504 Public Key Cryptography **Spring 2020-2021**
Homework #2 **Due: 30.04.2021**

Name: Sıla ÖZEREN
Student ID: 215808

Question 4.

Explain how you encoded the message to and from integers. What happens if the message length is larger? How would you divide the message into blocks? What are the shortcomings of this encoding? What is wrong with using RSA in this way? (A paragraph or two is enough.)

My plaintext:
"**I say let the world go to hell, but I should always have my tea. :):):):):):) :)**"

Answer: At first, I thought reading the plaintext message was a piece of cake. Since the message contains meaningful characters, which can be read by humans, using ifstream and treating my message as a string, then casting it to a constant char variable and plugging it into the mpz_import function felt like an easy task. So, I tried to follow the same logic while reading the ciphertext from the cipher.txt file but, it was a disaster. There are two interrelated reasons why reading ciphertext was a huge fail. First one, I did not know that there were some characters in ASCII table that are not meaningful to human eye and cannot be interpreted as string of characters. Let us see a few of these characters,

0 NUL (null)
1 SOH (start of heading)
2 STX (start of text)
3 ETX (end of text)
4 EOT (end of transmission)
5 ENQ (enquiry)
6 ACK (acknowledge)
7 BEL (bell)
8 BS (backspace)
9 TAB (horizontal tab)
10 LF (NL line feed, new line)

The second reason is that since my text-editor cannot read specific characters, and my ciphertext contains nothing but bunch of question marks in little boxes, I thought there was a mistake in encoding. Until Bartu Hoca warned me, I wasn't even questioning if my ciphertext could contain some characters that cannot be interpreted as a string. To demonstrate this, let us give an example. Suppose that

my ciphertext contains the "start of text" character, then imagine me treating this character as a string and trying to read it. It was impossible and was causing me a loss of my data. So, I stopped this approach and started to use the fopen, fread and fwrite functions offered by C++. These functions allowed me to read every character so that I could perform encoding and decoding perfectly fine.

The hard part was figuring out why the length of the plaintext was longer then the length of n. It took a lot of time to figure out. Here is the first version of generating prime parameters p and q,

```
gmp_randstate_t grt;
gmp_randinit_default (grt);
gmp_randseed_ui (grt, time (NULL));
mpz_class key_p, key_q;
mpz_urandomb (key_p.get_mpz_t(), grt, 512);
mpz_nextprime (key_p.get_mpz_t(), key_p.get_mpz_t());
```

The problem is I thought mpz_urandomb function was giving me exactly 512-bits long integer. Later, I check the document and saw this piece of information. "Generate a uniformly distributed random integer in the range 0 to 2^{n-1}, inclusive." At that point, I was enlightened and it was no longer a mystery why my program kept aborting due to this piece of code I've shown below.

```
if (mpz_cmp(pt.get_mpz_t(), n.get_mpz_t()) > 0){
            abort(); }

// pt for plaintext and n is the public modulus
```

Thus, I made sure that my primes has exactly 512 bits. Since n has 1024 digits, and we want our plaintext to be as big as possible, plaintext could contain at most 128 bytes, which is 1024. If plaintext is bigger than n, as a numeric value, encryption cannot be done uniquely, which is a serious problem. So, any text which has more than 128 characters, should be sliced into blocks of 128 bytes. However, since we do not shuffle and permute these blocks, this will result in loss of cryptographic properties such as diffusion. This is why we are using AES for fast and secure encryption whereas RSA is used as a virtual handshake.