2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)

Coverage-guided Learning-assisted Grammar-based Fuzzing

Yuma Jitsunari
Tokyo Institute of Technology
jitsunari.y@sde.cs.titech.ac.jp

Yoshitaka Arahori
Tokyo Institute of Technology
arahori@cs.titech.ac.jp

Abstract

Grammar-based fuzzing is known to be an effective technique for checking security vulnerabilities in programs, such as parsers, which take complex structured inputs. Unfortunately, most of existing grammar-based fuzzers require a lot of manual efforts of writing complex input grammars, which hinders their practical use. To address this problem, recently proposed approaches use machine learning to automatically acquire a generative model for structured inputs conforming to a complex grammar. Even such approaches, however, have major limitations: they fail to learn a generative model for instruction sequences, and they cannot achieve good coverage of instruction-parsing code.

To overcome such limitations. this paper proposes a collection of techniques for enhancing learning-assisited grammar-based fuzzing. Our approach allows for the learning of a generative model for instruction sequences by training a hybrid character/token-level recursive neural network. In addition, we exploit coverage metrics gathered during previous runs of fuzzing in order to efficiently refine (or fine-tune) the learnt model so that it can make high coverage-inducing new inputs. Our experiments with a real PDF parser show that our approach succeeded in generating new sequences of instructions (in PDF page streams) that induce better code coverage (of the PDF parser) than state-of-the-art learning-assisted grammar-based fuzzers.

1. Introduction

Fuzzing is an automated testing technique for finding security vulnerabilities or bugs in input-parsing code. It repeatedly synthesizes and feeds inputs to the input-parsing code so that it can find the vulnerabilities in the code. The purpose of synthesizing inputs is to guide the execution of the input-parsing code towards certain states that manifests security vulnerabilities. Existing fuzzing techniques [5][7] have succeeded in finding many security vulnerabilities such as buffer overflows and use-after-frees.

In terms of the way to synthesize inputs, fuzzing techniques can be categorized into either mutation-based or generation-based fuzzing. Based on randomization and/or heuristics, mutation-based fuzzing modifies (or mutates) existing (or previously synthesized) inputs to synthesize new ones, that are likely to explore those execution states (causing vulnerabilities) not found by existing inputs. Mutation-based fuzzing is easy to use, because it does not require the users of expert knowledge about the specification of valid test inputs. Generation-based fuzzing, on the other hand, synthesizes new inputs using a specification of valid inputs. It can always synthesize well-formed inputs, though it requires the (manually-written) input specification from which new inputs are synthesized.

Representative approaches for implementing a fuzz-testing tool include black-box, white-box, and grammar-based ones.

1. **Black-Box Fuzzing.** Black-box fuzzing [7] randomly mutates existing well-formed inputs, typically at byte granularity. It does not require any expert knowledge on the structure of the checked code, and thus easy to apply to various kinds of applications. Mutated inputs may, however, be ill-formed, often resulting in very low code coverage of certain class of code.

2. **White-Box Fuzzing.** White-box fuzzing takes into account the structure of the code under test to systematically generate new test inputs. Traditional white-box fuzzers [3] employ symbolic execution to collect path constraints that capture the structure of the target code, and then use a constraint solver to solve the constraints so that they can generate new test inputs. The generated new inputs are mostly well-formed, and are able to guide the target code towards unexplored execution paths, which results in increased code coverage.

3. **Grammar-based Fuzzing.** Grammar-based fuzzing [6][7] targets those programs, such as parsers, which take complex structured inputs. To generate new inputs, it uses an input-grammar, which specifies the valid form of the inputs. Grammar-based fuzzing enables the generation of new inputs at token granularity. The generated inputs are well-formed because they are guaranteed to conform to the grammar.

978-1-7281-0888-9/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSTW.2019.00065

2. Problems with Related Work

2.1. Grammar-based Fuzzing

Most of the existing grammar-based fuzzers [6][7] have two major limitations: they are (1) prohibitively costly to apply and (2) able to achieve only low code coverage. First, traditional grammar-based fuzzers require the manual description of a complex input-grammar, incurring prohibitively high application cost. Expert knowledge on the grammar is mandatory for correctly writing it by hand. This requirement hinders the practical and wide spread use of grammar-based fuzzing.

