Learn&Fuzz:
Machine Learning for Input Fuzzing

Patrice Godefroid[1], Hila Peleg[2][*], and Rishabh Singh[1]

[1] Microsoft Research
{pg,risin}@microsoft.com
[2] The Technion
hilap@cs.technion.ac.il

Abstract. Fuzzing consists of repeatedly testing an application with modified, or fuzzed, inputs with the goal of finding security vulnerabilities in input-parsing code. In this paper, we show how to automate the generation of an input grammar suitable for input fuzzing using sample inputs and neural-network-based statistical machine-learning techniques. We present a detailed case study with a complex input format, namely PDF, and a large complex security-critical parser for this format, namely, the PDF parser embedded in Microsoft's new Edge browser. We discuss (and measure) the tension between conflicting learning and fuzzing goals: learning wants to capture the structure of well-formed inputs, while fuzzing wants to break that structure in order to cover unexpected code paths and find bugs. We also present a new algorithm for this learn&fuzz challenge which uses a learnt input probability distribution to intelligently guide where to fuzz inputs.

1 Introduction

Fuzzing is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or *fuzzed*, inputs. There are three main types of fuzzing techniques in use today: (1) *blackbox random* fuzzing [27], (2) *whitebox constraint-based* fuzzing [12], and (3) *grammar-based* fuzzing [23, 27], which can be viewed as a variant of model-based testing [28]. Blackbox and whitebox fuzzing are fully automatic, and have historically proved to be very effective at finding security vulnerabilities in binary-format file parsers. In contrast, grammar-based fuzzing is not fully automatic: it requires an input grammar specifying the input format of the application under test. This grammar is typically written by hand, and this process is laborious, time consuming, and error-prone. Nevertheless, grammar-based fuzzing is the most effective fuzzing technique known today for fuzzing applications with complex structured input formats, like web-browsers which must take as (untrusted) inputs web-pages including complex HTML documents and JavaScript code.

In this paper, we consider the problem of *automatically* generating input grammars for grammar-based fuzzing by using machine-learning techniques and

---

[*] The work of this author was done mostly while visiting Microsoft Research.

```
                              xref
      2 0 obj                 0 6                              trailer
      <<                      0000000000 65535 f               <<
      /Type /Pages            0000000010 00000 n               /Size 18
      /Kids [ 3 0 R ]         0000000059 00000 n               /Info 17 0 R
      /Count 1                0000000118 00000 n               /Root 1 0 R
      >>                      0000000296 00000 n               >>
      endobj                  0000000377 00000 n               startxref
                              0000000395 00000 n               3661
           (a)                        (b)                           (c)
```

Fig. 1. Excerpts of a well-formed PDF document. (a) is a sample object, (b) is a cross-reference table with one subsection, and (c) is a trailer.

sample inputs. Previous attempts have used variants of traditional automata and context-free-grammar learning algorithms (see Section 5). In contrast with prior work, this paper presents the *first attempt* at using *neural-network-based statistical learning techniques* for this problem. Specifically, we use *recurrent neural networks* for learning a statistical input model that is also *generative*: it can be used to generate new inputs based on the probability distribution of the learnt model (see Section 3 for an introduction to these learning techniques). We use unsupervised learning, and our approach is fully automatic and does not require any format-specific customization.

We present an in-depth case study for a very complex input format: PDF. This format is so complex (see Section 2) that it is described in a 1,300-pages (PDF) document [1]. We consider a large, complex and security-critical parser for this format: the PDF parser embedded in Microsoft's new Edge browser. Through a series of detailed experiments (see Section 4), we discuss the *learn&fuzz challenge*: how to learn and then generate diverse well-formed inputs in order to maximize parser-code coverage, while still injecting enough ill-formed input parts in order to exercise unexpected code paths and error-handling code.

We also present a novel *learn&fuzz* algorithm (in Section 3) which uses a learnt input probability distribution to intelligently guide *where* to fuzz (statistically well-formed) inputs. We show that this new algorithm can outperform the other learning-based and random fuzzing algorithms considered in this work.

The paper is organized as follows. Section 2 presents an overview of the PDF format, and the specific scope of this work. Section 3 gives a brief introduction to neural-network-based learning, and discusses how to use and adapt such techniques for the learn&fuzz problem. Section 4 presents results of several learning and fuzzing experiments with the Edge PDF parser. Related work is discussed in Section 5. We conclude and discuss directions for future work in Section 6.

```
125 0 obj          88 0 obj            75 0 obj
[680.6 680.6]      (Related Work)      4171
endobj             endobj              endobj
     (a)                  (b)               (c)


47 1 obj
[false 170 85.5 (Hello) /My#20Name]
endobj
                        (d)
```

Fig. 2. PDF data objects of different types.

2 The Structure of PDF Documents

The full specification of the PDF format is over $1,300$ pages long [1]. Most of this specification – roughly 70% – deals with the description of *data objects* and their relationships between parts of a PDF document.

PDF files are encoded in a textual format, which may contain binary information streams (e.g., images, encrypted data). A PDF document is a sequence of at least one PDF body. A PDF body is composed of three sections: objects, cross-reference table, and trailer.

