MH4900

NANYANG TECHNOLOGICAL UNIVERSITY

DIVISION OF MATHEMATICAL SCIENCES
SCHOOL OF PHYSICAL & MATHEMATICAL SCIENCES

FINAL THESIS

Title: Machine Learning for Cryptanalysis

Submitted by: Yang Siwei, Allen

U1840573E

Supervisor: Assoc. Prof. Thomas Peyrin

Academic Year 2021 Semester 2

A Final Year Project submitted to the School of Physical and Mathematical Sciences, Nanyang Technological University in partial fulfillment of the requirements for the *Degree of Bachelor of Science*

Abstract

Lightweight cryptography involves cryptography subject to constraints such as area and power consumption. In 2019, NIST organised a currently ongoing competition for lightweight authenticated encryption. This competition, currently in its final stage, has narrowed down the initial 57 submissions to 10 finalists, of which includes the cipher known as ASCON. Concurrent with this was a research paper published by Gohr in 2019. Here, it was shown that it was possible to apply deep learning to cryptanalysis. More specifically, it was possible to design a neural distinguisher for the Speck 32/64 cipher, where for a specified input difference, it was possible to distinguish between ciphertext pairs that had this input difference and a random one via classification.

In this study, we provide an amalgamation of these two advances. Here, we first attempt to apply Gohr's neural network architecture to the round function of the ASCON cipher. Following this, we attempt to improve on the architecture in hopes of greater accuracy. The primary result found was that Gohr's network architecture had the ability to learn well up to 3.5 rounds of the ASCON round function. Following this, we were able to provide modifications for marginal improvement in accuracy of the 4 round case.

Keywords - cryptanalysis, deep learning, lightweight cryptography

Acknowledgements

I would like to extend my deepest gratitude to Prof. Peyrin and Quan Quan, both of whom have provided a surplus of suggestions and advice through the course of the project. They have always been readily available and have tirelessly guided me through the project. The most notable of these instances, for Quan Quan, would be the willingness to deal with my, now looking back on it, asinine questions regarding the implementation of the ASCON cipher at the beginning of the project, and for Prof. Peyrin, the willingness to arrange for a consultation even whilst on holiday.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In 2017, the National Institute of Standards and Technology(NIST) organised the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR). Among the submissions include the lightweight category, where cipher submissions are selected based on their suitability for devices with important constraints such as area and energy consumption (Gerault, Peyrin, & Tan, 2021). Also, in 2019, NIST organised the currently ongoing NIST Lightweight Cryptography competition. The ASCON cipher was chosen as the primary choice in the lightweight category of CAESAR and is currently one of the finalists for the Lightweight Cryptography competition, among 57 submissions. It consists of multiple variants such as ASCON-128, ASCON-HASH and ASCON-128a. Having gone through multiple instances of public evaluation, ASCON-128 and ASCON-128a both show no indication of weakness (Dobraunig, Eichlseder, Mendel, & Schläffer, 2019).

In recent years, deep learning has also progressed significantly with a multitude of applications in areas such as image recognition, speech processing and signal processing (Goodfellow, Bengio, & Courville, 2016). Among the progress includes work directed towards cryptographic applications (Breier et al., 2018). In 2019, Gohr used deep learning to aid in cryptanalysis of the Speck32/64 cipher (Gohr, 2019). Here, Gohr proposed a network architecture involving a residual tower of two-layer convolutional networks. With the use of this, it was possible to develop a neural distinguisher for the Speck32/64 cipher. Ciphertext pairs with fixed input differences were able to be distinguished from those of random differences. This also had provided an improvement in classification results compared to previously used purely differential distinguishers.

In this study, we adapt Gohr's network architecture to the ASCON cipher, specifically the ASCON-128 variant. Here, we use the architecture to create a neural distinguisher for the ASCON round function. For a fixed input difference, we attempt to classify ciphertext outputs with the fixed input difference apart from those with random input difference. This was conducted for multiple rounds of the ASCON round function. We then attempt to apply variants of the architecture, in hopes of better accuracy.

2 ASCON

We first provide a brief description of the ASCON cipher (Dobraunig et al., 2019). It is to be noted that the ASCON-128 variant has been coded in the appendix. This will include terminology, the overall algorithm and descriptions of the initialisation, processing associated data, processing plaintext and ciphertext and the round function itself.

2.1 Terminology

Table 1 is a list of terminology used for the description of the components of ASCON later as well as the ASCON algorithm itself.

Notation	
Term	Definition
K, k	Secret key K, where size of $K \leq 160$ bits, keysize is denoted as k
N	128 bit nonce
T	128 bit tag
P, C, A	Plaintext P, Ciphertext C and Associated data A (also expressed in r-bit blocks P_i, C_i, A_i)
M, H	Message M and hash value H ((also expressed in r-bit blocks M_i, H_i)
$\perp$	Error in verifying authenticated ciphertext
S, S_r, S_c	320 bit state S of the sponge construction, r-bit rate of S and c-bit capacity of S where $S = S_r \| S_c$ i.e. $c = 320 - r$
p, p^a, p^b	Permutation p, where p^a, p^b represent a and b rounds of permutation
$x \in \{0,1\}^k$	Bit string x of length k, if $k = *$ then x has variable length
0^k	Bit string of length k consisting of 0-bits, if $k = *$ then is of variable length
$\lvert x \rvert$	Bit-length of x
$\lfloor x \rfloor^k$, $\lceil x \rceil^k$	Bitstring x truncated to most and least significant k bits respectively
$x \| y$	Bitstring x concatenated with bitstring y
$x \oplus y$	Bitstring x XORed with bitstring y
$x \bigodot y$	Bitstring y ANDed with bitstring y
p_c, p_s, p_l	Constant addition, substitution and linear layers of the permutation p
$x \ggg i$	Circular shift/ rightwards rotation of bitstring x by i bits
$\mathcal{E}_{k,r,a,b}(K, N, A, P) = (C, T)$	Encryption with parameters k, r, a, b applied to K, N, A, P to output ciphertext C and tag T
$\mathcal{D}_{k,r,a,b}(K, N, A, C, T) = \{P, \perp\}$	Decryption with parameters k, r, a, b applied to K, N, A, C, T to output plaintext P and errorcheck $\perp$

Table 1: Terminology List

2.2 ASCON Specifications

Here in Table 2 we present a list of specifications for the some of the variants of ASCON. It is to be noted that for this study, we focus on the ASCON-128 variant.

ASCON Variant	Key size $k = \|K\|$	Nonce size $\|N\|$	Tag size $\|T\|$	Data block size	Permutation rounds a (p^a)	Permutation rounds b (p^b)
ASCON-128, $\mathcal{E}, \mathcal{D}_{128,64,12,6}$	128	128	128	64	12	6
ASCON-128a, $\mathcal{E}, \mathcal{D}_{128,128,12,8}$	128	128	128	128	12	8

Table 2: Specifications

2.3 ASCON Algorithm

2.3.1 Outline

Here, we provide the algorithm for ASCON as well as a diagram for visualisation.

Algorithm 1 Authenticated Encryption ($\mathcal{E}_{k,r,a,b}(K, N, A, P) = (C, T)$)

Inputs: K, N, A, P
Outputs: $C \in \{0,1\}^{|P|}$, T
Initialisation:
  $S \leftarrow IV_{k,r,a,b}||K||N$
  $S \leftarrow p^a(S) \oplus (0^{320-k}||K)$
Processing Associated Data:
if $|A| > 0$ **then**:
  $A_1, A_2, ..., A_s \leftarrow r$-bit blocks of $A||1||0^*$
  for i in range 1 to s **do**:
    $S \leftarrow p^b((S_r \oplus A_i)||S_c)$
  end for
  $S \leftarrow S \oplus (0^{319}||1)$
end if
Processing Plaintext:
  $P_1, P_2, ..., P_t \leftarrow r$-bit blocks of $P||1||0^*$
for i in range 1 to $t-1$ **do**:
  $S_r \leftarrow S_r \oplus P_i$
  $C_i \leftarrow S_r$
  $S \leftarrow p^b(S)$
end for
  $S_r \leftarrow S_r \oplus P_t$
  $C'_t \leftarrow \lfloor S_r \rfloor_{|P| \mod r}$
Finalization:
  $S \leftarrow p^a(S \oplus (0^r||K||0^{320-r-k}))$
  $T \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$
return $C = C_1||C_2||...||C_{t-1}||C'_t$, T

Algorithm 2 Verified Decryption ($\mathcal{D}_{k,r,a,b}(K, N, A, C, T) = \{P, \perp\}$)

Inputs: K, N, A, C, T
Outputs: P or $\perp$
Initialisation:
  $S \leftarrow IV_{k,r,a,b}||K||N$
  $S \leftarrow p^a(S) \oplus (0^{320-k}||K)$
Processing Associated Data:
if $|A| > 0$ **then**:
  $A_1, A_2, ..., A_s \leftarrow r$-bit blocks of $A||1||0^*$
  for i in range 1 to s **do**:
    $S \leftarrow p^b((S_r \oplus A_i)||S_c)$
  end for
  $S \leftarrow S \oplus (0^{319}||1)$
end if
Processing Ciphertext:
  $C_1, C_2, ..., C_{t-1}, C'_t \leftarrow r$-bit blocks of C, where $0 \leq |C'_t| \leq r$
for i in range 1 to $t-1$ **do**:
  $P_i \leftarrow S_r \oplus C_i$
  $S \leftarrow C_i||S_c$
  $S \leftarrow p^b(S)$
end for
  $P'_t \leftarrow \lfloor S_r \rfloor_{|C'_t|} \oplus C'_t$
  $S_r \leftarrow S_r \oplus (P'_t||1||0^*)$
Finalization:
  $S \leftarrow p^a(S \oplus (0^r||K||0^{320-r-k}))$
  $T^* \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$
if $T^* = T$ **then**
  return $P_1||P_2||...||P_{t-1}||P'_t$
else
  return $\perp$
end if

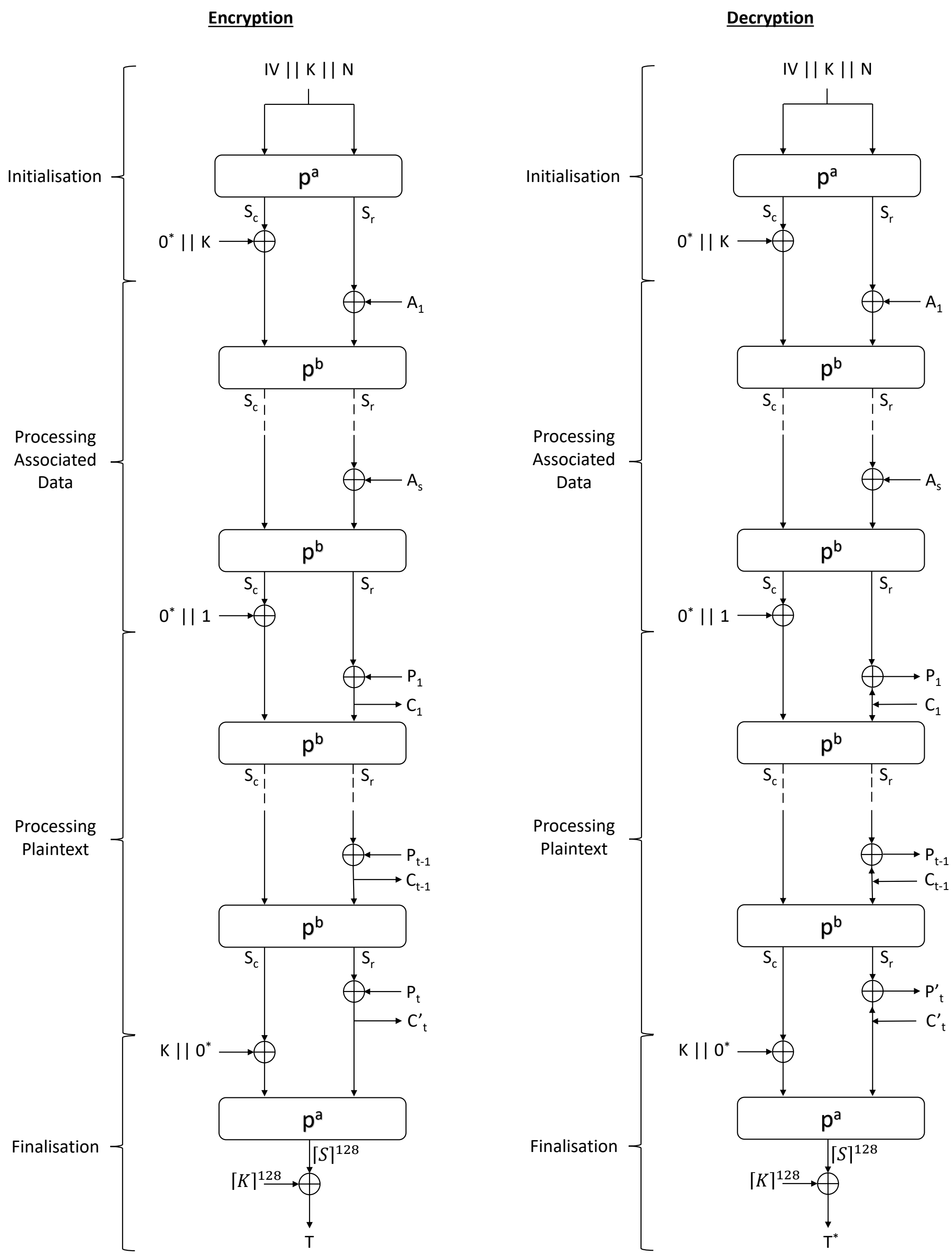

Figure 1: ASCON Algorithm Diagram

2.3.2 Initialisation

In the initialisation phase, an IV is first formed. To do so, secret key K of length k bits and a rate r is chosen. Values a and b for the permutation rounds are also chosen. We then assign the IV as follows.

$$IV \leftarrow k||r||a||b||0^{160-k} \tag{1}$$

In the case of ASCON-128, with reference to the specifications in section 2.2, $k = 128$, $a = 12$ and $b = 6$. Also, r is set to 64. Expressing each term k, r, a, b as 8 bit integers, we have the IV for ASCON-128 as follows.

$$IV = 0x80400c0600000000 \tag{2}$$

From here, the 320-bit state S assigned, permuted with a rounds of the round function p and XORed with secret key K.

$$\begin{aligned} S &\leftarrow IV||K||N \\ S &\leftarrow p^a(S) \oplus (0^{320-k}||K) \end{aligned} \tag{3}$$

It is to be noted that the state S can be represented as follows, $S = S_r||S_c = x_0||x_1||x_2||x_3||x_4$. Here, S_r and S_c represent the outer r and inner c bits of S, where $c = 320 - r$. Also, each x_i is a 64 bit block which is a partition of state S.

2.3.3 Processing Associated Data, Plaintext, Ciphertext

When the length of the associated data A is non-zero, we proceed to pad it with a 1-bit and 0-bits to attain a total length which would be a multiple of r. This is then split into r-bit blocks $A_1, A_2, ...A_s$. The same is done for the plaintext. More precisely,

$$\begin{aligned} A_1, A_2, ..., A_s &\leftarrow A||1||0^{r-1-(|A| \mod r)} \text{ split into } r\text{-bit blocks} \\ P_1, P_2, ..., P_t &\leftarrow P||1||0^{r-1-(|P| \mod r)} \text{ split into } r\text{-bit blocks} \end{aligned} \tag{4}$$

For the ciphertext, padding is only used in the last assignment of S_r in the decryption process. That is,

$$S_r \leftarrow S_r \oplus (P_t'||1||0^{r-1-|C_t'|}) \tag{5}$$

The rest of the details for processing and finalisation can be obtained directly from Algorithms 1 and 2.

2.3.4 Round function

The round function consists of three layers, constant addition p_c, substitution p_s and linear diffusion p_l such that round function $p = p_l \circ p_s \circ p_c$. We describe a single round of p. To begin, 320-bit state S is expressed as $x_0||x_1||x_2||x_3||x_4$ and constant addition is first applied.

	p^{12}	p^8	p^6
Index	Constant		
0	$0xf0$	$0xb4$	$0x96$
1	$0xe1$	$0xa5$	$0x87$
2	$0xd2$	$0x96$	$0x78$
3	$0xc3$	$0x87$	$0x69$
4	$0xb4$	$0x78$	$0x5a$
5	$0xa5$	$0x69$	$0x4b$
6	$0x96$	$0x5a$	
7	$0x87$	$0x4b$	
8	$0x78$		
9	$0x69$		
10	$0x5a$		
11	$0x4b$		

Table 3: Constant Addition Table

Depending on the round (indexed starting from 0) of permutation being run and the number of permutation rounds to be scheduled, we XOR the hexadecimal constant specified in Table 3 with x_2. For instance, if p^{12} were to be run and it was currently undergoing round 4 of the 12 permutation rounds, we have,

$$x_2 \leftarrow x_2 \oplus 0xc3 \tag{6}$$

From here, substitution is then applied. The five registers x_i are grouped into 5-bit blocks where if $x_i[j]$ were to represent the j^{th} bit in x_i, the j^{th} 5-bit slice would be $x_0[j]||x_1[j]||x_2[j]||x_3[j]||x_4[j]$. From here, each 5-bit slice denoted x is entered into the 5-bit substitution box. The substitution values can be observed from Table 4.

S-box																
x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$s(x)$	4	b	$1f$	14	$1a$	15	9	2	$1b$	5	8	12	$1d$	3	6	$1c$

x	10	11	12	13	14	15	16	17	18	19	$1a$	$1b$	$1c$	$1d$	$1e$	$1f$
$s(x)$	$1e$	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Table 4: S-box lookup table (hexadecimal)

Lastly, the linear diffusion layer is applied as follows,

$$\begin{aligned}
x_0 &\leftarrow x_0 \oplus (x_0 \gg 19) \oplus (x_0 \gg 28) \\
x_1 &\leftarrow x_1 \oplus (x_1 \gg 61) \oplus (x_1 \gg 39) \\
x_2 &\leftarrow x_2 \oplus (x_2 \gg 1) \oplus (x_2 \gg 6) \\
x_3 &\leftarrow x_3 \oplus (x_3 \gg 10) \oplus (x_3 \gg 17) \\
x_4 &\leftarrow x_4 \oplus (x_4 \gg 7) \oplus (x_4 \gg 41)
\end{aligned} \tag{7}$$

3 Deep Learning Preliminaries

We first provide a background for the concepts and terminologies to be used in the network architecture.

3.1 Loss functions

In deep learning, learning algorithms can often be framed in terms of the minimisation or maximisation of some objective function. By minimising or maximising the objective function, in some sense, the optimal parameters for prediction can be obtained. In the case of minimization, this is referred to as a loss function. An instance of a loss function used in this study would be the Mean Squared Error(MSE) (Goodfellow et al., 2016).

Letting $\hat{y}_i$ be the output predicted by the model and y_i the true value of the target variable of n samples, MSE is defined as follows,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{8}$$

More generally, a loss function J is expressed as

$$J(\theta) = \mathbb{E}_{(x,y)\ \hat{p}_{data}}\, L(f(x;\theta), y) \tag{9}$$

where L is the per-example loss function, f is the predicted output from data x and parameter θ, y the target output and $\hat{p}_{data}$ the empirical distribution.

3.2 Optimisation

To minimise the aforementioned loss function, there exists a myriad of optimisation algorithms suited to the task. As analytic methods for solving optimisation problems tend to be costly, iterative methods are usually preferred. Majority of the optimisation methods performed are often referred to as minibatch stochastic methods, where more than one but less than the whole training set is used at once in the optimisation process (Goodfellow et al., 2016).

We first present the concept of momentum, followed by its usage in the Adam optimisation algorithm. Let v be unit velocity and $\alpha \in [0, 1)$, ϵ be parameters chosen by the user. Then, the update process using momentum is as follows,

$$\begin{aligned} &v \leftarrow \alpha v - \epsilon \nabla_\theta\Big(\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})\Big) \\ &\theta \leftarrow \theta + v \end{aligned} \tag{10}$$

Here, v accumulates the gradient elements $\nabla_\theta(\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)}))$ and then updates the parameter θ accordingly. A larger value of α relative to ϵ increases the effects of previous gradients on the current direction. From this, the Adam algorithm is described in Algorithm 3.

Algorithm 3 Adam Optimiser

Fixed parameters (default value):

Step size $\epsilon(0.001)$

Exponential decay rates for moment estimates$\rho_1(0.9), \rho_2(0.999) \in [0, 1)$

Constant $\delta(10^{-8})$ for numerical stabilisation

Initial Parameter: θ

Initialisation:

Set first and second moment variables, $s, r = 0$

Set time step $t = 0$

while stopping criterion not attained **do**

Sample minibatch of size m from training set $x^{(1)}, ...x^{(m)}$ (Includes target variables $y^{(i)}$.)

