Recurrent Neural Networks for Fuzz Testing Web Browsers

Martin Sablotny(✉), Bjørn Sand Jensen, and Chris W. Johnson

School of Computing Science, University of Glasgow, Glasgow, Scotland
m.sablotny.1@research.gla.ac.uk,
{bjorn.jensen,christopher.johnson}@glasgow.ac.uk

Abstract. Generation-based fuzzing is a software testing approach which is able to discover different types of bugs and vulnerabilities in software. It is, however, known to be very time consuming to design and fine tune classical fuzzers to achieve acceptable coverage, even for small-scale software systems. To address this issue, we investigate a machine learning-based approach to fuzz testing in which we outline a family of test-case generators based on Recurrent Neural Networks (RNNs) and train those on readily available datasets with a minimum of human fine tuning. The proposed generators do, in contrast to previous work, not rely on heuristic sampling strategies but principled sampling from the predictive distributions. We provide a detailed analysis to demonstrate the characteristics and efficacy of the proposed generators in a challenging web browser testing scenario. The empirical results show that the RNN-based generators are able to provide better coverage than a mutation based method and are able to discover paths not discovered by a classical fuzzer. Our results supplement findings in other domains suggesting that generation based fuzzing with RNNs is a viable route to better software quality conditioned on the use of a suitable model selection/analysis procedure.

Keywords: Software security · Fuzz testing · Browser security

1 Introduction

Fuzz testing has recently enjoyed increased popularity in theoretical and practical software testing. This can be primarily attributed to the apparent capability to trigger unintended behaviour in complex software systems, e.g. the summary of bugs found by American Fuzzy Lop (AFL) [28] and further evidenced by the use of fuzz testing in software companies like Microsoft and Google (e.g. through their open-source tool ClusterFuzz [12]) which shows success and applicability in many different domains. However, the standard approach of combining mutation on a set of input examples with an evolutionary approach has its limitation with increasing necessity of keywords and compliance to syntactic rules (e.g. HTML as considered in this work). Those problems can be tackled by generation-based

K. Lee (Ed.): ICISC 2018, LNCS 11396, pp. 354–370, 2019.
https://doi.org/10.1007/978-3-030-12146-4_22

fuzzers that are able to comply to those rules, use the correct keywords and generate novel inputs. Traditionally, the time needed to develop generation-based fuzzers is dependent on the input specification's complexity. For example it is less time consuming to develop a generator for a network protocol, which has a single field with three different possible values compared to implementing the File Transfer Protocol (FTP) [16] with it various fields and states. In addition, it is necessary to find the right balance between introduced errors and overall correctness to trigger code paths that lead to unintended behaviour.

The main bottleneck in the development of generation-based fuzzers is the need for a strict understanding and implementation of the input file format. Therefore, the potentially complex input specification has to be studied carefully to transfer it into a test case generator, which then needs to be fine tuned in order to find the right balance between correctness and introduced errors into the test cases. This implicit optimization process looks to maximize code coverage by generating test cases that deviate in certain areas from the given specification and therefore are capable of exercising different low-level execution paths. Thus, it is clear that methods which could automatically derive or lean the input specification would be able to speed up software testing by faster deployment of generation-based fuzzing techniques. This would potentially lead to an increase in software security and stability.

Learning an input specification (e.g. syntactic rules) is obviously not trivial, especially due to the long time dependencies input specifications can apply. Those dependencies have an direct impact on the possible outputs at a certain position and therefore have to be captured by a learning algorithm to produce specification adhering outputs. However, recent advancements in generative machine learning models ([2,3,6,26]) have demonstrated how machine learning models can be use to learn complex rules and distributions from examples and generate new examples from acquired knowledge.

These advancements have been previously explored for fuzz testing by Godefroid *et. al.* [11]. They demonstrated the use of deep neural networks to generate PDF-objects, which were used as input for a rendering engine. Those input files were able to trigger new instructions in the rendering engine. However, they focused on the tension between learning the correct input structure and fuzzing - or in other words, finding the balance between adhering to the learned specification and deviating from it. They did not provide an analysis of the learning process itself and gave no comparison to a naive mutation based baseline. In addition, they have not provided any information about the overlap between the baseline and their proposed sampling strategies. In order to use deep learning models during fuzz testing, it is important to see whether it is worth the development and training. Therefore, it is necessary to compare it with an easy to implement approach, like a naive mutation algorithm. The analysis of an existing overlap between different approaches also gives more insight into the model and sampling choice, since it is important to trigger as much new execution paths as possible during testing to find the ones that trigger unintended behaviour.

In this work, we investigate how Recurrent Neural Networks (RNNs) with different types of cells can be trained and used as a HTML-fuzzers. The models are trained on a dataset created by a generation based HTML-fuzzer, which allowed us to adjust the dataset size and complexity in a fast and systematic way. We use the models to generate new HTML-tags from the resulting probability distribution, which were used to form test cases. Those were executed with Firefox [19] to gather their code coverage data and compared to a baseline generated by the HTML-tags from the dataset and a naive mutated dataset. Thus, the contribution of the paper includes:

- A systematic and robust approach for training and evaluating recurrent neural networks with different types of cells for HTML fuzz testing.
- A procedure and metrics for model-selection and comparison of machine learning fuzzers against standard and a vanilla mutation-based methods including a similarity-based analysis.
- An extensive empirical evaluation on a web browser, demonstrating that learned fuzzers are able to outperform standard test methodologies.
- Open-source implementation and data available via Github[1].

