C++ Implementation of Cryptographic Algorithms

Ghaith Arar, Amanda Ly, *Student Member, IEEE,* and Neel Haria
Stevens Institute of Technology
Hoboken, NJ 07030, USA
Email: garar@stevens.edu, aly@stevens.edu and nharia@stevens.edu

***Abstract*—Cybersecurity has gotten increasingly significant in the past few years and with the rise in the widespread use of technology, there has also been a rise in cybercrime and cyber-attacks. These cybersecurity threats consist of but are not limited to malware, phishing, data leakage, hacking, structured query language injection, denial-of-service attacks, and domain name system tunneling. Cybersecurity is important because it pertains to protecting user's sensitive data and personal information; however, securing information is a challenge. Cryptography is an integral part of modern world information security to make the virtual world a safer place. Cryptography is a process of making information unintelligible to an unauthorized person. Hence, providing confidentiality to genuine users. This paper will look into various cryptographic algorithms and implement the Rivest–Shamir–Adleman (RSA) algorithm, and Diffie-Hellman key exchange using C++.**

***Index Terms*—Rivest–Shamir–Adleman (RSA), Diffie-Hellman, Data Encryption, Cryptography, Data Decryption, GMP.**

I. Introduction

INFORMATION security plays an important role in protecting digital information against security threats and keeps information secret by protecting it from unauthorized access. Cryptography involves secret writing between two or more people so that others looking in cannot decipher what is being said or shared. There are two types of cryptosystems based on the number of keys involved: a symmetric key cryptosystem and an asymmetric key cryptosystem. Cybersecurity experts define cryptography in five components and the underlying technology of cryptography involves the encryption of plain text and the decryption of cipher text. Encryption is the process of conversion of data, called plain text, into an unreadable form, called cipher text, that cannot be easily understood by unauthorized people. Decryption is the process of converting encrypted data back into its original form, so that it can be understood by the people who are authorized to read the data [1]. The idea is that a recipient of plain text would not need special knowledge or equipment to understand what is being communicated, and the decryption algorithm decodes the cipher text to the original understandable plain text. This is all done with a set of keys - secret information known by members of the group that parameterize the encryption and decryption.

Cryptography, the use of codes and ciphers to protect secrets, began thousands of years ago and can be dated back to 1900 BCE during the time of Old Kingdom Egypt. During World War I and World War II, cryptography played a vital in the victory of Allied forces. Until the 1960s, secure cryptography was largely the preserve of governments. It was not until the creation of public encryption standard, and the invention of public-key cryptography did cryptography move into the public domain as well. With modern technology, cyphers using keys with these lengths are becoming easier to decipher so the basic cryptographic algorithms are no longer mainly used; instead a hybrid form of the basic algorithms are now being implemented to make information security more secure.

The remainder of this paper is organized as follows. Section II discusses the different cryptographic algorithms. Section III presents the team's novel implementation of RSA and Diffie-Hellman key exchange using C++. Section IV and IV-E will show the our design to both algorithms. Section V will discuss the observations of our implementation and the paper will be concluded in section VI.

II. Cryptographic Algorithms

Cryptographic algorithms are the most frequently used privacy protection method and is used for important tasks such as data encrpytion, authentication, and digital signatures. Cryptographic algorithms differ to two categories based on the key. A Key is a numeric or alpha numeric text or may be a special symbol. The Key is used at the time of encryption on the plain text and at the time of decryption on the cipher text. The selection of key in cryptography is very important since it is directly associated to the security of the encryption algorithm. The strength of the encryption algorithm relies on the secrecy of the key, length of the key, the initialization vector, and how they all work cohesively. There are two types of algorithms: symmetric algorithms and asymmetric algorithms.

A. Symmetric Algorithms

In Symmetric key, encryption consists of only one key for encrypting and decrypting data between the sender and the receiver, called secret key. Symmetric encryption algorithms are faster than asymmetric algorithms because they require less processing power for computations. Types of symmetric cryptosystems include but are not limited to Advanced Encryption Standard and Data Encryption Standard.

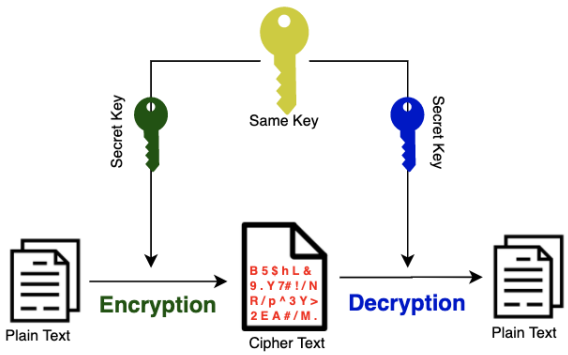

Fig. 1: Diagram of Symmetric Encryption

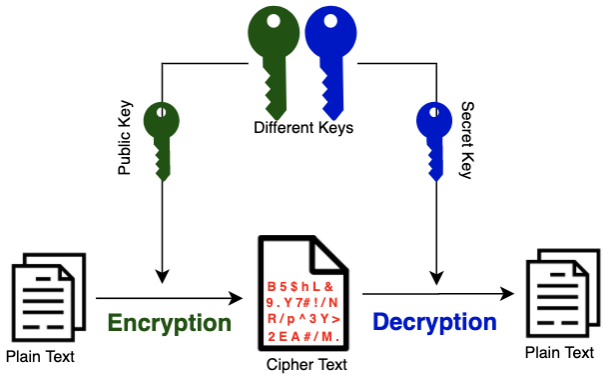

Fig. 2: Diagram of Asymmetric Encryption

*1) **Advanced Encryption Standard**:* Advanced Encryption Standard (AES) was developed by two Belgian cryptographers Vincent Rijmen and Joann Daeman and recommended by the National Institute of Standards and Technology (NIST) to replace DES in 2001. The AES algorithm can support any combination of data (128 bits) and key length of 128, 192, and 256 bits. The algorithm is referred to as AES-128, AES-192, or AES-256, depending on the key length. It has a relatively fast encryption and decryption, low power consumption and is excellently secured. It is commonly used in wireless security, processor security, file encryption, and SSL/TLS. The 256-bit key provides the strongest level of encryption because the hacker would need to try 2^{256} different combinations; thus making AES-256 the most secure AES implementation and virtually impenetrable using brute-force methods. AES was designed as a substitution-permutation network and brought additional security due to it using a key expansion process in which the initial key is used to come up with a series of new keys called round keys. These round keys are generated over multiple rounds of modification, each of which makes it harder to break the encryption [2]. AES is a sophisticated encryption algorithm and due to its open nature, can be used for public, private, commercial and noncommercial implementations.