Compute gradient $g \leftarrow \frac{1}{m}\nabla_\theta \sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)})$

Update biased moments $s \leftarrow \rho_1 s + (1-\rho_1)g, r \leftarrow \rho_2 r + (1-\rho_2 g) \odot g$ ($\odot$ represents the Hadamard product)

Update time step $t \leftarrow t + 1$

Correcting bias in moments $\hat{s} \leftarrow \frac{s}{1-p_1^t}, \hat{r} \leftarrow \frac{r}{1-p_2^t}$

Update $\theta \leftarrow \theta - \epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$

end while

3.3 Convolutional Neural Networks

Convolutional Neural Networks are most often used in image recognition tasks (Sultana, Sufian, & Dutta, 2018). They typically consist of three stages. First, multiple convolutions are performed in parallel using the convolution function. This produces a set of linear activations. From here, these activations are run through a non-linear activation function. Finally, a pooling function is used (Goodfellow et al., 2016).

The convolution function s can be interpreted as a weighted averaging of several measurements. More formally, in the case of a single variable t, it is denoted as

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{11}$$

Here, x is the input function and w the weight function. Intuitively, in the case of a two-dimensional input array, one may visualise it as the dragging of a grid/filter, known as the kernel, containing weights through the array. At each step, the weights contained in the grid have the dot product computed with the weights in the array and then passed through an activation function, resulting in a new two-dimensional array. From here, a final array is obtained via a pooling function. It is to be noted that the size of each step is defined as the stride. Figure 2 provides a visualisation for the kernel.

Pooling can be regarded as an attempt to provide summary statistics for the data. For instance, in the case of max pooling, a new grid of dimensions 2x2 may be dragged through the array, which would output a new array with

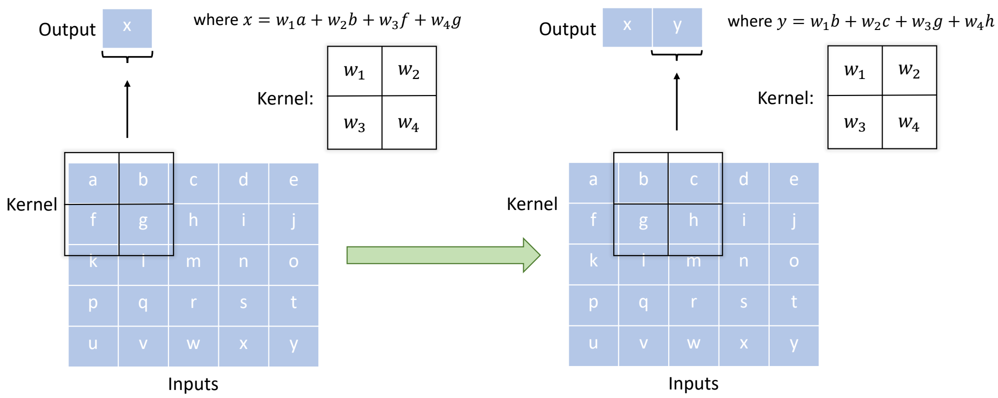

Figure 2: Convolution Diagram

entries being the maximum values of the 2x2 grid at each step.

3.4 Residual Learning

The primary goal of residual neural networks (ResNets) was to correct the degradation problem of neural networks (He, Zhang, Ren, & Sun, 2016). With more layers added into a neural network, the increased depth causes a saturation of accuracy (does not improve) and eventually, a rapid decrease of accuracy. This had not stemmed from overfitting but rather from the increase in the number of layers and is known as the degradation problem.

The solution presented by ResNets was to introduce shortcut connections between layers. For the purposes of this paper, shortcut connections perform the identity mapping and skip one or more layers. These add the inputs x to the output of a set number of layers $F(x)$. A block consisting of layers and a shortcut is defined to be a residual block. A visualisation of the features mentioned is provided in Figure 3.

3.5 Cyclical Learning Rates

Learning rates form a fundamental component of deep neural networks. In the case where stochastic gradient descent is used for parameter optimisation, the parameters are updated as follows,

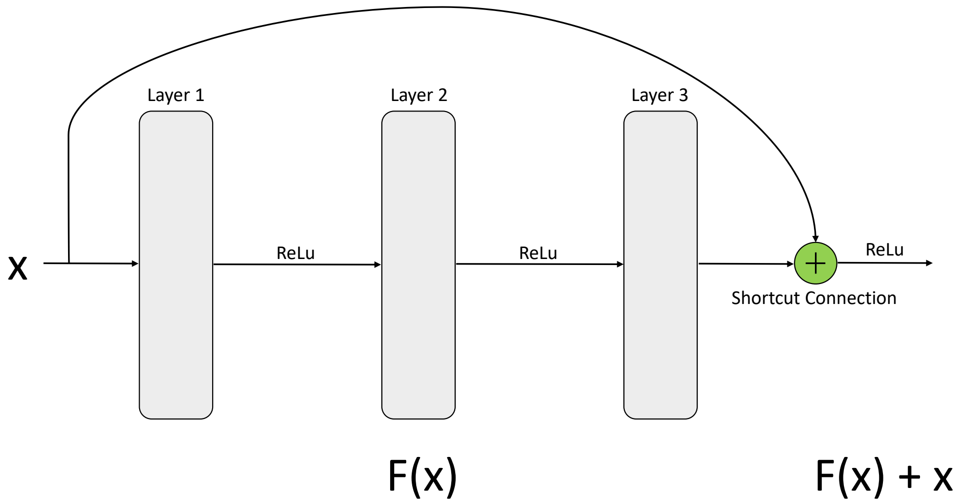

Figure 3: ResNet Block Diagram

$$\theta^t = \theta^{t-1} - \epsilon_t \frac{dL}{d\theta} \tag{12}$$

where t is the time step, L the loss function and ϵ_t the learning rate. The learning rate ϵ_t is typically a single value which decreases during training. Cyclical learning rates (CLR) represent a variant of learning rates and have been used extensively with ResNets (Smith, 2017). Here, the primary advantage that CLR offers would be that little to no additional computation cost is required for its implementation. This is in contrast to adaptive learning rates.

Conceptually, CLR allows for the learning rate to vary between a set range of values. Maximum and minimum values are chosen, with a corresponding step size. The learning rate then varies between the two values depending on the cycle length. Here, cycle length is defined as the number of iterations/batches until the learning rate returns to the initial value and step size, the number of iterations per half cycle. An illustration of this can be observed from Figure 4.

For the selection of the optimal cycle length, it is recommended to have $stepsize = n * numberofepochs$ where n is an integer between 2 and 10.

For the selection of minimum and maximum boundary values for the learning rate, the LR range test is employed. Here, the model is run for several epochs whilst the learning rate is incremented linearly between low and high values. Accuracy is plotted against learning rate and the plot is analysed. The learning rate value where accuracy begins to increase is set as the minimum value. The learning rate value where accuracy decreases, slows, or exhibits ragged behaviour

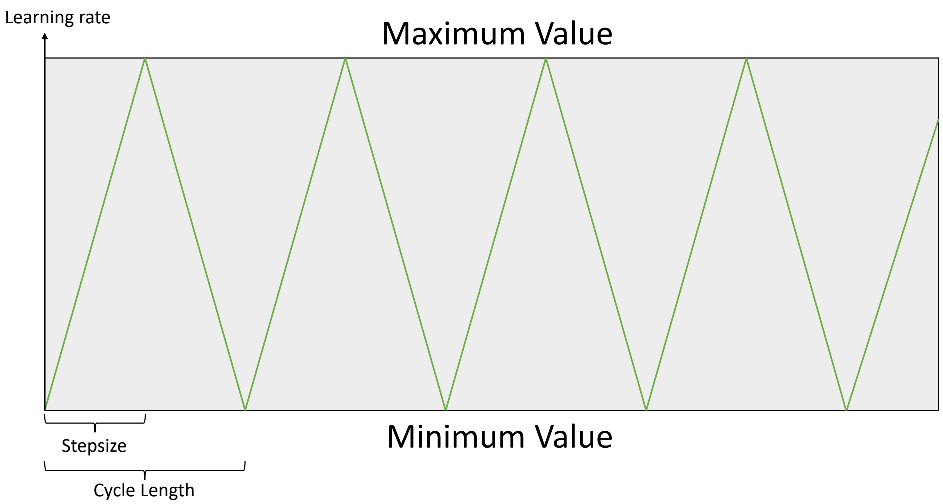

Figure 4: Cyclical Learning Rate (triangular learning rate policy) Diagram

is set as the maximum value. It is to be noted that this could also be done with a loss against learning rate plot.

3.6 Activation Functions

Here, we provide a brief listing of the activation functions to be used later on.

$$\text{Rectified Linear Unit (ReLU): } g(z) = \max\{0, z\} \tag{13}$$

$$\text{Sigmoid: } \sigma(x) = \frac{1}{1 + \exp(-x)} \tag{14}$$

4 Adaptation of the Neural Distinguisher

4.1 Overall Structure

A set of 10^6 320-bit plaintexts is first generated at random. From here, for a specified fixed difference, each of the plaintexts are XORed with the fixed difference with probability 0.5 to obtain a second 320-bit plaintext. Those that are not XORed with the difference have the corresponding second 320-bit plaintext randomly generated. The status of whether the plaintext was XORed is noted down. This way, 10^6 plaintext pairs $(plain_1, plain_2)$ are obtained.

From here, each plaintext is run through the round function p of the ASCON cipher for a specified number of rounds n, i.e. p^n. From this, corresponding

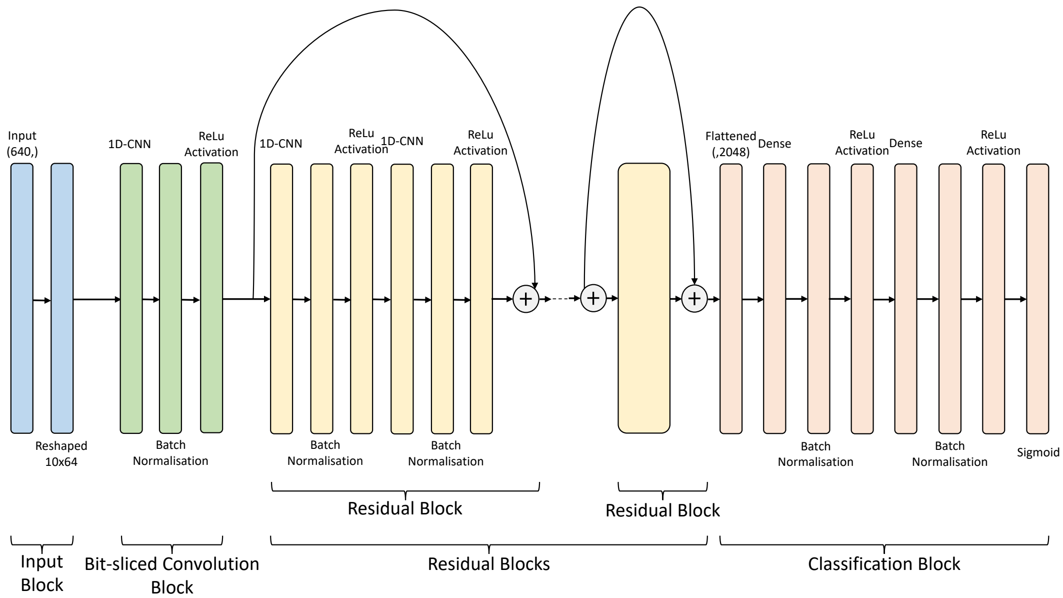

Figure 5: Neural Distinguisher Diagram

320-bit outputs (out_1, out_2) for 10^6 samples are obtained. We attempt to train the network to classify these outputs, separating the output pairs with the fixed difference from those with a random one. It is noted that the coded implementation is available in the appendix.

4.2 Network Architecture

We adopt a similar variant of the base network architecture presented in Gohr's paper. That is, the neural network first begins with a single bit-sliced convolution. It then takes the form of a residual tower of two-layer convolutional neural network and ends with a densely connected prediction head. Similar to the original architecture, the main parameter to be varied in the tuning of the network would be the depth of the residual tower. The Adam optimiser is used in the compilation. An overall illustration of the network architecture is provided in Figure 5.

4.2.1 Input

The network first takes as input an array of shape $(640,)$. This array is obtained from concatenating the two 320-bit outputs out_1 and out_2. From here, the input is reshaped into shape 10x64 and connected channels-first mode(transposed) into the bit-sliced convolution block. The 10x64 shape reflects an attempt to emulate the 64-bit partition of the state $S = x_0||x_1||x_2||x_3||x_4$, similar to how the original architecture mirrored the word-oriented structure of Speck32/64.

4.2.2 Bit-sliced Convolution Block

The bit-sliced convolution block starts with a 1 dimensional convolution (1D-CNN) consisting of 32 filters, no padding, kernel size 1, stride 1 and has output shape 64x32. After the filters of the 1D CNN are processed, a batch normalisation layer is then applied. Following this, the ReLu activation layer is used to obtain the output. The output of this block is then fed into a collection of residual blocks and is also connected via a shortcut to each residual block's output.

4.2.3 Residual Blocks

Each residual block first begins with a 1D-CNN with 32 filters, padding size 1, stride 1 and kernel size 3. From here, a batch normalisation layer followed by a ReLu activation function layer is applied. This is then repeated, continuing with another 1D-CNN with same specifications, batch normalisation and a ReLu activation function. Finally, the output is connected to the initial input of the residual block via a shortcut connection. The output and initial inputs are added, resulting in the final 64x32 output of the residual block. This is then passed into the subsequent residual block(assuming it is not the last) and the process repeats for the chosen depth/number of residual layers.

4.2.4 Classification Block

Finally, the classification block takes the 64x32 output of the residual block and flattens it into shape $(, 2048)$. This is fed into a dense layer with 64 units, followed by a batch normalisation layer and a ReLu activation function layer. The dense layer, batch normalisation and ReLu is then repeated again. The classification block then ends off with a final dense layer with a sigmoid activation function for the classification output.

4.3 Initial results

With the aid of the secrets module in python, we generated the 10^6 samples. Each sample consisted of an “x” value which was an array consisting of 640 bits from concatenating the outputs of the round function as well as a ”y” value which had value 1 if the specified input difference was used and 0 if the input difference was random. These were then fed into the neural network, with a 10/90 split in validation and training data. That is, 10^5 samples were used for validation and 9×10^5 were used for training. Batch sizes of 5000 were fed into the neural network.

Following this, fixed input differences $2^{63} + 2^{127}$ and 2^{63} were used. These had been chosen as they correspond to probability 1 truncated differentials for 4 and 3.5 rounds respectively (Tezcan, 2016). From here, for input difference $2^{63} + 2^{127}$, the neural network was run for 10 epochs with depth of the residual layer ranging from 1, 3, 5, 8 and 10 blocks. This was also conducted with a cyclical learning rate schedule where for epoch i, the corresponding learning rate was $\alpha + \frac{(n-i) \mod (n+1)}{n} \times (\beta - \alpha)$, where $\beta = 0.002$, $\alpha = 0.0001$. Here, β and α are the maximum and minimum values chosen respectively, as described in subsection 3.5.

It was found that for data that had undergone 1 to 3 rounds of permutation in the round function, a classification accuracy close to value 1 was attained. However, for data that had undergone 4 to 6 rounds of permutation in the round function, classification accuracy was closer to 0.5 in value.

Similarly, for input difference 2^{63}, 10 epochs were run with a neural network whose residual layer had depth 1. It was also found that the data that underwent rounds 1 to 3 of permutation had high classification accuracy close to value 1 while data that underwent rounds 4 to 6 of permutation had accuracy close to 0.5.

Listed in Table 5 are the best results for each specified input difference and specified number of permutation rounds. In the event that multiple values of residual layer depth attain the same accuracy, the lowest depth is recorded.

Best Results			
Input difference	Permutation Rounds	Depth	Validation Accuracy
$2^{63} + 2^{127}$	1	1	1
$2^{63} + 2^{127}$	2	1	1
$2^{63} + 2^{127}$	3	1	0.999899983406066
$2^{63} + 2^{127}$	4	5	0.503260016441345
$2^{63} + 2^{127}$	5	8	0.503979980945587
$2^{63} + 2^{127}$	6	8	0.503419995307922
2^{63}	1	1	1
2^{63}	2	1	1
2^{63}	3	1	0.999490022659301
2^{63}	4	1	0.504610002040863
2^{63}	5	1	0.501489996910095
2^{63}	6	1	0.502149999141693

Table 5: Classification Results

4.4 Improved results

We attempted to improve upon the baseline results in 5 by varying both parameters and architecture of the network. The primary focus of this was to allow the network to learn for permutation rounds 4 and above. The parameters - learning rate, number of epochs from 10 to 200, residual layer kernel size from 1 to 8, batch sizes 500 and 5000, filter number 32, 64 and 128, activation function stochastic gradient descent(SGD), the number of convolutional blocks in the residual layer(depth) and input shapes 10x64, 64x10, 5x128, 128x5, with and without channels-first mode, were tested individually. Of these, it was found that only changing learning rate had an improvement on the validation accuracy.

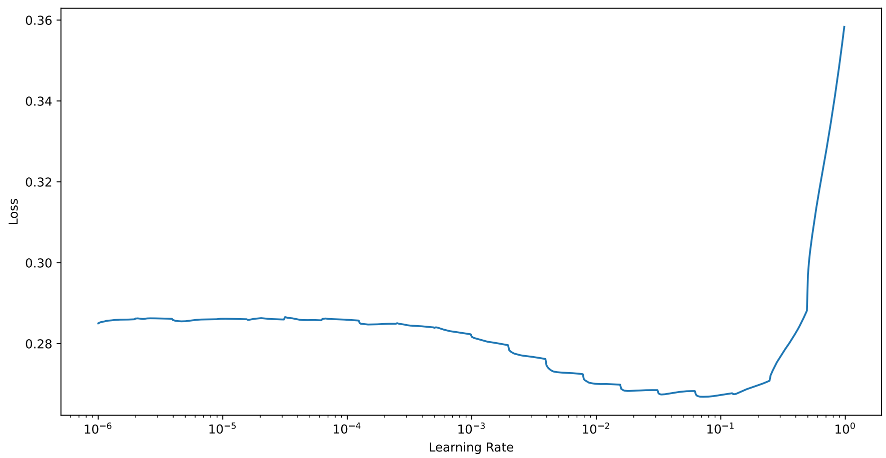

Figure 6: Learning Rate Diagram

As observed from the Figure 6 where the LR range test is performed, the maximum and minimum values lie around 10^{-1} and 10^{-3} respectively. Upon experimentally varying the boundaries, it was found that having the maximum and minimum values set to 0.02 and 0.0001 yielded the best results in validation accuracy. This was run for 50 epochs with the configurations specified in Table 6.

Best Results			
Input difference	Permutation Rounds	Depth	Validation Accuracy
$2^{63} + 2^{127}$	3	1	0.999989986
$2^{63} + 2^{127}$	4	1	0.519309997558593

Table 6: Classification Results

From Figures 7 and 8, it can be observed that the accuracy saturates and is marginally higher than that observed in Table 5.

5 Further Architectures

From here, further attempts at adjusting the network architecture were made. It is noted that the code for each variant is included in the appendix.