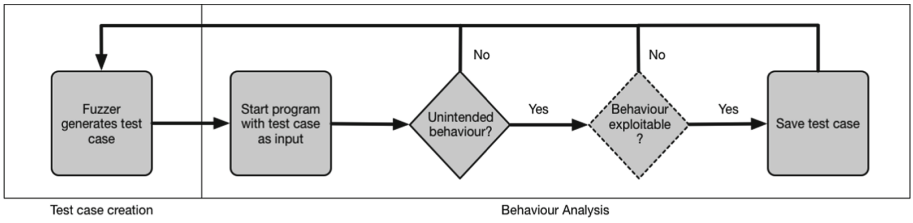


Fig. 1. Classic fuzzing workflow for finding security related flaws

2 Background

2.1 Fuzzing

Fuzz testing is a dynamic software testing approach, hereby dynamic means the software under test is actually executed in contrast to statically analysed. The goal of the fuzz test is to provoke unintended behaviour that was not detected in earlier testing stages, therefore software under test is executed with inputs created by a so-called fuzzer. Those inputs do not fully comply with the underlying input specification in order to find paths that lead to a state that triggers unintended behaviour. We adopt a broad definition of unintended behaviour, which makes it applicable for various kinds of software and devices [27]. For example,

[1] Code and data is available from https://github.com/susperius/icisc_rnnfuzz.

during fuzz testing desktop software, unintended behaviour can be the termination of a running process or even the possibility to take control over a process. Whereas during the test of a web application unintended behaviour might be defined as an information leak or the circumvention of access restriction both cases might happen due to a SQL-injection vulnerability, where arbitrary input is used as a valid SQL-statement.

As those examples highlight, a case of unintended behaviour becomes more severe if it could provide an attacker with an advantage. Here advantage can mean everything from accessing restricted information to taking over control of a device. In order to find those vulnerabilities fuzz testing is utilised. The general workflow during fuzz testing is shown in Fig. 1. The testing itself is split in two parts first the test case generation and secondly the behaviour analysis. In general, the creation of test cases during fuzzing can be divided into the two categories: mutation based and generation based [27], [8] and [20]. First mutation based fuzzing uses a valid input set and a mutation fuzzing in order to derive new test cases from the input set. This type of fuzzing can be implemented quickly if the input examples are available (e.g. JPG files). The main disadvantage is that test cases created by plain mutation based fuzzing are not able to quickly discover code paths deep in the call tree because many created test cases are filtered out in early program execution stages. A very prominent and successful example of this category is the aforementioned fuzzer AFL with its evolutionary mutation approach. Secondly, generation based fuzzing uses an approach where test cases are created from scratch, for example through grammar based creation. This method needs a lot of effort during studying the input structure and developing the generator but in general it is able to discover deeper lying code paths. However, a balance between complying to the rules and breaking them has to be found in order to provoke unintended behaviour in the target.

2.2 Recurrent Neural Networks

The input data for many software products is readily available on the internet (e.g. HTML, JPG, PNG) and deep learning algorithms have shown their performance in different use cases especially where they are trained on a large available dataset, for example text generation [26], program creation [3] and machine translation [2,6]. This led us to the use of a generative model for the test case creation during fuzz testing. In addition the structure of HTML and other input formats, where the actual character or byte is dependant on the previous positions in a sequence led to the use of RNNs.

RNNs are used to model sequential data, e.g. for text generation [26], language modelling and music prediction [21]. 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 vector $\mathbf{h_t}$ and an output $\hat{\mathbf{y}}_\mathbf{t}$ at time step t as follows

$$\mathbf{h_t} = f_h(\mathbf{x_t}, \mathbf{h_{t-1}}) \ , \ \hat{\mathbf{y}}_\mathbf{t} = f_o(\mathbf{h_t}),$$

with f_h and f_o being the hidden transformation and output function respectively. Hereby, the input $\mathbf{x_t}$ can be a N-dimensional vector, representing the input structure, e.g. a single pixel's RGB values at position t.

As described by Hochreiter [13] and later by Bengio et al. [4], RNNs suffer from either the vanishing or exploding gradient problem. This means that the weight updates are becoming infinitesimal during training, which consumes a lot of time but does not lead to a better optimised network. Hochreiter and Schmidhuber introduced the concept of Long-Short Term Memory (LSTM) cells [14] RNNs using those cells do not suffer from the vanishing (exploding) gradient problem. LSTM cells use a hidden state, a candidate value and three gates namely a forget gate, an input gate and an output gate. The gates control how much information is forgotten, used from the input and controlling the flow into the new hidden state respectively. They are default feed forward neural networks and each have their own trainable parameters.

