INF143A 23V Applied Cryptography
Second mandatory assignment

Nikolay Kaleyski

1 General information

- **Deadline:** The assignment is due on **April 7, 2023**, at 23:59.
- **Submission:** A copy of the solution should be uploaded to mitt, under the "Assignments" tab. Please put all of your files in a single ZIP archive.
- **Score:** The mandatory assignment accounts for 15% of the final grade.
- **Collaboration:** You can freely discuss the assignment with each other, but the solutions must be prepared individually; plagiarism will result in all involved parties failing the assignment.
- **Passing grade:** You must obtain at least 20% to successfully pass the assignment.

2 Problems

Important: For all problems, please ensure that the output provided by your program is in the same format as that of the sample output files. This is because your solution will in part be graded by automatically comparing its output against sample output produced by our implementation. Having extra spaces, commas, newlines, etc. will result in a mismatch. Thus, to get maximum points and avoid any possibility of a misunderstanding, please ensure that your output **exactly matches the format of the sample output files**.

Problem 1. ***40 %*** *Consider the block cipher from Problem 3 of the first mandatory assignment, which has a block size of 16 bits, so that its input and output are both 16 bits long. You will find an implementation of the cipher in* `cipher.py`*, and you can use the* `encrypt(P,K)` *and* `decrypt(C,K)` *functions to encrypt and decrypt a single block of bits, respectively. You can also use your own implementation from the previous assignment.*

You will also find the auxiliary functions `bytes_to_bits` *and* `bits_to_bytes` *as well as* `read_file` *and* `write_file` *which you can use to read or write a file in binary, and convert the binary contents to a list of bits, and vice-versa.*

Your goal is to ***extend the implementation so that it can handle inputs of arbitrary size*** *using the* ***CBC mode of operation****. Your implementation should be a pair of programs,* `enc` *and* `dec`*, which can be run like*

```
python enc plaintext key iv output_file
```

in order to encrypt a plaintext file with a given key and initialization vector, or

```
python enc ciphertext key iv output_file
```

in order to decrypt.

Sample data: *You will find the sample plaintext* `moo_in`*, key* `moo_key` *and* `moo_iv`*, and the corresponding ciphertext* `moo_out`*.*

NB: *You can assume that any plaintext or ciphertext that you get as input will have a size exactly divisible by 16, so that it can evenly be divided into blocks without any padding. In a real implementation of such a cipher, the last block might need to be padded if its size is less than 16 bits.*

Problem 2. ***30 %*** *Implement the* ***Elgamal digital signature scheme****. You should write a program* `Elgamal` *which can be run like*

```
python elgamal parameters private_key message output.
```

The `parameters` *file should be a text file, which on the first three lines should contain the public parameters p, g, and β, respectively. The* `private_key` *file should also be a text file consisting of a single line; this will be the private key d. The* `message` *file should contain a single number, which is the message to sign. Your program should record the signature (r, s) in the* `output` *file, with r on the first line and s on the second line.*

You do not have to implement functionality for verifying signatures, but you should keep in mind that this is how your program will be tested, i.e. by asking it to sign a series of randomly generated message with the given parameters, and then verifying whether the signatures are correct.

Sample data: *You will find two sets of parameter files:* `elg_parameters`*,* `elg_private`*; and* `elg_parameters_2` *and* `elg_private_2`*. Signing the message in* `elg_message` *with each set of parameters yields the signatures in* `elg_output` *and* `elg_output_2`*, respectively.*

NB: *Depending on the choice of the ephemeral key (which is randomized), it is possible to get different signatures. The ones in the sample files above are just examples.*

Problem 3. ***30 %*** *Implement a program computing the differential uniformity of a given (n, n)-function. The function is given as a lookup table where every line is in the format "x->f(x)". You can see an example of such a lookup table in the files* `sample1.tt` *and* `sample2.tt`*. You should write a program* `du` *which can be run like*

```
python du lookup_table
```

and will output a single number, which is the differential uniformity of the function represented by this truth table.

Sample data: *The functions represented by* `sample1.tt` *and* `sample2.tt` *have differential uniformity 2 and 8, respectively.*