Project I
Symmetric Cryptography

Yfantidis Dimitrios (AEM: 3938) 10

May 2023

Introduction

This is the report of the first part of the work in the course "Cryptography Foundations".

The report is written in LATEX and compiled by the **MiKTeX** compiler (ver: One MiKTeX Utility 1.6 - MiKTeX 23.1).

The implementations of the exercises were done in **Python (v 3.10.6)**. For each exercise that requires a code implementation, there is a corresponding folder:

proj. 1 crypto 3938/code/ex*

Where there is a single python script, **ex*.py,** containing the implementation and possibly some ".txt" files or other resources for the program.

Almost all the mathematical notations that appear in the text are commonly accepted. Exceptions are the following, which are specified:

x mod y : The remainder of the integer division of x by y (not an equivalence class).

$\mathbb{N}$: The set of natural numbers `including` 0, i.e. $[0, 1, 2, ...]$.

$\mathbb{N}^*$: The set of natural numbers `without` 0, i.e. $[1, 2, 3, ...]$.

Exercise 1 (2.2)

We do:

$$g(x) = x^2 + 3x + 1,\ x \in R$$

$$f(x) = x^5 + 3x^3 + 7x^2 + 3x^4 + 5x + 4,\ x \in R$$

x_0 some root of $g \quad \Rightarrow \quad g(x_0) = 0$

We calculate $f(x_0)$:

$$\begin{aligned} f(x_0) &= x^5 + 3x^3 + 7x^2 + 3x^4 + 5x_0 + 4 \\ &= x^5 + 3x^4 + 3x^3 + 7x^2 + 5x_0 + 4 \\ &= x^5 + 3x^4 + x^3 + 2x^3 + 7x^2 + 5x_0 + 4 \\ &= x^5 + 3x^4 + x^3 + 2x^3 + 6x^2 + 2x_0 + x^2 + 3x + 3x_0 + 1 + 3 \\ &= (x^5 + 3x^4 + x^3) + (2x^3 + 6x^2 + 2x_0) + (x^2 + 3x_0 + 1) + 3 \\ &= x^3 (x^2 + 3x_0 + 1) + 2x_0 (x^2 + 3x_0 + 1) + (x^2 + 3x_0 + 1) + 3 \\ &= x^3 \cdot g(x_0) + 2x_0 \cdot g(x_0) + g(x_0) + 3 \Rightarrow \\ f(x_0) &= 3 \end{aligned}$$

So, if $\mathrm{E} = \{\mathrm{E}, D, \mathrm{K}, \mathrm{M}, \mathrm{C}\}$ our cryptosystem then:

$$\mathrm{M} = \mathrm{C} = \{\alpha : 1, \beta : 2, \ldots, \omega\}$$

$$\mathrm{K} = \{f(x_0)\} = \{3\}$$

$$E(m, k)|_{k=3} = (m + k - 1)\ mod\ 24 + 1\ |_{k=3} = (m + 2)\ mod\ 24 + 1 = c$$

$$D(c, k)|_{k=3} = (m - k - 1)\ mod\ 24 + 1\ |_{k=3} = (m - 4)\ mod\ 24 + 1 = m$$

Είναι δηλαδή ένα σύστημα μετατόπισης, όπου η αρίθμηση δεν ξεκινάει από το 0, αλλά από το 1 (από εκεί προκύπτουν και οι επιπλέον άσσοι στις συναρτήσεις E και D). Προκύπτει ότι, αφού εισάγουμε το κρυπτομήνυμα "οκηθμφδζθγοθχυκχσφθμφμχγ", δίνεται ως έξοδος το αποκρυπτογραφημένο μήνυμα "μηδεισαγεωμετρητοσεισιτω".

(Implementation: `code/ex1/ex1/ex1`**.py**)

Exercise 2 (2.3)

Using Friedman's method, it follows that the key length of both messages is 7. A brute force key finding implementation is found within the script ex2.py where it searches for all possible strings of length 7 with 26 Latin characters (26^7 combinations).

Only those decryptions are printed whose text is relatively readable, i.e. the frequency of the letters is close to the real one.

