Homework 1 – Classical Cryptography

Cryptography and Security 2023

- You are free to use any programming language you want, although Python/SAGE is recommended.
- Put all your answers **and only your answers** in the provided `[id]-answers.txt` file where `[id]` is the student ID.[1] This means you need to provide us with all `Q`-values specified in the questions below. Personal files are to be found on Moodle under the feedback section of Parameters HW1.
- **Please do not put any comment or strange character or any new line** in the submission file and do **NOT** rename the provided files.
- Do **NOT** modify the `SCIPER`, `id` and `seed` headers in the `[id]-answers.txt` file.
- **Submissions that do not respect the expected format may lose points.**
- We also ask you to submit your **source code**. This file can of course be of any readable format and we encourage you to comment your code. Notebook files are allowed, but we prefer if you export your code as normal textual files containing Python/SAGE code. If an answer is incorrect, we may grant partial marks depending on the implementation.
- Be careful to always cite external code that was used in your implementation if the latter is not part of the public domain and include the corresponding license if needed. Submissions that do not meet this guideline may be flagged as plagiarism or cheating.
- Some plaintexts may contain random words. Do not be offended by them and search them online at your own risk. Note that they might be really strange.
- Please list the name of the **other** person you worked with (if any) in the designated area of the answers file.
- Corrections and revisions may be announced on Moodle in the "News" forum. By default, everybody is subscribed to it and does receive an email as well. If you decided to ignore Moodle emails, we recommend that you check the forum regularly.
- The homework is due on Moodle on **October 28, 2023** at 23h59.

[1] Depending on the nature of the exercise, an example of parameters and answers will be provided on Moodle.

Exercise 1 It's only fair play

You are a newly appointed cryptography apprentice in some fancy company, who was badly lacking a cryptography expert beforehand. Indeed, last week, the company's entire financial data was leaked due to some badly chosen encryption mechanism (Really, who uses Caesar nowadays...?). The entire company was put through a "Cryptography 101" workshop to learn the basics and started brainstorming on a new encryption algorithm to replace their previous one. Opening the drawer of an old desk in a forgotten room, they found the description of an encryption mechanism along with some examples and messages. The workshop has however managed to teach them caution, and before rolling it out as their new standard, they've put you in charge of analysing the following encryption scheme, together with the clues they found.

Key Generation:

1. Choose a password or passphrase.
2. Drop any duplicate letters.
3. Arrange it into a 5×5 square, row by row from left to right, and fill the remaining space with the other letters of the alphabet. Note that we identify I and J as one and the same.

Encryption

1. Break down the message into digrams.
2. If the message length is odd, append an X. If any letter repeats itself, separate them with an X.
3. Substitute every digram according to the following rules, in order :
   (a) Consider the two letters forming the digram as opposite corner of a rectangle in the table.
   (b) If the two letters are on the same row, replace them by the value to their right, wrapping around the row.
   (c) If the two letter are on the same column, replace them by the value immediately below, wrapping around.
   (d) If they are neither on the same row nor column, replace them by the letter on the same row forming the other pair of corner of the rectangle they define (via a y-axis reflection).

Along with this description, an example was provided.

Example: The original passkey is "foolishcompany" which gives rise to the following square :

$$\begin{pmatrix} f & o & l & i & s \\ h & c & m & p & a \\ n & y & b & d & e \\ g & k & q & r & t \\ u & v & w & x & z \end{pmatrix}$$

If one wants to encrypt "caesar", one decomposes it into digrams "ca","es", "ar".

1. "c" and "a" are on the same line, hence both letters are replaced by the ones to their right i.e. "c" becomes "m" and "a" becomes "h" (since we are wrapping around).
2. "e" and "s" are on the same column, hence they are replaced by the letter right below them i.e. "e" becomes "t" and "s" becomes "a".
3. "a" and "r" are neither on the same line, nor the same column. Hence one forms a rectangle with the opposite corners and "a" is encrypted as "p" and "r" as "t".

The resulting ciphertext is "mhtapt".

If one wants to encrypt a longer ciphertext, they concatenate all the words and apply the above encryption rules.

