tutorial_differential-linear_cryptanalysis

March 19, 2025

1 Differential-Linear Cryptanalysis

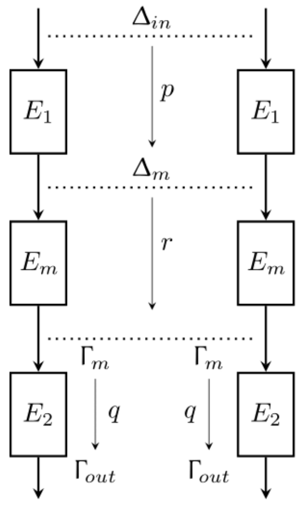


Figure 1: A differential-linear distinguisher with an improved structure that mitigates the independence assumption between the top and the bottom parts.

Assuming independence of $E1(x)$, Em(x) and $E2(x)$, the DL distinguisher correlation is given by:

prq2

Where

$p =$ **Pr**x 2n [$E1(x)$ $E1(x$ Δin) = Δm]

$r =$ **Cor**x [Γm, Em(x) Γm, Em$(x$ Δm)]

$q =$ **Cor**x [Γm, x Γout, E (x)]

and denotes the set of samples over which the correlation is computed. Then, the total correlation can be estimated as:

Thus, by preparing approximately p-2 r-2 q-4 pairs of chosen plaintexts $(x, \tilde{x})$, where x̃ = x Δin and is a small constant, one can distinguish the cipher from a random permutation.

The improved DL distinguisher is denoted as:

Δin →DL Γout

2 Automatic Differential-Linear Cryptanalysis using CLAASP

2.1 Creating a reduced version of Speck (6 rounds)

[1]:
```
from claasp.ciphers.block_ciphers.speck_block_cipher import SpeckBlockCipher
speck = SpeckBlockCipher(number_of_rounds=6)
speck.print()
```

```
cipher_id = speck_p32_k64_o32_r6
cipher_type = block_cipher
cipher_inputs = ['plaintext', 'key']
cipher_inputs_bit_size = [32, 64]
cipher_output_bit_size = 32
cipher_number_of_rounds = 6

    # round = 0 - round component = 0
    id = rot_0_0
    type = word_operation
    input_bit_size = 16
    input_id_link = ['plaintext']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 0 - round component = 1
    id = modadd_0_1
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_0_0', 'plaintext']
    input_bit_positions = [[0, 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]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 0 - round component = 2
    id = xor_0_2
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_0_1', 'key']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 0 - round component = 3
    id = rot_0_3
    type = word_operation
    input_bit_size = 16
    input_id_link = ['plaintext']
    input_bit_positions = [[16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 0 - round component = 4
    id = xor_0_4
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_0_2', 'rot_0_3']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 0 - round component = 5
    id = intermediate_output_0_5
    type = intermediate_output
    input_bit_size = 16
    input_id_link = ['key']
    input_bit_positions = [[48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63]]
    output_bit_size = 16
    description = ['round_key_output']

    # round = 0 - round component = 6
    id = intermediate_output_0_6
    type = intermediate_output
    input_bit_size = 32
    input_id_link = ['xor_0_2', 'xor_0_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 32
    description = ['round_output']

    # round = 1 - round component = 0
    id = constant_1_0
    type = constant
    input_bit_size = 0
    input_id_link = ['']
    input_bit_positions = [[]]
    output_bit_size = 16
    description = ['0x0000']
    # round = 1 - round component = 1
    id = rot_1_1
    type = word_operation
    input_bit_size = 16
    input_id_link = ['key']
    input_bit_positions = [[32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 1 - round component = 2
    id = modadd_1_2
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_1_1', 'key']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 1 - round component = 3
    id = xor_1_3
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_1_2', 'constant_1_0']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 1 - round component = 4
    id = rot_1_4
    type = word_operation
    input_bit_size = 16
    input_id_link = ['key']
    input_bit_positions = [[48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 1 - round component = 5
    id = xor_1_5
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_1_3', 'rot_1_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 1 - round component = 6
    id = rot_1_6
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_0_2']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 1 - round component = 7
    id = modadd_1_7
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_1_6', 'xor_0_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 1 - round component = 8
    id = xor_1_8
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_1_7', 'xor_1_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 1 - round component = 9
    id = rot_1_9
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_0_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 1 - round component = 10
    id = xor_1_10
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_1_8', 'rot_1_9']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 1 - round component = 11
    id = intermediate_output_1_11
    type = intermediate_output
    input_bit_size = 16
    input_id_link = ['xor_1_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['round_key_output']

    # round = 1 - round component = 12
    id = intermediate_output_1_12
    type = intermediate_output
    input_bit_size = 32
    input_id_link = ['xor_1_8', 'xor_1_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 32
    description = ['round_output']

    # round = 2 - round component = 0
    id = constant_2_0
    type = constant
    input_bit_size = 0
    input_id_link = ['']
    input_bit_positions = [[]]
    output_bit_size = 16
    description = ['0x0001']

    # round = 2 - round component = 1
    id = rot_2_1
    type = word_operation
    input_bit_size = 16
    input_id_link = ['key']
    input_bit_positions = [[16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
29, 30, 31]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 2 - round component = 2
    id = modadd_2_2
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_2_1', 'xor_1_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 2 - round component = 3
    id = xor_2_3
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_2_2', 'constant_2_0']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 2 - round component = 4
    id = rot_2_4
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_1_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 2 - round component = 5
    id = xor_2_5
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_2_3', 'rot_2_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 2 - round component = 6
    id = rot_2_6
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_1_8']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 2 - round component = 7
    id = modadd_2_7
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_2_6', 'xor_1_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 2 - round component = 8
    id = xor_2_8
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_2_7', 'xor_2_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 2 - round component = 9
    id = rot_2_9
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_1_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 2 - round component = 10
    id = xor_2_10
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_2_8', 'rot_2_9']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 2 - round component = 11
    id = intermediate_output_2_11
    type = intermediate_output
    input_bit_size = 16
    input_id_link = ['xor_2_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['round_key_output']

    # round = 2 - round component = 12
    id = intermediate_output_2_12
    type = intermediate_output
    input_bit_size = 32
    input_id_link = ['xor_2_8', 'xor_2_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 32
    description = ['round_output']

    # round = 3 - round component = 0
    id = constant_3_0
    type = constant
    input_bit_size = 0
    input_id_link = ['']
    input_bit_positions = [[]]
    output_bit_size = 16
    description = ['0x0002']

    # round = 3 - round component = 1
    id = rot_3_1
    type = word_operation
    input_bit_size = 16
    input_id_link = ['key']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 3 - round component = 2
    id = modadd_3_2
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_3_1', 'xor_2_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 3 - round component = 3
    id = xor_3_3
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_3_2', 'constant_3_0']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]
    # round = 3 - round component = 4
    id = rot_3_4
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_2_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 3 - round component = 5
    id = xor_3_5
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_3_3', 'rot_3_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 3 - round component = 6
    id = rot_3_6
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_2_8']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 3 - round component = 7
    id = modadd_3_7
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_3_6', 'xor_2_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 3 - round component = 8
    id = xor_3_8
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_3_7', 'xor_3_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 3 - round component = 9
    id = rot_3_9
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_2_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 3 - round component = 10
    id = xor_3_10
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_3_8', 'rot_3_9']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 3 - round component = 11
    id = intermediate_output_3_11
    type = intermediate_output
    input_bit_size = 16
    input_id_link = ['xor_3_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['round_key_output']

    # round = 3 - round component = 12
    id = intermediate_output_3_12
    type = intermediate_output
    input_bit_size = 32
    input_id_link = ['xor_3_8', 'xor_3_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 32
    description = ['round_output']

    # round = 4 - round component = 0
    id = constant_4_0
    type = constant
    input_bit_size = 0
    input_id_link = ['']
    input_bit_positions = [[]]
    output_bit_size = 16
    description = ['0x0003']

    # round = 4 - round component = 1
    id = rot_4_1
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_1_3']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 4 - round component = 2
    id = modadd_4_2
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_4_1', 'xor_3_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 4 - round component = 3
    id = xor_4_3
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_4_2', 'constant_4_0']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 4 - round component = 4
    id = rot_4_4
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_3_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 4 - round component = 5
    id = xor_4_5
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_4_3', 'rot_4_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 4 - round component = 6
    id = rot_4_6
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_3_8']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 4 - round component = 7
    id = modadd_4_7
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_4_6', 'xor_3_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 4 - round component = 8
    id = xor_4_8
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_4_7', 'xor_4_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 4 - round component = 9
    id = rot_4_9
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_3_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 4 - round component = 10
    id = xor_4_10
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_4_8', 'rot_4_9']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 4 - round component = 11
    id = intermediate_output_4_11
    type = intermediate_output
    input_bit_size = 16
    input_id_link = ['xor_4_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['round_key_output']

    # round = 4 - round component = 12
    id = intermediate_output_4_12
    type = intermediate_output
    input_bit_size = 32
    input_id_link = ['xor_4_8', 'xor_4_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 32
    description = ['round_output']

    # round = 5 - round component = 0
    id = constant_5_0
    type = constant
    input_bit_size = 0
    input_id_link = ['']
    input_bit_positions = [[]]
    output_bit_size = 16
    description = ['0x0004']

    # round = 5 - round component = 1
    id = rot_5_1
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_2_3']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]

    # round = 5 - round component = 2
    id = modadd_5_2
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_5_1', 'xor_4_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 5 - round component = 3
    id = xor_5_3
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_5_2', 'constant_5_0']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 5 - round component = 4
    id = rot_5_4
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_4_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 5 - round component = 5
    id = xor_5_5
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_5_3', 'rot_5_4']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 5 - round component = 6
    id = rot_5_6
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_4_8']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', 7]
    # round = 5 - round component = 7
    id = modadd_5_7
    type = word_operation
    input_bit_size = 32
    input_id_link = ['rot_5_6', 'xor_4_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['MODADD', 2, None]

    # round = 5 - round component = 8
    id = xor_5_8
    type = word_operation
    input_bit_size = 32
    input_id_link = ['modadd_5_7', 'xor_5_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 5 - round component = 9
    id = rot_5_9
    type = word_operation
    input_bit_size = 16
    input_id_link = ['xor_4_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['ROTATE', -2]

    # round = 5 - round component = 10
    id = xor_5_10
    type = word_operation
    input_bit_size = 32
    input_id_link = ['xor_5_8', 'rot_5_9']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 16
    description = ['XOR', 2]

    # round = 5 - round component = 11
    id = intermediate_output_5_11
    type = intermediate_output
    input_bit_size = 16
    input_id_link = ['xor_5_5']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15]]
    output_bit_size = 16
    description = ['round_key_output']

    # round = 5 - round component = 12
    id = cipher_output_5_12
    type = cipher_output
    input_bit_size = 32
    input_id_link = ['xor_5_8', 'xor_5_10']
    input_bit_positions = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
    output_bit_size = 32
    description = ['cipher_output']
cipher_reference_code = None
```

