An Automatic Test Data Generation Tool using Machine Learning

Ciprian Paduraru[1,2,3] and Marius-Constantin Melemciuc[2]
[1]*The Research Institute of the University of Bucharest (ICUB), University of Bucharest, Bd. M. Kogalniceanu 36-46, 050107, Bucharest, Romania*
[2]*Department of Computing Science, University of Bucharest, Romania*
[3]*Electronic Arts, Romania*

Keywords: Fuzz Testing, Recurrent Neural Networks, LSTM, Tensorflow, Pipeline.

Abstract: This paper discusses an open source tool that is capable to assist users in generating automatic test data for multiple programs under test. The tool works by clustering inputs data from a corpus folder and producing generative models for each of the clusters. The models have a recurrent neural network structure and their training and sampling are parallelized with Tensorflow. As features, the tool supports online updating of the corpus folder and the already trained models, and supports any kind of program under test or input file example. There is no manual effort for users, other than customizing per cluster parameters for optimizations or using function hooks that they could use through a data structure, which acts as an expert system. The evaluation section shows the efficiency of both learning and code coverage using some concrete programs and new tests sampling methods.

1 INTRODUCTION

The importance of security in software systems has increased year over year recently, because of the wide interconnectivity between different software pieces. Important resources are invested nowadays in detecting security bugs in these systems before being released on the market. Machine generated test data is desirable for automatizing the process of testing and ensuring a better coverage.

Ideally, the purpose of an automatic test data generation system for programs evaluation should be to generate test data that covers as many branches as possible from a program's code, with the least computational effort possible. The most common technique is *Fuzz testing* (Godefroid, 2007), which is a program analysis technique that looks for inputs causing errors such as buffer overflows, memory access violations, null pointer dereferences, etc, which in general have a high rate of being exploitable. Using this technique, testing data is generated using random inputs and the program under test is executing them for the purpose of detecting issues like the above mentioned ones. One of the main limitations of fuzz testing is that it takes a significant effort to produce inputs that covers almost all branches of a program's source code. This comes from the fact that using randomness, it results in a high chance of producing inputs that are not correct and rejected in the early outs of a program's execution.
Alternative methods that augment the classic random fuzz testing with different methods were created. Such ideas involved the use of genetic algorithms for better guiding the test data generation towards uncovered areas (Paduraru et al., 2017), or by using recurrent neural networks and predicting the probability distribution of the next character knowing a previously generated context (Godefroid et al., 2017), (Rajpal et al., 2017).

This paper discusses an open-source tool (from the authors' knowledge, the first one at the moment of writing this paper) that given a corpus of different existing test file formats, it performs cluster analysis, then learns a generative model for each cluster, which can be used later to quickly generate new tests with a high rate of being correct (i.e., touching more branches of a program instead of taking the early outs due to incorrect inputs). More specifically, the contributions of this paper in the field of using machine learning for automating software testing are:

- An open-source tool that is capable of storing a database of generative models for sampling new test data for multiple programs at once. These models are learned from a corpus of test data, which

Paduraru, C. and Melemciuc, M-C.
An Automatic Test Data Generation Tool using Machine Learning.
DOI: 10.5220/0006836604720481
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 472-481
ISBN: 978-989-758-320-9

can be updated online with newly added content. No manual clusterization of inputs is needed.

- Description of a parallelized implementation for learning the models and sampling from them using Tensorflow (Abadi et al., 2016). The models also permit checkpoints and online learning.
- Present a technique for assigning begin/end markers in the pre-processed training data that works for all kinds of files, not just the well-known ones. The previous work in the field that uses the same core system as our tool (i.e. recurrent neural networks) is focused only on PDF files.
- Allows users to leverage expert system in oversizing the work and perform custom optimizations and logs for learning or sampling certain categories of file types.

The paper is structured as follows. Next section presents some existing work in the field that inspired the work presented in this paper. Section 3 makes a quick introduction in one of the ways machine learning can be used to generate new texts based on an existing corpus of texts. Section 4 presents our methods for automating the process of test data generation. Evaluation of our tool and methods are discussed in Section 5. Finally, conclusions and future work are given in the last section.