Objects. The data and metadata in a PDF document is organized in basic units called objects. Objects are all similarly formatted, as seen in Figure 1(a), and have a joint outer structure. The first line of the object is its identifier, for indirect references, its generation number, which is incremented if the object is overridden with a newer version, and "`obj`" which indicates the start of an object. The "`endobj`" indicator closes the object.

The object in Figure 1(a) contains a dictionary structure, which is delimited by "`<<`" and "`>>`", and contains keys that begin with `/` followed by their values. `[ 3 0 R ]` is a cross-object reference to an object in the same document with the identifier 3 and the generation number 0. Since a document can be very large, a referenced object is accessed using random-access via a cross-reference table.

Other examples of objects are shown in Figure 2. The object in Figure 2(a) has the content `[680.6 680.6]`, which is an *array object*. Its purpose is to hold coordinates referenced by another object. Figure 2(b) is a string literal that holds the bookmark text for a PDF document section. Figure 2(c) is a numeric object. Figure 2(d) is an object containing a multi-type array. These are all examples of object types that are both used on their own and as the basic blocks from which other objects are composed (e.g., the dictionary object in Figure 1(a) contains an array). The rules for defining and composing objects comprises the majority of the PDF-format specification.

Cross reference table. The cross reference tables of a PDF body contain the address in bytes of referenced objects within the document. Figure 1(b) shows a cross-reference table with a subsection that contains the addresses for five

objects with identifiers 1-5 and the placeholder for identifier 0 which never refers to an object. The object being pointed to is determined by the row of the table (the subsection will include 6 objects starting with identifier 0) where `n` is an indicator for an object in use, where the first column is the address of the object in the file, and `f` is an object not used, where the first column refers to the identifier of the previous free object, or in the case of object 0 to object 65535, the last available object ID, closing the circle.

Trailer. The trailer of a PDF body contains a dictionary (again contained within "`<<`" and "`>>`") of information about the body, and `startxref` which is the address of the cross-reference table. This allows the body to be parsed from the end, reading `startxref`, then skipping back to the cross-reference table and parsing it, and only parsing objects as they are needed.

Updating a document. PDF documents can be *updated incrementally*. This means that if a PDF writer wishes to update the data in object 12, it will start a new PDF body, in it write the new object with identifier 12, and a generation number greater than the one that appeared before. It will then write a new cross-reference table pointing to the new object, and append this body to the previous document. Similarly, an object will be deleted by creating a new cross-reference table and marking it as free. We use this method in order to append new objects in a PDF file, as discussed later in Section 4.

Scope of this work. In this paper, we investigate how to leverage and adapt neural-network-based learning techniques to learn a grammar for *non-binary PDF data objects*. Such data objects are formatted text, such as shown in Figure 1(a) and Figure 2. Rules for defining and composing such data objects makes the bulk of the 1,300-pages PDF-format specification. These rules are numerous and tedious, but repetitive and structured, and therefore well-suited for learning with neural networks (as we will show later). In contrast, learning automatically the structure (rules) for defining cross-reference tables and trailers, which involve constraints on lists, addresses, pointers and counters, look too complex and less promising for learning with neural networks. We also do not consider binary data objects, which are encoded in binary (e.g., image) sub-formats and for which fully-automatic blackbox and whitebox fuzzing are already effective.

3   Statistical Learning of Object Contents

We now describe our statistical learning approach for learning a generative model of PDF objects. The main idea is to learn a generative language model over the set of PDF object characters given a large corpus of objects. We use a sequence-to-sequence (seq2seq) [5, 26] network model that has been shown to produce state-of-the-art results for many different learning tasks such as machine translation [26] and speech recognition [6]. The seq2seq model allows for learning arbitrary length contexts to predict next sequence of characters as compared

to traditional n-gram based approaches that are limited by contexts of finite length. Given a corpus of PDF objects, the seq2seq model can be trained in an unsupervised manner to learn a generative model to generate new PDF objects using a set of input and output sequences. The input sequences correspond to sequences of characters in PDF objects and the corresponding output sequences are obtained by shifting the input sequences by one position. The learnt model can then be used to generate new sequences (PDF objects) by sampling the distribution given a starting prefix (such as "`obj`").

3.1 Sequence-to-Sequence Neural Network Models

A recurrent neural network (RNN) is a neural network that operates on a variable length input sequence $\langle x_1, x_2, \cdots, x_T \rangle$ and consists of a hidden state h and an output y. The RNN processes the input sequence in a series of time stamps (one for each element in the sequence). For a given time stamp t, the hidden state h_t at that time stamp and the output y_t is computed as:

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

$$y_t = \phi(h_t)$$

where f is a non-linear activation function such as sigmoid, tanh etc. and ϕ is a function such as `softmax` that computes the output probability distribution over a given vocabulary conditioned on the current hidden state. RNNs can learn a probability distribution over a character sequence $\langle x_1, \cdots, x_{t-1} \rangle$ by training to predict the next character x_t in the sequence, i.e., it can learn the conditional distribution $p(x_t | \langle x_1, \cdots, x_{t-1} \rangle)$.