Another popular RNN cell, the Gated Recurrent Unit (GRU) was introduced by Cho et al. [6]. This unit only uses two gates, a reset and an update gate. Here the reset gate controls what information from the past hidden state is forgotten and the update gate controls the information flow into the new hidden state. This simpler model arguably makes it easier to train than a standard LSTM based model.

The capability to learn sequential structures, where dependencies to former inputs exist, is obviously an important characteristic when learning input format structures for test case generation. This is especially evident in for example HTML where there are long term dependencies between an opening-tag and the corresponding closing-tag.

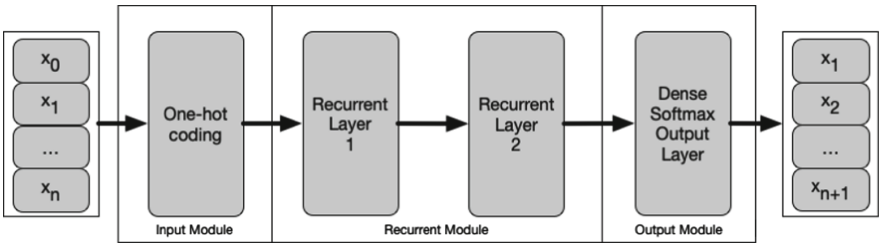


Fig. 2. Model overview for a stacked RNN with 2 recurrent layers (either LSTM or GRU)

3 Stacked RNN for HTML-Fuzzing

The basic concept of the model used in this work is shown in Fig. 2. The model consists of three modules. First, the input module, let $X = \{x_1, x_2, \ldots x_N\}$ be the sequence of input values with $x_t \in \mathbb{N}_0 \mid 1 \leq t \leq N$, where x_t is the

natural number representing the character at position t in the input sequence. For example the character 'f' is at position t in the input sequence, its assigned number is 17 and $x_t = 17$.

The input module then takes such a x_t and transforms it into a one-hot coded vector $\hat{\mathbf{x}}_\mathbf{t} \in \mathbb{R}^\mathbf{I}$ with $I = max(X) + 1$, the one is added to account for the zero. Let $\hat{\mathbf{x}}_\mathbf{t} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_I)^\intercal$ then

$$\hat{x}_j = 0 \; \forall \; 1 \leq j \leq I \; : \; j \neq x_t \; \vee \; \hat{x}_j = 1 \; \Leftrightarrow \; j = x_t,$$

and for the former example character 'f' all $\hat{x}_j = 0$, except for $\hat{x}_1 7$, which equals 1. This conversion from integer values is necessary as interpret our input as categorical data (each character is its own category) and those categories are handled as features during the training process.

Secondly, the recurrent module consists of LSTM or GRU nodes as described in Sect. 2.2 with $s, l \in \mathbb{N}$ hereby s is the internal size of the nodes and l the amount of layers used, e.g. $l = 2$ for the LSTM based model shown Fig. 2. LSTM cells have demonstrated a high performance gain compared to the basic RNN approach as demonstrated by Chung et al. [7]. Gated Recurrent Units (GRUs) introduced by Cho et al. [6] perform similar to LSTM cells [7], however Jozefowicz et al. [17] have shown that LSTM cells perform better during XML modelling. We decided to evaluate the performance of both cells to analysis whether the XML modelling results are transferable to HTML modelling.

Finally, the output layer consists of a default feed forward network with I nodes. It takes the output of the last recurrent layer $\mathbf{h}_t^l \in \mathbb{R}^s$ as input value and after computing its output the $softmax$ function is applied. The resulting $\hat{\mathbf{y}}_\mathbf{t}$ provides the probability distribution for predicting the next value of the input sequence. The goal during training is to minimise the cross entropy loss function

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

where Θ denotes the model's parameters (i.e. a collection of $\mathbf{W}$'s and $\mathbf{b}$'s). In order to find a Θ that minimises the above loss $\mathcal{L}$ the ADAM [18] optimisation algorithm is applied. It is a gradient-based optimisation algorithm which only needs first order gradients and has a reduced memory footprint compared to other algorithms. Additionally, Dropout (30% dropout probability) [25] is used as regularisation.

4 Experiments

The following sections present the methodology that was used to validate our application of RNNs to generate test cases for fuzz testing of cyber security in complex systems.

The basic idea is to train the aforementioned neural networks with different depths on a large collection of HTML-tags. After training those models are used

to generate HTML-tags directly using the probability distribution over characters given the sequence. The generated output is then used as input for a web browser. This browser is instrumented in order to gather the code coverage data during execution on a basic blocks basis. The collected code coverage data is then used to compare the models' performances with code coverage data collected by executing the dataset's HTML-tags and a naive mutation strategy performed on this HTML-tags.

4.1 Environmental Setup and Implementation

The model training took place on a Ubuntu 16.04 system equipped with a single NVIDIA GeForce 1080 Ti and a NVIDIA GeForce TITAN Xp, which shortens the necessary training time by utilising their parallel computational capabilities. The models were implemented using Google's TensorFlow framework [1] along with its Python bindings. This frameworks already provides the necessary cell types, optimisation algorithm and loss function for our model, which shortens the development time.

