For example, with a higher rate of randomly inserted HTML-tags the probability of inserting wrong HTML-tags in places where only certain HTML-tags are allowed increases. This could lead to the lower performance of the test cases with 70% randomly inserted opening HTML-tags.

6.6 Summary

The increased training difficulty induced by different languages (human and artificial) being present in the real-world training set (see Section 6.1) had an impact on the training performance, shown in a higher average loss and standard deviation. However, the average HTML-error was below the baseline dataset introduced in Section 3.4. The absolute numbers of basic blocks were lower, but the difference to the baseline dataset was a lot higher than during the former experiment, with approximately half of the triggered basic blocks not being triggered by the baseline dataset as shown on the left sides of Figure 6.5 and Figure 6.6 for the 12KB and 24KB HTML tags per file case, respectively. The right sides of Figure 6.5 and Figure 6.6 highlight the huge difference to the dataset. Furthermore, the graphs emphasize that the real-world models were able to achieve a higher at least 5000 basic block larger

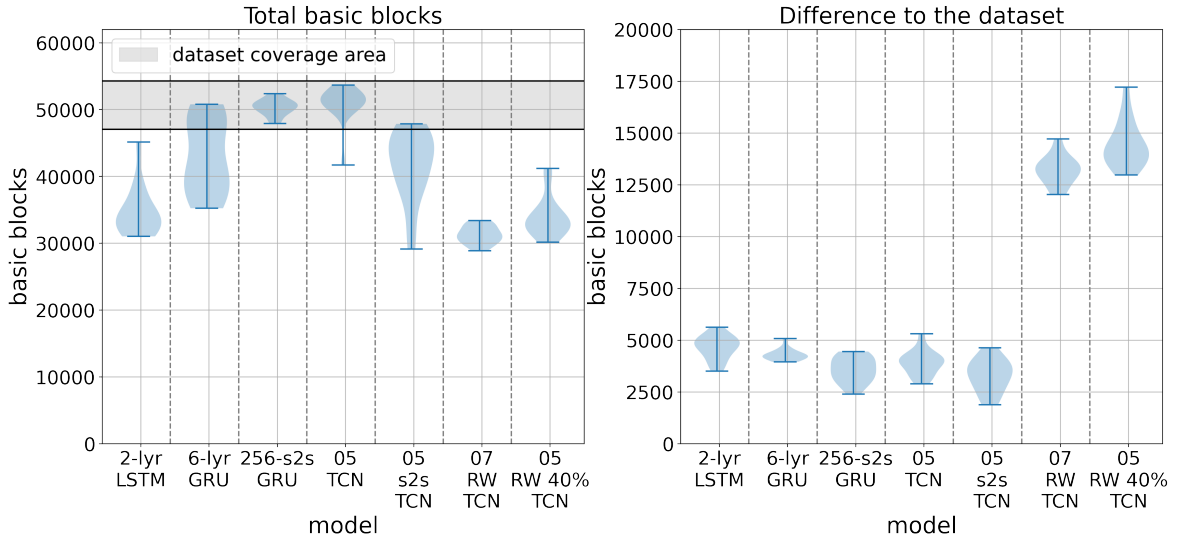

Figure 6.6: Comparison of the best performing models with 256 HTM-tags per file (24KB for real-world (RW) models). (left) total number of basic blocks achieved, (right) difference to the testset

difference to the dataset than the highest difference of the other models.

Applying additional preprocessing to the training set and training a model on it, leads to an average improvement of $1,673$ basic blocks. This demonstrated that decreasing the complexity of the training data by removing the human languages from the training leads to an improvement during the fuzz tests.

In addition, larger improvements were achieved by inserting random opening HTML-tag during the test case generation. This basically guided the model to create a more diverse set of HTML tags because the HTML tags were not evenly distributed in the real-world training set. Therefore the resulting models were biased to generating the higher frequency HTML tags with a larger probability. However, the results highlighted that finding the right probability of inserting HTML tags into the sampling process has an impact on the resulting code coverage and can also lead to worse results. Overall, real-world HTML data can be used to generate HTML test cases for fuzzing by applying additional preprocessing steps (**RQ2** Section 1.2) with postprocessing steps improving the code coverage. The preprocessing includes removing non-HTML-related data, like human languages and scripts.

Finding a well performaing distribution of HTML tags will be the topic of the next chapter, which applies reinforcement learning in order to find the right balance and maximize code coverage. Therefore, it analyzes an approach of utilizing feedback from the program (i.e., code coverage data) in the decision-making process of when to insert an HTML tag or to sample from the model directly. Hereby, the decision of which HTML tag to choose is also made by the model. Overall, this means the next chapter evaluates an approach to tackle the key problem that was not analyzed in the previous chapters, namely how to utilize the

gathered feedback in order to improve the code coverage.

Chapter 7

The Application of Reinforcement Learning on Generating HTML

In this chapter, the focus shifts from the sole generation of test cases to learning from examples to incorporate feedback from the former test cases into the whole process. Therefore this chapter takes the real-world TCN model introduced in Chapter 6 and applies a reinforcement learning algorithm to solve the problem of deciding when to introduce which HTML tag during the sampling process.

First, the overall model architecture is introduced in Section 7.2, where the differences to the formerly introduced approaches are highlighted, and the new extended overall setup is described with an emphasis on the DQN used.

Secondly, the model training is introduced in Section 7.3. It provides details about the structure used for the states and how the reward was determined.

Thirdly, the results gathered during the combined training and testing phase which goes hand in hand due to the chosen overall experiment setup provided in Section 7.5.

Finally, this chapter concludes by summarizing the results and highlighting their importance for the overall context in Section 7.7.

7.1 Extended Environment

The overall experiment setup has changed compared to the experiments conducted in previous chapters, with the goal of improving the code coverage performance by learning from a feedback signal the program under test provides instead of only learning to generate test cases. The updated overall setup includes additional components, which are shown in Figure 7.1, namely the DQN agent, the data module, the replay buffer, and network-connected VMs.