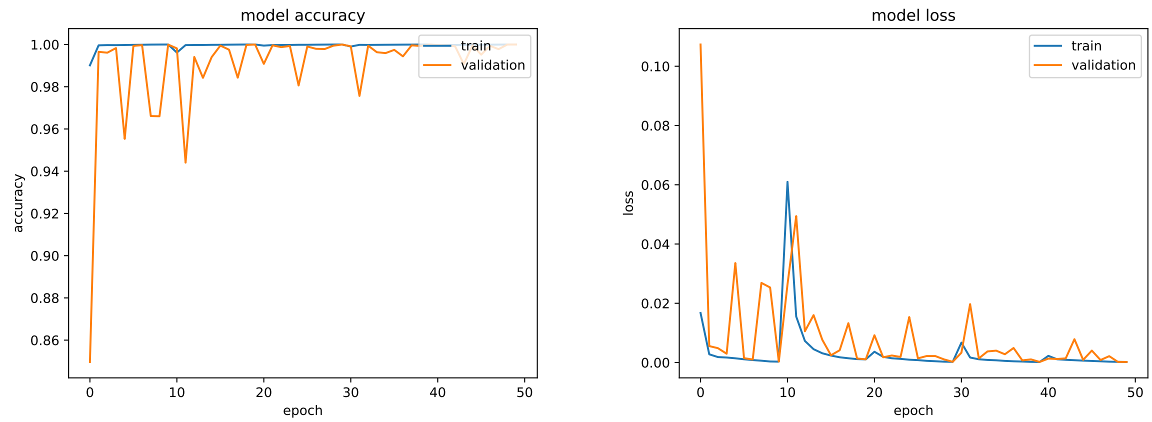

Figure 7: Accuracy and Loss plots of 3 permutation rounds with depth 1

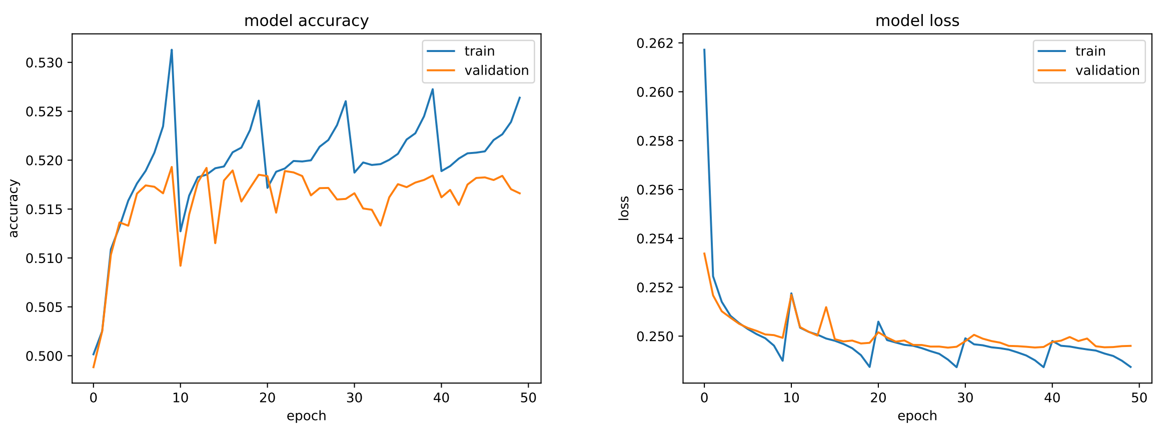

Figure 8: Accuracy and Loss plots of 4 permutation rounds with depth 1

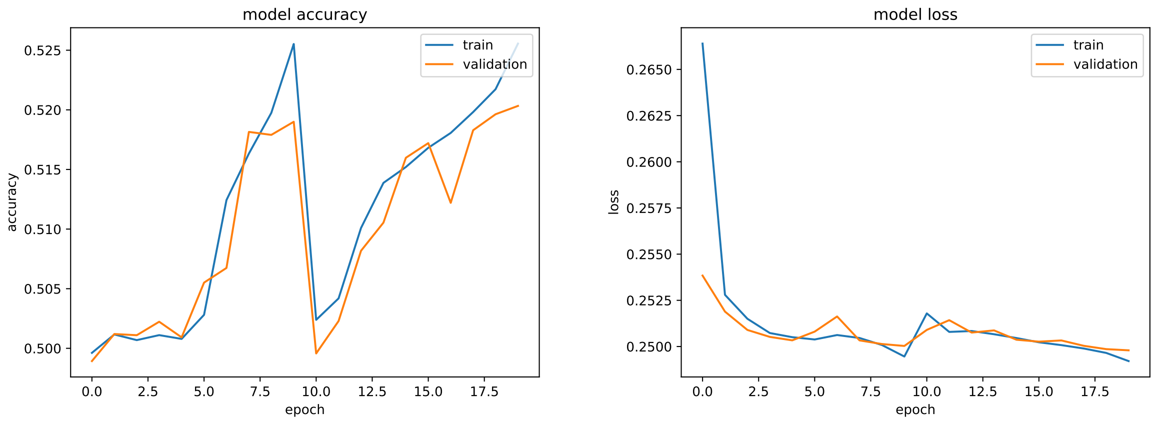

Figure 9: Accuracy and Loss plots of Reshaped inputs with kernel size 9

5.1 Input reshaping for rotations

5.1.1 Motivation and specifications

As will be discussed in later sections, training the network for 3.5 rounds of the round function had yielded high validation accuracy that was close to 1. That is, 3 rounds of the round function followed by only the constant addition and substitution layers. This, coupled with the accuracy dropping significantly once applied to the 4 round case, suggests that the difficulty faced by the network would lie in learning the linear diffusion layer. As an attempt to account for this, we proposed and tested the following input structure which was then fed into the bit-sliced convolution block of the network. The remainder of the network its specifications followed that of the results in Figure 6.

Here, we structured the inputs to have shape 10x128. For each row i of the 10x128 data, it will be of form $(x_{i,out_j}, x_{i,out_j})$, where the first 5 rows are for out_1 and the next 5 for out_2. An illustration of this is provided in Figure 10.

The motivation for this structure was to allow the network to capture the rotations of the linear diffusion layer, as the convolutions moved across each row. Different kernel sizes would be tested in capturing different degrees of rotation.

5.1.2 Results

After running for 20 epochs with a depth 1 residual layer and kernel sizes from 3 to 32 (note that only up to 32 is required to account for rotations), it was found that the highest validation accuracy of 0.520319998 was attained at kernel size 4, providing only marginal improvement of approximately 0.001. An instance of these experiments is provided in Figure 9.

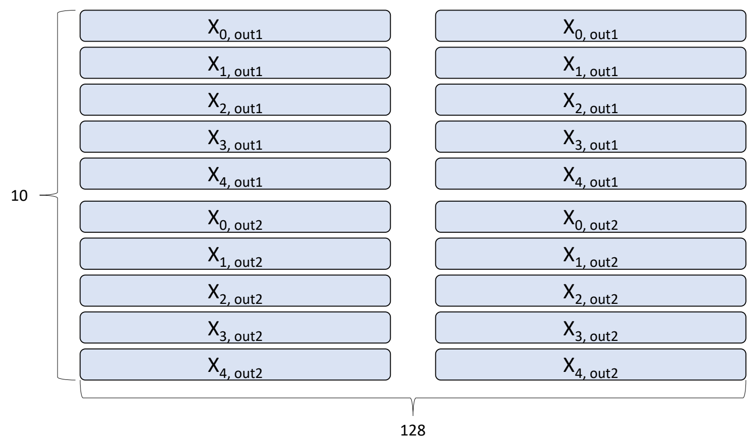

Figure 10: Reshaped Inputs

5.1.3 Adjustments

Following this, we further adjusted the kernel sizes of the residual layer to account for the rotations. To be precise, the depth of the residual layer was set to 7, having 7 residual blocks. The kernel sizes were chosen to be 3, 6, 7, 10, 17, 19, 28 from the various degrees of rotation for each block respectively. This was conducted in with the motivation that each residual block could perhaps individually capture aspects of each rotation.

This was run for 10 epochs. However, the highest validation accuracy attained was only 0.50119.

5.2 Multiple Differential

5.2.1 Background

The second variant involved attempting to make use of multiple differentials. For background, differential cryptanalysis involves the use of input and output differences to obtain key values of a cipher (Biham & Shamir, 1991). In brief, it starts by considering the highest probability input and output differences (α, β) to a round function. From here, inputs (x_1, x_2) that have difference α, are fixed, with round function outputs (y_1, y_2) and ciphertexts (c_1, c_2) obtained. Following this, different key values are brute-forced to map the ciphertexts (c_1, c_2) to the round function outputs and the key value which returns output difference β most frequently will likely be the correct value. An illustration is provided in Figure 11. Multiple differential cryptanalysis seeks to expand upon this by allowing for a set of input differences to be considered together, where their output differences can be different (Blondeau & Gérard, 2011).

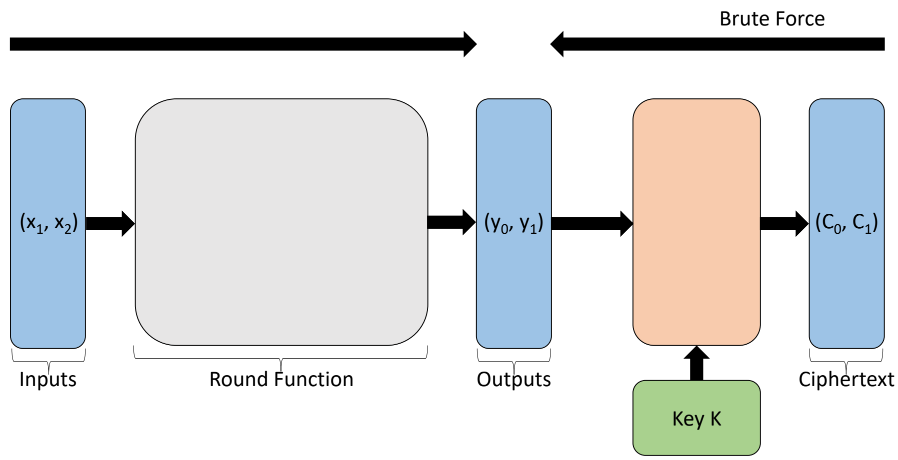

Figure 11: An illustration of differential cryptanalysis

5.2.2 Specifications

With this as motivation, we attempted to adapt a similar structure to this. To emulate this, we considered inputs (x_0, x_1, x_2, x_3), with each x_i being a 320-bit input to the round function. They were also chosen such that either both pairs, x_0, x_1 and x_2, x_3, have a fixed input difference or that both have random differences. Intuitively, the idea behind this was to increase the likelihood of the network detecting randomness by having multiple pairs which were related to each other. The remainder of the network followed the initial structure. We then attempted to observe if such a structure would improve the results of the 4 round case.

From here, all combinations of the following specifications were used. Input shapes of 10x128 and 20x64 were tested. The kernel size of the convolutional layers in the residual block was kept to 3 and the number of filters 32. The depth of the residual layer was set to either 1 or 5. Batch size was set to 5000 and the data generated was of size 10^6, used with a 10/90 split for validation and training. The difference was set to $2^{64} + 2^{127}$ for both differences and this was run for 20 epochs.

5.2.3 Results

Upon running, it was found that the validation accuracy was lower than that of the best observed in Table 6. The highest validation accuracy obtained was only approximately 0.502.

5.3 Biased Inputs

5.3.1 Bias Computation

To further gauge the viability of the experimental set up, we computed the biases of the data. The bias was computed as the probability that two bits, that are inputs for the network, would not share the same bitwise value. Specifically, for the 640-bit input data, the bias for entry i out of 320 was computed by taking XOR of entry i of the two 320-bit inputs. This XOR value was then counted if the value was 1 and the total count was averaged over the size of the dataset. Generally, it was observed that there was a significant drop in bias from the 3.5 rounds to 4 rounds of ASCON round function applied.

5.3.2 Input Modification

From here, we restricted the 640 bit input of the neural network to the bitwise entries with the highest bias values. Specifically, only the entries with bias deviating from 0.5 by at least 0.001 (≤ 0.499 and ≥ 0.501) had their inputs included. The bitwise entries that did not satisfy this criterion had their values set to 0. Intuitively, this was done to reduce the "random noise" induced by input bits whose difference was essentially random. At the same time, this would allow the network to better focus on the bits which carried more information.

This was conducted for data which had undergone 4 rounds of the round function and had input difference $2^{64}+2^{127}$ used. The table of biases is included as Figure 12. Here, the bitwise entries with the highest deviation from 0.5 in bias are highlighted in yellow. These are the bitwise entries included in each 320 bit neural network input which would make up the 640 bit input.

5.3.3 Results

The network was run on this input with specifications as per the results in Table 6. The highest validation accuracy obtained was 0.5216. The accuracy and loss diagrams are included in Figure 13. This was also rerun with the same specifications but with depth 3 for the residual layer, which yielded highest validation accuracy of 0.5223.

It is to be noted that, while the results produced are similar to those of the reshaped inputs, this had improvement in terms of network learning efficiency compared to that of the reshaped inputs. Each epoch had only taken approximately 80 seconds to run, while the reshaped inputs modification required anywhere from approximately 180 seconds to 590 seconds depending on kernel size, for each epoch.

5.4 Transfer Learning

5.4.1 3.5 rounds

The fourth variant we attempted involved attempting to apply transfer learning. Informally, transfer learning involves training a network in one setting, to have

