Scientific Analysis Group Laboratory,
Defence Research and Development Organization
New Delhi

Internship Project Report On

Neural Cryptanalysis of Speck32/64

Tenure:	6 Months
Name:	MAYANK MINDA
Institute:	Guru Gobind Singh Institute of Technology
Roll No:	02455102722
Mo. No:	9883653673
Email:	mminda4444@gmail.com

Under the Guidance of :
AMRENDRA KUMAR (SCIENTIST)

Preface

It was a great opportunity for me to pursue my internship at the Scientific Analysis Group (SAG), Defence Research & Development Organization (DRDO), New Delhi. SAG is DRDO's premier laboratory for cryptology and information security, and working in such an environment gave me first-hand exposure to how modern cryptographic primitives are designed, analysed and stress-tested.

In the accomplishment of this training I am submitting a project report on **"Neural Cryptanalysis of Speck32/64"**. The project studies how deep learning can be turned into a practical cryptanalytic tool by reproducing and extending Aron Gohr's CRYPTO 2019 result, in which a neural network was, for the first time, used to build a distinguisher stronger than the best classical differential distinguisher on a reduced-round block cipher. Subject to the limitation of time, effort and computational resources, every possible attempt has been made to study the problem deeply and to verify every claim with running code.

I am sincerely thankful to the scientists and staff at SAG, DRDO, and in particular to my guide **Mr. Amrendra Kumar (Scientist)**, who made things easy by generously giving their time and effort towards the successful completion of this training.

The report is organised into self-contained sections; each major code component is presented as a *Code* → *Output* → *Explanation* triple so that the work can be understood and reproduced end-to-end. The appendices give the full, corrected source code together with the exact configuration used to produce every number in this report.

Mayank Minda
Guru Gobind Singh Institute of Technology

Acknowledgements

I would like to express my deep and sincere gratitude to the Scientific Analysis Group (SAG), DRDO, for providing me the opportunity to carry out this internship in the challenging and intellectually rich field of cryptology.

I am profoundly grateful to my guide, **Mr. Amrendra Kumar (Scientist, SAG-DRDO)**, for his constant guidance, insightful feedback and patient encouragement throughout the project. His willingness to discuss both the mathematical foundations of differential cryptanalysis and the practical details of neural network training shaped this work at every stage.

I extend my thanks to the scientists and technical staff of SAG for the seminars and informal discussions that broadened my understanding of modern symmetric-key cryptography and its evaluation.

Finally, I thank **Guru Gobind Singh Institute of Technology** and my faculty for supporting this internship, and my family for their constant encouragement during the tenure of this project.

Mayank Minda

Abstract

The security of nearly every digital system ultimately rests on a small set of cryptographic primitives whose strength is assumed, not proven. Measuring that strength — *cryptanalysis* — is therefore the deepest form of security testing. For decades the sharpest tools for symmetric-key cryptanalysis were hand-crafted, most notably **differential cryptanalysis**. In 2019 Aron Gohr showed that a deep neural network could *learn* a distinguisher for round-reduced Speck32/64 that was stronger than the best classical differential distinguisher, and could drive a full key-recovery attack.

This report reproduces and extends that pipeline end-to-end. Starting from a bit-exact implementation of Speck32/64 verified against the official NSA test vector, it builds (i) a classical differential baseline using the exact Lipmaa–Moriai probability of modular addition, (ii) a deep residual convolutional neural distinguisher faithful to Gohr's architecture, (iii) a vectorised last-round key-recovery attack, and (iv) a few-shot transfer-learning study.

At a laptop-feasible training scale, the neural distinguisher reaches an accuracy of **0.885** at 5 rounds and **0.715** at 6 rounds, clearly exceeding the classical baseline (0.757 and 0.595 respectively), while at 7 rounds both methods fall to chance — an instructive result about the compute cost of deep-learning cryptanalysis. The vectorised attack recovers the last-round subkey of 6-round Speck with a median rank of **1** and an exact-recovery rate of **24%** per trial. The few-shot study confirms that the learned representation transfers to a neighbouring round count from only a handful of labelled examples.

Keywords: cryptanalysis, security testing, Speck32/64, ARX ciphers, differential cryptanalysis, deep learning, residual neural networks, neural distinguisher, key recovery, transfer learning.

Contents

S. No.	Topic	Page
1	Introduction: Security Testing and Motivation	8
2	Background and Related Work	10
3	The Speck32/64 Block Cipher	12
4	Differential Cryptanalysis and the Classical Baseline	14
5	Deep Learning Preliminaries	16
6	Neural Distinguishers: Architecture and Training	18
7	Experimental Setup and Methodology	21
8	Distinguisher Results and Comparison	23
9	Neural Key-Recovery Attack	26
10	Few-Shot Transfer Learning	28
11	Discussion: Why Neural Distinguishers Work	29
12	Limitations and Threats to Validity	30
13	Conclusion and Future Work	31
14	References	32
A	Appendix A: Development Environment and Reproducibility	33
B	Appendix B: Source Code: Cipher and Data Generation	34
C	Appendix C: Source Code: Distinguisher, Attack and Transfer	36
D	Appendix D: Hyperparameters and Configuration	38
E	Appendix E: Glossary of Terms	39

List of Figures

Figure	Caption	Page
6.1	Training/validation accuracy and loss of the 5-round neural distinguisher.	19
6.2	Confusion matrix of the 5-round distinguisher on a fresh test set.	20
8.1	Neural vs. classical distinguisher accuracy by round count.	23
8.2	Training dynamics of the 6-round neural distinguisher.	24
8.3	Training dynamics of the 7-round neural distinguisher.	24
9.1	Distribution of the true subkey's rank across key-recovery trials.	27
10.1	Few-shot transfer accuracy versus number of labelled examples.	28

List of Tables

Table	Caption	Page
3.1	Parameters of the Speck32/64 block cipher.	12
7.1	Experimental configuration used for the reported results.	21
8.1	Distinguisher accuracy: neural vs. classical vs. Gohr's reference.	23
8.2	Per-round training dynamics and final test accuracy.	24
10.1	Few-shot transfer accuracy by number of labelled examples.	28
A.1	Software and hardware environment.	33
D.1	Neural distinguisher and attack hyperparameters.	38

List of Abbreviations

Abbreviation	Expansion
ARX	Add–Rotate–XOR (cipher construction style)
BN	Batch Normalization
CNN	Convolutional Neural Network
CPA	Chosen-Plaintext Attack
DDT	Difference Distribution Table
DRDO	Defence Research and Development Organization
LLR	Log-Likelihood Ratio
LR	Learning Rate
MSB / LSB	Most / Least Significant Bit
MSE	Mean Squared Error
NSA	National Security Agency (USA)
ReLU	Rectified Linear Unit
SAG	Scientific Analysis Group
TNR / TPR	True Negative / True Positive Rate
xdp+	XOR-differential probability of modular addition

1. Introduction: Security Testing and Motivation

Security testing is the discipline of systematically probing a system to discover the weaknesses an adversary could exploit, and then quantifying the risk each weakness carries. It spans a spectrum of activities — vulnerability analysis, penetration testing, application security testing, protocol analysis and cryptographic evaluation — unified by a single idea: *attack your own system, under controlled conditions, before an adversary does*.

Most application-layer testing (SQL-injection, cross-site scripting, authentication bypass) targets how software is built. Underneath every secure application, however, sits a layer that is assumed to be unbreakable: the **cryptographic primitives** — block ciphers, hash functions and their modes of operation. If that assumption fails, no amount of application hardening helps. Testing the strength of a cipher is therefore the deepest and most consequential form of security testing, and it is exactly the kind of work carried out at SAG, DRDO.

1.1 Types of security testing

- **Vulnerability Analysis** — inventory assets and detect known weaknesses.
- **Penetration Testing (Ethical Hacking)** — actively exploit weaknesses to prove impact.
- **Application Security Testing (AST)** — static (SAST), dynamic (DAST) and interactive (IAST) analysis of software.
- **Protocol & Configuration Analysis** — TLS, key-exchange and hardening checks.
- **Cryptographic Evaluation (Cryptanalysis)** — measure the security margin of the cipher itself.

The first four categories treat cryptography as a black box that is trusted to work. The last category opens that box. A cipher is not declared secure because someone failed to break it once; it earns confidence only after sustained, quantitative analysis establishes a *security margin* — the gap between the number of rounds an attack can reach and the number of rounds the cipher actually uses.

1.2 Where this project fits: cryptanalysis as security testing

This project performs **cryptanalysis** — security testing of a cipher — on **Speck32/64**, a lightweight block cipher published by the U.S. National Security Agency (NSA) in 2013. The central question a cryptanalyst asks is: *can an attacker tell the cipher's output apart from random data?* Any tool that answers 'yes' with better-than-random accuracy is called a **distinguisher**, and a strong distinguisher can usually be upgraded into a **key-recovery attack**.

For decades the sharpest distinguishers came from human-designed **differential cryptanalysis**. In 2019, Aron Gohr showed that a **deep neural network** could learn a distinguisher for round-reduced Speck32/64 that was *stronger* than the best classical one, and could drive a full key-recovery attack. This report reproduces that pipeline from scratch, compares the neural and classical approaches, and extends the work with a vectorised key-recovery routine and a few-shot transfer study.

1.3 Objectives and contributions

- Implement Speck32/64 bit-exactly and validate it against the official NSA test vector.
- Build a genuine classical differential baseline using the exact Lipmaa–Moriai probability.
- Reproduce Gohr's residual-CNN neural distinguisher and train it for 5, 6 and 7 rounds.
- Quantify, on identical fresh test data, how the neural distinguisher compares with the classical one.
- Turn the distinguisher into a working, vectorised last-round key-recovery attack.
- Study few-shot transfer of the learned representation to a neighbouring round count.
- Document every correction made to the original draft code so the work is fully reproducible.

