NEUZZ: Efficient Fuzzing with Neural Program Smoothing

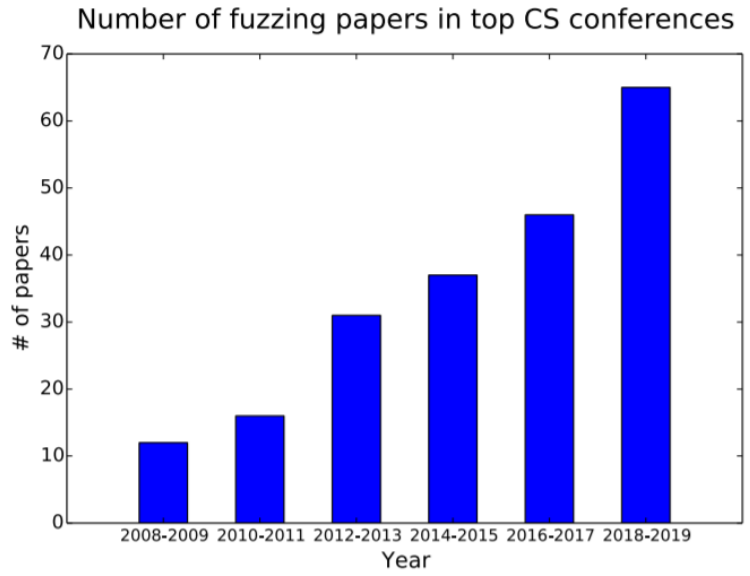
Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana

Columbia University

S&P 2019

Problems and solutions :

- Problem : How to efficiently detect bugs in a software ?
- Proposed solution : Neuzz , a fuzzer that uses neural program smoothing to create inputs which will trigger bugs
- Results : Successfully discovered new bugs in popular programs and outperformed state-of-art fuzzers.



Contributions of the paper :

- Identification of the significance of program smoothing for adopting efficient gradient-guided techniques for fuzzing
- New way of program smoothing :
  - Usage of a neural network to represent the program's behavior
  - Refinement technique to make the surrogate network more efficient
- Usage of gradients of the neural network model to efficiently generate new program inputs to find bugs in the target program
- Design, implementation and evaluation of Neuzz

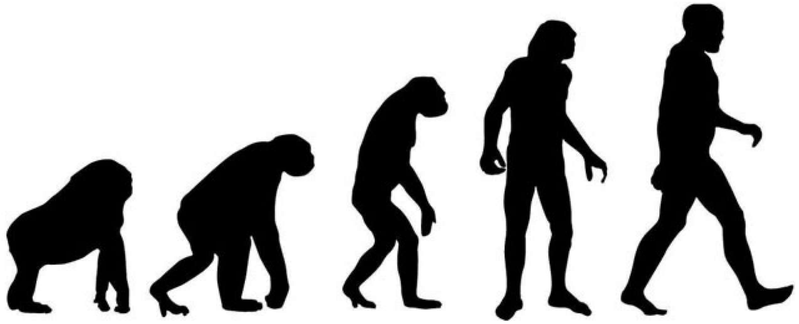
Meaning of the paper :

- The authors of created a framework Neuzz which uses a neural network to mimic the target program's behavior.

- They can also generate inputs which can trigger hard to find bugs in many programs.

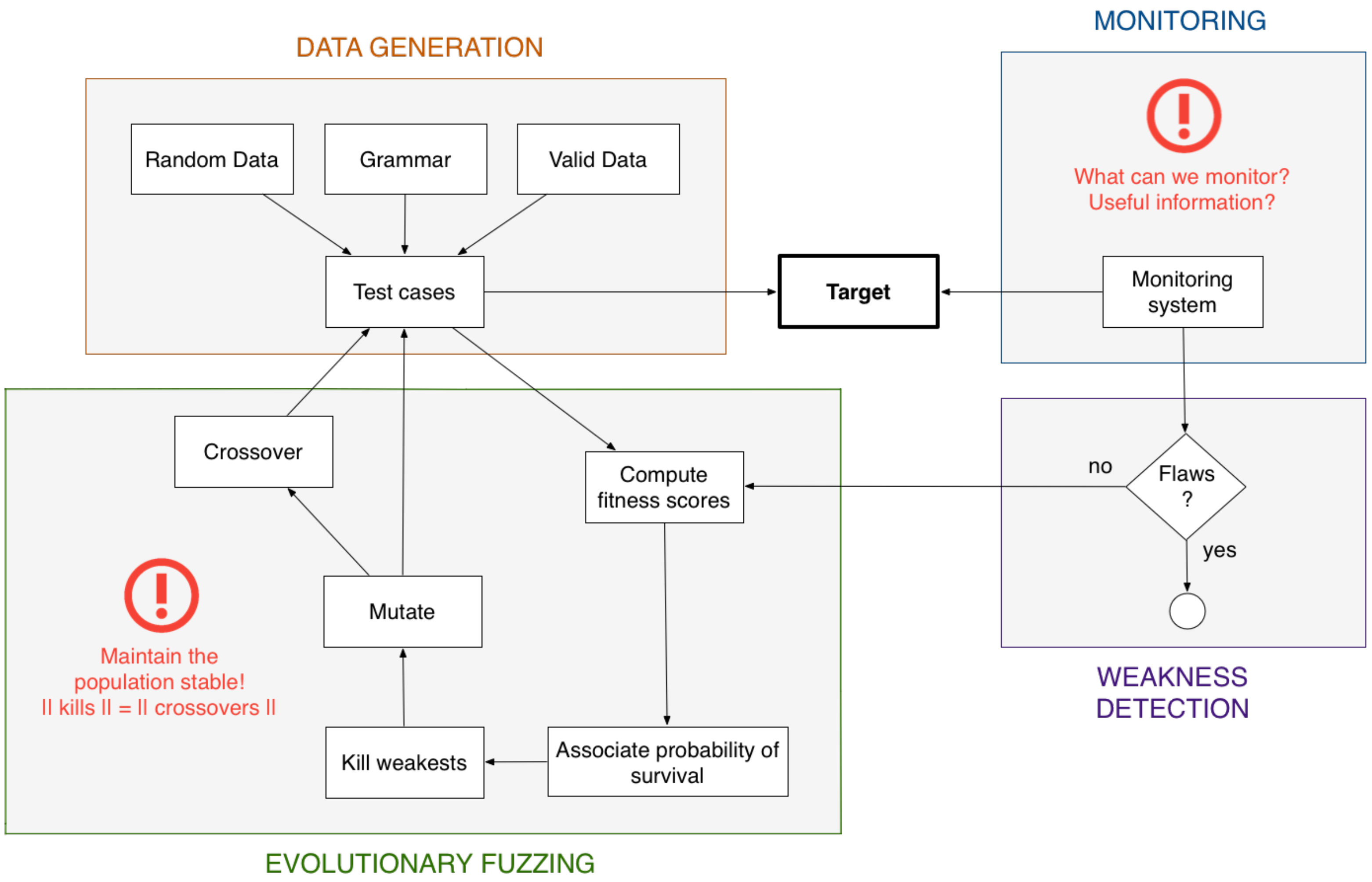
Evolutionary Fuzzing :