2.2 Mapping which rounds will be top part, middle part, and linear part

[2]:
```
top_part_components = []
for round_number in range(0, 2):
    top_part_components += speck.get_components_in_round(round_number)
middle_part_components = []
for round_number in range(2, 3):
    middle_part_components += speck.get_components_in_round(round_number)
bottom_part_components = []
for round_number in range(3, 6):
    bottom_part_components += speck.get_components_in_round(round_number)

top_part_component_ids = [component.id for component in top_part_components]␣
  ↪#ids
middle_part_component_ids = [component.id for component in␣
  ↪middle_part_components]
bottom_part_component_ids = [component.id for  component in␣
  ↪bottom_part_components]
print("Top Part Components:", top_part_component_ids)
print()
print("Middle Part Components:", middle_part_component_ids)
print()
print("Bottom Part Components:", bottom_part_component_ids)
```

```
Top Part Components: ['rot_0_0', 'modadd_0_1', 'xor_0_2', 'rot_0_3', 'xor_0_4',
'intermediate_output_0_5', 'intermediate_output_0_6', 'constant_1_0', 'rot_1_1',
'modadd_1_2', 'xor_1_3', 'rot_1_4', 'xor_1_5', 'rot_1_6', 'modadd_1_7',
'xor_1_8', 'rot_1_9', 'xor_1_10', 'intermediate_output_1_11',
'intermediate_output_1_12']

Middle Part Components: ['constant_2_0', 'rot_2_1', 'modadd_2_2', 'xor_2_3',
'rot_2_4', 'xor_2_5', 'rot_2_6', 'modadd_2_7', 'xor_2_8', 'rot_2_9', 'xor_2_10',
'intermediate_output_2_11', 'intermediate_output_2_12']

Bottom Part Components: ['constant_3_0', 'rot_3_1', 'modadd_3_2', 'xor_3_3',
'rot_3_4', 'xor_3_5', 'rot_3_6', 'modadd_3_7', 'xor_3_8', 'rot_3_9', 'xor_3_10',
'intermediate_output_3_11', 'intermediate_output_3_12', 'constant_4_0',
'rot_4_1', 'modadd_4_2', 'xor_4_3', 'rot_4_4', 'xor_4_5', 'rot_4_6',
'modadd_4_7', 'xor_4_8', 'rot_4_9', 'xor_4_10', 'intermediate_output_4_11',
'intermediate_output_4_12', 'constant_5_0', 'rot_5_1', 'modadd_5_2', 'xor_5_3',
'rot_5_4', 'xor_5_5', 'rot_5_6', 'modadd_5_7', 'xor_5_8', 'rot_5_9', 'xor_5_10',
'intermediate_output_5_11', 'cipher_output_5_12']
```

2.3 Creating the Differential-Linear Model

[3]:
```
from claasp.cipher_modules.models.sat.sat_models.sat_differential_linear_model␣
 ↪import SatDifferentialLinearModel
component_model_list = {
    'middle_part_components': middle_part_component_ids,
    'bottom_part_components': bottom_part_component_ids
}
sat_differential_linear_model = SatDifferentialLinearModel(speck,␣
 ↪component_model_list)
```