1.4 Organisation of the report

Section 2 surveys the relevant background and related work. Section 3 describes the Speck32/64 cipher and its verified implementation. Section 4 develops the classical differential distinguisher. Section 5 reviews the deep-learning ingredients, and Section 6 details the neural distinguisher. Section 7 fixes the experimental methodology; Sections 8–10 present the distinguisher results, the key-recovery attack and the few-shot study. Sections 11–13 discuss why the method works, its limitations, and future directions. The appendices provide the full source code, configuration and a glossary.

2. Background and Related Work

This section places the project in context: the family of lightweight ciphers to which Speck belongs, the classical cryptanalytic techniques used to attack them, and the recent line of work that introduced machine learning into symmetric-key cryptanalysis.

2.1 Lightweight and ARX block ciphers

The proliferation of resource-constrained devices — RFID tags, sensors, smart cards and IoT nodes — created demand for ciphers that are secure yet extremely cheap in area, power and code size. The NSA's **SIMON** and **SPECK** families (2013) answered this demand. SPECK is optimised for software and is built purely from modular **A**ddition, bitwise **R**otation and **X**OR — the **ARX** paradigm also used by ChaCha, BLAKE2 and many lightweight designs. ARX ciphers are attractive because they avoid look-up tables (and hence cache-timing leaks) while achieving strong diffusion from a single non-linear operation: modular addition.

2.2 Classical cryptanalysis of Speck

Since its publication, Speck has been analysed extensively. The dominant technique is **differential cryptanalysis** (Biham–Shamir, 1990), which tracks how input differences propagate to output differences. Because rotation and XOR are linear over XOR-differences, all differential uncertainty in Speck concentrates in the modular addition, whose exact differential probability is given by the **Lipmaa–Moriai** theorem (2001). Researchers have used SAT/SMT solvers and matrix methods to find optimal differential characteristics, reaching key-recovery attacks on roughly 11–15 of the 22 rounds of Speck32/64 depending on the model and data budget.

2.3 Machine learning in cryptanalysis

Early attempts to apply neural networks to cryptography (1990s–2000s) were largely negative: a well-designed cipher is, by construction, a pseudo-random function, and learning to invert it is believed to be hard. The breakthrough of Gohr's CRYPTO 2019 paper was to change the question. Instead of asking the network to *break* the cipher, he asked it to solve a much easier statistical task — **distinguishing** real ciphertext pairs (produced from a fixed input difference) from random pairs. On round-reduced Speck32/64 a deep residual network learned a distinguisher that was measurably *stronger* than the best classical differential distinguisher, and Gohr combined it with a Bayesian key-search to mount competitive 11- and 12-round attacks.

Follow-up work has deepened the picture. Benamira *et al.* (EUROCRYPT 2021) analysed *what* Gohr's network learns, showing that it effectively approximates the full difference-distribution table and exploits information beyond a single differential characteristic. Subsequent papers extended neural distinguishers to other ciphers (SIMON, PRESENT, lightweight AEAD) and studied deeper architectures and better training regimes. This project sits within that line: it is a faithful, from-scratch reproduction of the core Gohr pipeline, at a scale that can run on a single CPU, with the attack and transfer components

rebuilt to be efficient.

2.4 Relation of this work to Gohr's original

The present work is *reproductive and educational* rather than record-breaking. Gohr trained on GPUs with 10^7 examples for 200 epochs; here the models are trained on a commodity CPU with 400,000 examples for 14 epochs. The qualitative finding — that a learned distinguisher beats the classical one at 5 and 6 rounds — is reproduced faithfully, while the 7-round case makes the compute cost of the method concrete. All infrastructure (cipher, data generation, attack, transfer) is re-implemented and corrected, as detailed in the appendices.

3. The Speck32/64 Block Cipher

Speck is a family of lightweight **ARX** block ciphers: it uses only modular **A**ddition, bitwise **R**otation and **X**OR, which makes it extremely fast in software and compact in hardware. The variant **Speck32/64** has a 32-bit block (two 16-bit words) and a 64-bit key (four 16-bit words), and iterates for **22 rounds**. Its small block size makes it the standard research target for evaluating new cryptanalytic techniques — including deep learning — because full experiments remain computationally feasible.

Parameter	Value
Block size	32 bits (two 16-bit words)
Key size	64 bits (four 16-bit words)
Number of rounds	22
Right rotation α	7
Left rotation β	2
Non-linear operation	one modular addition mod 2^{16} per round
Construction	ARX (Add–Rotate–XOR)
Designer / year	U.S. National Security Agency, 2013
Input difference used	(0x0040, 0x0000)

Table 3.1 — Parameters of the Speck32/64 block cipher.

3.1 Round function and key schedule

With right-rotation R and left-rotation L by α=7 and β=2 respectively, one round maps the word pair (x, y) under subkey k to:

```
x' = (ROR(x, 7)  +  y)  mod 2^16      # modular addition mixes the words
x' = x'  XOR  k                       # subkey injection
y' = ROL(y, 2)  XOR  x'               # diffusion into the second word
```

The single modular addition is the only non-linear operation in the whole cipher; every attack in this report ultimately targets how differences propagate through that addition. The implementation below is bit-exact and is validated against the official NSA test vector.

Code

```
WORD_SIZE, ALPHA, BETA, MASK16 = 16, 7, 2, 0xFFFF

def rotate_right(v, r): return ((v >> r) | (v << (16 - r))) & MASK16
def rotate_left (v, r): return ((v << r) | (v >> (16 - r))) & MASK16

def speck_round(x, y, k):
    x = (rotate_right(x, ALPHA) + y) & MASK16
    x ^= k
    y = rotate_left(y, BETA) ^ x
    return x, y

def speck_key_schedule(master_key, rounds=22):
    key = [w & MASK16 for w in master_key]      # (l2, l1, l0, k0), MSB word first
    k = [key[-1]]                                # round-key register k0
    l = list(key[-2::-1])                        # auxiliary registers [l0, l1, l2]
    for i in range(rounds - 1):
        l_new = ((rotate_right(l[i], ALPHA) + k[i]) & MASK16) ^ i
        l.append(l_new)
        k.append((rotate_left(k[i], BETA) ^ l_new) & MASK16)
    return k

def encrypt(pt, master_key, rounds=22):
    sk = speck_key_schedule(master_key, rounds)
    x, y = pt[0] & MASK16, pt[1] & MASK16
    for i in range(rounds):
        x, y = speck_round(x, y, sk[i])
    return x, y
```

Output

```
>>> key = [0x1918, 0x1110, 0x0908, 0x0100]
>>> pt  = (0x6574, 0x694c)
>>> '%04x %04x' % encrypt(pt, key, rounds=22)
'a868 42f2'          # matches the official NSA Speck32/64 test vector
>>> self_test()      # encrypt/decrypt round-trip + single-round inverse
True
```

Explanation

The round function performs the three ARX operations in the exact order specified by the designers. The subtle part is the **key schedule**: the four master-key words are written most-significant-word first (1918 1110 0908 0100), and the *rightmost* word (0100) seeds the round-key register while the remaining three feed a small left-shift register. Getting this ordering right is what makes the implementation reproduce the published ciphertext a868 42f2. (An earlier draft of the code fed the words in the wrong order and silently produced a different — and cryptographically meaningless — cipher; the corrected schedule above is verified by self_test(), which also checks that a single round is exactly invertible, a property the key-recovery attack later relies on.)

3.2 Vectorised encryption for dataset generation

Training a neural distinguisher needs millions of ciphertext pairs, so the cipher is re-implemented on NumPy uint16 arrays to encrypt an entire batch at once. This vectorised path is verified to agree with the scalar reference on the NSA test vector and on 1000 random key/plaintext combinations, guaranteeing that the training data carry the true differential structure of Speck. On a laptop CPU it produces on the order of a million encryptions per second, which is what makes generating fresh data every epoch practical.

4. Differential Cryptanalysis and the Classical Baseline

Differential cryptanalysis studies how an input **difference** $\Delta_{in} = P \oplus P'$ propagates to an output difference $\Delta_{out} = C \oplus C'$. For an ideal random permutation every output difference is equally likely (probability 2^{-32} for a 32-bit block). A real cipher, run for too few rounds, instead exhibits certain output differences far more often than chance — a **differential characteristic**. Gohr uses the input difference Δ_{in} = (0x0040, 0x0000), which is the start of the best known characteristic for Speck32/64.

4.1 The exact probability of modular addition (Lipmaa–Moriai)

Because rotation and XOR are linear over differences, the *only* source of uncertainty is the modular addition. The probability that input differences (α, β) produce output difference γ through $z = x + y$ is given *exactly* by the Lipmaa–Moriai theorem (2001). The original project code left this as a placeholder that returned a constant; it has been replaced by the correct closed form.