- Technique inspired by evolutionary biology
- Aims at converging towards the discovery weaknesses
- Uses genetic algorithms to produce successive generations of test cases
- Test cases creation based on classic methods + feedback from target



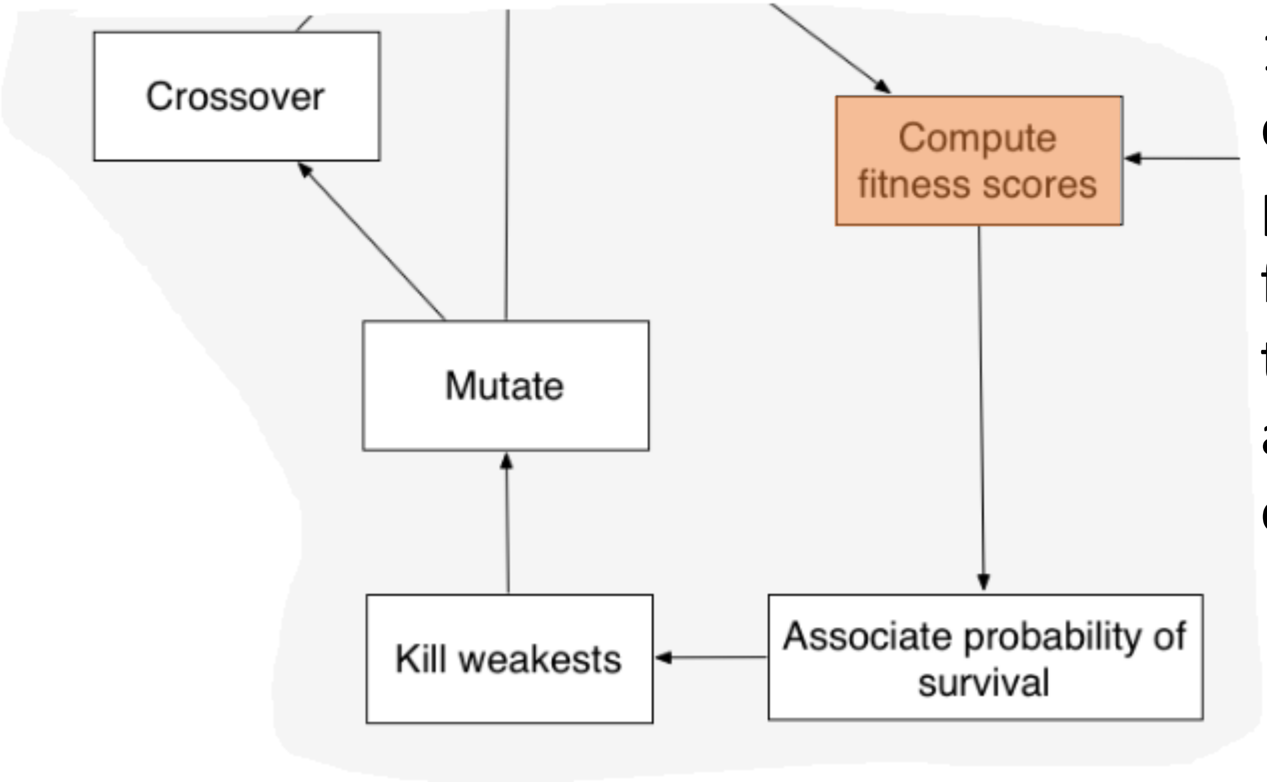
Example : human evolution

Evolutionary Fuzzing :



Evolutionary Fuzzing :

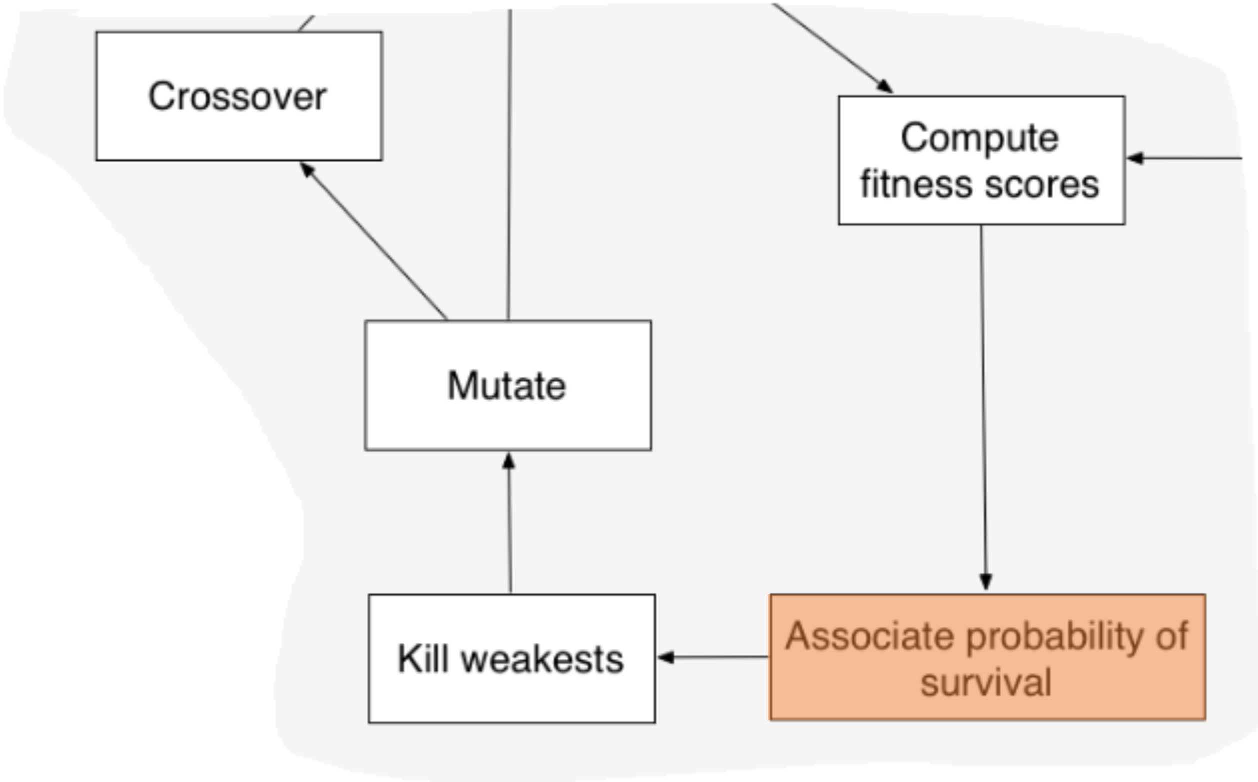
Steps to spawn new generation following the initial population :



1) Fitness score computation :
each test case, member of the current population, is given a score which is function of some metrics (impact on the target, diversity, and so on), which are calculated by the entity in charge of the monitoring aspects.