2 RELATED WORK

In the field of fuzzing techniques, there are three main categories currently: blackbox random fuzzing (Sutton et al., 2007), whitebox random fuzzing (Godefroid et al., 2012), and grammar based fuzzing (Purdom, 1972), (Sutton et al., 2007). The first two are automatic methods proving efficiency in finding vulnerabilities in binary-format file parsers. These methods are also augmented with others for better results. For example, in (Paduraru et al., 2017) authors present a distributed framework using genetic algorithms that generates new tests by looking at the probability of each branch encountered during the execution. Their fitness function scores a newly generated input test by the probability of the branches encountered in the program's execution trace. This way, the genetic algorithm tries to create input data that drives the program's execution towards rare (low probability) branches inside the program's control flow. They use Apache Spark for parallelization and dynamic tainting to know the paths taken during the execution. Their method obtains better scores than classical random fuzzers and it is one of the solutions that we compare against, using the same two examples: an HTTP parser and an XML parser.

On the other side, the grammar based fuzzing is not fully automatic: it requires a grammar specifying the input format of the application under test. Typically, this grammar is written by hand and the process becomes time consuming and error prone. It can be viewed as a model-based testing (Utting et al., 2012), and the work on it started with (Hanford, 1970), (Purdom, 1972). Having the input grammar, test generation from it can be done either (usually) random (Sirer and Bershad, 1999), (Coppit and Lian, 2005) or exhaustive (Lämmel and Schulte, 2006). Methods that combine whitebox fuzzing with grammar-based fuzzing were discussed in (Majumdar and Xu, 2007), (Godefroid et al., 2008). Recent work concentrates also on learning grammars automatically. For instance, (Bastani et al., 2017) presents an algorithm to synthesize a context-free grammar from a given set of inputs. The method uses repetition and alternation constructs for regular expressions, then merging non-terminals for the grammar construction. This can capture hierarchical properties from the input formats but, as mentioned in (Godefroid et al., 2017) the method is not well suited for formats such as PDF objects for instance, which include a large diverse set of content types and key-value pair.

Autogram, mentioned in (Höschele and Zeller, 2016) learns context-free grammars given a set of inputs by using dynamic tainting, i.e. dynamically observing how inputs are processed inside a program. Syntactic entities in the generated grammar are constructed hierarchically by observing what parts of the given input is processed by the program. Each such input part becomes an entity in the grammar. The same idea of processing input formats from examples and producing grammars, but this time associating data structures with addresses in the application's address space is presented in (Cui et al., 2008).

Both approaches described above for learning grammars automatically require access to the program for adding instrumentation. Thus, their applicability and precision for complex formats under proprietary applications such as PDF, DOC or XML parsers is unclear. Another disadvantage of these is that if the program's code changes, the input grammar must be learned again. The method presented in (Godefroid et al., 2017) uses neural-network models to learn statistical generative models for such formats. Starting from a base suite of input PDF files (not binaries) they concatenate all and use recurrent neural networks (RNN, and more specifically a sequence - to - sequence network) to learn a generative model for other PDF files. Their work is focused on generative

models for non-binary objects and, since for binary formats such as an image embedded in PDF, the existing methods for fuzz testing (classical ones) are already efficient. Their work is a foundation for our open-source pipeline solution, which is able to generate models from any kind of input files in a distributed environment that also supports online learning, and produce new test inputs based on a database of models. If the base method evaluation was done on PDF parsers, our tests also include HTTP and XML parsers.

3 USING MACHINE LEARNING TO LEARN GENERATIVE MODELS FOR TESTING

A statistical learning approach for learning generative models for PDF files was introduced in (Godefroid et al., 2017). Their main idea is to learn the model based on a large corpus of PDF objects using recurrent neural networks, and more specifically a sequence-to-sequence network model (Cho et al., 2014), (Sutskever et al., 2014). This model has been used for machine translation (Sutskever et al., 2014) and speech recognition (Chorowski et al., 2015), producing state of the art results in these fields. The model can be trained in an unsupervised manner to learn a generative model from the corpus folder, then used to produce new sequences of test data.

3.1 Sequence-to-Sequence Neural Network Model

Recurrent neural networks (RNN) are neural network models that operate on a variable input sequence $< x_1, x_2, ..., x_T >$ and have a hidden layer of states h, and an output y. At each time step (t) one element from the input sequence is consumed, modifying the internal hidden state and the output of the network as follows:

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

$$y_t = \sigma(h_t) \tag{2}$$

where σ is a function such as softmax (used typically in learning classifiers) that computes the output probability distribution over a given vocabulary by taking into account the current hidden state, while f is a non-linear activation function used to make the transition between hidden states (e.g. of functions: sigmoid, tanh, ReLU, etc). Thus, the RNN can learn the probability distribution of the next character (x_t) in the vocabulary, given a character sequence as input $< x_1, x_2, ..., x_{t-1} >$, i.e. it can learn the conditional distribution $p(x_t | < x_1, x_2, ..., x_{t-1} >)$.