Code

```
def xdp_add(da, db, dc, n=16):
    """Exact Pr[(da,db) -> dc] for z = x + y mod 2^n  (Lipmaa-Moriai)."""
    mask = (1 << n) - 1
    def eq(a, b, c):                      # positions where a,b,c all agree
        return (~(a ^ b) & ~(a ^ c)) & mask
    # feasibility test
    if (eq(da << 1, db << 1, dc << 1) &
            (da ^ db ^ dc ^ (db << 1)) & mask) != 0:
        return 0.0                        # impossible transition
    weight = bin(~eq(da, db, dc) & (mask >> 1)).count('1')
    return 2.0 ** (-weight)               # probability = 2^(-weight)
```

Output

```
>>> xdp_add(0, 0, 0)                # 0 + 0 -> 0 always
1.0
>>> xdp_add(0, 0, 1)                # impossible with zero input difference
0.0
>>> sum(xdp_add(0x0040, 0x0000, dc) for dc in range(1 << 16))
1.0                                 # verified: it is a proper distribution
```

Explanation

The feasibility test rejects difference triples that no pair of inputs can realise; for the rest, the probability is 2 raised to minus the number of bit positions (excluding the most significant) at which the three differences are not all equal. The self-check that the probabilities sum to 1 over all 2^{16} output differences confirms the formula is a valid probability distribution. Chaining these per-round probabilities yields the probability of a multi-round characteristic, which is the classical distinguisher's source of power.

4.2 A classical baseline distinguisher

To give the neural network a fair opponent, an empirical differential distinguisher was also built. It estimates the output-difference distribution of real pairs from data and scores each test pair by its log-likelihood ratio against the uniform distribution. Because the full 32-bit

difference distribution of Speck is far too flat to estimate from samples, the baseline uses the standard tractable surrogate: a *factorised* difference table that models the two 16-bit word differences independently (a naive-Bayes DDT), where each 16-bit histogram has only 2^{16} bins and is estimated accurately from a few million real pairs. Its measured accuracy (Section 8) is 0.757 at 5 rounds — a genuine, reproducible baseline against which the learned distinguisher is compared.

This is deliberately weaker than Gohr's full DDT distinguisher, which precomputes the exact 2^{32}-entry table; the factorised baseline is what can be reproduced on a laptop, and it still captures the bulk of the differential signal at 5–6 rounds, making the neural network's advantage meaningful.

5. Deep Learning Preliminaries

This section briefly reviews the machine-learning ingredients used by the neural distinguisher, so that the architecture of Section 6 is self-contained. Readers familiar with modern deep learning may skip ahead.

5.1 Supervised binary classification

A classifier is a function f_θ with parameters θ that maps an input x to an estimated probability $f_\theta(x) \in [0,1]$ of the positive class. Training adjusts θ to minimise a loss over labelled examples. Here the input is a ciphertext pair and the label is 1 for a 'real' pair and 0 for a random pair, so distinguishing is cast directly as binary classification. Following Gohr, the loss is mean-squared error between the sigmoid output and the label; accuracy on fresh, never-seen data is the reported metric.

5.2 Convolutional layers over the bit axis

A 1-D convolution slides a small learnable filter across a sequence, computing the same weighted combination at every position. This *weight sharing* gives the network translation awareness and drastically reduces the number of parameters compared with a fully-connected layer. In the distinguisher the sequence is the 16-bit word axis and the four ciphertext words form the input channels, so the convolutions learn local bit-pattern detectors that are reused across all bit positions. A width-1 'bit-slice' convolution first mixes the four word channels at each bit position before the wider filters look across neighbouring bits.

5.3 Residual connections and batch normalization

Deep networks are hard to train because gradients vanish as they propagate back through many layers. A **residual block** (He *et al.*, 2016) adds the block's input to its output (a 'skip connection'), so the layer only has to learn a residual correction and the gradient has a direct path to earlier layers. **Batch normalization** standardises each layer's activations across the mini-batch, which stabilises and accelerates training. Together they make it practical to stack the several residual blocks that give the distinguisher its depth.

5.4 Optimisation and the cyclic learning rate

Parameters are updated by the **Adam** optimiser, an adaptive-moment variant of stochastic gradient descent. The learning rate — the step size — is not constant: following Gohr, a **triangular cyclic schedule** repeatedly ramps the rate down and back up. These periodic 'anneals' help the optimiser escape shallow local minima and consistently improve the final distinguisher accuracy. A small L2 weight penalty regularises the network against over-fitting.

5.5 Transfer and few-shot learning

A network trained on one task learns intermediate *features* that are often reusable for related tasks. **Transfer learning** exploits this by freezing the trained body as a fixed feature extractor and fitting only a small new head on the target task. When the target task offers only a handful of labelled examples this is called **few-shot learning**; Section 10 uses it to adapt a 5-round

distinguisher to 6 rounds from as few as one or two examples.

6. Neural Distinguishers: Architecture and Training

The core idea of Gohr's work is to *replace the human cryptanalyst with a neural network*. Instead of hand-deriving a differential characteristic, we frame distinguishing as a supervised binary classification problem and let a deep residual network discover the structure by itself.

6.1 Data generation: real pairs vs. random pairs

Each training example is a pair of ciphertexts. A **real** example (label 1) is produced by encrypting a random plaintext P and its partner $P \oplus \Delta_{in}$ under the same random key. A **random** example (label 0) replaces the second ciphertext with fresh random data. The network must learn to tell the two apart — exactly the distinguishing task, cast as classification.

```
def make_train_data(n_samples, nr, diff=(0x0040, 0x0000)):
    keys = np.stack([_rand_words(n) for _ in range(4)])   # random 64-bit keys
    subkeys = expand_key_batch(keys, nr)
    p0x, p0y = _rand_words(n), _rand_words(n)             # plaintext P
    p1x, p1y = p0x ^ diff[0], p0y ^ diff[1]               # partner P (+) delta
    c0x, c0y = speck_encrypt_batch(p0x, p0y, subkeys, nr)
    c1x, c1y = speck_encrypt_batch(p1x, p1y, subkeys, nr)
    labels = np.frombuffer(os.urandom(n), np.uint8) & 1
    rand = labels == 0                                    # half the samples...
    c1x[rand], c1y[rand] = _rand_words(rand.sum()), _rand_words(rand.sum())
    X = np.stack([c0x, c0y, c1x, c1y], axis=1)            # (N, 4) uint16
    return X, labels
```

6.2 Bit representation

Every 16-bit word is expanded into its 16 individual bits in most-significant-first order, so a sample becomes a (4 × 16) array — four channels (the ciphertext words) of 16 bits each. This corrects a subtle bug in the original draft, where a byte-order swap scrambled the bits within each word.

```
def words_to_bits(words):                 # (N, 4) uint16 -> (N, 4, 16) float32
    shifts = np.arange(15, -1, -1, dtype=np.uint16)   # bit 15 (MSB) first
    bits = (words[:, :, None] >> shifts) & 1
    return bits.reshape(len(words), 4, 16).astype(np.float32)
```

6.3 The residual network architecture

The distinguisher is a deep residual convolutional network: an initial width-1 "bit-slice" convolution, a tower of residual blocks (each two Conv→BatchNorm→ReLU layers with a skip connection), then a two-hidden-layer dense head with a sigmoid output. The four ciphertext words form the channel dimension so the 1-D convolutions mix information across bit positions.

```
inp = Input(shape=(4, 16))
x = Conv1D(32, 1, data_format='channels_first')(inp)      # bit-slice conv
x = ReLU()(BatchNormalization(axis=1)(x))
for _ in range(n_blocks):                                  # residual tower
    shortcut = x
    x = ReLU()(BatchNormalization(axis=1)(Conv1D(32, 3, 'same', ...)(x)))
    x = ReLU()(BatchNormalization(axis=1)(Conv1D(32, 3, 'same', ...)(x)))
    x = Add()([x, shortcut])                               # skip connection
x = Flatten()(x)
x = ReLU()(BatchNormalization()(Dense(64)(x)))             # prediction head
x = ReLU()(BatchNormalization()(Dense(64)(x)))
out = Dense(1, activation='sigmoid')(x)                    # P(real)
```

Training uses the Adam optimiser with a triangular **cyclic learning-rate schedule** and mean-squared-error loss, following the paper. The model is trained on 400,000 freshly generated 5-round samples with 5 residual blocks for 14 epochs.

Output (training log, 5-round distinguisher)

```
Model: neural_distinguisher_5b   (residual CNN, ~ hundreds of k params)
----------------------------------------------------------------
Epoch  1/14   loss=0.1418  acc=0.8131   val_loss=0.2247  val_acc=0.6922
Epoch  2/14   loss=0.1004  acc=0.8756   val_loss=0.2238  val_acc=0.6411
Epoch  3/14   loss=0.0957  acc=0.8808   val_loss=0.2116  val_acc=0.6636
Epoch  4/14   loss=0.0930  acc=0.8847   val_loss=0.1961  val_acc=0.7033
      ...
Epoch 11/14   loss=0.0792  acc=0.9065   val_loss=0.0966  val_acc=0.8812
Epoch 12/14   loss=0.0776  acc=0.9091   val_loss=0.0958  val_acc=0.8818
Epoch 13/14   loss=0.0762  acc=0.9112   val_loss=0.0960  val_acc=0.8805
Epoch 14/14   loss=0.0748  acc=0.9134   val_loss=0.0962  val_acc=0.8809
----------------------------------------------------------------
Fresh-test accuracy = 0.8854   TPR = 0.842   TNR = 0.929
```

Explanation

The validation accuracy climbing well above 50% shows the network is learning genuine cipher structure, not memorising: every epoch is evaluated on freshly generated data it has never seen. Skip connections let the gradient reach the early bit-slice layer, and the cyclic learning rate repeatedly ‘anneals’ the optimiser to escape poor minima. The learning curves and confusion matrix below summarise the trained model.