Evolutionary Fuzzing :

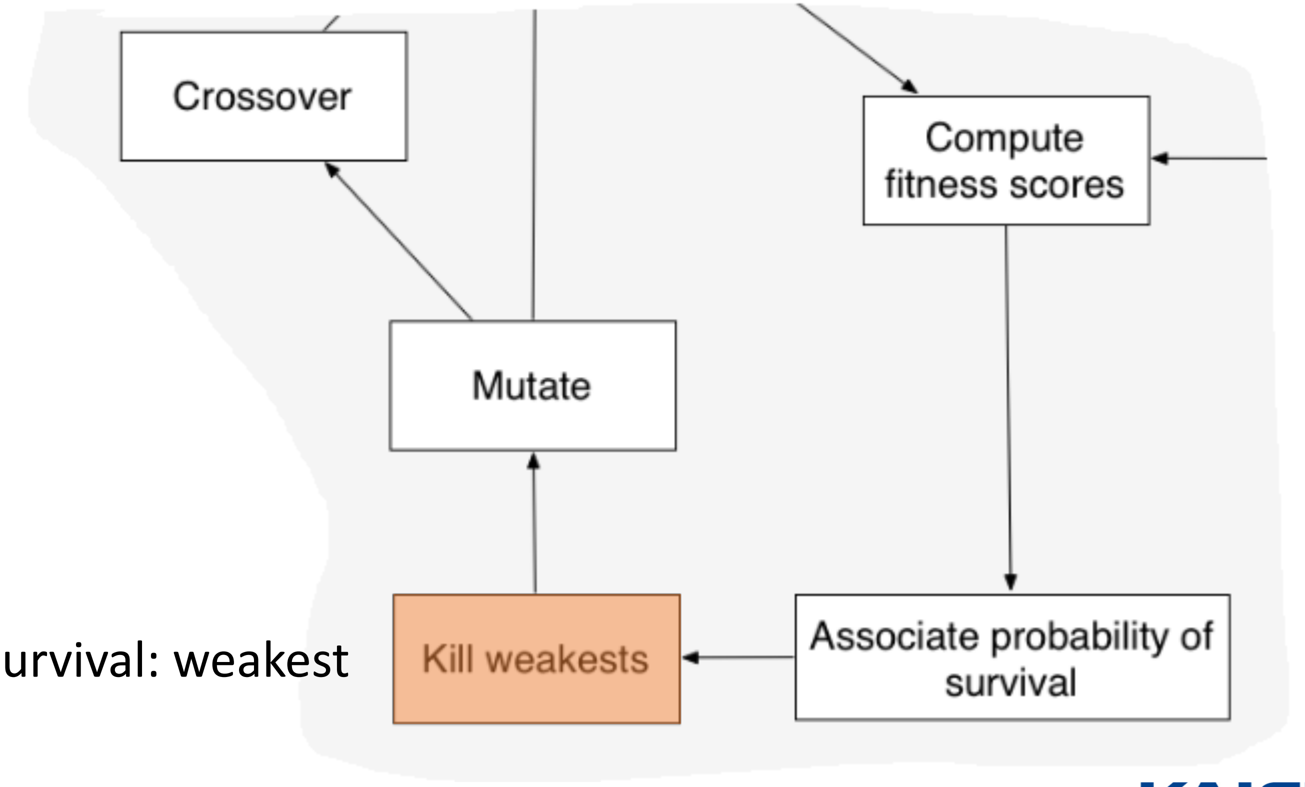
Steps to spawn new generation following the initial population :



2) Probabilities of survival association: depending on the score computed in the previous step, a probability of survival is associated to each individual.

Evolutionary Fuzzing :

Steps to spawn new generation following the initial population :

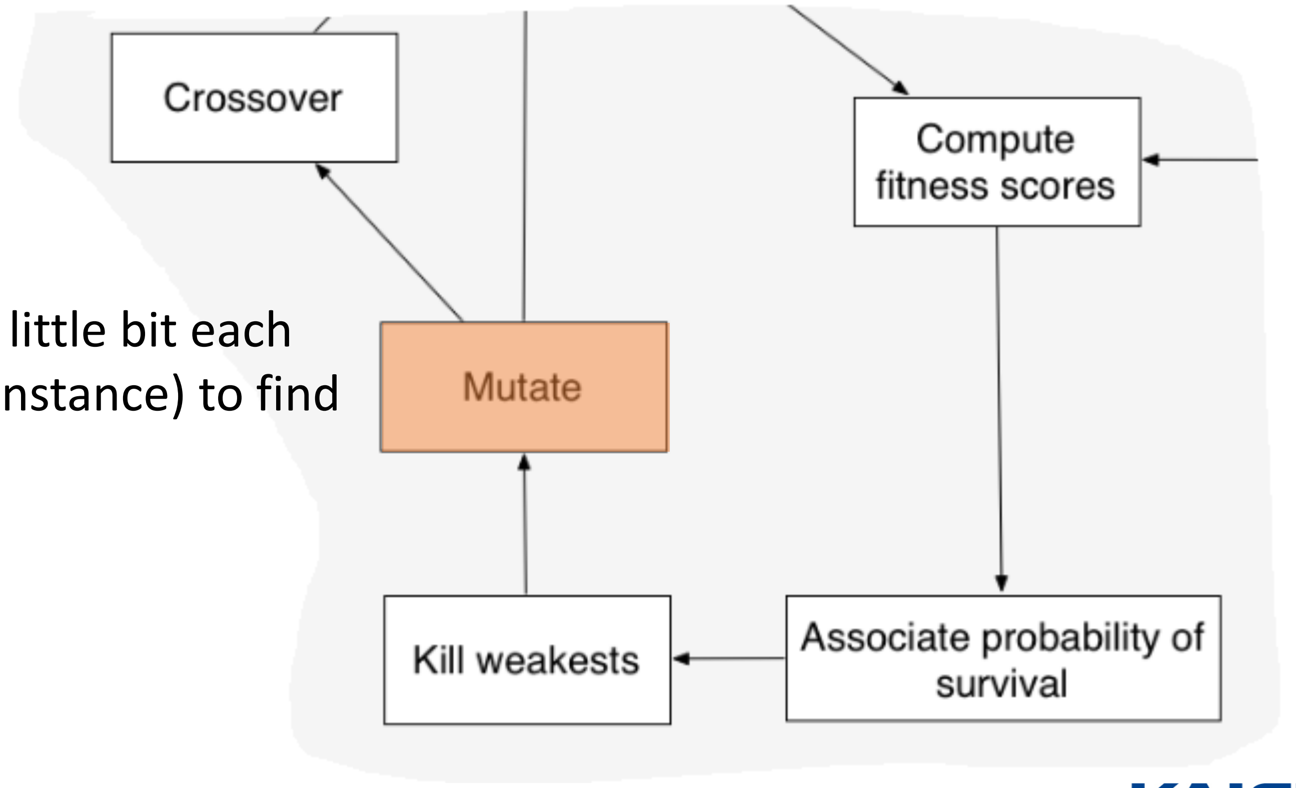


3) Dice are rolled:
using the probabilities of survival: weakest test cases are killed.

