tutorial_pre-executed-automating-neural-cryptanalysis

March 27, 2025

0.1 Automating Neural Cryptanalysis

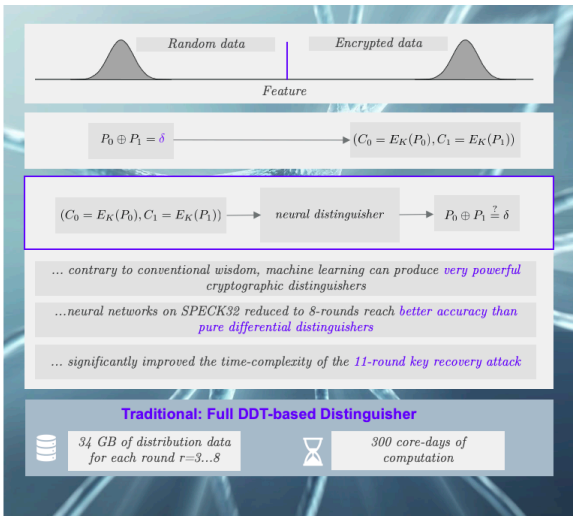
In this notebook, we review techniques to automatize neural cryptanalysis.

- Neural cryptanalysis overview
- Automating the input difference selection (in CLAASP)
- Automating the long and short range dependencies (in CLAASP)
- Training the whole pipeline (in CLAASP)
- Using your own (non-CLAASP) implementations in the framework

0.1.1 Neural Cryptanalysis Overview

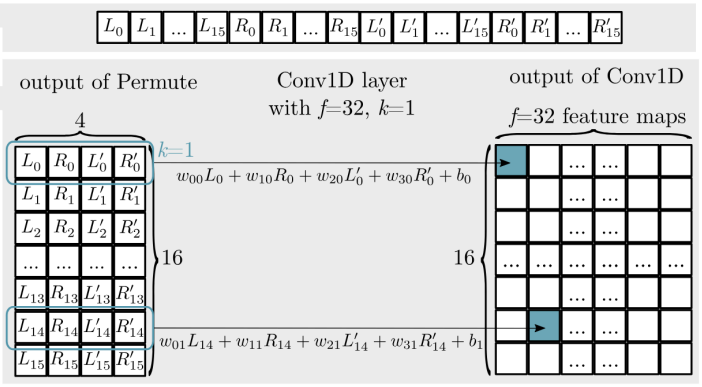
- A cryptanalytic distinguishing attack allows to distinguish encrypted data from random data.
- Differential Cryptanalysis: "... analyses the effect of particular differences in plaintext pairs..." [BS91].
- A Differential Neural Distinguisher is a deep neural network which performs the differential cryptanalytic task [Goh19]

[Goh19] Gohr, Aron. *"Improving Attacks on Round-Reduced Speck32/64 using Deep Learning."* Advances in Cryptology–CRYPTO 2019



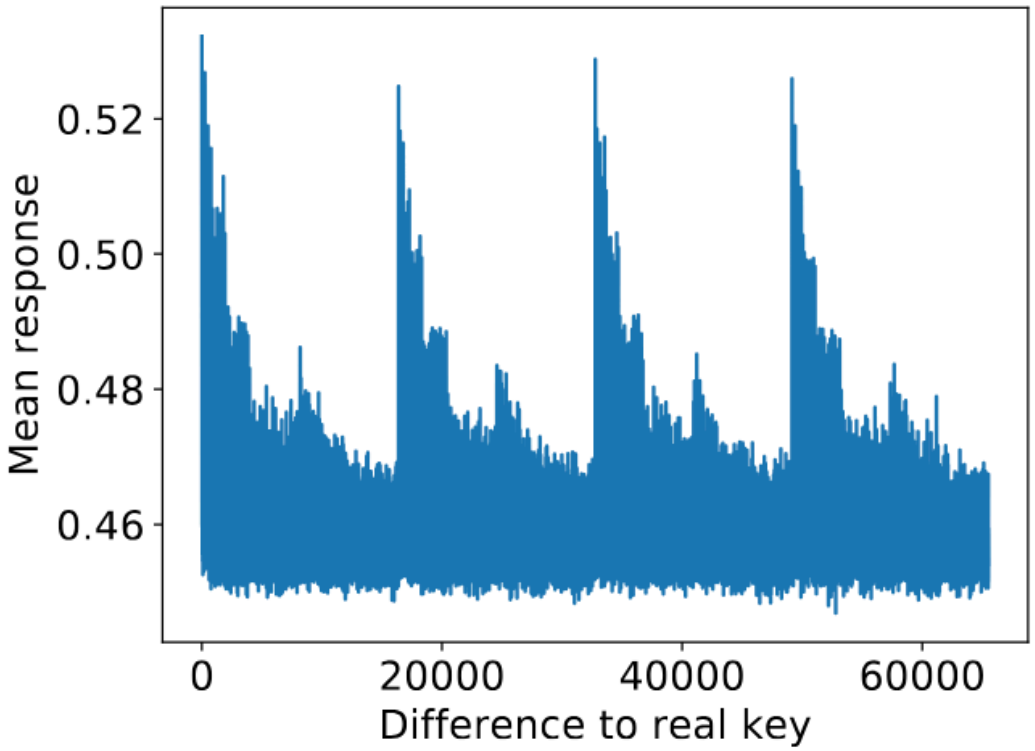
In neural cryptanalysis, introduced by A. Gohr at CRYPTO 2019, a neural network is used to distinguish pairs of ciphertexts that correspond to the encryption of related plaintexts under the same key from random.

```
Input(shape=(num_blocks*word_size*2,))
Reshape((2*num_blocks, word_size))
Permute((2,1))
# Initial convolution
Conv1D(f=32, k=1, 'L2')+BN+ReLU
# Residual blocks
Define 'shortcut'
for i in range(depth):
    Conv1D(f=32, k=3, 'L2')+BN+ReLU
    Conv1D(f=32, k=3, 'L2')+BN+ReLU
    Add 'shortcut'
    Define 'shortcut'
# Prediction head
Flatten()
Dense(64, 'L2')+BN+ReLU
Dense(64, 'L2')+BN+ReLU
Dense(1, 'sigmoid activation', 'L2')
```



These distinguishers have nice properties:

- They operate in a block-box setting
- Low data complexity (A single pair is sufficient to distinguish SPECK32-8 with over 50% accuracy)
- Multiple differential/differential linear 'for free'
- Possibly insightful wrong key response profile (see example below, which shows the variation in the output of the neural distinguisher for pairs of SPECK ciphertexts in which the last round is decrypted with a key that differs from the original key by different values)