Cho et al. [5] introduced a sequence-to-sequence (seq2seq) model that consists of two recurrent neural networks, an encoder RNN that processes a variable dimensional input sequence to a fixed dimensional representation, and a decoder RNN that takes the fixed dimensional input sequence representation and generates the variable dimensional output sequence. The decoder network generates output sequences by using the predicted output character generated at time step t as the input character for timestep $t+1$. An illustration of the `seq2seq` architecture is shown in Figure. 3. This architecture allows us to learn a conditional distribution over a sequence of next outputs, i.e., $p(\langle y_1, \cdots, y_{T_1} \rangle | \langle x_1, \cdots, x_{T_2} \rangle)$.

We train the seq2seq model using a corpus of PDF objects treating each one of them as a sequence of characters. During training, we first concatenate all the object files s_i into a single file resulting in a large sequence of characters $\tilde{s} = s_1 + \cdots + s_n$. We then split the sequence into multiple training sequences of a fixed size d, such that the ith training instance $t_i = \tilde{s}[i * d : (i + 1) * d]$, where $s[k : l]$ denotes the subsequence of s between indices k and l. The output sequence for each training sequence is the input sequence shifted by 1 position, i.e., $o_t = \tilde{s}[i * d + 1 : (i + 1) * d + 1]$. The seq2seq model is then trained end-to-end to learn a generative model over the set of all training instances.
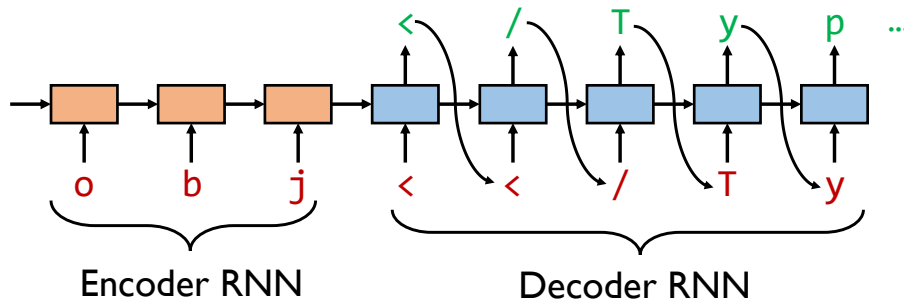
Fig. 3. A sequence-to-sequence RNN model to generate PDF objects.

3.2 Generating new PDF objects

We use the learnt seq2seq model to generate new PDF objects. There are many different strategies for object generation depending upon the sampling strategy used to sample the learnt distribution. We always start with a prefix of the sequence "`obj `" (denoting the start of an object instance), and then query the model to generate a sequence of output characters until it produces "`endobj`" corresponding to the end of the object instance. We now describe three different sampling strategies we employ for generating new object instances.

`NoSample`: In this generation strategy, we use the learnt distribution to greedily predict the best character given a prefix. This strategy results in generating PDF objects that are most likely to be well-formed and consistent, but it also limits the number of objects that can be generated. Given a prefix like "`obj`", the best sequence of next characters is uniquely determined and therefore this strategy results in the same PDF object. This limitation precludes this strategy from being useful for fuzzing.

`Sample`: In this generation strategy, we use the learnt distribution to *sample* next characters (instead of selecting the top predicted character) in the sequence given a prefix sequence. This sampling strategy is able to generate a diverse set of new PDF objects by combining various patterns the model has learnt from the diverse set of objects in the training corpus. Because of sampling, the generated PDF objects are not always guaranteed to be well-formed, which is useful from the fuzzing perspective.

`SampleSpace`: This sampling strategy is a combination of `Sample` and `NoSample` strategies. It samples the distribution to generate the next character only when the current prefix sequence ends with a whitespace, whereas it uses the best character from the distribution in middle of tokens (i.e., prefixes ending with non-whitespace characters), similar to the `NoSample` strategy. This strategy is expected to generate more well-formed PDF objects compared to the `Sample` strategy as the sampling is restricted to only at the end of whitespace characters.

Algorithm 1 $\text{SampleFuzz}(\mathcal{D}(\mathbf{x},\theta), t_{\texttt{fuzz}}, p_t)$

  $\texttt{seq} := \text{"obj "}$
  while $\neg$ $\texttt{seq.endswith}(\text{"endobj"})$ **do**
    $\text{c,p(c)} := \texttt{sample}(\mathcal{D}(\texttt{seq},\theta))$ (* Sample c from the learnt distribution *)
    $p_{\texttt{fuzz}} := \texttt{random}(0,1)$ (* random variable to decide whether to fuzz *)
    if $p_{\texttt{fuzz}} > t_{\texttt{fuzz}} \wedge p(c) > p_t$ **then**
      $\text{c} := \text{argmin}_{c'}\{p(c') \sim \mathcal{D}(\texttt{seq},\theta)\}$ (* replace c by c' (with lowest likelihood) *)
    end if
    $\texttt{seq} := \texttt{seq} + \text{c}$
    if $\texttt{len(seq)} > \texttt{MAXLEN}$ **then**
      $\texttt{seq} := \text{"obj "}$ (* Reset the sequence *)
    end if
  end while
  return $\texttt{seq}$

3.3 SampleFuzz: Sampling with Fuzzing

Our goal of learning a generative model of PDF objects is ultimately to perform fuzzing. A perfect learning technique would always generate well-formed objects that would not exercise any error-hanlding code, whereas a bad learning technique would result in ill-formed objects that woult be quickly rejected by the parser upfront. To explore this tradeoff, we present a new algorithm, dubbed `SampleFuzz`, to perform some fuzzing while sampling new objects. We use the learnt model to generate new PDF object instances, but at the same time introduce anomalies to exercise error-handling code.