Evolutionary Fuzzing :

Steps to spawn new generation following the initial population :

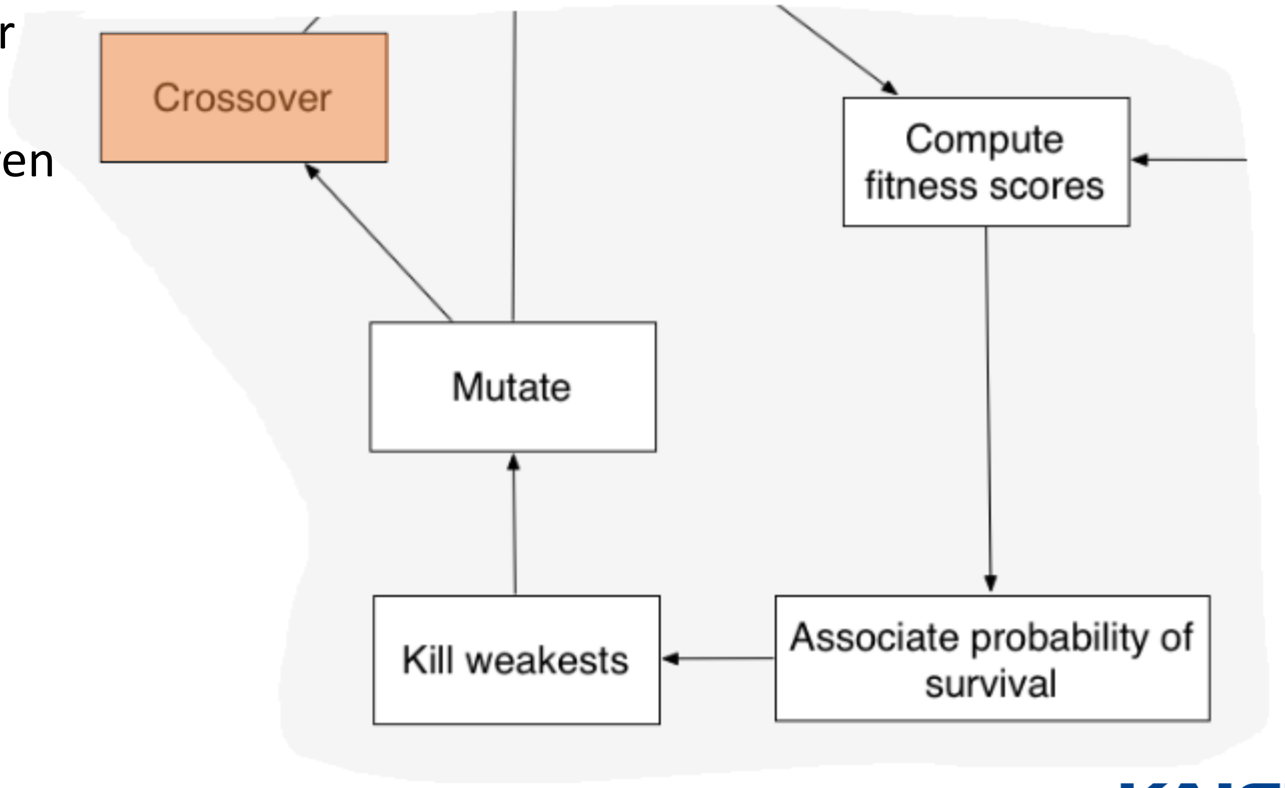
4) Mutation: aims to modify a little bit each individuals (flip some bits for instance) to find local optimums.



Evolutionary Fuzzing :

Steps to spawn new generation following the initial population :

5) Cross-over:
on the contrary, involves huge changes in order to find other optimums. It combines the test cases that are still alive in order to generate even better solutions. This process is also used to compensate the kills done in step 3.



Evolutionary Fuzzing :

- Advantage :
  - Easy to implement
- Disadvantage :
  - Random mutation not effective
  - Getting stuck in long sequence of meaningless mutations makes it inefficient

Optimization basics :

a) Function smoothness and optimization :

- Optimization algorithms : begin with an initial value x and try to find better solutions by iterating

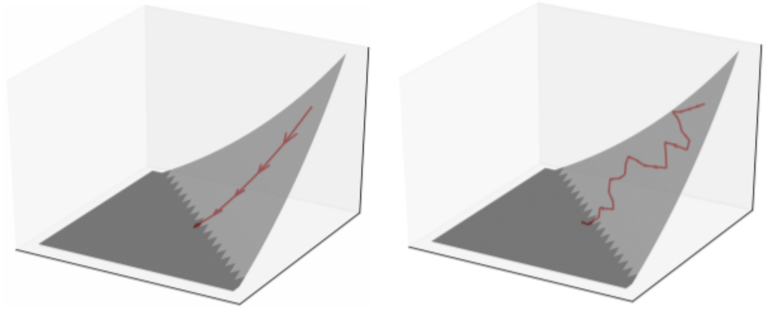
➔ What strategy does it uses to move from an initial input to the other ?

- Evaluate the objective function , the constraint functions and gradient/higher-order derivatives

- Example :
  - Gradient-descent
  - Linear search methods : $x_{k+1} = x_k + \alpha_k d_k$
    - d_k is the search direction
    - $\alpha_k > 0$ is chosen so that $f(x+1) < f(x_k)$

Optimization basics :

a) Convexity and gradient-guided optimization :
- Convex : gradient-guided optimization highly efficient
- Non-convex : may get stuck in local optimal solutions but easy fix with simple heuristics



(a) gradient descent (b) evolutionary algorithm

Optimization basics :

c) Fuzzing as unconstrained optimization :