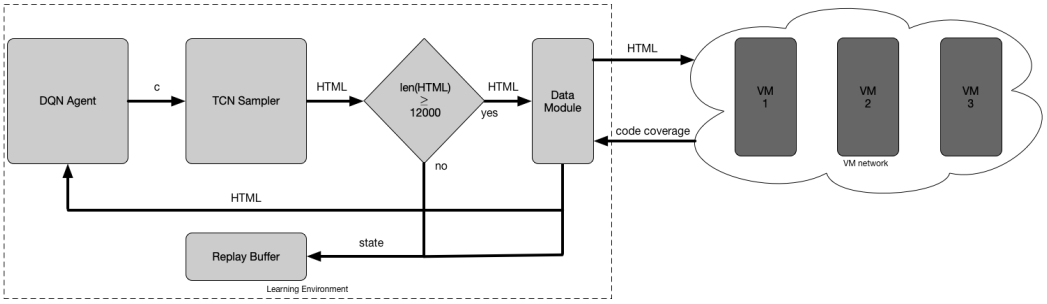

Figure 7.1: The overall setup for the reinforcement learning experiments.

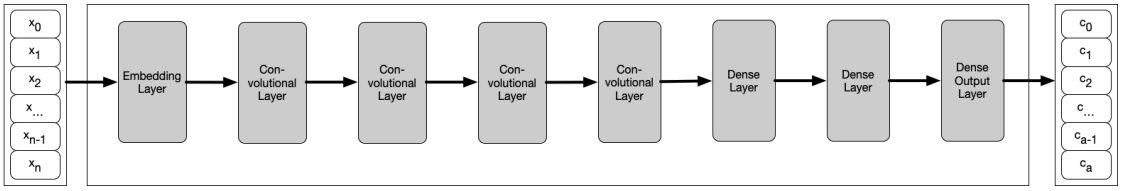

Figure 7.2: The deep neural networks used to approximate the optimal Q-function. Here, x_1 to x_n is the input integer sequence representing the HTML string and c_1 to c_a the Q values of the available actions.

The DDQN agent, which was introduced in Section 7.2, was used to guide the TCN sampler. The actions the DDQN predicted were the next opening HTML tag or allowing the TCN sampler to continue sampling. When the HTML string reached the maximum length of $12,000$ characters, including the template, were sent to the Data Module or used again as input for the DQN agent, and the actual state was stored in the replay memory.

The data module inserted the generated HTML into the HTML template (see Listing 3.4) and distributed it to available VMs over a network connection. There were up to 12 collection VMs involved, and the test cases were distributed round-robin. The connection between the data module and the VMs was established using custom gRPC [83] services. gRPC was used because it is able to handle error correction and retries without additional code. After executing the HTML test cases on the VMs the resulting code coverage was sent back to the data module. Here, the code coverage data was used to define the reward and state for the replay buffer. In addition, a new initial HTML state was issued to the DDQN agent.

The stored observation in the replay buffer was used to train the DQN agent. It had a fixed maximum size, and if the maximum size was reached, the oldest observation was discarded. In addition, the experiences were also sent to database storage to make them available for hyperparameter tuning. A more detailed and experiment-focused description of the approach is provided in Section 7.4.

Layer	Filter dim.	Kernel size	Strides
Conv. 1	32	8	4
Conv. 2	64	4	2
Conv. 3	64	3	1
Conv. 4	64	3	1

Table 7.1: The configurations used for the convolutional layers in Figure 7.2.

7.2 DDQN Architecture

The structure of the DDQN agent is shown in Figure 7.2 and the configurations of the convolutional layers are provided in Table 7.1. First, the input integer sequence with a padded length of $12,000$ was transformed by a pre-trained embedding layer. This embedding layer was restored from the TCN sampler used to generate the HTML test cases later on. The resulting vectors had a dimension of 256 and were used as input for the first convolutional layer. Since the first convolutional layer used strides of 4, the total sequence length was reduced to $2,999$. The second convolutional layer reduced it further to $1,498$. This was followed by two further convolutional layers with the same filter dimension and strides set to one resulting in no further reduction of the sequence length. The convolutional part can be described as

$$H_{conv} = (conv_4 \circ conv_3 \circ conv_2 \circ conv_1)(X_{emb}) \tag{7.1}$$

with X_{emb} being the result of the embedding layer. The extracted feature maps H_{conv} were flattened to be suitable as input for the two dense layers with 512 internal units. More importantly, the flattening allowed the final result to be a scalar value for each action in one output vector instead of a sequence of vectors. The final output was computed by a dense output layer with the internal size set to the number of available actions. So, the result was

$$\mathbf{y} = (dense_{out} \circ dense_2 \circ dense_1)(H_{conv}) \tag{7.2}$$

with $y \in \mathbb{R}^a$ and $a \in \mathbb{N}$ the number of available actions. All layers applied the $relu$ function (see Equation (2.13)) as activation, except for the last layer, which did not apply any activation. Overall, this architecture closely follows the one provided by Mnih et al. [57] and introduced in Section 2.2.6. However, it was not possible to exactly reconstruct their approach because of the different state representations. Here, the state consisted of a long sequence of integers transformed into a vector representation by the embedding layer, which only allowed to apply a one-dimensional convolution. This is in contrast to the multiple picture frame they used for the state representation.

7.3 DDQN Training

The DDQN agent consisted of two networks with the same architecture (see Section 2.2.6), called target and estimator networks. Initially, the target and estimator networks had the same values and deviated during the training from each other. For every $10,000$ steps, the parameter values of the estimator network were copied into the target network.

The DQN agent was trained by using 64 randomly selected states from the replay buffer. Those states consisted of the HTML used as input to the DQN agent, the action with the highest predicted Q-value, the resulting HTML state and a reward. The reward was set to zero when the HTML was not executed on a VM. Otherwise, it was set to the cardinality of the basic block set divided by the average cardinality of all basic block sets collected in Chapters 4 to 6. This mapped the interval $[0,\ 37956]$ of the absolute achieved values to the interval $[0.0,\ \approx 2.8145]$. In general, a smaller reward interval leads to a more stable gradient during training because large rewards lead to huge gradient steps during training.

