CS641: Chapter 5
Team name: team7 (chaos)

Komal Kalra (18111032) Riya James (18111054) Sristi Jaiswal (18111074)

March 26, 2019

Getting to the Cipher

The sequence of commands we used to get to the cipher:
go → wave → dive → go → read

Guessing the Character Mapping

We observed that the ciphertexts have characters only in the range f-u,so we guessed that the encoding should be in hexadecimal form as from f to u,there are 16 characters.As the field GF(128) has 128 elements,so we guessed elements from ff-mu.After analysing some of the ciphertexts,we find our observation may be somewhat right. 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.

Initial Approach

We collected 256 sets each containing 128 ciphertexts corresponding to plaintexts of the form $C^{i-1}PC^{8-i}$ where multisets P and C have the same meaning as defined in class. 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 i.e., the multiset Z, as described in class.

Any affine mapping of the the exponentiation function will result in the same set of equations, so the dimensionality of the solution space is 8 (because each 'unit' has 7 bits, and the vector in the affine mapping adds another dimension). Therefore we expected to get 120 linearly independent equations describing each byte transformation.

We defined variables z_i such that $E(z_i) = i$, where E is the exponentiation transformation. From each set (and for each byte) we inferred one homogeneous equation. Trying to solve for the variables, we found that we couldn't get 120 equations for the first 4 bytes no matter how many sets of texts we used. (Implemented in `structural_cryptanalysis.ipynb`. This used input texts generated from `generate_input.ipynb`)

Observation

From this we concluded the A mapping (since E is a byte transformation anyway) had to have very poor mixing for the first few bytes. With this knowledge, we tried manually feeding in inputs to discern a pattern. It quickly became clear that every byte only affected the encryption of every subsequent byte i.e., that the matrix A had to be lower triangular.

Modified Approach

Armed with the knowledge that the A matrix is lower triangular, we chose 3 sets of plaintexts such that the i^{th} element of each set has only byte-position i as non-zero. (We chose the non-zero bytes as 'hk', 'lu', and 'mt' for each of the three sets.) Then tracing the plaintext values through the $EAEAE$ transformation, we get

$$c_k = (a_{k,k} * ((a_{k,k} * c)^{e_k}))^{e_k}$$

where c_k is the k^{th} byte of the ciphertext (corresponding to the plaintext with only the k^{th} byte as non-zero), $a_{k,k}$ is the k^{th} diagonal element of the A matrix, e_k is the k^{th} element of the exponentiation vector. All operations (multiplication and exponentiation) are over F_{128} with the irreducible polynomial $x^7 + x + 1$. Bitwise addition is performed *modulo* 2.

Then, we brute force the values of $a_{k,k}$ and e_k (for the ranges $0 - 127$ and $1 - 126$ respectively), and calculate the output using the formula above. If the calculated value matches the value of the corresponding byte of the cipher text, we store it in a dictionary with the $e's$ (exponent entries) as keys and $a's$ (diagonal entries) as the corresponding value. The following are the (candidate) values we obtained:

Position 1: $18 : 96, 21 : 104, 88 : 126$

Position 2: $11 : 13, 82 : 64, 34 : 85$

Position 3: $20 : 4, 108 : 97$

Position 4: $29 : 11, 55 : 14, 43 : 20$

Position 5: $26 : 78, 113 : 121, 115 : 122$

Position 6: $54 : 73, 10 : 83, 63 : 85$

Position 7: $102 : 53, 33 : 95, 119 : 124$

Position 8: $56 : 28, 48 : 43, 23 : 50$

Semi-Cracked A matrix:

```
A =
[[126   0   0   0   0   0   0   0]
 [ 32  85   0   0   0   0   0   0]
 [  0 100   4   0   0   0   0   0]
 [  0   0  29  20   0   0   0   0]
 [  0   0   0  17  78   0   0   0]
 [  0   0   0   0  94  85   0   0]
 [  0   0   0   0   0  77  53   0]
 [  0   0   0   0   0   0  62  50]]
E = [ 88  34  20  43  26  63 102  23]
```

After this we calculated non diagonal elements by brute forcing like this: To calculate $a_{i,j}$,we required elements of matrix from $a_{j,j}$,$a_{j+1,j}$ to $a_{i,j}$ and from $a_{i,j+1}$,$a_{i,j+2}$ to $a_{i,i}$.So we calculated first all $a_{i+1,i}$,then $a_{i+2,i}$ and so on.While calculating $a_{i,j}$,we took the ciphertext corresponding to the plaintext whose only non-zero byte is at i^{th} position.
Proceeding in the same way,we were able to get all elements,also eliminating some values for diagonal elements.
We got following matrices for A and E :

```
A =
[[126   0   0   0   0   0   0   0]
 [ 32  85   0   0   0   0   0   0]
 [ 68 100   4   0   0   0   0   0]
 [126  91  29  20   0   0   0   0]
 [ 74  83 122  17  78   0   0   0]
 [ 76  99  69   0  94  85   0   0]
 [ 58 111  40  93 114  77  53   0]
 [ 39 122 108  68  78   2  62  50]]
E = [ 88  34  20  43  26  63 102  23]
```

$$E = [88, 34, 20, 43, 26, 63, 102, 23]$$

(Implemented in `eaeae_extract_key.ipynb`.)

An Alternate Approach

Because matrix A if lower triangular we got to know that any byte in ciphertext will depend on the corresponding and its previous bytes ie i^{th} byte of ciphertext will depend on j^{th} byte of plaintext if and only if $j \leq i$.

So we started by varying first byte of plaintext from ff to mu and noted down which plaintext byte corresponds to first byte of ciphertext(As first byte of ciphertext will only depend on first byte of plaintext). Then fixing that plaintext byte,we proceed in the same way for the second byte(second byte of ciphertext will depend only on first and second byte of plaintext).We did this iteratively for all 8 bytes of the 2 blocks (separately for each block),and finally we got the plaintext password after converting it into ASCII encoding. (Implemented in `break_eaeae.ipynb`.)

Key and Password

Key:

```
A =
[[126   0   0   0   0   0   0   0]
 [ 32  85   0   0   0   0   0   0]
 [ 68 100   4   0   0   0   0   0]
 [126  91  29  20   0   0   0   0]
 [ 74  83 122  17  78   0   0   0]
 [ 76  99  69   0  94  85   0   0]
 [ 58 111  40  93 114  77  53   0]
 [ 39 122 108  68  78   2  62  50]]
E = [ 88  34  20  43  26  63 102  23]
```

To decrypt the password, we first computed the inverse of the matrix A and E.

Inverting matrix A: To do this,we did the row reduction of the augmented matrix constructed using A and identity matrix to calculate the inverse.

```
A inverse
[[  3   0   0   0   0   0   0   0]
 [ 23 109   0   0   0   0   0   0]
 [105  72  97   0   0   0   0   0]
 [ 55 104  70  75   0   0   0   0]
 [122 119  55 104  95   0   0   0]
 [116  76  65  85  68 109   0   0]
 [ 86  31  40  88  28  94  81   0]
 [ 26  88 122  10  93  55  63  80]]
```

Inverting E: And, for E we calculated inverse by taking inverse of each element in it.And then we decrypted the password. Then we split it into 2 blocks and decrypted the two blocks of the password separately and converted it to ASCII.

$$\text{ASCII}(E^{-1}(A^{-1}(E^{-1}(A^{-1}(E^{-1}(\text{Ciphertext\_block}))))))$$

Password: `cgbhjmbixebsjcvr`

(Please install pyfinite by running `'pip install pyfinite'`, compatible only with Python3.)