Input Diff	Permutation Rounds																																																																	
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**63+2**127	3	x0	0.499867	0.480054	0.500189	0.492359	0.452553	0.468374	0.484739	0.499467	0.313419	0.453015	0.500599	0.500129	0.406414	0.515765	0.453002	0.43708	0.499807	0.499317	0.500707	0.500131	0.487817	0.375244	0.499633	0.498743	0.468507	0.499834	0.488619	0.312864	0.500077	0.499423	0.374699	0.452579	0.406978	0.476279	0.452933	0.500468	0.500192	0.499946	0.499104	0.515612	0.437555	0.438327	0.499522	0.374827	0.500251	0.500879	0.499788	0.476891	0.453373	0.499878	0.250322	0.4993	0.374667	0.429345	0.406632	0.406484	0.515498	0.500612	0.496716	0.406652	0.469585	0.476404	0.421161	0.374509
		x1	0.506991	0.500615	0.500917	0.499844	0.499548	0.50017	0.437048	0.499504	0.407238	0.375414	0.499958	0.485295	0.374965	0.50058	0.491889	0.468504	0.499767	0.453132	0.437621	0.531612	0.453269	0.43828	0.484346	0.499827	0.406079	0.499541	0.499684	0.250017	0.531578	0.405968	0.249882	0.468971	0.499498	0.468438	0.375601	0.499706	0.476651	0.406722	0.500365	0.497814	0.452751	0.500617	0.476496	0.438012	0.500252	0.483447	0.500023	0.469014	0.375453	0.437245	0.375009	0.405956	0.312462	0.499149	0.24979	0.499207	0.468413	0.562185	0.507008	0.437532	0.500641	0.495976	0.452935	0.40571
		x2	0.495621	0.516847	0.499297	0.500124	0.593295	0.547531	0.512805	0.484929	0.499711	0.499912	0.500861	0.45291	0.500403	0.500044	0.405985	0.406345	0.546998	0.546781	0.452887	0.515847	0.483938	0.249689	0.593328	0.437634	0.421577	0.405946	0.43723	0.313332	0.405923	0.444305	0.374567	0.499753	0.31317	0.249912	0.405927	0.344477	0.421416	0.40536	0.547533	0.484738	0.563258	0.499989	0.500412	0.375486	0.547097	0.54654	0.523465	0.429423	0.313402	0.406714	0.375132	0.49994	0.453342	0.375514	0.312758	0.374721	0.373873	0.406201	0.530594	0.530358	0.313214	0.547255	0.562045	0.499805
		x3	0.50435	0.500191	0.499317	0.499538	0.593717	0.375261	0.499371	0.516165	0.499799	0.405532	0.507516	0.531771	0.249706	0.507963	0.515092	0.499765	0.523123	0.500709	0.499629	0.484625	0.500045	0.49949	0.499785	0.499833	0.469363	0.579001	0.546906	0.405988	0.499955	0.482834	0.407493	0.405848	0.249791	0.578648	0.429872	0.503894	0.496122	0.312677	0.536376	0.546844	0.499567	0.43888	0.578758	0.374724	0.311932	0.49938	0.524751	0.437748	0.515335	0.562767	0.500309	0.452898	0.437919	0.405926	0.249874	0.51186	0.484192	0.500042	0.53033	0.437767	0.374988	0.547182	0.406416	0.500286
		x4	0.500301	0.493457	0.42182	0.471399	0.49208	0.480278	0.445567	0.500915	0.406659	0.249662	0.499689	0.476854	0.406249	0.465581	0.453459	0.50059	0.49991	0.405798	0.43748	0.453625	0.31215	0.453428	0.406058	0.500013	0.499817	0.496461	0.508234	0.312278	0.49975	0.406735	0.49931	0.374581	0.421172	0.499853	0.31289	0.500388	0.359675	0.375061	0.499941	0.484618	0.311964	0.500349	0.453565	0.406012	0.406384	0.464638	0.495552	0.417177	0.500297	0.499671	0.250314	0.499952	0.469322	0.40603	0.312523	0.461731	0.312087	0.499968	0.500437	0.500353	0.476852	0.484557	0.311947	0.500142
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**63+2**127	4	x0	0.499654	0.501094	0.499645	0.500356	0.500426	0.497587	0.499465	0.499563	0.499308	0.500495	0.49951	0.500246	0.500226	0.500394	0.500202	0.499457	0.499734	0.499672	0.494137	0.499618	0.499591	0.500771	0.499931	0.500761	0.499871	0.500156	0.499203	0.494945	0.500262	0.499901	0.500138	0.499655	0.500758	0.499058	0.499827	0.499329	0.501191	0.498385	0.500105	0.499565	0.498769	0.499546	0.499913	0.49922	0.49956	0.499613	0.49859	0.499961	0.50068	0.517127	0.498089	0.499671	0.498852	0.499714	0.499877	0.500434	0.500073	0.500257	0.499831	0.500558	0.499568	0.500342	0.499843	0.499569
		x1	0.500108	0.499942	0.499888	0.49988	0.499823	0.500173	0.500045	0.500054	0.500485	0.498808	0.500174	0.50005	0.500184	0.500296	0.500862	0.500174	0.499616	0.500047	0.499454	0.500299	0.500584	0.500254	0.499608	0.500443	0.49933	0.499406	0.499663	0.498603	0.500224	0.499919	0.500041	0.500167	0.499449	0.499333	0.501929	0.500652	0.499923	0.500661	0.499936	0.499199	0.499766	0.499947	0.500001	0.50041	0.499809	0.499631	0.500169	0.49913	0.500772	0.500432	0.500008	0.499806	0.499218	0.501074	0.499253	0.499393	0.500468	0.499725	0.498925	0.501022	0.499765	0.499613	0.500161	0.499907
		x2	0.498851	0.500483	0.499403	0.499551	0.499981	0.500071	0.499761	0.49992	0.500348	0.499511	0.500459	0.500321	0.499453	0.499236	0.500247	0.500408	0.500499	0.500356	0.504808	0.500268	0.500891	0.500223	0.500857	0.500082	0.500231	0.499706	0.4997	0.49874	0.500809	0.500839	0.499784	0.500172	0.498363	0.500244	0.499595	0.499449	0.499742	0.499166	0.499248	0.500215	0.500452	0.499232	0.498908	0.500363	0.499106	0.499647	0.500121	0.49967	0.49945	0.499296	0.500389	0.501489	0.499372	0.501091	0.499033	0.500453	0.499919	0.500127	0.49964	0.499609	0.504976	0.500072	0.49947	0.49808
		x3	0.499894	0.5005	0.499694	0.499787	0.500122	0.502457	0.499439	0.499826	0.500255	0.500383	0.50029	0.499592	0.50026	0.498932	0.500926	0.502531	0.500039	0.499845	0.499456	0.500453	0.499629	0.499619	0.500208	0.499939	0.500447	0.499768	0.499444	0.499767	0.49986	0.50024	0.499768	0.499016	0.501008	0.499668	0.499707	0.500195	0.499555	0.498057	0.501093	0.500373	0.500451	0.498533	0.50004	0.500582	0.500631	0.500199	0.501114	0.498788	0.500888	0.500622	0.500672	0.502014	0.500119	0.500631	0.500836	0.50023	0.500298	0.4993	0.499376	0.500356	0.49749	0.500857	0.497466	0.500398
		x4	0.498927	0.499177	0.499439	0.501088	0.499878	0.500148	0.500576	0.499589	0.497572	0.497689	0.500138	0.497441	0.500312	0.499746	0.499244	0.500889	0.499979	0.497592	0.503857	0.499653	0.498614	0.500186	0.500054	0.500544	0.50013	0.500252	0.499316	0.501047	0.500366	0.500037	0.498187	0.499114	0.49998	0.499643	0.499925	0.499431	0.498496	0.495573	0.499426	0.49991	0.499366	0.499885	0.500906	0.499585	0.501457	0.499587	0.500188	0.498207	0.500573	0.492968	0.493443	0.499509	0.498977	0.499384	0.49969	0.500432	0.500548	0.500666	0.499962	0.494711	0.499647	0.500718	0.495334	0.50191
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**63+2**127	5	x0	0.499401	0.499709	0.499989	0.500185	0.499426	0.499571	0.499041	0.500213	0.500361	0.500863	0.500141	0.499099	0.500376	0.499598	0.499405	0.500759	0.500539	0.500832	0.500037	0.498811	0.49915	0.499747	0.500049	0.500688	0.500435	0.499718	0.499391	0.500043	0.500538	0.500335	0.500354	0.500487	0.500239	0.49943	0.500584	0.49965	0.499774	0.50032	0.499677	0.499977	0.499879	0.499609	0.499854	0.500162	0.499646	0.499717	0.501239	0.500016	0.500472	0.50064	0.499381	0.50092	0.50103	0.50137	0.501273	0.500125	0.499851	0.499493	0.499357	0.500133	0.499757	0.500407	0.499204	0.500567
		x1	0.499887	0.499977	0.49968	0.500336	0.499764	0.500887	0.500672	0.500295	0.500106	0.500018	0.500214	0.5006	0.50109	0.501068	0.500353	0.499836	0.500415	0.499811	0.500753	0.49955	0.499038	0.499639	0.500406	0.500332	0.500357	0.500622	0.499326	0.499595	0.498796	0.500584	0.499637	0.499667	0.500089	0.500695	0.500067	0.49989	0.499839	0.500437	0.499694	0.500186	0.499828	0.500951	0.499967	0.499746	0.500935	0.499943	0.499657	0.500529	0.499838	0.499851	0.499838	0.500845	0.499673	0.500201	0.500232	0.499622	0.500434	0.500617	0.500422	0.500339	0.500301	0.500306	0.499794	0.499991
		x2	0.499568	0.500216	0.499443	0.500082	0.500187	0.500436	0.499727	0.500433	0.499665	0.499525	0.500044	0.499971	0.500182	0.500093	0.500215	0.500027	0.500016	0.499926	0.499524	0.499833	0.499103	0.500039	0.500183	0.500027	0.499206	0.4995	0.499614	0.499855	0.500623	0.500061	0.49912	0.50003	0.499665	0.499791	0.500444	0.499172	0.499291	0.499787	0.499198	0.500207	0.500407	0.500063	0.49881	0.499162	0.500899	0.499896	0.499656	0.499436	0.499719	0.499806	0.500559	0.50079	0.499702	0.500763	0.500255	0.499355	0.500112	0.500246	0.50086	0.500231	0.499995	0.500303	0.500479	0.500884
		x3	0.499701	0.499573	0.499909	0.500111	0.499868	0.500349	0.499838	0.498891	0.499571	0.500692	0.499782	0.500299	0.499337	0.499576	0.500185	0.500142	0.499608	0.500741	0.499354	0.499633	0.499498	0.500496	0.499562	0.499628	0.500726	0.499838	0.500036	0.499662	0.500542	0.500272	0.50027	0.500561	0.499543	0.500375	0.499783	0.499442	0.500123	0.499994	0.50014	0.500983	0.500411	0.500448	0.49949	0.499708	0.499638	0.501112	0.50101	0.5003	0.499992	0.50016	0.50005	0.498971	0.499092	0.499943	0.499645	0.500504	0.499441	0.500103	0.499855	0.499657	0.500441	0.500157	0.499326	0.500243
		x4	0.499668	0.499546	0.500262	0.500285	0.499714	0.500592	0.500053	0.500887	0.499398	0.500247	0.499758	0.499943	0.499859	0.500611	0.499337	0.499628	0.499791	0.499821	0.500267	0.499807	0.501716	0.500201	0.5006	0.499484	0.500518	0.499231	0.499443	0.499072	0.500221	0.499786	0.500747	0.500121	0.50069	0.499415	0.499718	0.499474	0.498801	0.500302	0.50002	0.499413	0.500426	0.499577	0.499869	0.500726	0.500411	0.499474	0.500694	0.499886	0.500209	0.499452	0.499933	0.500408	0.50062	0.500431	0.500131	0.500075	0.499513	0.499799	0.500117	0.499358	0.500218	0.500017	0.500103	0.500051
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
"2**63"	3	x0	0.500231	0.500467	0.499204	0.509785	0.500148	0.469252	0.507638	0.546869	0.312885	0.453702	0.499844	0.498866	0.406339	0.507408	0.593961	0.468316	0.500198	0.499807	0.500166	0.499275	0.511928	0.375104	0.503646	0.500038	0.483608	0.49674	0.499237	0.406289	0.500131	0.500059	0.406552	0.468718	0.437658	0.546753	0.487904	0.49987	0.50027	0.500221	0.500801	0.500014	0.500647	0.437439	0.500232	0.437743	0.500306	0.499448	0.500294	0.468735	0.476959	0.686985	0.250568	0.499941	0.437736	0.475935	0.406352	0.469157	0.546846	0.499713	0.491854	0.500361	0.468147	0.499675	0.499636	0.375334
		x1	0.500207	0.500093	0.500507	0.499717	0.499603	0.498812	0.453837	0.500357	0.429485	0.406132	0.500261	0.500442	0.375157	0.49941	0.500749	0.500292	0.500068	0.481812	0.453798	0.499703	0.500736	0.499803	0.499658	0.499897	0.49939	0.500685	0.499841	0.312944	0.499429	0.499447	0.250804	0.500343	0.499672	0.500289	0.406571	0.500307	0.500257	0.500985	0.500574	0.499127	0.500028	0.49958	0.499505	0.499769	0.499816	0.500294	0.499851	0.469008	0.500109	0.500751	0.406815	0.500278	0.312762	0.499254	0.250189	0.49954	0.500164	0.547169	0.50028	0.498963	0.499984	0.499126	0.500247	0.50056
		x2	0.500255	0.507731	0.515539	0.501345	0.561722	0.530869	0.50378	0.500535	0.500862	0.43733	0.500059	0.477143	0.530801	0.501413	0.499424	0.499594	0.515355	0.515565	0.476253	0.506968	0.499944	0.312635	0.531612	0.468857	0.461315	0.453338	0.453352	0.359785	0.429105	0.472558	0.43783	0.500018	0.312369	0.312301	0.429991	0.383118	0.500114	0.499049	0.530913	0.499738	0.500187	0.515621	0.500613	0.375267	0.530962	0.499762	0.515938	0.453369	0.31316	0.500703	0.500419	0.531174	0.484728	0.375081	0.312396	0.500712	0.499407	0.499242	0.499456	0.499499	0.313129	0.500104	0.499758	0.625002
		x3	0.499918	0.530452	0.499778	0.499866	0.49994	0.375294	0.516189	0.499239	0.562432	0.498924	0.508096	0.499505	0.250021	0.484921	0.500192	0.562156	0.492512	0.500007	0.530173	0.499937	0.499858	0.500404	0.500075	0.530622	0.499234	0.561733	0.516389	0.452906	0.500381	0.499892	0.429417	0.49938	0.250413	0.514377	0.465904	0.499419	0.50064	0.382899	0.523032	0.499934	0.561514	0.468554	0.562471	0.375501	0.382435	0.504291	0.500049	0.437163	0.484152	0.501057	0.625174	0.476119	0.437883	0.500787	0.25039	0.487718	0.49961	0.531046	0.499632	0.500236	0.374752	0.499999	0.452591	0.500467
		x4	0.508188	0.494395	0.43781	0.481003	0.507737	0.47622	0.476553	0.497978	0.406051	0.250261	0.500153	0.492319	0.406731	0.489175	0.476439	0.499794	0.49962	0.453498	0.438111	0.438182	0.406232	0.593767	0.452145	0.500166	0.499794	0.500802	0.522224	0.406366	0.50026	0.437248	0.500623	0.437079	0.437681	0.569395	0.406552	0.499968	0.452894	0.374439	0.499666	0.500112	0.313418	0.499507	0.452373	0.468565	0.437912	0.488302	0.508607	0.417571	0.54706	0.68694	0.250326	0.499971	0.484631	0.437617	0.312777	0.578587	0.382574	0.499843	0.500075	0.49966	0.468813	0.499401	0.31266	0.64017
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
"2**63"	4	x0	0.500147	0.50102	0.500443	0.499392	0.49954	0.499924	0.499307	0.498589	0.499956	0.499853	0.500786	0.500327	0.500455	0.499414	0.500214	0.501182	0.499768	0.500887	0.49797	0.500165	0.500403	0.498826	0.49958	0.500122	0.499797	0.499861	0.500106	0.498839	0.500161	0.500025	0.498959	0.499797	0.49957	0.499915	0.499991	0.50058	0.499547	0.500176	0.500253	0.500693	0.499644	0.500222	0.5007	0.499922	0.500042	0.500212	0.500755	0.499279	0.499843	0.499664	0.500404	0.501057	0.500598	0.499758	0.500586	0.49943	0.499308	0.500223	0.500396	0.499374	0.500707	0.500272	0.499556	0.499484
		x1	0.499549	0.499479	0.499315	0.500161	0.500146	0.498932	0.499395	0.499206	0.499681	0.50004	0.499759	0.49957	0.500505	0.500319	0.500553	0.500479	0.500177	0.499131	0.500586	0.500268	0.499923	0.500186	0.500477	0.499568	0.499964	0.50022	0.500445	0.499517	0.499279	0.500028	0.500185	0.499719	0.500139	0.499674	0.500303	0.499906	0.499889	0.500923	0.500213	0.499982	0.498833	0.499966	0.499983	0.50061	0.499595	0.500445	0.499474	0.500293	0.499609	0.500107	0.50061	0.500653	0.499894	0.500088	0.499407	0.499688	0.500552	0.500085	0.499157	0.500356	0.4994	0.500194	0.499011	0.499313
		x2	0.500178	0.499655	0.499492	0.50008	0.500359	0.499795	0.499187	0.499593	0.499676	0.499549	0.500531	0.500761	0.499219	0.500276	0.50056	0.499253	0.500768	0.499884	0.500664	0.499656	0.499784	0.500073	0.50017	0.500078	0.500228	0.500084	0.500202	0.499685	0.499603	0.500566	0.49926	0.499348	0.500589	0.500429	0.501125	0.4999	0.500008	0.499621	0.500847	0.499766	0.499858	0.500784	0.500254	0.499599	0.50026	0.500067	0.500259	0.500029	0.500524	0.499723	0.500063	0.500402	0.500021	0.499748	0.499582	0.49973	0.499891	0.499651	0.499992	0.500046	0.499846	0.500662	0.499217	0.500943
		x3	0.499397	0.500885	0.499527	0.500824	0.499668	0.499745	0.499845	0.500354	0.499583	0.500443	0.49947	0.499898	0.500773	0.500713	0.500088	0.49969	0.499524	0.499436	0.499674	0.49958	0.49959	0.500461	0.501407	0.499481	0.499732	0.500319	0.499849	0.49938	0.499904	0.499839	0.49978	0.500398	0.500655	0.499897	0.499975	0.499897	0.500346	0.499293	0.50051	0.499574	0.499737	0.499792	0.500077	0.500014	0.500884	0.499802	0.499897	0.500003	0.500338	0.50075	0.500811	0.500755	0.500043	0.499695	0.500275	0.500844	0.50016	0.499995	0.499985	0.499761	0.500826	0.500369	0.500034	0.499426
		x4	0.500389	0.500216	0.500894	0.500591	0.500082	0.50082	0.49918	0.499631	0.499661	0.500177	0.500374	0.500662	0.499895	0.49981	0.500508	0.500601	0.499805	0.499742	0.500134	0.499798	0.500511	0.499371	0.499377	0.500109	0.500591	0.499364	0.500015	0.501255	0.500801	0.501099	0.500203	0.500115	0.499902	0.499009	0.50022	0.500038	0.500137	0.49745	0.499442	0.499972	0.500234	0.500478	0.500385	0.500296	0.500206	0.500559	0.499886	0.499442	0.500482	0.501196	0.499371	0.499618	0.500427	0.499396	0.500657	0.501255	0.499591	0.499728	0.5001	0.499922	0.49876	0.501288	0.500085	0.500206
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
"2**63"	5	x0	0.499855	0.50039	0.499657	0.499398	0.500514	0.50069	0.499976	0.500848	0.499933	0.499518	0.501749	0.500916	0.499415	0.499976	0.500365	0.50008	0.499684	0.499262	0.501137	0.500558	0.500046	0.499551	0.499645	0.500965	0.499625	0.500287	0.499641	0.499201	0.498381	0.500047	0.499436	0.500954	0.500059	0.499311	0.499737	0.501081	0.500264	0.49993	0.499729	0.499963	0.499745	0.500602	0.501023	0.49895	0.500316	0.499735	0.500271	0.500409	0.499826	0.500145	0.500071	0.499733	0.500547	0.500147	0.500494	0.500836	0.499906	0.499895	0.500107	0.500553	0.500171	0.500714	0.498945	0.500671
		x1	0.500149	0.500507	0.500094	0.500119	0.500486	0.500247	0.50009	0.499791	0.499836	0.50022	0.499555	0.499943	0.499929	0.499602	0.500067	0.500134	0.499443	0.500526	0.500257	0.50037	0.500304	0.500099	0.499708	0.500658	0.500112	0.500193	0.499532	0.499905	0.50025	0.499858	0.499895	0.499942	0.498528	0.498979	0.499563	0.500117	0.499959	0.500901	0.500408	0.500044	0.499027	0.49956	0.499486	0.4999	0.499776	0.499303	0.49963	0.499535	0.500567	0.500222	0.499932	0.498716	0.500087	0.498977	0.5	0.500291	0.500921	0.499891	0.499738	0.498897	0.499605	0.499968	0.500208	0.499688
		x2	0.499899	0.499036	0.501067	0.49996	0.500305	0.500597	0.500559	0.499901	0.500167	0.500592	0.500575	0.500132	0.500362	0.499459	0.499484	0.500194	0.50061	0.499931	0.499712	0.499731	0.498871	0.500141	0.500599	0.500349	0.500533	0.499807	0.499932	0.500486	0.499922	0.50035	0.500525	0.499873	0.500017	0.499843	0.50003	0.499445	0.499916	0.500049	0.500679	0.499475	0.500185	0.499912	0.500659	0.500434	0.500225	0.499454	0.499915	0.499696	0.502034	0.500618	0.499545	0.50118	0.500517	0.500492	0.498921	0.499954	0.500351	0.500292	0.499662	0.500193	0.499262	0.500703	0.500925	0.499381
		x3	0.499388	0.500065	0.500742	0.499722	0.499929	0.500209	0.500652	0.500102	0.499999	0.50089	0.499059	0.500333	0.500335	0.500398	0.499529	0.500148	0.500395	0.501068	0.500068	0.50037	0.500053	0.500767	0.500504	0.500721	0.500454	0.499963	0.499128	0.500617	0.5001	0.499977	0.499996	0.501073	0.500756	0.500272	0.500092	0.500176	0.499757	0.500455	0.499541	0.500597	0.500039	0.49994	0.499956	0.500206	0.499658	0.500229	0.499762	0.49978	0.499495	0.499227	0.499815	0.500922	0.499403	0.499853	0.499298	0.499165	0.500057	0.50058	0.500031	0.500358	0.500355	0.50022	0.499334	0.500228
		x4	0.499146	0.500387	0.501318	0.499429	0.499386	0.500088	0.500124	0.500778	0.500457	0.500825	0.49939	0.499945	0.499625	0.500703	0.499767	0.499954	0.499727	0.499633	0.500026	0.499539	0.49968	0.499744	0.499942	0.499127	0.500106	0.499869	0.499848	0.499399	0.499231	0.499798	0.49982	0.500143	0.500354	0.499514	0.499626	0.49947	0.49998	0.500175	0.500165	0.500513	0.499849	0.499123	0.499828	0.499753	0.499942	0.500372	0.500402	0.498531	0.500225	0.499057	0.499518	0.499945	0.501059	0.500553	0.50029	0.499904	0.49977	0.499702	0.500364	0.499951	0.500203	0.499684	0.501218	0.499886
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**319	3	x0	0.501017	0.483993	0.499605	0.507341	0.50073	0.468869	0.499249	0.545932	0.31279	0.437586	0.499703	0.559427	0.312884	0.508751	0.593907	0.468726	0.499594	0.499306	0.374365	0.501243	0.531087	0.249401	0.500356	0.453197	0.485303	0.492162	0.500397	0.405684	0.501025	0.499727	0.311773	0.437576	0.438396	0.546934	0.468814	0.5003	0.500047	0.374645	0.499386	0.499112	0.500079	0.437499	0.558256	0.437927	0.500521	0.500264	0.468864	0.46805	0.469352	0.686961	0.249988	0.437241	0.438439	0.468531	0.40615	0.468548	0.546368	0.499036	0.492279	0.499785	0.468917	0.499479	0.499936	0.374557
		x1	0.499092	0.50028	0.500472	0.500489	0.499659	0.49975	0.406128	0.499948	0.359536	0.313353	0.499645	0.500135	0.250211	0.499799	0.500044	0.499786	0.49974	0.465086	0.406383	0.499877	0.500702	0.499641	0.500082	0.500382	0.500246	0.501034	0.500518	0.312868	0.500086	0.500425	0.250304	0.499799	0.499192	0.499819	0.312721	0.500617	0.50087	0.499363	0.500368	0.499558	0.49954	0.500698	0.500312	0.500027	0.500914	0.499875	0.499275	0.468776	0.500326	0.499617	0.312978	0.500583	0.3133	0.4999	0.249952	0.50033	0.500082	0.594845	0.499794	0.500617	0.500413	0.500536	0.499593	0.499977
		x2	0.49959	0.516577	0.530741	0.499869	0.624215	0.530938	0.507343	0.500448	0.515752	0.374822	0.499342	0.453025	0.562613	0.515453	0.500681	0.500544	0.531199	0.531869	0.452771	0.514595	0.500144	0.312917	0.562428	0.436847	0.421985	0.453251	0.452409	0.36027	0.429159	0.444356	0.437776	0.499031	0.312103	0.313027	0.42959	0.38249	0.499619	0.499988	0.532335	0.499627	0.500556	0.531559	0.499671	0.375025	0.531268	0.500328	0.53163	0.453581	0.313648	0.499657	0.499718	0.561923	0.485303	0.375468	0.312884	0.500402	0.500625	0.500291	0.499703	0.50006	0.312913	0.500148	0.49939	0.749536
		x3	0.500097	0.561953	0.499588	0.499938	0.49939	0.37491	0.531703	0.500222	0.624615	0.500806	0.507435	0.499779	0.250584	0.468772	0.499904	0.625129	0.483805	0.500728	0.562013	0.468092	0.499202	0.499549	0.374938	0.562299	0.500318	0.56249	0.51589	0.453204	0.500194	0.476941	0.407109	0.500149	0.250158	0.531038	0.45291	0.507327	0.500314	0.342856	0.523543	0.500182	0.62464	0.469114	0.562756	0.374245	0.343731	0.508529	0.499636	0.438733	0.484275	0.500262	0.749939	0.470235	0.437536	0.500528	0.250651	0.477214	0.500489	0.561407	0.501165	0.499453	0.375549	0.499553	0.406296	0.499996
		x4	0.515159	0.485171	0.4384	0.483805	0.508992	0.468881	0.437101	0.496299	0.250397	0.249923	0.500101	0.469245	0.407021	0.468843	0.477407	0.499845	0.500796	0.437588	0.437727	0.437544	0.374551	0.593957	0.437875	0.500739	0.499853	0.484575	0.523485	0.375525	0.499734	0.437197	0.250282	0.437092	0.437337	0.570354	0.375451	0.500591	0.437209	0.374912	0.500071	0.498821	0.250623	0.499354	0.375286	0.468841	0.437068	0.4855	0.506688	0.312926	0.546811	0.687686	0.25064	0.501344	0.437597	0.43768	0.312382	0.577034	0.38345	0.499495	0.499688	0.312944	0.468408	0.499855	0.250172	0.64008
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**319	4	x0	0.499903	0.499718	0.500323	0.500512	0.500048	0.499177	0.498968	0.500581	0.500572	0.50062	0.501679	0.500484	0.499758	0.500759	0.500163	0.499442	0.49996	0.500801	0.495625	0.499307	0.499603	0.499714	0.499916	0.49936	0.500239	0.499744	0.500453	0.497536	0.500413	0.500158	0.499833	0.500169	0.500379	0.500712	0.499896	0.499419	0.500615	0.500356	0.500149	0.49913	0.500074	0.500241	0.499962	0.499851	0.499331	0.499277	0.500374	0.498953	0.499464	0.498755	0.500838	0.5001	0.499969	0.500833	0.500227	0.500341	0.499852	0.500553	0.500386	0.499673	0.500341	0.499844	0.498775	0.499959
		x1	0.500304	0.499289	0.500113	0.49887	0.50063	0.499628	0.500082	0.499993	0.499661	0.500286	0.499793	0.500473	0.500452	0.500166	0.499637	0.500135	0.500207	0.50039	0.500504	0.500163	0.498689	0.499824	0.500094	0.499432	0.499746	0.500132	0.500186	0.498119	0.499735	0.499233	0.499914	0.50071	0.500113	0.500001	0.500318	0.499341	0.500043	0.499478	0.499478	0.500548	0.500834	0.500094	0.499588	0.500497	0.499998	0.500216	0.499312	0.500245	0.50044	0.500349	0.500126	0.499528	0.500935	0.499691	0.499659	0.500912	0.500188	0.501199	0.499429	0.49933	0.499712	0.500444	0.499792	0.500639
		x2	0.499429	0.499589	0.499791	0.500908	0.500826	0.500276	0.500507	0.499833	0.49996	0.500209	0.50009	0.50066	0.500172	0.499768	0.500434	0.500446	0.499592	0.500373	0.499252	0.501075	0.499277	0.500276	0.500497	0.500431	0.50008	0.499883	0.500296	0.50073	0.499447	0.500178	0.499504	0.50003	0.499182	0.500164	0.500227	0.500275	0.500362	0.499146	0.50064	0.500345	0.499198	0.500111	0.499437	0.499959	0.500058	0.499573	0.499946	0.500352	0.499942	0.500109	0.499945	0.500678	0.49926	0.500746	0.4995	0.499533	0.499536	0.499998	0.500937	0.49956	0.50073	0.500048	0.500295	0.500022
		x3	0.500279	0.499659	0.499989	0.499925	0.498993	0.500887	0.500702	0.499853	0.500133	0.499513	0.500599	0.500248	0.500141	0.501846	0.500214	0.500059	0.499087	0.49949	0.500427	0.500384	0.499372	0.500297	0.500186	0.500245	0.500067	0.499824	0.501081	0.500438	0.500045	0.500075	0.50021	0.5004	0.500585	0.500024	0.500519	0.500184	0.499755	0.500379	0.499321	0.499788	0.500483	0.500036	0.499251	0.499615	0.501155	0.499583	0.500198	0.500409	0.501085	0.499542	0.499374	0.501398	0.500706	0.499492	0.500603	0.499642	0.50021	0.499334	0.500334	0.499434	0.5001	0.50033	0.500114	0.499715
		x4	0.499875	0.499585	0.499584	0.500565	0.499451	0.500447	0.499925	0.499675	0.499724	0.499581	0.499664	0.499967	0.50017	0.500373	0.500256	0.49891	0.499459	0.500336	0.499828	0.500657	0.499148	0.500433	0.49929	0.499357	0.499579	0.499195	0.500456	0.502637	0.500317	0.501177	0.500288	0.49885	0.501013	0.498917	0.499544	0.500054	0.50006	0.494885	0.499968	0.500558	0.500508	0.500279	0.500561	0.499624	0.499366	0.499693	0.499971	0.499982	0.500147	0.499437	0.500315	0.499506	0.50024	0.499171	0.501658	0.500004	0.501188	0.499885	0.499668	0.500431	0.500115	0.500363	0.500008	0.499911
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**319	5	x0	0.499639	0.500542	0.49927	0.50067	0.499555	0.499595	0.499713	0.500091	0.499559	0.499771	0.500406	0.499885	0.499953	0.500636	0.500155	0.500132	0.49976	0.499801	0.500571	0.499536	0.500022	0.499619	0.499626	0.50038	0.499733	0.500533	0.50102	0.50038	0.500071	0.49939	0.500563	0.501237	0.500383	0.500304	0.499131	0.499315	0.499356	0.500553	0.49969	0.500608	0.500088	0.500322	0.50011	0.498116	0.499813	0.500352	0.500311	0.500167	0.500381	0.500903	0.499753	0.500081	0.500358	0.500454	0.500288	0.499519	0.500752	0.498954	0.500573	0.500189	0.499634	0.500556	0.500629	0.499701
		x1	0.498802	0.499785	0.500503	0.499747	0.499673	0.499468	0.500084	0.500224	0.49955	0.499984	0.500634	0.500552	0.500449	0.499942	0.500091	0.499627	0.500471	0.499289	0.500213	0.499503	0.500583	0.501123	0.501062	0.498409	0.500179	0.500496	0.49954	0.500551	0.499944	0.500059	0.500731	0.500022	0.499668	0.499282	0.50068	0.500635	0.499548	0.499082	0.499286	0.499218	0.499868	0.500057	0.500099	0.500548	0.500081	0.5003	0.499872	0.500979	0.501188	0.500034	0.499612	0.49948	0.500166	0.500273	0.500268	0.499406	0.500073	0.499865	0.499225	0.500325	0.500273	0.499953	0.499863	0.50025
		x2	0.50037	0.500346	0.500414	0.500165	0.50003	0.499179	0.499436	0.500585	0.49946	0.500115	0.500359	0.5009	0.500181	0.499896	0.500202	0.500829	0.499986	0.499597	0.500207	0.49961	0.49947	0.500209	0.499627	0.50087	0.499572	0.500572	0.499854	0.500543	0.499798	0.501506	0.500503	0.500247	0.499918	0.500621	0.499532	0.500267	0.500074	0.500118	0.499972	0.499765	0.499755	0.499744	0.500117	0.499789	0.499325	0.499936	0.50035	0.498942	0.499796	0.500315	0.498924	0.499983	0.50014	0.498981	0.499783	0.500753	0.499689	0.500294	0.500203	0.500833	0.499862	0.500202	0.499028	0.499389
		x3	0.499637	0.499363	0.500385	0.499149	0.500944	0.500443	0.499718	0.500264	0.500157	0.500879	0.499227	0.500438	0.500255	0.499912	0.499689	0.500566	0.500226	0.499921	0.499757	0.49977	0.499814	0.500123	0.499507	0.49958	0.500235	0.499974	0.499517	0.49942	0.500291	0.499808	0.500546	0.499233	0.500007	0.500338	0.49992	0.501162	0.499638	0.499584	0.500309	0.498999	0.500241	0.499755	0.49936	0.500021	0.500386	0.501103	0.499485	0.500162	0.500088	0.500026	0.500215	0.500766	0.499613	0.50052	0.500925	0.500622	0.500172	0.499006	0.500759	0.50044	0.499845	0.501058	0.499805	0.500417
		x4	0.500133	0.499943	0.50025	0.500638	0.499913	0.498985	0.500378	0.499209	0.500064	0.499795	0.500302	0.500402	0.500485	0.499923	0.499563	0.499882	0.499627	0.500321	0.500623	0.499389	0.50014	0.499854	0.500308	0.500106	0.500739	0.501041	0.500625	0.500273	0.499281	0.50035	0.5008	0.499234	0.49916	0.499302	0.499994	0.50009	0.499256	0.50036	0.500798	0.500033	0.50016	0.499929	0.499775	0.499542	0.499625	0.499567	0.499786	0.499618	0.499628	0.499928	0.500023	0.5007	0.50025	0.500189	0.50041	0.499515	0.499929	0.499896	0.499543	0.499663	0.499493	0.499414	0.500729	0.500565
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
2**63+2**127	3.5	x0	0.49971	0.502335	0.496503	0.500707	0.496077	0.480319	0.497966	0.504934	0.491307	0.463589	0.501395	0.511543	0.464423	0.499171	0.504908	0.49556	0.499043	0.500585	0.500464	0.491654	0.495726	0.481602	0.494953	0.498537	0.49789	0.500168	0.498021	0.441939	0.501745	0.49121	0.468512	0.475791	0.46941	0.513159	0.463314	0.500399	0.497375	0.456857	0.500485	0.497796	0.515381	0.499206	0.504133	0.464305	0.482394	0.500592	0.497409	0.481168	0.512025	0.541411	0.467982	0.499705	0.476872	0.492119	0.375706	0.507468	0.488561	0.49927	0.498287	0.511673	0.473133	0.493438	0.488394	0.502453
		x1	0.501644	0.500255	0.500306	0.50022	0.498245	0.4996	0.499446	0.498899	0.476345	0.480837	0.49562	0.499382	0.476537	0.500043	0.49831	0.499358	0.499108	0.503679	0.49665	0.500139	0.494224	0.475154	0.499942	0.4999	0.499802	0.499991	0.50045	0.418405	0.499223	0.500056	0.429812	0.500565	0.49947	0.501285	0.476641	0.500362	0.500434	0.494552	0.499342	0.501574	0.499937	0.500877	0.498815	0.500036	0.500676	0.500091	0.501833	0.493567	0.501027	0.446175	0.405828	0.499663	0.499133	0.499602	0.421727	0.497967	0.496022	0.500054	0.500111	0.495559	0.499209	0.495516	0.496198	0.502782
		x2	0.50003	0.500472	0.501551	0.49972	0.500428	0.486356	0.503824	0.500761	0.489303	0.50828	0.500119	0.500058	0.499854	0.49945	0.495009	0.490095	0.500548	0.51502	0.484287	0.499549	0.488944	0.45325	0.510182	0.500732	0.486396	0.499496	0.504497	0.389906	0.504387	0.455302	0.452772	0.488498	0.499489	0.494384	0.465273	0.500849	0.492617	0.452151	0.500815	0.501726	0.547027	0.499786	0.503923	0.476613	0.499392	0.504477	0.500554	0.483878	0.458251	0.487613	0.437168	0.499391	0.472134	0.499851	0.344551	0.5008	0.477138	0.503635	0.502094	0.526003	0.441112	0.498051	0.539482	0.481988
		x3	0.498699	0.500142	0.499056	0.499969	0.500698	0.483461	0.502477	0.495695	0.488673	0.503413	0.500965	0.500127	0.499888	0.500094	0.491768	0.487985	0.50043	0.514516	0.476377	0.499689	0.49135	0.460501	0.50599	0.500527	0.488739	0.499971	0.506298	0.401217	0.507309	0.47766	0.440937	0.484745	0.499743	0.491711	0.480021	0.500911	0.494874	0.471077	0.500054	0.499199	0.547594	0.499066	0.502189	0.468407	0.499626	0.503431	0.500074	0.49424	0.454873	0.488874	0.436499	0.499745	0.459478	0.499931	0.390493	0.500896	0.486281	0.504095	0.503298	0.523774	0.470168	0.497875	0.539335	0.502705
		x4	0.49931	0.494209	0.496594	0.494667	0.515061	0.476572	0.487919	0.504214	0.500271	0.44576	0.502145	0.514853	0.418038	0.501981	0.506419	0.497345	0.499594	0.500012	0.499471	0.491297	0.496351	0.481013	0.495048	0.496155	0.496039	0.499995	0.505182	0.426091	0.502764	0.490405	0.452629	0.477018	0.45655	0.513271	0.456635	0.500709	0.497723	0.44567	0.500257	0.502723	0.460518	0.47628	0.504185	0.465272	0.482517	0.500813	0.496489	0.474028	0.512671	0.547388	0.499435	0.488038	0.467973	0.4804	0.359221	0.505942	0.488572	0.500508	0.499428	0.471477	0.472378	0.493944	0.448713	0.50134
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
"2**63"	3.5	x0	0.500591	0.502592	0.500792	0.504679	0.500183	0.483818	0.498178	0.499627	0.527313	0.50064	0.501393	0.49957	0.479628	0.499807	0.499593	0.500438	0.500739	0.499599	0.515899	0.500132	0.4997	0.501201	0.497071	0.502671	0.499444	0.50019	0.498967	0.47102	0.500991	0.500511	0.480647	0.499899	0.477391	0.498245	0.480972	0.50087	0.499556	0.484744	0.499485	0.499738	0.51532	0.502987	0.503461	0.488993	0.493066	0.499643	0.499512	0.48039	0.508243	0.499635	0.55459	0.503817	0.492537	0.499967	0.374823	0.500434	0.499493	0.499407	0.499822	0.499834	0.468729	0.500513	0.487579	0.494195
		x1	0.499971	0.500403	0.500274	0.500458	0.500619	0.499441	0.500239	0.499656	0.499836	0.488466	0.498921	0.500765	0.484366	0.500181	0.50513	0.499071	0.499453	0.50043	0.504179	0.500835	0.507449	0.512226	0.499962	0.499789	0.500292	0.500247	0.500493	0.458452	0.500398	0.499803	0.455362	0.499896	0.500089	0.499952	0.490159	0.500303	0.500231	0.500477	0.499497	0.500474	0.500918	0.500684	0.503574	0.500258	0.499845	0.500662	0.50072	0.494753	0.500053	0.4534	0.461547	0.499956	0.499985	0.50026	0.421255	0.500468	0.509634	0.499937	0.500579	0.50056	0.500293	0.499689	0.500423	0.513142
		x2	0.500131	0.499552	0.501063	0.499851	0.499526	0.491291	0.500649	0.501121	0.499697	0.476744	0.499705	0.50064	0.524086	0.500085	0.500872	0.499708	0.499387	0.50376	0.498811	0.500029	0.499772	0.500171	0.499863	0.500421	0.500468	0.499885	0.503632	0.433264	0.497971	0.477885	0.477181	0.500341	0.500387	0.499772	0.478882	0.49937	0.50011	0.499423	0.499736	0.500456	0.500946	0.504934	0.499341	0.499988	0.499808	0.499887	0.499452	0.488786	0.507468	0.500016	0.500207	0.50924	0.489951	0.499601	0.343492	0.500085	0.50007	0.4994	0.499596	0.500272	0.437527	0.499274	0.500108	0.495665
		x3	0.500179	0.499871	0.499821	0.500332	0.499924	0.489674	0.499983	0.499154	0.500556	0.488475	0.498717	0.500015	0.523325	0.500239	0.500049	0.499195	0.50034	0.503376	0.49307	0.50088	0.500446	0.499984	0.500284	0.500112	0.500184	0.500939	0.5039	0.431102	0.499914	0.488993	0.470842	0.500524	0.500832	0.500726	0.48574	0.500061	0.49983	0.499712	0.500309	0.500816	0.500615	0.508444	0.499697	0.499956	0.498985	0.499558	0.4999	0.497052	0.49818	0.500107	0.499167	0.515795	0.486376	0.500597	0.390769	0.499843	0.500137	0.500991	0.499531	0.499604	0.468156	0.499253	0.499822	0.501773
		x4	0.500432	0.499829	0.499667	0.504822	0.500571	0.475526	0.498662	0.499892	0.569835	0.500628	0.501093	0.500089	0.418747	0.501063	0.500432	0.502209	0.49992	0.499213	0.523888	0.499152	0.499542	0.500322	0.497219	0.503677	0.499297	0.500658	0.497524	0.46314	0.500775	0.499313	0.469925	0.500076	0.46932	0.498119	0.480906	0.499159	0.499658	0.479276	0.499534	0.501161	0.51475	0.48876	0.503188	0.487583	0.492195	0.498853	0.500873	0.470965	0.508786	0.499537	0.610359	0.493719	0.485286	0.500879	0.359721	0.49869	0.500123	0.500881	0.499996	0.500237	0.468779	0.503251	0.487672	0.494438