2.4 Finding one Differential-Linear Trail on Speck

[4]:
```
from claasp.cipher_modules.models.utils import set_fixed_variables

plaintext_difference = set_fixed_variables(
    component_id='plaintext',
    constraint_type='not_equal',
    bit_positions=range(32),
    bit_values=(0,) * 32
    )
key_difference = set_fixed_variables(
    component_id='key',
    constraint_type='equal',
    bit_positions=range(64),
    bit_values=(0,) * 64
    )

ciphertext_output_mask = set_fixed_variables(
    component_id='cipher_output_5_12',
    constraint_type='not_equal',
    bit_positions=range(32),
    bit_values=(0,) * 32
)

trail = sat_differential_linear_model.
 ↪find_lowest_weight_xor_differential_linear_trail(
    fixed_values=[key_difference, plaintext_difference, ciphertext_output_mask],
    solver_name="CADICAL_EXT",
    num_unknown_vars=10# explain in terms of mask
)
import pprint
pprint.pprint(trail)
```

```
{'cipher': speck_p32_k64_o32_r6,
 'components_values': {'cipher_output_5_12': {'value': '02040205', 'weight': 0},
                       'constant_1_0': {'value': '0000', 'weight': 0},
                       'constant_2_0': {'value': '0000000000000000',
                                        'weight': 0},
                       'constant_3_0': {'value': '0001', 'weight': 0},
                       'constant_4_0': {'value': '0001', 'weight': 0},
                       'constant_5_0': {'value': '0001', 'weight': 0},
                       'intermediate_output_0_5': {'value': '0000',
                                                   'weight': 0},
                       'intermediate_output_0_6': {'value': '00081000',
                                                   'weight': 0},
                       'intermediate_output_1_11': {'value': '0000',
                                                    'weight': 0},
                       'intermediate_output_1_12': {'value': '00004000',
                                                    'weight': 0},
                       'intermediate_output_2_11': {'value': '0000000000000000',
                                                    'weight': 0},
                       'intermediate_output_2_12': {'value':
'?100000000000000?100000000000001',
                                                    'weight': 0},
                       'intermediate_output_3_11': {'value': '4001',
                                                    'weight': 0},
                       'intermediate_output_3_12': {'value': '00204020',
                                                    'weight': 0},
                       'intermediate_output_4_11': {'value': '4001',
                                                    'weight': 0},
                       'intermediate_output_4_12': {'value': '00804080',
                                                    'weight': 0},
                       'intermediate_output_5_11': {'value': '0000',
                                                    'weight': 0},
                       'key': {'value': '0000000000000000'},
                       'modadd_0_1': {'value': '0008', 'weight': 3},
                       'modadd_1_2': {'value': '0000', 'weight': 0},
                       'modadd_1_7': {'value': '0000', 'weight': 1},
                       'modadd_2_2': {'value': '0000000000000000', 'weight': 0},
                       'modadd_2_7': {'value': '?100000000000000', 'weight': 0},
                       'modadd_3_2': {'value': '0001', 'weight': 0},
                       'modadd_3_7': {'value': '4000', 'weight': 1},
                       'modadd_4_2': {'value': '0001', 'weight': 0},
                       'modadd_4_7': {'value': '4000', 'weight': 2},
                       'modadd_5_2': {'value': '0001', 'weight': 0},
                       'modadd_5_7': {'value': '0001', 'weight': 0},
                       'plaintext': {'value': '05020402'},
                       'rot_0_0': {'value': '040a', 'weight': 0},
                       'rot_0_3': {'value': '1008', 'weight': 0},
                       'rot_1_1': {'value': '0000', 'weight': 0},
                       'rot_1_4': {'value': '0000', 'weight': 0},
                       'rot_1_6': {'value': '1000', 'weight': 0},
                       'rot_1_9': {'value': '4000', 'weight': 0},
                       'rot_2_1': {'value': '0000000000000000', 'weight': 0},
                       'rot_2_4': {'value': '0000000000000000', 'weight': 0},
                       'rot_2_6': {'value': '0000000000000000', 'weight': 0},
                       'rot_2_9': {'value': '0000000000000001', 'weight': 0},
                       'rot_3_1': {'value': '0001', 'weight': 0},
                       'rot_3_4': {'value': '0001', 'weight': 0},
                       'rot_3_6': {'value': '6000', 'weight': 0},
                       'rot_3_9': {'value': '4020', 'weight': 0},
                       'rot_4_1': {'value': '0001', 'weight': 0},
                       'rot_4_4': {'value': '0001', 'weight': 0},
                       'rot_4_6': {'value': '4000', 'weight': 0},
                       'rot_4_9': {'value': '4080', 'weight': 0},
                       'rot_5_1': {'value': '0001', 'weight': 0},
                       'rot_5_4': {'value': '0001', 'weight': 0},
                       'rot_5_6': {'value': '0001', 'weight': 0},
                       'rot_5_9': {'value': '0205', 'weight': 0},
                       'xor_0_2': {'value': '0008', 'weight': 0},
                       'xor_0_4': {'value': '1000', 'weight': 0},
                       'xor_1_10': {'value': '4000', 'weight': 0},
                       'xor_1_3': {'value': '0000', 'weight': 0},
                       'xor_1_5': {'value': '0000', 'weight': 0},
                       'xor_1_8': {'value': '0000', 'weight': 0},
                       'xor_2_10': {'value': '?100000000000001', 'weight': 0},
                       'xor_2_3': {'value': '0000000000000000', 'weight': 0},
                       'xor_2_5': {'value': '0000000000000000', 'weight': 0},
                       'xor_2_8': {'value': '?100000000000000', 'weight': 0},
                       'xor_3_10': {'value': '4020', 'weight': 0},
                       'xor_3_3': {'value': '0001', 'weight': 0},
                       'xor_3_5': {'value': '0001', 'weight': 0},
                       'xor_3_8': {'value': '4000', 'weight': 0},
                       'xor_4_10': {'value': '4080', 'weight': 0},
                       'xor_4_3': {'value': '0001', 'weight': 0},
                       'xor_4_5': {'value': '0001', 'weight': 0},
                       'xor_4_8': {'value': '4000', 'weight': 0},
                       'xor_5_10': {'value': '0205', 'weight': 0},
                       'xor_5_3': {'value': '0001', 'weight': 0},
                       'xor_5_5': {'value': '0001', 'weight': 0},
                       'xor_5_8': {'value': '0001', 'weight': 0}},
 'memory_megabytes': 262.4,
 'model_type': 'XOR_DIFFERENTIAL_LINEAR_MODEL',
 'solver_name': 'CADICAL_EXT',
 'solving_time_seconds': 0.5,
 'status': 'SATISFIABLE',
 'test_name': 'find_lowest_weight_differential_linear_trail',
 'total_weight': 10.0}
```

2.5 Printing the distinguisher

[5]:
```
input_difference_str = trail['components_values']['plaintext']['value']
output_mask_str = trail['components_values']['cipher_output_5_12']['value']
print("Input difference:", input_difference_str)
print("Output mask:", output_mask_str)
```

```
Input difference: 05020402
Output mask: 02040205
```

[6]:
```
from claasp.cipher_modules.report import Report
report = Report(trail)
report.show()
```

```
_ _ _ _ _ 1 _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _
1 _ _ _ _ _ _ _ _ 1 _    plaintext

_ _ _ _ _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ _
_ _ _    intermediate_output_0_6

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ _ _ _ _ _ _
intermediate_output_1_12

? 1 _ _ _ _ _ _ _ _ _ _ _ _ _ _ ? 1 _ _ _ _ _ _ _ _ _ _ _
_ _ 1    intermediate_output_2_12

_ _ _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _
1 _ _ _ _ _    intermediate_output_3_12

_ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _
1 _ _ _ _ _ _ _    intermediate_output_4_12

_ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ 1 _
_ _ _ _ _ 1 _ 1    cipher_output_5_12


total weight = 10.0

KEY FLOW


_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _         key
```

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _   intermediate_output_0_5

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _   intermediate_output_1_11

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _   intermediate_output_2_11