*2) **Data Encryption Standard**:* Data Encryption Standard (DES) was the first encryption standards developed by NIST. It was developed by an IBM team around 1974 and adopted as a national standard in 1997 [3]. It takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. DES is a block cipher which is designed to encrypt and decrypt blocks of data consisting of 64 bits, but the effective key length is only 56 bits. The block size of DES is 64 bits and provides adequate security but it has a slow computational speed. DES is a also scalable algorithm due to varying the key size and block size. It also has an inherent vulnerability to brute forced, linenar and differential cryptoanalysis attack.

B. Asymmetric Algorithms

Asymmetric encryption, also known as public-key cryptography, is a relatively new method, compared to symmetric encryption, and uses two keys to encrypt plain text. Asymmetric encryption was introduced to complement the inherent problem of the need to share the key in the symmetric encryption model, eliminating the need to share the key by using a pair of public-private keys. A public key is made freely available to anyone who might want to send you a message. The second private key is kept a secret so that you can only know. This algorithm uses a key generation protocol to generate a key pair so both the keys are mathematically connected with each other. The benefits of asymmetric cryptography is the elimination of key distribution, an increased security due to the lack of key transmission and the use of digital signatures. The main disadvantage is the slower computation/processing speed compared to symmetric cryptography. Types of asymmetric cryptosystems [4] include but are not limited to RSA, Elliptic Curve Cryptosystem, Diffie-Hellman, and Digital Signature Algorithm.

*1) **RSA**:* There were several schemes and algorithms proposed for public key cryptography since its founding, and one of the earlier known algorithm is RSA [5]. RSA, also known as the Rivest–Shamir–Adleman algorithm, was founded in 1977 and is the most common public key algorithm in cryptography world. RSA is named for its inventors, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman, who created it as faculty at the Massachusetts Institute of Technology. Although the concept of RSA is still widely used in electronic commerce protocol, RSA itself has many flaws in its design therefore not preferred for the commercial use [6]. The key size of RSA is greater than or equal to 1024 bits with a minimum block size of 512 bits. This asymmetric algorithm has a high power consumption, a slow encryption, a slow decryption and is the least secure when compares to the symmetric algorithms AES and DES mentioned above. It also has an inherent vulnerability to brute forced and oracle attacks.

Even though, applying the algorithm is relatively simple, it lies behind powerful math theorems to ensure its strength. The security of RSA relies on the difficulty of factoring the product of two large prime numbers. This algorithm is based on the theory of Prime Numbers and the fact that it is easy to find and multiply large prime numbers, but it is extremely

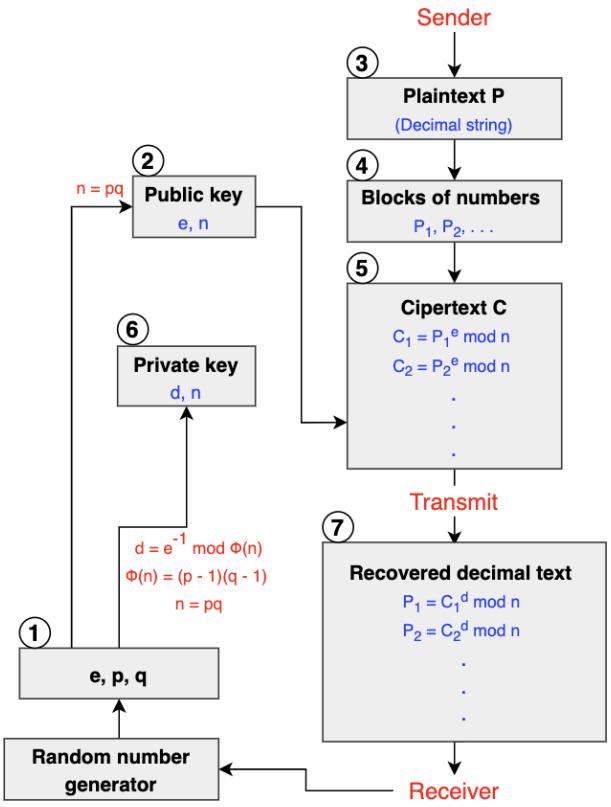

Fig. 3: Process of RSA [7]

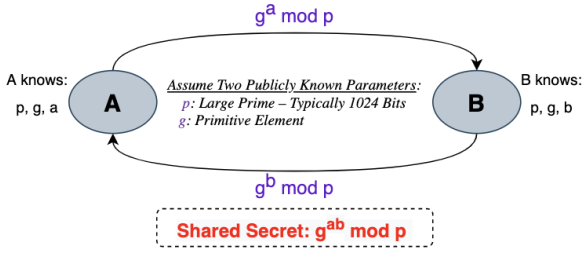

Fig. 4: Process of Diffie-Hellman Key Exchange

difficult to factor their product. It's security depends on the difficulty of decomposition of large numbers. This paper will later show a basic implementation (Section III) of RSA and design in Section IV using C++ and BigInteger Libraries GMP and Boost.

*2) **Diffie-Hellman**:* The Diffie-Hellman (DH) algorithm is one of the most important developments in public-key cryptography. It was founded between 1969-1973 but not published until 1976 by Whitfield Diffie and Martin Hellman researchers from Standford University and Ralph Merkle researcher from the University of California at Berkeley. The DH key exchange, a procedure which is one of the first public key cryptographic protocols, does not encrypt data, instead, it generates a secret key common to both the sender and the recipient. Although they never agreed on using a particular key, through mathematically linked processes the two parties can independently generate the same secret key and then use it to build a session key for use in asymmetric algorithm [8]. This method allows two parties that have no prior knowledge of each other to generate a shared private key with which they can exchange information across an insecure channel.