The `SampleFuzz` algorithm is shown in Algorithm 1. It takes as input the learnt distribution $\mathcal{D}(\mathbf{x},\theta)$, the probability of fuzzing a character $t_{\texttt{fuzz}}$, and a threshold probability p_t that is used to decide whether to modify the predicted character. While generating the output sequence $\texttt{seq}$, the algorithm samples the learnt model to get some next character c and its probability $p(c)$ at a particular timestamp t. If the probability $p(c)$ is higher than a user-provided threshold p_t, i.e., if the model is confident that c is likely the next character in the sequence, the algorithm chooses to instead sample another different character c' in its place where c' has the minimum probability $p(c')$ in the learnt distribution. This modification (fuzzing) takes place only if the result $p_{\texttt{fuzz}}$ of a random coin toss returns a probability higher than input parameter $t_{\texttt{fuzz}}$, which lets the user further control the probability of fuzzing characters. The key intuition of the `SampleFuzz` algorithm is to introduce unexpected characters in objects only in places where the model is *highly confident*, in order to trick the PDF parser. The algorithm also ensures that the object length is bounded by `MAXLEN`. Note that the algorithm is not guaranteed to always terminate, but we observe that it always terminates in practice.

3.4 Training the Model

Since we train the seq2seq model in an unsupervised learning setting, we do not have test labels to explicitly determine how well the learnt models are performing. We instead train multiple models parameterized by number of passes, called *epochs*, that the learning algorithm performs over the training dataset. An *epoch* is thus defined as an iteration of the learning algorithm to go over the complete training dataset. We evaluate the seq2seq models trained for five different numbers of epochs: 10, 20, 30, 40, and 50. In our setting, one epoch takes about 12 minutes to train the seq2seq model, and the model with 50 epochs takes about 10 hours to learn. We use an LSTM model [15] (a variant of RNN) with 2 hidden layers, where each layer consists of 128 hidden states.

4 Experimental Evaluation

4.1 Experiment Setup

In this section, we present results of various fuzzing experiments with the PDF viewer included in Microsoft's new Edge browser. We used a self-contained single-process test-driver executable provided by Microsoft for testing/fuzzing purposes. This executable takes a PDF file as input argument, executes the PDF parser included in the Microsoft Edge browser, and then stops. If the executable detects any parsing error due to the PDF input file being malformed, it prints an error message in an execution log. In what follows, we simply refer to it as the *Edge PDF parser*. All experiments were performed on 4-core 64-bit Windows 10 VMs with 20Gb of RAM.

We use three main standard metrics to measure fuzzing effectiveness:

Coverage. For each test execution, we measure instruction coverage, that is, the set of all unique instructions executed during that test. Each instruction is uniquely identified by a pair of values `dll-name` and `dll-offset`. The coverage for a set of tests is simply the union of the coverage sets of each individual test.

Pass rate. For each test execution, we programmatically check (`grep`) for the presence of parsing-error messages in the PDF-parser execution log. If there are no error messages, we call this test *pass* otherwise we call it *fail*. Pass tests corresponds to PDF files that are considered to be well-formed by the Edge PDF parser. This metric is less important for fuzzing purposes, but it will help us estimate the quality of the learning.

Bugs. Each test execution is performed under the monitoring of the tool AppVerifier, a free runtime monitoring tool that can catch memory corruptions bugs (such as buffer overflows) with a low runtime overhead (typically a few percent runtime overhead) and that is widely used for fuzzing on Windows.

4.2 Training Data

We extracted about 63,000 non-binary PDF objects out of a diverse set of 534 PDF files. This set of 534 files was itself the result of *seed minimization*, that is, the process of computing a subset of a larger set of input files which provides the same instruction coverage as the larger set. Seed minimization is a standard first step applied before file fuzzing [27, 12]. The larger set of PDF files came from various sources, like past PDF files used for fuzzing but also other PDF files collected from the public web.

These 63,000 non-binary objects are the training set for the RNNs we used in this work. Binary objects embedded in PDF files (typically representing images in various image formats) were not considered in this work.

We learn, generate, and fuzz PDF objects, but the Edge PDF parser processes full PDF files, not single objects. Therefore we wrote a simple program to correctly *append* a new PDF object to an existing (well-formed) PDF file, which we call a *host*, following the procedure discussed in Section 2 for updating a PDF document. Specifically, this program first identifies the last trailer in the PDF host file. This provides information about the file, such as addresses of objects and the cross-reference table, and the last used object ID. Next, a new body section is added to the file. In it, the new object is included with an object ID that overrides the last object in the host file. A new cross reference table is appended, which increases the generation number of the overridden object. Finally, a new trailer is appended.
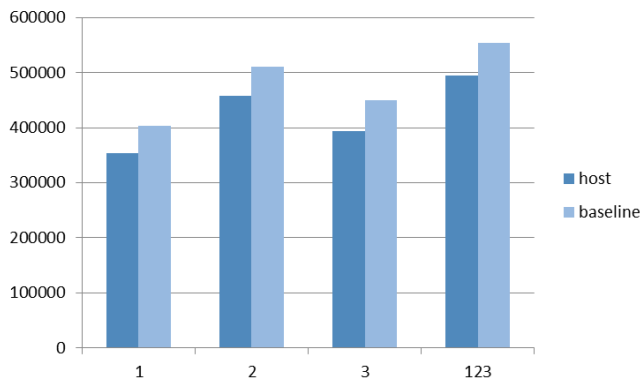
4.3 Baseline Coverage

To allow for a meaningful interpretation of coverage results, we randomly selected 1,000 PDF objects out of our 63,000 training objects, and we measured their coverage of the Edge PDF parser, to be used as a baseline for later experiments.