_ 1 _ _ _ _ _ _ _ _ _ _ _ _ _ 1   intermediate_output_3_11

_ 1 _ _ _ _ _ _ _ _ _ _ _ _ _ 1   intermediate_output_4_11

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _   intermediate_output_5_11
```

2.6 Checking the distinguisher

[7]:
```python
from claasp.cipher_modules.models.utils import␣
 ↪differential_linear_checker_for_block_cipher_single_key

input_difference = int(input_difference_str, 16)
output_mask =  int(output_mask_str, 16)
key_difference = 0x1
number_of_samples = 2 ** 14
corr = differential_linear_checker_for_block_cipher_single_key(
    speck,
    input_difference,
    output_mask,
    number_of_samples,
    block_size=32,
    key_size=64,
    fixed_key=key_difference
)
import math

abs_corr = abs(corr)
print("Correlation:", abs(math.log(abs_corr, 2)))
```

```
Correlation: 4.400087157812872
```

[ ]:
```

```

tutorial_division_property

March 19, 2025

# Three-Subset Bit-Based Division Property (Without Unknown Subset)

The division property module of CLAASP can automatically generate a model of the Three-Subset Division Property for all block ciphers and permutations available in CLAASP. The outline of this presentation is: 1. Analyze a toy SPN cipher using this variant of divsion property Given the number of rounds for a selected cipher and a chosen output bit, this module produces a model that can: - Obtain the Algebraic Normal Form (ANF) of the chosen output bit, - Find an upper bound for the degree of the ANF of the chosen output bit, or - Check the presence or absence of a specified monomial.

2. Based on these 3 features, one can mount:

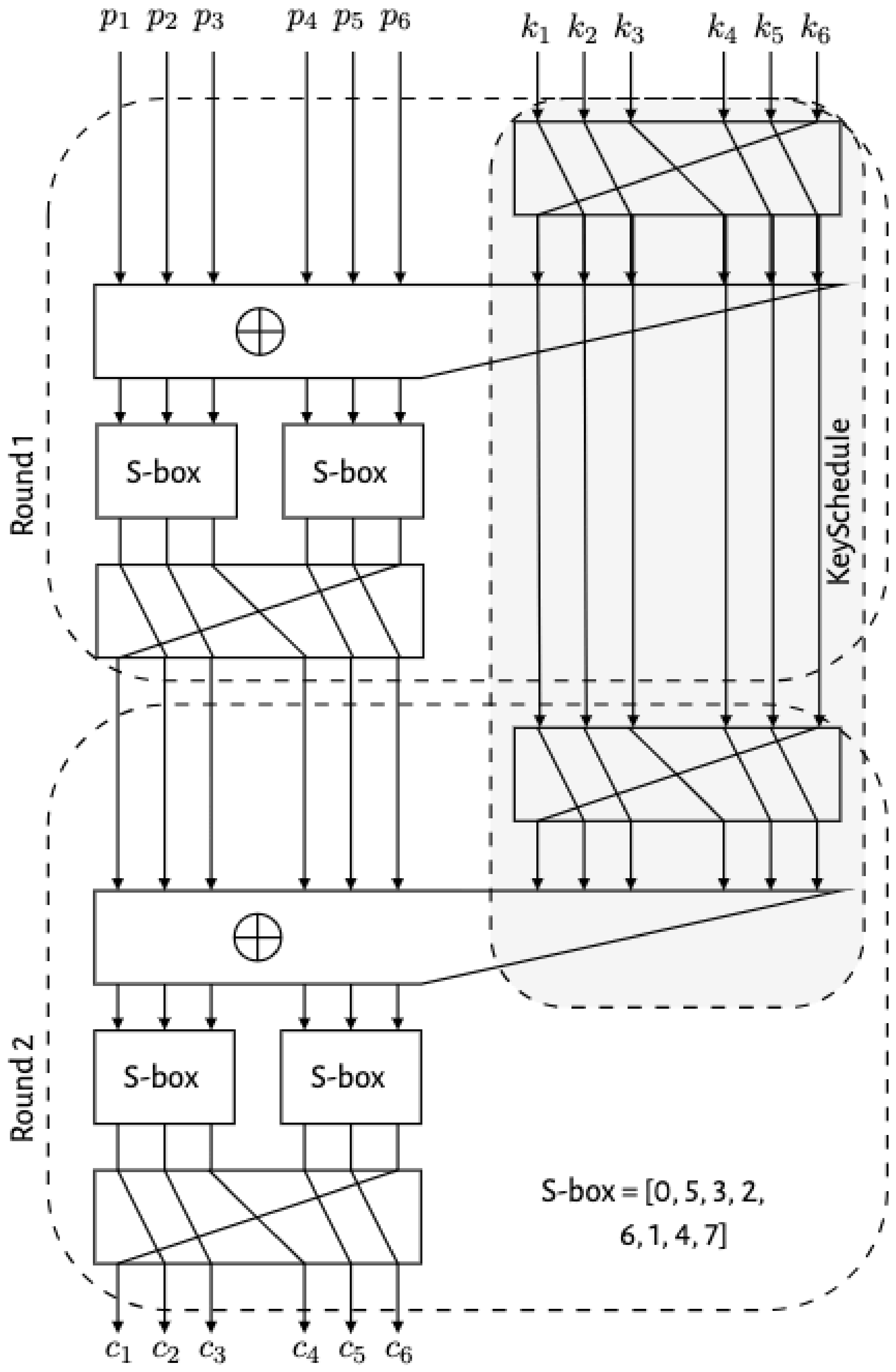
- Cube attacks by recovering the superpoly, or
- Integral distinguishers based on the degree.

In CLAASP, a symmetric cipher is a Python class that can be represented as a list of "connected components". By the term component, we refer to the building blocks of symmetric ciphers (S-Boxes, linear layers, word operations, etc.). The model generation process involves the following steps: - Gurobi variables are created for the cipher's inputs and outputs. - Gurobi variables are also created for all input and output bits of each component. - The module loops over the components and adds constraints based on their underlying operations, specifically modeling the COPY, XOR, and AND operations as defined in the Three-Subset Division Property. - Constraints are added to link the output of each component to the input of its dependent components.

For details on the modeling of these basic components, refer to the publication: https://eprint.iacr.org/2020/441

0.1 Overview on a Toy cipher

The cipher diagram of ToySPN2 is reported below:


p_1 p_2 p_3 p_4 p_5 p_6
k_1 k_2 k_3 k_4 k_5 k_6
$\oplus$
S-box
S-box
Round 1
KeySchedule
$\oplus$
S-box
S-box
Round 2
S-box = [0, 5, 3, 2, 6, 1, 4, 7]
c_1 c_2 c_3 c_4 c_5 c_6