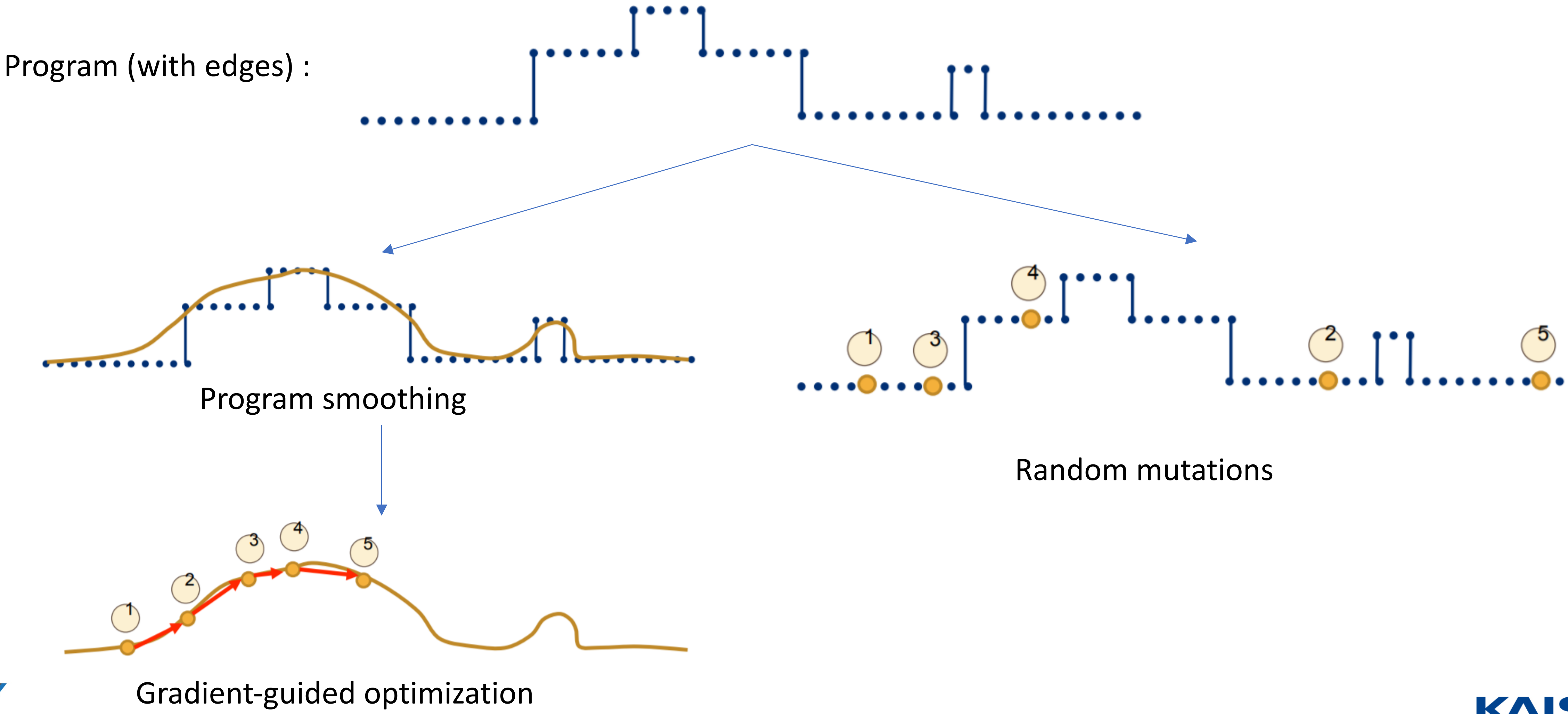
x ⇨ a program input $x \in X$
$G(x)$ ⇨ edge coverage of input x
$C(X)$ ⇨ generate *K* inputs from input space X

$$\textbf{Maximize} \sum_{x \in C(X)} G(x)$$

Find **C(X)** that can maximize total number of edges

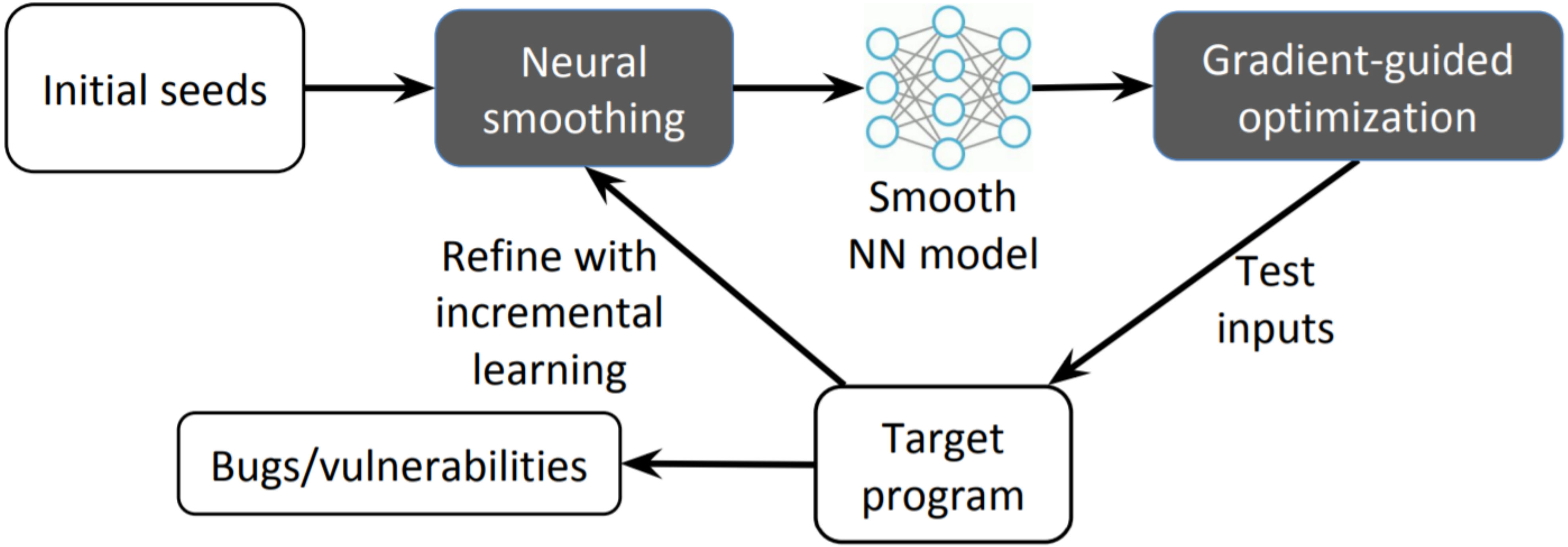
- Most existing fuzzers use evolutionary techniques

Optimization basics :



Web Security & Privacy Lab
KAIST

Overview of the approach

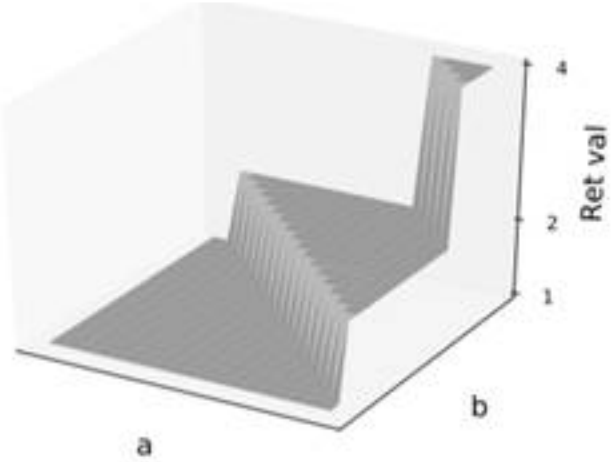


Program smoothing :