Similarly to RSA, Diffie-Hellman also uses prime numbers. The simplest and the original implementation [9] of the algorithm uses the multiplicative group of integers modulo p, where p is prime, and g is a primitive root modulo p. Next, both parties will pick a secret integer that they do not reveal to anyone else.

DH is usually utilized when you encrypt data on the Web utilizing either SSL (Secure Socket Layer) or TLS (Transport Layer Security) [10]. Some common DH protocols attacks are denial-of-service attacks, outsider attacks, insider attacks and man-in-the-middle attacks. In a denial-of-service attack, the attack could delete the messages or overwhelm the parties with unnecessary computations or communication. In an outsider attack, the attacker tries to disrupt the DH protocol by adding, removing replaying messages or other similar methods so the attacker is able to obtain interesting information. An insider attack is one where one of the parties creates a breakable protocol run on purpose to gain knowledge about the secret key of the other party. In a man-in-the-middle attack, the attacker would intercept one parties public value and send their own value to the other party. The attacker would then substitute the public value of the second party with their own and send it back to the first party. This attacker would then be able to decrypt and modify the messages send by either party [11]. This vulnerability is present because Diffie-Hellman key exchange does not authenticate the participants. This paper will later show a basic implementation (Section III) of the Diffie Hellman key exchange and our design in Section IV-E using C++ and BigInteger Library GMP.

*3) **Elliptic Curve Cryptosystem**:* The Elliptic Curve Cryptosystem (ECC) was invented independently by Neal Koblitz [12] and Victor S. Miller [13] in 1985, but the algorithm was not widely used until around 2004 because many cryptographers thought that the elliptic curve discrete logarithm problem had not been adequately examined to be used in security. ECC is based on the mathematical properties of elliptic curves over finite fields. Like DH and RSA, ECC is based on mathematical functions that are simple to compute in one direction, but very difficult to reverse. The different curves provide different level of security (cryptographic strength), different performance (speed) and different key length. Because the elliptic curve cryptosystem helps to establish equivalent security with lower computing power and battery resource usage, it is becoming widely used for mobile applications [14].

The equation for an elliptic curve in the Weierstrass form is $y^2 = x^3 + ax + b$ and inn the Koblitz curves form is $y^2 + xy = x^3 + ax^2 + b$. As previously stated, ECC is based on the discrete logarithm problem and thus it requires arithmetic modulo a

prime or in a Galois field (GF) - also known as final field. The ECC scheme [15] consists of a structure with four layers:

1) First layer (bottom layer): modular arithmetic consisting of addition, subtraction, multiplication, and inversion
2) Second layer: point doubling and point addition
3) Third layer: scalar multiplication
4) Fourth layer (top layer): protocols such as the Elliptic Curve Diffie–Hellman Key Exchange (ECDH) or the Elliptic Curve Digital Signature Algorithm (ECDSA)

ECC can be used for key exchange, for digital signatures and for encryption and provides the same level of security as RSA or discrete logarithm systems over $\mathbb{Z}^*{}_p$ with shorter operands. Where ECC may require 160–256 bit, RSA would require 1024–3072 bit, which results in shorter cipher texts and signatures. In many cases ECC has performance advantages over other public-key algorithms; however, signature verification with short RSA keys is still considerably faster than ECC.

*4) **Digital Signature Algorithm**:* The Digital Signature Algorithm (DSA) is used to create digital signatures for data transmission and was introduced in 1991 by the National Institute of Standards and Technology (NIST). DSA makes use of a unique mathematical functions that create a digital signature with two 160-bit numbers. These numbers are originated from the message digests and the private key. The digital signature scheme consists of three parts [16]:

a) Public and Private Key Generation:
   i) Choose a prime number *q*, and prime number *p*, such that $(p-1)\ mod\ q = 0$.
   ii) Choose an integer g, such that $1 < g < p$, $g^q\ mod\ p = 1$ and $g = h^{((p-1)/q)}\ mod\ p$
   iii) Choose an integer x such that $0 < x < q$
   iv) Compute $y = g^x\ mod\ p$
   v) Public key is {p,q,g,y}. Private key is {p,q,g,x}.

b) Signature Generation:
   i) Generate the message digest *h*, using a hash algorithm
   ii) Generate a random number k, such that $0 < k < q$
   iii) Compute *r* as $(g^k\ mod\ p)\ mod\ q$. If $r = 0$, select a new value for k
   iv) Compute i such that $k * i\ mod\ q = 1$
   v) Compute $s = i * (h + r * x)\ mod\ q$ and if s = 0, select a new value for k
   vi) The digital signature is {r,s}

c) Signature Verification:
   i) Generate the message *h*, using the same hash algorithm as before.
   ii) Compute w such that $s * w\ mod\ q = 1$
   iii) Compute $u_1 = h * w\ mod\ q$ and $u_2 = r * w\ mod\ q$
   iv) Compute $v = (((g^{u_1}) * (y^{u_2}))\ mod\ p)\ mod\ q$. If $v == r$, then the signature is valid.

The advantages of DSA is that it is patent free so it can be used free of cost and requires less storage to work as compared to other digital standards with strong strength levels and small signature lengths. The disadvantages are that it requires a lot of time for computation and authenticate as the verification process includes complicated remainder operators and the data in DSA is not encrypted, only authenticated.

