Learn & Fuzz

Patrice Godefroid, Hila Peleg, and Rishabh Singh
Microsoft Research

Presenter: Yoongbok Lee

Outline

- Fuzzing
- PDF files
- RNN (LSTM)
- Training methods
- Analysis

Fuzzing

- **Testing an input-parsing** code by **generating inputs**
  - Blackbox
    - No limitations on the input format
  - Whitebox
    - Maximize code coverage
  - Grammar-based
    - So far the most effective method

Grammar-based fuzzing

- Providing **specific constraints** for the inputs to be generated
  - Need to be done by hand
  - A lot of work
  - Error-prone

Objective

- Automatically generate input based on the grammar with machine learning
  - similar with grammar based fuzzing, but no manual specifications
  - large corpus of sample inputs
- Previous attempts
  - Genetic algorithm
  - Context-free grammar learning algorithms
- This paper
  - first attempt at using neural-network-based statistical learning techniques

Process

model

- Machine-learning model
- Need sample inputs for training for a specific input format

data

- Large data of sample input for an input format
- Used to train the model

fuzz

- Generate inputs/fuzz
- Test for pass rate and coverage.

Case Study: PDF

- PDF: a complicated input format
  - 1300 pages format specification
- Microsoft Edge browser's PDF parser
  - Specifically PDF objects parser

PDF format

- PDF
    - A sequence of at least one PDF body
- PDF body
    - Objects
    - Cross-reference table
    - 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)