Applications of neural cryptanalysis to different ciphers, or under different settings (e.g., multiple

pairs, rotional XOR, etc…) can be tedious, as the training is sensitive to a lot of hyperparameters that require significant finetuning and optimisation to obtain competitive results. In a FSE 2024 paper, we investigated the following question:

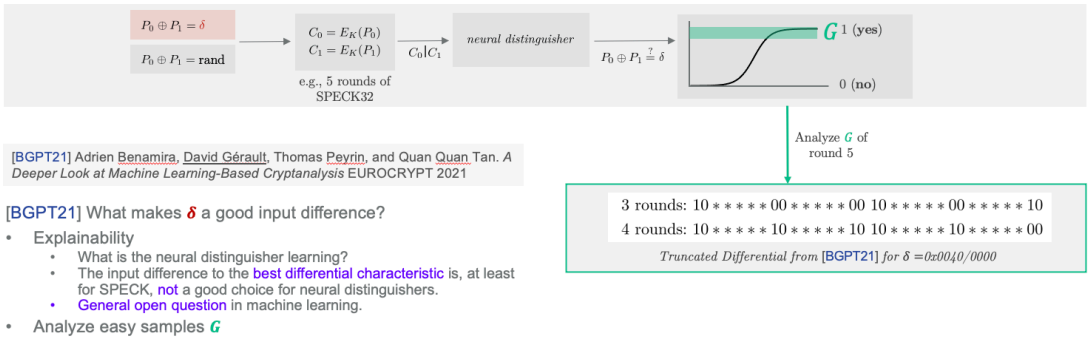
How to evaluate the resistance of a cipher to neural cryptanalysis automatically? The resulting framework, AutoND (https://github.com/Crypto-TII/AutoND), provides a generic, 'click-of-a-button' approach that eliminates the need for any human intervention besides providing an implementation of the cipher, while remaining competitive with dedicated approaches.

Primitive	**Arch.**	**Setting**^{S}	**Trn.**	**Val.**	**AutoND**	**Rounds**	**Acc.**	**Ref.**
SPECK32	MLP	2-1-δ-R	$2^{27.64}$	$2^{26.64}$	-	3^{*}	0.79	[YK21]
	ResNet	20-1-A-R	$2^{23.25}$	$2^{19.93}$	-	5	1	[BGPT21]
	ResNet	100-1-A-R	$2^{23.25}$	$2^{19.93}$	-	6	1	[BGPT21]
	ResNet	64-1-CT-R	$2^{23.25}$	$2^{19.93}$	-	8	0.939	[CSYY22]
	ResNet	2-1-CT-R	$2^{31.49}$	$2^{19.93}$	-	8	0.514	[Goh19b]
	DBitNet	*2-1-CT-R*	$2^{31.49}$	$2^{19.93}$	✓	*8*	*0.514*	*This work*
	ResNet	64-1-δ-R	$2^{28.25}$	/	-	8	0.564	[HRCF21]
SPECK64	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*8*	*0.537*	*This work*
	ResNet	128-1-δ-R	$2^{29.25}$	/	-	8	0.632	[HRCF21]
SPECK128	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*10*	*0.592*	*This work*
SIMON32	MLP	2-1-δ-R	2^{24}	$2^{27.64}$	-	5^{*}	0.570	[YK21]
	ResNet	4-3-CT-R	$2^{23.25}$	$2^{19.93}$	-	9	0.637	[SZM20]
	ResNet	2-1-CT-R	$2^{23.25}$	$2^{19.93}$	✓	9	0.661	[GLN23]
	ResNet	64-1-δ-R	$2^{28.25}$	/	-	10	0.611	[HRCF21]
	SENet	2-1-A-R	$2^{31.17}$	$2^{29.17}$	-	11	0.517	[BGL^{+}22]
	DBitNet	*2-1-CT-R*	$2^{31.49}$	$2^{19.93}$	✓	*11*	*0.518*	*This work*
	ResNet	2-1-CT-R	$2^{27.58}$	$2^{23.25}$	-	11	0.520	[GLN23]
	SE-ResNet	16-1-A-R	$2^{24.25}$	$2^{20.90}$	-	12	0.514	[LLS^{+}23]
SIMON64	ResNet	128-1-δ-R	$2^{29.25}$	/	-	12	0.695	[HRCF21]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*13*	*0.518*	*This work*
	SE-ResNet	16-1-A-R	$2^{24.25}$	$2^{20.90}$	-	14	0.519	[LLS^{+}23]
SIMON128	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*20*	*0.506*	*This work*
GIMLI	MLP	2-2-δ-D	$2^{17.6}$	$2^{14.3}$	-	8	0.510	[BBCD21]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*11*	*0.527*	*This work*
HIGHT	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*10*	*0.751*	*This work*
HIGHT^{RK}	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*14*	*0.563*	*This work*
KATAN	ResNet	2-1-δ-R	$2^{23.25}$	$2^{19.93}$	-	51	0.533	[LCLH22]
	ResNet	64-1-δ-R	$2^{23.25}$	$2^{19.93}$	-	59	0.575	[LCLH22]
	ResNet	2-1-CT-R	$2^{23.25}$	$2^{19.93}$	✓	66	0.505	[GLN23]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*69*	*0.505*	*This work*
PRESENT	ResNet	2-1-CT-R	$2^{23.25}$	$2^{19.93}$	✓	7	0.563	[GLN23]
	ResNet	8-1-CT-R	$2^{23.25}$	$2^{19.93}$	-	7	0.585	[CSYY22]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*9*	*0.509*	*This work*
TEA$^{*2,\ *3}$	MLP	*2-1-CT-R*$^{+}$	$2^{19.93}$	$2^{13.28}$	-	*4*	*0.545*	[BR21]
	DBitNet	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*5*	*0.563*	*This work*
XTEA^{*2}	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*5*	*0.598*	*This work*
LEA	*DBitNet*	*2-1-CT-R*	$2^{23.25}$	$2^{19.93}$	✓	*11*	*0.511*	*This work*

The AutoND framework automates 3 main aspects of neural cryptanalysis:

- Input difference selection
- Dataset constrution and reshaping
- Training pipeline

