Square Attack on 5-round AES – Report

Le Huy Duc

I. Introduction:

In this project I implement the basic Square attack as described in the lecture. The program has constant space complexity **O**(2^11) and time complexity **O**(2^48) in the worst case. If 4 bytes of the last round key is given as in the description, the program can find the secret key in a short time (complexity **O**(2^40)).

II. Technical details:

The Square attack is very basic. It guesses 5 bytes (so 2^40 tuples in total), and for each combination, it takes 2^8 steps to check if the resulting cell is balanced. So in total, the time complexity is **O**(2^48) to guess 1 byte of the round-5 key.

I will describe the program step by step. In the source code files, I comment all places that are important to understand.

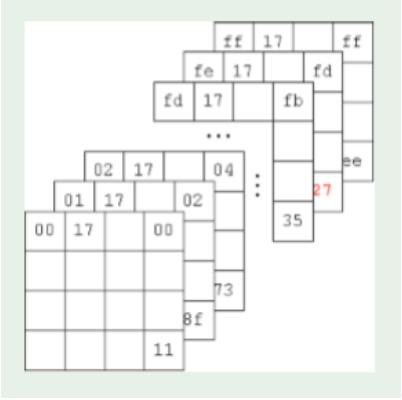
Abbreviation: Round-5-key (RO5k), Round 5 (RO5), end of Round 4 (eRO4)

1. Input data:

- Square attack needs delta-set as input. Definition of the delta-set is in the slide.

- Each delta-set is an array of 256 plaintexts, each plaintext is an array of 16 bytes.

- The code to generate delta-set can be seen in DeltaGenerator.cpp



2. Balanced-byte properties at round 3:

- We know that at the end of round 3 of AES, all cells are **balanced** across 256 plaintexts. This means if we xor all 256 cells (i,j) together, the result is 0.

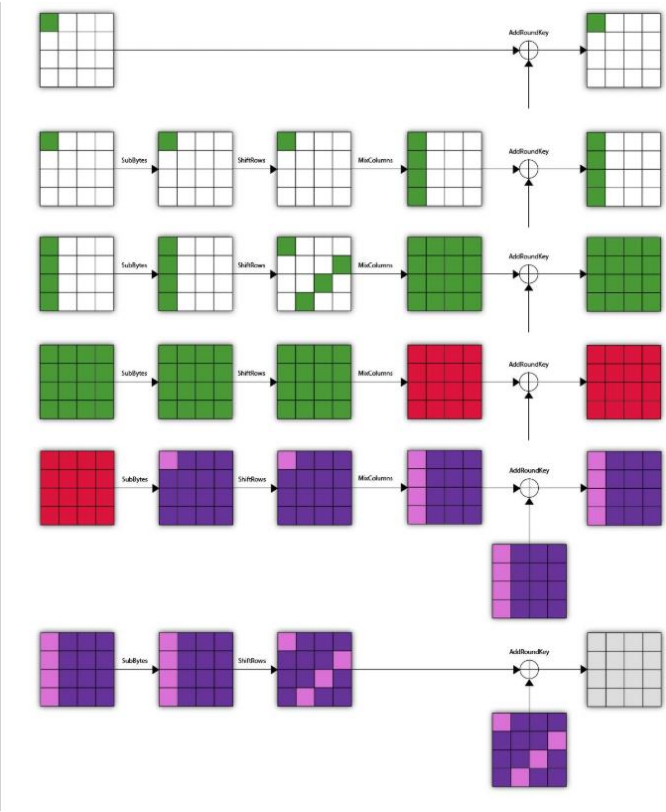
$$\bigoplus_{t=0}^{255} s_{i,j}^{(t)} = 0$$

- In AES-4, this property helps us find the secret key. We use the same properties in AES-5.

3. Square attack on AES-5:

- This is a **chosen-plaintext attack.** We know the input, and the output at the end of round 5.

a) Basic version:



- We can go from RO5 to eRO4 by guessing 4 bytes at 4 positions. Then, just apply AddRoundKey, then ShiftRowInv, and SubByteInv.

- For each 4-bytes tuple, we guess another 4 bytes of RO4 key. Using these 4 extra key bytes, we can go back to the eRO3.

- We can see that there is one byte (pink) we need to check for the balance property. If that cell is not balanced, then our 8-bytes guess is incorrect and we need to try another guess.

- If it is balanced, then it might be a false-positives. We need to check a tuple with at least 3 different delta-sets on average to make sure it is correct.

- Time complexity: O(2^(8*4) * 2^(8*4) * 2^8) = 2^72

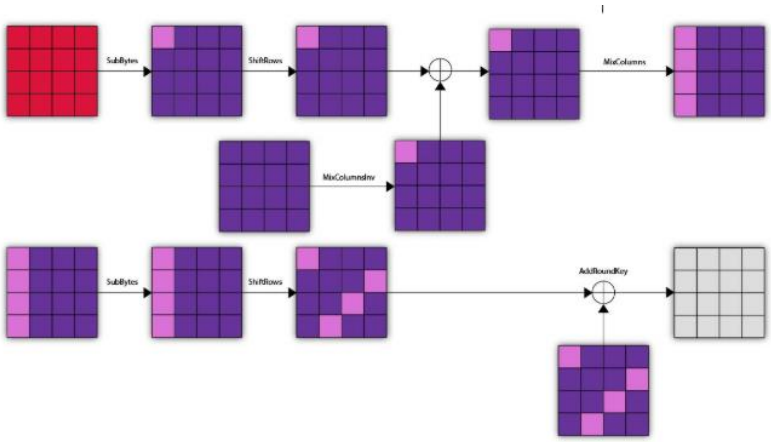
-> This attack takes too much time.

b) Improved version:

- MixColumns is linear with respect to each column input. So we have:

$$\begin{aligned}&\mathrm{MixColumns}(\mathrm{state}) \oplus \mathrm{RoundKey}\\ =&\mathrm{MixColumns}(\mathrm{state}) \oplus \mathrm{MixColumns}(\mathrm{MixColumnsInv}(\mathrm{RoundKey}))\\ =&\mathrm{MixColumns}(\mathrm{state} \oplus \mathrm{MixColumnsInv}(\mathrm{RoundKey}))\end{aligned}$$

- This allows us to perform MixColumnInv before AddRoundKey at the end of round 4. After MixColumnInv, we only need to care about the byte value on the first row. Therefore, we only need to search 1 byte of round 4 key, instead of 4.



- Thanks to this, we only need to search 2^40 keys and the time complexity is only O(2^48).

III. Program instructions:

- My project includes 5 source code files: *aes.h, DeltaGenerator.cpp, encrypter.cpp, decrypter.cpp,* and *AutomatedTesting.cpp*. Also, there're two text files: *plaint.txt* and *key5.txt*, which are input data. **See also the Appendix.**

- Note: in *aes.h,* I modified a few places to make it faster, such as precomputing xtime(x), MULT_2(x), ... Also I added the function *invertKeyExpansion()* that recovers the secret key from any round key.

- Step 0: make sure everything is in the same folder.

- **Step 1:** run DeltaGenerator.cpp to create the delta-set.

- **Step 2:** choose the value of RO5 key in *key5.txt*. If the user wants to select the main key, he needs to generate the RO5 key from that and input the RO5 key (very simple). Reason: **See Appendix**

- Step 3: run Decrypter.cpp. The program will output the RO5 key and the main key to the console and file *key5guess.txt*.

- **Step 4:** you can also run file *AutomatedTesting.cpp*. It generates a random key, then executes *encrypter.cpp* and then *decrypter.cpp*. Finally, it compares the output of *decrypter.cpp* to the answer, and notify the user if the guess is incorrect. I've performed 10000 tests which were all answered correctly.

IV. Future work:

- We can implement impossible differential attack to reduce the time further. However, since this is a basic cryptography course and project C is not my main topic, I do not implement that.

- We can also use Nvidia CUDA's GPU acceleration. This can speed up the attack by 200 times.

V. Conclusion:

In this project, I have made a successful attack on 5-round AES using the Square attack by guessing 2^40 keys. The performance can be improved easily using multithreading or GPU acceleration.

Appendix

I. Scripts to run on Command line: (for Ubuntu, please use sudo):

g++ DeltaGenerator.cpp -std=c++11 -O2 -o deltaGenerator

g++ encrypter.cpp -std=c++11 -O2 -o encrypter

g++ decrypter.cpp -std=c++11 -O2 -o decrypter

g++ AutomatedTesting.cpp -std=c++11 -O2 -o automatedTesting

./deltaGenerator

./encrypter

./decrypter

//***/

- The program will show the results in the console (and the file *key5guess.txt*)

- Note: for *deltaGenerator,* we should generate at least 5 delta-sets to ensure the program is 99.9% correct. 10 is recommended. *plaint.txt* uses 10 sets.

- Note 2: for *AutomatedTesting,* I strongly suggest MAX_KEY is set to a small value (such as 64) for quick demo. Big values need a long time to run (but produce better test cases, which I used before submitting).

II. Questions and answers:

a) Why does *encrypter.cpp* receives round 5 key as input, instead of main key?

- First, this is intended. There is no technical difficulty. *Aes.h* already contains code to generate *expanded_keys* from main key or from any round key.

- The decrypting process can take a very long time. So I want to be able to choose RO5 key such that all byte values are bounded (for example, all values <= 64). This means it can finish quickly and can be debugged more easily.

- Conclusion: this doesn't affect the correctness of *decrypter.cpp*. It can decrypt any input key. I have tested it using *AutomatedTesting* with MAX_KEY = 255