PDF data objects

- Similarly formatted
- First line identifier (for indirect reference)
- Generation number (incremented when the object is updated/overwritten)
- “obj” keyword to start the actual object
- “endobj” to end the object
- PDF objects are updated incrementally

```
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)

Scope of the paper

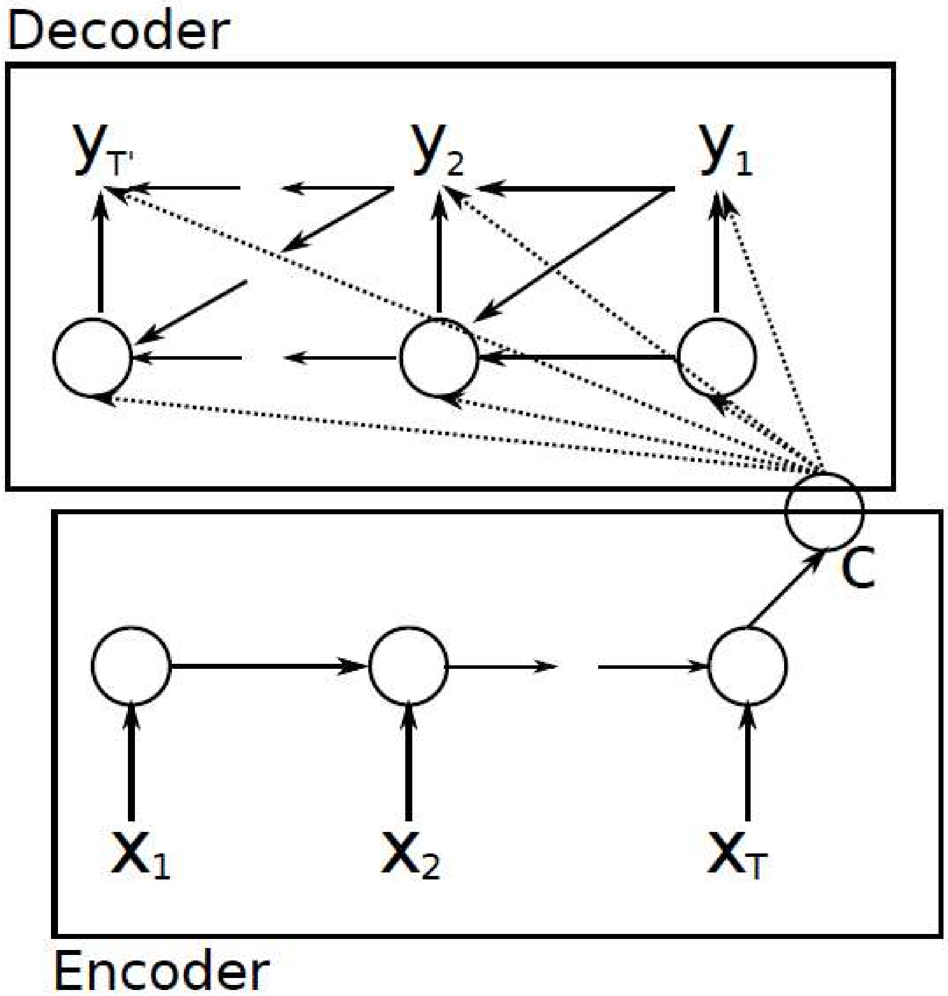
- Non-binary PDF data objects
  - Formatted text
  - Well-suited for learning with neural networks
- Binary PDF data objects
  - Blackbox and whitebox testings are sufficient enough for these formats

Recurrent Neural Networks

- *Given* $x_1\ x_2\ x_3\ x_4\ x_5 \ldots x_{t-2}\ x_{t-1}$
- Generate $x_t\ x_{t+1} \ldots$
- *Recurrent Neural Network (seq2seq)*
  - Operates on variable length inputs
    - Arbitrary length input, rather than n-grams
    - $h_t = f(h_{t-1}, x)$
      - x is the new input, h_i is the hidden state at character *i*.
    - $y_t = \phi(h_t)$
      - ϕ is the activation function, y_i is the i-th output.
    - i.e., learning the conditional distribution $P(x|\langle x_1, \ldots, x_{t-1}\rangle)$

Seq2seq

- Variant of RNN (LSTM)
  - Encoder-decoder
- Hidden state
  - $h_{<t>} = f(h_{<t-1>}, y_{t-1}, c)$
    - c is the summary of the input sequence
    - y_{t-1} of last symbol
    - h_{t-1} of last hidden state
- Conditional distribution
  - $P(y_t | y_{t-1}, y_{t-2}, \dots, y_1, c) = g(h_{<t>}, y_{t-1}, c)$



Decoder
$y_{T'}$
y_2
y_1
c
x_1
x_2
x_T
Encoder

Gated Recurrent Unit

Notations

x : input

$h_{<t-1>}$: previous state

σ: Logistic sigmoid function

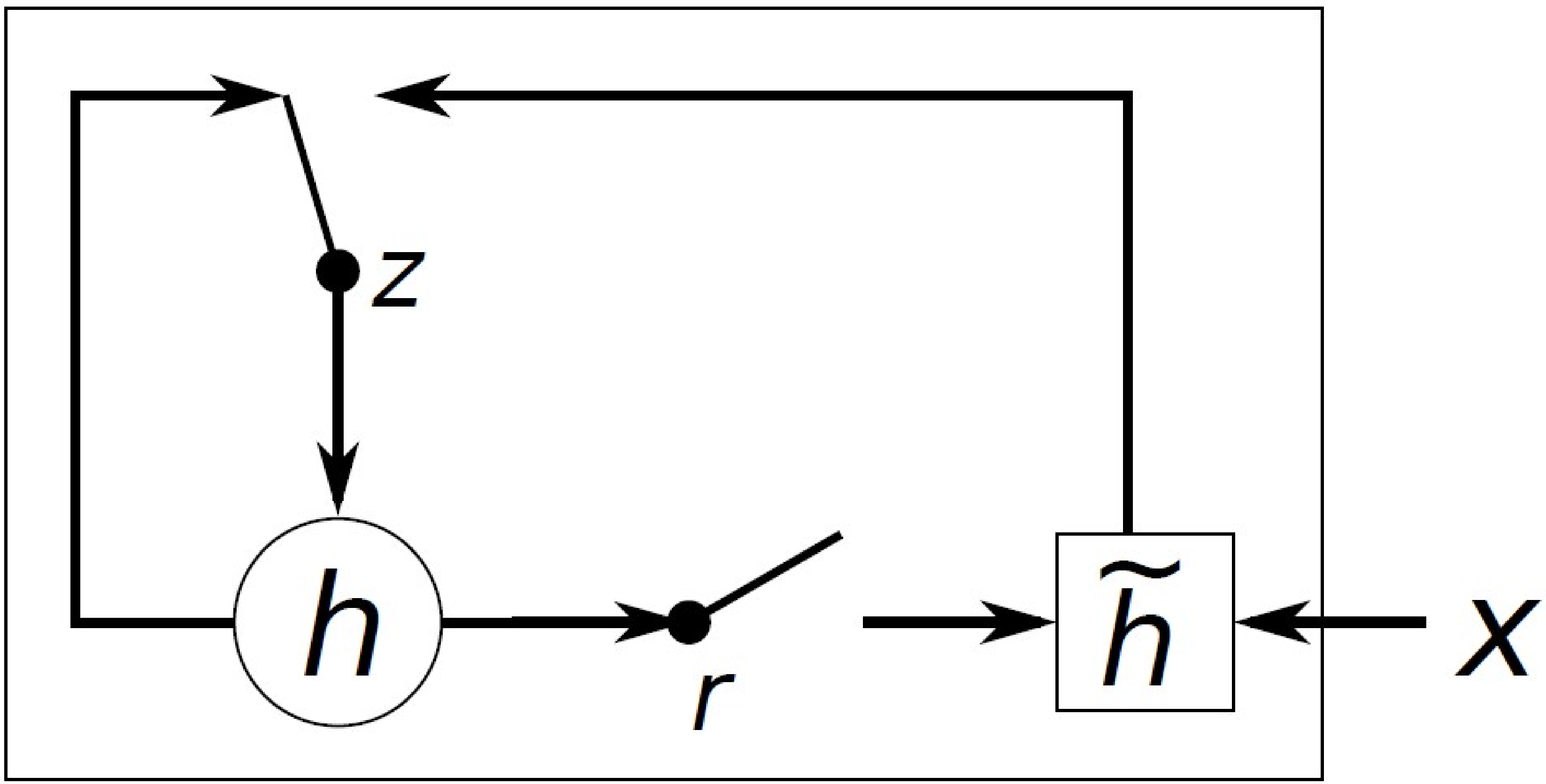