A first question is which host PDF file should we use in our experiments: since any PDF file will have some objects in it, will a new appended object interfere with other objects already present in the host, and hence influence the overall coverage and pass rate?

To study this question, we selected the smallest three PDF files in our set of 534 files, and used those as hosts. These three hosts are of size 26Kb, 33Kb and 16Kb respectively.

Figure 4 shows the instruction coverage obtained by running the Edge PDF parser on the three hosts, denoted `host1`, `host2`, and `host3`. It also show the coverage obtained by computing the union of these three sets, denoted `host123`. Coverage ranges from 353,327 (`host1`) to 457,464 (`host2`) unique instructions, while the union (`host123`) is 494,652 and larger than all three – each host covers some unique instructions not covered by the other two. Note that the smallest file `host3` does not lead to the smallest coverage.

Next, we recombined each of our 1,000 baseline objects with each of our three hosts, to obtain three sets of 1,000 new PDF files, denoted `baseline1`, `baseline2` and `baseline3`, respectively. Figure 4 shows the coverage of each set, as well as their union `baseline123`. We observe the following.

Fig. 4. Coverage for PDF hosts and baselines.

- The baseline coverage varies depending on the host, but is larger than the host alone (as expected). The largest difference between a host and a baseline coverage is 59,221 instruction for `host123` out of 553,873 instruction for `baseline123`. In other words, 90% of all instructions are included in the host coverage no matter what new objects are appended.
- Each test typically covers on the order of half a million unique instructions; this confirms that the Edge PDF parser is a large and non-trivial application.
- 1,000 PDF files take about 90 minutes to be processed (both to be tested and get the coverage data).

We also measured the pass rate for each experiment. As expected, the pass rate is 100% for all 3 hosts.

Main Takeaway: Even though coverage varies across hosts because objects may interact differently with each host, the re-combined PDF file is always perceived as well-formed by the Edge PDF parser.

4.4 Learning PDF Objects

When training the RNN, an important parameter is the number of epochs being used (see Section 3). We report here results of experiments obtained after training the RNN for 10, 20, 30, 40, and 50 epochs, respectively. After training, we used each learnt RNN model to generate 1,000 unique PDF objects. We also compared the generated objects with the 63,000 objects used for training the model, and found no exact matches.

As explained earlier in Section 3, we consider two main RNN generation modes: the `Sample` mode where we sample the distribution at every character position, and the `SampleSpace` mode where we sample the distribution only after whitespaces and generate the top predicted character for other positions.

The pass rate for `Sample` and `SampleSpace` when training with 10 to 50 epochs is reported in Figure 5. We observe the following:
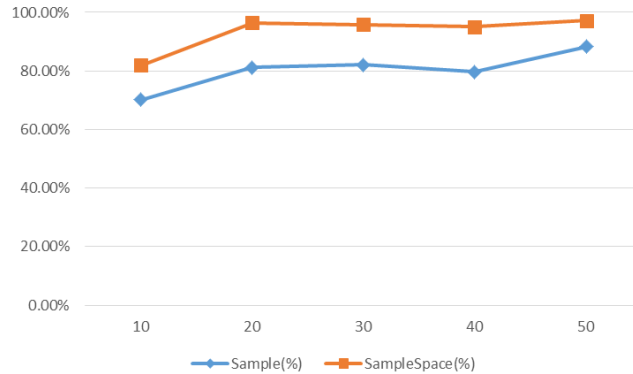
Fig. 5. Pass rate for `Sample` and `SampleSpace` from 10 to 50 epochs.

– The pass rate for `SampleSpace` is consistently better than the one for `Sample`.
– For 10 epochs only, the pass rate for `Sample` is already above 70%. This means that the learning is of good quality.
– As the number of epochs increases, the pass rate increases, as expected, since the learned models become more precise but they also take more time (see Section 3).
– The best pass rate is 97% obtained with `SampleSpace` and 50 epochs.

Interestingly, the pass rate is essentially the same regardless of the host PDF file being used: it varies by at most 0.1% across hosts (data not shown here).

Main Takeaway: The pass rate ranges between 70% and 97% and shows the learning is of good quality.