The main objective was to minimize the Huber loss, defined as

$$\mathcal{L}(\sigma) = \begin{cases} \frac{1}{2} * \sigma^2 & \text{for } |\sigma| \leq 1.0 \\ \frac{1}{2} + (|\sigma| - 1.0) & \text{otherwise} \end{cases} \tag{7.3}$$

with σ being Equation (2.21). The ADAM optimization (see Section 2.2.7) algorithm with a learning rate of 0.0001 was used to minimize the loss. However, in order to provide a stable and not fluctuating optimization process, it was necessary to wait for new observations in the replay buffer before another training step took place. The gamma value was set 0.98888, which resulted in approximately 90 future time steps[1] taken into account for the Q-value computation. Furthermore, ϵ[2] was set to 1.0 in the beginning and reduced every 250 time steps by a factor of 0.99 until ϵ was equal to 0.1.

For every 250 DQN agent steps, the setup was used to compute a validation test case. In order to compute this test case, ϵ was set to 0.001, and an initial HTML state was used as the DQN agent's input. The then sampled test case was sent to a VM to be executed. If the test case achieved a higher reward than the validation test cases before a new model checkpoint was created. The last five model checkpoints were kept.

[1] $\frac{1}{1-0.98888} \approx 89.9281$

[2] the probability to choose a random action (see Section 2.2.6)

config	k_1	s_1	f_1	k_2	s_2	f_2	k_3	s_3	f_3	k_4	s_4	f_4	d_1	d_2	d_3	d_4
01	8	2	8	4	2	16	3	1	32	3	1	64	128	128	128	64
02	8	2	16	4	2	32	3	1	64	3	1	64	128	128	128	64
03	8	2	16	4	2	32	3	1	64	3	1	64	128	128	128	128
04	8	2	32	4	2	64	3	1	64	3	1	64	256	256	–	–

Table 7.2: Evaluated hyper-parameter configurations for the DDQN agent. The parameters k_i, s_i, f_i provide the kernel size, stride and filter dimensions of the CNN layer i respectively. d_i provides the number of internal units of the dense layer i.

7.4 Experiments

For the experiments, the TCN sampling process had to be changed in order to accept the action from the DDQN agent. The agent's actions were defined to either insert a certain opening HTML tag linked to the numerical value of the action or continue sampling. Therefore, it was necessary to observe the generated HTML string while it was sampled in order to identify a state where a new opening HTML tag was needed. The sampler routine checked the generated HTML string for the start of a closing HTML-tag sequence (i.e., "</"). This triggered a flag, and the sampling process was then interrupted when the next "<" character was encountered. The so-far generated HTML string was used as input to the DQN agent, and the process started all over again until a length of $11,000$ characters was reached.

The data module consisted of a communication and dataset-handling module. First, the communication module was responsible for distributing the test cases to the VMs and also receiving the replies from the VMs containing the code coverage data. It was then dispatched to the dataset handling module. Secondly, the dataset handling module kept track of the issued test cases, saved them together with results, and computed the reward for the DQN agent. The reward and the initial test cases as integer sequences were then put into the replay buffer to be available for training.

The available VMs were executing software handling the network connections from and to the data module. After receiving a test case with a new job id (defined by the data module), the test case was saved to disk and executed in Firefox. The resulting code coverage data was evaluated, and the set difference with the zero line (see Section 3.4) was computed. If the number of discovered basic blocks was less than $2,500$ the used profile was marked as corrupted, and a backup was restored. After that, the same test case was executed again. This helped to avoid outliers; therefore, incorrect code coverage data did not influence the training process. The maximum throughput per VM was eight parallel test cases.

The system was used to gather data by choosing random actions during the test case generation. This was repeated until the database held $2,872,990$ in state transitions or 176.1 GB of data. The state transitions were then used to find well-performing hyperparameters.

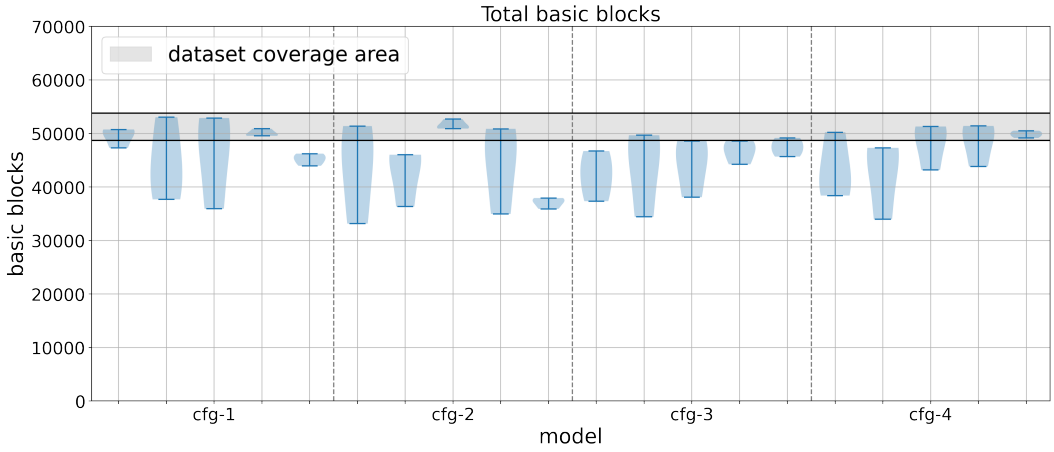

Figure 7.3: Code coverage performance results of the models trained during hyper-parameter search. Per configuration 5 different learning rates were tested. The violin plots per configuration are ordered by decreasing learning rate.

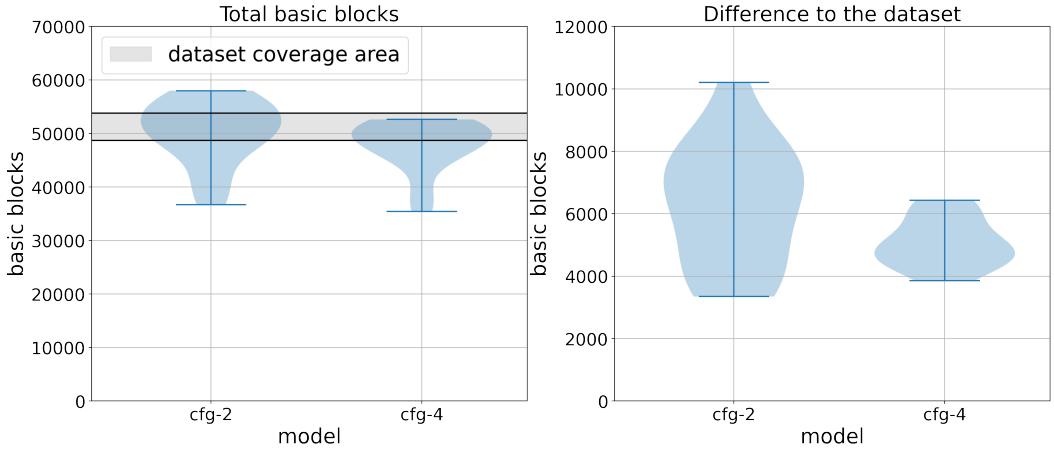

Figure 7.4: (left) DDQN agents' performance compared to the fuzzer baseline; (right) DDQN agents uniquely discovered basic blocks compared to the fuzzer baseline