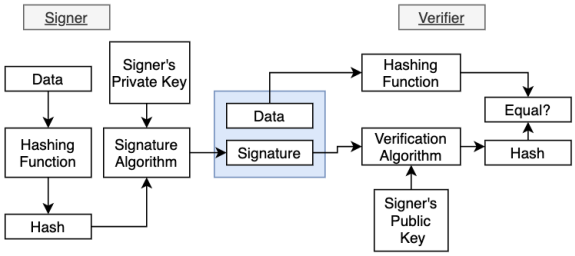

Fig. 5: Block Diagram of Digital Signature [16]

III. Implementation

A. Steps of RSA

1) Generating the Keys:
   i) Select two large prime numbers, p and q. The prime numbers need to be large so that they will be difficult for an attacker to guess.
   ii) Calculate $n = p * q$.
   iii) Calculate the totient function $\phi(n) = (p-1)(q-1)$.
   iv) Select an integer e, such that e is co-prime to $\phi(n)$ and $1 < e < \phi(n)$. The pair of numbers(n,e) makes up the public key.
   v) Calculate d such that $e * d = 1\ mod\ \phi(n)$ or $d = e^{-1}\ mod\ \phi(n)$.

2) Encryption: Given a Plain-text T, represented as a number the Cipher text C is calculated as $C = T^e mod\ n$

3) Decryption: Using the private key(n,d) the plain-text can be found using $T = C^d mod\ n$

B. Steps of Diffie-Hellman

Let Alice be the first party and Bob be the second party.

1) Public Parameter Generation: A trusted party chooses and publishes a large prime number p and a nonzero integer g modulo p

2) Private Computations:
   i) Alice chooses a secret integer *a* and then computes $A \equiv g^a\ mod\ p$.
   ii) Bob chooses a secret integer *b* and then computes $B \equiv g^b\ mod\ p$.

3) Public Exchange of Values: Alice will send A to Bob and in exchange, Bob will send B to Alice.

4) Final Private Computations:
   i) Since Alice now knows B, it will use its own secret integer a and compute $B^a\ mod\ p$
   ii) Likewise, Bob now knows A and with its own secret integer *b* will compute $A^b\ mod\ p$

The shared secret value is $B^a mod p \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b\ mod\ p$.

C. GMP-Library

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. According to the library documentation, there is no practical limit to the precision except the ones implied by the available memory in the machine running the GMP [17].

The main motivations behind GMP are cryptography applications and research. The library uses full words as the basic arithmetic type to achieve maximum speed. It uses highly optimized assembly code for the most common inner loops.

Although the library is known to work on Windows in both 32-bit and 64-bit mode, it's main target platform are Unix-type systems, such as GNU/Linux, Solaris, HP-UX, and Mac OS X/Darwin.

This project uses mainly the high-level signed integer arithmetic functions (mpz_t). There are about 150 arithmetic and logic functions in this category.

All declarations needed to use GMP are collected in the include file gmp.h. It is designed to work with both C and C++ compilers. Also, all programs using GMP must link against the libgmp library. On a typical Unix-like system this can be done with '*-lgmp*'. GMP C++ functions are in a separate libgmpxx library. This is built and installed if C++ support has been enabled.

In this project, among the multiple types GMP provides, we only used the integer type and GMP integer arithmetic functions. The C data type that GMP provides to represent multiple precision integers is *mpz_t*.

All GMP integer objects must be initialized before passing them to functions, and cleared when they are no longer needed. The GMP library provide multiple functions for Initialization and clearing. We mainly used **void mpz_init (mpz_t x)** which Initializes x, and set its value to 0. However, **void mpz_inits (mpz_t x,...)** can be used in the same fashion to initialize multiple objects at once. This function takes a NULL-terminated list of mpz_t variables

In a similar way, **void mpz_clear (mpz_t x)** and **void mpz_clears (mpz_t x,...)** can be used to free memory sapes occupied by GMP mpz_t objects.

The library offers multiple functions for assignment, however all such functions assume the GMP mpz_t objects have been initialized before calling the function. We list below the functions used in this project. For the full list we suggest visiting the GMP library website.

- **void mpz_set (mpz_t rop, const mpz_t op)**
- **void mpz_set_ui (mpz_t rop, unsigned long int op)**
- **void mpz_set_si (mpz_t rop, signed long int op)**

Initialization and assignment can be combined. GMP library provides functions a parallel series of initialize-and-set functions which initialize the output and then store the value there. It's noteworthy to mentioned that programmers should not use an initialize-and-set function on a variable already initialized.

- **void mpz_init_set (mpz_t rop, const mpz_t op)**
- **void mpz_init_set_ui (mpz_t rop, unsigned long int op)**
- **void mpz_init_set_si (mpz_t rop, signed long int op)**

A crucial step of RSA algorithm and many other Cryptography algorithms is generating seemingly random and very large prime number. In general, for RSA, the greater the numbers used to generate the keys, the harder it takes to crack the algorithm. GMP library offers a good set of functions for arithmetic operations, division, exponentiation, random numbers, and primality test. The library offer two groups of function, older function that rely on a global state, and newer functions that accept a state parameter that is read and modified. This project uses the newer group only.

To generate very large large prime using the GMP library, we followed the following step.

1) We used **void mpz_urandomb (mpz_t rop, gmp_randstate_t state, mp_bitcnt_t n)** to generate p and q. The generated numbers are uniformly distributed random integer in the range 0 to $2^n - 1$ inclusive. However, it's important to understand that the length of p and q will determine the key length. The function requires a **gmp_randstate** object which holds an algorithm selection and current state data. This object needs to be initialized before passing it to the function. GMP provides multiple functions to do, but in this project we believe that **void gmp_randinit_mt (gmp_randstate_t state)** will give us the best result. This function initializes state for a Mersenne Twister algorithm. This algorithm is fast and has good randomness properties.
2) After generating a random number, we check if the number is large enough. We regenerate another random number if the number isn't large enough.
3) We then check if the generated number is prime by using **int mpz_probab_prime_p (const mpz_t q, int reps)**. The function determines whether n is prime. Return 2 if n is definitely prime, return 1 if n is probably prime (without being certain), or return 0 if n is definitely non-prime. This function performs some trial divisions, a Baillie-PSW probable prime test, then reps-24 Miller-Rabin probabilistic primality tests. A higher reps value will reduce the chances of a non-prime being identified as "probably prime". Reasonable values of reps are between 15 and 50.

