ECE 2420 Programming Exercise #3

(RSA Implementation)

# Overview

This programming exercise is the third and final cryptographic transform that will be implemented according to our specified interface. In this case, we will draw on the interface from the first PEX, the stream/buffer management and linking concepts from the second, and then implement a fully functional public key cryptographic system.

# RSA Cryptographic Algorithm

The RSA cryptographic algorithm truly revolutionized how the internet works. RSA can provide confidentiality, authenticity, and non-repudiation in a single algorithm. Beyond that, it can provide all these features with no pre-shared secrets. The Cormen book contains a nice introduction to the algorithm beginning on page 146. There is a simplified wiki page on the topic located at: <https://simple.wikipedia.org/wiki/RSA_algorithm>. A much more technical article is locate here: [https://en.wikipedia.org/wiki/RSA\_(cryptosystem)](https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29). In addition, you may desire to re-read the lecture notes for asymmetric key cryptography.

# Programming Concepts

In this exercise you are expected to understand and implement a real world public key cryptographic algorithm. A significant piece of this algorithm is the ability to manipulate very large integers. For this, there is a math library called GNU Multi-Precision Arithmetic Library which you are encouraged to leverage. It contains many traditional math functions you may be familiar with in addition to modular arithmetic functions which will be extremely helpful for this development effort. In particular, calculating the modular multiplicative inverse can be done using this library (extended Euclidean algorithm). Also, large power modulus operations and probabilistic prime creation are a breeze. Documentation for the GMP library can be found at: <https://gmplib.org/gmp-man-6.1.2.pdf>.

# System Requirements

This design is intended to be quite flexible not only in the transforms that are supported, but in the ways that the transforms are used. It would be equally easy to use this code to transfer data over a socket as it would be to read/write files on a disk. The system is designed to support encryption, decryption, and key management in a single simple library. Note that these requirements are identical to the last PEX; this is by design.

A typical use of this library could be as follows (secure data file storage):

1. The user instantiates a Crypto object. During this instantiation, the user is required to provide callbacks. The purpose of these callbacks is to provide instruction to the library on what do with data that has been successfully transformed, (encrypted or decrypted).
2. The static factory function parameters dictate what cryptographic transform shall be used.
3. The library will be used to generate keys for this action
4. The keys may be stored for use on a later execution, (e.g. to decrypt the file)
5. The file to be encrypted/secured is opened by the user and read into a memory buffer.
6. The contents of the memory buffer are then written to the encryption engine.
7. The encryption engine will transform the data appropriately. It may then do any of the following:
	1. Call the callback in which the user program can write the obfuscated data to disk.
	2. Call the associated callback with only a portion of the encrypted data, (some encryption algorithms need to handle data in specific block sizes.)
	3. Make no callbacks and buffer the data until a certain amount has been collected
8. Once all of the data has been written, one additional write shall be made to the engine where the length of the data is specified as zero. This will instruct the encryption engine to flush all buffers completely. Note that some algorithms may need padding to fill a complete block.
9. The entire process is executed in reverse to decrypt the file when needed.

Other requirements:

1. The system shall utilize the interface header file provided (Crypto.hpp) verbatim. This will allow the instructor to execute test/grading code using your implementation. This header file is located at: <http://classes.ece.usu.edu/2420/>. It is very well documented in-line and students are encouraged to study it closely.
2. The student may construct the implementation of the provided header in any way they choose.
3. Additional classes/files may be used as the student sees fit.
4. Students should consider building a new class derived from the provided interface for each new transform supported.
5. The system shall support transforms in which the size of the plain text and cipher text streams are not the same.
6. The system shall accept all data provided to an encrypt/decrypt function call.
7. The system may buffer and return encrypted/decrypted data via any number of callback executions.
8. The system may execute callbacks from the encrypt/decrypt function, (i.e. it is possible to see a callback before the encrypt/decrypt call completes)
9. The user callback shall consume all data provided to it before returning to the caller.
10. The system shall support shifting binary data, not just alpha-numeric strings

# Turn-in Procedures

Turn in all source code via canvas by 11:59p.m. on <DUE DATE>

Grading Rubric

(ECE 2420 PEX3)

|  |  |  |
| --- | --- | --- |
| Requirement / Criteria | Available Points | Student’s Score |
| Compiles and links correctly; uses provided interface verbatim.Links against the libgmp shared library.  | 10 |  |
| Static factory function creates correct object | 10 |  |
| Keys are generated from 1024 bit primes | 10 |  |
| Key management function operate correctly | 10 |  |
| Totient is calculated correctly | 10 |  |
| Modular multiplicative inverse is calculated correctly | 10 |  |
| Blocking/padding implemented correctly | 20 |  |
| Encrypted/decrypted complex file checked with MD5 sum | 20 |  |
| Total | **100** |  |