The evaluated hyperparameters can be seen in Table 7.2. The hyperparameters were tested by training the model on the transitions received from the database and then generating test cases to execute every 100 training steps. During the evaluation step, four test cases were executed, and the average code coverage performance was used as the indicator of the model's performance. The weights of the best-performing models were saved. Per configuration, five different learning rates ranged from 0.0645 to $0.0645 * 10^{-4}$. The two best-performing model configurations were then used to train 15 models. The resulting models then generated 128 test cases each to evaluate the overall performance as in prior experiments.

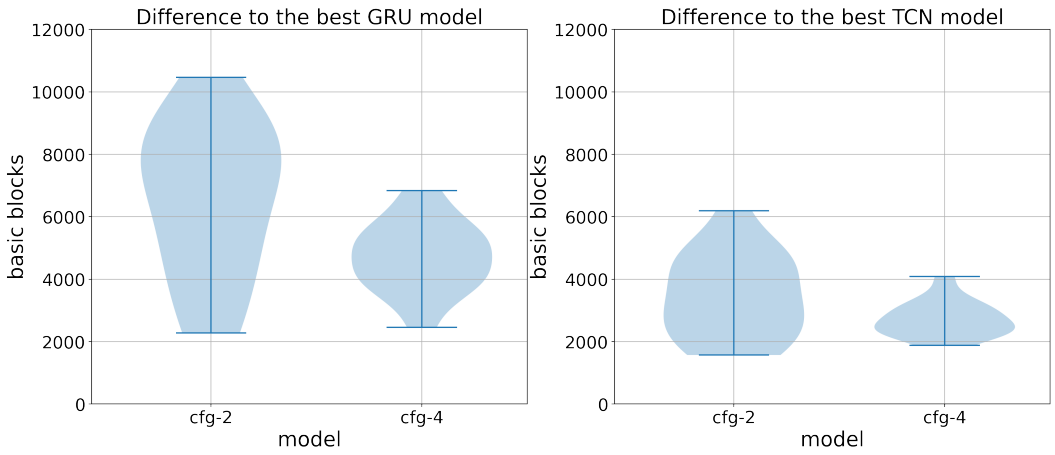

Figure 7.5: (left) DDQN agents' uniquely discovered basic blocks compared to the best performing GRU model; (right) DDQN agents uniquely discovered basic blocks compared to the best performing TCN model

7.5 Results

The training phase of the DDQN agent already showed that it is important to take frequent snapshots of the model's state and test the performance regularly to ensure a good-performing configuration is recognized. The average reward for the test cases fluctuated strongly between training steps. Furthermore, this also provides a reason why the lower learning rates perform better on average than higher learning rates because the higher learning rates diverge quicker from well-performing states.

First, a hyper-parameter search was conducted in order to find parameters that provide stable and promising results. The hyper-parameter search provided two promising candidates for further testing, namely configuration two and four of Table 7.2. The overall results of the hyper-parameter search can be seen in Figure 7.3. Configuration two and four models have the highest average code coverage performance, and both also performed best with the learning rate set to 0.000645.

Figure 7.4 (left) highlights the overall results of the DDQN agents compared to the baseline results. In total, three configuration two ($\mathcal{C}2$) models outperformed the best baseline set in terms of basic blocks. The best performing DDQN agent achieved $57,993$ basic blocks in total, a 7.7% performance increase over the best baseline set. Furthermore, the figure also shows that the majority of $\mathcal{C}2$ DDQN agents either performed close to or above the maximum data set performance. Only three of the fifteen $\mathcal{C}2$ DDQN models performed below the minimum data set code coverage. The configuration four ($\mathcal{C}4$) models could not outperform the best baseline and achieved a maximum of $52,614$ basic blocks, 2.2% below the maximum baseline performance. Six out of the fifteen $\mathcal{C}4$ models could not perform above the minimum data set performance.

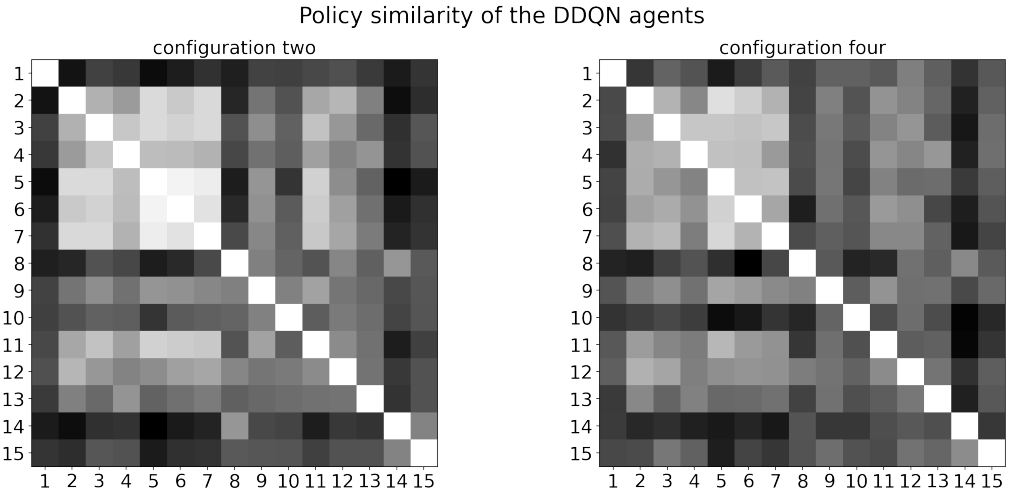

Figure 7.6: Policy similarity of the 15 trained DDQN agents per configuration. A lighter shade indicates a smaller Kullback Leibler Divergence [84] between the policies.