The code coverage data was collected on a Virtual Machine (VM) also running Ubuntu 16.04 and Firefox 57.0.1, which allows to run in so-called headless mode. In this mode Firefox does not display the graphical user interface, but it still renders the webpage. We also modified the standard configuration in order to disable internal services to avoid as much false code coverage data as possible. Furthermore safe mode was disabled, because during the automated code coverage collection Firefox was not closed correctly and therefore might tries to start in safe mode after just a few test cases. The use of the headless mode also saves time during the code coverage collection, which was collected by DynamoRIO's drcov tool (see Subsect. 4.4). The VM itself utilises 16 GB of RAM and a Solid State Disk. A VM was used to facilitate parallel data collection via cloning and deploying onto multiple host systems.

4.2 Data Set Generation

In order to provide a reproducible and controlled experiment, the training (and ground-truth) data set was generated by an existing HTML-fuzzer included in PyFuzz2 [24]. It provides a controllable generator thus ensuring less uncertainty about the variation within the training dataset in comparison to collecting a dataset from the Internet. Therefore, it was possible to control the complexity of the generated HTML on a per tag basis, whereas a collected set would have to be parsed and then filtered for unwanted HTML-tags to control the resulting dataset.

The pre-existing fuzzer was modified in order to avoid nesting of HTML tags, remove all Cascading Style Sheets and output exactly one HTML tag per line. Due to the restriction of not having nested HTML-tags some like *td* or *th* are excluded because they need an outer tag in this example *table*. Those restrictions were introduced to reduce to focus on the fundamental problem by reducing the

overall data set complexity. This further reduced the necessary model complexity and effectively the time needed to train those models.

Listing 1.1 shows an excerpt from the data set used for training the models, which highlights the modification mentioned above. The created file consisted of 409,000 HTML-tags, which results in a total size of 36 MB.

4.3 Training

All models were trained to predict the input shifted by one on a per character basis. For example take "$< h2\ i$" from line 1 in Listing 1.1 as input sequence of length 5 then the label for that particular input sequence would be "$h2\ id$". The actual sequence length used during training was 150 characters and each model was trained for 50 epochs, which has shown sufficient for the models to converge. In order to train the models we used the previously mentioned ADAM [18] optimisation algorithm. The starting learning rate was set to 0.001 and halved every 10 epochs. The models were trained with a batch size of 512. The internal size of the LSTM and GRU cells was set to 256 for all models trained and the number of layers varied from 1 to 6. The weights of the layer were initialised by the Glorot uniform initializer [10]. So the weights are drawn from a uniform distribution in the interval $(-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}})$, with n_j being the internal size of layer j.

The first 30 MB of the data set were used for training and an additional generated 1MB for validation. All models were trained on 5 different training/validation splits repeated 3 times with different initialization (to mitigate extremely poor local minima) which results in a total of 90 trained models per cell type. The splits were chosen randomly without overlapping parts.

4.4 Data Collection

The code coverage data was collected by executing Firefox instrumented by DynamoRIO's *drcov* [9]. This tool gathers data about the executed basic blocks of the program under test. The collected code coverage data was parsed for uniquely executed basic blocks inside of Firefox's `libxul.so` library, which includes the whole web engine responsible for HTML rendering. It is possible to identify those basic blocks even when the process is restarted because the recorded data uses the offset of the basic block from the base address of the

```
1 <h2 id="id0" style="style" spellcheck="false" dir="rtl"
    title="eval(n1, $)"> 2e100 </h2>
2 <ul id="id3" style="style" translate="no"
    contenteditable="true" tabindex="4400000000">
    4400000000 </ul>
```

Listing 1.1. Example from the training set.

library in memory and this offset is always the same for a fixed version. Hereby a basic block is defined as a linear sequence of machine instructions with a single entry (branch target) and single exit (branch instruction).

All test cases consisted of a basic HTML-template with the HTML-tags inserted into the body tag. Initial experiments showed that executing the same test case multiple times returns different code coverage data. This is due to the other functions that are bundled into the `libxul.so` library, which are not part of the web engine itself. Those functions might for example only be executed after a number of restarts or in fixed time intervals. In order to identify the corresponding basic blocks the blank HTML-template was executed $1,024$ times and the resulting code coverage was store for later use.

The comparison baseline was established by using the HTML fuzzer to create $6 \times 16,384$ HTML-tags Each collection of $16,384$ HTML-tags was then used to create two datasets, one containing 64 files with 256 HTML-tags each and a second one with 128 files containing 128 HTML-tags. This resulted in twelve datasets.