0.1.2 Input Difference Selection



In AutoND, we use an evolutionary algorithm to identify input difference that leads to high probability truncated differentials (using the probabilities of the individual output difference bits as a proxy metric, combined into a bias score). In CLAASP, a slightly modified version is implemented:

```
def find_good_input_difference_for_neural_distinguisher(self, difference_positions,
                                                        initial_population=32, number_of_genera
                                                        nb_samples=10 ** 3, previous_generatio
                                                        verbose=False):
    """
    Return good neural distinguisher input differences for a cipher, based on the AutoND pipel

    INPUT:

    - ``difference_positions`` -- **table of booleans**; one for each input to the cipher. Tru
      differences are allowed
    - ``initial_population`` -- **integer** (default: `32`); parameter of the evolutionary alg
    - ``number_of_generations`` -- **integer** (default: `50`); number of iterations of the ev
    - ``nb_samples`` -- **integer** (default: `1000`); number of samples for testing each inpu
    - ``previous_generation`` -- (default: `None`); optional: initial table of differences to
    - ``verbose`` -- **boolean** (default: `False`); verbosity
```

(The score formula a weighted sum of bias scores accross rounds)

Definition 2 (Bias score)**.** Let $E\colon \mathbb{F}_2^n \times \mathbb{F}_2^k \to \mathbb{F}_2^n$ be a block cipher, and $\delta \in \mathbb{F}_2^n$ be an input difference. The bias score for δ, $\tilde{b}^t(\delta)$ is the sum of the biases of each bit position j in the output difference, computed for t samples *i.e.*,

$$\tilde{b}^t(\delta) = \frac{1}{n} \cdot \sum_{j=0}^{n-1} \left| 0.5 - \frac{\sum_{i=0}^{t-1} (E_{K_i}(X_i) \oplus E_{K_i}(X_i \oplus \delta))_j}{t} \right|$$

[1]:
```
from claasp.ciphers.block_ciphers.speck_block_cipher import SpeckBlockCipher
from claasp.cipher_modules.neural_network_tests import NeuralNetworkTests
```

```
plain_bits=32
key_bits=64
speck = SpeckBlockCipher(block_bit_size=plain_bits, key_bit_size=key_bits)
tester = NeuralNetworkTests(speck)
diff, scores, highest_round = tester.
 ↪find_good_input_difference_for_neural_distinguisher([True, False], verbose =␣
 ↪True, number_of_generations=5) # random
```

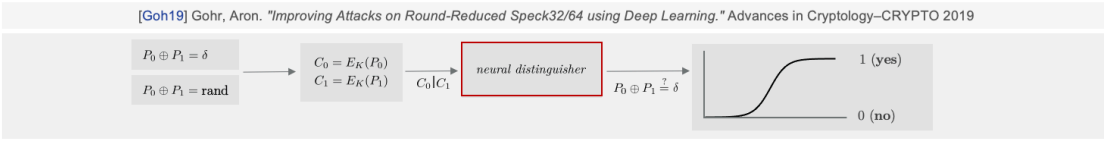
```
2025-03-15 08:00:32.105510: I tensorflow/core/util/port.cc:110] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-03-15 08:00:32.150127: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2025-03-15 08:00:32.861161: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT

Generation 0/5, 1602 nodes explored, 32 current, best is ['0x82591e0a',
'0x3761837c', '0xff0616fb', '0x7e60877f'] with [0.38703125 0.38984375 0.414
0.4701875 ]
Generation 1/5, 2157 nodes explored, 32 current, best is ['0x80', '0x8000000',
'0x20000000', '0x400000'] with [1.80709375 2.23053125 2.261375   3.4943125 ]
Generation 2/5, 2688 nodes explored, 32 current, best is ['0x8000000',
'0x20000000', '0x408000', '0x400000'] with [2.23053125 2.261375   3.03715625
3.4943125 ]
Generation 3/5, 3043 nodes explored, 32 current, best is ['0x200000',
'0x600000', '0x408000', '0x400000'] with [2.59509375 2.6073125  3.03715625
3.4943125 ]
Generation 4/5, 3260 nodes explored, 32 current, best is ['0x200000',
'0x600000', '0x408000', '0x400000'] with [2.59509375 2.6073125  3.03715625
3.4943125 ]
The highest reached round was 6
The best differences found by the optimizer are…
0x400000 , with score 3.4943125
0x408000 , with score 3.03715625
0x600000 , with score 2.6073125
0x200000 , with score 2.5950937499999998
0x8000 , with score 2.3938125
0x20000000 , with score 2.261375
0x8000000 , with score 2.2305312500000003
0x10200 , with score 2.2030000000000003
0x40000080 , with score 2.1684062500000003
0x10000 , with score 1.9855625
```

[2]:
```
delta_in = diff[-1]
hex(delta_in)
```

[2]:
```
'0x400000'
```

The input difference that receives the highest score, 0x400000, is indeed the one that was identified as optimal for 5 rounds in the initial work.

0.1.3 Neural Network Architecture and Pipeline



[Goh19] neural distinguisher

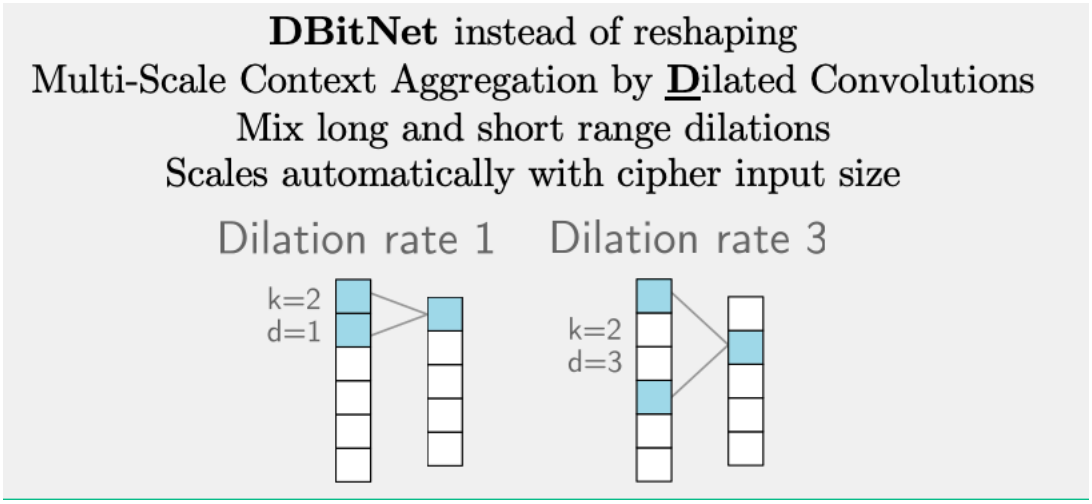
"Conventional" choices
residual network with a **fully connected** prediction head; **ReLU** activations in hidden layers; final layer **sigmoid**; **MSE** loss