$[.]_i$: i-th element of a vector

W, U: Learnt Weight matrices

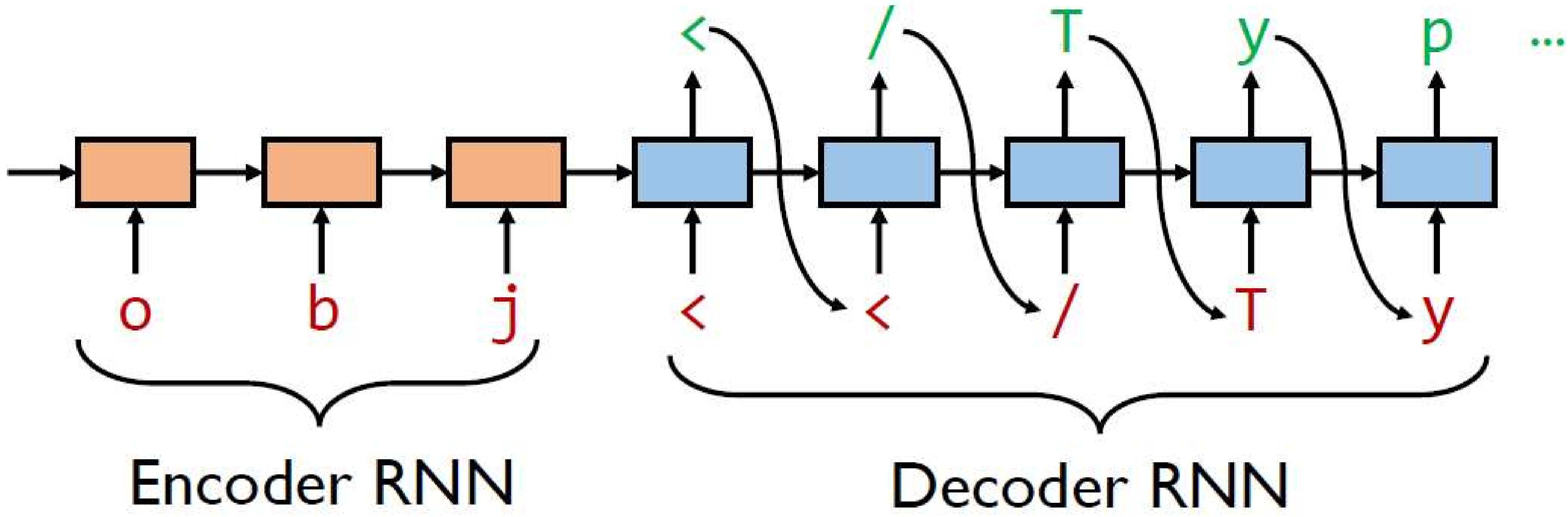
$\odot$: Element-wise product

$h_0 = 0$ vector

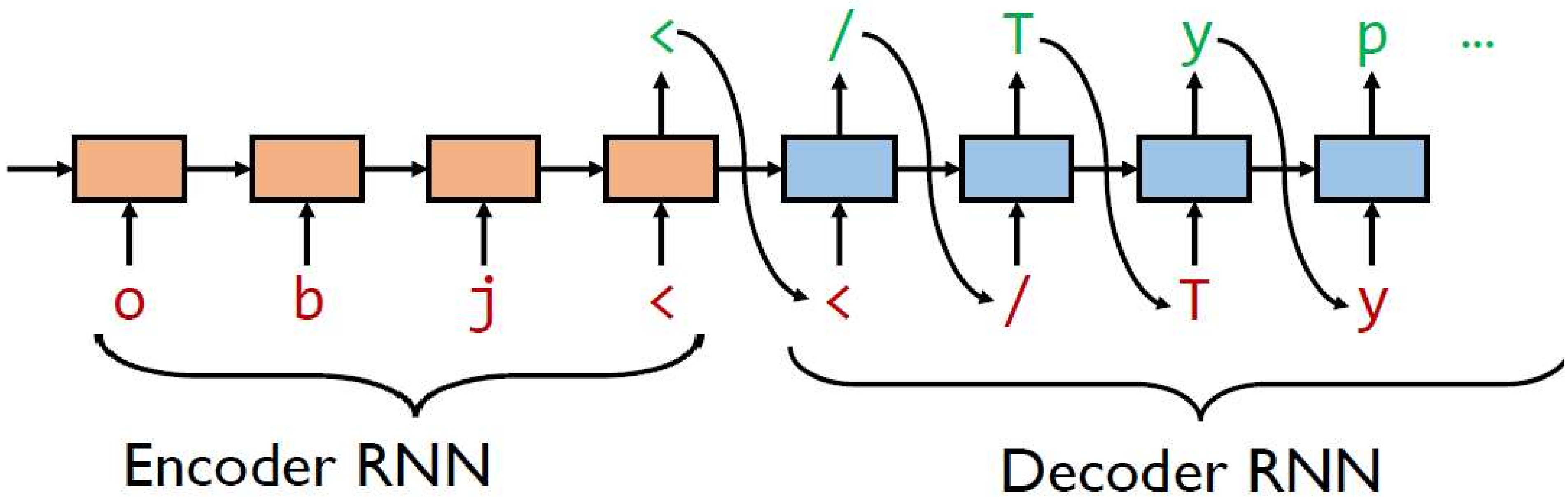
- Reset gate
  - $r_j = \sigma([W_r x]_j + [U_r h_{<t-1>}]_j)$
- Update gate
  - $z_j = \sigma([W_z x]_j + [U_z h_{<t-1>}]_j)$
- Activation
  - $\hbar_j^{<t>} = \phi([Wx]_j + [U(r \odot h_{<t-1>})]_j)$
  - $h_j^{<t>} = z_j h_j^{<t-1>} + (1 - z_j)\hbar_j^{<t>}$



z
h
r
h̃
x



< / T y p ...
o b j < < / T y
Encoder RNN
Decoder RNN



< / T y p ...
o b j <
< / T y
Encoder RNN
Decoder RNN

Training Process

- From a large corpus of PDF object files $s_1, s_2, s_3, \dots, s_n$ make a concatenation of all of the files $s = s_1 + s_2 + s_3 + \cdots + s_n$
- Put multiple training sets of fixed size d.
    - Thus the i-th training sequence $t_i = s[i * d: (i+1) * d]$
- Put output sequence as the input sequence shifted by 1 position
    - Thus the i-th output sequence $o_i = s[i * d + 1: (i+1) * d + 1]$
- Then seq2seq trained end-to-end to learn a generative model over the instances

Generating PDF objects

- Basic idea
  - Start with the prefix “obj”, query the model until it generates “endobj”
- Strategies
  - NoSample
  - Sample
  - SampleSpace

NoSample

- Greedy algorithm to generate the best character given a prefix.
- Most likely to generate well-formed objects, but less likely to create diverse formats of objects
- Precluded from being useful in fuzzing

Sample

- Given a prefix, sample the set of next possible characters (rather than picking the best one)
- Allows generation of diverse objects by combining various different patterns