In comparison to the baseline, the $\mathcal{C}2$ DDQN agents were able to discover between $3,349$ and $10,206$ unique basic blocks as highlighted by Figure 7.4 (right). Furthermore, the $\mathcal{C}4$ DDQN agents discovered between $3,859$ and $6,435$ unique basic blocks. In both configurations, the DDQN agents were able to improve the overall code coverage by at least 6% and, in the best case, by 18.9%.

The number of unique basic blocks compared to the underlying TCN generator unsurprisingly is lower than compared to the baseline as seen in Figure 7.5 (right). The $\mathcal{C}2$ DDQN agents achieved $1,573$ to $6,197$ unique basic blocks and the $\mathcal{C}4$ DDQN agents ranged between $1,886$ and $4,092$ This results in an improvement rate of 2.9% and 11.5% for the worst and best case respectively.

The generator model achieving the overall highest performance in total basic blocks was a five-layer stacked GRU introduced in Chapter 4. It achieved $57,059$ basic blocks, which is 1.6% below the best-performing DDQN model. Figure 7.5 (left) highlights the differences in basic blocks between the DDQN models and the best-performing five-layer stacked GRU model. The maximum difference achieved was $10,460$ basic blocks and the minimum $2,284$ with an average of $6,533$ basic blocks for the $\mathcal{C}2$ models. The difference was smaller for the $\mathcal{C}4$ models with a maximum of $6,836$, a minimum of $2,454$, and an average of $4,724$ basic blocks. This results in the best-case improvement of 18.3% and 4.0% in the worst case.

7.6 Discussion

The results have shown that it is possible to improve the results of an existing fuzzer by applying a generative deep learning model and a reinforcement learning model to the fuzzer.

The distribution of chosen actions by the DDQN agents gives interesting insights into the performance of the model and is highlighted in Figure 7.6. For instance, for the $\mathcal{C}2$ agents, it seems like the low-performing models got stuck with a policy that predicts simple render HTML tags over and over again, whereas the well-performing models all have the ”input” tag in their top ten actions taken. In contrast, the $\mathcal{C}4$ agents achieve similar performance with a more balanced tag distribution. Nonetheless, the low-performing models also show an unbalanced policy with a tendency to 'render only' tags, like 'a'. Furthermore, computing the symmetrized 'Kullback Leibler Divergence'[84] between the pairwise distributions:

$$D = D_{KL}(P||Q) + D_{KL}(Q||P) \, . \tag{7.4}$$

It shows that the well-performing action policies have a smaller distance from each other than the low-performing ones. For example, the distance between the training runs six and seven of the $\mathcal{C}2$ models have a distance of ≈ 2.3211, whereas the distance between training runs six and eight is ≈ 9.9995. The training runs six, seven, and eight had a total code coverage performance of $55,594$, $54,874$, and $36,678$ basic blocks, respectively. The difference in policies is also highlighted by Figure 7.6 where the policies of runs two to seven build a similarity area and policy eight is clearly separated from that area.

The rewards returned by the VMs indicate a high instability in the training process. They potentially vary significantly in a few training steps. This effect explains why a lower learning rate worked better during all training runs, especially with growing model complexity. The initial data collection for the DDQN agent also indicated that it is beneficial to reuse the TCN embedding weights in the DDQN agent and disable training of the embedding.

7.7 Summary

This chapter provided a detailed analysis of the use of a DDQN agent to improve the code coverage of an existing TCN generator model. The new environment (see Section 7.1) included a setup for online and offline training by utilizing a replay memory for online training and a storage device for offline training. Furthermore, test cases were distributed automatically to a network of code coverage collection VMs to receive the reward signal. The DDQN architecture (Section 7.2) utilizes the TCN generator's output as the state of the world and is trained to provide an HTML tag that maximizes future code coverage.

The experiments (Section 7.4) started with a data collection phase where the DDQN agents were trained online with a set of changing hyper-parameters. The state data and rewards were stored in long-term storage to be used during offline training. The following hyper-parameter search made use of the stored data to identify to promising configurations that had

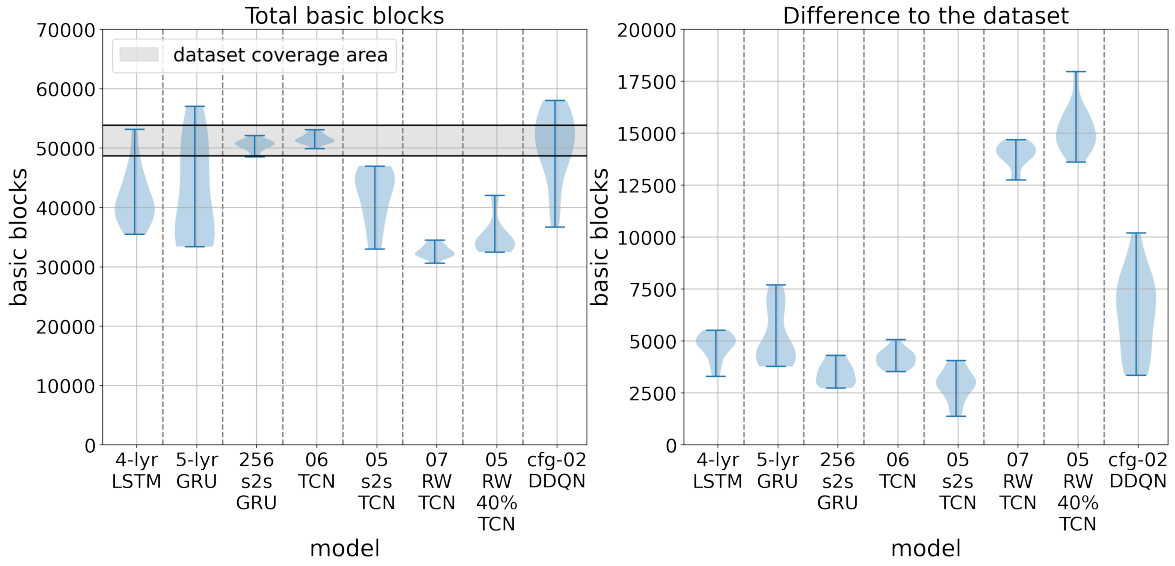

Figure 7.7: Comparison of the best-performing models with 128 HTM-tags per file (12KB for real-world (RW) and DDQN models). (left) total number of basic blocks achieved, (right) difference to the testset

a high and stable performance. The resulting two configurations were then trained 15 times each, as in prior experiments.