Figure 12: Bitwise Bias Values for XORed Differences

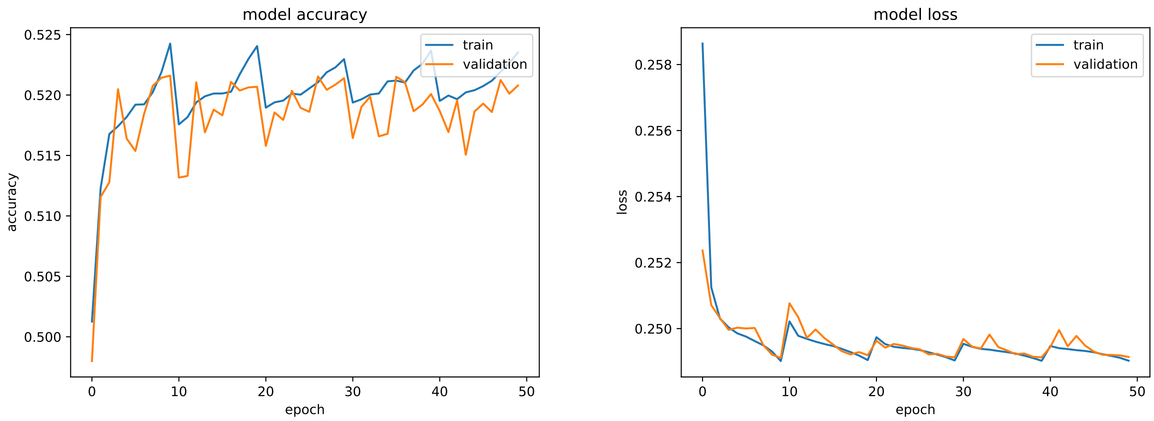

Figure 13: Accuracy and Loss plots of Biased Inputs (Depth 1)

it generalise to another setting (Goodfellow et al., 2016). To do so, we trained the network on data passed through 3.5 rounds of the round function. That is, 3 rounds of the round function followed by only the constant addition and substitution layers. This was done to have the data as "close" to the 4 round case as possible while still attaining good accuracy, so that a higher degree of transference could be attained.

This training for 3.5 rounds was conducted with the same architecture and specifications as per the results in Table 6 and was run for 20 epochs. The highest validation accuracy for 3.5 rounds attained was 0.999279976.

5.4.2 Basic Modifications

From here, the following four modification of the architecture to the already trained 3.5 round network were experimented with - replacing prediction head while freezing weights of everything else, replacing classification block with a new one while freezing weights of everything else, removing classification block and adding new residual and classification blocks while freezing weights of everything else, and removing prediction head and adding an additional duplicate classification block to the end while freezing weights of everything else.

Each of these were then run for 100 epochs on the 4 round case with specifications as per the results in Table 6. However, it was found that there was no improvement to the validation accuracy.

5.4.3 Inverse XOR Modification

The previous results suggest that these models are insufficient for generalisation to the 4 round case. Furthermore, coupled with the evidence that the accuracy is extremely close to 1 for the 3.5 round case, it suggests that the fundamental

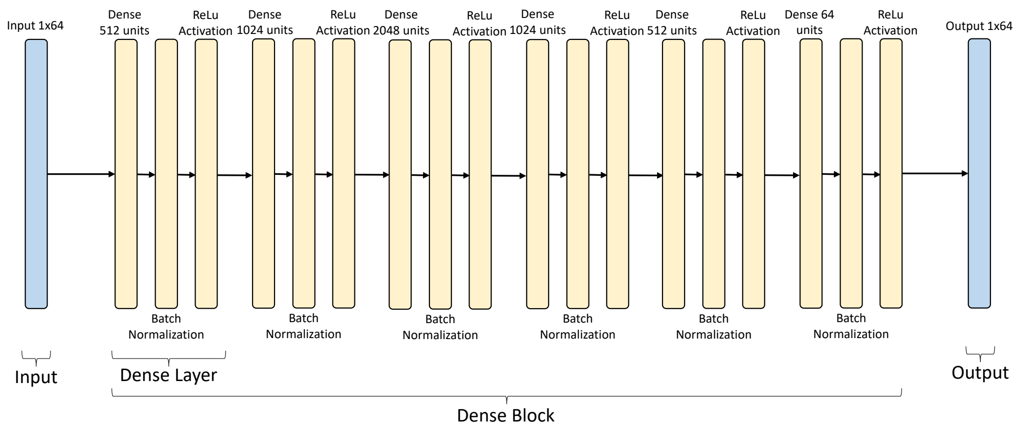

Figure 14: Inverse XOR Architecture

issue at had would be resolving the inverse XOR problem as per the linear diffusion layer.

To overcome the inverse XOR problem, we proceeded with the following architecture in Figure 14 to first be trained. Here, a dataset of size 10^6 was generated and applied in at 90/10 training-validation split. Each data entry x, of shape 1x320, was passed through only the linear diffusion layer. From here, the first 64-bits of outputs y were fed as inputs to the network, with the first 64 bits of x values to be predicted. That is, x_0 and its corresponding output were used. This was chosen over applying it to the 320 bits itself due to feasibility constraints. This was conducted with batch size 500, Adam optimiser with learning rate 0.1, MSE loss and for 50 epochs. The size of the network in terms of the number of neurons was chosen based off the fact that each 64 bit partition, x_i, requires approximately 30-34 XORs to be conducted for inversion of the linear layer. Specifically for x_0, 31 XORs are required. This was computed via code in the appendix. Therefore, $31 \log(31) \approx 154$ neurons was used as a baseline as per a binary tree to evaluate 31 XORs.

It was found that the MSE had hovered around approximately 0.25 as seen from Figure 15. From here, to evaluate the ability of the network to solve the inverse XOR problem, the hamming distance between predicted and actual data was computed for the validation dataset. It was found that the average hamming distance was 32.00547. This implies that there were approximately 32 bits different when comparing the predicted and actual outputs, essentially being random. This would imply that it failed to learn the inverse XOR problem.

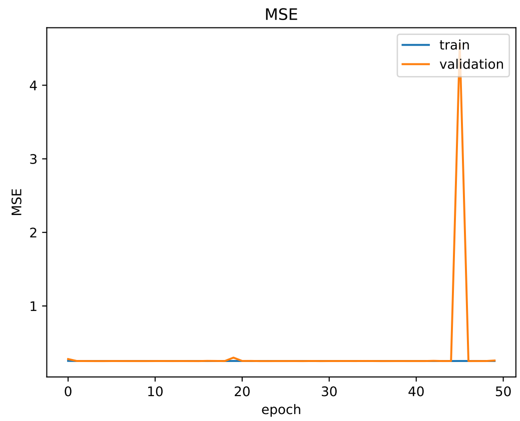

Figure 15: Loss Plot of Inverse XOR Network

This was further verified by the transfer learning. Here, the network began with an input layer of shape 1x640 followed by a dense layer of 64 units with batch normalisation and ReLu activation. From here, the dense block of the Inverse XOR network was connected, followed by another dense layer of 640 units with batch normalisation and ReLu activation. Following this, everything from the bit-sliced convolution block onwards of the initial network was appended. All weights except the those of the last layer of the initial network were frozen. This was then run for 50 epochs with the specifications as per the results in Table 6. The highest accuracy obtained was only 0.502, pointing to it failing to learn the inverse XOR.