D. Boost-Library

Boost is a group of libraries for the C++ that provides support for functionality and data structures including pseudo-random number generation, and Multi-precision integers.

Boost libraries aim at a wide range of C++ users and application domains. They range from the general purpose libraries to the smart pointer library, to operating system abstractions like Boost File System.

In order to ensure efficiency and flexibility, boost makes extensive use of templates. Most boost libraries are header based, consisting of inline functions and templates.

In this RSA implementation we use boost's multi-precision library. We import the multi-precision library using boost-/multiprecision/cpp_int.hpp . The cpp_int_backend type is

normally used via one of the convenience type-defs. The backend can represent both fixed and arbitrary precision integer types, and both signed and unsigned types.

Default constructed cpp_int backends have the value zero. Division by zero results in a std::overflow_error being thrown. The cpp_int types support constexpr arithmetic, provided it is a fixed precision type with no allocator. It may also be a checked integer: in which case a compiler error will be generated on overflow or undefined behaviour. In addition the free functions abs, swap, multiply, add, subtract, divide_qr, integer_modulus, powm, lsb, msb, bit_test, bit_set, bit_unset, bit_flip, sqrt, gcd, lcm are all supported. Use of cpp_int in this way requires either a C++2a compiler (one which supports std::is_constant_evaluated()), or GCC-6 or later in C++14 mode. Compilers other than GCC and without std::is_constant_evaluated() will support a very limited set of operations: expect to hit roadblocks rather easily.

IV. RSA Designs

A. Design 1: RSA with GMP Library

The implementation below is often called a textbook implementation. It depicts the basic steps to implement the RSA algorithm. In real life however, such an implementation is never used as it is because it has many vulnerabilities and is subject to attacks. We will discuss later, how this algorithm is applied in reality; However, a deep understanding of this basic implementation is the cornerstone to implement real-life RSA encryption. [15]

```
// Amanda Ly & Ghaith Arar & Neel Haria
// RSA using GMP
// Programs should be linked with
//the libgmpxx and libgmp libraries

#include <iostream>
#include "gmpxx.h"
#include <string>
#include <time.h>
#include <string>
#include <ctime>
#include <math.h> // to use fmod
#include <random> // for stackoverflow rand
#include <tuple> //for generate_keys function

mpz_t e, d;
vector <int> arr;
```

```
int main() {
mpz_t p, q, n;
mpz_inits(n, p, q, NULL);
gmp_randstate_t P_rn, Q_rn;
gmp_randinit_mt(P_rn);
gmp_randinit_mt(Q_rn);
gmp_randseed_ui (P_rn, time(NULL));
gmp_randseed_ui (Q_rn, time(NULL)*3);
mp_bitcnt_t P_N = 512;
mp_bitcnt_t Q_N = 512;

mpz_urandomb (p, P_rn, P_N);
mpz_urandomb (q, Q_rn, Q_N);

while (1){
if (mpz_probab_prime_p(p, 20) > 0){
    cout << " p is: " << p << endl;
    cout << mpz_get_str(NULL, 2, p) << endl;
    break; }
else
        {gmp_randseed_ui (P_rn, time(NULL));
        mpz_urandomb (p, P_rn, P_N);}
    }

while (1){
    if (mpz_probab_prime_p(q, 20) > 0){
    break; }
    else
        {gmp_randseed_ui (Q_rn, time(NULL)*3);
        mpz_urandomb (q, Q_rn, Q_N);}
    }
mpz_mul(n, p, q);
cout << " n is: " << p << endl;
cout << mpz_get_str(NULL, 2, n) << endl;

cout << p << " " << q << " " << n;

cout << "\nGenerate public/private key" << endl;
generate_keys(p , q);
cout << "Public Key: " << e << endl;
cout << "Private Key: " << d << endl;

string message = "THIS IS A TEST";
cout<<"\nOriginal Message: "<< endl;
cout<< message << endl;
cout << "... Encrypting message ..." << endl;
cout << "Encrypted Message: ";
encryptMsg(e, message, n);

cout << "\n. Decrypting message ." << endl;
cout << "Decrypted Message: " << endl;
decryptMsg(d, n);
cout << "Private Key: " << d << endl;
return 0;
}
```

```
void generate_keys(mpz_t p, mpz_t q) {
if (mpz_probab_prime_p(p, 50) == 2
    && mpz_probab_prime_p(q, 50) == 2){
    if (p == q){
    cout << "Cannot have identical prime numbers";
    cout <<endl;
        return;}

mpz_t phi, check;
mpz_inits(phi, e, check, d, NULL);
mpz_sub_ui(p, p, 1);
mpz_sub_ui(q, q, 1);
mpz_mul(phi, p, q);

gmp_randstate_t rd;
gmp_randinit_mt(rd);
unsigned long int seed = 20201204;
gmp_randseed_ui (rd, seed);

mpz_urandomm(e, rd, phi);

while (mpz_cmp(e, phi) < 0){
    mpz_gcd(check, e, phi);
    if (mpz_cmp_ui(check, 1) == 0){
    break;}
    else {
        seed = seed + 3;
        gmp_randseed_ui (rd, seed);
        mpz_urandomm(e, rd, phi);}
    }

mpz_invert(d, e, phi);
mpz_clears(check, phi, NULL);}
}
```

```
void encryptMsg(mpz_t pbk, string msg, mpz_t n){
mpz_t ciphertxt, x;
unsigned long int tst;
mpz_init(ciphertxt);

for (int i = 0; i < msg.length(); i++)
{
    mpz_init_set_ui(x, int(msg.at(i)));

    mpz_powm(ciphertxt, x, e, n);
    ciphertxt << endl;
    tst = mpz_get_ui(ciphertxt);

    tst << endl;
    arr.push_back(int(tst));
    cout << tst;
    }
cout << endl;
mpz_clears(x, ciphertxt, NULL);
cout << endl;
}
```

B. RSA in reality

Practical implementation has to take a few things into account. We will discuss below some of the main requirement for real-life implementation.