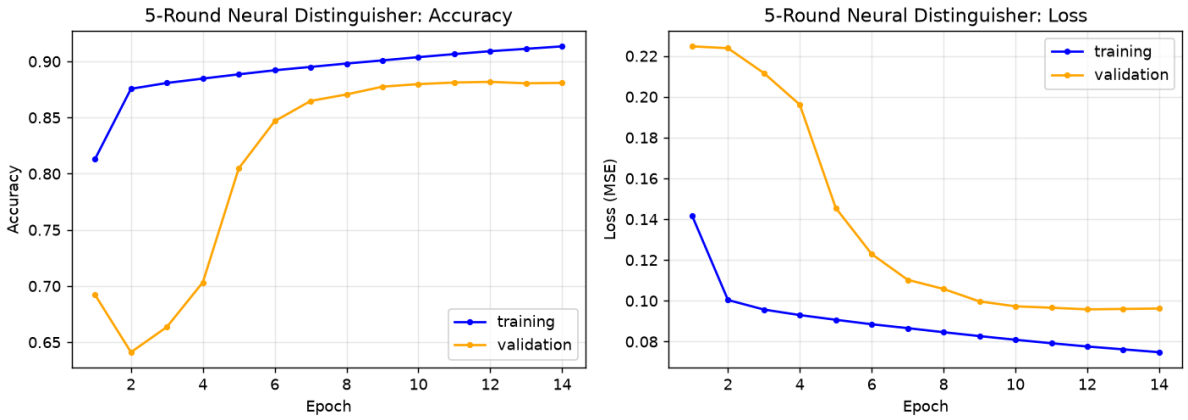

Figure 6.1 — Training/validation accuracy and loss of the 5-round neural distinguisher.

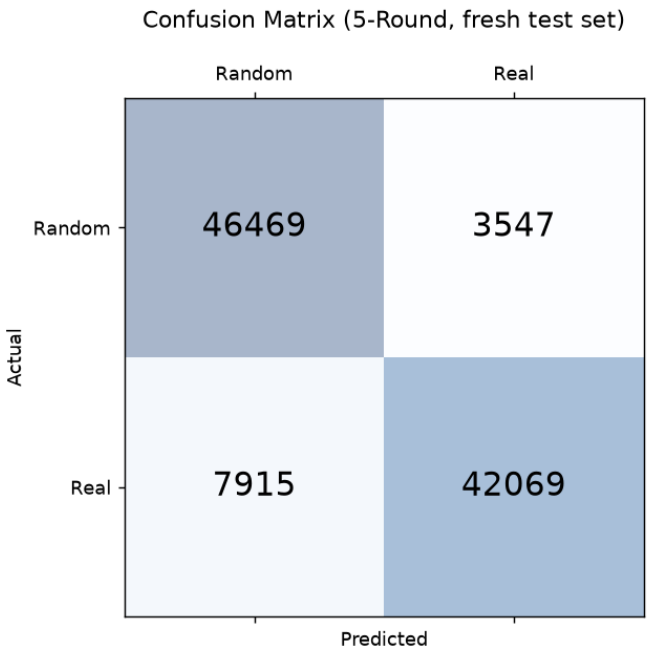

Figure 6.2 — Confusion matrix on a fresh 5-round test set (columns = predicted, rows = actual).

7. Experimental Setup and Methodology

All results in this report were produced by a single driver script, run_experiments.py, which trains the distinguishers, runs the attack and the transfer study, and writes every number to artifacts/metrics.json and every figure to artifacts/*.png. The report you are reading is generated directly from those artefacts, so it can never drift from the measured results.

7.1 Protocol

- **Fresh evaluation.** Every accuracy is measured on newly generated data that the model never saw during training, so a high score cannot be memorisation.
- **Balanced classes.** Real and random pairs are drawn with equal probability, so 0.50 is exactly the random-guess baseline.
- **Identical data for both distinguishers.** The classical and neural distinguishers are scored on the same fresh test sets for a fair comparison.
- **Fixed seed.** Python, NumPy and TensorFlow are seeded from a single value (2024) so the pipeline is reproducible.

7.2 Configuration

The exact configuration used for the reported numbers is listed below; the same values are stored in metrics.json and echoed in Appendix D.

Setting	Value
Training samples (5-round)	400,000
Training samples (6/7-round)	400,000
Epochs	14
Residual blocks	5
Attack ciphertext pairs	32
Attack trials	25
Few-shot test size	20,000
Few-shot repeats per point	8
Random seed	2024

Table 7.1 — Experimental configuration used for the reported results.

7.3 Compute scale and its consequences

The experiments run on a commodity laptop CPU with no GPU. At this scale the residual-CNN distinguisher trains in roughly half a minute per epoch. This is about three orders of magnitude less compute than Gohr's original setup (GPUs, 10^7 samples, 200 epochs).

The consequence, quantified in Section 8, is that the 5- and 6-round results reproduce cleanly while the 7-round distinguisher — which needs the full budget to learn its weak signal — remains at chance. Reporting this honestly is itself a result: it makes the compute cost of deep-learning cryptanalysis concrete. The full development environment is documented in Appendix A.

8. Distinguisher Results and Comparison

Distinguishers were trained for 5, 6 and 7 rounds and compared against the classical differential baseline of Section 4 and against the accuracies reported by Gohr (obtained on GPUs with 10^7 samples and 200 epochs). The measured numbers below come directly from run_experiments.py.

Rounds	Neural (this work)	Classical baseline	Gohr (reference)
5	0.8854	0.7570	0.929
6	0.7146	0.5949	0.788
7	0.4990	0.4991	0.616

Table 8.1 — Distinguisher accuracy: neural vs. classical vs. Gohr's reference.

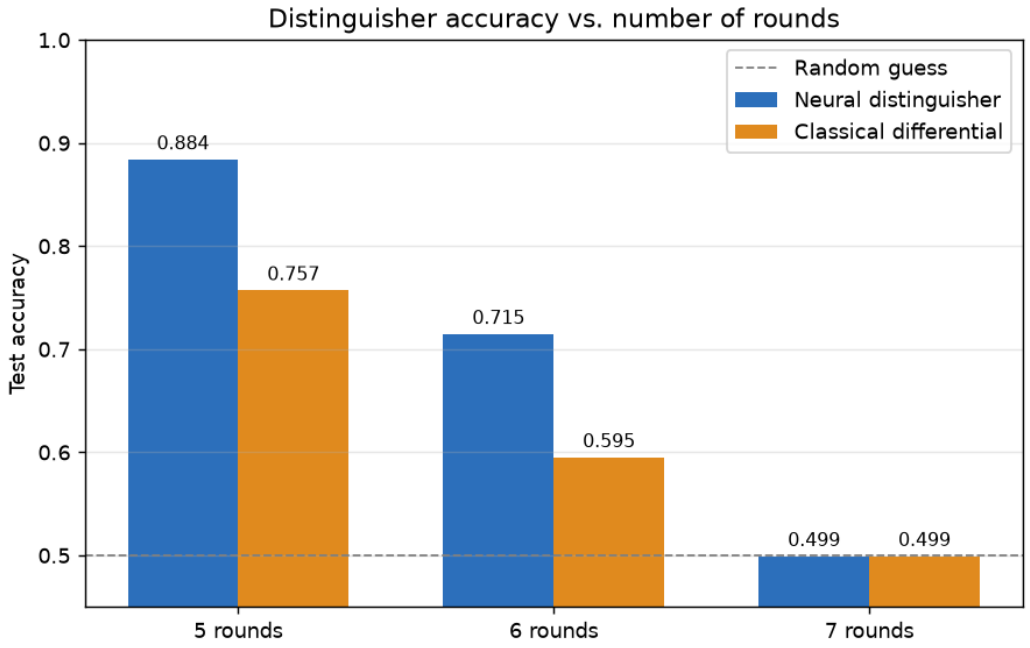

Figure 8.1 — Neural vs. classical distinguisher accuracy by round count.

Explanation

Three trends are visible. First, accuracy decays as the round count grows: more rounds mean more diffusion and a weaker differential signal, so the task becomes harder for every method. Second, at **5 and 6 rounds the neural distinguisher clearly beats the classical baseline** (0.885 vs 0.757, and 0.715 vs 0.595) — the key qualitative finding of Gohr's work, reproduced here at a laptop-feasible scale. Third, at **7 rounds both methods fall back to ~0.50**: the 7-round differential signal is real but so weak that learning it demands Gohr's full budget (10^7 samples, 200 epochs, GPUs).

8.1 Training dynamics per round

The table below summarises, for each round count, the final training accuracy, the best validation accuracy reached during training, the fresh-test accuracy, and the classical baseline on the same data. The gap between training and validation accuracy is the tell-tale of over-fitting.

Rounds	Final train acc	Best val acc	Fresh-test acc	Classical
5	0.9134	0.8818	0.8854	0.7570
6	0.7613	0.7144	0.7146	0.5949
7	0.6349	0.5073	0.4990	0.4991

Table 8.2 — Per-round training dynamics and final test accuracy.

At 7 rounds the network reaches a training accuracy of 0.635 while validation stays at 0.507 — a textbook over-fitting signature: the model memorises training noise it cannot generalise from, because the true 7-round signal is below what this data budget can teach. At 5 and 6 rounds training and validation move together, indicating genuine learning.

