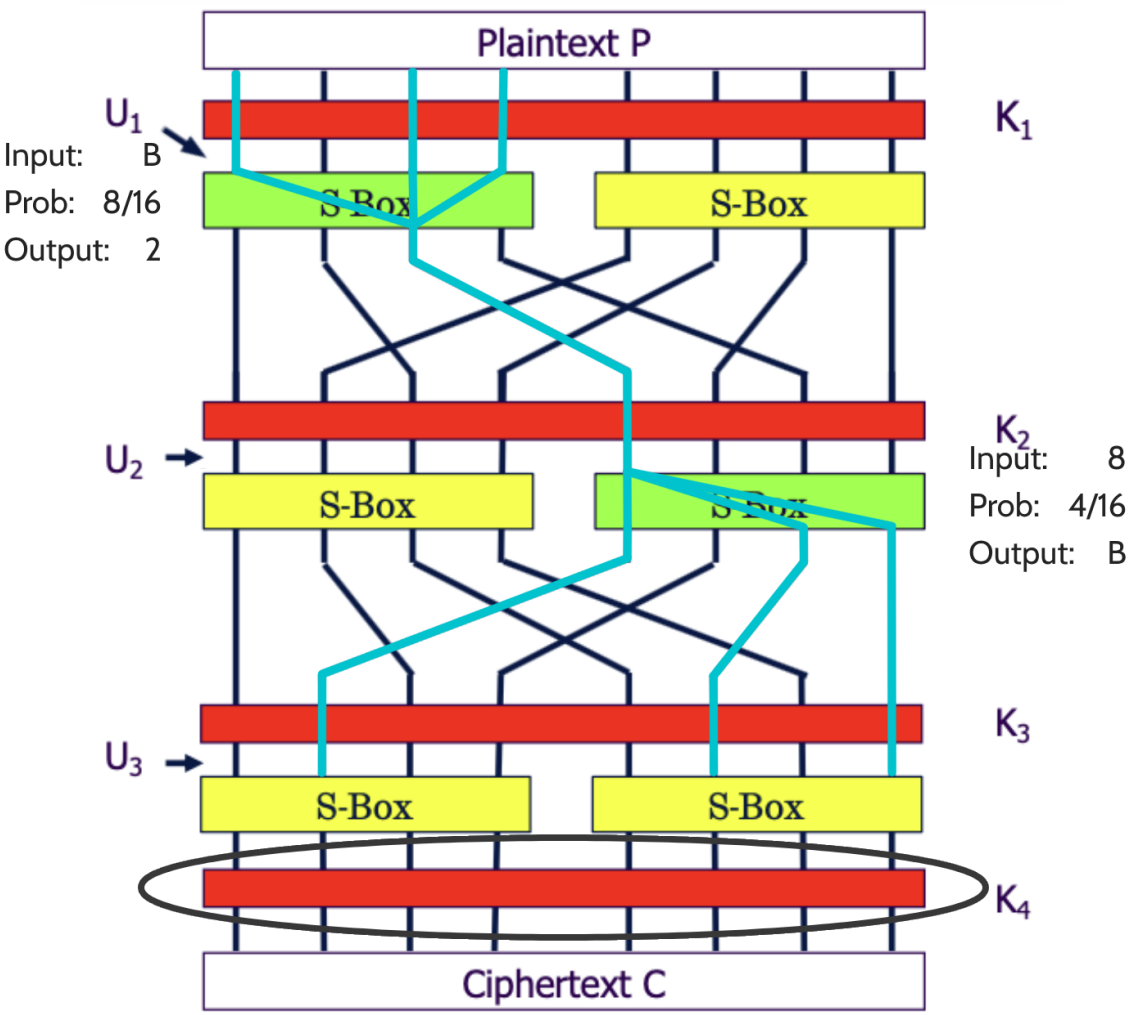
Cryptanalysis Report of
2-Round Differential Approximation

Name: Yan Kong

Introduction of the 2-round differential approximation

Combining S-box difference pairs from round to round enables us to find a high probability differential consisting of the plaintext difference ΔP and the difference of the input ΔU_3 to the last round. The key bits of the cipher end up disappearing from the difference expression because they are involved in both data sets.