Sequence-to-sequence model (seq2seq) was introduced in (Cho et al., 2014). It consists of two connected recurrent neural networks: one that acts as an encoder, processing a variable input sequence and producing a fixed dimensional representation, and another one that acts as a decoder by taking the fixed dimensional input sequence representation and generating a variable dimensional output sequence. The decoder network uses the output character at time step t as an input character for time step $t+1$. Thus, it learns a conditional distribution over a sequence of next outputs, i.e. $p(< y_1, ..., y_{T1} > | < x_1, ..., x_{T2} >)$. Figure 2 shows the architecture of the model.

The test data of generative models presented in this paper uses the seq2seq models. The corpus of input files are treated as a sequence of characters, so the model itself contains the distribution of the next character in the vocabulary based on a previously generated context. An *epoch* is defined in the machine learning terminology as a full iteration of the learning algorithm over the entire training database (i.e. input files in the corpus). In the evaluation section, we use different epochs (10, 20, 30, 40 and 50) to correlate the time needed to train versus the quality of the trained model.

3.2 Using the Model to Generate New Inputs

After the seq2seq model is trained, it can be used to generate new inputs based on the probability distribution of next characters and the previously generated context. The work in (Godefroid et al., 2017) always starts with "obj" string and continuously generates characters using different policies to draw the next characters from the model, until the output produced is the string "endobj". These markers are the ones used to represent the beginning and ending of PDF objects. While our tool is capable of dynamically adapting to new / unknown file types or without any expert knowledge, we use a different strategy for defining the beginning / end markers (see the next section for details).

There are four documented policies that can be used when deciding which character a model should output next:

- No sampling : just use the model as it is without randomness ; this will produce deterministic results from any starting point, i.e. the highest probability character will be chosen always.

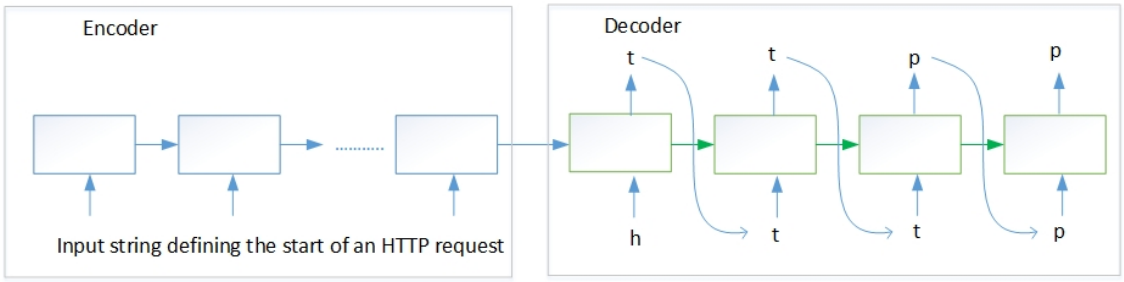

Figure 1: A sequence-to-sequence graphical representation. In this example, the encoder part takes as input the internal string marker representing the beginning of an HTTP request, while the decoder produces the beginning text of such a request.

- Sample: random sampling at each next character according to the probability distribution encapsulated in the model. This strategy produces a diverse set of new inputs combining the patterns learned from data but also mixing with random fuzzing.
- SampleSpace: random sampling only at white spaces. According to the evaluation section, this produces better well-formed new inputs that are not deterministic but that are not as diverse as the Sample model.
- SampleFuzz: A parameter defines the threshold probability for deciding how to choose the next character from the learned model. Then, a random value is drawn at each time step and if it is higher than the threshold, the next character chosen is the one with the highest probability from the learned model. Otherwise, the character with the lowest probability is selected in an attempt to trick the PDF parser. The idea was analysed in (Godefroid et al., 2017). However, in our analysis this shows worse results than Sample and SampleSpace methods.

4 PIPELINE FOR GENERATING NEW TESTS BASED ON EXISTING CORPUS

The tool presented in this paper is open-source and currently available at: https://github.com/AGAPIA/AutomaticTestDataRNN. It receives as input a corpus of different input file types, with no previous classification made manually by the user. The content of the folder can be updated online in both directions: either adding new files of existing types, or adding new file types. This is an important requirement since the main requirements from software security companies (such as the one we collaborated with, Bitdefender) are: (1) to be able to learn and produce new inputs of different kinds for many different programs with the purpose of security evaluation, and (2) to automatically and dynamically collect data from users, i.e. new input tests are added online and used to improve the trained model).