(Implementation: `code/ex1/ex1/ex1`**.py**)

Exercise 3 (2.4)

We have, $E:\{0,1\}^{16} \rightarrow \{0,1\}^{16}$ with $E(m) = m \oplus (m << 6) \oplus (m << 10) = c$. We write the binary sequences in the form of vectors:

$$\begin{pmatrix} c_{15} \\ c_{14} \\ c_{13} \\ c_{12} \\ c_{11} \\ c_{10} \\ c_9 \\ c_8 \\ c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} m_{15} \\ m_{14} \\ m_{13} \\ m_{12} \\ m_{11} \\ m_{10} \\ m_9 \\ m_8 \\ m_7 \\ m_6 \\ m_5 \\ m_4 \\ m_3 \\ m_2 \\ m_1 \\ m_0 \end{pmatrix} \oplus \begin{pmatrix} m_9 \\ m_8 \\ m_7 \\ m_6 \\ m_5 \\ m_4 \\ m_3 \\ m_2 \\ m_1 \\ m_0 \\ m_{15} \\ m_{14} \\ m_{13} \\ m_{12} \\ m_{11} \\ m_{10} \end{pmatrix} \oplus \begin{pmatrix} m_5 \\ m_4 \\ m_3 \\ m_2 \\ m_1 \\ m_0 \\ m_{15} \\ m_{14} \\ m_{13} \\ m_{12} \\ m_{11} \\ m_{10} \\ m_9 \\ m_8 \\ m_7 \\ m_6 \end{pmatrix} = \begin{pmatrix} m_{15} \oplus m_9 \oplus m_5 \\ m_{14} \oplus m_8 \oplus m_4 \\ m_{13} \oplus m_7 \oplus m_3 \\ m_{12} \oplus m_6 \oplus m_2 \\ m_{11} \oplus m_5 \oplus m_1 \\ m_{10} \oplus m_4 \oplus m_0 \\ m_9 \oplus m_3 \oplus m_{15} \\ m_8 \oplus m_2 \oplus m_{14} \\ m_7 \oplus m_1 \oplus m_{13} \\ m_6 \oplus m_0 \oplus m_{12} \\ m_5 \oplus m_{15} \oplus m_{11} \\ m_4 \oplus m_{14} \oplus m_{10} \\ m_3 \oplus m_{13} \oplus m_9 \\ m_2 \oplus m_{12} \oplus m_8 \\ m_1 \oplus m_{11} \oplus m_7 \\ m_0 \oplus m_{10} \oplus m_6 \end{pmatrix}$$

We can visually conclude that

$$c_i = m_i \oplus m_{(i+10) \bmod 16} \oplus m_{(i+6) \bmod 16}, \qquad i = 0, 1, \dots, 15$$

or else

$$c_{i \bmod 16} = m_{i \bmod 16} \oplus m_{(i+10) \bmod 16} \oplus m_{(i+6) \bmod 16}, \qquad \forall i \in \mathbb{N}$$

It turns out (after experimentation) that if we **XOR** by members $c_2, c_4, c_6, c_8, c_{10}$ we take the sixth digit of the original message (m_6). The proof is given:

$$\begin{aligned}
& c_{(i+2) \bmod 16} = m_{(i+2) \bmod 16} \oplus m_{(i+12) \bmod 16} \oplus m_{(i+8) \bmod 16} \\
& c_{(i+4) \bmod 16} = m_{(i+4) \bmod 16} \oplus m_{(i+14) \bmod 16} \oplus m_{(i+10) \bmod 16} \\
& c_{(i+6) \bmod 16} = m_{(i+6) \bmod 16} \oplus m_{(i+16) \bmod 16} \oplus m_{(i+12) \bmod 16} \\
& c_{(i+8) \bmod 16} = m_{(i+8) \bmod 16} \oplus m_{(i+2) \bmod 16} \oplus m_{(i+14) \bmod 16} \\
\oplus\ & c_{(i+10) \bmod 16} = m_{(i+10) \bmod 16} \oplus m_{(i+4) \bmod 16} \oplus m_{(i+16) \bmod 16} \\
\hline
& \qquad\qquad m_{(i+6) \bmod 16}
\end{aligned}$$

That is,

$$c_{(i+2)\ mod\ 16} \oplus c_{(i+4)\ mod\ 16} \oplus c_{(i+6)\ mod\ 16} \oplus c_{(i+8)\ mod\ 16} \oplus c_{(i+10)\ mod\ 16} = m_{(i+6)\ mod\ 16}$$

Setting $u = i - 10 \Rightarrow i = u + 10$ the above relation is written as:

$$mu \bmod 16 = c_{(u+12) \bmod 16} \oplus c_{(u+14) \bmod 16} \oplus cu \bmod 16 \oplus c_{(u+2) \bmod 16} \oplus c_{(u+4) \bmod 16}$$

As we saw at the beginning, the left circular scrolling by 6 corresponds to the mathematical expression $\mathbf{m}_{(i+10) \bmod 16}$ while the left circular scrolling by 10 corresponds to the mathematical expression $\mathbf{m}_{(i+6) \bmod 16}$.

More generally, cyclic scrolling by α, $0 < \alpha < N$ in a number of N bits shifts the first α MSBs to the position of the first α LSBs. So the LSB (position 0) will now be the digit that was previously in position $(N-1)\alpha$. So the expression we obtained as a result for the digits of m as a function of the digits of c can be transformed into the expression:

$$m = (c << 4) \oplus (c << 2) \oplus c \oplus (c << 14) \oplus (c << 12) = D(c)$$

The above conclusion is experimentally confirmed by the script "ex3.py" as it terminates with **error flag = False**.