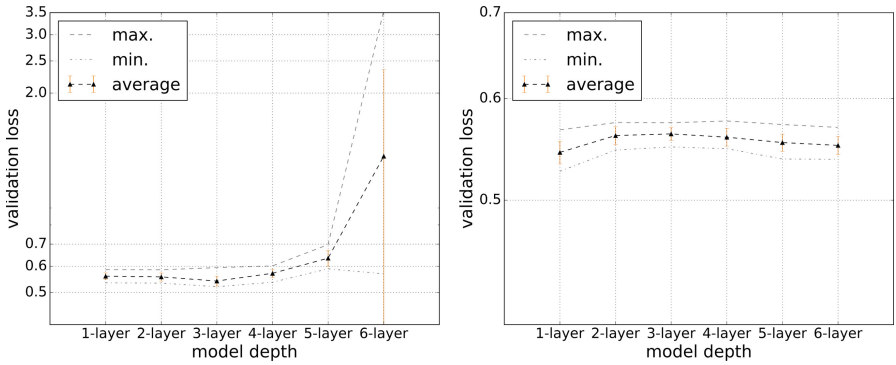
In order to establish a second baseline for comparison, additional test sets were created by mutating the dataset test cases and collecting the code coverage from those. A simple mutation function was applied with a fixed chance that a position is replaced by a randomly chosen character (only characters that were already present in the dataset). The results were 20 additional test case sets, 10 sets consisting of 128 cases with 128 HTML-tags each and 10 sets consisting of 64 cases with 256 HTML-tags each, resulting in a total of $1,920$ additional cases. The replacement probability varied between 0.1% and 51.2%. This was done to ensure that there is difference and therefore an incentive to use a trained model for test case creation instead of implementing a naive mutation based approach.

For each trained model, a total $16,384$ HTML-tags were generated and then used to create two different sets of test cases. The first set used 128 HTML-tags per case, which resulted in 128 cases per model trained, whereas the second set used 256 HTML-tags per case, which resulted in 64 cases per model. This was done to analyse the impact of HTML-tags on code coverage and to observe the relationship with the model performance. The HTML-tags were generated by using the "<" character as starting input, sampling the next character from the resulting probability distribution, which was then used as new input. This was repeated until a "\n" (newline character) was sampled, since it marks the end of a HTML-tag.

Finally, the set difference between the collections of basic block sets from the test cases and the blank cases was computed to filter out the aforementioned irrelevant basic blocks.

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

Listing 1.2. Example HTML-tag from a 1-layer LSTM model



(a) LSTM (b) GRU

Fig. 3. Average validation loss for models of different complexity (i.e. number of layers) models and dataset splits. Error-bars indicate the standard deviation.

4.5 Results

The training phase already showed a difference in behaviour between the two cell types. The LSTM based models showed a decrease in average validation loss and standard deviation up to three layers, as shown in Fig. 3a, with an increase afterwards. Especially, the 6-layer models show a large standard deviation and a huge increase in average validation loss compared to the other models This indicates that those models have too many parameters in order to be trained on our problem and training set. This behaviour is to be expected from a general machine learning perspective and since the training process is the same compared to other similar applications using generative neural networks, like generating text.

In contrast the training of the GRU based models showed a small increase from the 1-layer models to the 2-layers case, but a decrease afterwards with overall small differences in the standard deviation. This indicates that the GRU based models are either better suited to reproduced the input structure or do not reach the overall complexity of the 6-layer LSTM based model, which is also supported by comparing the trainable parameters of those models. The GRU based model has $2,276,971$ compared to $3,026,795$.