- A motivating example :

Non-linear exponential function
with a switch like code pattern

```
z = pow(3, a+b);
if(z < 1){
   return 1;
}
else if(z < 2){
   //vulnerability
   return 2;
}
else if(z < 4){
   return 4;
}
```



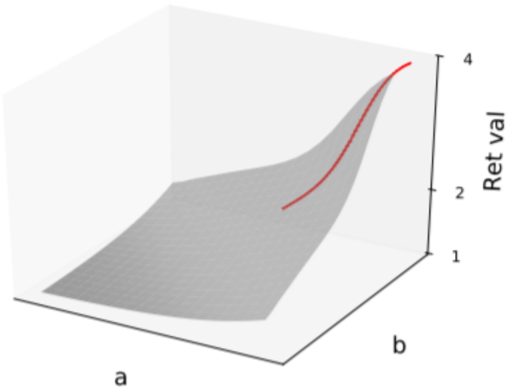
Program smoothing :

- Evolutionary fuzzers like AFL explored the branches in line 2 and 9
- Failed to explore line 5
- Training dataset for the neural network are values that only activated 2 edges
- NN cannot model correctly the program

```
1  z = pow(3, a+b);
2  if(z < 1){
3     return 1;
4  }
5  else if(z < 2){
6     //vulnerability
7     return 2;
8  }
9  else if(z < 4){
10    return 4;
11 }
```

Program smoothing :

- Representation by the neural network model :

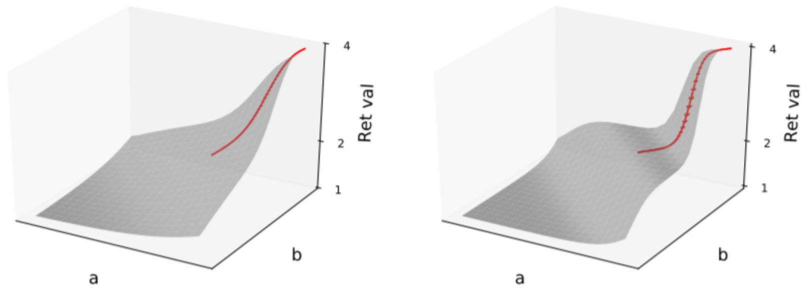


(b) NN smoothing

- How can we refine this representation ?

Program smoothing :

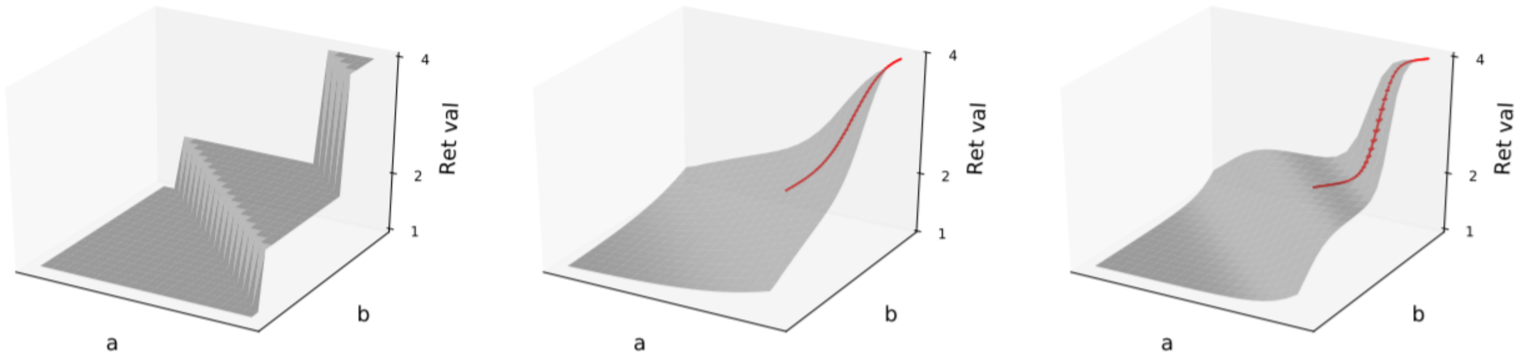
- Perform more effective gradient-guided optimization to find the desired values of a and b
- Incrementally refine the model by retraining it with the new inputs until the desired branch is executed



(b) NN smoothing (c) NN smoothing + refining

- Better representation of the program's behavior

Program smoothing :



(a) Original (b) NN smoothing (c) NN smoothing + refining

```
z = pow(3, a+b);
if(z < 1){
  return 1;
}
else if(z < 2){
  //vulnerability
  return 2;
}
else if(z < 4){
  return 4;
}
```

Program smoothing :

- Important for making the gradient-optimization efficient
- Avoid getting stuck in local extremums
- Smoothing of a discontinuous function = convolution operation between f and a smooth mask function g
- 2 types of program smoothing :
  - Black box :
    - Picks discrete samples from the input space of f
    - Computes convolution numerically using these samples
  - White box :
    - Analyzes statements and instructions
    - Summarizes using symbolic analysis and abstract interpretation

Program smoothing :

- Important for making the gradient-optimization efficient
- Avoid getting stuck in local extremums
- Smoothing of a discontinuous function = convolution operation between f and a smooth mask function g
- 2 types of program smoothing :
  - Black box :
    - Picks discrete samples from the input space of f
    - Computes convolution numerically using these samples
  - White box :
    - Analyzes statements and instructions
    - Summarizes using symbolic analysis and abstract interpretation

Large approximation errors

Expansive in computational cost and time consuming

Gray box approach used in NEUZZ

Web Security & Privacy Lab

KAIST

Neural program smoothing :

- Usage of surrogate neural networks to learn and iteratively refine smooth approximation of the target program
- Advantages of using a NN :
  - can model non-linear and non-convex behaviors
  - Efficient for computing gradients
- For training, inputs have a fixed size
- Input = input bytes, Outputs = edge coverage bitmap
- Fully connected Neural network to approximate the program
- Approach agnostic to the source of the training data

Training Data preprocessing :

- Some edges might always be triggered
- To counter this problem ➔ Dimensionality reduction :
  - Merging edges appearing together into one edge
  - Consider only the edges that have been activated at least once
- Reduced the dataset from 65546 to 4000

Gradient-guided optimization :

- Gradient indicates critical parts of the input
- Critical parts of the input affect program branches
- Focus mutations on the critical parts of the input to improve the edge coverage