- Sampling process creates some possibility that the generated object is not well-formed (good for fuzzing)

SampleSpace

- Combination of NoSample and Sample
- Samples the distribution to generate the next character ***only*** when the current character is a whitespace
  - While in middle of a word, generate using NoSample method
  - After completing a word, generate using Sample method
- Expected to generate more well-formed objects than Sample

Challenge

- Challenge
  - Too good training technique:
    - Would mostly consist of well-formed objects that would not execute error-handling code
  - Too bad training technique:
    - Would mostly consist of ill-formed objects that would be rejected by the parser before entering major parts of the parser
- Solution: SampleFuzz

SampleFuzz

- Input
  - Learnt distribution *D(x, Θ)*
  - Probability of fuzzing a character (t_{fuzz})
  - Threshold probability (p_t)
- While generating,
  - Sample the model to get next character *c* and its probability *p(c)*
  - If *p(c)* is greater than the threshold probability, replace *c* with *c’* where *c’* is the character least likely in the learnt distribution
  - This happens only of a random function p_{fuzz} returns greater than the probability of fuzzing a character t_{fuzz}

SampleFuzz

- Characteristic
  - Introduce anomalies only in places where the model is *highly confident* in the next character
  - Generated object length bounded by MAXLEN
    - Algorithm itself not guaranteed to terminate, but made to terminate after MAXLEN

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