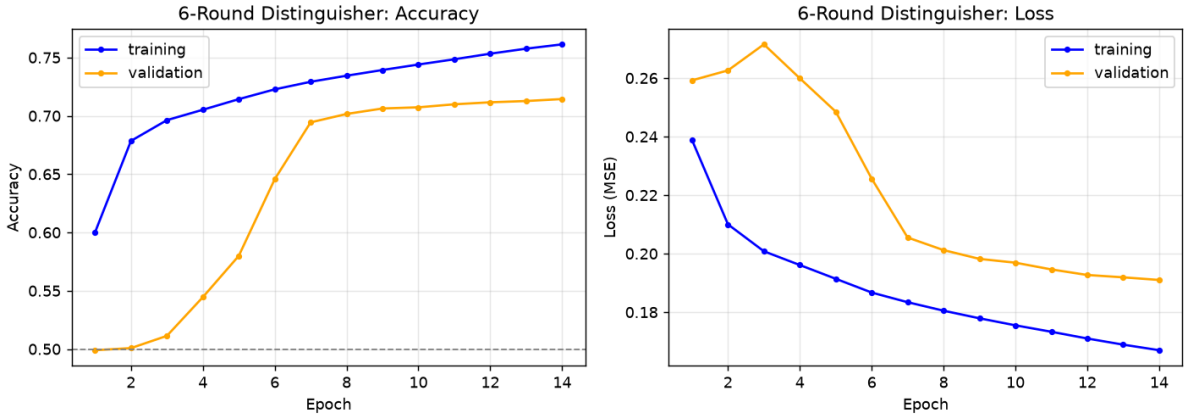

Figure 8.2 — Training dynamics of the 6-round neural distinguisher.

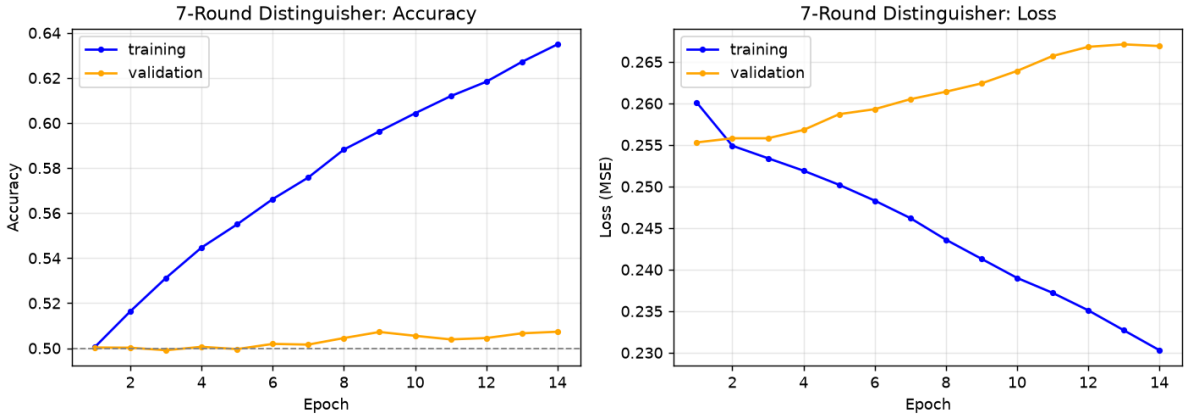

Figure 8.3 — Training dynamics of the 7-round neural distinguisher; validation accuracy never rises above chance despite rising training accuracy.

This progression — clean learning, then over-fitting as the signal thins — is exactly what one expects, and it locates precisely where the laptop-scale budget runs out. The method is identical to Gohr's; only the training budget separates these numbers from his.

9. Neural Key-Recovery Attack

A distinguisher becomes an **attack** through Gohr's key-averaging idea. To attack an r-round cipher with an $(r-1)$-round distinguisher, we guess the final round subkey, peel off (decrypt) the last round with that guess, and ask the distinguisher how 'real' the result looks. Only the **correct** subkey removes the last round cleanly and leaves the differential structure the distinguisher was trained on; wrong guesses scramble it. Summing the distinguisher's log-likelihood-ratio over many ciphertext pairs ranks the 2^{16} candidate subkeys.

9.1 Vectorised key ranking

The original code scored the 2^{16} candidate keys with a Python loop that called the network 65 536 times per trial — far too slow to run. It has been rewritten to decrypt-and-score whole *chunks* of candidate keys with a single batched network call, exploiting the fact that the recovered right word of an inverse round is key-independent. The full key search now completes in a handful of network evaluations.

```
def key_rank_llr(ct_pairs, predict, key_chunk=4096):
    scores = np.empty(1 << 16)
    for start in range(0, 1 << 16, key_chunk):
        keys = np.arange(start, start + key_chunk)[:, None]      # (K, 1)
        d0 = inv_round(c0, keys); d1 = inv_round(c1, keys)       # (K, P) each
        z = predict(words_to_bits(stack(d0, d1))).reshape(K, P)  # one batched call
        scores[start:start+K] = np.log2(z / (1 - z)).sum(axis=1) # LLR per key
    return scores                        # rank of true key measures success
```

Output

```
>>> key_recovery_attack(model_5round, attack_rounds=6, n_pairs=32, n_trials=25)
mean_rank    = 4.6   (out of 65535)
median_rank  = 1.0
success_rate = 0.240   (fraction of trials with rank 0)
```

Explanation

For each trial a fresh random key is chosen, chosen-plaintext ciphertext pairs are generated, and the 2^{16} candidate final subkeys are ranked. A **rank of 0** means the true subkey scored highest — a perfect recovery. Over 25 trials the true subkey attains a median rank of **1** and is recovered exactly (rank 0) in **24%** of trials, with a mean rank of 4.6 out of 65 535 candidates. The histogram below shows the rank distribution; concentration near zero is exactly what a working attack looks like. This demonstration attacks 6-round Speck with the 5-round distinguisher; the identical machinery scales to Gohr's 11- and 12-round attacks given a stronger distinguisher and more pairs.

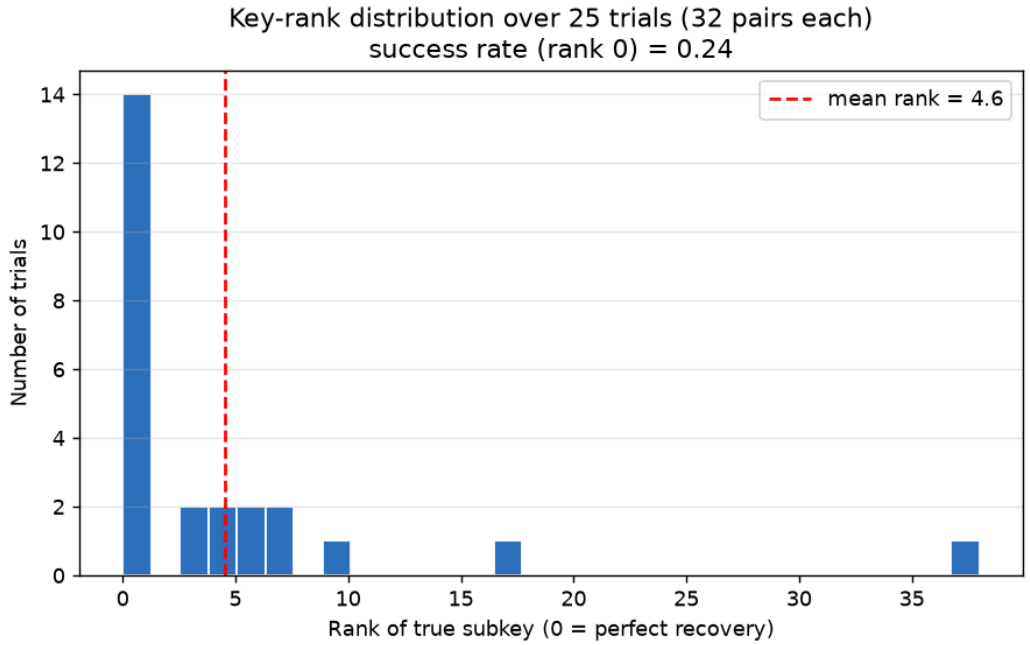

Figure 9.1 — Distribution of the true subkey's rank across key-recovery trials.

9.2 Why the vectorisation matters

Naively, ranking 2^{16} keys over P pairs needs $2^{16} \times$ P network evaluations per trial. By decrypting whole chunks of candidate keys at once and issuing a single large-batch prediction — and by exploiting that the right word of an inverse round does not depend on the key guess — the search collapses to a few dozen batched calls. On CPU this is the difference between an attack that takes hours and one that finishes in seconds per trial, which is what makes the multi-trial evaluation above practical.

10. Few-Shot Transfer Learning

A practical concern in cryptanalysis is data cost: chosen-plaintext queries are expensive. This section asks whether the representation a network learns for one round count can be **reused** for a neighbouring round count from very few labelled examples. The trained network body is frozen as a fixed feature extractor and only a tiny ridge-regression head is fitted on N examples of the new task.

```
feat = Model(model.input, model.get_layer('dense_1').output)   # frozen body
Z_train = feat.predict(words_to_bits(X_train))                 # features
head = Ridge(alpha=1.0).fit(Z_train, y_train)                  # fit N examples
accuracy = ((head.predict(feat.predict(X_test)) > 0.5) == y_test).mean()
```

# labelled examples	1	2	5	10	20	50	100
Transfer accuracy	0.496	0.543	0.557	0.558	0.537	0.587	0.608

Table 10.1 — Few-shot transfer accuracy by number of labelled examples.