(Implementation: `code/ex3/ex3/ex3`**.py**)

Exercise 4 (2.5)

A shifting system could have perfect security if the key changed for each character in a random, uniform way.

In particular, let $E = \{E, D, K, M, C\}$ with

$$K = M = C = \{0, 1, ..., 23\}$$

$$E(m, k) = (m + k) \bmod 2 = c$$

$$D(c, k) = (c - k) \bmod 2 = m$$

Given a text t of length ℓ characters. If $m_0, m_1 \in M$ any two characters of text $t \in M^{\ell}$ the perfect security equation holds:

$$Pr(k \xleftarrow{\$} K : E(k, m_0) = c) = Pr(k \xleftarrow{\$} K : E(k, m_1) = c)$$

since each encrypted character c can be the result of a shift by any number of k positions.

In conclusion, it follows that each specific letter will not be converted to another specific letter, but to any other letter for each separate occurrence (so there is no risk in terms of finding the key with statistical methods etc.).

This implementation of a displacement system is similar to the One Time Pad.

Exercise 5 (2.6)

The table for converting symbols to 5-bit binary sequences is implemented as a dictionary (specifically, two dictionaries for symbol-to-bits and bits-to-symbol conversion).
The encryption function is the same as the decryption function

$$D(x, y) = E(x, y) = x \oplus y$$

as is evident in the code.
The text to be used as a message is in the file **"sample text.txt"**.

Note that the key is chosen pseudorandomly using the random library. In fact in OTP the key is actually random, but because the key must be the same length as the text, choosing a key manually is overkill in this case.

The program displays (in order) the random key, the ciphertext and the decrypted text, which in each case is the same as the contents of the file.

(Implementation: `code/ex5/ex5/ex5`**.py**)

Exercise 6 (3.6)

The exercise was implemented in Python.

Function **is prime(n: int)**, returns in time $O(\sqrt{n})$ if n is prime.

Function **prime factors(n: int)**, returns a list $[p_1, p_2, ..., p_k]$ where

$$\prod_{i=1} p_i = n, \qquad p_i \text{ prime numbers}$$

Function **divisors(n: int)**, returns a list of all positive divisors of n.

Function **mu(n: int)**, implementation of the Möbius function, $\mu(n)$.

Function **N 2(n: int)**, the objective of the exercise.

The program picks 10 random integers, $\mathbf{n_i}$, in the interval $[1, 100]$. It prints, in the following order, its divisors, its prime factors and the number of reduced polynomials of degree n_i in the corpus F_2 . The following results are printed for $n = 30$:

```
[30] info:
-- divisors:    [1, 2, 3, 5, 6, 10, 15, 30]
-- prime factorization:    [2, 3, 5]
-- N 2(30) = 35790267
```

(Implementation: `code/ex6/ex6/ex6`**.py**)

Exercise 7 (3.8)

As in previous exercises, the script uses dictionaries for the encodings and decodings of the 32 given characters in 5-bit binary sequences (as shown in the table in exercise 3.5) and, of course, the implementation of XOR.

The RC4 algorithm requires 8-bit character encoding, but our characters are encoded in 5 bits each. A first thought would be to padding three zeros in front of the binary representations of the characters. But this is not a solution to the problem as if S[i] > 31 for some i, then the result of XORing a character with S[i] would cause overflow.

So RC4 was simply implemented with a limit of 32 instead of 256. So S is $S = [0, 1, ..., 31]$ instead of $S = [0, 1, ..., 255]$, the indices i, j are reset every 32 iterations instead of 256, etc.

The same function rc4() is used for encryption and decryption, which implements the initialization algorithm, the transposition scheme and the cryptographic algorithm. at the same time.

For message "MISTAKESAREASSERIOUSASTHERESULTSTHEYCAUSE" and key "HOUSE" get take as ciphertext "IGD!APO-TJUQPDSPDMAOZUIAZ(VF(VFGQ.IIWMB(WX". Putting the ciphertext into the same function and key, we get the original message again, which confirms that the encryption and decryption functions are the same.

(Implementation: `code/ex7/ex7/ex7`**.py**)

Exercise 8 (4.3)

The script implements the differential covariance calculation formula:

$$Diff\,(S) = \max_{x \in F_2^n - \{0\},\, y \in F_2^m} |\{z \in F_2^n : S(x \oplus z) \oplus S(z) = y\}|$$

For $n = 6$, $m = 4$ and S $\{0, 1\}^6 \to \{0, 1\}^4$ with S-box type (4.2.3), the program terminates by displaying Diff(S) = 14 as output.

Some information about the code:

List S i: implementation of the internal lookup table of function S

Function S(x: str) -> str: the function $S : \{0, 1\}^6 \to \{0, 1\}^4$

Function xor(x: str, y: str) -> str: returns $x \oplus y$

Differential uniformity function(...): accepts the function S and the length of its input strings. Returns $Diff\,(S)$

(Implementation: `code/ex8/ex8`**.py**, a copy of the code for the second question of homework 4 with minor changes)

Exercise 9 (4.7)

Exercise 4.7 requires the existence of an implementation of the AES crypt algorithm. Since there is no pre-installed Python package with this implementation, the **PyCryptodome** package **(v 3.17)** - (documentation link) was used. The installation was done at the command line with the command: pip install pycryptodome (Similarly, information about the conda environment is provided here).

All the tools in the library are in the **Crypto** module. It is mentioned in the documentation that PyCryptodome has problems if PyCrypto (old PyCryptodome) is installed at the same time. In this case PyCry- ptodomeX must be installed with pip install pycryptodomex. In case of installing this independent version the tools are located in the Cryptodome module.

Regarding AES, there are instructions in the documentation used to implement the exercise (as well as from another source to be mentioned later).

The file "messages.txt" contains 78 messages that will be used for the statistical test to determine the diffusion rate of the AES algorithm. The steps of the program are listed:

1. Messages are read from the file and stored in a table.
2. The messages are stored in the same order and their edited versions (a bit-flip) in another table.
3. A key of 16 bytes is generated (AES-128).
4. We create two AES objects with the key, one in ECB mode and one in CBC mode function.
5. A structure (zip object) is created with pairs (c_i, c_i') resulting from the encryption of (m_i, m_i') with AES-128 ECB mode, where m_i, m_i' are 256-bit messages that differ by one bit. We do the same for the CBC mode. Note that the non-CBC mode messages are less than 32 bytes so padding is done using Padding.pad (For this topic, code from this post on Stack Overflow was used).
6. A file is created in which the percentage of different bits between c_i and c_i' (by XORing and counting the number of aces in the result) for each $i = 1, 2, ..., 78$. Also recorded is the average digit change both in the file and on the screen. This procedure is done twice, once for each AES operation.

It is observed that AES-128 is not characterized by high diffusion in ECB mode as the average percentage of digits affected in the cipher message is about 25%, i.e. $< 50\%$.

Otherwise if the cryptalgorithm is run in CBC mode then the result changes drastically, with about 50.5% 51% vote change. Therefore the snowball effect is observed in the Rijndael algorithm, but not in all of its functions.

(Implementation: `code/ex9/ex9/ex9`**.py**)

Exercise 10

1. We import **course-1-introduction.pdf** into this PDF Metadata Viewer.
2. Copy the Base64 string located at the XMP → TIFF:ARTIST

   aHR0cHM6Ly9jcnlwdG9sb2d5LmNzZC5hdXRoLmdyOjgwODAvaG9tZS9wdWIvMTUv

   and input it into a decoder. Thus, the link is formed:

   https://cryptology.csd.auth.gr:8080/home/pub/15/

3. The website says "Chemist Walter White found the following message in his laboratory: #2-75-22-6!". The given string is not the code, but if we make some correlation with chemistry, we conclude that these numbers are atomic element numbers, hence (2, 75, 22, 6) (*He, Re, T i, C*). Therefore, by entering in secure.zip the code **#heretic!**, then this opens.
4. Inside secret.zip is secret.txt. Decoding the large Base64 string results in a picture of Bobby Fischer playing chess. In the metadata of the photo is a tinyurl.
5. The website writes (in algebraic form) the current state of a chess game. Making prompt the ChatGPT with command:

   Calculate the next best move:
   1.e4 e5 2.f4 exf4 3.Bc4 g5 4.Nf3 g4 5.O-O gxf3 6.Qxf3 Qf6 7.e5 Qxe5
   8.d3 Bh6 9.Nc3 Ne7 10.Bd2 Nbc6 11.Rae1 Qf5 12.Nd5 Kd8
   13.Bc3 Rg8 14.Bf6 Bg5 15.Bxg5 Qxg5 16.Nxf4 Ne5 17.Qe4 d6 18.h4 Qg4
   19.Bxf7 Rf8 20.Bh5 Qg7 21.d4 N5c6 22.c3 a5 23.Ne6+ Bxe6
   24.Rxf8+ Qxf8 25.Qxe6 Ra6 26.Rf1 Qg7 27.Bg4 Nb8

   this answers:

   In this position, White has a strong initiative, but Black is still defending well.
   28.Rf7
   ...

6. We compute hashlib.md5(b'Rf7').hexdigest() which is equal t o :

   f1f5e44313a1b684f1f7e8eddec4fcb0 which is indeed the key to secure2.zip. The mastersecret.txt contains the hash as the answer:

   be121740bf988b2225a313fa1f107ca1