The results (Section 7.5) have shown that the DDQN agent is able to outperform the underlying TCN generator model as well as the fuzzer created baseline with improvements of up to 18.9% and 7.7%, respectively. Furthermore, the DDQN agent was able to outperform the best-performing five-layer GRU model by 1.6% in total basic blocks. The DDQN agent also showed that it was able to discover up to $10,206$ basic blocks that were not discovered by the baseline. Figure 7.7 emphasizes that the DDQN-agent was able to improve the TCN-based approach to outperform all other proposed and evaluated models from prior chapters.

Overall, the experiments have shown that the introduced architecture provides a method to utilize the code coverage results as a signal to guide test case creation (RQ3 Section 1.2). Furthermore, it highlights that the output of the TCN generator is a valid data representation to use as the DDQN agent's input. The results also indicate that the embedding weights of the TCN generator also provide a good embedding for the DDQN agent.

Chapter 8

Conclusions and Future Work

This chapter starts in Section 8.1 by taking a look back at the thesis statement and sets it into the context of the results acquired in this thesis. In light of the thesis statement the key results are summarized in Section 8.2. This is followed by a list of key contribution in Section 8.3. Finally, Section 8.4 provides on overview of possible future work to cover shortcomings of this thesis.

8.1 Thesis Statement revisited

The goal of this thesis was to show that machine learning algorithms can reduce the time needed to develop test case generators and also guide a test case generator in order to improve the fuzz testing capabilities. The results demonstrate that it is possible to train generative deep learning algorithms to be test case generators for HTML and improve the code coverage performance compared to fuzzer generated base line. It also showed that those algorithms can improve the results of already existing fuzzer. However, it also demonstrated the limitation of the approach without utilizing additional knowledge to modify the dataset and make it better suited for training. Furthermore, the results have shown that the performance data collected can be used to guide test case generators to achieve higher code coverage. This approach is also limited by the time and effort put into studying the input structure and designing the feedback loop.

8.2 Summary of the Key Results

The first experiment conducted in Section 4.1 analyzed the performance of two different RNN cells in stacked architecture, namely GRU and LSTM cells. Both cell types were able to generate novel HTML-tags that were not part of the training set. The GRU based models

were able to achieve a lower HTML-tag error rate and a higher code coverage in terms of basic blocks. This was followed an analysis of a seq2seq (see Section 2.2.4) based architecture with GRU cells in Section 4.2. The results have shown that the seq2seq architecture is able to outperform the stacked RNN architecture. In general both experiments have shown that the trained models can act as test case generators and discover code areas that were not included in the baseline created by manually developed test case generator.

In order to avoid the downsides of using RNN based cells, namely the memory requirements and the strict dependency of an output at time t on the former output at $t-1$ a non recurrent cell type was introduced and used for the experiments in Chapter 5. The TCN cells were also used in a stacked and seq2seq architecture. The stacked TCN models showed a surprisingly reliable training performance, low HTML-error rate and code coverage performance. It outperformed all the other analyzed approaches and perform well in the area of the underlying test case generator in terms of total coverage, while also discovering new basic blocks. However, the results for the TCN seq2seq architecture have shown that the proposed architecture does not perform well. This kind of mixture could be trained but it was not able to reach the low HTML-error rate of the other models nor the code coverage performance.

In general, both of the tested base architecture namely recurrent and temporal convolutional neural networks were able to generate new unique test cases after being trained on a fuzzer generated dataset. The TCN based models were more reliable during training and in code coverage performance and therefore more suitable for the task in general. The results in Chapters 4 and 5 have shown that it is possible to make a performance prediction with the validation loss and the HTML-error rate at hand, for a generalized statement this means the use of second source of validation that is dependent on the actual input structure.

Chapter 6 introduced a training set based on real world data. The results demonstrate that the increased training set complexity made training more difficult. The first real world dataset included a mixture of different languages including different human languages, HTML as well as CSS. The only alteration done to it was the removal of JavaScript. The first experiment in Section 6.4.1 resulted in a lower total number of basic blocks but achieved a large number of new unique basic block not seen in the test case generator set and the former experiments. In order to decrease the complexity of training set and increase the overall code coverage additional preprocessing in form of removing the human languages from the dataset was applied. The yielded a slightly higher model performance (see Section 6.4.2) and led to the assumption that the distribution of HTML-tag in the training set might have a negative influence on the code coverage performance. The experiment results in Section 6.4.3 confirmed that inserting random opening HTML-tag during the sampling phase from the model leads to an on average better code coverage performance. It basically forced the model to divert from the real world dataset HTML-tag distribution which was very repetitive and therefore the models only rarely generated HTML-tags, which were not used often on real websites.

Overall, the results highlight that a real world dataset can be used to train TCN based test case generator. However, it also became clear that the performance is not as high as with a manually written test case generator, but the models were able to discover large areas that were not executed by the baseline test case generator. This chapter also highlighted the need to guide the distribution of the HTML-tags which requires the acquisition of knowledge about the underlying input structure to able to identify the problem in the first place.

Results introduced in Chapter 7 show that the feedback of an executed test case can be used to guide a reinforcement learning based fuzzer to improve its performance. The feedback can be used in the form of a score dependent on the achieved code coverage performance which informs the reinforcement learning algorithm how to steer the test case creation process.

8.3 Summary of Contributions

Chapter 3 provides an experimental environment for code coverage collection and training of the deep learning algorithms.

Chapter 4 demonstrates how different RNN architectures can be used as HTML test case generators and delivers a comparison of the performances in terms of training stability, HTML error rate and code coverage.

Chapter 5 provides TCN based architectures for generating HTML and showcases their performance in comparison to the RNN based models. It identifies them as a suitable test case generator.