Expert or heuristic choices
- 64-bit ciphertext pair is reshaped into 4 x 16-bit words
- Hyperparameters (22 in total, 8 with significant influence)
  - Number of filters; Cyclic learning rate schedule
- Training pipeline: Elaborate 8 round training scheme
  - Network depth-10 for SPECK32 rounds 5 and 6 vs depth-1 for rounds 7 and 8

Training an 8-Round Distinguisher For 8 rounds, the training scheme described above fails, i.e. the model does not learn to approximate any useful function. We still succeeded in training an 8-round distinguisher slightly superior to the difference distribution table by using several stages of pre-training. First, we retrained our best seven-round distinguisher to recognize 5-round Speck32/64 with the input difference $0x8000/840a$ (the most likely difference to appear three rounds after the input difference $0x0040/0000$). This was done on 10^7 examples for ten epochs with a batch size of 5000 and a learning rate of 10^{-4}. Then, we trained the distinguisher so obtained to recognize 8-round Speck with the input difference $0x0040/0000$ by processing 10^9 freshly generated examples once with batch size 10000, keeping the learning rate constant. Finally, learning rate was dropped twice to 10^{-5} and finally to 10^{-6} after processing another 10^9 fresh examples each, again with a batch size of 10000.

Is not clear that Gohr's network [Goh19] is suitable for all ciphers [GLN23]

Gohr's neural architecture encodes the word structure of SPECK, implicitely adding knowledge on the primitive. The choice of the best structure to use is not always that straightforward: for instance, for AES, dependencies exists between the bits of a difference byte, but also between the bytes of a column, and sometimes (in related key) between the bytes of a given row. The representation choice has a significant impact on the learning performance, so it is useful to automatically handle such long and short range dependencies; this also preserves the black box aspect of the distinguisher.



[3]:
```
dbitnet = tester.get_neural_network(network_name='dbitnet', input_size=64)
dbitnet.summary()

# For Gohr's resnet:
# tester.get_neural_network(network_name='gohr_resnet', input_size=64,␣
 ↪word_size=16)
```

```
2025-03-15 08:00:46.098853: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 47365 MB memory:  -> device:
0, name: Quadro RTX 8000, pci bus id: 0000:1d:00.0, compute capability: 7.5
2025-03-15 08:00:46.100136: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:1 with 47365 MB memory:  -> device:
1, name: Quadro RTX 8000, pci bus id: 0000:1e:00.0, compute capability: 7.5
2025-03-15 08:00:46.101135: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:2 with 47365 MB memory:  -> device:
2, name: Quadro RTX 8000, pci bus id: 0000:1f:00.0, compute capability: 7.5
2025-03-15 08:00:46.102092: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:3 with 47365 MB memory:  -> device:
3, name: Quadro RTX 8000, pci bus id: 0000:20:00.0, compute capability: 7.5
2025-03-15 08:00:46.103039: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:4 with 47365 MB memory:  -> device:
4, name: Quadro RTX 8000, pci bus id: 0000:21:00.0, compute capability: 7.5
2025-03-15 08:00:46.103999: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:5 with 47365 MB memory:  -> device:
5, name: Quadro RTX 8000, pci bus id: 0000:22:00.0, compute capability: 7.5
2025-03-15 08:00:46.104924: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:6 with 47365 MB memory:  -> device:
6, name: Quadro RTX 8000, pci bus id: 0000:23:00.0, compute capability: 7.5
2025-03-15 08:00:46.105862: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device
/job:localhost/replica:0/task:0/device:GPU:7 with 47365 MB memory:  -> device:
7, name: Quadro RTX 8000, pci bus id: 0000:24:00.0, compute capability: 7.5
```

```
Model: "model"
_________________________________________________________________________________
__________________
 Layer (type)                Output Shape                 Param #   Connected to
=================================================================================
==================
 input_1 (InputLayer)        [(None, 64, 1)]              0         []
```

```
 tf.math.subtract (TFOpLamb  (None, 64, 1)               0
['input_1[0][0]']
 da)

 tf.math.truediv (TFOpLambd  (None, 64, 1)               0
['tf.math.subtract[0][0]']
 a)

 conv1d (Conv1D)             (None, 33, 32)              96
['tf.math.truediv[0][0]']

 batch_normalization (Batch  (None, 33, 32)              128
['conv1d[0][0]']
 Normalization)

 conv1d_1 (Conv1D)           (None, 33, 32)              2080
['batch_normalization[0][0]']

 add (Add)                   (None, 33, 32)              0
['conv1d_1[0][0]',
'batch_normalization[0][0]']

 batch_normalization_1 (Bat  (None, 33, 32)              128
['add[0][0]']
 chNormalization)

 conv1d_2 (Conv1D)           (None, 18, 48)              3120
['batch_normalization_1[0][0]'
                                                                 ]

 batch_normalization_2 (Bat  (None, 18, 48)              192
['conv1d_2[0][0]']
 chNormalization)

 conv1d_3 (Conv1D)           (None, 18, 48)              4656
['batch_normalization_2[0][0]'
                                                                 ]

 add_1 (Add)                 (None, 18, 48)              0
['conv1d_3[0][0]',
'batch_normalization_2[0][0]'
                                                                 ]

 batch_normalization_3 (Bat  (None, 18, 48)              192
['add_1[0][0]']
 chNormalization)

 conv1d_4 (Conv1D)           (None, 11, 64)              6208
```

```
['batch_normalization_3[0][0]'
                                                                  ]

 batch_normalization_4 (Bat  (None, 11, 64)              256
['conv1d_4[0][0]']
 chNormalization)

 conv1d_5 (Conv1D)           (None, 11, 64)              8256
['batch_normalization_4[0][0]'
                                                                  ]

 add_2 (Add)                 (None, 11, 64)              0
['conv1d_5[0][0]',
'batch_normalization_4[0][0]'
                                                                  ]

 batch_normalization_5 (Bat  (None, 11, 64)              256
['add_2[0][0]']
 chNormalization)

 conv1d_6 (Conv1D)           (None, 8, 80)               10320
['batch_normalization_5[0][0]'
                                                                  ]

 batch_normalization_6 (Bat  (None, 8, 80)               320
['conv1d_6[0][0]']
 chNormalization)

 conv1d_7 (Conv1D)           (None, 8, 80)               12880
['batch_normalization_6[0][0]'
                                                                  ]

 add_3 (Add)                 (None, 8, 80)               0
['conv1d_7[0][0]',
'batch_normalization_6[0][0]'
                                                                  ]

 batch_normalization_7 (Bat  (None, 8, 80)               320
['add_3[0][0]']
 chNormalization)

 flatten (Flatten)           (None, 640)                 0
['batch_normalization_7[0][0]'
                                                                  ]

 dense (Dense)               (None, 256)                 164096
['flatten[0][0]']
```

```
 batch_normalization_8 (Bat  (None, 256)                  1024
['dense[0][0]']
 chNormalization)

 activation (Activation)     (None, 256)                  0
['batch_normalization_8[0][0]'
                                                                 ]

 dense_1 (Dense)             (None, 256)                  65792
['activation[0][0]']

 batch_normalization_9 (Bat  (None, 256)                  1024
['dense_1[0][0]']
 chNormalization)

 activation_1 (Activation)   (None, 256)                  0
['batch_normalization_9[0][0]'
                                                                 ]

 dense_2 (Dense)             (None, 64)                   16448
['activation_1[0][0]']

 batch_normalization_10 (Ba  (None, 64)                   256
['dense_2[0][0]']
 tchNormalization)

 activation_2 (Activation)   (None, 64)                   0
['batch_normalization_10[0][0]
                                                                 ']

 dense_3 (Dense)             (None, 1)                    65
['activation_2[0][0]']

==================================================================================
==================
Total params: 298113 (1.14 MB)
Trainable params: 296065 (1.13 MB)
Non-trainable params: 2048 (8.00 KB)
----------------------------------------------------------------------------------
------------------
```

