Chapter 5 : The Fall

Team Obfuscated

Rohit Singh(19111074) Virendra Nishad(19111099) Suvasree Biswas(19111416)

Final password : mvcpldwwlmjabzfm

Motivation Behind:

The most effective attacks on reduced-round variants of the AES are variants oftheSquareattack which is due to Knudsen. Since this attack was used against a predecessor [10] of the AES it was accounted for by the AES designers [11].In this attack we take a set of 256 plaintexts where the first byte takes all possible values. The other 15 bytes of the input can take any value but the same value in a given byte position must be used across all 256 texts. We will describe a set of texts that have this property as an integral. Imagine one beginsan AES-round with such an integral. In the following we shall denote the byte-position containing a variable value with an "a" (for "all").

The`AddRoundKey`operation adds the same round key to each of the 256 texts in the integral, therefore any integral before`AddRoundKey`will yield an integral after. Consider a second round of transformation.

It follows that after two rounds of encryption and for each byte position,every possible value in a given byte position is taken once and only once in the set of 256 texts.

The interesting part is what happened

during the`MixColumns`operation. Before the operation, in each byte position the 256 values were a permutation of the values 0,...,255.`MixColumns`com-bines four bytes to yield one byte in a linear way. This means that after the application of`MixColumns`every byte position will be balanced, that is, if we exclusive-or all 256 values in any single byte position we will get zero as a result.Note how this property, after three rounds of AES encryption does not depend on the details of the S-box nor on the value of the secret key.

Overall objective:

Step 1: Reach chapter 5 ciphertext

Step 2: The plaintext “password” , gives a unique ciphertext **c1**

Step 3: Using Cryptanalysis of a variant of aes, namely EAEAE problem, we find the key

Step 4: The problem had input with a block of size 8 bytes as 8 x 1 vector
over F_{128}.

Step 5: It was noticed that upon giving any inputs of the form ‘gf’ or ‘hf’, same output as of inputs ‘g’ or ‘h’ respectively were produced. That implies that ‘f’ serves the purpose of being a padding.

Step 6: find Expansion function Expansion to decrypt the ciphertext gotten from caves for the plaintext ‘password’. Feed that as the final answer.

Initial Observation :

It was observed that the ciphertexts have characters only in the range f-u. This led us to guessing that the encoding should be in hexadecimal form as from f to u, justifying the presence of 16 characters. GF(128) has 128 elements. This paved the way for us to extrapolate that the
guessed elements are from ff-mu. After plugging in some of the ciphertexts,we find our observation may be not on the incorrect path. For the input character mapping,we assumed ASCII mapping.We also checked that ciphertext
corresponding to some letter (eg d) is exactly same as prefixed f with it (fd), so we assumed f mapping to
0.Therefore from ff to mu,we assumed mapping 0 to 127

Further Observations :

1. We were trying to infer a pattern by feeding plaintext in the cave interface. We did find that 16 letters in the output are from ‘f’ to ‘u’.
2. Therefore, It could also be inferred that the input also consists of these letters only, owing to the fact that only 16 letters could be represented by four bits. Thereby, it was prudent to infer that each alphabet was mapped with a number from 0 to 15.
3. Input was 8 bytes but the field was GF(128). The only explanation of these two contradictory lines could have been that the inputs from ’ff’ to ‘mu’ considering MSB for any letter will always be 0.
4. We generated 256 sets each containing 128 ciphertexts corresponding to plaintexts of the form C i−1 P C 8−i where multisets P and C . Tracing the plaintexts through the encryption process (EAEAE), we saw that after the second A transformation, the texts have the property that the multisets (formed by each byte position in the set of 128 texts) add up to 0. This happens since running across 256 plaintext over field of degree 8 yields all possible combinations, thereby making the sum to be 0. (Square Attack)
   Consider the input plaintext represented as I_0 , I_1 , I_2 ... I_7 and
   the output cipher text as C_0,C_1 ... C_ 7 .

On varying the inputs for k from 0 to 7, making I_0 to I_(k-1) as ‘ff’ we get the output ciphertext values also as ‘ff’ for C_0 to C_k . In other words whenever the input plaintext has I_0 ... I_j = ‘ff’ we get the ciphertext as C_0,....C_j = ‘ff’.

The output changes only after k, this implies that the ‘ff’ present in each row have to be at the end row. Thus, we get a Lower Triangular matrix.

Steps to reach Step 1: (getting to the cipher)

After feeding the correct plaintext of password in Chapter 4, we land in chapter 5.

Try 1 : Initially we feed **go,** and then we feed **dive.** This makes us reach our early demise. So we go back again and restart.

Try 2: now we 1st feed **go,** then -- > **wave** -- > **dive** -- > **go** -- > **read**

Details of the Files enclosed with chronology:

1. **getInput.ipynb -** output of this is inputs.txt. This essentially generates the input taking the lower triangular matrix in consideration.
2. **StoreCipher.py -** output of this is outputs.txt. This file takes the inputs generated by inputs.txt. And runs these generated inputs through the caves interface, thereby storing the generated corresponding ciphertexts. These output ciphers are stored in the file named ‘outputs.txt’
3. **AlignCipher.py -** this takes as input the file ‘outputs.txt’. It aligns it according to the format as is needed for ‘BruteForce.ipynb’. It outputs the file ‘new_output.txt’. Every new ‘ffffffffffffffffffffffff’ is then manually made into a new line such that BruteForce can take the input as it wants. If this manual transformation is not done then, the other cells of the matrix is yielding to zero such that only 1 cell is non-zero.
4. **BruteForce.ipynb -**

PART A:

this file takes as input the files ‘inputs.txt’ and ‘new_output.txt’. It thereafter runs through all the ciphertext to bruteforce it one by one. The chronology of the the linear transformation functions and the expansion functions have all been taken care of here. We then implemented operations over finite field of 128 with generator x 7 + x + 1 which will be used to carry out operations

We found our **Linear Transformation function** as

[[13, 118, 82, 92, 126, 38, 19, 2], [0, 121, 10, 118, 32, 103, 54, 66], [0, 0, 118, 99, 22, 36, 125, 33], [0, 0, 0, 57, 85, 11, 3, 62], [0, 0, 0, 0, 86, 59, 31, 116], [0, 0, 0, 0, 0, 35, 100, 61], [0, 0, 0, 0, 0, 0, 81, 34], [0, 0, 0, 0, 0, 0, 0, 77]]

We found our **Expansion function** as:

[82, 8, 72, 31, 89, 38, 94, 83]

PART B:

After the Linear Transformation function and the Expansion function have been found, we then use these in the equation

ASCII(E -1 (A -1 (E -1 (A -1 (E -1 (Ciphertext block))))))

So the ciphertext given by the caves interface for the plaintext ‘password’ is now fed as input in the program as input. We need divide this input up in 16 bits and provide the last part of the file, so that it gets us the corresponding plaintext of the plaintext ‘password’. We feed this plaintext to the caves interface and that gets us landed on chapter 6, clearing the chapter 5.

Final password : **mvcpldwwlmjabzfm**

Reference:

1. https://link.springer.com/content/pdf/10.1007%2F11506447_1.pdf
2. Git repos