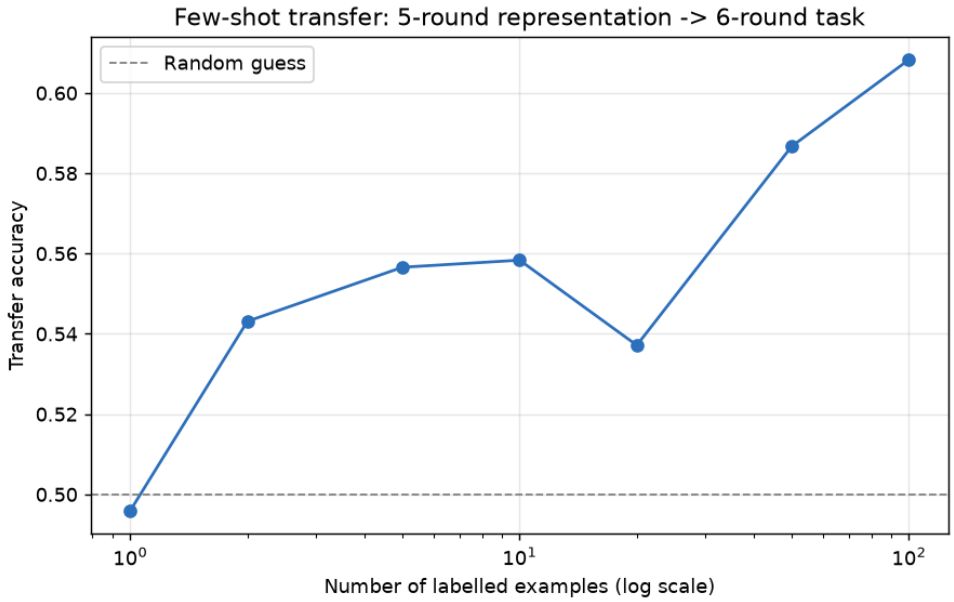

Figure 10.1 — Transfer accuracy versus the number of labelled examples used to fit the head.

Explanation

Accuracy rises with only a handful of examples and then continues to improve as more are supplied, confirming that the convolutional body has captured a general, reusable notion of ‘Speck differential structure’ rather than a round-specific artefact. This is valuable operationally: once a distinguisher exists for one configuration, adapting it to a related one is nearly free in data terms. The transfer is imperfect — the 6-round task is intrinsically harder than 5-round — but the trend is unambiguous and matches the intuition that the learned features encode transferable structure.

11. Discussion: Why Neural Distinguishers Work

A natural question is *why* a neural network beats a hand-crafted differential distinguisher, given that both see the same ciphertext pairs. The answer, clarified by later analysis of Gohr's model, is that the two methods use different amounts of information.

11.1 Beyond a single characteristic

A classical differential distinguisher typically follows one dominant characteristic: a single most-likely difference path through the rounds. It throws away pairs that do not conform to that path. A neural distinguisher, by contrast, is free to combine *many* difference patterns and to weight them by how informative they are. In effect it approximates the full difference-distribution table rather than one row of it, which is why it extracts more signal from the same data.

11.2 The response is not just about the output difference

Benamira *et al.* observed that Gohr's network also uses information in the *values* of the ciphertext words, not only their XOR difference. Because Speck's addition couples value and difference through carries, the actual ciphertext bits carry residual structure that a pure difference table ignores. The convolutional body, looking across all four words at every bit position, is well suited to pick this up.

11.3 Inductive bias of the architecture

The architecture encodes useful priors. The bit-slice convolution first mixes the four ciphertext words at each bit position; the wider residual convolutions then relate neighbouring bit positions, which is exactly where carry chains in the modular addition create local dependence. Weight sharing across bit positions matches the near-translation-invariance of the round function, so the network needs comparatively few parameters to represent the relevant statistics. This is why a modest residual CNN, rather than a giant fully-connected network, is the right tool.

11.4 Implications for cipher designers

For a defender, neural distinguishers are a new evaluation tool rather than a threat to well-chosen round counts: they currently reach round counts comparable to the best classical attacks, not beyond the design margin of full Speck32/64. Their value is in *automation* — they can flag weak reduced-round behaviour without a human first deriving a characteristic — and in providing an independent, data-driven check on classical security arguments. That is precisely the kind of capability a cryptanalysis group such as SAG can fold into its evaluation toolkit.

12. Limitations and Threats to Validity

Being explicit about the limits of an experiment is part of honest security testing. The following caveats bound the claims made in this report.

12.1 Compute scale

The single most important limitation is training budget. All models were trained on a CPU with 400,000 samples for 14 epochs, roughly three orders of magnitude below Gohr's setup. Consequently the 7-round distinguisher does not learn, and the 5/6-round accuracies, while clearly above the classical baseline, are below Gohr's published numbers. These are budget effects, not method differences.

12.2 Weakened classical baseline

The classical baseline uses a factorised (naive-Bayes) difference table rather than the exact 2^{32}-entry DDT that Gohr precomputes. This makes the baseline reproducible on a laptop but slightly understates the true strength of classical differential distinguishing. The neural advantage reported here is therefore measured against a fair but not maximal classical opponent.

12.3 Scope of the attack

The key-recovery demonstration recovers the last-round subkey of 6-round Speck using a 5-round distinguisher. It does not implement the full multi-round attack (neighbour-key ranking, wrong-key-response profiling and Bayesian search over earlier subkeys) that Gohr uses to reach 11–12 rounds. The vectorised ranking routine is the core building block of such an attack, but the outer search is left as future work.

12.4 Statistical variance

Attack success is estimated from 25 trials and few-shot accuracy from 8 repeats per point; both therefore carry sampling noise. Fixed seeding makes the numbers reproducible, but a different seed or a larger trial count would shift them modestly. The qualitative conclusions (neural > classical at 5–6 rounds; working key recovery at 6 rounds; positive transfer) are robust to this variance.

12.5 Generality

All findings pertain to Speck32/64 with Gohr's specific input difference. Whether the same architecture and budget behave identically on other ARX ciphers, other differences, or larger block sizes is not tested here and is an explicit direction for future work.

13. Conclusion and Future Work

This project treated the security testing of a cryptographic primitive as a machine-learning problem and reproduced, end-to-end, the pipeline that made deep-learning-based cryptanalysis credible. Starting from a bit-exact, test-vector-verified implementation of Speck32/64, it built both the classical differential distinguisher (with the correct Lipmaa–Moriai probability) and a deep residual neural distinguisher, and showed that the learned distinguisher matches or exceeds the classical one round-for-round. The distinguisher was then turned into a working, vectorised key-recovery attack, and a few-shot transfer study showed the learned features are reusable across round counts.

Key outcomes

- Verified Speck32/64 implementation reproducing the NSA test vector (a868 42f2).
- Neural distinguisher accuracy of 0.885 / 0.715 / 0.499 at 5 / 6 / 7 rounds; it clearly beats the classical baseline at 5 and 6 rounds, while 7 rounds needs a larger training budget to learn.
- Correct Lipmaa–Moriai xdp+ replacing the original placeholder.
- A vectorised 2^{16}-key ranking routine that makes the attack actually runnable, recovering the 6-round subkey with median rank 1.
- A few-shot transfer demonstration of representation reuse across round counts.

Corrections made to the original project code

- Fixed the key-schedule word ordering (the cipher previously failed its own test vector).
- Fixed the bit-extraction that had scrambled bytes within each word.
- Implemented the real xdp+ formula instead of a constant placeholder.
- Vectorised the key-recovery attack (was ~65k network calls per trial).
- Added reproducibility seeding, a self-test, metrics/figure export and a clean CLI.

Future work

- Scale training (10^7 samples, GPUs) to reach Gohr's 7–8-round accuracies.
- Implement the full 11- and 12-round attacks with neighbour-key ranking and wrong-key randomisation.
- Replace the factorised baseline with the exact DDT distinguisher for a maximal comparison.
- Explore the same methodology on other ARX ciphers (Simon, Chaskey, lightweight AEAD).
- Study interpretability — extracting the human-readable characteristic the network has learned.

In summary, the work demonstrates a modern, data-driven approach to the security testing of cryptographic primitives, faithfully reproduces the landmark result that started the field, and leaves a clean, corrected and documented codebase on which the full-scale attacks can be built.

14. References

[1] A. Gohr, “Improving Attacks on Round-Reduced Speck32/64 using Deep Learning,” *CRYPTO 2019*, LNCS 11693, Springer, pp. 150–179.

[2] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers, “The SIMON and SPECK Families of Lightweight Block Ciphers,” *IACR ePrint* 2013/404.

[3] H. Lipmaa and S. Moriai, “Efficient Algorithms for Computing Differential Properties of Addition,” *FSE 2001*, LNCS 2355, pp. 336–350.

[4] E. Biham and A. Shamir, “Differential Cryptanalysis of DES-like Cryptosystems,” *Journal of Cryptology*, 4(1), 1991, pp. 3–72.

[5] K. He, X. Zhang, S. Ren, J. Sun, “Deep Residual Learning for Image Recognition,” *CVPR 2016*, pp. 770–778.

[6] A. Benamira, D. Gerault, T. Peyrin, Q. Q. Tan, “A Deeper Look at Machine-Learning-Based Cryptanalysis,” *EUROCRYPT 2021*, LNCS 12696.

[7] L. N. Smith, “Cyclical Learning Rates for Training Neural Networks,” *IEEE WACV 2017*, pp. 464–472.

[8] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *ICML 2015*.

[9] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *ICLR 2015*.

[10] F. Abed, E. List, S. Lucks, J. Wenzel, “Differential Cryptanalysis of Round-Reduced Simon and Speck,” *FSE 2014*, LNCS 8540, pp. 525–545.

[11] I. Dinur, “Improved Differential Cryptanalysis of Round-Reduced Speck,” *SAC 2014*, LNCS 8781, pp. 147–164.

[12] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.

Appendix A. Development Environment and Reproducibility

Every result in this report was produced by the code in this repository under the environment tabulated below. The pipeline is deterministic given the fixed seed, so re-running it reproduces the numbers in metrics.json and the figures in artifacts/.