Overall, a small numeric difference in validation loss can lead to a big difference in the quality of the resulting HTML-tags. For example Listing 1.2 shows an

```
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 1.3. Example HTML-tag form a 3-layer LSTM model

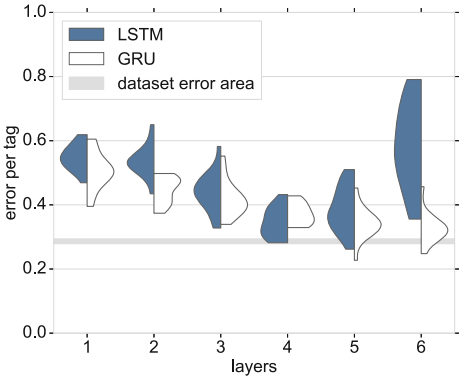


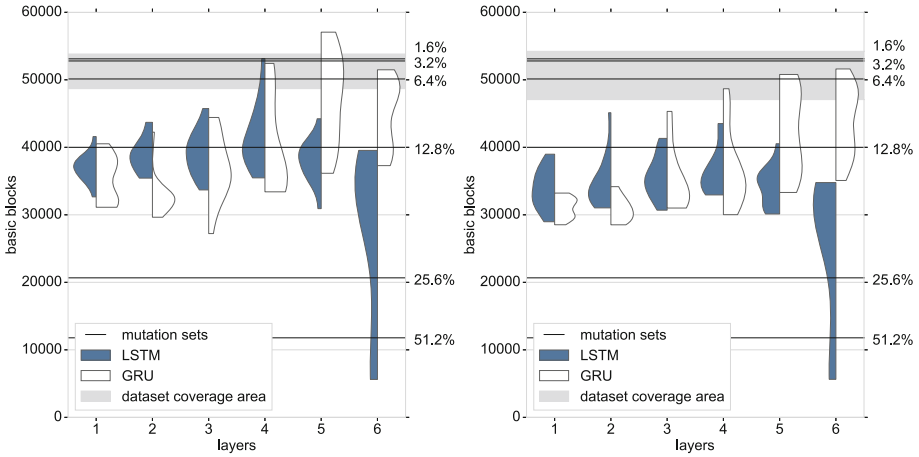
Fig. 4. Average error rate per HTML-tag generated by the LSTM and GRU based model in comparison to the datasets.

excerpt generated by a 1-layer LSTM model. It is barely recognisable as HTML and the model did not generate existing HTML-opening and closing tags and two of the generated HTML-attributes are misspelled in this particular example. In contrast to that Listing 1.3 shows two HTML-tags generated by a 3-layer LSTM model. Both use only existing HTML-tags, however the second one does not use the correct closing tag and misspelled one attribute name. Further evidence regarding the quality differences between the models of both cell types is provided by Fig. 4. It shows how the HTML error rate per tag follows the trend of the validation loss and highlights how small differences has a large effect on the HTML quality. The high spread of the 6-layer LSTM HTML error rate reflect the large standard deviation observed during training.

Test Cases with 128 HTML-Tags

In terms of code coverage performance the overall trend also follows the validation loss and standard deviation, where a smaller validation loss and standard deviation indicates a better performance. Figure 5a shows the total discovered basic blocks of both cell types per layer. It highlights that both types of 4-layer models and the GRU 5 and 6-layer models are able to discover basic blocks in the range of the datasets or even outperform it.

In addition, Fig. 6a shows the difference in number of basic blocks to the best performing dataset. It shows that all models were able to discover basic blocks not triggered by the dataset, with the 5-layer GRU models performing best on average. In comparison with the different mutation sets the maximum overlap reaches 90% with a mutation chance of 1.6%, which is not surprising because the same mutation set has an overlap of 87.6% with the best performing dataset, as also shown in Fig. 7. The best performing 5-layer GRU models have an overlap of 78% with the union of different mutation chances, highlighting the models ability to discover basic blocks, which can not be triggered by the naive mutation approach. The overall best performing models are also those with the largest overlap with the dataset.



(a) 128 HTML-tags per case (b) 256 HTML-tags per case

Fig. 5. Total number of uniquely discovered basic blocks on a per model basis. The dataset coverage area and the different mutation sets are included as baselines with the mutation probability indicated on the right vertical axis.

Test Cases with 256 HTML-Tags

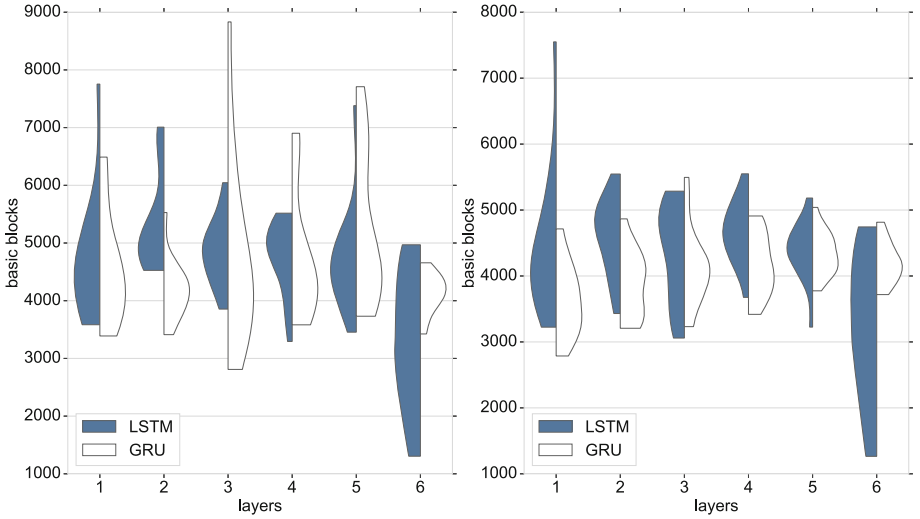
The code coverage results for the test cases with 256 HTML-tags each showed a similar development, but a slightly lower overall performance, as shown in Figs. 5b and 6b. The lower overall performance was expected, because both runs basically use the same HTML-tags and only the number of inserted HTML-tags is different.

In terms of absolute basic blocks the 4-layer model was the best LSTM based model, however in this setting it did not reach the dataset coverage area. However, the 4-, 5- and 6-layer GRU based models were able to reach the dataset coverage area with the 6-layer model having the highest number of uniquely triggered basic blocks.

Considering the overlap with the mutation test cases the overall result is the same as in the 128 HTML-tags case. The best performing four layer models have an average overlap with the mutation sets of 74.6%. This shows that the 256 HTML-tags cases were also able to trigger new code paths in the web rendering engine.

5 Discussion

The results demonstrate that is is indeed possible to successfully train models and generate test HTML cases using the RNN based model. However, it is crucial to monitor this process to get robust results, e.g., the 6-layer LSTM model was not trainable in a reliable way. This may very well have been due to a lack of training data, or the high amount of parameters involved in the optimisation.



(a) 128 HTML-tags per case (b) 256 HTML-tags per case

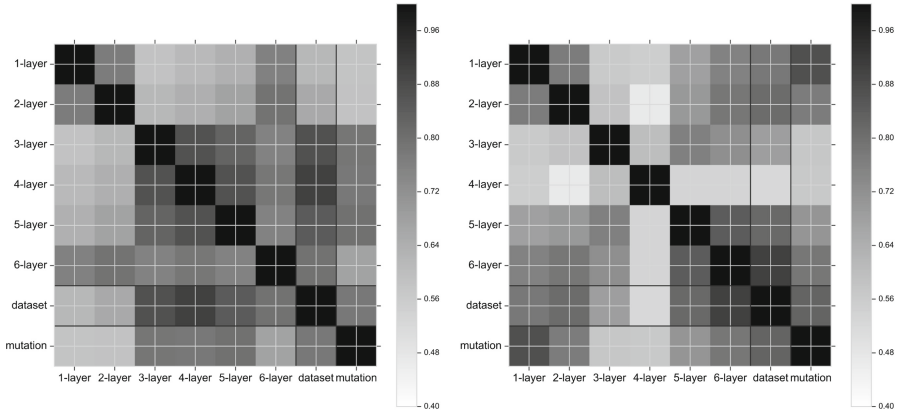
Fig. 6. Number of uniquely discovered basic blocks that were not triggered by the best performing dataset.

Once the models have been trained the results indicates that the average validation loss can be used as good initial selection criteria for choosing a good model for generation of test cases despite the implicit coupling with the code coverage metric. This is particularly interesting, since there is no code coverage data available during the model selection phase and covering as many code paths as possible during fuzz testing is important to discover software bugs. The results also have shown that the HTML error rate can be used to determine a good generative model and therefore augment the selection process. This is especially helpful, since the average validation loss and standard deviation alone might indicate a low difference between two models, see for example the Listings 1.2 and 1.3. The highest average validation loss difference between those models is $\leq$0.02, but the difference in the HTML error rate is 0.3. This means that the worst performing 1-layer LSTM model has twice as many error per tag than the best performing 3-layer LSTM model.