Question 1.1 Encryption

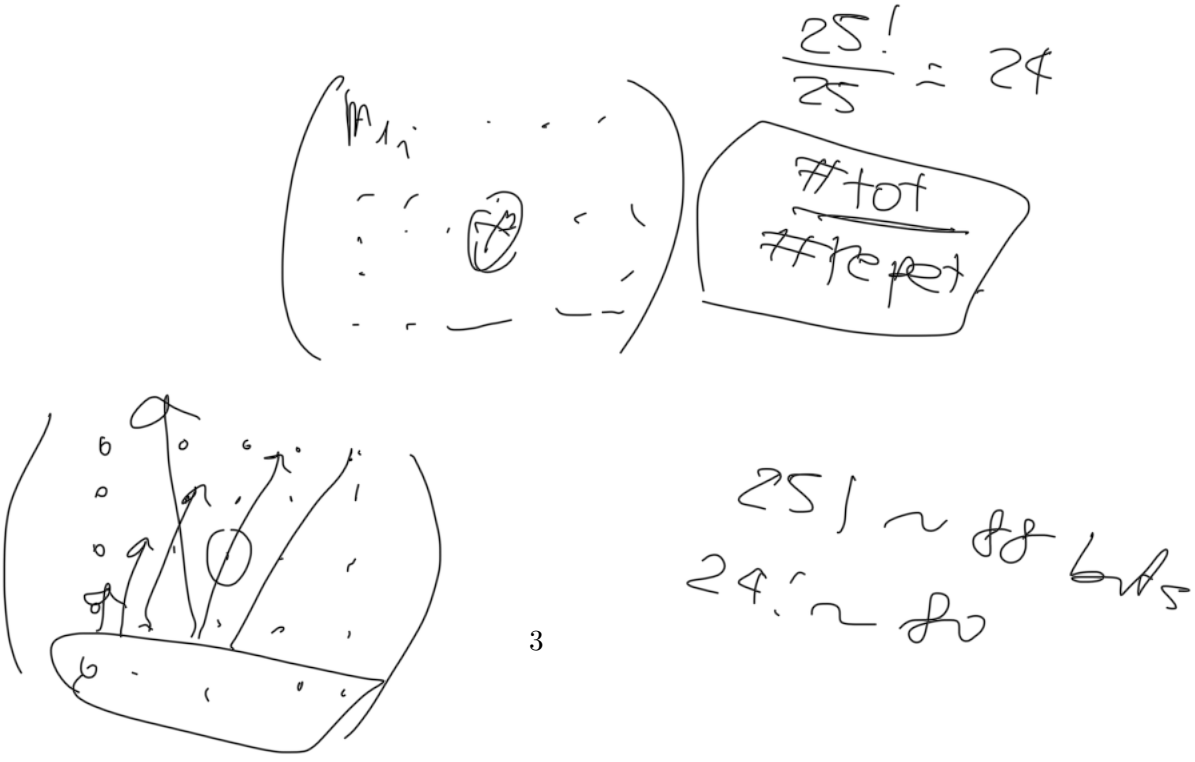
Encrypt the text given under Q1a_m with the key given under Q1a_k. Report it as a **string** under Q1a_c.

Question 1.2 Decryption

Figure out what the decryption algorithm should look like and decrypt the ciphertext given under Q1b_m with the key given under Q1b_k. Report it as a **string** under Q1b_m. You do **not** need to remove the added "x" if any.

Question 1.3 Cryptanalysis

What is the total number of possible non-equivalent keys ?
We call two keys equivalent if they give rise to the same ciphertexts.
Report it under Q1c_k as a **string** containing a symbolic expression, i.e. do not round your value but rather write it exactly. For example, you ~~should write~~ 1/3 ~~instead of~~ 1.33333.



Exercise 2 Cube Cipher

The crypto apprentice recently got interested in speedcubing and after playing with a Rubik's Cube for a while, CA realized that the state of the cube gets mixed up quite fast and wondered if this would be a good candidate for a key generator. If the key generator is good enough, the CA can use Vigenère cipher to encrypt some messages.

The CA has learned the following moves in Figure 1:

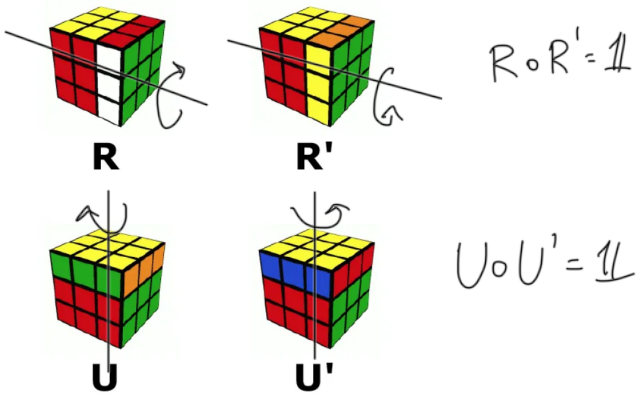


Figure 1: The moves that the CA knows.

- **R**: Moves the right face in clockwise direction.
- **U**: Moves the top face in clockwise direction.
- **R'**: Moves the right face in counter clockwise direction.
- **U'**: Moves the top face in counter clockwise direction.