[4]:
```
X, Y = tester.get_differential_dataset(input_differences=[delta_in, 0],␣
  ↪number_of_rounds=5, samples=10**6)
print(X.shape, Y.shape)
```

```
(1000000, 64) (1000000,)
```

[5]:
```
h = dbitnet.fit(X, Y, epochs=int(5), batch_size=5000, validation_split=0.1) #␣
↪Very slow on a CPU
```

```
Epoch 1/5
```

```
2025-03-15 08:00:59.484977: I
tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:432] Loaded cuDNN
version 8907
2025-03-15 08:00:59.878269: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x7f6e54265470 initialized for platform CUDA (this does not
guarantee that XLA will be used). Devices:
2025-03-15 08:00:59.878314: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (0): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878321: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (1): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878328: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (2): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878336: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (3): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878342: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (4): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878349: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (5): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878356: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (6): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.878363: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (7): Quadro RTX 8000, Compute Capability 7.5
2025-03-15 08:00:59.883439: I
tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:255] disabling MLIR
crash reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
2025-03-15 08:01:00.033324: I ./tensorflow/compiler/jit/device_compiler.h:186]
Compiled cluster using XLA!  This line is logged at most once for the lifetime
of the process.
```

```
180/180 [==============================] - 13s 31ms/step - loss: 0.1426 - acc:
0.8086 - val_loss: 0.2408 - val_acc: 0.6270
Epoch 2/5
180/180 [==============================] - 5s 29ms/step - loss: 0.0996 - acc:
0.8774 - val_loss: 0.1888 - val_acc: 0.7304
Epoch 3/5
180/180 [==============================] - 5s 28ms/step - loss: 0.0972 - acc:
0.8797 - val_loss: 0.0989 - val_acc: 0.8775
Epoch 4/5
180/180 [==============================] - 5s 28ms/step - loss: 0.0959 - acc:
0.8809 - val_loss: 0.0963 - val_acc: 0.8796
Epoch 5/5
180/180 [==============================] - 5s 28ms/step - loss: 0.0948 - acc:
0.8821 - val_loss: 0.0959 - val_acc: 0.8800
```

In AutoND, we use curriculum learning to replace the non-blackbox staged training approach from related works; in practice, we retrain the neural distinguisher from round r-1 on round r data, so that basic properties (e.g., XOR) do not need to be relearnt. (Bao et al. (Asiacrypt 2023) use freezing of the first layers instead)

In CLAASP, the whole pipeline can be run as follows:

