Reinforcement Learning on Testing

Presented by : Ji Gao

[1]Department of Computer Science, University of Virginia
https://qdata.github.io/deep2Read/

August 26, 2018

Overview

Outline

1. Learn&Fuzz: Machine Learning for Input Fuzzing

2. Deep Reinforcement Fuzzing

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 Microsofts 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.

intuition

- Grammar based fuzzing: Knows the grammar of the model
- Claimed as most effective fuzzing technique known today for fuzzing applications
- This work: Learn a generative language model over the set of PDF object characters given a large corpus of objects

PDF format

```
2 0 obj
<<
/Type /Pages
/Kids [ 3 0 R ]
/Count 1
>>
endobj
```

(a)

```
xref
0 6
0000000000 65535 f
0000000010 00000 n
0000000059 00000 n
0000000118 00000 n
0000000296 00000 n
0000000377 00000 n
0000000395 00000 n
```

(b)

```
trailer
<<
/Size 18
/Info 17 0 R
/Root 1 0 R
>>
startxref
3661
```

(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.

- A PDF body is composed of three sections: objects, cross-reference table, and trailer.

Object

```
125 0 obj
[680.6 680.6]
endobj
```
(a)

```
88 0 obj
(Related Work)
endobj
```
(b)

```
75 0 obj
4171
endobj
```
(c)

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

Fig. 2. PDF data objects of different types.

- Object: first line: ID + generation number + obj
- marked by "endobj"
- Different type of data inside

Cross reference table

```
2 0 obj
<<
/Type /Pages
/Kids [ 3 0 R ]
/Count 1
>>
endobj
```

(a)

```
xref
0 6
0000000000 65535 f
0000000010 00000 n
0000000059 00000 n
0000000118 00000 n
0000000296 00000 n
0000000377 00000 n
0000000395 00000 n
```

(b)

```
trailer
<<
/Size 18
/Info 17 0 R
/Root 1 0 R
>>
startxref
3661
```

(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.

- Cross reference tables contain the address of referenced objects within the document
- 2nd number indicates previous free object
- n = object in use, f = not used

Trailer

```
2 0 obj
<<
/Type /Pages
/Kids [ 3 0 R ]
/Count 1
>>
endobj
```

(a)

```
xref
0 6
0000000000 65535 f
0000000010 00000 n
0000000059 00000 n
0000000118 00000 n
0000000296 00000 n
0000000377 00000 n
0000000395 00000 n
```

(b)

```
trailer
<<
/Size 18
/Info 17 0 R
/Root 1 0 R
>>
startxref
3661
```

(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.

- a dictionary of information about the body
- startxref which is the address of the cross-reference table.

Seq2seq generation model

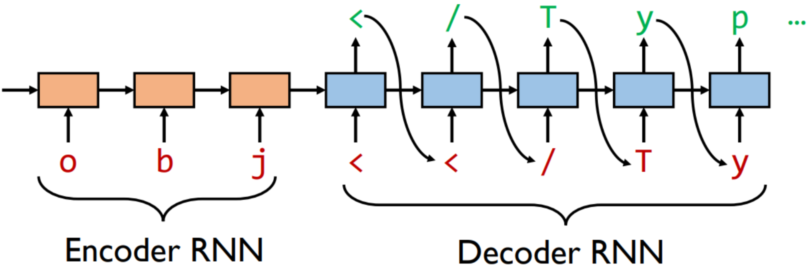

Fig. 3. A sequence-to-sequence RNN model to generate PDF objects.

- Learn from objects

Seq2seq generation model(Contd.)

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

```
seq := "obj "
while ¬ seq.endswith("endobj") do
    c,p(c) := sample(D(seq,θ)) (* Sample c from the learnt distribution *)
    p_fuzz := random(0, 1) (* random variable to decide whether to fuzz *)
    if p_fuzz > t_fuzz ∧ p(c) > p_t then
        c := argmin_c' {p(c') ~ D(seq, θ)} (* replace c by c' (with lowest likelihood) *)
    end if
    seq := seq + c
    if len(seq) > MAXLEN then
        seq := "obj " (* Reset the sequence *)
    end if
end while
return seq
```

- Do revised sampling to guarantee it provide well formed object

Experiment result

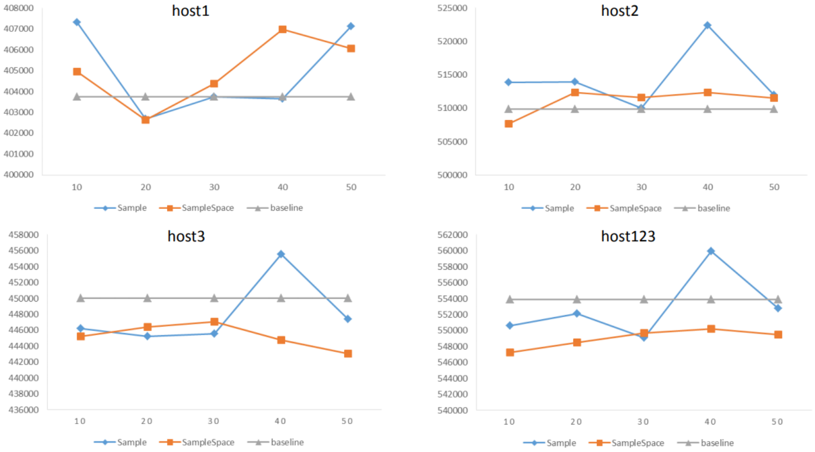

Fig. 6. Coverage for `Sample` and `SampleSpace` from 10 to 50 epochs, for `host 1, 2, 3,` and `123`.