Overall the best performing models generated more valid HTML-tags than the other models, which leads to the use of existing HTML-tags. Those generated and generally valid HTML-tags are not always closed with right corresponding HTML-tag. This results in the best performing models building nested valid HTML-tags by accident, because those models use a valid opening HTML-tag, but do not generate the corresponding closing HTML-tag. However, this might still be generated at a later stage in the file. The assumed rendering behaviour and the creation of nested HTML-tags trigger code paths that have not been triggered by the baseline set, since in the baseline set every opened tag is closed with the corresponding closing tag in each line.

The similarity in terms of overlapping basic blocks (see Fig. 7a) between the LSTM models and the baseline set is lower than the overlap with the mutation sets and the models between each other in the 128 HTML-tag case. This might indicate that the models are not able to fully replicate the given input structure and therefore another model choice would be better suited to learn this structure or the provided training set was too small to capture the input structure with the chosen model architecture. For the GRU models the best performing models also show that the overlap with dataset is higher than the one with the mutation sets (see Fig. 7b). This further strengthens the assumption that a certain quality has to be reached by the models in order perform well.

Overall, we were able to demonstrate that especially GRU-based RNNs are capable of creating HTML-tags, which then can be used during fuzz testing a browser. Critically, the generated HTML test cases are also able to trigger a significant number of unique basic blocks, which were not reached by the dataset's baseline and the naive mutation approach.



(a) LSTM (b) GRU

Fig. 7. The similarity between all the models, the dataset and mutation-based fuzzer in terms of their overlapping basic blocks for test cases with 128 HTML-tags.

6 Related Work

The closest related work was done by Godefroid et al. [11]. They studied the achievable code coverage using a two layer stacked RNN to sample PDF-objects and focused on the effects the training duration has on this. The code coverage results they achieved were compared against a baseline, which was randomly selected from the training set. In contrast to that we used data not seen by the models during the training phase to establish our baseline for comparison. In addition, they analysed different approaches of creating test cases and compared

those. They also highlighted an observed tension between learning and fuzzing and proposed an algorithm called SampleFuzz. This algorithm uses the lowest predicted probability, if the model's highest predicted probability is above a certain threshold value and a random coin toss is successful. Whereas our work studied a different input format, namely HTML, which is a more structure-reliant input format compared to PDF-objects. We also researched the effects of the model depth on the resulting code coverage. We were not able to observe the former described tension between learning and fuzzing. This might be connected to the relative large size of our training set or indicate that their models started to overfit to the training examples, thus requiring additional stochasticity to produce novel test cases. Regardless, we did not identify the need to introduce additional random values (e.g. through the use of SampleFuzz).

Other related works make use of the control and data flow during the execution in order to generate new test cases. Rawat et al. [23] utilise so-called evolutionary algorithms to derive new test cases. Whereas Höschele et al. [15] derives an input grammar from the collected execution information. Both approaches need direct access to the program under test to instrument it and to collect the necessary data. In contrary, our approach is able to learn the input structure directly from input examples, which shortens the design and learning process.

A different approach utilising code coverage and mutation-based fuzzing was presented by Böhme et al. [5]. They augmented AFL with Markov Chains in the mutation process. Their AFLFast called approach uses Markov Chains to determine the state transitions into new test inputs. They have shown that they shorten the time necessary for finding bugs in an ensemble of tested software. However, they have not provided any information on highly structure dependent input formats like HTML, which is described as a shortfall in the general AFL approach.