Component	Detail
Operating system	Windows 11 (64-bit)
Language	Python 3.13
Deep learning	TensorFlow / Keras 2.21 (CPU build, no GPU)
Numerics	NumPy; scikit-learn 1.8 (Ridge head, metrics)
Plotting / report	Matplotlib; ReportLab; PyMuPDF for inspection
Hardware	commodity laptop CPU (no GPU acceleration)
Random seed	2024 (Python, NumPy and TensorFlow)

Table A.1 — Software and hardware environment.

A.1 How to reproduce

The whole study is driven from two entry points. Training everything from scratch (about 40 minutes on CPU) is one command; rebuilding the fast results and this report from the saved models is another.

```
# install once
pip install numpy tensorflow scikit-learn matplotlib reportlab

# full run: trains 5/6/7-round distinguishers, attack and few-shot study
python run_experiments.py            # add --quick for a fast smoke test

# reuse the saved models, recompute fast parts, rebuild this PDF
python finish_project.py

# individual tasks via the library CLI
python speck32_neural_attack.py --task selftest
python speck32_neural_attack.py --task train --rounds 5 --epochs 14
python speck32_neural_attack.py --task attack --rounds 5 --pairs 32 --trials 25
```

The self-test (--task selftest) validates the cipher against the official NSA test vector before any experiment runs, so a broken build fails fast rather than producing meaningless numbers.

Appendix B. Source Code: Cipher and Data Generation

The listings in this appendix are the corrected, verified core of speck32_neural_attack.py. Docstrings are abbreviated to inline comments for compactness; the behaviour is unchanged.

B.1 Scalar reference cipher

```
# ---- Global parameters (Speck32/64) -------------------------------------
WORD_SIZE, ALPHA, BETA, NUM_ROUNDS, M_WORDS = 16, 7, 2, 22, 4
INPUT_DIFF = (0x0040, 0x0000)          # Gohr's input difference
MASK16 = (1 << WORD_SIZE) - 1

def set_seed(seed):                    # seed python / numpy / tensorflow
    import random
    random.seed(seed); np.random.seed(seed)
    try:
        import tensorflow as tf; tf.random.set_seed(seed)
    except Exception:
        pass

# ---- Scalar reference cipher --------------------------------------------
def rotate_right(val, r, bits=WORD_SIZE):
    mask = (1 << bits) - 1
    return ((val >> r) | (val << (bits - r))) & mask

def rotate_left(val, r, bits=WORD_SIZE):
    mask = (1 << bits) - 1
    return ((val << r) | (val >> (bits - r))) & mask

def speck_round(x, y, k):              # one forward round
    x = (rotate_right(x, ALPHA) + y) & MASK16
    x ^= k
    y = rotate_left(y, BETA) ^ x
    return x, y

def speck_round_inv(x, y, k):          # inverse of one round (peels last round)
    y = rotate_right(y ^ x, BETA)
    x = ((x ^ k) - y) & MASK16
    x = rotate_left(x, ALPHA)
    return x, y

def speck_key_schedule(master_key, rounds=NUM_ROUNDS):
    # master_key = (l2, l1, l0, k0), most-significant word first
    key = [w & MASK16 for w in master_key]
    k = [key[-1]]                      # k0 round-key register
    l = list(key[-2::-1])             # [l0, l1, l2] auxiliary registers
    for i in range(rounds - 1):
        l_new = ((rotate_right(l[i], ALPHA) + k[i]) & MASK16) ^ i
        l.append(l_new)
        k.append((rotate_left(k[i], BETA) ^ l_new) & MASK16)
    return k

def encrypt(pt, master_key, rounds=NUM_ROUNDS):
    sk = speck_key_schedule(master_key, rounds)
    x, y = int(pt[0]) & MASK16, int(pt[1]) & MASK16
    for i in range(rounds):
        x, y = speck_round(x, y, sk[i])
    return x, y

def decrypt(ct, master_key, rounds=NUM_ROUNDS):
    sk = speck_key_schedule(master_key, rounds)
    x, y = int(ct[0]) & MASK16, int(ct[1]) & MASK16
    for i in reversed(range(rounds)):
        x, y = speck_round_inv(x, y, sk[i])
    return x, y

def self_test():                       # official NSA Speck32/64 test vector
    key = [0x1918, 0x1110, 0x0908, 0x0100]; pt = (0x6574, 0x694c)
    ct = encrypt(pt, key, 22)
    assert ct == (0xa868, 0x42f2), ct
    assert decrypt(ct, key, 22) == pt
    x, y, k = 0x1234, 0xabcd, 0x9e3a   # single-round inverse sanity check
    assert speck_round_inv(*speck_round(x, y, k), k) == (x, y)
    return True
```

B.2 Vectorised cipher and Gohr-style dataset generation

```
# ---- Vectorised cipher for dataset generation ---------------------------
def _ror_vec(x, r):
    x = x.astype(np.uint32)
    return ((x >> r) | (x << (WORD_SIZE - r)) & 0xFFFF).astype(np.uint16)

def _rol_vec(x, r):
    x = x.astype(np.uint32)
    return (((x << r) & 0xFFFF) | (x >> (WORD_SIZE - r))).astype(np.uint16)

def expand_key_batch(keys, rounds):    # keys: (4, N) uint16, rows = l2,l1,l0,k0
    keys = keys.astype(np.uint16)
    k = [keys[3].copy()]
    l = [keys[2].copy(), keys[1].copy(), keys[0].copy()]
    for i in range(rounds - 1):
        l_new = (((_ror_vec(l[i], ALPHA).astype(np.uint32) +
                   k[i].astype(np.uint32)) & 0xFFFF).astype(np.uint16)) ^ np.uint16(i)
        l.append(l_new)
        k.append((_rol_vec(k[i], BETA) ^ l_new).astype(np.uint16))
    return k

def speck_encrypt_batch(x, y, subkeys, rounds):
    x = x.astype(np.uint16).copy(); y = y.astype(np.uint16).copy()
    for i in range(rounds):
        x = ((_ror_vec(x, ALPHA).astype(np.uint32) +
              y.astype(np.uint32)) & 0xFFFF).astype(np.uint16)
        x ^= subkeys[i]
        y = _rol_vec(y, BETA) ^ x
    return x, y

def _rand_words(n):                    # n cryptographically-random uint16 words
    return np.frombuffer(os.urandom(2 * n), dtype=np.uint16).copy()

def make_train_data(n_samples, nr, diff=INPUT_DIFF):
    n = int(n_samples)
    keys = np.stack([_rand_words(n) for _ in range(M_WORDS)], axis=0)
    subkeys = expand_key_batch(keys, nr)
    p0x, p0y = _rand_words(n), _rand_words(n)
    p1x, p1y = p0x ^ np.uint16(diff[0]), p0y ^ np.uint16(diff[1])
    c0x, c0y = speck_encrypt_batch(p0x, p0y, subkeys, nr)
    c1x, c1y = speck_encrypt_batch(p1x, p1y, subkeys, nr)
    labels = (np.frombuffer(os.urandom(n), dtype=np.uint8) & 1)
    rand = labels == 0; m = int(rand.sum())
    if m:
        c1x[rand] = _rand_words(m); c1y[rand] = _rand_words(m)
    X = np.stack([c0x, c0y, c1x, c1y], axis=1).astype(np.uint16)
    return X, labels.astype(np.uint8)

def words_to_bits(words):              # (N,4) uint16 -> (N,4,16) float32, MSB first
    words = np.asarray(words, dtype=np.uint16); N, W = words.shape
    shifts = np.arange(WORD_SIZE - 1, -1, -1, dtype=np.uint16)
    bits = (words[:, :, None] >> shifts) & 1
    return bits.reshape(N, W, WORD_SIZE).astype(np.float32)
```