The second problem is the low code coverage achieved. New test inputs generated by grammar-based fuzzing give no guarantee that they can guide the code under test towards unexplored execution paths, even though the generated inputs are, by principle, well-formed with respect to a complex input grammar. The reason is that, in general, only a relatively-small subset of the vast space of well-formed inputs can contribute to better code coverage of the input parser. Unlike coverage-guided fuzzing [3][10][11][12][13], grammar-based fuzzing alone cannot generate such inputs that induce better code coverage.

2.2. Learning-assisted Grammar-based Fuzzing

To address the high application-cost problem with grammar-based fuzzing, Godefroid *et al.* propose Learn&Fuzz [1] that exploits machine learning to automatically generate well-formed inputs with respect to a complex grammar. Learn&Fuzz uses a large corpus of example structured-inputs (i.e., PDF objects) to train a recurrent neural network. As a result, it learns a generative model that can automatically generate new structured inputs that are statistically well-formed with respect to the complex input grammar. Learn&Fuzz relieves the users of manually writing the complex input grammar, because it automates the input-grammar synthesis, which is a significant advantage over traditional grammar-based fuzzing. In this respect, Learn&Fuzz has solved the application-cost problem with grammar-based fuzzing. DeepSmith [2] uses a very similar machine-learning technique with Learn&Fuzz to solve the same problem in the context of fuzzing compilers, not PDF parsers.

2.3. Problem Statement

Unfortunately, even the state-of-the-art learning-assisted grammar-based fuzzer, Learn&Fuzz, still has two major drawbacks.

1. **Generation of Instruction Sequences.** The generative model in Learn&Fuzz fails to generate sequences of instructions, a more complex form of structured inputs. Learn&Fuzz cannot learn a generative model for generating complex instruction sequences of PDF page streams, though it synthesizes a simple input grammar, which specifies the text format of PDF objects. Instruction sequences of page streams constitute an important binary-format part of the entire PDF file, which are responsible for many kinds of programmable tasks. To generalize, Learn&Fuzz cannot automatically generate instruction sequences. Because instruction sequences often appear in structured inputs, not limited to PDF files, this is a major drawback.

2. **Improvement of Code Coverage.** Learn&Fuzz fails to achieve good code coverage of programs that take instruction sequences as a part of structured inputs. Some mutation-based fuzzers [10][12] use code-coverage metrics to steer themselves towards previously unexplored code. However, to our knowledge, existing learning-assisted grammar-based fuzzers have no capability of coverage-guided code exploration, resulting in not-enough code coverage.

3. Our Approach

To address the major problems with Learn&Fuzz, we propose coverage-guided learning-assisted grammar-based fuzzing. Figure 1 shows the overview of our approach in the context of fuzzing a PDF parser (i.e., PDFium [14]). As the figure shows, the key mechanism of our approach is the feedback loop, which exploits the coverage information collected during the previous runs of fuzzing to fine-tune the already-learnt model so that it can generate interesting new inputs (i.e., new instruction sequences of page streams) that are likely to induce better code coverage. Now we describe, in more detail, each step of our approach in the context of fuzzing the PDF parser.

3.1. Learning a Generative Model for New Test Inputs

As the first step, we learn a generative model for new inputs (i.e. PDF files) by training a recurrent neural network (RNN, in short) using a large corpus of PDF files collected from the Internet. More specifically, we first collect a number of PDF files and extract page streams (i.e., instruction sequences) from the collected files. In our experiments described later, we extracted 10988 page streams from the 261 collected PDF files. Using these page streams, we train an RNN to learn a generative model for new page streams to be embedded into new PDF files. The generative