4.5 Coverage with Learned PDF Objects

Figure 6 shows the instruction coverage obtained with `Sample` and `SampleSpace` from 10 to 50 epochs and using `host1` (top left), `host2` (top right), `host3` (bottom left), and the overall coverage for all hosts `host123` (bottom right). The figure also shows the coverage obtained with the corresponding `baseline`. We observe the following:

– Unlike for the pass rate, the host impacts coverage significantly, as already pointed out earlier. Moreover, the shapes of each line vary across hosts.
– For `host1` and `host2`, the coverage for `Sample` and `SampleSpace` are above the `baseline` coverage for most epoch results, while they are mostly below the `baseline` coverage for `host3` and `host123`.
– The best overall coverage is obtained with `Sample` 40-epochs (see the `host123` data at the bottom right).
– The `baseline123` coverage is overall second best behind `Sample` 40-epochs.
– The best coverage obtained with `SampleSpace` is also with 40-epochs.

Main Takeaway: The best overall coverage is obtained with `Sample` 40-epochs.
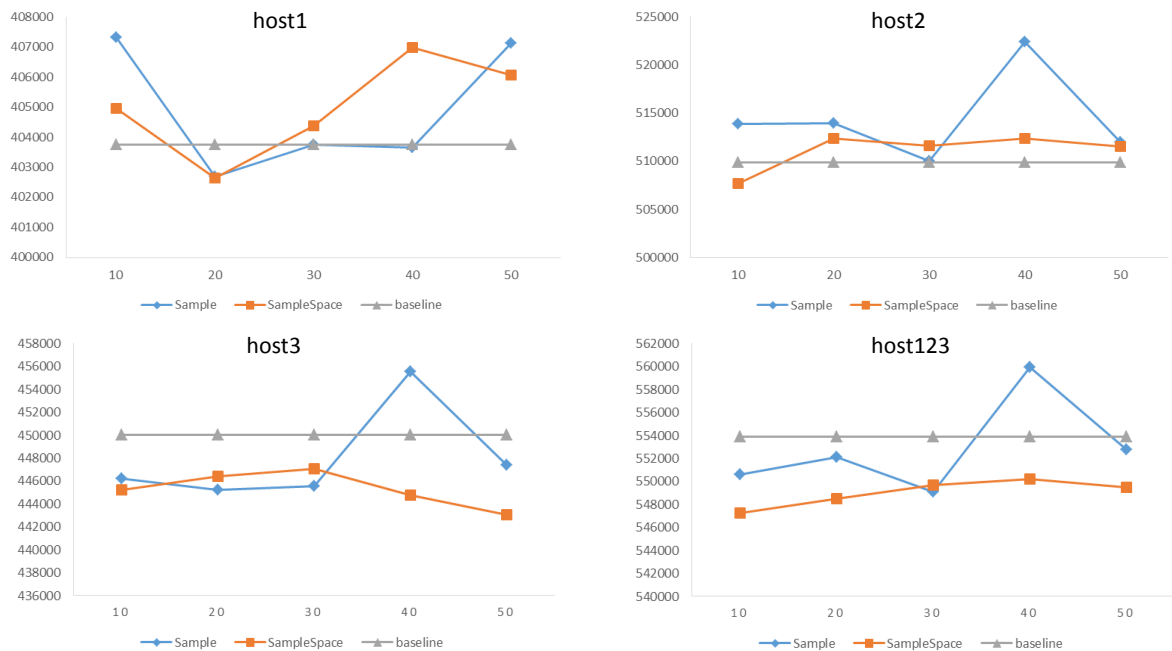
Fig. 6. Coverage for `Sample` and `SampleSpace` from 10 to 50 epochs, for `host 1, 2, 3,` and `123`.

4.6 Comparing Coverage Sets

So far, we simply counted the number of unique instructions being covered. We now drill down into the overall `host123` coverage data of Figure 6, and compute the overlap between overall coverage sets obtained with our 40-epochs winner `Sample`-40e and `SampleSpace`-40e, as well as the `baseline123` and `host123` overall coverage. The results are presented in Figure 7. We observe the following:

- All sets are almost supersets of `host123` as expected (see the `host123` row), except for a few hundred instructions each.
- `Sample`-40e is almost a superset of all other sets, except for 1,680 instructions compared to `SampleSpace`-40e, and a few hundreds instructions compared to `baseline123` and `host123` (see the `Sample`-40e column).
- `Sample`-40e and `SampleSpace`-40e have way more instructions in common than they differ (10,799 and 1,680), with `Sample`-40e having better coverage than `SampleSpace`-40e.
- `SampleSpace`-40e is incomparable with `baseline123`: it has 3,393 more instructions but also 6,514 missing instructions.

Main Takeaway: Our coverage winner `Sample`-40e is almost a superset of all other coverage sets.

Row\Column	Sample-40e	SampleSpace-40e	baseline123	host123
Sample-40e	0	10,799	6,658	65,442
SampleSpace-40e	1,680	0	3,393	56,323
baseline123	660	6,514	0	59,444
host123	188	781	223	0

Fig. 7. Comparing coverage: unique instructions in each row compared to each column.

Algorithm	Coverage	Pass Rate
SampleSpace+Random	563,930	36.97%
baseline+Random	564,195	44.05%
Sample-10K	565,590	78.92%
Sample+Random	566,964	41.81%
SampleFuzz	567,634	68.24%

Fig. 8. Results of fuzzing experiments with 30,000 PDF files each.

4.7 Combining Learning and Fuzzing

In this section, we consider several ways to combine learning with fuzzing, and evaluate their effectiveness.