*1) **Padding**:* RSA has to be used with a padding scheme. Padding schemes are extremely important, and if not implemented properly, an RSA implementation may be insecure. RSA encryption is deterministic, i.e., for a specific key, a particular plaintext is always mapped to a particular ciphertext. An attacker can derive statistical properties of the plaintext from the ciphertext. Furthermore, given some pairs of plaintext–ciphertext, partial information can be derived from new ciphertexts which are encrypted with the same key.

A possible solution to all these problems is the use of padding, which embeds a random structure into the plaintext before encryption and avoids the above mentioned problem. Padding is a large topic and outside the scope of this prject, however, programmers need to understand the concept.

*2) **Attacks**:* There have been numerous attacks proposed against RSA since it was invented in 1977. None of the attacks are serious, and moreover, they typically exploit weaknesses in the way RSA is implemented or used rather than the RSA algorithm itself. There are three general attack families against RSA:

a) Protocol attacks: Protocol attacks exploit weaknesses in the way RSA is being used. There have been several protocol attacks over the years. Among the better known ones are the attacks that exploit the malleability of RSA, which was introduced in the previous section. Many of them can be avoided by using padding. Modern security standards describe exactly how RSA should be used, and if one follows those guidelines, protocol attacks should not be possible.

b) Mathematical attacks: The best mathematical cryptanalytical method we know is factoring the modulus. An attacker, Oscar, knows the modulus n, the public key e and the ciphertext y. His goal is to compute the private key. In order to prevent this attack, the modulus must be sufficiently large. This is the sole reason why moduli of 1024 or more bit are needed for a RSA.

c) Side-channel attacks: A third and entirely different family of attacks are side-channel attacks. They exploit information about the private key which is leaked through physical channels such as the power consumption or the timing behavior. In order to observe such channels, an attacker must typically have direct access to the RSA implementation, e.g., in a cell phone or a smart card.

C. Design 2: RSA with Boost

The implementation below is a basic implementation of the RSA Encryption and Decryption. In this implementation we have used the Boost Library for extensive computation during encryption and decryption.

```
//Written by: Neel Haria, Amanda Ly and Ghaith Arar

#include<boost/multiprecision/cpp_int.hpp>
#include<iostream>
#include<cmath>
#include<vector>
#include<numeric>
using namespace std;


class KeyGeneration {
public:
        unsigned p; //First prime Number
        unsigned q; //Second Prime Number
        unsigned n; //Product of p and q
        unsigned phi_n; //Product of p-1 and q-1
        unsigned e; //e is co-prime to phi_n,1<e<phi_n
        unsigned d; //Such that e.d = 1 mod phi_n
        unsigned m; //Message to be encrypted

        //cipher text message
        boost::multiprecision::cpp_int c;

        //Decryption Result
        boost::multiprecision::cpp_int m_0;

        void prompter();
        void primality_check();
        unsigned calculate_n();
        unsigned calculate_phi();
        unsigned calculate_e();
        unsigned calculate_d();
        unsigned GCD(unsigned i, unsigned phi_n);

        /*Encryption and Decryption functions both
        use Boost because they involve large
        */computations
        boost::multiprecision::cpp_int encryption();
        boost::multiprecision::cpp_int decryption();

};

void KeyGeneration::prompter() {
cout << "Please Enter first prime number p: ";
cin >> p;
cout << "Please Enter first prime number q: ";
cin >> q;
}

void KeyGeneration::primality_check() {
//Check if number is prime or not
vector<unsigned> p_factors;
```

```
for (unsigned i = 2;
i <= static_cast<unsigned>(sqrt(p)); i++) {
        if (p % i == 0) {
                p_factors.push_back(i);
                p_factors.push_back(p / i);
        }
}

vector<unsigned> q_factors;
for (unsigned i = 2;
i <= static_cast<unsigned>(sqrt(q));i++) {
        if (p % i == 0) {
                q_factors.push_back(i);
                q_factors.push_back(q / i);
        }
}

if(p_factors.size() != 0 || q_factors.size() != 0){
        throw(runtime_error("Non Prime Input"));
        }
}

unsigned KeyGeneration::calculate_n(){
        n = p * q;
        return n;
}

unsigned KeyGeneration::GCD(unsigned i,unsigned phi_n)
{
//Calculate GCD to calculate e
for (unsigned j = min(i, phi_n); j > 2; j--) {
if (i % j == 0 && phi_n % j == 0) {
        return j;
        }
}
}

unsigned KeyGeneration::calculate_e() {
//Calculate e, such that 1<e<phi_n
//e is co-prime to phi_n
        unsigned i = 2;

        while (GCD(i, phi_n) != 1) {
                i++;
        }
        e = i;
        if (e > phi_n) {
        throw(runtime_error("e is greater phi_n"));
        }
        return e;
}

unsigned KeyGeneration::calculate_phi() {
//Calculate phi_n = (p-1) * (q-1)
        phi_n = (p - 1) * (q - 1);
        return phi_n;
}

unsigned KeyGeneration::calculate_d() {
//Calculate d such that e.d = 1 mod phi_n
        /*
        e*d mod phi_n = 1
        */
unsigned k = 1;
while ((k * phi_n + 1) % e != 0) {
        k++;
}
d = ((k * phi_n) + 1) / e;
return d;
}

boost::multiprecision::cpp_int KeyGeneration::
encryption() { //C=P^e mod n.
cout << "Enter message m to encrypt: ";
cin >> m;
c = (boost::multiprecision::cpp_int(boost::
multiprecision::pow(boost::
    multiprecision::cpp_int(m), e))) % n;
return c;
}

boost::multiprecision::cpp_int KeyGeneration::
decryption() {
//P = C^d mod n
//c^d % n = m
m_0 = (boost::multiprecision::cpp_int(boost::
multiprecision::pow(boost::
    multiprecision::cpp_int(c), d))) % n;
return m_0;
}


int main(int argc, const char* argv[]) {
try {
KeyGeneration obj1;
obj1.prompter();
obj1.primality_check();
cout << "n: "<<obj1.calculate_n() << endl;
cout << "phi: "<<obj1.calculate_phi() << endl;

cout << obj1.calculate_e() << endl;
cout << obj1.calculate_d() << endl;

cout << obj1.encryption()<<endl;
cout << obj1.decryption() << endl;
//cout<<obj1.gcd(53,61);
}
catch (runtime_error& s) {
        cout << s.what() << endl;
}
return 0;

}
```

