Fault Attacks

AES Repeat

State

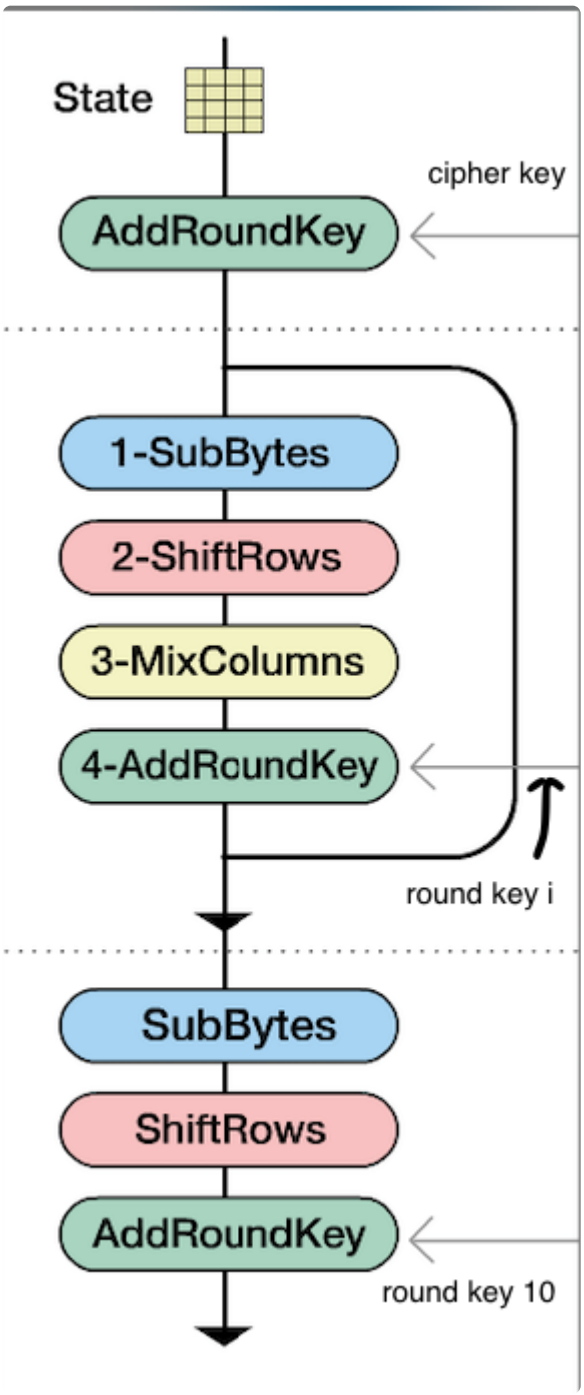
- A block from the plaintext message to be encrypted.
- Arranged as a 4x4 matrix. => 16bytes
- Goes to encryption process

Cipher key

- 16 bytes
- Goes to key schedule

Encryption process

Performs the encryption of the given plaintext block using 4 different transformation in the initial round, the 9 main rounds and the final round.



SubBytes

Uses the "S-box", byte substitution table. A bijection on the bytes.