We consider a widely-used simple blackbox random fuzzing algorithm, denoted `Random`, which randomly picks a position in a file and then replaces the byte value by a random value between 0 and 255. The algorithm uses a *fuzz-factor* of 100: the length of the file divided by 100 is the average number of bytes that are fuzzed in that file.

We use `random` to generate 10 variants of every PDF object generated by 40-epochs `Sample`-40e, `SampleSpace`-40e, and `baseline`. The resulting fuzzed objects are re-combined with our 3 host files, to obtain three sets of 30,000 new PDF files, denoted by `Sample+Random`, `SampleSpace+Random` and `baseline+Random`, respectively.

For comparison purposes, we also include the results of running `Sample`-40e to generate 10,000 objects, denoted `Sample`-10K.

Finally, we consider our new algorithm `SampleFuzz` described in Section 3, which decides where to fuzz values based on the learnt distribution. We applied this algorithm with the learnt distribution of the 40-epochs RNN model, $t_{\texttt{fuzz}} = 0.9$, and a threshold $p_t = 0.9$.

Figure 8 reports the overall coverage and the pass-rate for each set. Each set of 30,000 PDF files takes about 45 hours to be processed. The rows are sorted by increasing coverage. We observe the following:

- After applying `Random` on objects generated with `Sample`, `SampleSpace` and `baseline`, coverage goes up while the pass rate goes down: it is consistently below 50%.
- After analyzing the overlap among coverage sets (data not shown here), all fuzzed sets are almost supersets of their original non-fuzzed sets (as expected).

- Coverage for `Sample`-10K also increases by 6,173 instructions compared to `Sample`, while the pass rate remains around 80% (as expected).
- Perhaps surprisingly, the best overall coverage is obtained with `SampleFuzz`. Its pass rate is 68.24%.
- The difference in absolute coverage between `SampleFuzz` and the next best `Sample+Random` is only 670 instructions. Moreover, after analyzing the coverage set overlap, `SampleFuzz` covers 2,622 more instructions than `Sample+Random`, but also misses 1,952 instructions covered by `Sample+Random`. Therefore, none of these two top-coverage winners fully "simulate" the effects of the other.

Main Takeaway: All the learning-based algorithms considered here are competitive compared to `baseline+Random`, and three of those beat that baseline coverage.

4.8 Main Takeaway: Tension between Coverage and Pass Rate

The main takeaway from all our experiments is the *tension we observe between the coverage and the pass rate*.

This tension is visible in Figure 8. But it is also visible in earlier results: if we correlate the coverage results of Figure 6 with the pass-rate results of Figure 5, we can clearly see that `SampleSpace` has a better pass rate than `Sample`, but `Sample` has a better overall coverage than `SampleSpace` (see `host123` in the bottom right of Figure 6).

Intuitively, this tension can be explained as follows. A pure learning algorithm with a nearly-perfect pass-rate (like `SampleSpace`) generates almost only well-formed objects and exercises little error-handling code. In contrast, a *noisier* learning algorithm (like `Sample`) with a lower pass-rate can not only generate many well-formed objects, but it also generates some ill-formed ones which exercise error-handling code.

Applying a random fuzzing algorithm (like `random`) to previously-generated (nearly) well-formed objects has an even more dramatic effect on lowering the pass rate (see Figure 8) while increasing coverage, again probably due to increased coverage of error-handling code.

The new `SampleFuzz` algorithm seems to hit a sweet spot between both pass rate and coverage. In our experiments, the sweet spot for the pass rate seems to be around $65\% - 70\%$: *this pass rate is high enough to generate diverse well-formed objects that cover a lot of code in the PDF parser, yet low enough to also exercise error-handling code in many parts of that parser.*

Note that instruction coverage is ultimately a better indicator of fuzzing effectiveness than the pass rate, which is instead a learning-quality metric.

4.9 Bugs

In addition to coverage and pass rate, a third metric of interest is of course the number of bugs found. During the experiments previously reported in this

section, no bugs were found. Note that the Edge PDF parser had been thoroughly fuzzed for months with other fuzzers before we performed this study, and that all the bugs found during this prior fuzzing had been fixed in the version of the PDF parser we used for this study.

However, during a longer experiment with `Sample+Random`, 100,000 objects and 300,000 PDF files (which took nearly 5 days), a stack-overflow bug was found in the Edge PDF parser: a regular-size PDF file is generated (its size is 33Kb) but it triggers an unexpected recursion in the parser, which ultimately results in a stack overflow. This bug was later confirmed and fixed by the Microsoft Edge development team. We plan to conduct other longer experiments in the near future.

5 Related Work

Grammar-based fuzzing. Most popular blackbox random fuzzers today support some form of grammar representation, e.g., Peach[3] and SPIKE[4], among many others [27]. Work on grammar-based test input generation started in the 1970's [14, 23] and is related to model-based testing [28]. Test generation from a grammar is usually either random [20, 25, 8] or exaustive [18]. Imperative generation [7, 10] is a related approach in which a custom-made program generates the inputs (in effect, the program encodes the grammar). Grammar-based fuzzing can also be combined with whitebox fuzzing [19, 11].

