Title

Kent Odde

October 24, 2020

Contents

Abstract

This is my submission for the home exam in Discrete Mathmatics, fall 2020.

Introduction

The assignment consists of implementing the ElGamal encryption algorithm, in a programming language of my own choosing. As a requirement for the implementation is speed, I found the most suitable language for the task to be C++. Luckily this is also the language I'm most proficient in.

One downsize of using C++ is the fact that there exists no built in support for large numbers. Building my own library for this would be to time-consuming, so I decided to make use of the Gnu Multiple Precision library (GMP)[3].

Quite a bit of overhead were required to encrypt larger inputs. However, because this course is concerned with the mathematics behind ElGamal and the fact that the professor wanted a short paper, I will not cover this part of the implementation to a detailed extent.

ElGamal

ElGamal is an asymmetric encryption scheme based on Diffie-Hellman. That it is asymmetric means that the algorithm uses two different keys for encryption and decryption. As encouraged by the professor, I have used the Wikipedia article[2] as my main source of information on ElGamal. Instead of reiterating the details of the algorithm from the article, I will just focus on the main points here.

We choose a cyclic group G, with a generator g. The private key is denoted by x. We use the private key to generate a part of the public key, h:

$$h := g^x \tag{1}$$

If we choose a bad g, the discrete logarithm problem which gives ElGamal its security, will decrease in complexity.

Implementation

Overhead

Padding

Asymmetric encryption algorithms like RSA and ElGamal, requires a padding scheme in order to make the length of all strings uniform and henceforth increase the security. A common padding scheme for RSA is PKCS#1.5. As it was hard to find sources for padding schemes commonly used with ElGamal, my choice also fell on PKCS for this implementation.

A visualization of the scheme can be seen in figure 1.

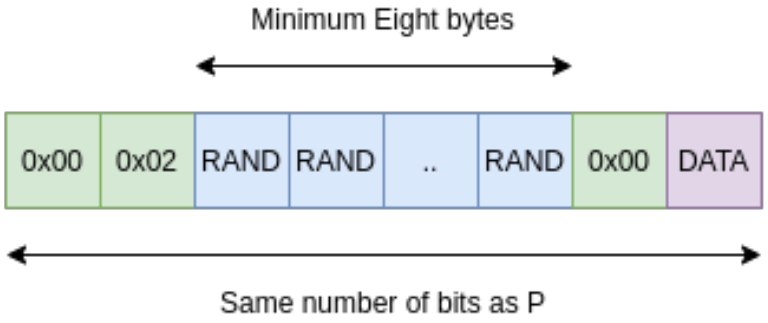

Figure 1: PKCS padding scheme

Block Handling

ElGamal is not a common choice for encrypting larger texts, however after conversations with the lecturer, I understood that this was a requirement for this assignment. To keep the overhead to a minimum, ECB(Electronic Codebook) was chosen. An illustration of ECB can be seen in figure 2. If the plaintext is larger than the number of bits of P, we divide it up into blocks, encrypt them seperately, and concatenate them.

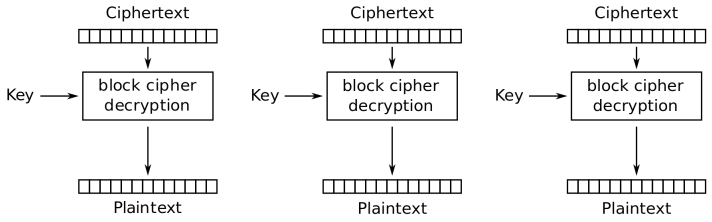

Figure 2: Electronic Codebook[1]

An extra complication with ElGamal is that the algorithm produces two numbers, in which the size of bits may vary. In order to be able to parse the concatenated strings later, I decided to prepend zeroes to c1 and c2, until their length were at 512 bits.

Putting the padding together with Elgamal and and the post-encryption padding the full overview of this implementation encrypts a string, can be seen in figure 3