4.1 The Training Pipeline

Given the path to an existing corpus folder (*data*), the training pipeline writes its output in two folders:

(1) *data_preprocesses*

(2) *data_models*

Folder (1) stores the clusterized and preprocessed corpus data. Since the types of the files in there is unknown, our first target is to cluster them by identifying the type of each file in the corpus then put them in a different subfolder corresponding to each file type. As an example, if the corpus folder (*data*) contains three different input file types such as XML, PDF and HTTP requests, then the first step will create (if not already existing) three clusters (folders) and add each input file to the corresponding one. Currently, the classification of files to clusters is done using the $file-l$ command in Unix, and getting the output string of the command (we plan to improve this classification in the future work by using unsupervised learning and perform clusterization based on common identified features). Since at each training epoch the entire sequence of character in each file must be processed, and considering that seek operations on disk can be expensive, the strategy used by our training pipeline is to concatenate together all files in each cluster (folder) in a single file to make the training process faster. Thus, each of the three folders in the concrete example above will contain a single file with the aggregated context from the initial ones. The neural-network model of each cluster is trained by splitting the aggregated file content ($C_{Content}$) in multiple training sequences of a fixed size L, which can be customized by user. Thus, the i^{th} training sequence contains $t_i = C_{Content}[i*L\ :\ (i+1)*L]$ (where F[a : b]

denotes the subsequence of characters in F between indices a and b). For each of these training sequences, the expected output that the network is trained against is the input one shifted by 1 position to the right, i.e, $o_i = C_{Content}[i*L+1 \ : \ (i+1)*L+1]$. The model is then trained with all these input/output sequences from a cluster's content and using backpropagation to correct the weights, it learns the probability map of next characters having a given context (prior sequence). This previous context is modeled with the hidden state layer.

However, we need a generic way to mark the beginning and ending of an individual file content, such that the sampling method knows how to start and when to stop. At this moment, the beginning marker is a string *BEGIN#CLUSTERID*, while the end marker is a string *END#CLUSTERID*, where $CLUSTERID$ is an integer built using a string to integer mapping heuristic. The input string used for mapping is the full classification output string given by the $file \ -l$ command when the file was classified in a cluster. A supervisor map checks if all hashcodes are unique and tries different methods until for each cluster there is a unique identifier. The equation below shows the content of a cluster's aggregated file, where the $\sum$ and $+$ operators acts as concatenation of strings, and C is a given cluster type.

$$Identifier(C) = GetUniqueClusterIdentifier(C)$$
$$Cluster(C) = \sum_{eachfileF \in C} (\text{“}BEGIN'' + Identifier(C) + FileContent(F) + \text{“}END'' + Identifier(C))$$

The tool uses Tensorflow (Abadi et al., 2016) for implementing both learning and sampling processes. Each cluster will have its own generative model, saved in *data_models* folder. In the example given above, three models will be created, one for each XML, PDF and HTTP input types. A mapping from $CLUSTERID$ to the corrensponding model will be created (and stored on disk) to let the sampling process know where to get data from. In the network built using Tensorflow implementation we use LSTM cells for avoiding the problems with exploding or vanishing gradients (Zaremba et al., 2014). By default, the network built has two hidden layers each with 128 hidden states. However, the user can modify this network using expert knowledge per cluster granularity as stated in Section 4.3 (the starting point of the process described in this section is defined in *generateModel.py* script, which has a documented set of parameters as help). Tensorflow is also able to parallelize automatically the training/sampling in a given cluster. On a high-level view, the framework allows users to customize a network and its internal compiler / executer decides where to run tasks with the scope of optimizing performance (e.g. minimize communication time, GPU-CPU memory transfer, etc).

Our tool takes advantage of the checkpointing feature available in the Tensorflow framework, i.e. at any time the learned model up to a point can be saved to disk. This helps users by letting them update the generative models if new files were added dynamically to the clusters after the initial learning step. This way, the learned weights in the neural network are reused and if the new files are not completely different in terms of features from the initial ones, the training time scales proportionally to the size of the new content added. At the implementation level, an indexing service keeps the track of the new content in each cluster and informs a service periodically to start the generative models updating for each of the modified clusters. Another advantage of the checkpoint feature is that it allows users to take advantage of the intermediate trained models. Although not optimal, these can be used in parallel with the training process (until convergence) to generate new test data.