Learning grammars for grammar-based fuzzing. Bastani et al. [2] present an algorithm to synthesize a context-free grammar given a set of input examples, which is then used to generate new inputs for fuzzing. This algorithm uses a set of generalization steps by introducing repetition and alternation constructs for regular expressions, and merging non-terminals for context-free grammars, which in turn results in a monotonic generalization of the input language. This technique is able to capture hierarchical properties of input formats, but is not well suited for formats such as PDF objects, which are relatively flat but include a large diverse set of content types and key-value pairs. Instead, our approach uses sequence-to-sequence neural-network models to learn *statistical* generative models of such flat formats. Moreover, learning a statistical model also allows for guiding additional fuzzing of the generated inputs.

AUTOGRAM [16] also learns (non-probabilistic) context-free grammars given a set of inputs but by dynamically observing how inputs are processed in a program. It instruments the program under test with dynamic taints that tags memory with input fragments they come from. The parts of the inputs that are processed by the program become syntactic entities in the grammar. Tupni [9] is another system that reverse engineers an input format from examples using a taint tracking mechanism that associate data structures with addresses in the application address space. Unlike our approach that treats the program under

---

[3] http://www.peachfuzzer.com/

[4] http://resources.infosecinstitute.com/fuzzer-automation-with-spike/

test as a black-box, AUTOGRAM and Tupni require access to the program for adding instrumentation, are more complex, and their applicability and precision for complex formats such as PDF objects is unclear.

Neural-networks-based program analysis. There has been a lot of recent interest in using neural networks for program analysis and synthesis. Several neural architectures have been proposed to learn simple algorithms such as array sorting and copying [17, 24]. Neural FlashFill [21] uses novel neural architectures for encoding input-output examples and generating regular-expression-based programs in a domain specific language. Several seq2seq based models have been developed for learning to repair syntax errors in programs [3, 13, 22]. These techniques learn a seq2seq model over a set of correct programs, and then use the learnt model to predict syntax corrections for buggy programs. Other related work optimizes assembly programs using neural representations [4]. In this paper, we present a novel application of seq2seq models to learn grammars from sample inputs for fuzzing purposes.

6 Conclusion and Future Work

Grammar-based fuzzing is effective for fuzzing applications with complex structured inputs provided a comprehensive input grammar is available. This paper describes the first attempt at using neural-network-based statistical learning techniques to automatically generate input grammars from sample inputs. We presented and evaluated algorithms that leverage recent advances in sequence learning by neural networks, namely `seq2seq` recurrent neural networks, to automatically learn a generative model of PDF objects. We devised several sampling techniques to generate new PDF objects from the learnt distribution. We show that the learnt models are not only able to generate a large set of new well-formed objects, but also results in increased coverage of the PDF parser used in our experiments, compared to various forms of random fuzzing.

While the results presented in Section 4 may vary for other applications, our general observations about the tension between conflicting learning and fuzzing goals will remain valid: learning wants to capture the structure of well-formed inputs, while fuzzing wants to break that structure in order to cover unexpected code paths and find bugs. We believe that the inherent statistical nature of learning by neural networks is a powerful tool to address this learn&fuzz challenge.

There are several interesting directions for future work. While the focus of our paper was on learning the structure of PDF objects, it would be worth exploring how to learn, as automatically as possible, the higher-level hierarchical structure of PDF documents involving cross-reference tables, object bodies, and trailer sections that maintain certain complex invariants amongst them. Perhaps some combination of logical inference techniques with neural networks could be powerful enough to achieve this. Also, our learning algorithm is currently agnostic to the application under test. We are considering using some form of reinforcement learning to guide the learning of `seq2seq` models with coverage

feedback from the application, which could potentially guide the learning more explicitly towards increasing coverage.

References

1. Adobe Systems Incorporated. *PDF Reference*, 6th edition, Nov. 2006. Available at `http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf`.
2. Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *CoRR*, abs/1608.01723, 2016.
3. Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.
4. Rudy R. Bunel, Alban Desmaison, Pawan Kumar Mudigonda, Pushmeet Kohli, and Philip H. S. Torr. Adaptive neural compilation. In *NIPS*, pages 1444–1452, 2016.
5. Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.
6. Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*, pages 577–585, 2015.
7. K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of ICFP'2000*, 2000.
8. D. Coppit and J. Lian. yagg: an easy-to-use generator for structured test inputs. In *ASE*, 2005.
9. Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM, 2008.
10. Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *FSE*, 2007.
11. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of PLDI'2008 (ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation)*, pages 206–215, Tucson, June 2008.
12. P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.
13. Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, 2017.
14. K.V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal*, 9(4), 1970.
15. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

16. Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *ASE*, pages 720–725, 2016.
17. Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.
18. R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, 2006.
19. R. Majumdar and R. Xu. Directed Test Generation using Symbolic Grammars. In *ASE*, 2007.
20. P.M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4), 1990.
21. Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.
22. Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. *CoRR*, abs/1607.02902, 2016.
23. P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3), 1972.
24. Scott Reed and Nando de Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.
25. E.G. Sirer and B.N. Bershad. Using production grammars in software testing. In *DSL*, 1999.
26. Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.
27. M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.
28. M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing. *Department of Computer Science, The University of Waikato, New Zealand, Tech. Rep*, 4, 2006.