```
[ ]: from claasp.ciphers.toys.toyspn2 import ToySPN2
     cipher = ToySPN2()
     cipher.print()
```

0.1.1 How to obtain the ANF of an output bit:

```
[4]: from claasp.cipher_modules.division_trail_search import MilpDivisionTrailModel
     from IPython.display import display, Markdown
     import os, sys, io
     os.environ["GUROBI_COMPUTE_SERVER"] = "10.191.12.120"
     milp = MilpDivisionTrailModel(cipher)
     old_stdout = sys.stdout
     sys.stdout = io.StringIO()
     # Retrieving c_1 ANF
     anf_output_bit_0 = milp.find_anf_of_specific_output_bit(0)
     anf_output_bit_0
```

```
[4]: ['p3p5', 'p0p1p3', 'p0p3k0', 'p1p3k5', 'p2p3', 'p0p1p4p5', 'p0p4p5k0', 'p4',
     'p0p1p4k4', 'p0p4k0k4', 'p2p4p5', 'p0p1p5', 'p1p4p5k5', 'p0p5k0', 'p0k0k4',
     'p0p1k4', 'p5', 'p2p4k4', 'p1p4k4k5', 'p4p5', 'p2p5', 'p1p5k5', 'p2k4',
     'p0p5k0k3', 'p0p1p5k3', 'p0p1k3k4', 'p0k0k3k4', 'p1k4k5', 'p5k2', 'p2k3k4',
     'p2p5k3', 'k4', 'p1k3k4k5', 'k0k3k4k5', 'p5k3', 'p1k4', 'p3k0k5', 'p1p5k3k5',
     'p5k0k5', 'p4k0k4k5', 'k0k4', 'k0k4k5', 'p5k0', 'p1k3k4', 'p4k0k4', 'k0k3k4',
     'p5k0k3k5', 'p1p3', 'p1p4p5', 'p3k0', 'p1p5k3', 'p1p4k4', 'p1p5', 'p4k4',
     'p5k0k3', 'k3k4', 'p3k4', 'k2', 'p4p5k0', 'p4p5k0k5', 'k2k4']
```

0.1.2 Showing the ANF using SAGE

```
[5]: from sage.all import *
     var_names = ['p0','p1','p2','p3','p4','p5','k0','k1','k2','k3','k4','k5']
     R = PolynomialRing(GF(2), names=var_names, order='lex')
     p0, p1, p2, p3, p4, p5, k0, k1, k2, k3, k4, k5 = R.gens()
     import re

     def parse_monomial(m_str):
         tokens = re.findall(r'(p\d+|k\d+)', m_str)
         poly = R(1)
         for var_name in tokens:
             poly *= R(var_name)
         return poly

     def build_anf_from_strings(mon_list):
         return sum(parse_monomial(m) for m in mon_list)