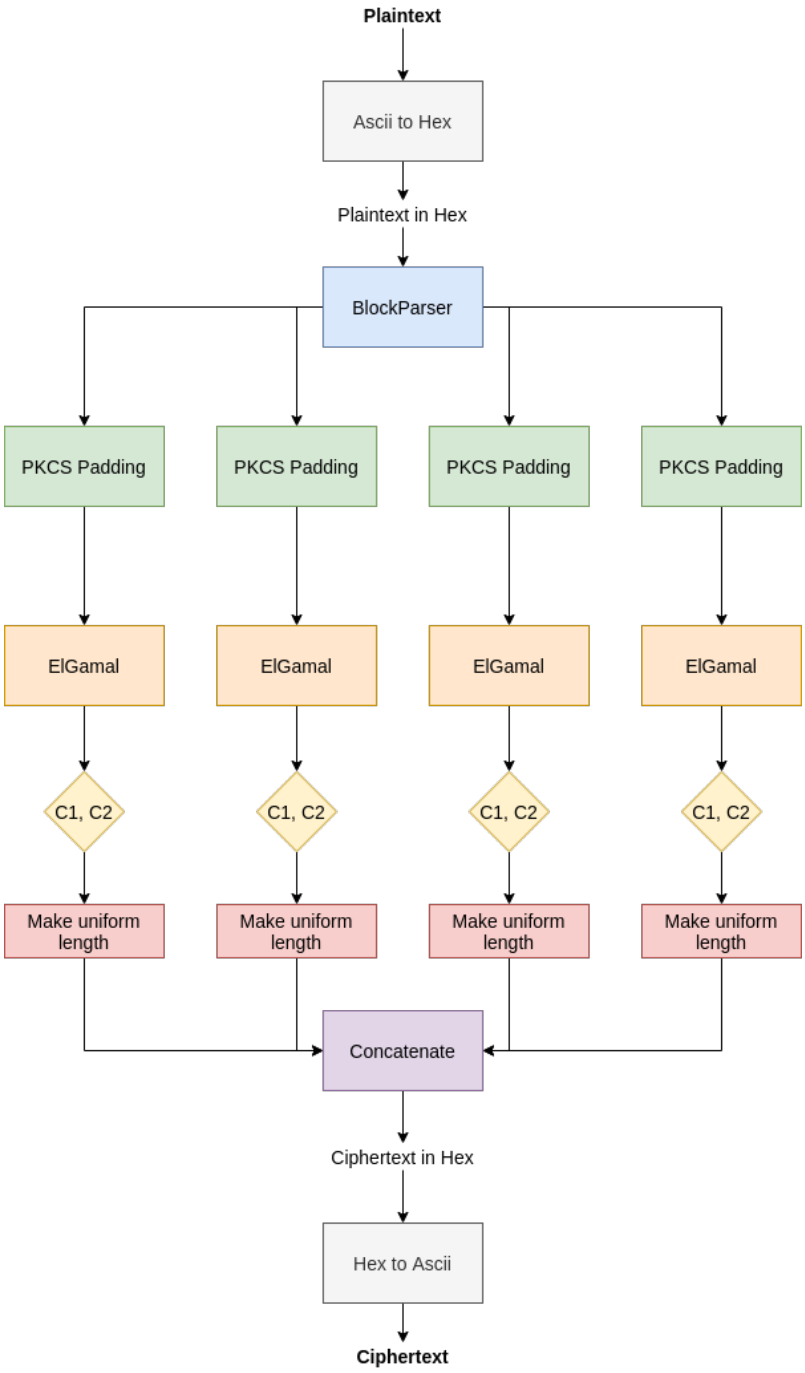

Figure 3: Overview of the overhead

ElGamal

Generating Keys

$$G = (\mathbb{Z}/p)^* \tag{2}$$

$$q = ord(G) = ord(\mathbb{Z}/p)^* = p - 1 \tag{3}$$

$$privateKey = x \in \{1, ..., q-1\} \tag{4}$$

$$h := g^x \tag{5}$$

$$publicKey = (G, q, g, h) \tag{6}$$

Finding and choosing a generator

$$Y = g^x \bmod p \tag{7}$$

For every value of x, we should get a unique value of Y

All generators are factors of p-1

Encryption

We choose a random number y, from the set

$$\{1, ..., q-1\} \tag{8}$$

Decryption

Conclusion

Appendices

References

[1] ECB, wikimedia-commons. https://commons.wikimedia.org/wiki/File:ECB_decryption.svg. Accessed: Oct-20.

[2] Elgamal, wikipedia. https://en.wikipedia.org/wiki/ElGamal_encryption. Accessed: Oct-20.

[3] Gnu Multiple Precision Library, gmplib. https://gmplib.org/. Accessed: Oct-20.

[4] RSA & PKCS, di-mgt. https://www.di-mgt.com.au/rsa_alg.html. Accessed: Oct-20.

[5] Kenneth H. Rosen. *Discrete Mathematics & its Applications, Modular exponentiation.* McGraw-Hill, 2019.