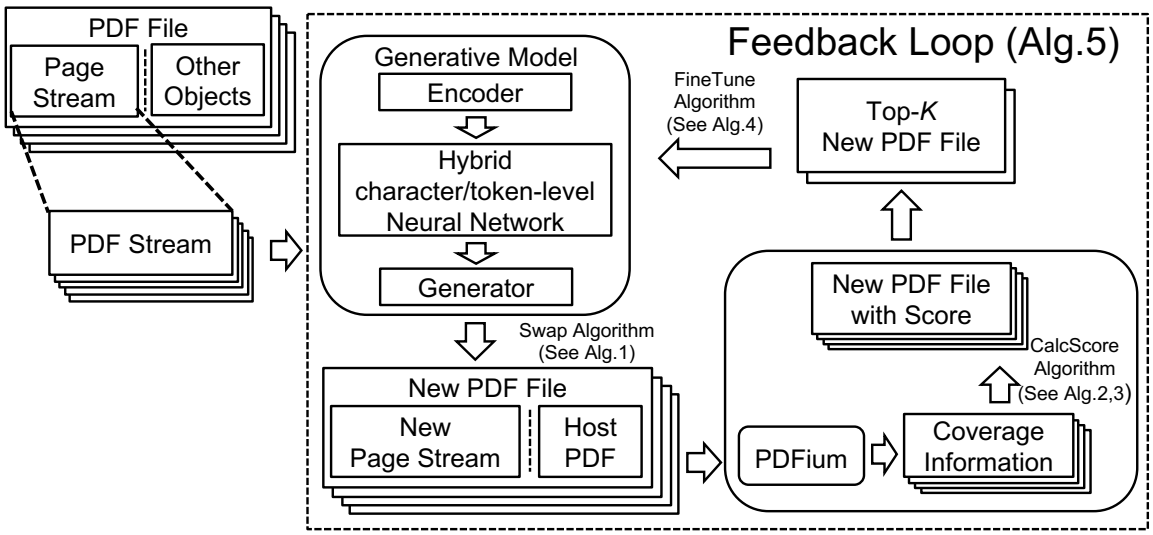

Figure 1. Overview of coverage-guided learning-assisted grammar-based fuzzing.

model consists conceptually of three parts: encoder, hybrid char/token-level neural network, and generator.

Encoder. In order to learn a generative model for well-formed instruction sequences of page streams, we use a hybrid char/token-RNN model. The char/token-RNN model enables us to avoid generating ill-formed instructions with invalid opcode, while allowing for the generation of interesting instruction operands. In our model, we treat instruction opcodes of more than two characters as individual tokens, while the rest part (including operands) is encoded at character-level.

Hybrid char/token-level Neural Network. Our implementation of the generative model is an LSTM [4] with two hidden layers, each of which consists of 128 hidden nodes (or states). We train the model using Adam [8] for 10 epochs, with a learning rate of 0.001. In our experimental setting described later, the learning finally generates 134 output nodes for char/token-RNN.

Generator. Using the learnt model, we generate new page streams to be embedded into new PDF files (which will later be fed into the PDF parser). The learnt model represents a probability distribution, which takes as an input a character-sequence and predicts and produces a next character with its probability. For the generation of new page streams, we use the `SampleFuzz` algorithm in Learn&Fuzz [1]. In our setting, the algorithm takes as an input the start character-sequence "`stream`" and repeat generating next characters by sampling the learnt probability-distribution until producing the end character-sequence "`endstream`".

In this generation process, we modify (or fuzz) the predicted sequence of page-stream instructions, using the `Swap` algorithm (see Algorithm 1). This algorithm scans a page stream generated from the learnt model and, for each line, checks if a random value (between 0 and 1) exceeds a threshold parameter t_{swap} (set to 0.9 in our experiments). If the random value exceeds the threshold, then it swaps the current line with the previous one. The effect of this algorithm is meant to be the injection of anomalies at instruction-level granularity, so that it can induce better code coverage of the parser of page-stream instructions.

Algorithm 1 `Swap`(*generated_page_stream*, t_{swap})

```
1: for each line ∈ generated_page_stream do
2:   p_swap := random(0, 1)
3:   if p_swap > t_swap then
4:     DoSwap(line.prev, line)
5:   end if
6: end for
```

3.2. Extracting Top-K High Coverage-Inducing Inputs

The second step of our approach is to extract those top-K PDF files (from the new PDF files generated by the learnt model), which induce high code coverage when fed into the target PDF parser. At the later third step, these top-K PDF files are used to fine-tune the already-learnt model so that it can generate interesting new page streams, which are likely to induce better code coverage of the PDF parser. The prototype implementation of our approach sets $K = 100$.