D. Standard C++ Library

This section contains the code for RSA Encryption and Decryption using the C++ Standard Library.

```
lass KeyGeneration {
public:
        double p, q, n, e, d;
        double phi_n,track,m,c,m_0,enck,enc;
        double d1,dec,deck;

        void prompter();
        void prime_check();
        double calculate_n();
        double calculate_e();
        double calculate_d();
        double calcualte_phi();
        double GCD(double i , double phi);
        double encryption();
        double decryption();
};

void KeyGeneration::prompter() {
        cout << "enter first prime number p: ";
        cin >> p;
        cout << "enter first prime number q: ";
        cin >> q;
}

void KeyGeneration::prime_check() {
vector<double> p_factors;
for (double i = 2; i <= static_cast<double>(sqrt(p));
i++) {
```

```
        if (fmod(p,i) == 0) {
                p_factors.push_back(i);
                p_factors.push_back(p / i);
                }
        }
vector<double> q_factors;
for (double i = 2; i <= static_cast<double>(sqrt(q)); i
{
        if (fmod(q,i) == 0) {
                q_factors.push_back(i);
                q_factors.push_back(p / i);
                }
        }
        if (p_factors.size() != 0 ||
        q_factors.size() != 0)
        {
        throw(runtime_error("Non␣Prime␣Input"));
        }
}

double KeyGeneration::calculate_n() {
        n = p * q;
        return n;
}
double KeyGeneration::GCD(double i, double phi) {
        double temp;
        for (;;) {
                temp = fmod(i, phi);
                if (temp == 0) {
                        return phi;
                }
                i = phi;
                phi = temp;
        }
}

double KeyGeneration::calculate_e() {
        e = rand() % 100;
        while (e < phi_n) {
                track = GCD(e, phi_n);
                if (track == 1) {
                        break;
                }
                else {
                        e++;
                }
        }
        return e;
}

double KeyGeneration::calcualte_phi() {
        phi_n = (p - 1) * (q - 1);
        return phi_n;
}

double KeyGeneration::calculate_d() {
         d1 = 1 / e;
         d = fmod(d1, phi_n);
//private key
        return d;


}

double KeyGeneration::encryption() {
        cout << "Enter␣Message:␣";
        cin >> m;
        enck = pow(m, e);
        d1 = 1 / e;
        d = fmod(d1, phi_n);
//private key
        deck = pow(c, d);
        enc = fmod(enck, n);
        return enck;
}

double KeyGeneration::decryption() {
        double d1 = 1 / e;
        double d = fmod(d1, phi_n);
//private key
        double deck = pow(enck, d);
//decryption key

        double dec = fmod(deck, n);
        return deck;
}

int main(int argc, const char* argv[]) {
try {
string message;
KeyGeneration obj1;
obj1.prompter();
obj1.prime_check();
cout << "n:"<<obj1.calculate_n() << endl;
cout << "phi:␣" << obj1.calcualte_phi() << endl;
cout << "e:␣" << obj1.calculate_e() << endl;
cout << "d:␣" << obj1.calculate_d() << endl;
cout << "Encryption:␣" << obj1.encryption() << endl;
cout << "Decryption:␣" << obj1.decryption() << endl;

        }
catch (runtime_error& s) {
cout << s.what() << endl;
}
return 0;

}
```