Chapter 6 showcases methods to use real world data to train a TCN based test case generator. The chapter highlights the need to modify the training set in order to improve the overall code coverage performance.

Chapter 7 introduces a novel approach to utilize the code coverage data to further improve the performance by applying a reinforcement learning algorithm. The chapter demonstrates how reinforcement learning improves code coverage of a trained algorithm by up 18.9%.

8.4 Future Work and Final Remarks

In order to further improve the methods introduced in this thesis, one direction could be to analyze the performance of different inputs. This would provide insight into the transferabil-

ity of the results and indicate the feasibility of the methods for different inputs. This could be done by starting with similar file formats like XML and then moving to more and more different file formats, for example, picture formats instead of structured document description formats. Furthermore, leveraging modern techniques for explainability in deep learning (see Ras et al. [**?**] for a survey) could provide a better understanding of the model's behavior during test case generation. Therefore, it would provide valuable insights for improving the model's performance.

One of the shortcomings of this thesis is that the reinforcement learning techniques introduced in Chapter 7 do not provide a way to introduce errors into the test cases. This limits the ability to discover flaws in software upfront. So, it is necessary to research methods to learn when to introduce errors into the test case. This could lead to the discovery of flaws lying deeply in the execution path. Another improvement for the reinforcement learning part could be to research ways to directly learn the generator and not only guide it by the reinforcement learning part. It could also provide deeper execution paths and therefore hidden flaws.

One main time-consuming factor during all experiments was the code coverage collection. It might be worth optimizing that part by introducing a prediction network that provides the estimated code coverage given a test case input and only use the test case if the prediction is not inside defined parameters. This would speed up the process, especially in long-running experiments, since the collected data could be used to further improve the prediction.

The overall performance could be further improved by combining the reinforcement learning-based fuzzer with other fuzzing techniques like AFL [18]. This will entail time-consuming changes to the code coverage collection to line up with AFL's branch coverage algorithm. The combination could include providing a seed set for mutation-based fuzzing and also learning in an online way to mutate the seeds. The advantage of mutating the seeds is that a test case mutation is only a single forward pass through the network and not repeatedly passing through the model to create a complete test case.

Appendix A

Appendix

A.1 HTML-tag Distribution

Table A.1 shows how many times each HTML-tag occurred in the fuzzer created and real world data set. It was created by first converting the datasets to lower case characters only in order to avoid ignoring differently written HTML-tags for example ”figure”, ”fIgUrE” and ”FIGURE” are all valid HTML-tags. After the conversion the valid HTML-tags occurring in the dataset was counted. This was done by using the HTML-tags name (Table A.1 Column 1) prepended with ”<” as search string.

HTML-tag	Fuzzer created dataset	Real world dataset
a	5595	1459703
abbr	5605	1690
address	5429	307
area	0	2736
article	5420	8311
aside	5509	3218
audio	5488	53
b	5566	4112
base	0	31
bdo	5643	198
blockquote	5584	630
body	5490	250
br	5543	179970
button	5452	20280
canvas	5515	53
caption	0	58

cite	5540	261
code	5492	444
col	0	826
colgroup	0	36
datalist	5520	8
dd	0	26682
del	5573	426
details	5607	114
dfn	5529	109
dialog	5645	2
div	5516	1181136
dl	5666	13015
dt	0	10649
em	5513	2347
embed	5378	238
fieldset	0	1760
figcaption	0	385
figure	5444	1996
footer	5614	4315
form	5501	20800
h1	5328	8126
h2	5502	15950
h3	5552	21038
h4	5611	6288
h5	5590	2192
h6	5479	589
head	5571	362
header	0	5521
hr	5498	12317
html	5538	524
i	5487	54717
iframe	5595	3243
img	5502	217697
input	5437	97496
ins	5454	961
kbd	5540	135

keygen	0	0
label	0	24415
legend	0	217
li	0	416693
main	5520	1316
map	5674	359
mark	5616	39
meta	0	6320
meter	5507	15
nav	5534	8300
object	5548	467
ol	5727	1819
optgroup	0	877
option	0	200777
output	5631	18
p	5477	90936
pre	5620	1080
progress	5637	24
q	5557	184
rp	0	24
rt	0	19
ruby	5582	21
s	5565	152
samp	5487	71
section	5396	10350
select	5583	7501
small	5398	2424
source	0	1134
span	5500	608473
strong	5461	3814
style	5498	1875
sub	5574	101
summary	0	27
sup	5513	1914
table	5479	39138
tbody	0	490

td	0	230860
textarea	5628	1789
tfoot	0	105
th	0	12334
thead	0	180
time	5577	4116
title	0	85
tr	0	34448
track	5357	16
u	5351	510
ul	5466	136635
var	5460	123
video	5426	126
wbr	5590	53
Total	409000	5247069

Table A.1: Absolute distribution of HTML-tags in the fuzzer created dataset and the real world dataset.

A.2 Hello World Example

The example C ”Hello World” program was compiled with GCC invoking the following cmd:

```
gcc hello.c
```

Furthermore, the number of assembly instructions was obtained with the ”objdump” command invoked as:

```
objdump -x86-asm-syntax=intel -d a.out
```

The output was

```
a.out:  file format Mach-O 64-bit x86-64

Disassembly of section __TEXT,__text:
__text:
100000f60:  55  push  rbp
100000f61:  48 89 e5  mov rbp, rsp
100000f64:  48 83 ec 10   sub rsp, 16
100000f68:  48 8d 3d 33 00 00 00  lea rdi, [rip + 51]
100000f6f:  b0 00   mov al, 0
100000f71:  e8 0a 00 00 00  call  10 <dyld_stub_binder+0x100000f80>
100000f76:  89 45 fc  mov dword ptr [rbp - 4], eax
100000f79:  48 83 c4 10   add rsp, 16
100000f7d:  5d  pop rbp
100000f7e:  c3  ret

_main:
100000f60:  55  push  rbp
100000f61:  48 89 e5  mov rbp, rsp
100000f64:  48 83 ec 10   sub rsp, 16
100000f68:  48 8d 3d 33 00 00 00  lea rdi, [rip + 51]
100000f6f:  b0 00   mov al, 0
100000f71:  e8 0a 00 00 00  call  10 <dyld_stub_binder+0x100000f80>
100000f76:  89 45 fc  mov dword ptr [rbp - 4], eax
100000f79:  48 83 c4 10   add rsp, 16
100000f7d:  5d  pop rbp
100000f7e:  c3  ret
Disassembly of section __TEXT,__stubs:
__stubs:
100000f80:  ff 25 8a 00 00 00   jmp qword ptr [rip + 138]
Disassembly of section __TEXT,__stub_helper:
__stub_helper:
100000f88:  4c 8d 1d 79 00 00 00  lea r11, [rip + 121]
100000f8f:  41 53   push  r11
100000f91:  ff 25 69 00 00 00   jmp qword ptr [rip + 105]
```

```
100000f97:  90  nop
100000f98:  68 00 00 00 00  push  0
100000f9d:  e9 e6 ff ff ff  jmp -26 <__stub_helper>
```