To select top-K PDF files, we calculate a score for each newly generated PDF file in terms of code coverage that PDFium achieves when parsing the input file (see Algorithm 2). Our code-coverage metric captures how many lines of the PDFium code are executed for parsing a given

PDF file. Based on our coverage metric, we calculate a total score for each input PDF file by accumulating a per-line score for each executed line. Such a per-line score is weighted if an executed line includes a branch attributed to keywords such as `if`, `else`, `switch`, and `case` (see Algorithm 3). The reason for this branch-weighting is that a branch tends to be an entrance to unexplored code fragments. We believe that our input-file scoring based on branch-weighting allows for the selection of interesting PDF files that have a structure to induce high code coverage of the PDF parser.

Algorithm 2 `CalcScore`(*input_file*)

```
1: score := 0
2: /* Do parse and get line coverage of Parser. */
3: Parser.parse(input_file)
4: for each line ∈ Parser do
5:    if line.cov > 0 then
6:       score += WeightLine(line)
7:    end if
8: end for
9: return score
```

Algorithm 3 `WeightLine`(*line*)

```
1: if line includes a branch then
2:    return M /* In our setting, M=100. */
3: else
4:    return 1
5: end if
```

3.3. Fine-Tuning the Learnt Model using Top-K High Coverage-Inducing Inputs

The third step of our approach is to fine-tune the learnt model using top-K new PDF files (Algotirhm 4), so that the fine-tuned model can generate interesting new page-stream instructions that are likely to result in better code coverage of PDFium.

Algorithm 4 `FineTune`(*learnt_model*, *topK_input_files*)

```
1: FT_model = train(learnt_model, topK_input_files)
2: return FT_model
```

Fine tuning [9] is an efficient learning technique for reusing and refining an already-learnt statistical model so that it can better perform a similar but more specialized task. Using an initial dataset for some task, the preliminary step for fine tuning trains a neural network to learn a statistical model (or a probability distribution), which can be used to perform that task. Then the fine-tuning step reuses and refines the learnt model, i.e., further trains the learnt model with another small dataset for a similar but different target task. As a result, thus refined (or fine-tuned) model can perform the target task. Because it requires only a small dataset to adapt the already-learnt model into the target task, fine tuning allows for the efficient learning of a model to perform the target task.

In our setting, the purpose of applying fine tuning is to learn a refined generative model for page-stream instructions which induce high code coverage of the PDF parser. For this purpose, the preliminary step for our fine-tuning is to learn a base generative model for page streams by training a char/token-RNN using a large corpus of PDF files collected from the Internet (line 1 of Algorithm 5). We then use the learnt model to generate new page streams to be embedded into new input PDF files (line 3 of Algorithm 5), and we extract top-K PDF files which induce top-K high code coverage of the PDF parser (line 4 of Algorithm 5). Then we fine-tune the learnt model by further training it with the top-K PDF files (line 5 of Algorithm 5). Our approach repeats such *coverage-guided* fine-tuning steps until the total number of fine-tuning iterations reach a user-specified count *num_FTs* (line 2 to 6 of Algorithm 5). Here, we apply fine tuning to all layers of the trained network, using a learning rate of 0.0001. Finally, as a result of our coverage-guided fine-tuning, we obtain a refined generative model for those page streams that are likely to induce high code coverage of the PDF parser.

Algorithm 5 `FeedbackLoop`(*init_inputs*, *num_FTs*)

```
1: learnt_model := trainCharTokenRNN(init_inputs)
2: for i=1 to num_FTs do
3:    new_inputs := genNewInputs(learnt_model)
4:    topK_inputs := getTopK(new_inputs, K)
5:    learnt_model :=
        FineTune(learnt_model, topK_inputs)
6: end for
7: return learnt_model
```

4. Experiments

4.1. Goal and Setup

To confirm the effectiveness of our coverage-guided learning-assisted grammar-based fuzzing, we performed comparative experiments with PDFium [14], a real PDF parser. The goal of our experiments is to measure to what degree our approach is able to achieve better code coverage than state-of-the-art learning-assisted grammar-based fuzzers, i.e., Learn&Fuzz [1] and DeepSmith [2]. Our experiments were conducted on a hardware/software environment, shown in Table 1.

Table 1. Experimental setup.