Identify the active S-boxes and associated probability of each such S-box approximation

As shown in the graph, the active S-boxes are the left S-box in the first round (denoted as 1st S-box) and the right one in the second round (denoted as 2nd S-box). Associated probability are derived from the Difference Distribution Table in Heys Tutorial.
1st S-box {input difference : B, output difference : 2, associated probability : 8/16}
2nd Sbox {input difference : 8, output difference : B, associated probability : 4/16}

		Output Difference															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Input Difference	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
	2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
	3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
	4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
	5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
	6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
	7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
	8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
	9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
	A	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
	B	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
	C	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
	D	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
	E	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
	F	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

Calculate the probability for the 2-round differential approximation

We assume differentials at each stage occur independently, hence the probability of the 2-round differential approximation is determined by the product of the probabilities : 8/16 * 4/16 = 1/8

Extracting key bits K_4

The entire 8 bits of K_4 are targeted by the 2-round approximation, because both S-boxes in the last round are influenced by non-zero differences in the differential output.

Principle of constructing the 2-round approximation

In general, the larger the differential probabilities of the active S-boxes, the larger the characteristic probability for the complete cipher. Also, the fewer active S-boxes, the larger the characteristic probability. Therefore, the principle is to choose a differential approximation with high probability while minimising the total number of S-boxes.
In the 1st round approximation, pick the differential approximation based on the input and output difference pairs which gives the highest probability.
In the 2nd round approximation, since the input is determined, choose the corresponding output difference which gives the highest probability.

Implementing the cipher

To see the complete code and the output all together, please view in google colab via the link below:
2-Round Differential Approximation.ipynb

```
  def sBox(U): # define S-box, use dict for faster data processing
      sBox_dict = {0:14, 1:4, 2:13, 3:1, 4:2, 5:15, 6:11, 7:8, 8:3, 9:10, 10:6, 11:12,
12:5, 13:9, 14:0, 15:7}
      return sBox_dict[U]

  def encrypt(U, Key): # simply XOR
      return U ^ Key

  def substitute(U): # apply S-box for each half then form a new substitution together
      U1 = (U >> 4) & 0b1111
      U2 = U & 0b1111
      new_substitution = (sBox(U1) << 4) | sBox(U2)
      return new_substitution

  def encrypt_and_substitute(U, Key): # combine encrypt and substitute for conciseness
      U = encrypt(U, Key)
      new_substitution = substitute(U)
      return new_substitution

  def permute(U): # function to apply the permutation based on Heys Tutorial
      permutation_order = [0, 2, 4, 6, 1, 3, 5, 7]
      new_permutation = 0
      for i, new_position in enumerate(permutation_order):
          bit_value = (U >> (7 - i)) & 1
          new_permutation |= (bit_value << (7 - new_position))
      return new_permutation

  # function to generate cipher using plaintext and given keys
  def generate_cipher(P, K1 = 0x54, K2 = 0x32, K3 = 0x23, K4 = 0x45):
      V1 = encrypt_and_substitute(P, K1)
      U2 = permute(V1)
      V2 = encrypt_and_substitute(U2, K2)
      U3 = permute(V2)
      V3 = encrypt_and_substitute(U3, K3)
      C = encrypt(V3, K4)
      return C

  # function to generate plaintexts pairs with specific delta
  def generate_plaintexts_pairs(delta_P):
      plaintexts_pairs = set()
      for i in range(256):
          plaintext1 = i
          plaintext2 = i ^ delta_P
          pair = tuple(sorted([plaintext1, plaintext2]))  # Ensure each pair is unique
          plaintexts_pairs.add(pair)
```

```
    plaintexts_pairs = list(plaintexts_pairs) # Convert to list for convenience
    return plaintexts_pairs

# function to generate ciphers pairs according to given plaintexts pairs
def generate_ciphers_pairs(plaintexts_pairs):
    ciphers_pairs = []
    for plaintexts_pair in plaintexts_pairs:
        cipher_pair = (generate_cipher(plaintexts_pair[0]),
                       generate_cipher(plaintexts_pair[1]))
        ciphers_pairs.append(cipher_pair)
    return ciphers_pairs

# generate plaintext pairs satisfying ΔP = 0b10110000
plaintexts_pairs = generate_plaintexts_pairs(delta_P = 0b10110000)
print(plaintexts_pairs[:64])

# generate corresponding ciphertext pairs
ciphers_pairs = generate_ciphers_pairs(plaintexts_pairs)
print(ciphers_pairs [:64])
```

```
# Print Outputs
# first 64 plaintexts pairs
[(94, 238), (53, 133), (30, 174), (97, 209), (25, 169), (1, 177), (90, 234), (127, 207),
(21, 165), (31, 175), (86, 230), (123, 203), (27, 171), (119, 199), (59, 139), (124,
204), (2, 178), (60, 140), (3, 179), (35, 147), (92, 236), (112, 192), (93, 237), (88,
232), (36, 148), (125, 205), (68, 244), (29, 173), (84, 228), (126, 206), (32, 144),
(121, 201), (4, 180), (69, 245), (62, 142), (101, 213), (117, 197), (5, 181), (0, 176),
(10, 186), (58, 138), (65, 241), (37, 149), (102, 214), (6, 182), (95, 239), (33, 145),
(43, 155), (98, 210), (91, 235), (70, 246), (63, 143), (39, 151), (34, 146), (11, 187),
(66, 242), (76, 252), (7, 183), (72, 248), (44, 156), (67, 243), (99, 211), (8, 184),
(40, 152)]

# first 64 ciphers pairs
[(159, 240), (165, 178), (113, 227), (48, 70), (52, 28), (107, 79), (130, 51), (186, 93),
(234, 117), (140, 149), (177, 171), (192, 5), (231, 19), (137, 212), (99, 181), (53,
238), (75, 68), (24, 222), (39, 56), (182, 220), (133, 73), (34, 116), (33, 71), (207,
150), (254, 146), (42, 175), (245, 153), (36, 230), (161, 103), (156, 169), (30, 173),
(129, 191), (11, 64), (211, 226), (3, 162), (218, 119), (204, 2), (144, 219), (131, 247),
(37, 203), (126, 221), (249, 85), (179, 210), (143, 76), (106, 38), (10, 187), (244,
197), (105, 213), (62, 72), (80, 255), (147, 201), (158, 205), (120, 4), (111, 208),
(246, 17), (31, 224), (100, 14), (66, 155), (115, 217), (20, 190), (214, 236), (61, 46),
(252, 123), (163, 233)]
```