```
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
```

Training

- Seq2seq model
  - unsupervised
- Epochs
  - Divided up into five different number of epochs: 10, 20, 30, 40, 50
  - Each epoch takes about 12 minutes
  - 50 epochs → ~10 hours
- LSTM model (variant of RNN)
  - 2 hidden layers
  - 128 hidden states within a layer

Test environment

- Edge browser
  - Self-contained single-processor test-driver
  - Takes PDF file, executes PDF parser within Microsoft Edge browser
  - Upon encountering an error, prints the error message in the log
- Machine
  - 4-core
  - 64-bit
  - 20G RAM
  - Windows 10

Considerations

- Coverage
  - Union of the instruction coverage for all test cases
- Pass rate
  - If no error log, *pass*. Otherwise, *fail*
  - *Pass* means the generated PDF document is well-formed
  - Helps in estimating the quality of the learning
- Bugs
  - Each tests are run under AppVerifier to catch memory corruption bugs with low overhead
  - Used widely while fuzzing in Windows environment

Training Data

- 63000 non-binary PDF object out of 534 PDF files, provided by Windows fuzzing team
  - PDF files previously used for Windows Edge PDF parser fuzzing
- 534 files
  - Result of seed minimization
  - Larger set of PDF files

Edge PDF parser

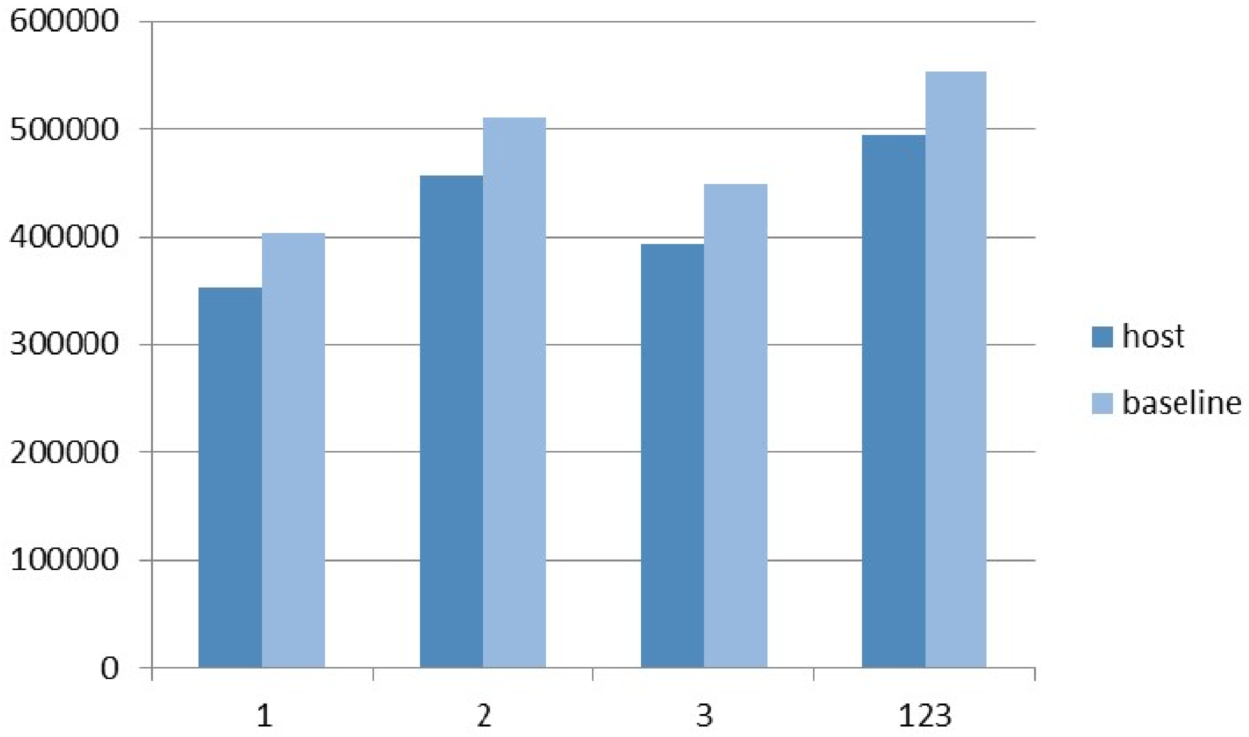
- Only processes full PDF documents (not objects)
- Workaround
  - Simple program to append the generated PDF objects to a well-formed PDF documents (*host*)
- Steps
  - Find the last trailer, and gather information
  - Add a new PDF body

Baseline Coverage

- Coverage without fuzzing
  - Selected 1000 out of the sample 63000 objects and measured the instruction coverage of the parser
  - Used as the baseline coverage
- Can a newly inserted objects interfere with the previous objects?
  - Could influence the resulting coverage

Testing interference