[6]:
```
from claasp.cipher_modules.report import Report
results = tester.run_autond_pipeline(number_of_epochs=5, neural_net='dbitnet',␣
 ↪optimizer_generations=10, training_samples=10**6, testing_samples=10**5,␣
 ↪optimizer_samples=10**3, verbose=True)

report = Report(results)
report.show()
```

```
Generation 0/10, 1619 nodes explored, 32 current, best is ['0xbd3df7fa',
'0xd3efc3e', '0x8878b038', '0x104e0c30'] with [0.4315     0.4775625  0.4815625
0.60665625]
Generation 1/10, 2177 nodes explored, 32 current, best is ['0x45eb41a',
'0x47eb41a', '0x2000', '0x200000'] with [0.63171875 0.63296875 1.92809375
2.59165625]
Generation 2/10, 2670 nodes explored, 32 current, best is ['0x202000',
'0x208000', '0x2000', '0x200000'] with [1.69390625 1.6968125  1.92809375
2.59165625]
Generation 3/10, 3027 nodes explored, 32 current, best is ['0x2000',
'0x4000000', '0x8000', '0x200000'] with [1.92809375 2.1394375  2.43546875
2.59165625]
Generation 4/10, 3404 nodes explored, 32 current, best is ['0x4000000',
'0x8000', '0x102000', '0x200000'] with [2.1394375  2.43546875 2.562
2.59165625]
Generation 5/10, 3669 nodes explored, 32 current, best is ['0x10000000',
'0x8000', '0x102000', '0x200000'] with [2.244875   2.43546875 2.562
2.59165625]
Generation 6/10, 3932 nodes explored, 32 current, best is ['0x204000', '0x8000',
'0x102000', '0x200000'] with [2.32428125 2.43546875 2.562      2.59165625]
Generation 7/10, 4148 nodes explored, 32 current, best is ['0x102000',
'0x200000', '0x600000', '0x400000'] with [2.562      2.59165625 2.64390625
3.42484375]
Generation 8/10, 4342 nodes explored, 32 current, best is ['0x200000',
'0x600000', '0x408000', '0x400000'] with [2.59165625 2.64390625 3.1311875
3.42484375]
Generation 9/10, 4474 nodes explored, 32 current, best is ['0x200000',
'0x600000', '0x408000', '0x400000'] with [2.59165625 2.64390625 3.1311875
3.42484375]
The highest reached round was 6
The best differences found by the optimizer are…
0x400000 , with score 3.42484375
0x408000 , with score 3.1311875000000002
```

```
0x600000 , with score 2.64390625
0x200000 , with score 2.59165625
0x102000 , with score 2.5620000000000007
0x1408000 , with score 2.5424375
0x502000 , with score 2.4449375000000004
0x8000 , with score 2.43546875
0x604000 , with score 2.3515937500000006
0x204000 , with score 2.3242812500000003
Training dbitnet on input difference ['0x400000', '0x0'] (['plaintext', 'key']),
from round 3…
Epoch 1/5
200/200 [==============================] - 13s 30ms/step - loss: 0.0296 - acc:
0.9694 - val_loss: 0.4717 - val_acc: 0.5011
Epoch 2/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0065 - acc:
0.9996 - val_loss: 0.4077 - val_acc: 0.5406
Epoch 3/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0050 - acc:
0.9999 - val_loss: 0.0053 - val_acc: 0.9998
Epoch 4/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0039 - acc:
0.9999 - val_loss: 0.0036 - val_acc: 0.9999
Epoch 5/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0030 - acc:
1.0000 - val_loss: 0.0028 - val_acc: 0.9999
Validation accuracy at 3 rounds :0.9998999834060669
Epoch 1/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0319 - acc:
0.9660 - val_loss: 0.0342 - val_acc: 0.9628
Epoch 2/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0230 - acc:
0.9755 - val_loss: 0.0224 - val_acc: 0.9766
Epoch 3/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0216 - acc:
0.9766 - val_loss: 0.0217 - val_acc: 0.9762
Epoch 4/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0205 - acc:
0.9775 - val_loss: 0.0218 - val_acc: 0.9761
Epoch 5/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0196 - acc:
0.9782 - val_loss: 0.0202 - val_acc: 0.9774
Validation accuracy at 4 rounds :0.9773600101470947
Epoch 1/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0993 - acc:
0.8728 - val_loss: 0.0948 - val_acc: 0.8786
Epoch 2/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0934 - acc:
0.8793 - val_loss: 0.0938 - val_acc: 0.8804
```

```
Epoch 3/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0908 - acc:
0.8838 - val_loss: 0.0938 - val_acc: 0.8831
Epoch 4/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0874 - acc:
0.8888 - val_loss: 0.0912 - val_acc: 0.8866
Epoch 5/5
200/200 [==============================] - 6s 29ms/step - loss: 0.0851 - acc:
0.8920 - val_loss: 0.0859 - val_acc: 0.8913
Validation accuracy at 5 rounds :0.891319990158081
Epoch 1/5
200/200 [==============================] - 6s 29ms/step - loss: 0.1927 - acc:
0.7102 - val_loss: 0.1861 - val_acc: 0.7207
Epoch 2/5
200/200 [==============================] - 6s 29ms/step - loss: 0.1829 - acc:
0.7261 - val_loss: 0.1833 - val_acc: 0.7256
Epoch 3/5
200/200 [==============================] - 6s 29ms/step - loss: 0.1810 - acc:
0.7291 - val_loss: 0.1824 - val_acc: 0.7270
Epoch 4/5
200/200 [==============================] - 6s 29ms/step - loss: 0.1797 - acc:
0.7312 - val_loss: 0.1814 - val_acc: 0.7275
Epoch 5/5
200/200 [==============================] - 6s 29ms/step - loss: 0.1765 - acc:
0.7371 - val_loss: 0.1766 - val_acc: 0.7372
Validation accuracy at 6 rounds :0.7372400164604187
Epoch 1/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2497 - acc:
0.5457 - val_loss: 0.2463 - val_acc: 0.5631
Epoch 2/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2432 - acc:
0.5736 - val_loss: 0.2430 - val_acc: 0.5747
Epoch 3/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2417 - acc:
0.5797 - val_loss: 0.2424 - val_acc: 0.5770
Epoch 4/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2409 - acc:
0.5826 - val_loss: 0.2422 - val_acc: 0.5762
Epoch 5/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2402 - acc:
0.5845 - val_loss: 0.2422 - val_acc: 0.5754
Validation accuracy at 7 rounds :0.576960027217865
Epoch 1/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2523 - acc:
0.4992 - val_loss: 0.2518 - val_acc: 0.5003
Epoch 2/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2516 - acc:
0.5052 - val_loss: 0.2516 - val_acc: 0.5025
```

```
Epoch 3/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2514 - acc:
0.5086 - val_loss: 0.2514 - val_acc: 0.5033
Epoch 4/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2511 - acc:
0.5118 - val_loss: 0.2514 - val_acc: 0.5035
Epoch 5/5
200/200 [==============================] - 6s 29ms/step - loss: 0.2509 - acc:
0.5153 - val_loss: 0.2515 - val_acc: 0.5027
Validation accuracy at 8 rounds :0.5034599900245667



RESULTS
plaintext_input_diff : 0x400000
key_input_diff : 0x0
                         0
accuracy_round3  0.99990
accuracy_round4  0.97736
accuracy_round5  0.89132
accuracy_round6  0.73724
accuracy_round7  0.57696
accuracy_round8  0.50346

//////

SCORES
              scores
0x402000    1.867187
0x800       1.872469
0x50a000    1.883219
0x14000008  1.888469
0x20c000    1.924656
0x2000      1.928094
0x20400000  1.936719
0x700000    1.937438
0x408002    1.966562
0x4000      1.995187
0x60c000    2.024469
0x404000    2.057437
0x1000000   2.079625
0x80000     2.138000
0x4000000   2.139438
0x100000    2.150000
0x20000000  2.165563
0x1000002   2.173688
0x302000    2.179250
0x702000    2.183875
```

```
0x4000008   2.185563
0x10000000  2.244875
0x204000    2.324281
0x604000    2.351594
0x8000      2.435469
0x502000    2.444938
0x1408000   2.542438
0x102000    2.562000
0x200000    2.591656
0x600000    2.643906
0x408000    3.131188
0x400000    3.424844
```

0.1.4 Using AutoND with custom implementations

The original AutoND, as well as CLAASP, require specific implementation formats for the primitives. However, it is easy to implement a wrapper for custom implementations and use it within AutoND.

[7]:
```
!git clone https://github.com/Crypto-TII/AutoND.git
```

```
fatal: destination path 'AutoND' already exists and is not an empty directory.
```

But first, some path shennanigans for the notebook to see AutoND...

[8]:
```
import sys
from pathlib import Path
# =========== Set up PATH to be able to import modules from scr ==============
# Define a constant for the base directory
if '__file__' in globals():
    # Running in a script
    BASE_DIR = Path(__file__).resolve().parent
else:
    # Running in a Jupyter Notebook
    BASE_DIR = Path().resolve()
# Use BASE_DIR to modify sys.path
sys.path.append(str(BASE_DIR / 'AutoND'))
```

The AutoND framework API requires an encryption_function, with inputs:

- p: A binary uint8 numpy matrix of plaintexts, with one row per sample and one column per bit
- k: A binary uint8 numpy matrix of keys, with one row per sample and one column per bit
- number_of_rounds: the number of encryption rounds.