Algorithm 1 Gradient-guided mutation

Input: $seed \leftarrow$ initial seed
$iter \leftarrow$ number of iterations
$k \leftarrow$ parameter for picking top-k critical bytes for mutation
$g \leftarrow$ computed gradient of seed

for $i = 1$ to $iter$ **do**
  $locations \leftarrow top(g, k_i)$
  for $m = 1$ to 255 **do**
    for $loc \in locations$ **do**
      $v \leftarrow seed[loc] + m * sign(g[loc])$
      $v \leftarrow clip(v, 0, 255)$
      $gen\_mutate(seed, loc, v)$
    for $loc \in locations$ **do**
      $v \leftarrow seed[loc] - m * sign(g[loc])$
      $v \leftarrow clip(v, 0, 255)$
      $gen\_mutate(seed, loc, v)$

Implementation :

a) Neural Network architecture :

- Three fully-connected layers (1 hidden layer)
- Hidden layer uses ReLU activation
- Activation function of the output = sigmoid
- Trained for 50 epochs with the whole dataset
- Training time :
  - With GPU (GTX 1080ti) : 2 minutes
  - With CPU (I7-7700K) : 20 minutes
- Test accuracy : 95%

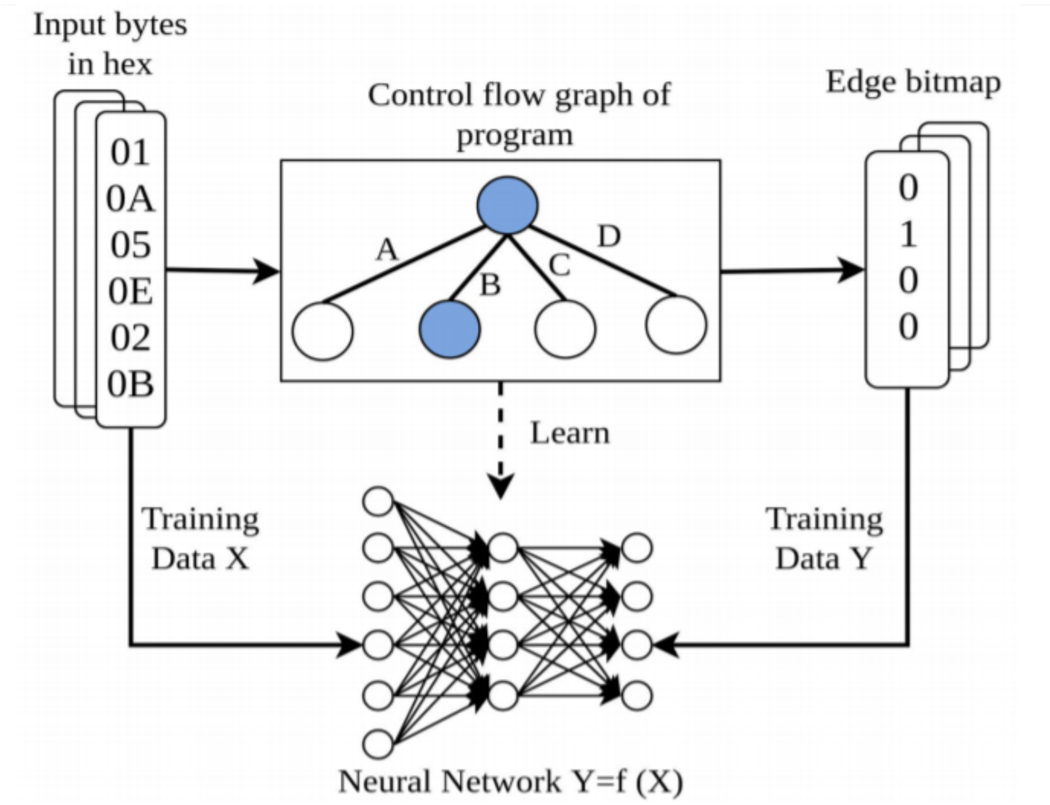
Implementation :

b) Training data collection :

- Run AFL-2.52b for one hour on each program
- 2K inputs collected per program
- Split ratio 5:1
- Threshold file size : 10KB

c) Mutation and retraining :

- Test the target program with 1M mutated inputs



Model parameter selection :

- Edge coverage achieved by mutations generated in different iterations

Programs	Iteration i		
	7	10	11
readelf -a	1,678	**1,800**	1,529
libjpeg	**107**	89	93
libxml	161	**256**	174
mupdf	**294**	266	266

- k=2 , number of critical bytes to be mutated in the initial seed

Model parameter selection :

- Edge coverage comparison of 1M mutations with different NN models

Programs	1 hidden layer		3 hidden layers	
	n=4096	n=8192	n=4096	n=8192
readelf -a	1,800	1,658	1,714	1,584
libjpeg	89	57	80	79
libxml	256	172	140	99
mupdf	260	94	82	88

Evaluation :

- 10 real world programs
- Lava-M and DARPA CGC datasets
- Comparison with 10 state-of-the-art fuzzers
- Comparison with RNN-based fuzzers
- Performance of different model choices

Evaluation :

- Experimental setup :
  - ➢ Run AFL for 1 hour to generate the initial seed corpus
  - ➢ Run each fuzzer for a fixed time
  - ➢ Compare edge coverage and number of bugs found

	Time
10 real world programs	24 hours
LAVA-M	5 hours
CGC	6 hours

Evaluation :

- Studied fuzzers :

Fuzzer	Technical Description
AFL [88]	evolutionary search
AFLFast [11]	evolutionary + markov-model-based search
Driller [82]‡	evolutionary + concolic execution
VUzzer [73]	evolutionary + dynamic-taint-guided search
KleeFL [32]	evolutionary + seeds generated by symbolic execution
AFL-laf-intel [47]	evolutionary + transformed compare instruction
RNNfuzzer [72]	evolutionary + RNN-guided mutation filter
Steelix [55]†	evolutionary + instrumented comparison instruction
T-fuzz [69]†	evolutionary + program transformation
Angora [22]†	evolutionary + dynamic-taint-guided + coordinate descent + type inference

Evaluation :

- Studied programs :

Programs		# Lines	NEUZZ train (s)	AFL coverage
Class	Name			1 hour
binutils-2.30 ELF Parser	readelf -a	21,647	108	4,490
	nm -C	53,457	63	3,779
	objdump -D	72,955	104	5,196
	size	52,991	52	2,578
	strip	56,330	55	5,789
TTF	harfbuzz-1.7.6	9,853	94	82,79
JPEG	libjpeg-9c	8,857	56	3,117
PDF	mupdf-1.12.0	123,562	62	4,624
XML	libxml2-2.9.7	73,920	95	6,691
Zip	zlib-1.2.11	1,893	65	1,479

Results :

- Edge coverage :