- Select smallest 3 PDF files out of the 534 set
    - host1~host3
    - Coverage ranges from 353,327~457,464 unique instructions
    - Union 494,652 instructions
    - Each host covers some unique instructions not covered by the other two
    - Smallest file doesn’t mean smallest coverage.
- Combine 3 files with 1000 selected baseline objects to create 3 * 1000 = 3000 files



- 90% of instructions are covered by host
- 1000 PDF files took ~90 minutes to be processed by the Edge parser

Learning

- Trained with 10~50 epochs
- After training, generate 1000 new objects
  - Compared with 63000 existing samples with no exact match
  - Generation method
    - Sample
    - SampleSpace

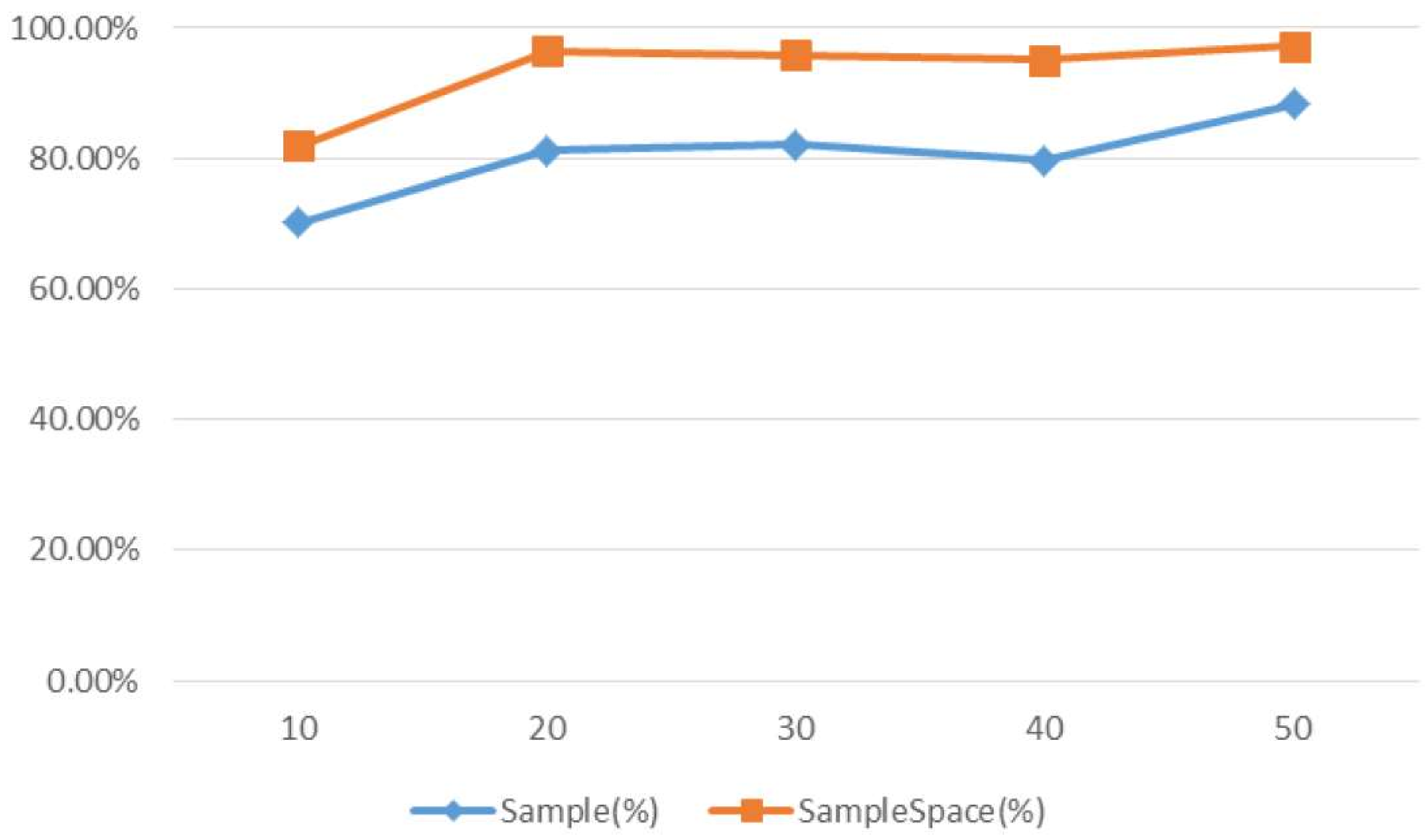
Pass Rate

- SamplePace pass rate significantly better than Sample

- After 10 epochs Sample already at 70% pass rate → learning is of good quality

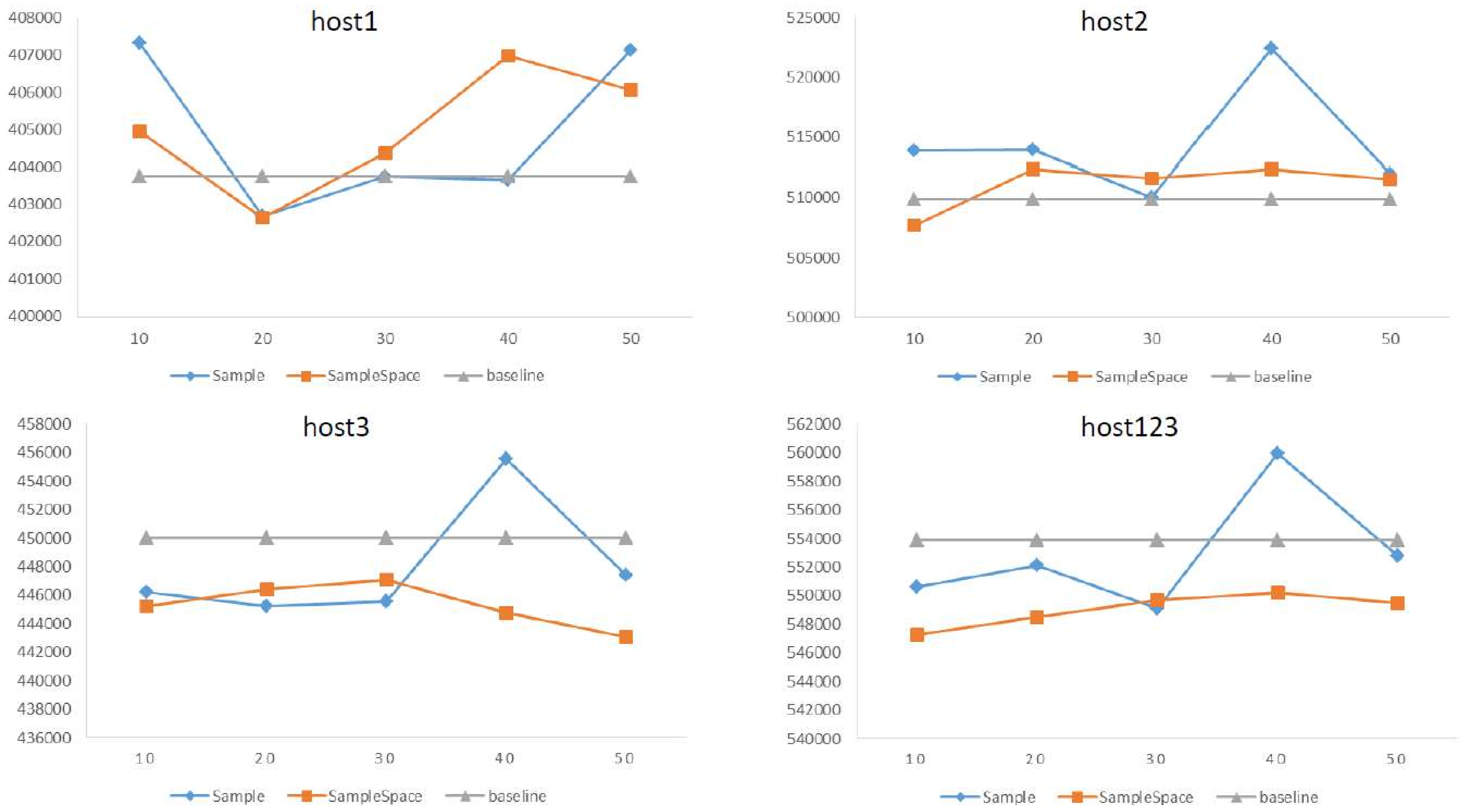
- More epochs → higher pass rate, more time consumption

- Best pass rate: 97% with SampleSpace and 50 epochs


100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
10
20
30
40
50
Sample(%)
SampleSpace(%)

Coverage

- Combined with hosts (mentioned before) to measure coverage
- Depends heavily on host
- Coverage change over epochs varies with host
- Best coverage tended to happen at Sample 40-epochs
    - Baseline123 is second best behind Sample 40-epochs
    - Best with SampleSpace is also 40-epochs


host1
408000
407000
406000
405000
404000
403000
402000
401000
400000
10
20
30
40
50
Sample
SampleSpace
baseline
host2
525000
520000
515000
510000
505000
500000
10
20
30
40
50
Sample
SampleSpace
baseline
host3
458000
456000
454000
452000
450000
448000
446000
444000
442000
440000
438000
436000
10
20
30
40
50
Sample
SampleSpace
baseline
host123
562000
560000
558000
556000
554000
552000
550000
548000
546000
544000
542000
540000
10
20
30
40