and counting the assembly instructions shown resulted in 25.

Bibliography

[1] W3 Consortium, "Html standard," ser. https://html.spec.whatwg.org/print.pdf, August 2018.

[2] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[4] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[5] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[6] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[8] Microsoft, "The history of !exploitable crash analyzer," ser. https://blogs.technet.microsoft.com/srd/2009/04/08/the-history-of-the-exploitable-crash-analyzer, April 2009.

[9] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," 2017.

[10] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE international symposium on secure software engineering*, vol. 1. IEEE, 2006, pp. 13–15.

[11] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[12] R. Shapiro, S. Bratus, E. Rogers, and S. Smith, "Identifying vulnerabilities in scada systems via fuzz-testing," in *International Conference on Critical Infrastructure Protection*. Springer, 2011, pp. 57–72.

[13] D. Coppit and J. Lian, "Yagg: an easy-to-use generator for structured test inputs," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 356–359.

[14] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 720–725.

[15] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 95–110.

[16] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[17] Peach-Tech, "Peach fuzzer," ser. http://www.peachfuzzer.com, 2017.

[18] M. Zalewski, "American fuzzy lop," ser. http://lcamtuf.coredump.cx/afl/, 2017.

[19] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Probabilistic grammar fuzzing," in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023, retrieved 2023-01-07 15:01:16+01:00. [Online]. Available: https://www.fuzzingbook.org/html/ProbabilisticGrammarFuzzer.html

[20] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 44, no. 1.2, pp. 206–226, 1959.

[21] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[22] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[23] M. Minsky and S. Papert, "Perceptrons." 1969.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[25] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1017–1024.

[26] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," *arXiv preprint arXiv:1312.6026*, 2013.

[27] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, 1991.

[28] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions On Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[30] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[31] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *International Conference on Machine Learning*, 2015, pp. 2067–2075.

[32] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[33] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, pp. 947–951, 2000.

[34] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[35] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1701–1708.

[36] D. Han, J. Chen, and J. Sun, "A parallel spatiotemporal deep learning network for highway traffic flow forecasting," *International Journal of Distributed Sensor Networks*, vol. 15, no. 2, p. 1550147719832792, 2019.

[37] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[38] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," pp. 1–9, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[39] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

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

[41] J. Gauthier, "Conditional generative adversarial nets for convolutional face generation," *Class Project for Stanford CS231N: Convolutional Neural Networks for Visual Recognition, Winter semester*, vol. 2014, no. 5, p. 2, 2014.

[42] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017.

[43] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in Neural Information Processing Systems*, 2017, pp. 5767–5777.

[44] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint arXiv:1803.01271*, 2018.

[45] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[46] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[48] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[49] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[50] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *arXiv preprint arXiv:1410.3916*, 2014.

[51] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "End-To-End Memory Networks," pp. 1–11, 2015. [Online]. Available: http://arxiv.org/abs/1503.08895

[52] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[53] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[54] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[55] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[56] L.-J. Lin, "Reinforcement learning for robots using neural networks," Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.

[57] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[58] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[59] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems*, 2010, pp. 2613–2621.

[60] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[61] G. Hinton, "Neural networks for machine learning: Lecture 6a overview of mini-batch gradient descent," ser. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, June 2022.

[62] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[63] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," *Automated Software Engineering (ASE 2017)*, 2017.

[64] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2613–2630.

[65] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *arXiv preprint arXiv:1801.04589*, 2018.

[66] R. Fan and Y. Chang, "Machine learning for black-box fuzzing of network protocols," in *International Conference on Information and Communications Security*. Springer, 2017, pp. 621–632.

[67] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 95–105.

[68] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 1044–1051, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/3895

[69] H. Xu, S. Fan, Y. Wang, Z. Huang, H. Xu, and P. Xie, "Tree2tree structural language modeling for compiler fuzzing," in *Algorithms and Architectures for Parallel Processing: 20th International Conference, ICA3PP 2020, New York City, NY, USA, October 2–4, 2020, Proceedings, Part I 20*. Springer, 2020, pp. 563–578.

[70] M. Höschele, A. Kampmann, and A. Zeller, "Active learning of input grammars," *arXiv preprint arXiv:1708.08731*, 2017.

[71] D. Cohn, Z. Ghahramani, and M. Jordan, "Active learning with statistical models," *Advances in neural information processing systems*, vol. 7, 1994.

[72] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, vol. 15, no. 8, p. e0237749, 2020.

[73] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané,

R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[74] DynamoRIO, "Dynamorio," ser. http://dynamorio.org/, June 2017.

[75] Intel, "Pin - a dynamic binary instrumentation tool," ser. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2017.

[76] M. Sablotny, "Pyfuzz2 - fuzzing framework," ser. https://github.com/susperius/PyFuzz2, 2017.

[77] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[78] M. Sablotny, B. S. Jensen, and C. W. Johnson, "Recurrent neural networks for fuzz testing web browsers," in *Information Security and Cryptology – ICISC 2018*, K. Lee, Ed. Cham: Springer International Publishing, 2019, pp. 354–370.

[79] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[80] G. Team, "libxml2," ser. https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home, 2022.

[81] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[82] Common Crawl, "Common crawl," ser. http://commoncrawl.org/, August 2018.

[83] gRPC authors, "grpc," ser. https://grpc.io, November 2021.

[84] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.