It is relatively straightforward to build such a wrapper, for instance extracting the CLAASP encryption function implementation.

[9]:
```
import random
from claasp.cipher_modules import code_generator
```

```
from claasp.cipher_modules.generic_functions_vectorized_byte import␣
 ↪integer_array_to_evaluate_vectorized_input

import numpy as np
from types import ModuleType

# Speck encryption function generation
python_code_string = code_generator.
 ↪generate_byte_based_vectorized_python_code_string(speck,␣
 ↪store_intermediate_outputs=True)
speck_encryption_functions = ModuleType("speck_encryption_functions")
exec(python_code_string, speck_encryption_functions.__dict__)
speck_encrypt = lambda p, k, nr: speck_encryption_functions.evaluate([p, k],␣
 ↪True)['round_output'][nr-1]

def encryption_function_wrapper(p, k, number_of_rounds):
    '''
    optimize requires an encryption_function that takes as input a matrix of␣
 ↪bits (one row per sample); the wrapper does the conversion
    '''
    p_bytes = np.packbits(p, axis=1).transpose()
    k_bytes = np.packbits(k, axis=1).transpose()
    c_bytes =  speck_encrypt(p_bytes, k_bytes, number_of_rounds)
    c_bits = np.unpackbits(c_bytes, axis=1)
    return c_bits
```

We can then run the optimizer to find a good input difference…

[10]:
```
from AutoND.optimizer import optimize
input_diffs, highest_round = optimize(plain_bits=32, key_bits=64,␣
 ↪encryption_function=encryption_function_wrapper, nb_samples=10**3, scenario␣
 ↪= "single-key", log_file = None, epsilon=0.1)
```

```
Evaluating differences at round  1
Generation 0/5, 528 nodes explored, 32 current, best is ['0x805db204',
'0xfe23c7f2', '0xddffffaf', '0xff0dd9fc'] with [0.2844375 0.2946875 0.30125
0.302    ]
Generation 1/5, 1023 nodes explored, 32 current, best is ['0xff0dd9fc',
'0x81034300', '0xcc0103b0', '0xd33f7ff0'] with [0.302     0.30375   0.31
0.3110625]
Generation 2/5, 1513 nodes explored, 32 current, best is ['0xd33f7ff0',
'0x80490f80', '0xe0028740', '0xf03dfbe0'] with [0.3110625 0.3290625 0.335625
0.3810625]
Generation 3/5, 2005 nodes explored, 32 current, best is ['0xd', '0x20000044',
'0xf03dfbe0', '0x200'] with [0.363625  0.3745    0.3810625 0.4371875]
Generation 4/5, 2497 nodes explored, 32 current, best is ['0xf03dfbe0',
'0x6000000', '0x208800', '0x200'] with [0.3810625 0.4025625 0.414875  0.4371875]
Evaluating differences at round  2
```

```
Generation 0/5, 528 nodes explored, 32 current, best is ['0x80400200',
'0x2040000', '0x6000000', '0x200'] with [0.2288125  0.23984375 0.25946875
0.2840625 ]
Generation 1/5, 1020 nodes explored, 32 current, best is ['0x2040000',
'0x6000000', '0x200', '0x400000'] with [0.23984375 0.25946875 0.2840625
0.439375  ]
Generation 2/5, 1507 nodes explored, 32 current, best is ['0x40400000',
'0x40000', '0x20000000', '0x400000'] with [0.291125   0.32159375 0.32403125
0.439375  ]
Generation 3/5, 1984 nodes explored, 32 current, best is ['0x2000004',
'0x40000', '0x20000000', '0x400000'] with [0.3150625  0.32159375 0.32403125
0.439375  ]
Generation 4/5, 2455 nodes explored, 32 current, best is ['0x40000',
'0x20000000', '0x404000', '0x400000'] with [0.32159375 0.32403125 0.3314375
0.439375  ]
Evaluating differences at round  3
Generation 0/5, 521 nodes explored, 32 current, best is ['0x4000000',
'0x20000000', '0x2000004', '0x400000'] with [0.1989375  0.20653125 0.2213125
0.31371875]
Generation 1/5, 974 nodes explored, 32 current, best is ['0x8000000', '0x8000',
'0x2000004', '0x400000'] with [0.21       0.22009375 0.2213125  0.31371875]
Generation 2/5, 1426 nodes explored, 32 current, best is ['0x8000', '0x2000004',
'0x600000', '0x400000'] with [0.22009375 0.2213125  0.2433125  0.31371875]
Generation 3/5, 1839 nodes explored, 32 current, best is ['0x200000',
'0x600000', '0x408000', '0x400000'] with [0.23934375 0.2433125  0.28903125
0.31371875]
Generation 4/5, 2246 nodes explored, 32 current, best is ['0x200000',
'0x600000', '0x408000', '0x400000'] with [0.23934375 0.2433125  0.28903125
0.31371875]
Evaluating differences at round  4
Generation 0/5, 490 nodes explored, 32 current, best is ['0x600000', '0x200000',
'0x408000', '0x400000'] with [0.1169375  0.11696875 0.1514375  0.17509375]
Generation 1/5, 912 nodes explored, 32 current, best is ['0x600000', '0x200000',
'0x408000', '0x400000'] with [0.1169375  0.11696875 0.1514375  0.17509375]
Generation 2/5, 1321 nodes explored, 32 current, best is ['0x600000',
'0x200000', '0x408000', '0x400000'] with [0.1169375  0.11696875 0.1514375
0.17509375]
Generation 3/5, 1699 nodes explored, 32 current, best is ['0x600000',
'0x200000', '0x408000', '0x400000'] with [0.1169375  0.11696875 0.1514375
0.17509375]
Generation 4/5, 2049 nodes explored, 32 current, best is ['0x600000',
'0x200000', '0x408000', '0x400000'] with [0.1169375  0.11696875 0.1514375
0.17509375]
Evaluating differences at round  5
Generation 0/5, 488 nodes explored, 32 current, best is ['0x200000',
'0x1408000', '0x408000', '0x400000'] with [0.0374375  0.04315625 0.04421875
0.05934375]
Generation 1/5, 924 nodes explored, 32 current, best is ['0x200000',
```

```
'0x1408000', '0x408000', '0x400000'] with [0.0374375  0.04315625 0.04421875
0.05934375]
Generation 2/5, 1330 nodes explored, 32 current, best is ['0x200000',
'0x1408000', '0x408000', '0x400000'] with [0.0374375  0.04315625 0.04421875
0.05934375]
Generation 3/5, 1714 nodes explored, 32 current, best is ['0x200000',
'0x1408000', '0x408000', '0x400000'] with [0.0374375  0.04315625 0.04421875
0.05934375]
Generation 4/5, 2093 nodes explored, 32 current, best is ['0x200000',
'0x1408000', '0x408000', '0x400000'] with [0.0374375  0.04315625 0.04421875
0.05934375]
Evaluating differences at round  6
Generation 0/5, 493 nodes explored, 32 current, best is ['0xa04000',
'0x80e04001', '0xa2400004', '0x400000'] with [0.01796875 0.01803125 0.018125
0.01984375]
Generation 1/5, 978 nodes explored, 32 current, best is ['0xa2400004',
'0x8000050', '0x256000', '0x400000'] with [0.018125   0.01828125 0.01928125
0.01984375]
Generation 2/5, 1459 nodes explored, 32 current, best is ['0x8000050',
'0x48808050', '0x256000', '0x400000'] with [0.01828125 0.019      0.01928125
0.01984375]
Generation 3/5, 1940 nodes explored, 32 current, best is ['0x8000050',
'0x48808050', '0x256000', '0x400000'] with [0.01828125 0.019      0.01928125
0.01984375]
Generation 4/5, 2408 nodes explored, 32 current, best is ['0x8000050',
'0x48808050', '0x256000', '0x400000'] with [0.01828125 0.019      0.01928125
0.01984375]
```