4.2 New Inputs Generation

The pseudocode in the listing below shows the method used to generate a new input test. The function receives as input a cluster type (considering that there exists a trained generative model for the given cluster), and a policy functor pointing to one of the four policies defined in the previous section. The first step is to get the custom parameters and the begin/end marker strings for the given cluster. The next step is to feed the entire begin marker string (starting with a zero set hidden layer) and get the resulted hidden state. This will capture the context learned from the training data at the beginning of the files in that cluster. Then, the code loops producing output characters one by one using the probability distribution map (P) returned by the *FeedForward* function in the current state (h_state). At each iteration, as seen in Figure 2, the last produced output character and state are given as parameters to find the probability distribution map over vocabulary. The loop ends when the last part of the output (suffix) is exactly the end marker string (or until a certain maximum size was produced to avoid blocking if the training was not good enough to get to the end marker). The starting point of the concrete implementation can be found in the script file named *sampleModel.py*.

```
SampleNewTest(Cluster, PolicyType):

  Params = GetParams(Cluster)
```

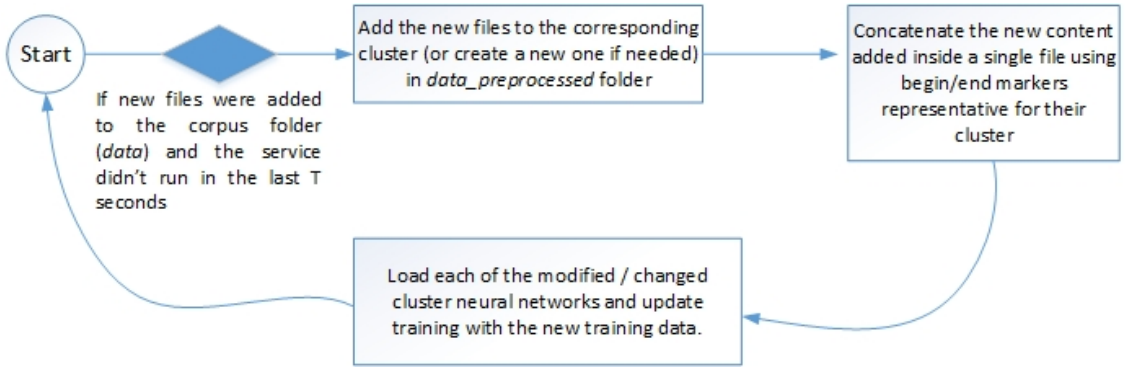

Figure 2: The process of updating the generative models.

```
  BeginMarker, EndMarker = GetMarkers(Params)

  foreach c in BeginMarker:
    h_state, P = FeedForward(h0, internalRNN, c)
    lastChar = c

  output = ""

  while the suffix of output != EndMarker :
    lastChar = Policy(PolicyType, P, lastChar)
    output += lastChar
    h_state, P = FeedForward(h_state,
                      internalRNN, lastChar)
return output
```

A pseudocode defining sampling policies is presented in the listing below. Roulette-wheel based random selection is used with the Sample policy, and with the SampleSpace one when the previous character generated was a whitespace. If SampleSpace is used but still inside a word, or if SampleFuzz sampling method is used and the random value drawn is higher than the fuzz threshold, then the character with the highest probability from the vocabulary is chosen. Instead, if the random value is smaller than fuzz threshold, the character with the lowest probability is chosen in an attempt to trick the program under test.

```
Policy(PolicyType, P, C):
  switch Type:
     case NoSample:
        return argmax(P)
     case SampleSpace:
        if C == " " return roulettewheel(P)
        else return argmax(P)
     case Sample:
        return roulettewheel(P)
     case SampleFuzz:
        if rand < FuzzThreshold:
            return argmin(P)
        else
            return argmax(P)
     default:
        assert "no such policy"
```

4.3 Expert Knowledge

Different clusters might need different parameters for optimal results. For example, training PDF objects might require more time to get to the same loss result than the threshold set for learning HTTP requests. The optimal parameters can differ starting from simple thresholds to the configuration of the neural network structure, i.e. the number of hidden layers or states. The tool allows users to inject their own parameters for both learning and sampling new results, by using a map data structure that looks more like an expert system. If custom data is available in that map (e.g. $[\text{"}HTTP\ request\ cluster\text{"}, num\ hidden\ layers] = 1$) for a particular cluster and parameters, then those are used instead of the default ones. Another example is the customization of the beginning/end markers used to know when a certain input data starts and ends. For well-known types, the user can override our default method for assigning the markers with the correct ones (e.g. PDF objects start with "obj" and end with "endobj"). Also, since Tensorflow can provide graphical statistics added by users (Tensorboard) during both training and sampling, the tool allows users to insert customized logs and graphics per cluster type using the function hooks provided.

5 EVALUATION

As the previous work in the field (Godefroid et al., 2017) already evaluated the training efficiency of the core method, i.e. learning a generative model with RNNs and do inference over it to find new inputs, using PDF file types, we evaluate our tool using two more parser applications: XML parser [1] and HTTP parser [2]. However, we use our own mark system for

[1] http://xmlsoft.org
[2] https://github.com/nodejs/http-parser

beginning/ending of a file, which works for generic (any kind of) file types as mentioned in Section 4. The two new mentioned test applications were used to compare the results directly against the work in (Paduraru et al., 2017), which uses random fuzz testing driven by a genetic algorithm to get better coverage over time, and the same two programs for evaluation.

5.1 Experiment Setup and Methodology

The experiments described below involved a cluster of 8 PCs, each one with 12 physical CPU cores, totaling 96 physical cores of approximately the same performance (Intel Core i7-5930K 3.50 Ghz). Each of the PC had one GPU device, an Nvidia GTX 1070. The user should note that adding more GPUs into the system could improve performance with our tool since the benchmarks show that the GPU device was in average about 15 times faster than the CPU both for learning models and generating new tests.

In our tests, we ultimately care about the coverage metric of a database of input tests: how many branches of a program are evaluated using all the available tests, and how much time did we spend to get to that coverage? Our implementation uses a tool called Tracer that can run a program P against the input test data and produce a trace, i.e., an ordered list of branch instructions $B_0, ..., B_n$ that a program encountered while executing with the given input test: $Tracer(P, test) = B_0 B_1 ... B_n$. Because a program can make calls to other libraries or system executables, each branch is a pair of the module name and offset where the branch instruction occurred: $B_i = (module, offset)$. Note that we divide our program in basic blocks, which are sequences of x86 instructions that contain exactly one branch instruction at its end. We used a tracer tool developed by Bitdefender company, which helped us in the evaluation process, but there are also open-source tracer tools such as Bintrace [3]. Having a set of input test files, we name coverage the set of different instructions (pairs of $(module, offset)$) encountered by Tracer when executing all those tests. We are interested in maximizing the size of this set usually, and/or minimizing the time needed to obtain good coverage.

Specifically, when training generative models, another point of interest is how efficient is the trained model with different setups, i.e. how many newly generated tests are correctly compiled by the HTTP and XML parsers (*Pass Rate* metric) ? This could help us make a correlation between the Pass Rate and coverage metrics.

[3] https://bitbucket.org/mihaila/bintrace

5.2 Training Data and Generation of New Tests

The training set consisted of XML and PDF files that were taken using web-crawling different websites. A total of 12.000 files were randomly selected and stored for each of these two categories. For HTTP requests, we used an internal logger to collect 100.000 of such request. The folder grouping all these inputs is named in our terminology *corpus test set*. A metric to understand how well does the trained model learn is named *Pass Rate*. This estimates (using the output from *grep* tool) the percent of tests (from the generated suite) that are well formatted for the parser under test. As Figure 3 shows, and as expected, the quality of trained model grows with the number of epochs used for training (i.e. the number of full passes over the entire training data set). Randomizing only on spaces (i.e. using SampleSpace) gives better results for Pass Rate metric since more data is used as indicated as being optimal by the trained model. Tensorflow was used for both training and inference, and the hardware system considered was the one described at the beginning of this section.

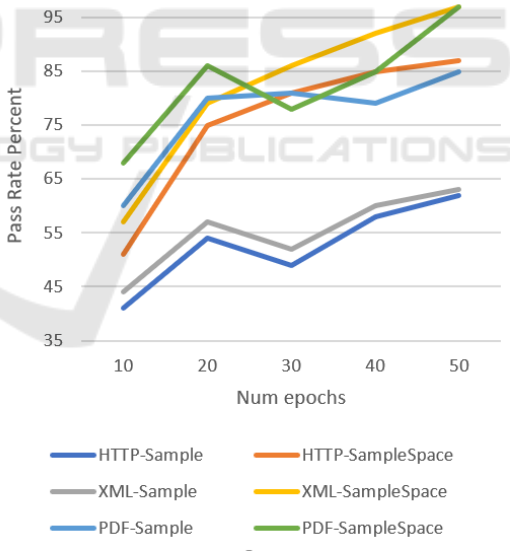

Figure 3: Pass Rate metric evaluation for different number of epochs and models used to generate new tests.

Table 1 shows the time needed to perform model learning over the entire corpus folder of 12.000 PDF and XML files, and 100.000 HTTP requests using a different number of epochs. Other parameters are also important, the user should also take a look at the description of those inside the tool's repository and try to parametrize with expert knowledge for more optimizations when dealing with new file types. Table 2

shows the timings for producing 10.000 new inputs for PDF and XML files, and 50.000 of HTTP requests. As expected, since there is only inference through a learned model, the timings are almost equal between all models (we do not even show the difference between Sample and SampleSpace since the difference is negligible). Actually, from profiling the data tests generation it takes more time to write the output data (i.e. input tests) on disk rather than spending cycles on inference.

Table 1: Time in hours to train models on different number of epochs and using 12.000 files for PDF and XML, and 100.000 HTTP requests as training dataset.

Num epochs	HTTP	XML	PDF objects
50	8h:25	7h:19	9h:11
40	6h:59	5h:56	8h:04
30	5h:35	4h:20	6h:15
20	3h:48	3h:42	4h:17
10	2h:10	1h:12	3h:02

Table 2: The average time needed to produce 10.000 new inputs for PDF and XML files, and 50.000 new HTTP requests.

File type	Time in minutes
XML	49
HTTP	25
PDF	51

Main Takeaway: The time needed to train the model is fixed, depending on the number of epochs and a few other parameters. After the training phase, the tool can create huge databases of new inputs (valid ones) quickly, which in the end can provide better code coverage than existing fuzzing methods. Those do not need the training phase, but the new tests generated are often rejected from early tests inside the program because of their incorrect format.

5.3 Coverage Evaluation

For the coverage evaluation tables below, we considered only the model trained with 30 epochs, which was the winner in terms of performance versus training cost. Using 40 or 50 epochs increased just with a few new lines the coverage over time, but the training time is significantly higher. Of course, the user should experiment and find the optimal number of epochs depending on training data size for example, and their budget time limit allocated for training.

Tables 3 and 4 show the coverage for XML and HTTP file types by using three different evaluation methods. The first one, XML-fuzz+genetic / HTTP - fuzz + genetic, considers the fuzzing method driven by genetic algorithms as explained in (Paduraru et al., 2017). The Sample and SampleSpace are the two models used for sampling defined above in this paper, and which uses our tool. The main observation is that with simple fuzzing (i.e. no use of generative models), the coverage value converges quickly to a value, without necessarily growing by having more time allocated. This happens mainly because the random fuzzing methods produce many times inputs that are not correct, being rejected by early outs, or difficult to deviate from a few common branches inside a program even when adopting different policies to guide fuzzing ((Paduraru et al., 2017), (Godefroid et al., 2017)). However, fuzzing without learning the input context techniques have their own advantage: they are simple to implement and require no training time. For instance, if smoke tests (Kaner et al., 2001) are needed after changing the user application's source code and input grammar, quick random fuzzing methods are very efficient since they do not require any training time. Learning a generative model is not feasible in this situation due to the limited time needed to respond to the new code change. Actually, techniques can be combined: classic fuzzing can be used for smoke tests, while fuzzing with generative models such as the one presented in this paper can be used to perform longer and more performant tests.

Table 3: The number of branch instructions touched in comparison between random fuzzing driven by genetic algorithms, Sample and SampleSpace models for XML files.

Model	9h	15h	24h	72h
XML-fuzz+genetic	1271	1279	1285	1286
XML-Sample	1290	1364	1455	1549
XML-SampleSpace	1291	1375	1407	1553

Table 4: The number of branch instructions touched in comparison between random fuzzing driven by genetic algorithms, Sample and SampleSpace models for HTTP requests.

Model	9h	15h	24h	72h
HTTP-fuzz+genetic	229	230	230	232
HTTP-Sample	238	249	257	271
HTTP-SampleSpace	241	245	269	279

In 72 hours using the system described in the setup, the system was able to get approximately 20% more coverage than the best documented model on the XML and HTTP cases. Also, please note again that the two models evaluated were chosen to compare against other documented results. Our tool is able to produce generative models and training tests after training on any kind of user inputs formats (e.g. HTML, DOC, XLS, source code for different programming languages, etc). An interesting aspect is

that the Sample method has better results than SampleSpace one, although the Pass Rate metric shows inverse results. Remember that by sampling each character according to the probability distribution in the generative model, it has a higher rate of making inputs incorrect (Figure 3). One possible explanation for this is that having a high rate of correct inputs can make the program avoid some instructions that were verifying the code's correctness in more detail. Thus, those instructions might be encountered by Tracer only when the inputs given are a mix between correct and (slightly) invalid. In (Godefroid et al., 2017) there is also a discussion about performing random fuzzing over the inputs learned using RNN methods, but similar to our evaluation, the results are not better than the Sample method. The other technique presented in (Höschele and Zeller, 2016) that learns the grammar of the input through dynamic tainting and applicable currently only to Java programs, could not be evaluated since the tool is not (yet) open-source and could not be retrieved in any other way.

6 CONCLUSIONS AND FUTURE WORK

This paper presented an open-source tool that is able to assist users in automatic generation of test data for evaluating programs, having as initial input a corpus of example tests. Support for any kind of input file formats, operating efficiently in distributed environments, online learning, and checkpoints are one of its strongest features. The evaluation section shows the efficiency of using recurrent neural networks to learn generative models that are able to produce new tests, from two main perspectives: improved instruction coverage over random fuzzing and the percent of correct input files produced from the learned model. As future work, we plan to improve the clusterization of files using autoencoders techniques that are able to learn features from existing inputs, study the effectiveness of using Generative adversarial networks (GANs) in improving tests coverage. Another topic is to improve the usability of the tool by providing a visual interface for controlling parameters and injecting expert knowledge in learning and generation processes in an easier way.

ACKNOWLEDGMENTS

This work was supported by a grant of Romanian Ministry of Research and Innovation CCCDI-UEFISCDI. project no. 17PCCDI/2018 We would like to thank our colleagues Teodor Stoenescu and Alexandra Sandulescu from Bitdefender, and to Alin Stefanescu from University of Bucharest for fruitful discussions and collaboration.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

Bastani, O., Sharma, R., Aiken, A., and Liang, P. (2017). Synthesizing program input grammars. *SIGPLAN Not.*, 52(6):95–110.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

Chorowski, J., Bahdanau, D., Serdyuk, D., Cho, K., and Bengio, Y. (2015). Attention-based models for speech recognition. *CoRR*, abs/1506.07503.

Coppit, D. and Lian, J. (2005). Yagg: An easy-to-use generator for structured test inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 356–359, New York, NY, USA. ACM.

Cui, W., Peinado, M., Chen, K., Wang, H. J., and Irun-Briz, L. (2008). Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 391–402, New York, NY, USA. ACM.

Godefroid, P. (2007). Random testing for security: blackbox vs. whitebox fuzzing. In *RT '07*.

Godefroid, P., Kiezun, A., and Levin, M. Y. (2008). Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA. ACM.

Godefroid, P., Levin, M. Y., and Molnar, D. (2012). Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: machine learning for input fuzzing. In Rosu, G., Penta, M. D., and Nguyen, T. N., editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 50–59. IEEE Computer Society.

Hanford, K. V. (1970). Automatic generation of test cases. *IBM Syst. J.*, 9(4):242–257.

Höschele, M. and Zeller, A. (2016). Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA. ACM.

Kaner, C., Bach, J., and Pettichord, B. (2001). *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

Lämmel, R. and Schulte, W. (2006). Controllable combinatorial coverage in grammar-based testing. In Uyar, M. Ü., Duale, A. Y., and Fecko, M. A., editors, *Testing of Communicating Systems*, pages 19–38, Berlin, Heidelberg. Springer Berlin Heidelberg.

Majumdar, R. and Xu, R.-G. (2007). Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, New York, NY, USA. ACM.

Paduraru, C., Melemciuc, M., and Stefanescu, A. (2017). A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In Bosman, P. A. N., editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1857–1863. ACM.

Purdom, P. (1972). A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375.

Rajpal, M., Blum, W., and Singh, R. (2017). Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR*, abs/1711.04596.

Sirer, E. G. and Bershad, B. N. (1999). Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *CoRR*, abs/1409.2329.