E. Diffie-Hellman Key Exchange Design

In this protocol we have two parties, Alice and Bob, who would like to establish a shared secret key. There is possibly a trusted third party that properly chooses the public parameters which are needed for the key exchange. However, it is //also possible that Alice or Bob generate the public parameters Strictly speaking, the DHKE consists of two protocols, the set-up protocol and the main protocol, which performs the actual key exchange.

```
#include <stdio.h>
#include <iostream>
#include <gmp.h>
#include <cmath>
#include <ctime>
using namespace std;

int main() {
cout << "test" << endl;
mpz_t p;
//1. Choose a large prime p.
//p should have a similar length as the
//RSA modulus n, i.e., 1024 or beyond,                  //double de
//in order to provide strong security

mpz_t alpha;
//2. Choose an integer        {2,3, . . . , p 2 }.
//The integer    needs to have a special property:
//It should be a primitive element, a topic
//which we discuss in the following sections. The

mpz_t op;
// Randomly generated number;
//used to get a prime number through mpz_nextprime

gmp_randstate_t state1, state2;
```

```
//The variable must be initialized by
//calling one of the gmp_randinit functions

gmp_randinit_mt (state1);
//Initialize state for a
//Mersenne Twister algorithm.

gmp_randinit_mt (state2);
//This algorithm is fast and
//has good randomness properties.


mp_bitcnt_t n1 = 1024;
//Counts of bits of a multi-precision
//number are represented in the C type mp_bitcnt_t.

mpz_t n2;
mpz_init(n2);

mpz_inits(p, alpha, op, NULL);
//The functions for integer arithmetic assume
//that all integer objects are initialized.
//You do that by calling the function mpz_inits.

gmp_randseed_ui(state1, time(NULL) );
gmp_randseed_ui(state2, time(NULL) );

mpz_urandomb (op, state1, n1);
//Generate a uniformly distributed random integer
//in the range 0 to 2^(n1-1), inclusive.

mpz_nextprime(p, op);
//Set p to the next prime greater than op.
//This function uses a probabilistic
//algorithm to identify primes.
//For practical purposes  i t s  adequate,
//the chance of a composite passing will
//be extremely small.

mpz_sub_ui(n2, p, 2);
//n2 holds the value of P-2 in
//the formula         {2,3, . . . ,  p 2 }.

mpz_urandomm(alpha, state2, n2);
// Generate Random number less than p-2

cout << "P is:" << endl;
gmp_printf("attempt p: %Zd \n", p);
cout << endl;

cout << "n2 is:" << endl;
gmp_printf("attempt n2: %Zd \n", n2);
cout << endl;

cout << "Alpha is:" << endl;
gmp_printf("attempt alpha: %Zd \n", alpha);
cout << endl;

cout << "op is:" << endl;
gmp_printf("attempt op: %Zd \n", op);
cout << endl;

cout << "n1 is:" << n1 << endl;

cout <<"Is P prime:";
cout << mpz_probab_prime_p(p, 60) << endl;
cout <<"Is alpa prime:";
cout << mpz_probab_prime_p(alpha, 60) << endl;

//These two values (p, alpha) are sometimes
//referred to as domain parameters. If Alice and
//Bob both know the public parameters p
//and    computed in the set-up phase, they
//can generate a joint secret key k as below
// Below:
// Alice choose a = kpr,A      {2, . . . ,  p 2 }
// Alice compute A = kpub,A    a    mod p
// a is Alice's secret key
// Alice computes A , and sent it to Bob
mpz_t a;
mpz_t A;
mpz_inits(a, A, NULL);

gmp_randstate_t state_a;
gmp_randinit_mt (state_a);
gmp_randseed_ui(state_a, time(NULL) );
mpz_urandomm(a, state_a, n2);
//choose a = kpr,A      {2, . . . ,  p 2 }

mpz_powm(A, alpha, a, p);
//compute A = kpub,A       ^a mod p

cout << "Alice secret key a:" << endl;
gmp_printf("%Zd \n", a);
cout << endl;

cout << "Alice public key A:" << endl;
gmp_printf("%Zd \n", A);
cout << endl;

// Bob choose b = kpr,B      {2, . . . ,  p 2 }
// Bob compute B = kpub,B    b    mod p
// b is Bob's secret key
// Bob computes B , and sent it to Alice
mpz_t b;
mpz_t B;
mpz_inits(b, B, NULL);

gmp_randstate_t state_b;
gmp_randinit_mt (state_b);
gmp_randseed_ui(state_b, (time(NULL) + 50) );
//Added 50 to the seeding to ensure
//seeding is different than Alice's

mpz_urandomm(b, state_b, n2);
//choose b = kpr,B      {2, . . . ,  p 2 }

mpz_powm(B, alpha, b, p);
//compute B = kpub,B    b    mod p


cout << "Bob secret key b:" << endl;
gmp_printf("%Zd \n", b);
cout << endl;

cout << "Bob public key B:" << endl;
gmp_printf("%Zd \n", B);
cout << endl;

return 0;
}
```

V. Discussion

We implemented the RSA algorithm using the GMP Library, Boost Library and the Standard Library. We faced a lot of challenges while implementing each of those, keeping in mind the limitations posed by each method. The most efficient implementation of the algorithm we observed was with the GMP Library, we came to this conclusion due to factors such as ease of installation and capability to handle large computations. In comparison to GMP, boost was very difficult to install on a Windows OS, though we did not try it on the

Linux system GMP becomes preferable in terms of it's ease of installation in both Windows and Linux.

VI. Conclusion

The simple implementation of RSA using C++, gave us clear and extensive knowledge about how cryptography algorithms work. Though, we implemented only the RSA algorithm but during our research we also studied other algorithms such as AES and DES. During our research we learned how symmetric and asymmetric algorithms differ from each other. Since, encryption plays and important role in data security, it is important to make encryption as robust as possible. To do so, there is heavy mathematics involved with extensive computation involving large numbers. The standard C++ library can not handle such calculations which causes users to look at specific libraries to do so, in our implementation we have used two different libraries (ie. GMP and Boost Multi-precision library) to execute the same algorithm.

Although cryptography algorithms seem simple in the textbook, designing a secure implementation that is both fast and secure is a challenge. There are many factors that must be taken into account.

References

[1] A. Kahate, *Cryptography and network security*. Tata McGraw-Hill Education, 2013.

[2] S. MSP, "Understanding AES 256 encryption," 2019. [Online]. Available: https://www.solarwindsmsp.com/blog/aes-256-encryption-algorithm

[3] D. E. Standard *et al.*, "Federal information processing standards publication 46," *National Bureau of Standards, US Department of Commerce*, vol. 23, 1977.

[4] K. Brush, L. Rosencrance, and M. Cobb, "What is asymmetric cryptography and how does it work?" 2020. [Online]. Available: https://searchsecurity.techtarget.com/definition/asymmetric-cryptography

[5] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[6] X. Zhou and X. Tang, "Research and implementation of rsa algorithm for encryption and decryption," in *Proceedings of 2011 6th international forum on strategic technology*, vol. 2. IEEE, 2011, pp. 1118–1121.

[7] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Pearson Education/Prentice Hall, 5th Edition., 2010.

[8] D. A. Carts, "A review of the diffie-hellman algorithm and its use in secure internet protocols," *SANS institute*, pp. 1–7, 2001.

[9] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[10] S. Kallam, "Diffie-hellman: Key exchange and public key cryptosystems," *Master degree of Science, Math and Computer Science, Department of India State University, USA*, pp. 5–6, 2015.

[11] J.-F. Raymond and A. Stiglic, "Security issues in the diffie-hellman key agreement protocol," *IEEE Transactions on Information Theory*, vol. 22, pp. 1–17, 2000.

[12] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[13] V. S. Miller, "Use of elliptic curves in cryptography," in *Conference on the theory and application of cryptographic techniques*. Springer, 1985, pp. 417–426.

[14] G. Raju and R. Akbani, "Elliptic curve cryptosystem and its applications," in *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*, vol. 2. IEEE, 2003, pp. 1540–1543.

[15] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[16] M. Sharma, "Digital signature algorithm (dsa) in cryptography," 2020. [Online]. Available: https://www.includehelp.com/cryptography/digital-signature-algorithm-dsa.aspx

[17] "Gnu mp library manual," 2020. [Online]. Available: https://gmplib.org/manual/