Expanding upon this, a variant of the architecture in Figure 14 was attempted. It is illustrated in Figure 16. Here, the dense block was used for transfer learning as per the paragraph above. This was run for 50 epochs and with learning rate varied between upper and lower bounds 0.05, 0.01 respectively. The learning rate was decided from conducting the learning rate test and the remaining specifications were as per the results in Table 6. Similar to the paragraph above, the highest accuracy obtained was only 0.502.

5.5 Summary of Results

Of the experiments conducted on the variants, only the reshaped inputs and biased inputs variants outperformed Gohr's architecture, marginally. In addition, the biased input variant required significantly less time per epoch over the reshaped input variant, being more efficient.

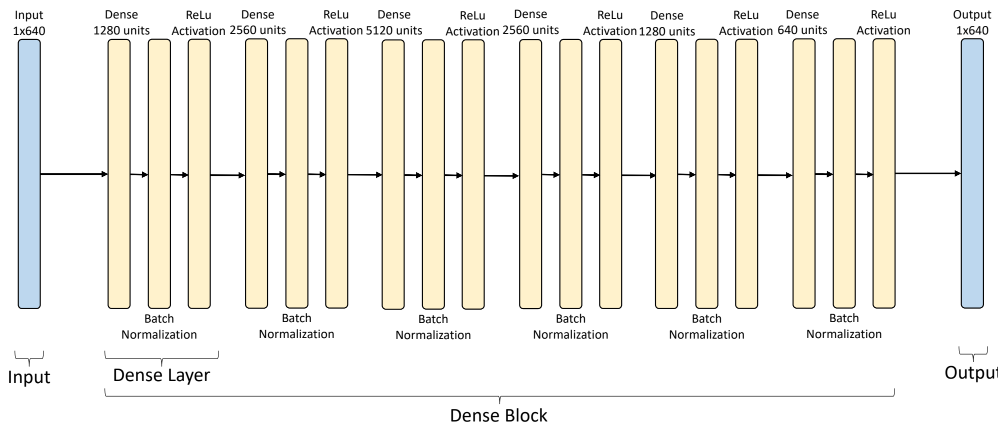

Figure 16: Inverse XOR Architecture

6 Conclusion

In summary, we were able to successfully adapt Gohr's network architecture for up to 3.5 rounds of the ASCON round function, having attained validation accuracies close to 1. The specifications were then further tuned for greater accuracy too. Following this, we proposed and tested a multitude of different architectures. This included variants which attempted to account for rotations, variants which took motivation from multiple differential cryptanalysis, variants which restricted inputs to biased inputs and variants which involved transfer learning and linear diffusion inversion. Of these, only reshaping and biased inputs had provided marginally improved results to Gohr's architecture, with the rest performing worse than it.

From these results, there are several further directions that could be explored. One would be to improve upon the inverse XOR attempt at transfer learning. Larger dataset sizes and larger model capacity could be applied to improve on the accuracy of the inverse XOR layer. Different architectures could also be attempted for the inverse XOR. This may then provide improved results when concatenated with the initial layer. Another direction would be to rerun the experiments that had been conducted with larger parameters. That is, larger dataset sizes, larger network depth and more epochs. This would require more computing power but may potentially yield better results. Finally, a third direction would be to combine the variants with the biased inputs variant. That is, restrict the inputs of the variants to those biased.

References

Biham, E., & Shamir, A. (1991). Differential cryptanalysis of des-like cryptosystems. *J. Cryptology*, *4*, 3-72. doi: 10.1007/BF00630563

Blondeau, C., & Gérard, B. (2011). *Multiple differential cryptanalysis: Theory and practice (corrected).* Cryptology ePrint Archive, Report 2011/115. (`https://ia.cr/2011/115`)

Breier, J., Hou, X., Jap, D., Ma, L., Bhasin, S., & Liu, Y. (2018). Practical fault attack on deep neural networks. In *Proceedings of the 2018 acm sigsac conference on computer and communications security* (pp. 2204–2206).

Dobraunig, C., Eichlseder, M., Mendel, F., & Schläffer, M. (2019). *Ascon v1.2.* Submission to Round 1 of the NIST Lightweight Cryptography project. Retrieved from `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf`

Gerault, D., Peyrin, T., & Tan, Q. Q. (2021). Exploring differential-based distinguishers and forgeries for ascon. *Cryptology ePrint Archive*.

Gohr, A. (2019). Improving attacks on round-reduced speck32/64 using deep learning. In *Crypto.*

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep learning.* Cambridge, MA, USA: MIT Press. (`http://www.deeplearningbook.org`)

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *2017 ieee winter conference on applications of computer vision (wacv)* (pp. 464–472).

Sultana, F., Sufian, A., & Dutta, P. (2018). Advancements in image classification using convolutional neural network. In *2018 fourth international conference on research in computational intelligence and communication networks (icrcicn)* (pp. 122–129).

Tezcan, C. (2016). Truncated, impossible, and improbable differential analysis of ascon. *IACR Cryptol. ePrint Arch.*, *2016*, 490.

A Appendix

A.1 ASCON

A.1.1 Round Function

```python
import math

def split(s): #split s into 5 64 bit blocks
    y = []
    for i in range(5):
        y.insert(0,((s >> (i*64)) & 0xFFFFFFFFFFFFFFFF)) #shift s by a mult. of 64 then AND with 64 1s.
    return y

def merge(s): #merge 5 64 bit blocks into 1
    y=0
    for i in range(5):
        y = y ^ (s[4-i] << (i * 64)) #starting from last block of s, shift left by mult of 64 then XOR iteratively
    return y

def addConstant(y, r, power): #power is a or b, y here is a list containing xis
    p12 = [0xf0, 0xe1, 0xd2, 0xc3, 0xb4, 0xa5, 0x96, 0x87, 0x78, 0x69, 0x5a, 0x4b]
    p8 = [0xb4, 0xa5, 0x96, 0x87, 0x78, 0x69, 0x5a, 0x4b]
    p6 = [0x96, 0x87, 0x78, 0x69, 0x5a, 0x4b]
    if power == 12: cr = p12[r]
    elif power == 8: cr = p8[r]
    else:
        cr = p6[r]
    y[2] = y[2]^cr

    return y

def sub(y):
    sbox = [0x4, 0xb, 0x1f, 0x14, 0x1a, 0x15, 0x9, 0x2, 0x1b, 0x5, 0x8, 0x12, 0x1d, 0x3, 0x6, 0x1c, 0x1e, 0x13, 0x7, 0xe,
            0x0, 0xd, 0x11, 0x18, 0x10, 0xc, 0x1, 0x19, 0x16, 0xa, 0xf, 0x17]
    x = []

    for i in range(64): #splitting into 5 bit blocks and substituting
        temp = 0b0
        for j in range(5):
            temp ^= (((y[4-j] & (0b1 << i)) >> i) << j) #note, sbox counts from bottom up.
        x.insert(0, temp)
        x[0] = sbox[x[0]]

    for j in range(5): #merging back into 5 64 bit blocks.
        temp = 0b0
        for i in range(64):
            temp ^= (((x[i] & (0b1 << j)) >> j) << (63-i))
        y[4-j] = temp
```

```

    return y #return as list of 5 64 bit blocks

def lindiff(y): #y here is a list containing xis
    def circ_shift(s,r):
        temp = 0
        for i in range(r): #create 11111s
            temp += 2**(i)
        return ((s>>r)^((s & temp) << (64-r)))
    temp = y
    y[0] = temp[0] ^ circ_shift(temp[0],19) ^ circ_shift(temp[0],28)
    y[1] = temp[1] ^ circ_shift(temp[1],61) ^ circ_shift(temp[1],39)
    y[2] = temp[2] ^ circ_shift(temp[2],1) ^ circ_shift(temp[2],6)
    y[3] = temp[3] ^ circ_shift(temp[3],10) ^ circ_shift(temp[3],17)
    y[4] = temp[4] ^ circ_shift(temp[4],7) ^ circ_shift(temp[4],41)

    return y

def perm(s,power): #s is data, a/b is power i.e number of rounds
    if power - math.floor(power) > 0: #for when there is a 3.5 round component in power
        s = split(s)
        power = math.floor(power)
        for i in range(power):
            s = addConstant(s, i, power)  # note this used p^6 if <= 6 power used
            s = sub(s)
            s = lindiff(s)

        s = addConstant(s, power, power+1)  # note this used p^6 if <= 6 power used
        s = sub(s)
        return merge(s)

    else:
        s = split(s)

        for i in range(power):
            s = addConstant(s, i, power)
            s = sub(s)
            s = lindiff(s)

        return merge(s)
```

A.1.2 Main ASCON file

```
from Perm import perm
import math

def s_split(s,r): #splits state S in sr and sc
    sr = (s >> (320 - r))
    temp = 0
    for i in range(320-r):  # create 11111s
```

```
        temp += 2 ** (i)
    sc = s & temp
    return sr, sc

def s_merge(sr,sc,r):#merges sr and sc into s
    s = (sr << (320-r)) ^ sc
    return s

def c_merge(c,r): #merges everything but the last ciphertext block, used for plaintext in decryption too
    y = 0
    for i in range(len(c)-1):
        y = y ^ (c[len(c) - i - 2] << (i * r))
    return y

def int_to_bytes(x: int) -> bytes:
    return x.to_bytes((x.bit_length() + 7) // 8, ’big’)

def int_from_bytes(xbytes: bytes) -> int:
    return int.from_bytes(xbytes, ’big’)

def get_random_bytes(num):
    import os
    return bytes(bytearray(os.urandom(num)))

def enc(K,N,A,P):

    #Initialisation
    K = int.from_bytes(K, "big")
    N = int.from_bytes(N, "big")
    iv = 0x80400c0600000000
    a = 0x0c
    b = 0x06
    r = 0x40
    k = 0x80
    s = (((iv << k) ^ K) << 128) ^ N  #K.bit_length()) ^ K #had set size to 128 based off test code.
    s = (perm(s, a) ^ ((0 << 320) ^ K))

    #Processing Associated data
    Asplit = []
    if A != 0 and A!=b’’:
        count = 0 #computing number of 0s to append
        temp = len(A)*8
        while temp > 0:
            temp-=r
            count+=1
        number_of_zeros = r*count - len(A)*8
        if number_of_zeros != 0: number_of_zeros -= 1
        else: number_of_zeros = 63

        A = int.from_bytes(A, "big")
        temp = ((A << 1) ^ 1) << (number_of_zeros) #note that 0s in front of bit string are not to be removed

        for i in range(math.ceil(temp.bit_length()/r)): #splitting A padded into blocks
```

```
            Asplit.insert(0, ((temp >> (i * r)) & 0xFFFFFFFFFFFFFFFF))

        for i in range(len(Asplit)):
            sr,sc = s_split(s,r)
            s = perm(((sr^Asplit[i])<<(320-r))^sc,b)
    s = s ^ 1
    # Processing Plaintext
    count = 0  # computing number of 0s to append
    plaintextlen = len(P)
    temp = len(P) * 8
    while temp > 0:
        temp -= r
        count += 1
    number_of_zeros = r * count - len(P) * 8
    if P == b'': number_of_zeros = 63
    elif number_of_zeros != 0:
        number_of_zeros -= 1  # if length isnt a mult of 64, -1 for the extra 1 to append
    else:
        number_of_zeros = 63  # if its a mult of 64
    P = int.from_bytes(P, "big")
    Psplit = []
    temp = ((P << 1) ^ 1) << (number_of_zeros)  # padded P
    for i in range(math.ceil(temp.bit_length() / r)):
        Psplit.insert(0, ((temp >> (i * r)) & 0xFFFFFFFFFFFFFFFF))
    Csplit = []
    for i in range(len(Psplit) - 1):  # preparing all but last ciphertext, algo
        sr, sc = s_split(s, r)
        sr = sr ^ Psplit[i]
        Csplit.append(sr)
        s = s_merge(sr, sc, r)
        s = perm(s, b)
    sr, sc = s_split(s, r)
    sr = sr ^ Psplit[-1]  # the last Ci
    s = s_merge(sr, sc, r)
    Csplit.append(sr >> (number_of_zeros + 1))
    ciphertext = (c_merge(Csplit, r) << (r - number_of_zeros - 1)) ^ Csplit[-1]


    #Finalisation
    s = perm(s ^ (K << (320-r-k)),a)
    temp = 0
    for i in range(128):  # create 11111s to take ceiling
        temp += 2 ** (i)
    T = (s & temp) ^ (K & temp)
    print("Ciphertext: " + hex(ciphertext), "Tag:" + hex(T))
    if len(int_to_bytes(ciphertext))!=plaintextlen: cipher = (plaintextlen - len(int_to_bytes(ciphertext)))*b'\00' + int_to_bytes(ciphertext) #adding 0s in front to match byte format
    else: cipher =  int_to_bytes(ciphertext)
    return [cipher, int_to_bytes(T)]
```

```

def dec(K,N,A,C,T):

    #Initialisation
    K = int.from_bytes(K, "big")
    N = int.from_bytes(N, "big")
    iv = 0x80400c0600000000
    a = 0x0c
    b = 0x06
    r = 0x40
    k = 0x80
    s = (((iv << k) ^ K) << 128) ^ N  #K.bit_length()) ^ K #had set size to 128 based off test code.
    s = (perm(s, a) ^ ((0 << 320) ^ K))

    #Processing Associated data
    Asplit = []
    if A != 0 and A!=b'':
        count = 0 #computing number of 0s to append
        temp = len(A)*8
        while temp > 0:
            temp-=r
            count+=1
        number_of_zeros = r*count - len(A)*8
        if number_of_zeros != 0:number_of_zeros -= 1 #if len A isnt a mult of 64
        else:number_of_zeros = 63 #if len A is a mult of 64

        A = int.from_bytes(A, "big")
        temp = ((A << 1) ^ 1) << (number_of_zeros) #note that 0s in front of bit string are not to be removed
        for i in range(math.ceil(temp.bit_length()/r)): #splitting A padded into blocks, +1 is to take up
            Asplit.insert(0, ((temp >> (i * r)) & 0xFFFFFFFFFFFFFFFF))

        for i in range(len(Asplit)):
            sr,sc = s_split(s,r)
            s = perm(((sr^Asplit[i])<<(320-r))^sc,b)
    s = s ^ 1

    # Processing Ciphertext
    cipherlen = len(C)
    lastlen = (len(C) * 8)%64 #last ciphertext block length
    if (int.from_bytes(C, "big") == 0) and (C!=b''): lastlen = 8 #for when its b'\x00' instead of b''
    temp = 0
    for i in range(lastlen):  # create 11111s
        temp += 2 ** (i)
    C = int.from_bytes(C, "big")
    Csplit = []
    Csplit.insert(0,C & temp) #insert last block
    temp = C >> lastlen #removing last block to split the rest
    for i in range(math.ceil(temp.bit_length() / r)): #splitting into 64 bit blocks
        Csplit.insert(0, ((temp >> (i * r)) & 0xFFFFFFFFFFFFFFFF))
    Psplit = []
```

```
    for i in range(len(Csplit)-1): #applying algo to everything but last plaintext
        sr, sc = s_split(s, r)
        pi = sr ^ Csplit[i]
        Psplit.append(pi)
        s = s_merge(Csplit[i],sc,r)
        s = perm(s,b)
    sr, sc = s_split(s, r)

    pi = (sr>>(r-lastlen)) ^ Csplit[-1] #applying algo to last plaintext block
    Psplit.append(pi)
    sr = sr ^ (((pi << 1) ^ 1) << (64-1-lastlen))
    s = s_merge(sr,sc,r)
    plaintext = (c_merge(Psplit, r) << (lastlen)) ^ Psplit[-1] #combining Psplit list to plaintext integer

    #Finalisation
    s = perm(s ^ (K << (320-r-k)),a)
    temp = 0
    for i in range(128):  # create 11111s
        temp += 2 ** (i)
    T_new = (s & temp) ^ (K & temp)
    if T_new == int_from_bytes(T):
        print("Plaintext: "+hex(plaintext))
        if len(int_to_bytes(plaintext)) != cipherlen: #adding 0s in front to match byte format
            plaintext = (cipherlen - len(int_to_bytes(plaintext))) * b'\00' + int_to_bytes(plaintext)
        else:
            plaintext = int_to_bytes(plaintext)
        return plaintext
    else:
        print("Different Tag")
        return None
```

A.2 Adapted Neural Network

```
from Perm import perm
import secrets
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, CSVLogger
from keras.models import Model, load_model
from keras.layers import Dense, Conv1D, Input, Reshape, Permute, Add, Flatten, BatchNormalization, Activation
from keras.regularizers import l2
from LRFinder.keras_callback import LRFinder
import math
import matplotlib.pyplot as plt

def gen_inputs(data_size,input_diff, rounds):
    data = []
    diff = []
```

```python
    for i in range(data_size):
        x1 = secrets.randbits(320) #generate random input
        temp = secrets.randbits(1) #1 or 0 i.e fixed or random input diff
        if temp == 1: x2 = x1 ^ input_diff #generate second input with inputdiff
        elif temp == 0: x2 = secrets.randbits(320) #generate second input randomly
        entry1 = []
        entry2 = []
        y1 = perm(x1,rounds) #perm outputs
        y2 = perm(x2,rounds) #perm outputs

        for j in range(0,320): #converting yis to bitwise entries in string
            entry1.insert(0,((y1 >> j) & 1))
            entry2.insert(0,((y2 >> j) & 1))

        entry1 = np.asarray(entry1).astype(np.uint8)
        entry2 = np.asarray(entry2).astype(np.uint8)
        entry = np.concatenate((entry1, entry2)) #(320 bit output,320 bit output) i.e 640 bit list with bitwise entries
        if temp == 1: diff+=[1]
        elif temp == 0: diff+=[0] #if random input diff
        data.append(entry) #add entry to dataset

    data = np.array(data)
    diff = np.array(diff)

    return data, diff

def neural_net(num_filters=32, num_outputs=1, d1=64, d2=64, ks=3, depth=5, reg_param=0.0001, final_activation='sigmoid'):
  #Input and preprocessing layers
  inp = Input(shape=(640,))
  rs = Reshape((10,64))(inp)
  perm = Permute((2, 1))(rs)

  #single residual layer(bit sliced)(block 1)
  conv0 = Conv1D(num_filters, kernel_size=1, padding='same', kernel_regularizer=l2(reg_param))(perm)
  conv0 = BatchNormalization()(conv0)
  conv0 = Activation('relu')(conv0)

  #add residual blocks(blocks 2-i)
  shortcut = conv0
  for i in range(depth):
    conv1 = Conv1D(num_filters, kernel_size=ks, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    conv2 = Conv1D(num_filters, kernel_size=ks, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    shortcut = Add()([shortcut, conv2])