Shitrows

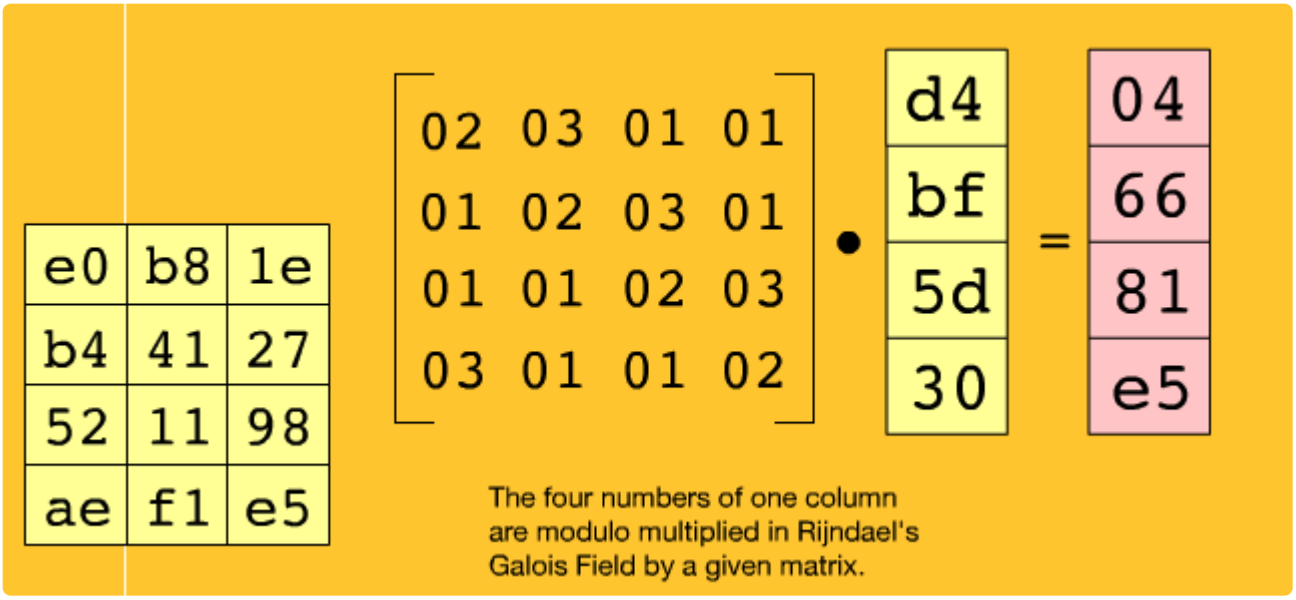
Rotate

- 0th row 0 bytes

- 1st row 1 bytes
- 2nd row 2 bytes
- 3rd row 3 bytes

MixColumns

The four numbers of one column are modulo multiplied in Rijndael's Galois Field by a given matrix.



Then this pink result if put into the column that yellow was in.
This step along with the ShiftRows is the primary source of diffusion in Rijndael.

AddRoundKey

Per column xor with the round key (remember both 4x4 matrices).

These transformations are applied to the State for 9 more rounds. The final round does not include the MixColumns transformation.

Key Schedule

Expansion of the given Cipher key into 11 partial keys, used in the initial round, the 9 main rounds, and the final round.
The expanded key can be seen as an array of 32-bit words (columns), numbered from 0 to 43.
The first 4 columns are filled with the given Cipher key.

$$W_0|W_1|\dots|W_{43}$$

4 columns a 16 byte key => 11 of them.

Words in positions that are a multiple of 4 $(W_4, W_8, \ldots, W_{40})$ are calculated by:

- Applying the RotWord (rotate 1 byte) and SubBytes transformations to the previous word W_{i-1}
- Adding (XOR) this result to the word 4 positions earlier (W_{i-4}) plus a round constant *Rcon*.

The remaining 32-bit words W_i are calculated by adding (XOR) the previous word W_i, with the word 4 positions earlier W_{i-4}

DFA against the AES

- How much is learnt by changing the ciphertext?
  - Not so useful
  - nothing, because the change is not key dependent
  - we need a fault(change) before a bijection
- Changing the AddRoundKey's input in final round?
  - model 1: Random byte error
    - nothing ????
  - model 2: toggle bits
    - nothing ????
  - model 3: set bits
    - will learn the value of the last round key since the ARK, will do the XOR and if the ciphertext does not change with respect to the encryption without the fault, we will know that the set bit is the same as in the normal operation.

Exercise 1:

Question 1

Why can you actually tell the moment of the fault injection?What would change if the fault is injected one round earlier?

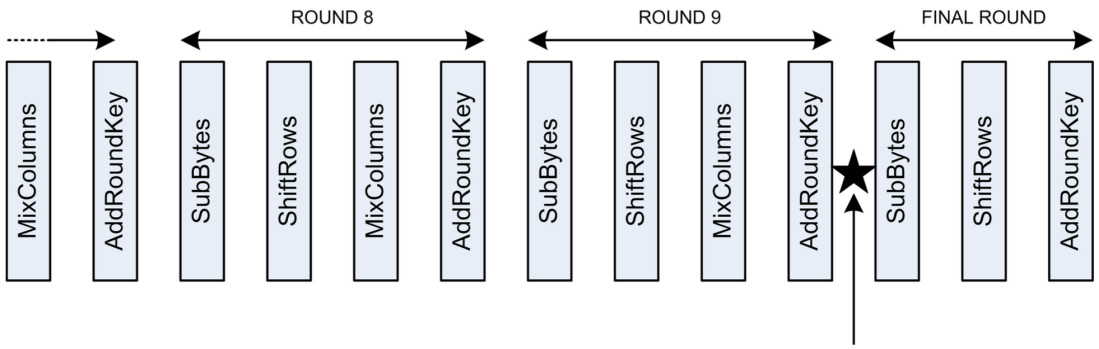
```
Difference
[[  0   0   0 214]
 [  0   0 246   0]
 [  0 252   0   0]
 [102   0   0   0]]
```

- This difference could result in following possibilities:
  - bit/byte error before {mix columns, shiftrows, sub bytes} of round 9
  - bit/byte error before {add round key} of round 8
- If the fault is injected one round earlier, at round 8, and if this happened before mix columns, we would end up with a completely different ciphertext

Question 2

Where has the fault been injected?

- We are said to assume the fault is injected before the last round which would be before the SubBytes operation.



Question 3

What is the key value for this position (see key matrices)?

see e1.py

Question 4

What is the minimum number of fault injections needed to recover the full key?

3 in the implementation, but we said 2 in the lectures. Slide 8 of lecture 8
1 byte => 2 faults
32 faults for the full key.

Question 5

Assume that the implementation incorporates a countermeasure which detects the fault injection with a probability of 1/2. What is the probability that the adversary can recover the full key without being detected? Can you conclude that this low detection probability is sufficient against the present adversary?

- Adversary needs 32 faults for the full key

$$Pr[\text{no detection}] = (\frac{1}{2})^{32}$$

- It is sufficiently small
- => use some counter measure that can detect if someone faulted with **good probability**
  - You want to detect pr: ≈ 0.99
  - $1/8$ not enough to be sure
- If you do the fault after the s-box the key is independent

Exercise 2:

The attack on first exercise is limited by:

- the assumption that the adversary can manipulate a single bit is a strong one.
- the number of needed faulty encryptions were high

Question 6

In the previous section the fault was injected before the last round. Can you also succeed by injecting a random byte-fault before the last round? If not, why?

- We cannot since we know that the Hamming weight of the xor between faulty state byte and the normal state byte is 1.
- With the random byte fault we are not sure about the Hamming weight

Question 7

What can be done to enable the exploitation of random byte-faults?

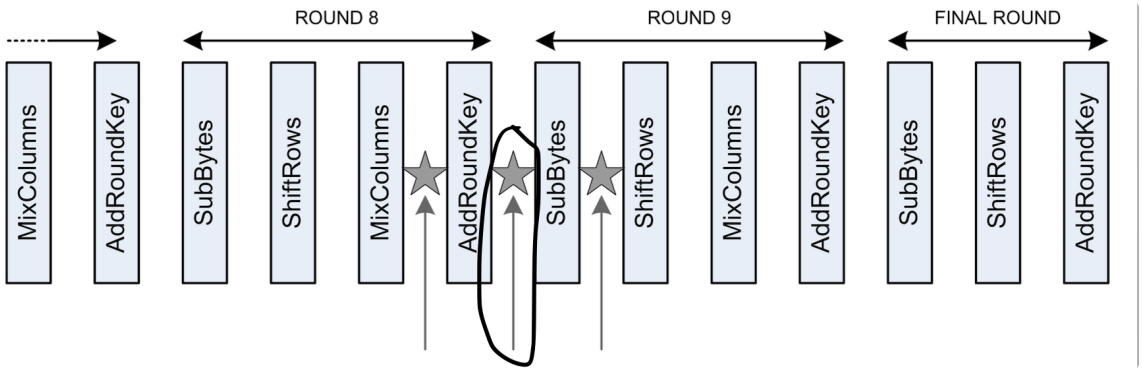
- The part with e

Attack

For this alternative fault model it would be impractical to follow the same strategy like in the previous section, that is to invert all seven operations. Therefore, we mount the attack from both sides and meet either after MixColumns (MC) or AK.

The idea is:

- There are 255 possible byte differences before the second last round.

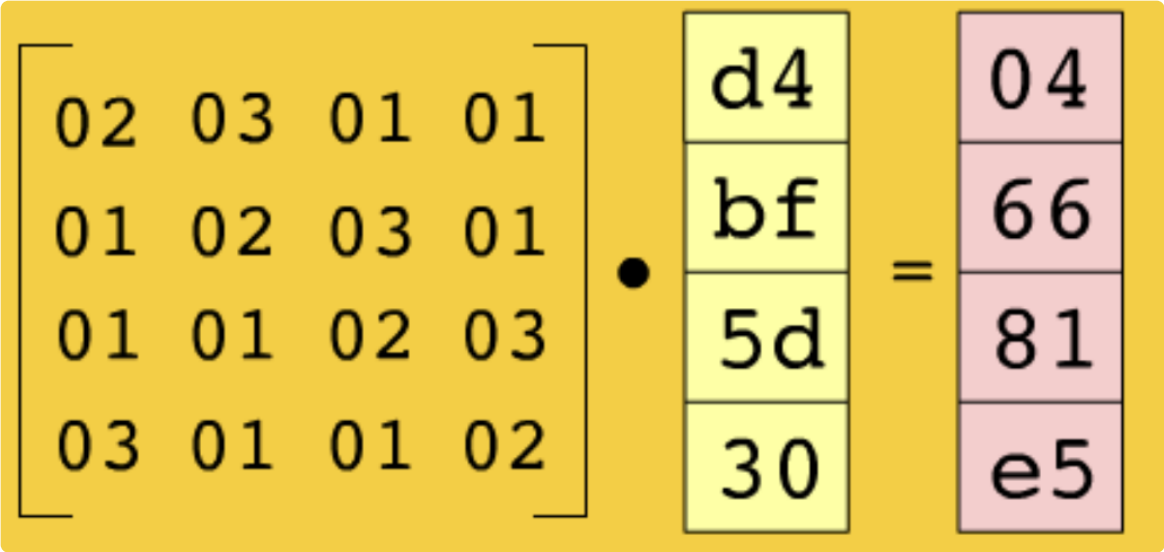


Note

Generally we only know the column of the injected fault and not the exact position. Therefore, we would have 4×255 differences. However, in this

exercises we assume to also know the exact position. (Otherwise, when MC happens we know that there are 4 random looking bytes)

- Therefore, also after MC we say that there are 255 possible differences.
  - It is important to note that now those 4 differences that the Note above talks about now dependent on each other because we know where the fault happened.
  - This dependence comes from the Galois field matrix of AES.



  - So if we assume that d4 was the random byte, and let's say it will have a difference of e with the original one:
    - The first row in pink will have the difference $2e$
    - Second row will have the difference e
    - Third row will have the difference e
    - Fourth row will have the difference $3e$
- In general tho, after the last round we will see a difference in four bytes and therefore we can guess four sub-key bytes, which leads to 2^{32} guesses.
  - But now with our knowledge of differences only a few of these key guesses fill fullfill the differences in rows.

  The attack steps:

1. Generate the set of differences which can occur after MC from the 255 possible initial differences.
2. For every difference:
   - Invert the last three transformations for the correct and faulty ciphertext
   - Find the 4-tuple of possible sub-keys.
     - Search for them byte by byte
3. Now we have 255 4-tuples
   - Look at these 4-tuples

- Some of them have no possible sub-keys in one or more positions
  - discard them
- Now you are left with only some tuples and each of them should contain only a few entries in each column.
- If the number of remaining tuples is greater than one, start from step two with another faulty ciphertext and pair-wise intersect the set of old tuples with the new one.

Step 1

```
def generate_faults_patterns_postMC(locxSB9):
    """
        Generate all the possible faults patterns after the MC
operation, considering
        a fault occuring at a specific line.
        locxSB9: The line index of the fault.

        Return a list of tuples, each of tuples being a possible error
patterns.
    """
    #TODO
    patterns = np.zeros((256, 4), dtype=int)
    for diff in range(256):
        patterns[diff,locxSB9] = diff
        mix_single_column(patterns[diff,:])
    return patterns
```

Step 2

Step 2.1

For each fault position:

- for each possible key
  - invert the faulty and correct byte from the ciphertexts
  - invert the sbox
  - calculate the difference between the resulting bytes (difference in GF of AES is xor)

- collect the differences in a matrix where the columns are indexed by the x of the fault

Step 2.2

The strategy here is that the byte differences should correspond to a row in the expected differences matrix. Therefore, we can look for byte by byte to make it easier.
Goes as follows:

- for each possible key1:
  - get the first diff byte from the calculated deltas for that key
  - if this diff value is in the first column of expected differences (256,)
  - Now use expected diffs, find the 2nd 3rd and 4th expected diff (by taking the row values that first diff corresponds to)
  - then find which keys give that in the calculated diffs
  Basically first we go from deltas to fpatterns for the first byte diff, then we go from fpatterns to deltas to see which keys correspond to the found first byte diff row.

```
def compute_4tuples_subkeys(ct,ctf,fpos,fpatterns):
    """
        Generate all the 4-bytes subkeys possibilities, taking into
account
        the fault positions in the ciphertext and the fault patterns.
        ct: correct ciphertext matrix.
        ctf: a faulty ciphertext matrix, resulting of the same
encryption process as the correct one.
        fpos: list of position of faulty bytes in the ciphertext matrix.
        fpatterns: possible faults patterns.

        Return a list of tuples, each tuples being composed of 4 bytes
values. Each tuple
        is a possible subkey of 32-bits.
    """

    # TODO
    # Step 2.1
    deltas = np.zeros((256,4))
    for pos in fpos:
        byte = ct[pos[0]][pos[1]]
        byte_f = ctf[pos[0]][pos[1]]
        for key in range(256):
            # undo the AK
```

```
            b = byte ^ key
            bf = byte_f ^ key
            # undo the s-box
            b = sbox_inv[b]
            bf = sbox_inv[bf]
            # do the diff
            diff = b ^ bf
            deltas[key, pos[0]] = diff

        # Step 2.2
    ret = []
    for key1 in trange(256):
        first_byte_0 = deltas[key1,0]
        if first_byte_0 in fpatterns[:,0]:
            idx = np.where(fpatterns[:,0]==first_byte_0)[0]

            expected_diff_1 = fpatterns[idx,1]
            k2 = np.where(deltas[:,1] == expected_diff_1)[0]

            expected_diff_2 = fpatterns[idx,2]
            k3 = np.where(deltas[:,2] == expected_diff_2)[0]

            expected_diff_3 = fpatterns[idx,3]
            k4 = np.where(deltas[:,3] == expected_diff_3)[0]
            for k in k2:
                for kk in k3:
                    for kkk in k4:
                        ret.append((key1,k,kk,kkk))
    return ret
```

Step 3

```
def attack_rndbyte_SB9(ct,ctfs,locxSB9,locySB9):
    """
        Attack a subkey of 32-bits.
        ct: correct ciphertext matrix.
        ctfs: a list of faulty ciphertext matrix, for the same inpus as
the correct one.
        locxSB9: [0-3], line where fault(s) have been injected.
        locySB9: [0-3], column where the fault(s) have been injected.
    """
    #TODO main attack function
    fpatterns = generate_faults_patterns_postMC(locxSB9)
    fpos = compute_ciphertext_fpositions(locxSB9,locySB9)
```

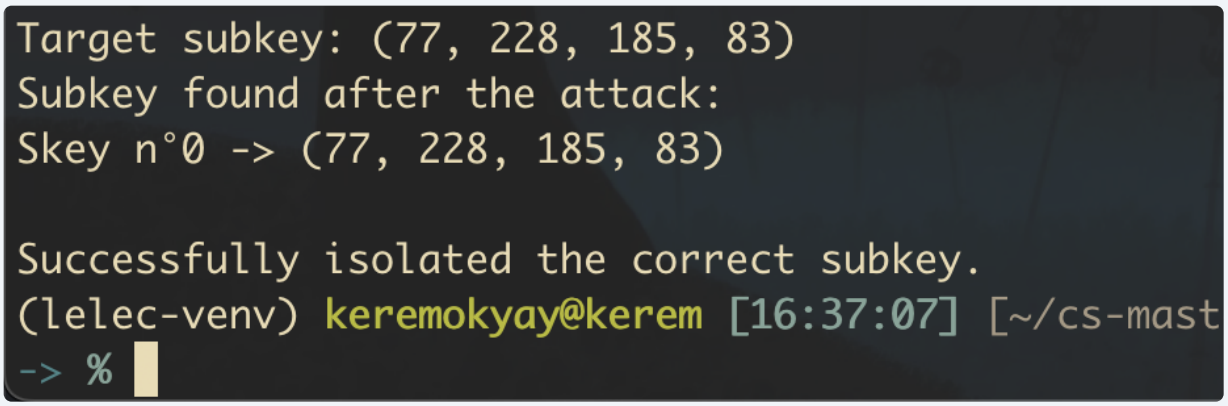
```
    ret = []
    for ct_f in ctfs:
        tuples = compute_4tuples_subkeys(ct, ct_f, fpos, fpatterns)
        if len(ret) == 0:
            for i in tuples:
                ret.append(i)
        else:
            # intersect tuples with ret
            ret = set(ret)
            tuples = set(tuples)
            ret.intersection_update(tuples)

    return ret
```

- Lastly, intersect the keys you get from 2 ciphertexts and return the intersection.

Question 8

What are the values of the four key bytes?



Question 9

What is the minimum number of fault injections needed to recover the full key?

- 2 faults

$$H(K) = 32$$

$$H(K|F) = \log_2(256 * 4) \approx 10$$

$$I(K;F) \approx 22$$

Question 10

Assume that the implementation incorporates a countermeasure which detects the fault injection with a probability of 1/2. What is the probability that the adversary can recover the full key without being detected? Can you conclude that this low detection probability is sufficient against the present adversary?

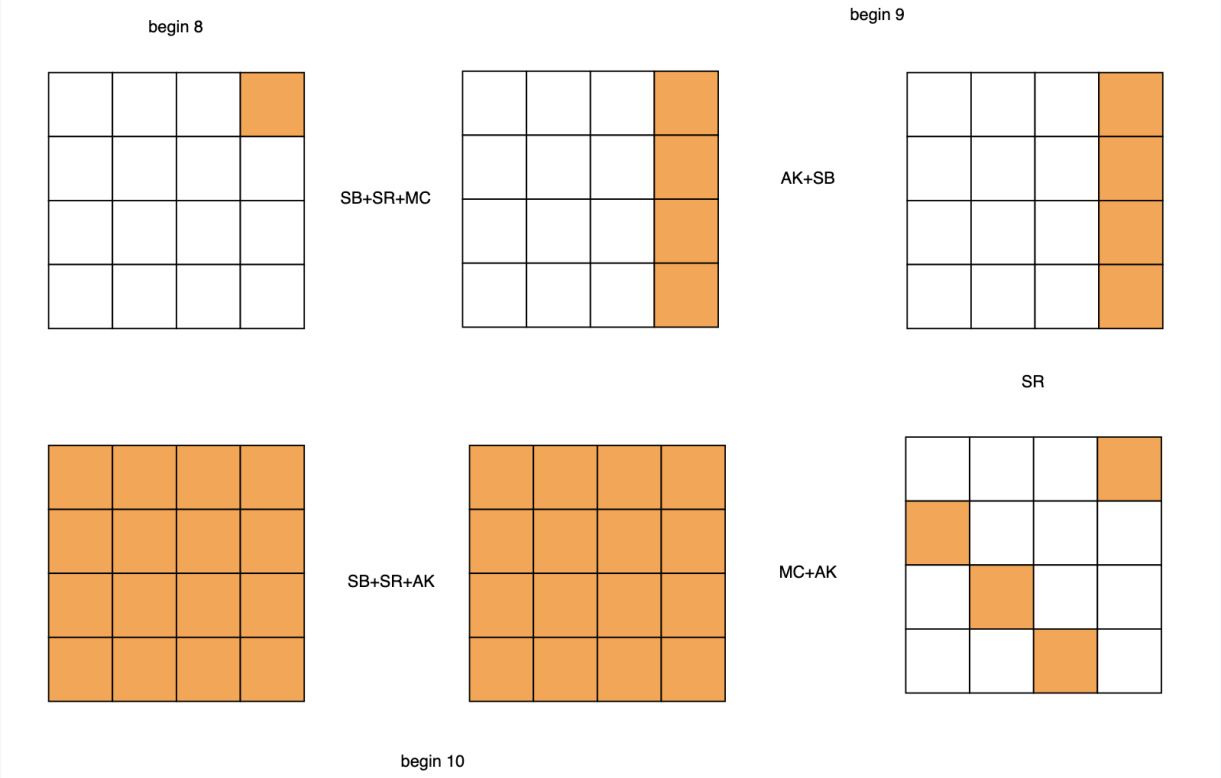
- full key 128 bits
- 2 faults per 32 bits
- 4 faults for 128 bits

$$Pr = (\frac{1}{2})^4 = \frac{1}{16} \approx 93\%$$

Exercise 3

Question 11

What does the difference pattern look like in this case after the last round?



Question 12

How many faults do you expect to be sufficient to recover the whole key in this case?

- A single fault could give the whole key now, but need to be sure you hit before 8th round.

Question 13

What is the disadvantage of this attack when you look at the difference after the last round? Can the adversary see a pattern and judge if his attack succeeded as intended?

- With random byte faults I imagine this would be hard since the whole ciphertext could look completely random

---

This attack is already quite powerful, however it can still be improved. In fact, AES can be attacked with the minimum number of necessary faults (i.e. one). The basic idea is to also invert the second last round with the key candidates from the first attack and to in addition use the relation between the round keys to filter out the remaining candidates. After this attack the search space for the full key is reduced to 28, which is easily feasible exhaustively.

Question 14

After learning about fault attacks against AES, what do you conclude about the number of undetected faults can be tolerated in a cryptographic implementation? What does this mean for the detection probability of countermeasures?

- Well if true no single fault should be tolerated