Implementing the attack

Enumerate all possible values of K_4, then run all ciphertext pairs backwards through each possible K_4 and the last round S-boxes to determine the ΔU_3(the difference for the input to the last round).
For each possible K_4, increment a count when ΔU_3 corresponds to the value expected from the differential characteristic (`0b01000101`). The count that is the largest is taken to be the correct value since we assume that we are observing the high probability occurrence of the right pair.

```
  def inverse_sBox(U): # function to inverse the S-box
      inv_sBox_dict = {14:0, 4:1, 13:2, 1:3, 2:4, 15:5, 11:6, 8:7, 3:8, 10:9, 6:10,
                       12:11, 5:12, 9:13, 0:14, 7:15}
      return inv_sBox_dict[U]

  # an inverted version of substitute function above
  def inverse_substitute(U):
      U1 = (U >> 4) & 0b1111
      U2 = U & 0b1111
      inversed_substitution = (inverse_sBox(U1) << 4) | inverse_sBox(U2)
      return inversed_substitution

  # implement decryption as outlined in the report
  def decrypt_heys_differential(delta_U):
      keys_prob = dict()
      for K4 in range(256):
          count = 0
          for ciphers_pair in ciphers_pairs:
               # run ciphers pair backwards through each possible K4 and the last round
S-boxes to determine the difference for the input to the last round
              difference = inverse_substitute(ciphers_pair[0] ^ K4) ^
                           inverse_substitute(ciphers_pair[1] ^ K4)
              if difference == delta_U:
                  count += 1 # count the differences that are consistent with right pairs
          prob = count / 128
          keys_prob[hex(K4)] = prob
      return keys_prob

  # run the decrypt function
  keys_prob = decrypt_heys_differential(delta_U = 0b01000101)
  sorted_keys_prob = sorted(keys_prob.items(), key=lambda item: item[1], reverse=True)
  sorted_keys_prob[:20] # get the top 20 of the sorted list
```

Below is the output of the first 20 key bits sorted with probability, with correct K_4 bits (0x45) holding the highest probability of 0.125 (1/8). That indicates the attack was successful.

```
# Print Outputs of the first 20 key bits sorted with probability
[('0x45', 0.125), #correct K4
 ('0x75', 0.09375),
 ('0xf5', 0.078125),
 ('0x65', 0.0703125),
 ('0x25', 0.0625),
```

```
('0x4d', 0.0625),
('0xa5', 0.0625),
('0xc5', 0.0625),
('0xb5', 0.0546875),
('0xd5', 0.0546875),
('0x15', 0.046875),
('0x29', 0.046875),
('0x7d', 0.046875),
('0x95', 0.046875),
('0x5', 0.0390625),
('0x44', 0.0390625),
('0x4c', 0.0390625),
('0x74', 0.0390625),
('0x7c', 0.0390625),
('0x85', 0.0390625)]
```

Recover all remaining keys

We can recover K_3 using the similar approach, just notice to also inverse the permutations before running the data backwards though the 2nd round S-boxes to derive ΔU_2.
To recover K_2, we enumerate all possible K_2 to derive all corresponding ΔU_1. For correct K_2, ΔP should equal ΔU_1.
At last, simply XOR the plaintext with K_2 to recover the K_1.