```

```
  #add classification block
  flat1 = Flatten()(shortcut)
  dense1 = Dense(d1,kernel_regularizer=l2(reg_param))(flat1)
  dense1 = BatchNormalization()(dense1)
  dense1 = Activation('relu')(dense1)
  dense2 = Dense(d2, kernel_regularizer=l2(reg_param))(dense1)
  dense2 = BatchNormalization()(dense2)
  dense2 = Activation('relu')(dense2)
  out = Dense(num_outputs, activation=final_activation, kernel_regularizer=l2(reg_param))(dense2)
  model = Model(inputs=inp, outputs=out)
  return(model)

def cyclic_lr(n, beta, alpha): #learning rate function
  to_return = lambda i: alpha + (n - i % (n+1))/n * (beta - alpha)
  return(to_return)

#Data Prep
differences = [2**63+2**127, 2**63, 2**319] #only state[3][0] = 1 & state[4][0] = 1, only state[4][0] = 1, some random state
round = [1,2,3,"3_5",4,5,6]

#for input_diff in differences: #to generate data size 10**7
#     for rounds in round:
#         print("Rounds = "+str(rounds)+" Input diff = "+str(hex(input_diff)))
#         df_X, df_y = gen_inputs(data_size=10 ** 7, input_diff=input_diff, rounds=rounds)
#         np.save("10p7_data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_X)
#         np.save("10p7_data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_y)

for input_diff in differences: #to generate data size 10**6
    for rounds in round:
        print("Rounds = "+str(rounds)+" Input diff = "+str(hex(input_diff)))
        df_X, df_y = gen_inputs(data_size=10 ** 6, input_diff=input_diff, rounds=rounds)
        np.save("data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_X)
        np.save("data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_y)

rounds = round[4] #rounds of round function used
input_diff = differences[0] #input diff
folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/" #folder link

#df_X = np.load(folder + "10p7_data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True)
#df_y = np.load(folder + "10p7_data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True)

df_X = np.load(folder + "data_X_rounds=" + str(rounds) + "
```

```
    _input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True
    )
df_y = np.load(folder + "data_y_rounds=" + str(rounds) + "
    _input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True
    )

#folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(
    hex(input_diff)) + "/" + "testing_" #temp link to save test
    files

#Neural Network Parameters
num_epochs = 1000
depth = 10
batch_size = 5000

tuning of learning rate(minmax)
start_lr = 1e-6
end_lr = 1e0
lr_finder = LRFinder(min_lr=start_lr, max_lr=end_lr)

net = neural_net(depth=depth, reg_param=10**-5, ks=3, num_filters
    =32, d1 = 64, d2=64) #generate network
net.compile(optimizer='Adam',loss='mse',metrics=['acc'])

X_train, X_test, y_train, y_test = train_test_split(df_X, df_y,
    test_size=0.1, random_state=42)

#set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel_depth='+str(depth)
    +'.h5', monitor='val_loss', save_best_only = True)
#CSV_Logger
log_csv = CSVLogger(folder + 'log_depth='+str(depth)+'.csv',
    separator=',', append=True)
#cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9, 0.02, 0.0001))#0.002,
    0.0001))
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
    batch_size,validation_data=(X_test, y_test), callbacks=[
    lr_finder])
fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
    batch_size,validation_data=(X_test, y_test), callbacks=[lr,
    checkpoint, log_csv])
#print(net.summary())
print("Best validation accuracy: ", np.max(fitted.history['val_acc'
    ]))

#Accuracy Plot
plt.plot(fitted.history['acc'])
plt.plot(fitted.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

#Loss Plot
plt.plot(fitted.history['loss'])
```

```
plt.plot(fitted.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

A.3 Reshaped Input Neural Network

```

from Perm import perm
import secrets
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras.callbacks import ModelCheckpoint, LearningRateScheduler,
     CSVLogger
from keras.models import Model, load_model
from keras.layers import Dense, Conv1D, Input, Reshape, Permute,
    Add, Flatten, BatchNormalization, Activation
from keras.regularizers import l2
from LRFinder.keras_callback import LRFinder
import math
import matplotlib.pyplot as plt


def gen_inputs(data_size, input_diff, rounds, reshape):
    data = []
    diff = []
    for i in range(data_size):
        x1 = secrets.randbits(320)  # generate random input
        temp = secrets.randbits(1)  # 1 or 0 i.e fixed or random
    input diff
        if temp == 1:
            x2 = x1 ^ input_diff  # generate second input with
    inputdiff
        elif temp == 0:
            x2 = secrets.randbits(320)  # generate second input
    randomly
        entry1 = []
        entry2 = []
        y1 = perm(x1, rounds)  # perm outputs
        y2 = perm(x2, rounds)  # perm outputs

        for j in range(0, 320):  # converting yis to bitwise
    entries in string
            entry1.insert(0, ((y1 >> j) & 1))
            entry2.insert(0, ((y2 >> j) & 1))

        entry1 = np.asarray(entry1).astype(np.uint8)
        entry2 = np.asarray(entry2).astype(np.uint8)

        if reshape == True:
            entry = np.concatenate((entry1, entry2)) #joining (c0,
    c1)
```

```python
            entry = np.reshape(entry,(10,64)) #having shape (c0x0, c0x1,..,c1x0,c1x1...) in 10x64
            entry = np.concatenate((entry, entry), axis=1)

        else: entry = np.concatenate((entry1, entry2))  # (320 bit output,320 bit output) i.e 640 bit list with bitwise entries

        if temp == 1:
            diff += [1]
        elif temp == 0:
            diff += [0]  # if random input diff
        data.append(entry)  # add entry to dataset

    data = np.array(data)
    diff = np.array(diff)

    return data, diff


def neural_net(num_filters=32, num_outputs=1, d1=64, d2=64, ks=3, depth=5, reg_param=0.0001,
               final_activation='sigmoid'):
    # Input and preprocessing layers
    inp = Input(shape=(10,128))
    perm = Permute((2, 1))(inp)
    #perm = inp

    # single residual layer(bit sliced)(block 1)
    conv0 = Conv1D(num_filters, kernel_size=1, padding='same', kernel_regularizer=l2(reg_param))(perm)
    conv0 = BatchNormalization()(conv0)
    conv0 = Activation('relu')(conv0)

    # add residual blocks(blocks 2-i)
    shortcut = conv0
    for i in range(depth):
        conv1 = Conv1D(num_filters, kernel_size=ks, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
        conv1 = BatchNormalization()(conv1)
        conv1 = Activation('relu')(conv1)
        conv2 = Conv1D(num_filters, kernel_size=ks, padding='same', kernel_regularizer=l2(reg_param))(conv1)
        conv2 = BatchNormalization()(conv2)
        conv2 = Activation('relu')(conv2)
        shortcut = Add()([shortcut, conv2])

#Uncomment portion below and comment out for loop above for adjusted architecture

    #conv1 = Conv1D(num_filters, kernel_size=3, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=3, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
```

```
    #shortcut = Add()([shortcut, conv2])

    #conv1 = Conv1D(num_filters, kernel_size=6, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=6, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
    #shortcut = Add()([shortcut, conv2])

    #conv1 = Conv1D(num_filters, kernel_size=7, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=7, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
    #shortcut = Add()([shortcut, conv2])

    #conv1 = Conv1D(num_filters, kernel_size=10, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=10, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
    #shortcut = Add()([shortcut, conv2])

    #conv1 = Conv1D(num_filters, kernel_size=17, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=17, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
    #shortcut = Add()([shortcut, conv2])

    #conv1 = Conv1D(num_filters, kernel_size=19, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=19, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
    #shortcut = Add()([shortcut, conv2])

    #conv1 = Conv1D(num_filters, kernel_size=28, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    #conv1 = BatchNormalization()(conv1)
    #conv1 = Activation('relu')(conv1)
    #conv2 = Conv1D(num_filters, kernel_size=28, padding='same',
```

```
    kernel_regularizer=l2(reg_param))(conv1)
    #conv2 = BatchNormalization()(conv2)
    #conv2 = Activation('relu')(conv2)
    #shortcut = Add()([shortcut, conv2])

    # add classification block
    flat1 = Flatten()(shortcut)
    dense1 = Dense(d1, kernel_regularizer=l2(reg_param))(flat1)
    dense1 = BatchNormalization()(dense1)
    dense1 = Activation('relu')(dense1)
    dense2 = Dense(d2, kernel_regularizer=l2(reg_param))(dense1)
    dense2 = BatchNormalization()(dense2)
    dense2 = Activation('relu')(dense2)
    out = Dense(num_outputs, activation=final_activation, kernel_regularizer=l2(reg_param))(dense2)
    model = Model(inputs=inp, outputs=out)
    return (model)


def cyclic_lr(n, beta, alpha):  # learning rate function
    to_return = lambda i: alpha + (n - i % (n + 1)) / n * (beta - alpha)
    return (to_return)


# Data Prep
differences = [2 ** 63 + 2 ** 127, 2 ** 63]  # only state[3][0] = 1 & state[4][0] = 1, only state[4][0] = 1
round = [3, 4, 5]

for input_diff in differences:
    for rounds in round:
        print("Rounds = "+str(rounds)+" Input diff = "+str(hex(input_diff)))
        df_X, df_y = gen_inputs(data_size=10 ** 6, input_diff=input_diff, rounds=rounds, reshape = True)
        np.save("data_X_reshaped_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_X)
        np.save("data_y_reshaped_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_y)

rounds = round[1]  # rounds of round function used
input_diff = differences[0]  # input diff
folder = "./Models/reshaped_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/"  # folder link
df_X = np.load(folder + "data_X_reshaped_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",
               allow_pickle=True)
df_y = np.load(folder + "data_y_reshaped_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",
               allow_pickle=True)
# folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/" + "testing_" #temp link to save test files

# Neural Network Parameters
num_epochs = 20
```

```
depth = 1
batch_size = 5000

# tuning of learning rate(minmax)
# start_lr = 1e-6
# end_lr = 1e0
# lr_finder = LRFinder(min_lr=start_lr, max_lr=end_lr)

net = neural_net(depth=depth, reg_param=10 ** -5, ks=12, num_filters=32)  # generate network
net.compile(optimizer='Adam', loss='mse', metrics=['acc'])

X_train, X_test, y_train, y_test = train_test_split(df_X, df_y, test_size=0.1, random_state=42)

# set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel_depth=' + str(depth) + '.h5', monitor='val_loss', save_best_only=True)
# CSV_Logger
log_csv = CSVLogger(folder + 'log_depth=' + str(depth) + '.csv', separator=',', append=True)
# cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9, 0.02, 0.0001))  # 0.002, 0.0001))
# fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr_finder])
fitted = net.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, validation_data=(X_test, y_test),
                 callbacks=[lr, checkpoint, log_csv])
# print(net.summary())
print("Best validation accuracy: ", np.max(fitted.history['val_acc']))

# Accuracy Plot
plt.plot(fitted.history['acc'])
plt.plot(fitted.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

# Loss Plot
plt.plot(fitted.history['loss'])
plt.plot(fitted.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

A.4 Multiple Differential Network

```
from Perm import perm
import secrets
```

```python
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, CSVLogger
from keras.models import Model, load_model
from keras.layers import Dense, Conv1D, Input, Reshape, Permute, Add, Flatten, BatchNormalization, Activation
from keras.regularizers import l2
from CLR.clr_callback import CyclicLR
from LRFinder.keras_callback import LRFinder
import math
import matplotlib.pyplot as plt
from pickle import dump


def gen_inputs(data_size, input_diff_1, input_diff_2, rounds):
    data = []
    diff = []
    for i in range(data_size):
        x0 = secrets.randbits(320)  # generate random input
        x2 = secrets.randbits(320)  # generate random input
        temp = secrets.randbits(1)  # 1 or 0 i.e either both fixed or both random input diff
        if temp == 1:
            x1 = x0 ^ input_diff_1  # generate second input with inputdiff
            x3 = x2 ^ input_diff_2  # generate second input with inputdiff
        elif temp == 0:
            x1 = secrets.randbits(320)  # generate second input randomly
            x3 = secrets.randbits(320)  # generate second input randomly

        entry0 = []
        entry1 = []
        entry2 = []
        entry3 = []
        y0 = perm(x0, rounds)  # perm outputs
        y1 = perm(x1, rounds)
        y2 = perm(x2, rounds)
        y3 = perm(x3, rounds)

        for j in range(0, 320):  # converting yis to bitwise entries in string
            entry0.insert(0, ((y0 >> j) & 1))
            entry1.insert(0, ((y1 >> j) & 1))
            entry2.insert(0, ((y2 >> j) & 1))
            entry3.insert(0, ((y3 >> j) & 1))

        entry0 = np.asarray(entry0).astype(np.uint8)
        entry1 = np.asarray(entry1).astype(np.uint8)
        entry2 = np.asarray(entry2).astype(np.uint8)
        entry3 = np.asarray(entry3).astype(np.uint8)

        entry = np.concatenate((entry0, entry1, entry2, entry3))  #
```

```
     (320 bit output,320 bit output,...) i.e 1280 bit list with
    bitwise entries

        if temp == 1:
            diff += [1]
        elif temp == 0:
            diff += [0]  # if random input diff
        data.append(entry)  # add entry to dataset

    data = np.array(data)
    diff = np.array(diff)

    return data, diff


def neural_net(num_filters=32, num_outputs=1, d1=64, d2=64, ks=3,
    depth=5, reg_param=0.0001, final_activation='sigmoid'):
  #Input and preprocessing layers
  inp = Input(shape=(1280,))
  rs = Reshape((10,128))(inp)
  perm = Permute((2, 1))(rs)

  #single residual layer(bit sliced)(block 1)
  conv0 = Conv1D(num_filters, kernel_size=1, padding='same',
    kernel_regularizer=l2(reg_param))(perm)
  conv0 = BatchNormalization()(conv0)
  conv0 = Activation('relu')(conv0)

  #add residual blocks(blocks 2-i)
  shortcut = conv0
  for i in range(depth):
    conv1 = Conv1D(num_filters, kernel_size=ks, padding='same',
    kernel_regularizer=l2(reg_param))(shortcut)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    conv2 = Conv1D(num_filters, kernel_size=ks, padding='same',
    kernel_regularizer=l2(reg_param))(conv1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    shortcut = Add()([shortcut, conv2])

  #add classification block
  flat1 = Flatten()(shortcut)
  dense1 = Dense(d1,kernel_regularizer=l2(reg_param))(flat1)
  dense1 = BatchNormalization()(dense1)
  dense1 = Activation('relu')(dense1)
  dense2 = Dense(d2, kernel_regularizer=l2(reg_param))(dense1)
  dense2 = BatchNormalization()(dense2)
  dense2 = Activation('relu')(dense2)
  out = Dense(num_outputs, activation=final_activation,
    kernel_regularizer=l2(reg_param))(dense2)
  model = Model(inputs=inp, outputs=out)
  return(model)


def cyclic_lr(n, beta, alpha):  # learning rate function
    to_return = lambda i: alpha + (n - i % (n + 1)) / n * (beta -
```

```
    alpha)
    return (to_return)


# Data Prep
differences = [2 ** 63 + 2 ** 127, 2 ** 63]  # only state[3][0] = 1 & state[4][0] = 1, only state[4][0] = 1
round = [3, 4, 5]
differences = [2 ** 63 + 2 ** 127]
round = [4]
input_diff1 = 2 ** 63 + 2 ** 127
input_diff2 = 2 ** 63

#for input_diff in differences:
#    for rounds in round:
#        print("Rounds = "+str(rounds)+" Input diff = "+str(hex(input_diff)))
#        df_X, df_y = gen_inputs(data_size=10 ** 6, input_diff_1=input_diff, input_diff_2=input_diff, rounds=rounds)
#        np.save("multi_data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff1)) + "_"+ str(hex(input_diff2)) + ".npy", df_X)
#        np.save("multi_data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff1)) + "_"+ str(hex(input_diff2)) + ".npy", df_y)

rounds = round[0]  # rounds of round function used
input_diff = differences[0]  # input diff
folder = "./Models/reshaped_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/"  # folder link
df_X = np.load("multi_data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff1)) + "_"+ str(hex(input_diff2)) +".npy", allow_pickle=True)
df_y = np.load("multi_data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff1)) + "_"+ str(hex(input_diff2)) +".npy", allow_pickle=True)
# folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/" + "testing_" #temp link to save test files

# Neural Network Parameters
num_epochs = 20
depth = 1
batch_size = 5000

# tuning of learning rate(minmax)
# start_lr = 1e-6
# end_lr = 1e0
# lr_finder = LRFinder(min_lr=start_lr, max_lr=end_lr)

net = neural_net(depth=depth, reg_param=10 ** -5, ks=3, num_filters=32)  # generate network
net.compile(optimizer='Adam', loss='mse', metrics=['acc'])

X_train, X_test, y_train, y_test = train_test_split(df_X, df_y, test_size=0.1, random_state=42)

```

```
# Cyclic learning rate
# clr_step_size = int(4 * (len(X_train)/batch_size))
# max_lr = 1e-1
# base_lr = max_lr * 0.3#1e-4
# mode='triangular'
# clr = CyclicLR(base_lr=base_lr, max_lr=max_lr, step_size=clr_step_size, mode=mode)

# set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel_depth=' + str(depth) + '.h5', monitor='val_loss', save_best_only=True)
# CSV_Logger
log_csv = CSVLogger(folder + 'log_depth=' + str(depth) + '.csv', separator=',', append=True)
# cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9, 0.02, 0.001))  # 0.002, 0.0001))
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr_finder])
fitted = net.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, validation_data=(X_test, y_test),
                 callbacks=[lr, checkpoint, log_csv])
# print(net.summary())
print("Best validation accuracy: ", np.max(fitted.history['val_acc']))

# Accuracy Plot
plt.plot(fitted.history['acc'])
plt.plot(fitted.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

# Loss Plot
plt.plot(fitted.history['loss'])
plt.plot(fitted.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

A.5 Biased Inputs

A.5.1 Bias Computation

```
import numpy as np

#Computing Bias on difference for 640bit input. i.e. computing probability that for a given entry in 320bits, the two correspondng points are diff.
#Data Prep
```

```
differences = [2**63+2**127, 2**63, 2**319] #only state[3][0] = 1 &
     state[4][0] = 1, only state[4][0] = 1, some random state
round = [1,2,3,"3_5",4,5,6]
bias = []
f = open("Bias.csv","a",newline = "")
for d in range(2):
    for r in range(2,5):
        bias = []
        rounds = round[r]  # rounds of round function used
        input_diff = differences[d]  # input diff
        folder = "./Models/rounds=" + str(rounds) + "_input_diff="
    + str(hex(input_diff)) + "/"  # folder link
        df_X = np.load(folder + "data_X_rounds=" + str(rounds) + "
    _input_diff=" + str(hex(input_diff)) + ".npy", allow_pickle=
    True)

        for j in range(320): #index
            ones = 0
            zeros = 0
            for i in range(len(df_X)): #sample
                if df_X[i][j] ^ df_X[i][j + 320] == 1:
                    ones += 1
                else:
                    zeros += 1
            bias += [ones / len(df_X)]
        bias = np.reshape(bias, (5, 64))
        print(bias)
        f = open("Bias.csv","a",newline = "")
        writer = csv.writer(f)
        writer.writerow(["rounds=" + str(rounds) + "_input_diff=" +
     str(hex(input_diff))])
        writer.writerows(bias)
        f.close()
```

A.5.2 Network with Biased Inputs

```
from Perm import perm
import secrets
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras.callbacks import ModelCheckpoint, LearningRateScheduler,
     CSVLogger
from keras.models import Model, load_model
from keras.layers import Dense, Conv1D, Input, Reshape, Permute,
    Add, Flatten, BatchNormalization, Activation
from keras.regularizers import l2
from LRFinder.keras_callback import LRFinder
import math
import matplotlib.pyplot as plt

def onlyBias(df_X): #to remove unbiased bits from 4 round input of
    diff 2**63+2**127
    indexes = [1, 5, 18, 27, 36, 37, 40, 46, 49, 50, 52, 73, 91,
    98, 117, 122, 123,
```

```
                128, 146, 155, 160, 170, 179, 181, 188, 191, 197,
    205, 207, 224, 229, 230, 233, 238, 239, 243, 252, 254,
                256, 259, 264, 265, 267, 273, 274, 276, 283, 286,
    292, 293, 300, 303, 305, 306, 308, 315, 318, 319]
    output = []
    for j in range(len(df_X)):
        temp1 = []
        temp2 = []

        for i in range(320):
            if i in indexes:
                temp1 += [df_X[j][i]]
                temp2 += [df_X[j][i + 320]]
            else:
                temp1 += [0]
                temp2 += [0]
        temp1 = np.asarray(temp1).astype(np.uint8)
        temp2 = np.asarray(temp2).astype(np.uint8)

        temp = np.concatenate((temp1, temp2))
        output.append(temp)

    output = np.array(output)
    return output

def neural_net(num_filters=32, num_outputs=1, d1=64, d2=64, ks=3,
    depth=5, reg_param=0.0001, final_activation='sigmoid'):
  #Input and preprocessing layers
  inp = Input(shape=(640,))
  rs = Reshape((10,64))(inp)
  perm = Permute((2, 1))(rs)

  #single residual layer(bit sliced)(block 1)
  conv0 = Conv1D(num_filters, kernel_size=1, padding='same',
    kernel_regularizer=l2(reg_param))(perm)
  conv0 = BatchNormalization()(conv0)
  conv0 = Activation('relu')(conv0)

  #add residual blocks(blocks 2-i)
  shortcut = conv0
  for i in range(depth):
    conv1 = Conv1D(num_filters, kernel_size=ks, padding='same',
    kernel_regularizer=l2(reg_param))(shortcut)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    conv2 = Conv1D(num_filters, kernel_size=ks, padding='same',
    kernel_regularizer=l2(reg_param))(conv1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    shortcut = Add()([shortcut, conv2])

  #add classification block
  flat1 = Flatten()(shortcut)
  dense1 = Dense(d1,kernel_regularizer=l2(reg_param))(flat1)
  dense1 = BatchNormalization()(dense1)
  dense1 = Activation('relu')(dense1)
  dense2 = Dense(d2, kernel_regularizer=l2(reg_param))(dense1)
```