     anf_output_bit_0_sage = build_anf_from_strings(anf_output_bit_0)
     display(Markdown(f"anf_output_bit_0_sage = ${anf_output_bit_0_sage}$"))
```

anf_output_bit_0_sage $= p0*p1*p3+p0*p1*p4*p5+p0*p1*p4*k4+p0*p1*p5*k3+p0*p1*p5+p0*$

$p1*k3*k4+p0*p1*k4+p0*p3*k0+p0*p4*p5*k0+p0*p4*k0*k4+p0*p5*k0*k3+p0*p5*k0+p0*k0*k3*k4+p0*k0*k4+p1*p3*k5+p1*p3+p1*p4*p5*k5+p1*p4*p5+p1*p4*k4*k5+p1*p4*k4+p1*p5*k3*k5+p1*p5*k3+p1*p5*k5+p1*p5+p1*k3*k4*k5+p1*k3*k4+p1*k4*k5+p1*k4+p2*p3+p2*p4*p5+p2*p4*k4+p2*p5*k3+p2*p5+p2*k3*k4+p2*k4+p3*p5+p3*k0*k5+p3*k0+p3*k4+p4*p5*k0*k5+p4*p5*k0+p4*p5+p4*k0*k4*k5+p4*k0*k4+p4*k4+p4+p5*k0*k3*k5+p5*k0*k3+p5*k0*k5+p5*k0+p5*k2+p5*k3+p5+k0*k3*k4*k5+k0*k3*k4+k0*k4*k5+k0*k4+k2*k4+k2+k3*k4+k4$

0.1.3 How to obtain the monomials of certain degree in the ANF of an output bit:

[6]:
```
cipher = ToySPN2()
milp = MilpDivisionTrailModel(cipher)
anf = milp.find_anf_of_specific_output_bit(0, fixed_degree=2)
anf
```

[6]:
```
['p1p5k3k5', 'p2p5k3', 'p0p3k0', 'p0p1k4', 'p2p5', 'p1p5k5', 'p0p1k3k4', 'p4p5',
'p1p4k4k5', 'p0p4k0k4', 'p0p5k0k3', 'p2p3', 'p0p5k0', 'p2p4k4', 'p1p3k5',
'p1p5k3', 'p3p5', 'p1p4k4', 'p1p5', 'p4p5k0k5', 'p1p3', 'p4p5k0']
```

0.1.4 How to find an upper bound for the degree of the ANF of an output bit:

[7]:
```
cipher = ToySPN2()
milp = MilpDivisionTrailModel(cipher)
degree = milp.find_degree_of_specific_output_bit(0)
degree
```

[7]:
```
4.0
```

0.1.5 How to obtain the ANF of an intermediate component:

[8]:
```
cipher = ToySPN2()
cipher.get_all_components_ids()
```

[8]:
```
['rot_0_0', 'intermediate_output_0_1', 'xor_0_2', 'sbox_0_3', 'sbox_0_4',
'rot_0_5', 'intermediate_output_0_6', 'rot_1_0', 'intermediate_output_1_1',
'xor_1_2', 'sbox_1_3', 'sbox_1_4', 'rot_1_5', 'intermediate_output_1_6',
'cipher_output_1_7']
```

[9]:
```
cipher = ToySPN2()
milp = MilpDivisionTrailModel(cipher)
anf = milp.find_anf_of_specific_output_bit(0, chosen_cipher_output='sbox_0_3')
anf
```

[9]:
```
['p2', 'p1k1', 'p0', 'k5', 'p1p2', 'p2k0', 'k0k1', 'k1']
```

0.2 Cube attack on ToySPN

The cube attack was proposed by Dinur and Shamir at EUROCRYPT 2009.

Let $f(x)$ be a Boolean function from $\mathbb{F}_2^n$ to $\mathbb{F}_2$, and $u \in \mathbb{F}_2^n$ be a constant vector.
Then $f(x)$ can be represented uniquely as:

$$f(x) = x^u p(x) + q(x)$$

where $x^u = x_0^{u_0} x_1^{u_1} \cdots x_{n-1}^{u_{n-1}}$ and each monomials of $q(x)$ is not divisible by x^u. If we compute the sum of f over the cube C_u, we have:

$$\sum_{x \in C_u} f(x) = \sum_{x \in C_u} (x^u p(x) + q(x)) = p(x)$$

We call $p(x)$ the superpoly of the cube C_u.

0.2.1 Finding $x^u p(x)$ with $u = (0, 0, 1, 0, 0, 1)$

The method `check_presence_of_particular_monomial_in_specific_anf` allow us to find all monomials that are multiple of the monomial given as input, namely p2p5. A simple factorisation can then give us a superpoly.

In our example, we are working over the ANF of the first output bit of the ToySPN2 cipher from $\mathbb{F}_2^6$ to $\mathbb{F}_2$.
We can represent this ANF as:

$$f(x) = p_2 p_5 \cdot (1 + p_4 + k_3) + q(x)$$

$p(x) = 1 + p_4 + k_3$ is the superpoly of the cube composed by the monomial p_2 and p_5.

Therefore,

$$\sum_{x \in C_u} f(x) = 1 + p_4 + k_3$$

[10]:
```
cipher = ToySPN2()
milp = MilpDivisionTrailModel(cipher)
firstterm = milp.
  ↪check_presence_of_particular_monomial_in_specific_anf([("plaintext", 2),␣
  ↪("plaintext", 5)], 0)
firstterm_sage = build_anf_from_strings(firstterm)
q = anf_output_bit_0_sage-firstterm_sage
display(Markdown(f'$x^up(x) = {firstterm_sage}$'))
display(Markdown(f'$q(x) = {q}$'))
```

$x^u p(x) = p2*p4*p5 + p2*p5*k3 + p2*p5$

$q(x) = p0*p1*p3+p0*p1*p4*p5+p0*p1*p4*k4+p0*p1*p5*k3+p0*p1*p5+p0*p1*k3*k4+p0*p1*k4+p0*p3*k0+p0*p4*p5*k0+p0*p4*k0*k4+p0*p5*k0*k3+p0*p5*k0+p0*k0*k3*k4+p0*k0*k4+p1*p3*k5+p1*p3+p1*p4*p5*k5+p1*p4*p5+p1*p4*k4*k5+p1*p4*k4+p1*p5*k3*k5+p1*p5*k3+p1*p5*k5+p1*p5+p1*k3*k4*k5+p1*k3*k4+p1*k4*k5+p1*k4+p2*$

$p3 + p2*p4*k4 + p2*k3*k4 + p2*k4 + p3*p5 + p3*k0*k5 + p3*k0 + p3*k4 + p4*p5*k0*k5 + p4*p5*k0 + p4*p5 + p4*k0*k4*k5 + p4*k0*k4 + p4*k4 + p4 + p5*k0*k3*k5 + p5*k0*k3 + p5*k0*k5 + p5*k0 + p5*k2 + p5*k3 + p5 + k0*k3*k4*k5 + k0*k3*k4 + k0*k4*k5 + k0*k4 + k2*k4 + k2 + k3*k4 + k4$

0.2.2 Computing C_u with $u = (0, 0, 1, 0, 0, 1)$

[11]:
```
secret_key = 0b101100
C_u = [0b000000, 0b001000, 0b000001, 0b001001]
ciphertexts = [format(cipher.evaluate([v, secret_key]), '06b') for v in C_u]
display(Markdown(f'$C_u$={C_u}'))
display(Markdown(f'ToySpn $(C_u)$={ciphertexts}'))
```

C_u=[0, 8, 1, 9]

ToySpn (C_u)=['100000', '100010', '011011', '011000']

0.2.3 Checking that when summing over C_u then $q(x) = 0$

[12]:
```
sum_over_cube = R(0)
for v in C_u:
    subs_dict = {
        p0: (v >> 5) & 1,
        p1: (v >> 4) & 1,
        p2: (v >> 3) & 1,
        p3: (v >> 2) & 1,
        p4: (v >> 1) & 1,
        p5: (v >> 0) & 1
    }
    q_eval = q.subs(subs_dict)
    sum_over_cube += q_eval