Hardware/Software	Spec/Version
CPU	Intel Core $i7$-6700
GPU	NVIDIA GeForce GTX980 (4GB)
RAM	16GB DDR4 SDRAM
OS	Ubuntu 16.04 LTS
ML-Framework	CUDA V8.0.61, TensorFlow GPU 1.4.0, Keras 2.1.3

4.2. Methodology

To examine to what degree our approach can achieve better code coverage than the state-of-the-arts techniques Learn&Fuzz and DeepSmith, we measure and compare line and function coverage of PDFium achieved by our approach and the state-of-the-arts. For this coverage measurement, we made PDFium parse each set of new PDF files generated by our approach and the state-of-the-arts, respectively. We generated these new PDFs in the following way. First, we collected 261 PDF files from the Internet and, extracted 10988 page streams, as a training dataset, from the collected PDF files. We then used the extracted page streams to train the char/token-RNN of our approach and the recurrent neural networks of Learn&Fuzz and DeepSmith, respectively. By this training, we learned, for each approach, a generative model for new page streams. Table 2 shows the learnt models (i.e., comparison targets in our coverage measurement) with their leaning configuration. Then we made each of the learnt models generate 1000 new page streams, and we embedded a host PDF file with newly generated page streams to obtain 1000 new PDF files for each learnt model. Finally, we fed each set of newly generated PDF files into PDFium to measure and compare the line and function coverage of PDFium, which corresponds to code coverage achieved by our approach and the state-of-the-arts, respectively.

4.3. Results

Figure 2 and 3 show the results of our line- and function-coverage measurements, respectively. The y-axis indicates the line-/function-coverage achieved, while the x-axis indicates the number of iterations of fine-tunings (i.e., *num_FTs* in Algorithm 5) by which our neural networks (i.e., our models 1 and 2 in Table 2) were trained.

The results show that both of the line- and function-coverage achieved by our models (i.e., LF+CT+FT and LF+CT+Swap+FT) tend to increase as the number of fine-tuning iterations do. This means that, within our experiments, our coverage-guided fine-tuning of the already-learnt model is effective for enhancing the model's capability of generating those new instruction sequences that are likely to induce better code coverage of the target parser.

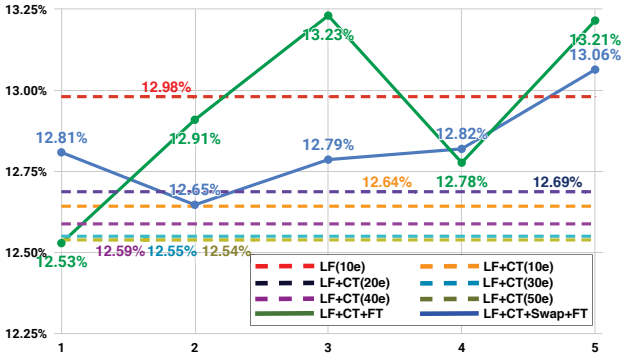

Figure 2. Line coverage for our approach vs. state-of-the-arts.

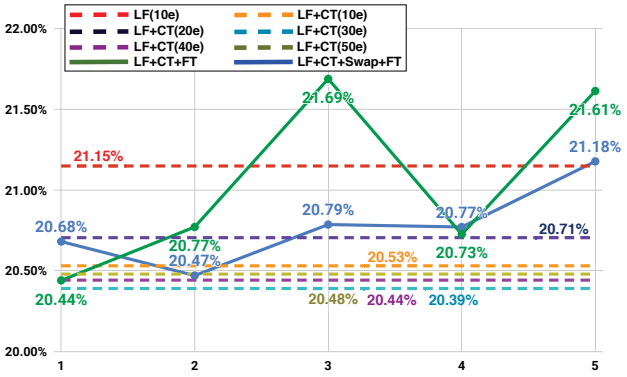

Figure 3. Function coverage for our approach vs. state-of-the-arts.

Fortunately, our models trained by five times of fine-tunings succeeded in achieving better coverage than the state-of-the-art fuzzers, Learn&Fuzz (i.e., LF) and DeepSmith (i.e., LF+CT). Notably, our models trained for short (=10) epochs plus five times of fine-tunings marked better coverage, even compared to LF+CT(20/30/40/50e), Learn&Fuzz with char/token-RNN (i.e., the DeepSmith model) trained for long (=20/30/40/50) epochs. This result implies that our approach is *efficient* at learning a generative model for instruction sequences that induce better coverage. The reason is that taking long epochs to learn a high-coverage model typically requires much more time than short-epoch learning plus a small number of fine-tunings. According to our learning-time measurements (not described in this paper), the overall learning time of our feedback loop (i.e., initial 10-epoch training plus five iterationss of fine-tunings)

Table 2. Comparison targets.