50
Sample
SampleSpace
baseline

Comparing 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

- Table above indicates how many unique instructions the row method generated objects cover that are not covered by the column method

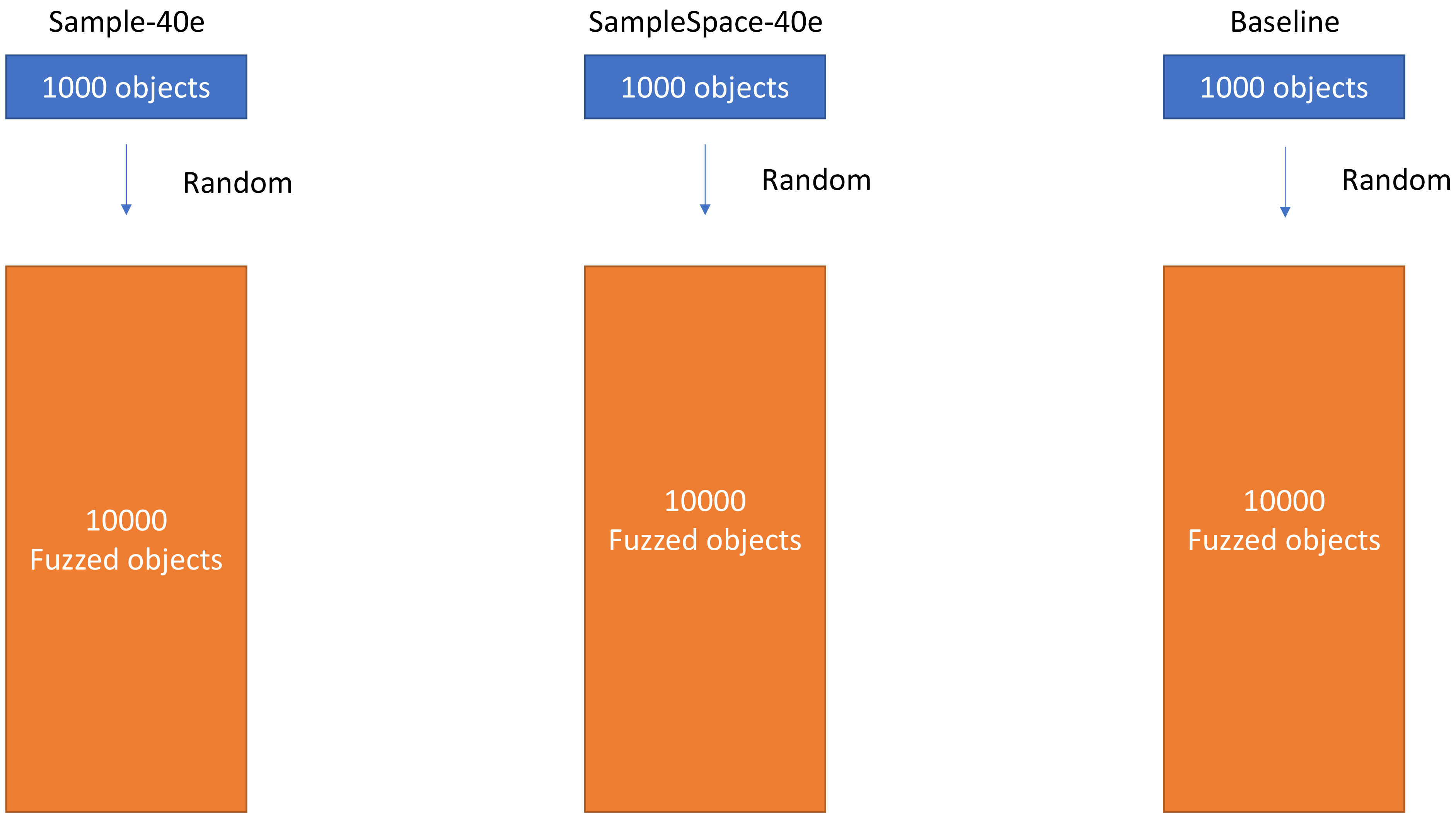
Comparing Coverage sets

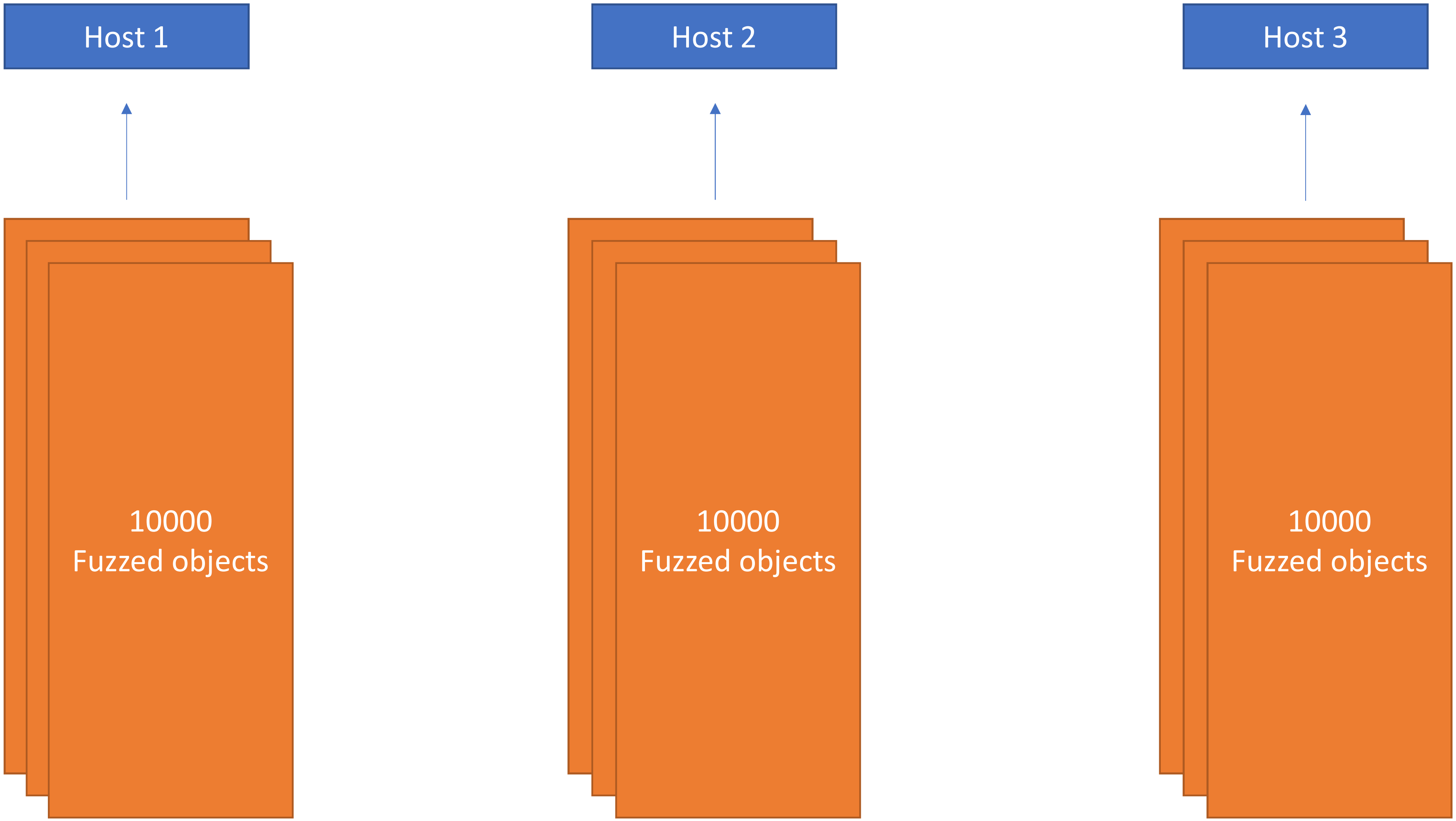
- Sample-40e and SampleSpace-40e have way more instructions in common than they differ (10,799 and 1,680), with Sample-40e having better coverage than SampleSpace-40e.

- SampleSpace-40e is incomparable with baseline123: it has 3,393 more instructions but also 6,514 missing instructions.

Combining Learning and Fuzzing

- Random, a widely used blackbox fuzzing algorithm
  - Randomly picks a position of a file, replace a byte with random bytes
  - Fuzz factor of 100: length of file / 100 will be the average number of bytes replaced
- Use Random to generate 10 random variants for each of the generated object with Sample-40e, SampleSpace-40e, and baseline
  - (result: 30000 files for each of the methods)
- For extra comparison, Sample-10K is added to the list (10,000 objects generated by Sample-40e)
- Finally, SampleFuzz discusses before is added to the list
  - Learnt distribution of 40-epochs RNN model with $t_{fuzz} = 0.9$ and $p_t = 0.9$


Sample-40e
1000 objects
Random
10000
Fuzzed objects
SampleSpace-40e
1000 objects
Random
10000
Fuzzed objects
Baseline
1000 objects
Random
10000
Fuzzed objects


Host 1
Host 2
Host 3
10000
Fuzzed objects
10000
Fuzzed objects
10000
Fuzzed objects

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%

Analysis of the result

- After applying Random on objects generated with Sample, SampleSpace and baseline, coverage goes up while the pass rate goes down to below 50%
- 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%
- 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.
    - SampleFuzz covers 2622 more instructions than Sample+Random
    - Sample+Random covers 1952 more instructions than SampleFuzz

Coverage and Pass Rate

- As the coverage increased, the pass rate decreased

- Intuitive explanation:
  - Pure-learning algorithm with nearly perfect pass rate (SampleSpace) almost only generates well-formed objects and covers less error handling code
  - Noise-making algorithm with decent pass rate (Sample) not only generates well-formed objects, but also generates some ill-formed objects to exercise error handling code
  - Applying random fuzzing on the generated objects lowered the pass rate even more but increased coverage

SampleFuzz

- Seemed to be the best option so far
- Pass rate around 65% ~ 70%
  - High enough to generate enough well-formed objects
  - Low enough to allow execution of error handling codes

Bugs

- Fuzzing effectiveness metric
- No bugs were found
  - Edge parser had been thoroughly fuzzed for months with other fuzzers
  - All the bugs found had been fixed
- However, a stack overflow bug was found with larger training data
  - Sample+Random
  - 100000 objects, 300000 PDF files
  - Took 5 days
  - Regular-sized PDF file triggering unexpected recursion
  - Later confirmed and fixed

Conclusion

- First attempt at neural-network based statistical learning of grammars to generate input grammars
- Devised several sampling techniques to generate new PDF objects from learnt distribution
- Able to generate well-formed objects as well as ill-formed objects for fuzzing and code coverage
- Learning & Fuzzing
    - Learning wants to capture the pattern or structure
    - Fuzzing wants to diverge from that pattern

Future work

- Entire PDF file rather than objects

- Reinforcement learning of seq2seq with coverage feedback from the application (parser) → guide the learning towards more coverage