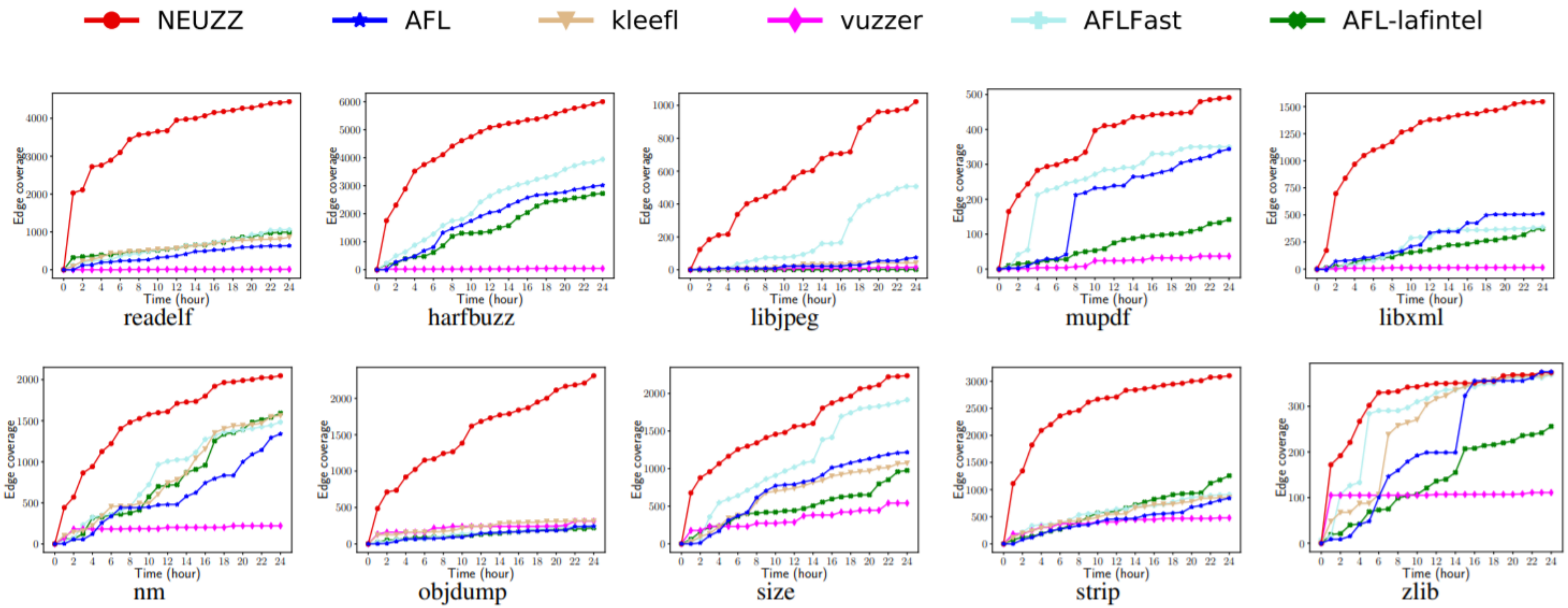


Fig. 4: **The edge coverage of different fuzzers running for 24 hours.**

Neuzz achieved on average 3x more edge coverage than other fuzzers

Results :

- Edge coverage :

Programs	NEUZZ	AFL	AFLFast	VUzzer	KleeFL	AFL-laf-intel
readelf -a	4,942	746	1,073	12	968	1,023
nm -C	2,056	1,418	1,503	221	1,614	1,445
objdump -D	2,318	257	263	307	328	221
size	2,262	1,236	1,924	541	1,091	976
strip	3,177	856	960	478	869	1,257
libjpeg	1,022	94	651	60	67	2
libxml	1,596	517	392	16	n/a†	370
mupdf	487	370	371	38	n/a	142
zlib	376	374	371	15	362	256
harfbuzz	6,081	3,255	4,021	111	n/a	2,724

†indicates cases where Klee failed to run due to external dependencies

Web Security & Privacy Lab

KAIST

Results :

- Bug finding on 6 programs :

Programs	AFL	AFLFast	VUzzer	KleeFL	AFL-laf-intel	NEUZZ
Detected Bugs per Project						
readelf	4	5	5	3	4	16
nm	8	7	0	0	6	9
objdump	6	6	0	3	7	8
size	4	4	0	3	2	6
strip	7	5	2	5	7	20
libjpeg	0	0	0	0	0	1
Detected Bugs per Type						
out-of-memory	✓	✓	✗	✓	✓	✓
memory leak	✓	✓	✓	✓	✓	✓
assertion crash	✗	✓	✗	✗	✓	✓
interger overflow	✗	✗	✗	✗	✗	✓
heap overflow	✓	✗	✗	✗	✗	✓
Total	29	27	7	14	26	60

NEUZZ finds the most number of bugs.
It also found 31 unknown bugs.

Results :

- Bug finding on the LAVA-M datasets and CGC binaries :

	base64	md5sum	uniq	who
#Bugs	44	57	28	2,136
FUZZER	7	2	7	0
SES	9	0	0	18
VUzzer	17	1	27	50
Steelix	43	28	24	194
Angora	48	57	29	1,541
AFL-laf-intel	42	49	24	17
T-fuzz	43	49	26	63
NEUZZ	48	60	29	1,582

LAVA-M datasets

50 CGC binaries

Fuzzers	AFL	Driller	NEUZZ
Bugs	21	25	31

Neuzz outperforms all state-of-the-art fuzzers on LAVA-M and CGC

Results :

- Neuzz vs RNN-based fuzzer :

Programs	Edge Coverage			Training Time (sec)		
	NEUZZ	RNN	AFL	NEUZZ	RNN	AFL
readelf -a	1,800	215	213	108	2,224	NA
libjpeg	89	21	28	56	1,028	NA
libxml	256	38	19	95	2,642	NA
mupdf	260	70	32	62	848	NA

Neuzz outperforms the RNN-based fuzzers by a large margin
6x more edge coverage and 20x less training time

Results :

- Edge coverage by Neuzz using different machine learning models :

Programs	Linear Model	NN Model	NN + Incremental
readelf -a	1,723	1,800	2,020
libjpeg	63	89	159
libxml	117	256	297
mupdf	93	260	329

NN models outperform linear models and incremental learning makes NNs even more accurate over time.

Key takeaways of Neuzz :

- Use NN gradients to identify the critical locations of program inputs
- Focus mutations on the critical locations
- Minimize runtime overhead by using simple feed-forward neural networks
- Retrain the network incrementally to find new critical locations

Conclusion :

- Neuzz is an efficient fuzzer using a surrogate neural network to approximate the target program's behavior
- Gradient-guided techniques used to generate new inputs uncovered new bugs for the target program
- Neuzz outperformed 10 state-of-the-art fuzzers in terms of number in both edge coverage and bugs found

Questions ?