Later, the CA encrypts messages using the Vigenère Cipher with the following key generation algorithm:

1. **Setup**: Write random letters from the english uppercase alphabet `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"` on each square on the cube.
2. **Key Generation**: Execute the following moves: `R U R' U'`. Read the upper right corner from the top face. For the next key execute the same moves but read lower right corner from the top face. Repeat this procedure for all characters.

Example: Assume we would like to encrypt `"HELLO"`

1. **Setup**: We run the setup algorithm as described above and obtain the following state in Figure 2.

Figure 2: State of the cube after setup.

2. **Key Generation**: We illustrate the execution above in Figure 3.

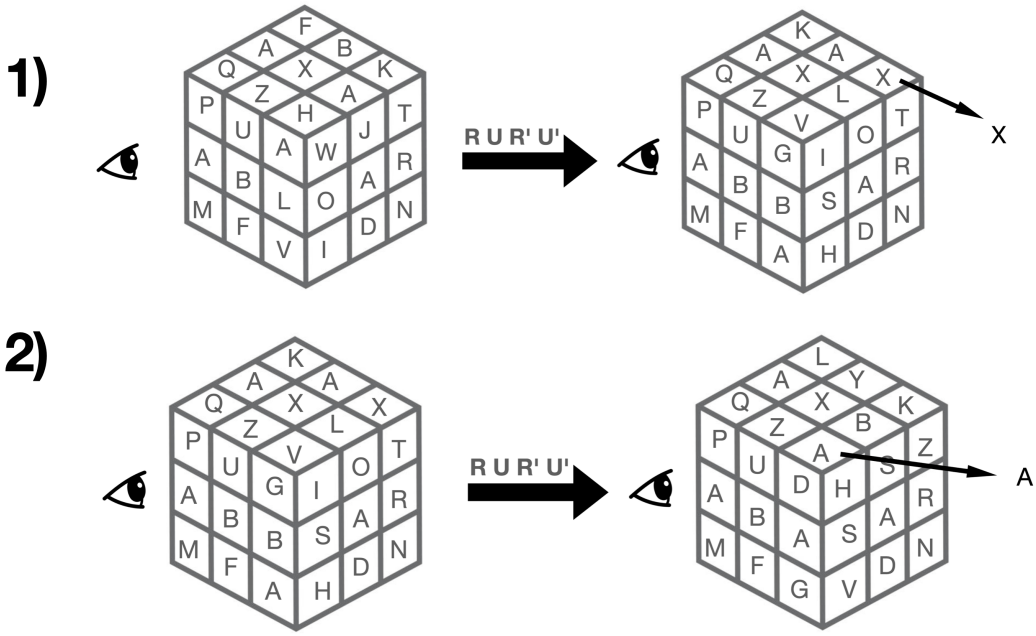


Figure 3: Key Generation Procedure.

Note that we first encrypt 'H' with 'X' which is what we have read from the upper right corner of the top face. Next, we encrypt 'E' with 'A' which is what we have read from the lower right corner of the top face. We continue this procedure and alternate between these two corners while executing `R U R' U'` in between. This behaves like a Vigenère cipher where the key is generated freshly for each character.For 'LL' we generate another sequnce of keys and proceed similarly (Execute `R U R' U'` encrypt 'L' with the character from upper right corner, execute `R U R' U'` encrypt the second 'L' with the character from lower right corner.), for the last 'O' we apply the same procedure and read the key from the upper right corner.

Things to note:

- To solve the exercise, you will be using an additional python library that can be installed via `pip install rubik-cube`.
- The given implementation file `cubecipher.py` will include sufficiently commented code that lets you understand how to use the library.
- The plaintext might contain non-english characters. Anything that is not included in the English upper case alphabet is skipped both during encryption and decryption (e.g. The ciphertext for `"HELLO WORLD-!"` would look like `"XYABK DJFGL-!"`).

Question 2.1

The implementation in `cubecipher.py` already includes the encryption procedure.

- ▷ Implement the decryption procedure for this cryptosystem. subtract
- ▷ Given a seed Q2a_seed and a ciphertext Q2a_c, return the plaintext Q2a_m. Note that this part does not include any cryptanalysis. You can check whether your solution is correct or not by using the checksum value Q2a_mhash. The assertion to check is:

```
hashlib.sha256(Q2a_m.encode()).hexdigest() == Q2a_mhash
```

Question 2.2 Cryptanalysis

- ▷ Given a ciphertext Q2b_c, recover the plaintext Q2b_m and the key Q2b_k. You can check whether your solution is correct or not by using the checksum value Q2b_mhash. The assertion to check is:

```
hashlib.sha256(Q2b_m.encode()).hexdigest() == Q2b_mhash
```