The implementation can also be used to train the neural network as follows.

[ ]:
```
from AutoND.train_nets import train_neural_distinguisher
from AutoND.main import make_train_data, integer_to_binary_array

# Using the best input difference returned by the optimizer
delta_state_bin = integer_to_binary_array(input_diffs[-1], plain_bits)

# And a wrapper for the dataset generation
def data_generator(num_samples, nr, delta_state = delta_state_bin):
    return make_train_data(encryption_function_wrapper, plain_bits, key_bits,␣
 ↪num_samples, nr, delta_state, delta_key=0)

# We train from the last round reached by the optimizer.
starting_round = highest_round
train_neural_distinguisher(starting_round, data_generator, 'gohr_amsgrad',␣
 ↪int(32), int(16), log_prefix = './', _epochs = int(5), _num_samples=10**6,␣
 ↪_num_val_samples=10**5)
```

```
Training on 5 epochs …
```

```
Epoch 1/5
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3000:
UserWarning: You are saving your model as an HDF5 file via `model.save()`. This
file format is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
200/200 - 6s - loss: 0.1287 - acc: 0.8268 - val_loss: 0.2138 - val_acc: 0.6310 -
6s/epoch - 30ms/step
Epoch 2/5
200/200 - 3s - loss: 0.0968 - acc: 0.8765 - val_loss: 0.0992 - val_acc: 0.8755 -
3s/epoch - 13ms/step
Epoch 3/5
200/200 - 3s - loss: 0.0935 - acc: 0.8804 - val_loss: 0.0944 - val_acc: 0.8798 -
3s/epoch - 13ms/step
Epoch 4/5
200/200 - 3s - loss: 0.0915 - acc: 0.8835 - val_loss: 0.0927 - val_acc: 0.8823 -
3s/epoch - 13ms/step
Epoch 5/5
200/200 - 3s - loss: 0.0889 - acc: 0.8877 - val_loss: 0.0897 - val_acc: 0.8867 -
3s/epoch - 13ms/step
gohr_amsgrad, round 5. Best validation accuracy: 0.886650025844574
Epoch 1/5
200/200 - 3s - loss: 0.2000 - acc: 0.6964 - val_loss: 0.1999 - val_acc: 0.6980 -
3s/epoch - 15ms/step
Epoch 2/5
200/200 - 3s - loss: 0.1886 - acc: 0.7138 - val_loss: 0.1921 - val_acc: 0.7138 -
3s/epoch - 13ms/step
Epoch 3/5
200/200 - 3s - loss: 0.1833 - acc: 0.7241 - val_loss: 0.1847 - val_acc: 0.7224 -
3s/epoch - 13ms/step
Epoch 4/5
200/200 - 3s - loss: 0.1799 - acc: 0.7305 - val_loss: 0.1812 - val_acc: 0.7268 -
3s/epoch - 13ms/step
Epoch 5/5
200/200 - 3s - loss: 0.1763 - acc: 0.7370 - val_loss: 0.1783 - val_acc: 0.7327 -
3s/epoch - 13ms/step
gohr_amsgrad, round 6. Best validation accuracy: 0.732699990272522
Epoch 1/5
200/200 - 3s - loss: 0.2505 - acc: 0.5441 - val_loss: 0.2470 - val_acc: 0.5542 -
3s/epoch - 16ms/step
Epoch 2/5
200/200 - 3s - loss: 0.2446 - acc: 0.5674 - val_loss: 0.2450 - val_acc: 0.5671 -
3s/epoch - 13ms/step
Epoch 3/5
200/200 - 3s - loss: 0.2430 - acc: 0.5753 - val_loss: 0.2445 - val_acc: 0.5685 -
3s/epoch - 15ms/step
Epoch 4/5
```

```
200/200 - 3s - loss: 0.2420 - acc: 0.5796 - val_loss: 0.2441 - val_acc: 0.5718 -
3s/epoch - 13ms/step
Epoch 5/5
200/200 - 3s - loss: 0.2412 - acc: 0.5826 - val_loss: 0.2440 - val_acc: 0.5710 -
3s/epoch - 13ms/step
gohr_amsgrad, round 7. Best validation accuracy: 0.5717599987983704
Epoch 1/5
200/200 - 3s - loss: 0.2536 - acc: 0.4997 - val_loss: 0.2527 - val_acc: 0.5007 -
3s/epoch - 16ms/step
Epoch 2/5
200/200 - 3s - loss: 0.2519 - acc: 0.5093 - val_loss: 0.2524 - val_acc: 0.4996 -
3s/epoch - 13ms/step
Epoch 3/5
200/200 - 3s - loss: 0.2516 - acc: 0.5137 - val_loss: 0.2522 - val_acc: 0.5012 -
3s/epoch - 13ms/step
Epoch 4/5
200/200 - 2s - loss: 0.2513 - acc: 0.5190 - val_loss: 0.2522 - val_acc: 0.4999 -
2s/epoch - 11ms/step
Epoch 5/5
```

0.1.5 What's next?

- Improved optimizer with GPU-based parallel encryption
- Prepended differential integration (future work of FSE'24)
- Feature engineering automation through partial decryption

-> Will be made open source through the AutoND repo with a focus on modularity and compatibility with other libraries

[ ]: