ELEC2760 - Exercise Session #4: Fault attacks

This session aims at giving an understanding of the impact of fault attacks and the meaning of countermeasures. In particular, we approach the topic by looking at different fault attack scenarios against the AES Rijndael, from simple but not very effective ones to more sophisticated ones. At the same time we discuss what the decreasing number of necessary faults means for the level of security provided by possible countermeasures.

Introduction. In this section we discuss the basic format of the tutorial, describe the available material (python code,...) and explain the tasks during the exercise.

Python code for AES. During the exercise we use a simple pure python implementation of the AES[1]. It is recommended to have a brief look at the script `aes.py` which contains the implementation of AES building blocks as well as full encryption procedure. Especially the latter one should help to recall the four different AES transformations and the order in which they are performed (see `encrypt_block()`). Especially, you will have to reuse some of these functions during the pratical session (e.g., Sbox, MixColumns, ShiftRows). This is also important to see how a fault evolves during the last (few) round(s) (see `encrypt_block_fault()`). Understanding how the data is encoded should save you quite some debugging afterwards.

An example for the usage of the `aes.py` is given in `main.py`. Concretely, an `AES` object has to be initialized with a key. The round key derivation is then performed in each round key is contained in the `_key_matrices` (see function `print_round_keys()`). Then, we initialize a plaintext (which is a array of 16 bytes) and obtain the ciphertext.

Basic principle of fault attacks against AES. Fault attacks allow, like side-channel attacks and linear cryptanalysis, recovering the key in a divide-and-conquer manner. For all the following attacks one generates one or more ciphertext pairs consisting of a correct and a faulty ciphertext. The correct one is generated by performing an ordinary encryption. The faulty one is obtained by injecting a fault into the AES state at a certain point during the encryption. At the very moment after the injection the correct and the faulty state represent a difference. Opposed to differential cryptanalysis, the difference occurs with probability one, however the imprecision of the fault model allows several possible differences. Thus, a single fault injection allows recovering a list of key candidates. Depending on the precision and the power of the fault model, the number of faults needed to recover the key varies.

To insert a fault, we use the functions `random_bit_pattern/random_byte_pattern` which take as argument the location of the inserted fault. It then returns all the possible faults (given the fault type). That pattern can then be used by `encrypt_block_fault()` as well as the round the fault is inserted and its location (i.e. before the Sbox, before MixColumns, ...).

Warming up. Every fault attack starts with the assumption about the injected faults. Throughout this session, we assume two types of fault, a bit fault (which flips one bit) and a random byte-fault (which sets one byte to a random value). Furthermore, we assume different points in time when the fault injection takes place. For instance, Figure 1 shows a

[1]Based on `https://github.com/boppreh/aes`

fault injection before the last AES round. Note, that the last round misses the MixColumns transformation. Usually, we are only interested in the difference between the correct and the faulty state, thus such diagrams as depicted in the figure also only show differences. That is, the white boxes indicate that the correct and the faulty state are identical for these positions. The green box indicates that a fault was injected into this byte of the AES state. At this point we might still have some knowledge about the difference caused by the fault. For instance, if we somehow managed to flip only one bit within this byte (for instance by shooting at the memory with a laser), we know that the difference has a Hamming weight of one. After the SubBytes (SB) transformation, the difference is unknown, as it also depends on the unknown correct state. Thus, we indicate this unknown difference by a blue box. Finally, ShiftRows (SR) and AddRoundKey (AK) do not change the difference.

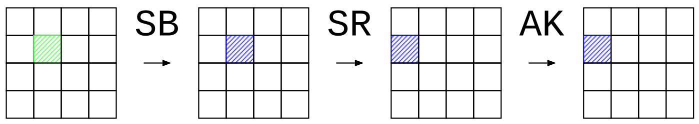

Figure 1: Manipulation of a fault during the last 3 transformations of an encryption.

Given such a difference the adversary can deduce two things. First, he sees that after the last round, only one byte is erroneous, which tells him that the fault was induced before or within the last round. Second, assuming that the error was induced before the last round, he can also deduce the position, where the fault was injected.

Launch the script `main.py` and observe the printed difference pattern.

Q1: Why can you actually tell the moment of the fault injection?
What would change if the fault is injected one round earlier?

We now assume that we know that our fault injection caused only one bit to flip in the green box. Given the output difference after such an injection, the following strategy can be pursued to recover the key. First, we take the correct and the faulty ciphertext and reverse the last transformation (AK). However, as we do not know the round key which was added during this transformation, we have to guess it. On the bright side, we only need to guess between 256 sub-key candidates. Afterwards, we reverse also the ShiftRows[2] and the SubBytes transformation. Figure 2 shows the above mentioned reversing process when a wrong key guess is made. The result of such a wrong guess is that after the inverse SubBytes transformation of the correct and the faulty state, their difference is random. That is, the difference does not have the expected property of having a Hamming weight of one. For a possibly correct key guess, as depicted in Figure 3, we can observe such a pattern.

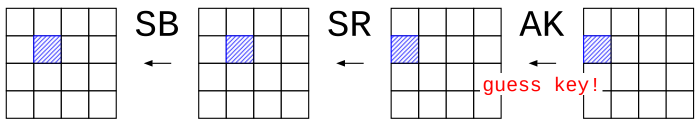

Figure 2: Reversing the transformations with a wrong key guess.

[2]This is actually not necessary in this case, but for the sake of completeness it is done anyway.

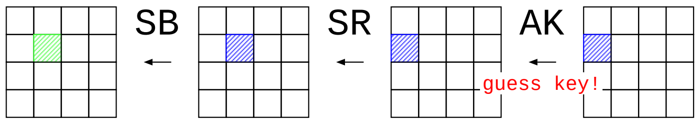

Figure 3: Reversing the transformations with a possibly correct key.

Exercise #1. First implement the attack described in the previous section. Assume that you know that a bit-fault was injected before the last round.

You can start from the script `e1.py`. First have a look to the function `gen_faults_bit_flip_last_round` that generates the correct / faulty ciphertext pairs to recover the key byte at the location (`locx`,`locy`) in the AES state matrix. Then, you have to implementation the function `attack_bit_flip_last_round` that perform a key recovery based on the obtained correct / faulty ciphertext pairs. By implementing this attack, you will have to answer the questions:

Q2: Where has the fault been injected?

Q3: What is the key value for this position (see `key_matrices`)?

Q4: What is the minimum number of fault injections needed to recover the full key?

Q5: 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?

Exercise #2 (random-byte faults before the second-last round). The attack from the last section is able to recover the key, but it is limited by two problems. First, the assumption that the adversary can manipulate a single bit is a strong one. Therefore, we first relax this assumption and now assume that he can randomly manipulate a byte. This can either be achieved by manipulating between 1 and 8 bits in memory or by a program-flow manipulation. For instance, by manipulating the clock signal of the device or the supply voltage instructions can be skipped or yield a wrong result. On an 8-bit platform, such a skipped or wrongly executed instruction results in a random byte-fault. Second, the number of needed faulty encryptions was quite high. We also aim at improving this situation now (or rather with the new fault model it will automatically improve as you will see).

Q6: 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?

Q7: What can be done to enable the exploitation of random byte-faults?

Below, you see the last seven state transformations of an AES encryption. Assume that the fault is injected at position (0,3) and sketch how it propagates until the end.

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. Essentially, the idea is that there are 255 byte differences before the second last round[3]. Therefore, also after

[3]In general we only know the column of the injected fault and not the exact position. Therefore, we would have 4×255 differences. However, for reasons of simplicity we assume to also know the exact position.

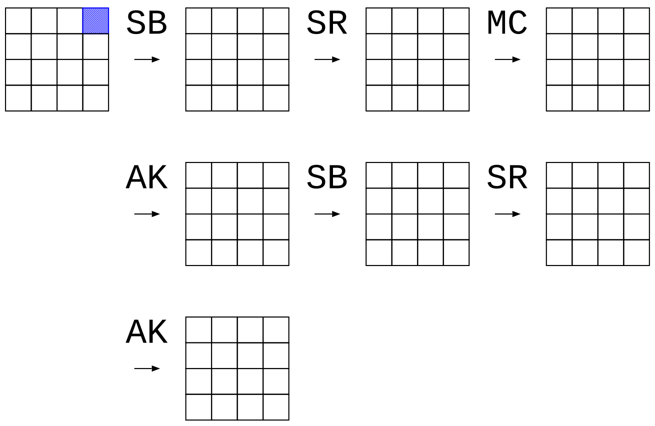

Figure 4: Complete the fault propagation throughout the last transformations of the cipher.

MC there are only 255 differences possible. It is important that those four differences are now dependent on each other. In particular if the difference in (0,3) before MC is e, then after MC we will observe the following differences:

`(0,3) = 2e, (1,3) = e, (2,3) = e, (3,3) = 3e.`

On the other hand, after the last round we will see a difference in four bytes and therefore, we can guess four sub-keys, which leads to 2^{32} guesses. However, out of of these 2^{32}, only few fulfill the difference. Thus the attack can be performed in three 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 the faulty ciphertext and find the 4-tuple of possible sub-keys. Search for them byte by byte (otherwise it would be computationally too heavy). Afterwards, you have 255 such 4-tuples. (3) Look at these 4-tuples, some of them have no possible sub-keys in one or more positions, those tuples can be discarded immediately. Afterwards, you should be 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 one, start from step two with another faulty ciphertext and pairwisely intersect the set of old tuples with the new one.

Your task is it to implement this attack in the script `e2.py` similarly as what you did for the previous exercise. Note that for each of the three steps described above, one function has to be implemented. By doing so, you will answer the following questions:

Q8: What are the values of the four key bytes?

Q9: What is the minimum number of fault injections needed to recover the full key?

Q10: 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?

Exercise #3 (one fault is enough). By now you are familiar with the basic techniques which are common to all fault attacks against AES. In this section we will sketch possible improvements which lead to the state-of-the-art attacks against AES. The most straightforward improvement is to inject the fault one round earlier, that is, before round eight.

Q11: What does the difference pattern look like in this case after the last round?

Q12: How many faults do you expect to be sufficient to recover the whole key in this case?

Q13: 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?

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 2^8, which is easily feasible exhaustively.

Q14: 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?