$\texttt{encrypt\_2bits}_{g,p}(K_{EXP}, m_j \in \mathbb{Z}_4, j)$

1 : $K_{XOR} \leftarrow\!\!\$\ \texttt{fault\_key\_xor\_gen}() \in \{0,1\}^{256}$ // bits at position Q3_n and Q3_n + 1 are constant
2 : $r \leftarrow\!\!\$\ \{0, \ldots, p-1\}$
3 : $c \leftarrow K_{XOR} \oplus (g^{K_{EXP}+52r+m_j+2(j+1)} \bmod p)$
4 : **return** c

$\texttt{encrypt\_key}_{g,p}(K_{EXP} \in \mathbb{Z}_{2^{128}}, K_{AES} \in \mathbb{Z}_{2^{128}})$

1 : $j \leftarrow 0$
2 : cts $\leftarrow []$
3 : // Bits of the AES key are encrypted two by two, from LSB to MSB
4 : **while** $K_{AES} > 0$:
5 : $\quad m_j \leftarrow K_{AES}$ & 0b11 // & stands for the logical AND gate
6 : $\quad c \leftarrow \texttt{encrypt\_2bits}(K_{EXP}, m_j, j)$
7 : $\quad$ cts$[j] \leftarrow c$
8 : $\quad j \leftarrow j + 1$
9 : $\quad K_{AES} \leftarrow K_{AES} \gg 2$
10 : **return** cts

Figure 4: Encryption functions for Exercise 3.

Exercise 3 Order, order!

The federal elections are approaching and the Swiss government, in its infinite wisdom, mandated the Crypto Apprentice (CA) to design the brand new e-voting system that will be used. Having followed the drama surrounding the topic, the CA decided to rebuild a system from scratch based on simple cryptographic primitives. The voting procedure works as follows:

1. The voter chooses a list L of two candidates to elect, which is converted to a bytes object.
2. This object is then encrypted with AES-GCM using some 128-bit key K_{AES}.
3. The AES key K_{AES} is then converted to an integer and encrypted using the `encrypt_key` routine defined in Figure 4, where p and $g \in \mathbf{Z}_p$ are some public parameters, and K_{EXP} is a secret key. The `encrypt_key` function encrypts two bits of K_{AES} at a time using the `encrypt_2bits` sub-routine defined in Figure 4.
4. The final ciphertext is comprised of the AES ciphertext (with tag and nonce), and the list of the 64 ciphertexts output by `encrypt_key`.

However, the CA uncharacteristically made an awful mistake in the implementation of the function that samples the K_{XOR} key (called `fault_key_xor_gen` in Figure 4). That is, two bits of K_{XOR} are always constant! More precisely, if we consider K_{XOR} as a bitstring of 256 bits $b_0 b_1 b_2 \ldots b_{255}$ where b_0 is the **most** significant bit, then the bits at position Q3_n and Q3_n + 1 are constant, where Q3_n is provided in your parameter file.

Being aware of this flaw, you decide to make the most of it by trying to decrypt your flatmate's ballot. You will finally discover whether they are a respectable crypto-anarchist as they have always claimed to be or, on the contrary, whether you should terminate their lease

and your friendship immediately.

You start straightaway to tap your home network, and you manage to intercept some data Q3_known_cts := encrypt_key(Q3_known_pt) generated by your flatmate's computer in a *previous session* (note that K_{EXP} stays the same across sessions).

▷ Given the encrypted ballot (Q3_ct_aes, Q3_nonce, Q3_tag, Q3_cts), the intercepted values from the previous session Q3_known_cts, Q3_known_pt, the value Q3_n, and the parameters (Q3_p, Q3_g), your goal is to recover the original list of candidates, which is a *string* of the form "Firstname1 Lastname1, Firstname2 Lastname2". Report it under Q3_pt.

HINT1: The goal is to recover K_{AES}; once you have it, recovering the plaintext is simply decrypting with AES-GCM, knowing that the integer-representation of the key is converted to bytes with big endian representation. That is, once you have the integer representation of K_{AES}, you can use the following code snippet to decrypt (after installing the `pycryptodome` package):

```
from Crypto.Cipher import AES
def decrypt(Q3_ct_aes, Q3_tag, Q3_nonce, rec_k_aes):
    # rec_k_aes is assumed to be an int
    rec_k_aes_bytes = rec_k_aes.to_bytes(16, "big")

    cipher = AES.new(rec_k_aes_bytes, AES.MODE_GCM, Q3_nonce)
    Q3_pt = cipher.decrypt_and_verify(Q3_ct_aes, Q3_tag)
    return Q3_pt
```

HINT2: What can you say about g?

Goal: recover K_{AES}

plaintext = $dec_{AES\text{-}GCM}$