```
  dense2 = BatchNormalization()(dense2)
  dense2 = Activation('relu')(dense2)
  out = Dense(num_outputs, activation=final_activation, kernel_regularizer=l2(reg_param))(dense2)
  model = Model(inputs=inp, outputs=out)
  return(model)

def cyclic_lr(n, beta, alpha): #learning rate function
  to_return = lambda i: alpha + (n - i % (n+1))/n * (beta - alpha)
  return(to_return)

folder = "./Models/rounds=" + str(4) + "_input_diff=" + str(hex(2**63+2**127)) + "/"  # folder link
#df_X = np.load(folder + "data_X_rounds=" + str(4) + "_input_diff=" + str(hex(2**63+2**127)) + ".npy", allow_pickle=True)
#df_X = onlyBias(df_X) #including only biased bits, with unbiased all set to 0
#np.save("Biased_data_X_rounds=" + str(4) + "_input_diff=" + str(hex(2**63+2**127)) + ".npy", df_X)
df_X = np.load(folder + "Biased_data_X_rounds=" + str(4) + "_input_diff=" + str(hex(2**63+2**127)) + ".npy", allow_pickle=True)
df_y = np.load(folder + "data_y_rounds=" + str(4) + "_input_diff=" + str(hex(2**63+2**127)) + ".npy",allow_pickle=True)

#Neural Network Parameters
num_epochs = 50
depth = 1
batch_size = 5000

#tuning of learning rate(minmax)
#start_lr = 1e-6
#end_lr = 1e0
#lr_finder = LRFinder(min_lr=start_lr, max_lr=end_lr)

opt = tf.keras.optimizers.Adam(learning_rate=0.01)

net = neural_net(depth=depth, reg_param=10**-5, ks=3, num_filters=32, d1 = 64, d2=64) #generate network
net.compile(optimizer='Adam',loss='mse',metrics=['acc'])

X_train, X_test, y_train, y_test = train_test_split(df_X, df_y, test_size=0.1, random_state=42)

#set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel_depth='+str(depth)+'.h5', monitor='val_loss', save_best_only = True)
#CSV_Logger
log_csv = CSVLogger(folder + 'log_depth='+str(depth)+'.csv', separator=',', append=True)
#cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9, 0.02, 0.0001))#0.002, 0.0001))
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr_finder])
fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
```

```
    batch_size,validation_data=(X_test, y_test), callbacks=[lr,
    checkpoint, log_csv])
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
    batch_size,validation_data=(X_test, y_test), callbacks=[
    checkpoint, log_csv])
#print(net.summary())
print("Best validation accuracy: ", np.max(fitted.history['val_acc'
    ]))

#Accuracy Plot
plt.plot(fitted.history['acc'])
plt.plot(fitted.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

#Loss Plot
plt.plot(fitted.history['loss'])
plt.plot(fitted.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

A.6 Transfer Learning

A.6.1 Neural Network with Basic Modifications

```
from Perm import perm
import secrets
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras.callbacks import ModelCheckpoint, LearningRateScheduler,
     CSVLogger
from keras.models import Model, load_model
from keras.layers import Dense, Conv1D, Input, Reshape, Permute,
    Add, Flatten, BatchNormalization, Activation
from keras.regularizers import l2
from keras import layers
from LRFinder.keras_callback import LRFinder
import math
import matplotlib.pyplot as plt

def gen_inputs(data_size,input_diff, rounds):
    data = []
    diff = []
    for i in range(data_size):
        x1 = secrets.randbits(320) #generate random input
        temp = secrets.randbits(1) #1 or 0 i.e fixed or random
    input diff
        if temp == 1: x2 = x1 ^ input_diff #generate second input
    with inputdiff
```

```
        elif temp == 0: x2 = secrets.randbits(320) #generate second input randomly
        entry1 = []
        entry2 = []
        y1 = perm(x1,rounds) #perm outputs
        y2 = perm(x2,rounds) #perm outputs

        for j in range(0,320): #converting yis to bitwise entries in string
            entry1.insert(0,((y1 >> j) & 1))
            entry2.insert(0,((y2 >> j) & 1))

        entry1 = np.asarray(entry1).astype(np.uint8)
        entry2 = np.asarray(entry2).astype(np.uint8)
        entry = np.concatenate((entry1, entry2)) #(320 bit output,320 bit output) i.e 640 bit list with bitwise entries
        if temp == 1: diff+=[1]
        elif temp == 0: diff+=[0] #if random input diff
        data.append(entry) #add entry to dataset

    data = np.array(data)
    diff = np.array(diff)

    return data, diff

def neural_net(num_filters=32, num_outputs=1, d1=64, d2=64, ks=3, depth=5, reg_param=0.0001, final_activation='sigmoid'):
  #Input and preprocessing layers
  inp = Input(shape=(640,))
  rs = Reshape((10,64))(inp)
  perm = Permute((2, 1))(rs)

  #single residual layer(bit sliced)(block 1)
  conv0 = Conv1D(num_filters, kernel_size=1, padding='same', kernel_regularizer=l2(reg_param))(perm)
  conv0 = BatchNormalization()(conv0)
  conv0 = Activation('relu')(conv0)

  #add residual blocks(blocks 2-i)
  shortcut = conv0
  for i in range(depth):
    conv1 = Conv1D(num_filters, kernel_size=ks, padding='same', kernel_regularizer=l2(reg_param))(shortcut)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    conv2 = Conv1D(num_filters, kernel_size=ks, padding='same', kernel_regularizer=l2(reg_param))(conv1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    shortcut = Add()([shortcut, conv2])

  #add classification block
  flat1 = Flatten()(shortcut)
  dense1 = Dense(d1,kernel_regularizer=l2(reg_param))(flat1)
  dense1 = BatchNormalization()(dense1)
  dense1 = Activation('relu')(dense1)
  dense2 = Dense(d2, kernel_regularizer=l2(reg_param))(dense1)
```

```
  dense2 = BatchNormalization()(dense2)
  dense2 = Activation('relu')(dense2)
  out = Dense(num_outputs, activation=final_activation, kernel_regularizer=l2(reg_param))(dense2)
  model = Model(inputs=inp, outputs=out)
  return(model)

def cyclic_lr(n, beta, alpha): #learning rate function
  to_return = lambda i: alpha + (n - i % (n+1))/n * (beta - alpha)
  return(to_return)

#Data Prep
differences = [2**63+2**127, 2**63, 2**319] #only state[3][0] = 1 & state[4][0] = 1, only state[4][0] = 1, some random state
round = [1,2,3,"3_5",4,5,6]
for input_diff in differences:
    for rounds in round:
        print("Rounds = "+str(rounds)+" Input diff = "+str(hex(input_diff)))
        df_X, df_y = gen_inputs(data_size=10 ** 6, input_diff=input_diff, rounds=rounds)
        np.save("data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_X)
        np.save("data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy", df_y)

rounds = round[4] #rounds of round function used
input_diff = differences[0] #input diff
folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/" #folder link
df_X1 = np.load(folder + "data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True)
df_y1 = np.load(folder + "data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True)
#folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/" + "testing_" #temp link to save test files
rounds = round[3] #rounds of round function used
input_diff = differences[0] #input diff
folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + "/" #folder link
df_X2 = np.load(folder + "data_X_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True)
df_y2 = np.load(folder + "data_y_rounds=" + str(rounds) + "_input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True)

#Neural Network Parameters
num_epochs = 20
depth = 1
batch_size = 5000

#tuning of learning rate(minmax)
#start_lr = 1e-6
```

```
#end_lr = 1e0
#lr_finder = LRFinder(min_lr=start_lr, max_lr=end_lr)

net = neural_net(depth=depth, reg_param=10**-5, ks=3, num_filters=32) #generate network
##net.compile(optimizer='Adam',loss='mse',metrics=['acc'])

X_train, X_test, y_train, y_test = train_test_split(df_X2, df_y2, test_size=0.1, random_state=42)

#set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel_depth='+str(depth)+'.h5', monitor='val_loss', save_best_only = True)
#CSV_Logger
log_csv = CSVLogger(folder + 'log_depth='+str(depth)+'.csv', separator=',', append=True)
#cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9, 0.02, 0.0001))#0.002, 0.0001))
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr_finder])
fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr, checkpoint, log_csv])

print("Best validation accuracy: ", np.max(fitted.history['val_acc']))
net.save("myModel.h5")
net = load_model("myModel.h5")

for i in range(13): #freezing first 13 layers
    net.layers[i].trainable = False

for i in range(13,20): #allows classification block to be trained (21 layers in total)
    net.layers[i].trainable = True

shortcut = net.layers[19].output #replaces classification layer
flat1 = Flatten()(shortcut)
dense1 = Dense(64,kernel_regularizer=l2(0.0001))(flat1)
dense1 = BatchNormalization()(dense1)
dense1 = Activation('relu')(dense1)
dense2 = Dense(64, kernel_regularizer=l2(0.0001))(dense1)
dense2 = BatchNormalization()(dense2)
dense2 = Activation('relu')(dense2)
out = Dense(1, activation='sigmoid', kernel_regularizer=l2(0.0001))(dense2)
net = Model(inputs=net.input, outputs=out)
net.compile(optimizer='Adam',loss='mse',metrics=['acc'])


num_epochs = 20
X_train, X_test, y_train, y_test = train_test_split(df_X1, df_y1, test_size=0.1, random_state=42)
#set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel_depth='+str(depth)
```

```
    +'.h5', monitor='val_loss', save_best_only = True)
#CSV_Logger
log_csv = CSVLogger(folder + 'log_depth='+str(depth)+'.csv',
    separator=',', append=True)
#cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9, 0.02, 0.0001))#0.002,
    0.0001))
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
    batch_size,validation_data=(X_test, y_test), callbacks=[
    lr_finder])
fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
    batch_size,validation_data=(X_test, y_test), callbacks=[lr,
    checkpoint, log_csv])
#print(net.layers)
#print(len(net.layers))
print("Best validation accuracy: ", np.max(fitted.history['val_acc'
    ]))

#Accuracy Plot
plt.plot(fitted.history['acc'])
plt.plot(fitted.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

#Loss Plot
plt.plot(fitted.history['loss'])
plt.plot(fitted.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

A.6.2 XOR Count to Invert Linear Diffusion layer

```
import numpy as np

m1 = np.zeros((64, 64),dtype = np.uint8) #expressing the lin diff
    of first 64 bits i.e. x0, y = mx where y is the output of the
    lindiff layer.
firstrot = -19
secondrot = -28
for i in range(64):
    m1[i][i] = 1
    m1[i][firstrot] = 1
    m1[i][secondrot] = 1

    firstrot += 1
    secondrot += 1

m2 = np.zeros((64, 64),dtype = np.uint8) #generate id matrix to
    concatenate to find inverse.
for i in range(64): m2[i][i] = 1
```

```python
m = np.concatenate((m1,m2),axis=1)

#Binary Gaussian Elimination
def GJElim(m):
    a,b = m.shape
    i=0
    j=0

    while True:
        k = np.argmax(m[i:, j]) + i #to have the largest element in
 the remaining portion of col j, and get index

        temp = np.copy(m[k]) #row swap
        m[k] = m[i]
        m[i] = temp
        temp2 = m[i, j:]

        col = np.copy(m[:, j])

        col[i] = 0 #to prevent pivot from XORing itself

        flipped = np.outer(col, temp2) #computing tensor prod.

        m[:, j:] = m[:, j:] ^ flipped
        j = j + 1
        i = i + 1

        if (i >= a) or (j >= b): break

    return m

m = GJElim(m)
m = np.hsplit(m,2)[1]
print(m)
#Checking that it works
#folder = "./Models/InvXor/"  # folder link
#df_X = np.load(folder+"InvXor_data_Xs.npy",allow_pickle=True)
#df_y = np.load(folder+"InvXor_data_ys.npy",allow_pickle=True)
#for i in range(10000):
#    for j in range(64):
#        temp = np.matmul(m, df_X[i])%2
#        if temp[j] != df_y[i][j]: print("Error")


for i in range(64): print(m[i]) #printing rows of matrix

for j in range(64): #counting number of nonzero entries in each row
    . i.e. number of XORs
    count = 0
    for i in range(len(m[j])):
        if m[j][i] == 1:
            count += 1
    print(count)
```

A.6.3 Inverse XOR Transfer Learning

```python
import math
import secrets
import keras
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras.backend import round
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, CSVLogger
from keras.models import Model, load_model
from keras.layers import Dense, Conv1D, Input, Reshape, Permute, Add, Flatten, BatchNormalization, Activation
from keras.regularizers import l2
import matplotlib.pyplot as plt
from scipy.spatial.distance import hamming
from LRFinder.keras_callback import LRFinder

def split(s): #split s into 5 64 bit blocks
    y = []
    for i in range(5):
        y.insert(0,((s >> (i*64)) & 0xFFFFFFFFFFFFFFFF)) #shift s by a mult. of 64 then AND with 64 1s.
    return y

def cyclic_lr(n, beta, alpha): #learning rate function
  to_return = lambda i: alpha + (n - i % (n+1))/n * (beta - alpha)
  return(to_return)

def merge(s): #merge 5 64 bit blocks into 1
    y=0
    for i in range(5):
        y = y ^ (s[4-i] << (i * 64)) #starting from last block of s, shift left by mult of 64 then XOR iteratively
    return y

def lindiff(y): #y here is a list containing xis
    def circ_shift(s,r):
        temp = 0
        for i in range(r): #create 11111s
            temp += 2**(i)
        return ((s>>r)^((s & temp) << (64-r)))
    temp = y
    y[0] = temp[0] ^ circ_shift(temp[0],19) ^ circ_shift(temp[0],28)
    y[1] = temp[1] ^ circ_shift(temp[1],61) ^ circ_shift(temp[1],39)
    y[2] = temp[2] ^ circ_shift(temp[2],1) ^ circ_shift(temp[2],6)
    y[3] = temp[3] ^ circ_shift(temp[3],10) ^ circ_shift(temp[3],17)
    y[4] = temp[4] ^ circ_shift(temp[4],7) ^ circ_shift(temp[4],41)

    return y

def applyXOR(s):
    s = split(s)
    s = lindiff(s)
    return merge(s)
```

```python

def gen_inputs(data_size,temp):
    input = []
    output = []
    for i in range(data_size):
        x = secrets.randbits(320) #generate random input
        y = applyXOR(x) #perm outputs
        entry1 = []
        entry2 = []

        for j in range(0,320): #converting yis to bitwise entries in string
            entry1.insert(0,((x >> j) & 1))
            entry2.insert(0,((y >> j) & 1))

        entry1 = entry1[0:64]
        entry2 = entry2[0:64]
        entry1 = np.asarray(entry1).astype(np.uint8)
        entry2 = np.asarray(entry2).astype(np.uint8)

        input.append(entry1) #add entry to dataset
        output.append(entry2)

    input = np.array(input)
    output = np.array(output)

    return input, output

def neural_net():
  #Input and preprocessing layers
  inp = Input(shape=(64,))
  rs = Reshape((64,1))(inp)
  perm = Permute((2, 1))(rs)

  dense1 = Dense(512)(perm)
  dense1 = BatchNormalization()(dense1)
  dense1 = Activation('relu')(dense1)

  dense2 = Dense(1024)(dense1)
  dense2 = BatchNormalization()(dense2)
  dense2 = Activation('relu')(dense2)

  dense3 = Dense(2048)(dense2)
  dense3 = BatchNormalization()(dense3)
  dense3 = Activation('relu')(dense3)

  dense4 = Dense(1024)(dense3)
  dense4 = BatchNormalization()(dense4)
  dense4 = Activation('relu')(dense4)

  dense5 = Dense(512)(dense4)
  dense5 = BatchNormalization()(dense5)
  dense5 = Activation('relu')(dense5)

  dense6 = Dense(64)(dense5)
  dense6 = BatchNormalization()(dense6)
  out = Activation('relu')(dense6)
```

```
  out = Permute((2, 1))(out)
  out = Reshape((64,))(out)

  model = Model(inputs=inp, outputs=out)
  return(model)

folder = "./Models/InvXor/"  # folder link

#Data prep
print("Generating inputs")
#df_y, df_X = gen_inputs(data_size=10 ** 6)
#np.save(folder+"InvXor_data_X.npy", df_X)
#np.save(folder+"InvXor_data_y.npy", df_y)
df_X = np.load(folder+"InvXor_data_Xs.npy",allow_pickle=True)
df_y = np.load(folder+"InvXor_data_ys.npy",allow_pickle=True)

#Neural Network Parameters
num_epochs = 50
batch_size = 500

net = neural_net()#generate network
opt = tf.keras.optimizers.Adam(learning_rate=0.1)
net.compile(optimizer=opt,loss='mse',metrics=['mse'])
X_train, X_test, y_train, y_test = train_test_split(df_X, df_y,
    test_size=0.1, random_state=42)

#set up model checkpoint
checkpoint = ModelCheckpoint(folder + 'bestmodel'+'.h5', monitor='
    val_loss', save_best_only = True)
#CSV_Logger
log_csv = CSVLogger(folder + 'log.csv', separator=',', append=True)
#fitted = net.fit(X_train,y_train,epochs=num_epochs,batch_size=
    batch_size,validation_data=(X_test, y_test), callbacks=[
    checkpoint, log_csv])
##net.save(folder+"InvXorModel.h5")

net = load_model(folder+"InvXorModel.h5")

#Computing hamming distance
#pred = round(net.predict(X_test))
#count = 0
#for i in range(100000):
#    count += hamming(pred[i],y_test[i]) * 64
#print(count)
#print(count/(100000))

#Concatenation of Network for Transfer Learning
net2 = load_model("myModel.h5")
inp = Input(shape=(640,)) #Adjusting inputs of inverse XOR network
x = Reshape((1,640))(inp)
temp = 640
for i in range(1,4):
    temp = temp*2
    x = Dense(temp)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
#x = Dense(round(temp*1.5))(x)
```

```
#x = BatchNormalization()(x)
#x = Activation('relu')(x)
for i in range(1,4):
    temp = temp/2
    x = Dense(temp)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
#for layer in net.layers[3:-2]: x = layer(x)
#x = Dense(640,kernel_regularizer=l2(0.0001))(x)
#x = BatchNormalization()(x)
#x = Activation('relu')(x)
model = Model(inputs=inp, outputs=x)


x = model.layers[-1].output

count = 0
shortcut = 0
for layer in net2.layers[1:]: #joining the two networks
    layer.trainable = False
    layer._name = layer.name + str("_new")
    count+=1
    if count == 5:
        x = layer(x)
        shortcut = x
    elif count == 12: x = layer([shortcut,x])
    elif count == 20:
        layer.trainable = True
        x = layer(x)
    else: x = layer(x)
model = Model(inputs=model.input, outputs=x)
print(model.summary())
#opt = tf.keras.optimizers.Adam(learning_rate=0.1)
model.compile(optimizer='Adam',loss='mse',metrics=['acc'])

#Data Prep
differences = [2**63+2**127, 2**63, 2**319] #only state[3][0] = 1 &
     state[4][0] = 1, only state[4][0] = 1, some random state
round = [1,2,3,"3_5",4,5,6]
rounds = round[4] #rounds of round function used
input_diff = differences[0] #input diff
folder = "./Models/rounds=" + str(rounds) + "_input_diff=" + str(
    hex(input_diff)) + "/" #folder link
df_X1 = np.load(folder + "data_X_rounds=" + str(rounds) + "
    _input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True
    )
df_y1 = np.load(folder + "data_y_rounds=" + str(rounds) + "
    _input_diff=" + str(hex(input_diff)) + ".npy",allow_pickle=True
    )
X_train, X_test, y_train, y_test = train_test_split(df_X1, df_y1,
    test_size=0.1, random_state=42)

num_epochs = 50
depth = 1
batch_size = 5000
#CSV_Logger
log_csv = CSVLogger(folder + 'log_depth='+str(depth)+'.csv',
```

```
    separator=',', append=True)
#cyclic learnrate scheduler
lr = LearningRateScheduler(cyclic_lr(9,0.05, 0.01))#0.002, 0.0001))

#tuning of learning rate(minmax)
start_lr = 1e-6
end_lr = 1e0
lr_finder = LRFinder(min_lr=start_lr, max_lr=end_lr)
#fitted = model.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr_finder])

fitted = model.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[lr,log_csv])
#fitted = model.fit(X_train,y_train,epochs=num_epochs,batch_size=batch_size,validation_data=(X_test, y_test), callbacks=[log_csv])

#Loss Plot
plt.plot(fitted.history['loss'])
plt.plot(fitted.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

#Accuracy Plot
plt.plot(fitted.history['acc'])
plt.plot(fitted.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```