display(Markdown(f'Sum of q over the cube $C_u$ on $q(x)$ is {sum_over_cube}'))
```

Sum of q over the cube C_u on $q(x)$ is 0 and x_u=1

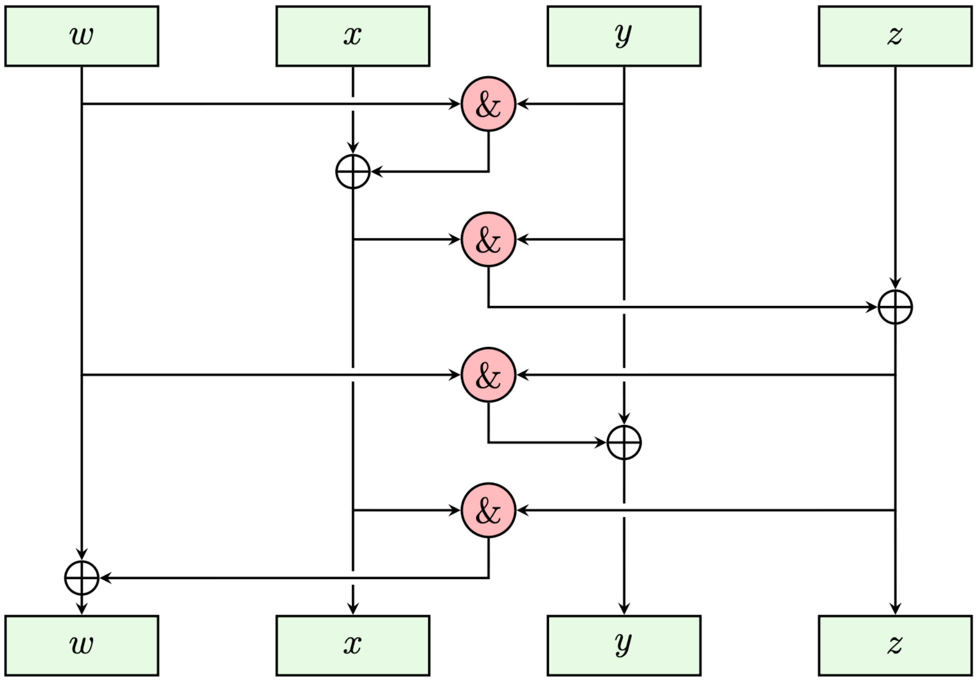
Therefore, we obtain a linear equation on the key bit:

$$1 + p_4 + k_3 = 0$$

Since $p_4 = 0$, this directly reveals the value of k_3.

0.3 Integral Distinguisher on Aradi

0.3.1 Aradi Cipher



0.3.2 Integral cryptanalysis

Integral cryptanalysis is a type of attack that exploits the properties of integrals (sums) over sets of plaintexts. An output bit of a block cipher is said to be balanced over a set of n chosen plaintexts if the XOR of all corresponding ciphertexts for that bit equals zero. One approach to identifying a balanced output bit is by analyzing the degree of its underlying Algebraic Normal Form (ANF). If the degree of an output bit is d, then the XOR sum of that bit over any affine subspace of dimension $d+1$, containing 2^{d+1} inputs, will be zero.

In the following table, we give the degree bounds for the monomials corresponding to the cube-index sets in the second column, for up to 8 rounds. We refer to the paper https://eprint.iacr.org/2024/1559.pdf for more details.

Round r	**Indices**	**Degree in W_i^r, Y_i^r**		**Degree in X_i^r, Z_i^r**		**Cube dimension**
		min	**max**	**min**	**max**	
4	$I_W = \{0\}, I_X = \{0\}$ $I_Y = \{0\}, I_Z = \{0\}$	3	4	3	4	4
5	$I_W = I_X = \{0, \ldots, 4\}$ $I_Y = I_Z = \{0, \ldots, 4\}$	15	16	15	16	16
6	$I_W = I_X = \{0, \ldots, 22\}$ $I_Y = I_Z = \{0, \ldots, 22\}$	92	92	90	90	92
7	$I_W = I_X = \{0, \ldots, 28\}$ $I_Y = I_Z = \{0, \ldots, 28\}$	116	116	115	115	116
8	$I_W = I_X = \{0, \ldots, 30\}$ $I_Y = I_Z = \{0, \ldots, 30\}$	124	124	123	123	124

[13]:
```python
from claasp.ciphers.block_ciphers.aradi_block_cipher import AradiBlockCipher
from claasp.cipher_modules.division_trail_search import MilpDivisionTrailModel
```

[14]:
```python
import time
start = time.time()
cipher = AradiBlockCipher(number_of_rounds=4)
milp = MilpDivisionTrailModel(cipher)
for output_bit_position in range(2):
    degree = milp.find_degree_of_specific_output_bit(output_bit_position,␣
 ↪cube_index=[0,32,64,96])
    display(Markdown(f"output_bit_position = {output_bit_position}, degree =␣
 ↪{degree}"))
    degree
end = time.time()
solving_time = end - start
display(Markdown((f"solving_time : {solving_time}")))
```

output_bit_position = 0, degree = 4.0

output_bit_position = 1, degree = 3.0

solving_time : 64.2635247707367

[15]:
```python
start = time.time()
cipher = AradiBlockCipher(number_of_rounds=6)
milp = MilpDivisionTrailModel(cipher)
cube = sum([list(range(i, i + 23)) for i in [0, 32, 64, 96]], [])
sys.stdout = io.StringIO()
degree = milp.find_degree_of_specific_output_bit(96, cube_index=cube)
sys.stdout = old_stdout
display(Markdown(f"degree = {degree}"))
end = time.time()
solving_time = end - start
display(Markdown((f"solving_time : {solving_time}")))
```

```
degree = 90.0

solving_time : 27.604275941848755
```

```
[ ]:
```