Another way of combining deep learning in order to find bugs in software was evaluated by Pradel et al. [22]. They used trained models in order classify potential buggy source code. Hereby they trained their models as individual classifiers for a certain bug category. In contrast to them we trained our models to generate inputs, which then can be used to trigger and observe bugs in software. Furthermore, their approach needs direct access to the source code, whereas we need access to enough input examples to train a RNN model.

7 Conclusion and Future Work

Our work provides evidence that it is possible to use a stacked RNN to generate HTML-tags in order generate novel test cases for fuzz testing a browser's rendering engine. The results also clearly show that the GRU based models are able to outperform LSTM ones even with less trainable parameters. Furthermore, the proposed evaluation procedure and similarity-based analysis demonstrates that the overlap in basic blocks between the dataset and the model generated test cases are very low on average. In addition, the overlap with the naively mutated sets is approximately 70% on average, which indicates that the trained networks

are able to discover new code paths formerly not discovered by the naive mutation approach with different mutation chances. This provides amble evidence that RNNs can be trained and used as an effective HTML-fuzzer provided that a suitable model-selection and analysis procedure is applied.

We are currently looking to extent the present work in least three ways: Firstly, investigating more complex/suitable neural network models is necessary to improve the overall quality of the generated HTML as other prevalent web technologies, like JavaScript, cannot be used on broken HTML-tags. Secondly, it is important to validate the generalisation of the current work on real-world HTML-examples in contrast to the fuzzer generated training data considered here. Lastly, we are exploring ways to utilise the gathered code coverage data during the training process and rewarding the learning algorithm when discovering unintended behaviour or new code paths. We speculate that this can be achieved with the help of reinforcement learning to systematically trade-off the model fit vs exploration.

Acknowledgements. We gratefully acknowledge the support of NVIDIA Corporation with the provision of the GeForce 1080 Ti and the GeForce TITAN Xp used for this research. We also like to thank Chris Schneider from NVIDIA for his ongoing interest in our research and his support.

References

1. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems (2015). http://tensorflow.org/
2. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)
3. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: learning to write programs. arXiv preprint arXiv:1611.01989 (2016)
4. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**(2), 157–166 (1994)
5. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based Greybox Fuzzing as Markov Chain. IEEE Trans. Softw. Eng., 1 (2018). https://doi.org/10.1109/TSE.2017.2785841. ISSN 0098-5589
6. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
7. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555 (2014)
8. DeMott, J.: The evolving art of fuzzing. DEF CON **14** (2006)
9. DynamoRIO: Dynamorio, June 2017. http://dynamorio.org/
10. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256 (2010)
11. Godefroid, P., Peleg, H., Singh, R.: Learn&fuzz: machine learning for input fuzzing. In: Automated Software Engineering (ASE 2017) (2017)
12. Google: Using clusterfuzz. http://dev.chromium.org/Home/chromium-security/bugs/using-clusterfuzz

13. Hochreiter, S.: Untersuchungen zu dynamischen neuronalen netzen. Diploma Technische Universität München **91** (1991)
14. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
15. Höschele, M., Zeller, A.: Mining input grammars from dynamic taints. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 720–725. ACM (2016)
16. Postel, J., Reynolds, J.: File transfer protocol. Technical report, October 1985. https://tools.ietf.org/html/rfc959
17. Jozefowicz, R., Zaremba, W., Sutskever, I.: An empirical exploration of recurrent network architectures. In: International Conference on Machine Learning, pp. 2342–2350 (2015)
18. Kingma, D., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
19. Mozilla Corporation: Firefox, August 2018. https://www.mozilla.org/en-US/firefox/
20. Oehlert, P.: Violating assumptions with fuzzing. IEEE Secur. Priv. **3**(2), 58–62 (2005)
21. Pascanu, R., Gulcehre, C., Cho, K., Bengio, Y.: How to construct deep recurrent neural networks. arXiv preprint arXiv:1312.6026 (2013)
22. Pradel, M., Sen, K.: Deep learning to find bugs (2017)
23. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: application-aware evolutionary fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2017)
24. Sablotny, M.: Pyfuzz2 - fuzzing framework (2017). https://github.com/susperius/PyFuzz2
25. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**(1), 1929–1958 (2014)
26. Sutskever, I., Martens, J., Hinton, G.E.: Generating text with recurrent neural networks. In: Proceedings of the 28th International Conference on Machine Learning (ICML 2011), pp. 1017–1024 (2011)
27. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education (2007)
28. Zalewski, M.: American fuzzy lop (2017). http://lcamtuf.coredump.cx/afl/