

Title

Kent Odde

October 19, 2020

USN

Contents

Abstract	3
Introduction	3
Asymmetric Encryption	3
ElGamal	3
Implementation	3
Overhead	3
Padding	3
Block Handling	4
ElGamal	6
Generating Keys	6
Encryption	6
Decryption	6
Conclusion	6
Appendices	6
References	7

Abstract

This is my submission for the home exam in Discrete Mathematics, fall 2020. The assignment consists of implementing the ElGamal encryption algorithm, in a programming language of my own choosing.

Quite a bit of overhead were required to encrypt larger inputs. Because this course is concerned with the mathematics behind ElGamal and to satisfy the professors requirement for a short paper, I will not cover this part of the implementation to a detailed extent.

Introduction

Asymmetric Encryption

ElGamal

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.

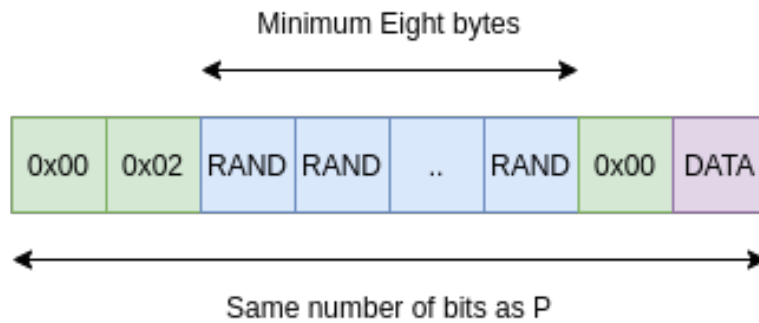


Figure 1: PKCS padding scheme

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 separately, and concatenate them.

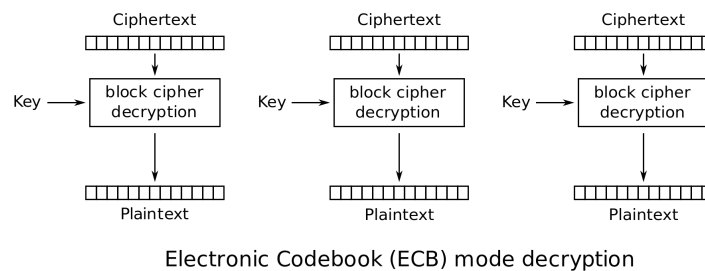


Figure 2: Electronic Codebook[1]

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 c_1 and c_2 , until their length were at 512 bits.

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

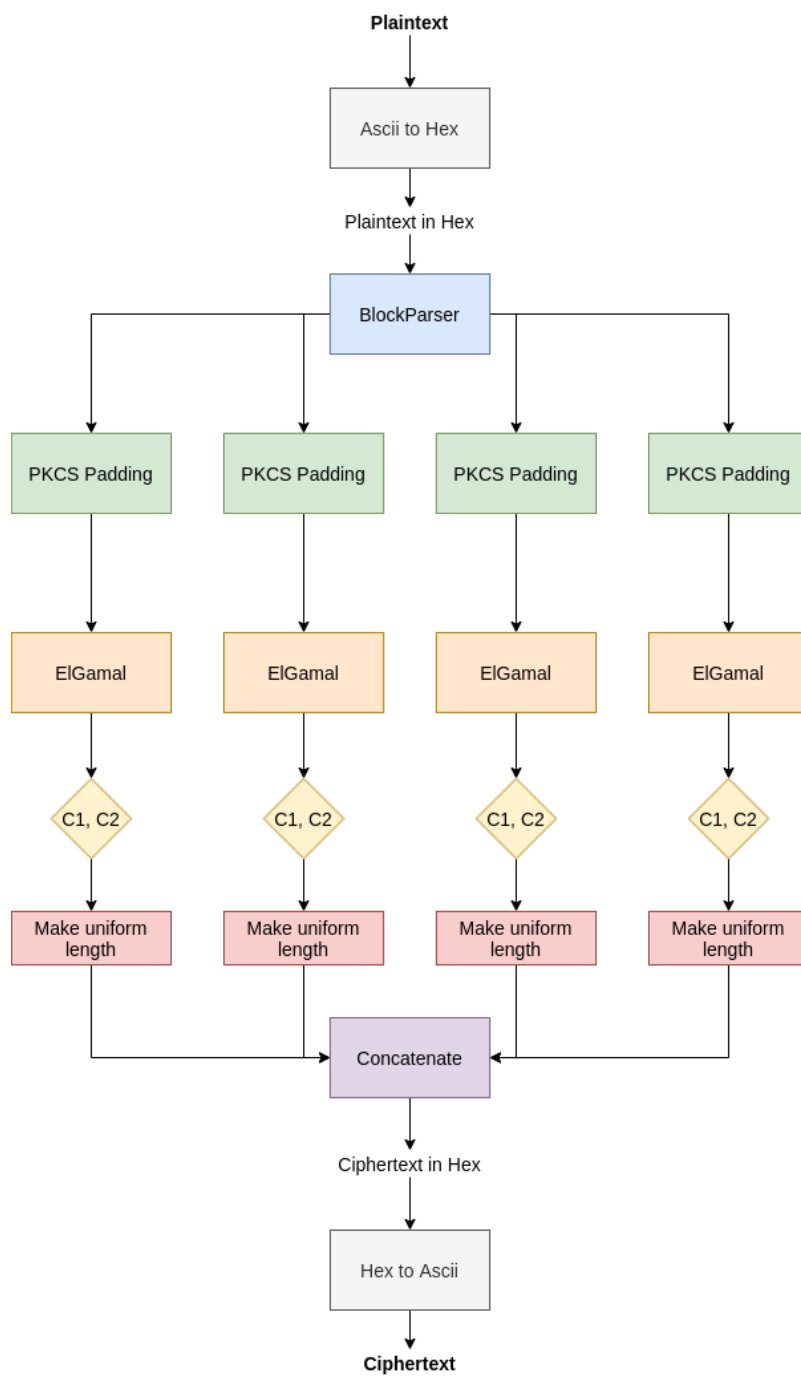


Figure 3: Overview of the overhead

ElGamal

Generating Keys

$$G = (\mathbb{Z}/p)^* \quad (1)$$

$$q = \text{ord}(G) = \text{ord}(\mathbb{Z}/p)^* = p - 1 \quad (2)$$

$$\text{privateKey} = x \in \{1, \dots, q - 1\} \quad (3)$$

$$h := g^x \quad (4)$$

$$\text{publicKey} = (G, q, g, h) \quad (5)$$

Finding and choosing a generator

$$Y = g^x \bmod p \quad (6)$$

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

$$\{1, \dots, q - 1\} \quad (7)$$

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, gmp. <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.