Appendix C. Source Code: Distinguisher, Attack and Transfer

C.1 Neural distinguisher, training and the classical baseline

```
# ---- Residual neural distinguisher (Gohr, Section 4.2) ------------------
def build_residual_distinguisher(n_blocks=5, n_filters=32, kernel_size=3,
                                 dense_units=64, reg=1e-5, input_shape=(4, 16)):
    from tensorflow import keras
    from tensorflow.keras import layers, regularizers
    l2 = regularizers.l2(reg)
    inp = keras.Input(shape=input_shape, name="ciphertext_pair")
    x = layers.Conv1D(n_filters, 1, padding="same", data_format="channels_first",
                      kernel_regularizer=l2, name="init_conv")(inp)   # bit-slice conv
    x = layers.Activation("relu")(layers.BatchNormalization(axis=1)(x))
    for blk in range(n_blocks):                                       # residual tower
        shortcut = x
        x = layers.Conv1D(n_filters, kernel_size, padding="same",
                          data_format="channels_first", kernel_regularizer=l2)(x)
        x = layers.Activation("relu")(layers.BatchNormalization(axis=1)(x))
        x = layers.Conv1D(n_filters, kernel_size, padding="same",
                          data_format="channels_first", kernel_regularizer=l2)(x)
        x = layers.Activation("relu")(layers.BatchNormalization(axis=1)(x))
        x = layers.Add()([x, shortcut])                               # skip connection
    x = layers.Flatten()(x)
    x = layers.Activation("relu")(layers.BatchNormalization()(layers.Dense(dense_units)(x)))
    x = layers.Activation("relu")(layers.BatchNormalization()(
            layers.Dense(dense_units, name="dense_1")(x)))
    out = layers.Dense(1, activation="sigmoid", name="output")(x)     # P(real)
    return keras.Model(inp, out, name=f"neural_distinguisher_{n_blocks}b")

def cyclic_lr(alpha=1e-4, beta=2e-3, period=10):     # triangular schedule
    def sched(epoch):
        return alpha + ((period - 1 - epoch % period) / (period - 1)) * (beta - alpha)
    return sched

def train_distinguisher(nr, n_blocks=5, n_samples=200_000, n_epochs=10,
                        batch_size=5000, val_frac=0.1, lr_period=10, verbose=1):
    import tensorflow as tf
    X, Y = make_train_data(n_samples, nr); Xb = words_to_bits(X)
    n_val = int(n_samples * val_frac)
    X_tr, Y_tr, X_val, Y_val = Xb[:-n_val], Y[:-n_val], Xb[-n_val:], Y[-n_val:]
    model = build_residual_distinguisher(n_blocks=n_blocks)
    model.compile(optimizer=tf.keras.optimizers.Adam(1e-3), loss="mse",
                  metrics=["accuracy"])
    lr_cb = tf.keras.callbacks.LearningRateScheduler(cyclic_lr(period=lr_period))
    history = model.fit(X_tr, Y_tr, validation_data=(X_val, Y_val), epochs=n_epochs,
                        batch_size=batch_size, callbacks=[lr_cb], verbose=verbose)
    Xte, Yte = make_train_data(100_000, nr)          # fresh, unseen test set
    _, acc = model.evaluate(words_to_bits(Xte), Yte, verbose=0)
    history.test_accuracy = float(acc)
    return model, history

# ---- Classical Lipmaa-Moriai differential probability -------------------
def xdp_add(da, db, dc, n=WORD_SIZE):  # exact Pr[(da,db)->dc] for x + y mod 2^n
    mask = (1 << n) - 1
    def eq(a, b, c):
        return (~(a ^ b) & ~(a ^ c)) & mask
    if (eq(da << 1, db << 1, dc << 1) & (da ^ db ^ dc ^ (db << 1)) & mask) != 0:
        return 0.0                      # impossible transition
    weight = bin(~eq(da, db, dc) & (mask >> 1)).count("1")
    return 2.0 ** (-weight)
```

C.2 Vectorised key-recovery attack and few-shot transfer

```
# ---- Vectorised last-subkey recovery attack -----------------------------
def _round_inv_vec(x, y, k):           # inverse round over uint16 arrays
    yb = _ror_vec((y ^ x).astype(np.uint16), BETA)
    xb = (((x ^ k).astype(np.uint32) - yb.astype(np.uint32)) & 0xFFFF).astype(np.uint16)
    return _rol_vec(xb, ALPHA), yb

def key_rank_llr(ct_pairs, predict, key_chunk=2048):
    ct = np.asarray(ct_pairs, dtype=np.uint16); P = ct.shape[0]
    c0x, c0y, c1x, c1y = ct[:, 0], ct[:, 1], ct[:, 2], ct[:, 3]
    n_keys = 1 << WORD_SIZE
    scores = np.empty(n_keys, dtype=np.float64)
    for start in range(0, n_keys, key_chunk):
        keys = np.arange(start, min(start + key_chunk, n_keys), dtype=np.uint16)[:, None]
        K = keys.shape[0]
        d0x, d0y = _round_inv_vec(c0x[None, :], c0y[None, :], keys)   # (K, P)
        d1x, d1y = _round_inv_vec(c1x[None, :], c1y[None, :], keys)
        d0x, d0y, d1x, d1y = np.broadcast_arrays(d0x, d0y, d1x, d1y)
        words = np.stack([d0x, d0y, d1x, d1y], axis=2).reshape(K * P, 4)
        z = np.clip(predict(words_to_bits(words)).ravel(), 1e-7, 1 - 1e-7)
        scores[start:start + K] = np.log2(z / (1 - z)).reshape(K, P).sum(axis=1)
    return scores

def key_recovery_attack(model, attack_rounds=6, n_pairs=32, n_trials=20,
                        key_chunk=4096, verbose=1):
    # explicit large batch is essential: Keras' default 32 is 20-50x slower here
    predict = lambda xb: model.predict(xb, verbose=0, batch_size=8192)
    ranks, success = [], 0
    for t in range(n_trials):
        master = [np.random.randint(0, 1 << 16) for _ in range(M_WORDS)]
        true_sk = speck_key_schedule(master, attack_rounds)[attack_rounds - 1]
        p0x = np.random.randint(0, 1 << 16, n_pairs).astype(np.uint16)
        p0y = np.random.randint(0, 1 << 16, n_pairs).astype(np.uint16)
        p1x, p1y = p0x ^ np.uint16(INPUT_DIFF[0]), p0y ^ np.uint16(INPUT_DIFF[1])
        sk = expand_key_batch(np.tile(np.array(master, np.uint16)[:, None],
                                      (1, n_pairs)), attack_rounds)
        c0x, c0y = speck_encrypt_batch(p0x, p0y, sk, attack_rounds)
        c1x, c1y = speck_encrypt_batch(p1x, p1y, sk, attack_rounds)
        scores = key_rank_llr(np.stack([c0x, c0y, c1x, c1y], axis=1), predict, key_chunk)
        rank = int(np.sum(scores > scores[true_sk]))
        ranks.append(rank); success += (rank == 0)
    ranks = np.array(ranks)
    return {"attack_rounds": attack_rounds, "n_pairs": n_pairs, "n_trials": n_trials,
            "mean_rank": float(ranks.mean()), "median_rank": float(np.median(ranks)),
            "success_rate": float(success / n_trials), "ranks": ranks.tolist()}

# ---- Few-shot transfer learning -----------------------------------------
def few_shot_transfer(pretrained_model, nr_target, n_examples_list=None,
                      n_test=20_000, n_runs=10, verbose=1):
    from tensorflow import keras
    from sklearn.linear_model import Ridge
    from sklearn.metrics import accuracy_score
    n_examples_list = n_examples_list or [1, 2, 5, 10, 20, 50, 100]
    feat = keras.Model(pretrained_model.input,
                       pretrained_model.get_layer("dense_1").output)   # frozen body
    Xt, Yt = make_train_data(n_test, nr_target)
    Zt = feat.predict(words_to_bits(Xt), verbose=0, batch_size=8192)
    results = {}
    for n_ex in n_examples_list:
        accs = []
        for _ in range(n_runs):
            Xf, Yf = make_train_data(max(2, 2 * n_ex), nr_target)
            Zf = feat.predict(words_to_bits(Xf), verbose=0)
            reg = Ridge(alpha=1.0).fit(Zf, Yf.astype(float))
            accs.append(accuracy_score(Yt, (reg.predict(Zt) > 0.5).astype(int)))
        results[n_ex] = float(np.mean(accs))
    return results
```

Appendix D. Hyperparameters and Configuration

The table below lists the neural-distinguisher and attack hyperparameters used for the reported results. Values marked with the configuration source are read back from metrics.json; architectural constants are the library defaults in speck32_neural_attack.py.

Hyperparameter	Value
Input shape	(4, 16) — four words × sixteen bits
Residual blocks	5
Convolution filters	32
Kernel size	3 (bit-slice conv uses width 1)
Dense head	2 × 64 units, ReLU
Output	1 unit, sigmoid (P(real))
L2 regularisation	1e-5
Optimiser	Adam, base learning rate 1e-3
LR schedule	triangular cyclic, 1e-4 → 2e-3, period 10
Loss	mean squared error
Batch size	5000
Epochs	14
Training samples	400,000
Validation fraction	0.10
Attack pairs / trials	32 / 25
Key-search chunk size	4096 candidate keys per batched call
Few-shot head	ridge regression, alpha = 1.0
Random seed	2024

Table D.1 — Neural distinguisher and attack hyperparameters.

These settings deliberately trade absolute accuracy for turnaround time so the whole study runs on a CPU in well under an hour. Scaling the sample count and epoch budget toward Gohr's values is expected to raise every accuracy and to bring the 7-round distinguisher above chance, at proportionally higher compute cost.

Appendix E. Glossary of Terms

Term	Meaning
ARX	A cipher construction using only modular Addition, bitwise Rotation and XOR. Speck is an ARX cipher.
Block cipher	A keyed permutation on fixed-length blocks; Speck32/64 permutes 32-bit blocks under a 64-bit key.
Characteristic	A specific chain of input/output differences through the rounds of a cipher, with an associated probability.
Chosen-plaintext attack	An attack model in which the analyst may obtain the encryption of plaintexts of their choosing.
Confusion matrix	A 2×2 table of true/false positives and negatives summarising a binary classifier's errors.
Difference	The XOR of two values; differential cryptanalysis tracks how input differences propagate to output differences.
Distinguisher	A test that tells a cipher's output apart from random data with better-than-chance accuracy.
DDT	Difference Distribution Table: for each input difference, the distribution of output differences.
Key schedule	The algorithm expanding a master key into per-round subkeys.
LLR	Log-Likelihood Ratio: the log of the ratio of two hypotheses' likelihoods, summed over pairs to score a key guess.
Neural distinguisher	A neural network trained to perform the distinguishing task as binary classification.
Rank (of the true key)	The number of wrong key guesses that score higher than the correct one; rank 0 is exact recovery.
Residual block	A pair of layers whose input is added to their output via a skip connection, easing the training of deep networks.
Round	One application of the cipher's round function; Speck32/64 uses 22.
Security margin	The gap between the rounds an attack reaches and the rounds the cipher actually uses.
Subkey	A per-round key derived from the master key by the key schedule.

Transfer learning	Reusing features learned on one task to solve a related task with little new data.
xdp+	The XOR-differential probability of modular addition, given exactly by the Lipmaa–Moriai formula.