Outline

ArXiv:1801.04589

Abstract: Fuzzing is the process of finding security vulnerabilities in input-processing code by repeatedly testing the code with modified inputs. In this paper, we formalize fuzzing as a reinforcement learning problem using the concept of Markov decision processes. This in turn allows us to apply state-of-the-art deep Q-learning algorithms that optimize rewards, which we define from runtime properties of the program under test. By observing the rewards caused by mutating with a specific set of actions performed on an initial program input, the fuzzing agent learns a policy that can next generate new higher-reward inputs. We have implemented this new approach, and preliminary empirical evidence shows that reinforcement fuzzing can outperform baseline random fuzzing.

Fuzzing

- Fuzzing is the process of finding security vulnerabilities in input-processing code by repeatedly testing the code with modified, or fuzzed, inputs.
- Fuzzing heuristics: The algorithm to prioritize what (parts of) inputs to fuzz next.
  Can be pure random or optimizing for a specific goal, such as maximizing code coverage.
- Fuzzing $\approx$ Adversarial sample.

State-of-the-art Fuzzing

- Proposed as a cheap technique.
- Cheap and easy to be automatic implemented $\rightarrow$ One of the most popular technique in testing

State-of-the-art Fuzzing

- Proposed as a cheap technique.
- Cheap and easy to be automatic implemented $\rightarrow$ One of the most popular technique in testing
- State-of-the-art: coverage based.
  - SAGE from Microsoft: Based on SMT solver.
    One sentence: Build symbolic SMT equations on the branches and try to optimize the coverage by solving them.
  - AFL: Based on genetic programming.

State-of-the-art Fuzzing

- Proposed as a cheap technique.
- Cheap and easy to be automatic implemented $\rightarrow$ One of the most popular technique in testing
- State-of-the-art: coverage based.
  - SAGE from Microsoft: Based on SMT solver.
    One sentence: Build symbolic SMT equations on the branches and try to optimize the coverage by solving them.
  - AFL: Based on genetic programming.
- Been used on many large projects and found expensive bugs.

Setting

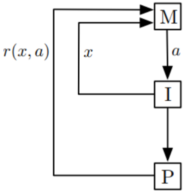

Fig. 1. Modeling Fuzzing as a Markov decision process.

- M: Fuzzer
- a: Fuzzing action
- P: Target program
- I: Input
- Can be viewed as a RL process

Reinforcement Learning

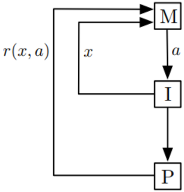

Fig. 1. Modeling Fuzzing as a Markov decision process.

- RL process: State set $x \in X$, Action set $a \in A$, Transition P
- Goal of the agent: Maximize the cumulative reward $R = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$
- Policy $\pi(\cdot|x)$

Method

- State: A string indicates program input
- Action: Suppose to be $I \rightarrow (I \times I, F, P)$. I: Sequence space. F: σ-algebra of the sample space, measurement of I. P: Probability for given rules
- Reward: $r(x, a) = E(x) + G(a)$
  In experiment E is defined as a combination of number of newly discovered basic blocks, execution path length, and execution time of the target.

Process

- Start with initial seed input $x \in I$, unconstrained.
- Initialize Q function as a deep neural net
- After that:

Input: Program P

$x \leftarrow \text{Seed}()$
$Q \leftarrow \text{Qnet}()$

do:

$x' \leftarrow \text{State}(x)$
$a \leftarrow \text{Action}(x', Q)$
$x \leftarrow \text{Mutate}(x, a)$
$r \leftarrow \text{Reward}(P, x)$
$Q \leftarrow \text{Update}(Q, x', a, r)$
$x \leftarrow \text{Reset}()$

while (true)

Fig. 2. Reinforcement fuzzing algorithm.

Process - Details

- State(x): get a sub-string x' at offset o and length l from state x.
- Action(x',Q): Sampling current Q function on state x_0 to get an action $a \in A$
- Mutate(x,a): Applying action a on x
- Reward(): $r(x, a) = E(x) + G(a)$
- Update: Update the Q function based on Reward. Use memory replay (x_t, a_t, r_t, x_{t+1})
- Reset: Set the input to a valid input.

Target: PDF

- PDF: A complicated format, whose introduction has 1300 more pages
- PDF document: a sequence of PDF bodies, each contains three sections – objects, cross-reference table, and trailer
- Test against pdftotext parser

Implementation

- Actions:
  - Random Bit Flips.
  - Insert Dictionary Tokens: Tokens from a dictionary, which selects from other valid input files
  - Shift Offset and Width. Change offset and with.
  - Shuffle: We define two actions for shuffling substrings. The first action shuffles bytes within the pointer, the second action shuffles three segments of the PDF object that is located around offset o.
  - Copy Window. copy the x' to a random place in x, considering both overwrite and insert.
  - Delete Window. Remove x'

Implementation

- Reward: three different types: Code coverage, execution time and combination of both.
- Baseline: A fuzzer that random select actions in A.
- Coverage: Use code coverage, measured using tools

Experiment

	Improvement
Reward functions	
Code coverage r_1	7.75%
Execution time r_2	7%
Combined r_3	11.3%
State width $w = \|x'\|$	
r_2 with $w = 32$ Bytes	7%
r_2 with $w = 80$ Bytes	3.1%
Generalization to new inputs	
r_2 for new input x	4.7%

TABLE I

THE IMPROVEMENTS COMPARED TO THE BASELINE (AS DEFINED INVI-C1) IN THE MOST RECENT 500 ACCUMULATED REWARDS AFTER TRAINING THE MODELS FOR 1000 GENERATIONS.

tanh	sigmoid	elu	softplus	softsign	relu
7.75%	6.56%	5.3%	2%	6.4%	1.3%