Comparison target	Description
LF(10e)	Learn&Fuzz (i.e., char-RNN) trained for 10 epochs.
LF+CT(10/20/30/40/50e)	DeepSmith (i.e., Learn&Fuzz w/ char/token-RNN) trained for 10/20/30/40/50 epochs.
LF+CT+FT	Our model 1: Learn&Fuzz w/ char/token-RNN trained by fine tuning.
LF+CT+Swap+FT	Our model 2: Learn&Fuzz w/ char/token-RNN and `Swap`-based generator trained by fine tuning.

is much less than that of LF+CT(20/30/40/50e), while our approach achive better coverage than any of them. We thus believe that our fine-tuning-based approach is an efficient technique for achieving better coverage.

Note that our results do not imply that we can always achieve better coverage as the number of iterations increase. In our future work, we plan to find better hyperparameters (including the number of fine-tuning iterations, a learning rate, the value of K in top-K, and the number of layers in the trained network, and the stochastic optimization method other than Adam) so that we can achieve better code coverage.

Unfortunately, in our experiments, we could not confirm the effectiveness of the `Swap` algorithm for generating instruction sequences that induce better coverage because, for five times of fine-tunings, LF+CT+Swap+FT achieved lower coverage than LF+CT+FT. This result leaves a challenge to pursue a more sophisticated fuzzing algorithm for achieving better coverage by injecting anomalies into instruction sequences generated by the learnt model.

5. Conclusion and Future Work

Grammar-based fuzzing is effective for checking security bugs in input parsers with complex structured inputs. Despite their effectiveness, most of existing grammar-based fuzzers require a large amount of manual efforts of writing a complex input grammar. Recently proposed learning-assisted grammar-based fuzzers are promising in that they can automatically learn a generative model for complex structured inputs from a large corpus of training inputs. However, they have a major problem that they fail to generate interesting instruction sequences that are likely to induce good coverage of the instruction-parsing code.

To address the problem, this paper presents coverage-guided learning-assisted grammar-based fuzzing. Our approach first trains a hybrid character/token-level recurrent neural network to learn a generative model for instruction sequences. Then we repeatedly fine-tune the learnt model using top-K high coverage-inducing inputs generated by the learnt model. Our coverage-guided iteration of fine-tunings enables the learning of a refined generative model that captures the characteristics of the input structure that induces better code coverage of the instruction parser.

The results of our experiments with a real PDF parser show that our approach is able to achieve better code coverage of the parser of PDF page streams, which contain instruction sequences, compared to state-of-the-art learning-assisted grammar-based fuzzers (i.e., Learn&Fuzz and DeepSmith). The results also leave future work to find better hyperparameters and to establish a more sophisticated fuzzing algorithm to inject anomalies into generated instruction sequences so that it can achieve higher code coverage of the instruction parser.

References

[1] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn & Fuzz: Machine Learning for Input Fuzzing. In *ASE 2017*, pages 50–59.

[2] Chris Cummins , Pavlos etoumenos , Alastair Murray , and Hugh Leather. Compiler Fuzzing through Deep Learning. In *ISSTA 2018*, pages 95–105.

[3] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *NDSS 2008*, pages 151–166.

[4] Sepp Hochreiter and Jurgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[5] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: A Survey. *Cybersecurity 2018*, 1(1).

[6] Paul Purdom. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics 1972*, 12(3).

[7] Michael Sutton, Adam Greene, and Pedram Amini. Fuzzing: Brute force vulnerability discovery.

[8] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *ICLR 2014*, *CoRR*, abs/1412.6980, 2014.

[9] Karl Weiss, Taghi Khoshgoftaar, and DingDing Wang. A Survey of Transfer Learning. *Journal of Big Data*, 3(1):1–40, 2016.

[10] Marcel Bohme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *CCS 2016*, pages 1032–1043.

[11] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS 2017*.

[12] Konstantin Bottinger, Patrice Godefroid, and Rishabh Singh. Deep Reinforcement Fuzzing. In *SPW 2018*, pages 116–122.

[13] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Coverage-Guided Fuzzing for Deep Neural Networks. *CoRR*, abs/1809.01266, 2018.

[14] Google PDFium. *https://www.pdfium.org/*.