Cryptography Final Project

Implementing cryptographic and mathematical algorithms with efficient execution and rigorous testing.

by

Valeria Espinoza[*]

Supervised by Professor Geoffrey Pascoe

Submitted December 9, 2024

[*] Student ID: U64430526, Email: `valdlaw@bu.edu`

Table of Contents

List of Figures

Main Content

1. Manual: Using and Testing the Algorithms

This section provides a comprehensive guide to running the implemented algorithms and their associated unit tests. Each algorithm can be executed directly from the command line, and the unit tests validate the algorithms' correctness and reliability.

1.1. Running the Algorithms

Each algorithm is implemented to solve specific cryptographic or mathematical problems. They can be executed from the project root directory using the `-m` flag with Python. Below is a detailed explanation of how to run each algorithm, including the required parameters and expected outputs.

1.1.1. Blum Blum Shub (BBS)

The Blum Blum Shub algorithm generates a cryptographically secure pseudorandom number. Specify the number of bits for the random number. Optionally, you can provide two large prime numbers p and q such that $p \equiv 3 \mod 4$ and $q \equiv 3 \mod 4$.

Command:

```
python3 -m algorithms.blum_blum_shub <bits> [--p <prime_p>] [--q <prime_q>]
```

Example:

```
python3 -m algorithms.blum_blum_shub 128
python3 -m algorithms.blum_blum_shub 128 --p 11 --q 19
```

1.1.2. Naor-Reingold (NR)

The Naor-Reingold pseudorandom number generator produces a random number based on a secret key, a prime modulus p and its primitive root g. Specify the number of bits, p, g and a list of secret key bits.

Command:

```
python3 -m algorithms.naor_reingold <bits> <p> <g> <secret_key_bits...>
```

Example:

```
python3 -m algorithms.naor_reingold 4 29 2 5 6 3 1
```

1.1.3. Primitive Root Search

Find the smallest primitive root modulo a given prime p.

Command:

```
python3 -m algorithms.primitive_root_search <p>
```

Example:

```
python3 -m algorithms.primitive_root_search 29
```

1.1.4. Baby-Step Giant-Step

Solve the discrete logarithm problem $g^x \equiv h \mod p$. Provide g, h, and p.

Command:

```
python3 -m algorithms.baby_giant_step <g> <h> <p>
```

Example:

```
python3 -m algorithms.baby_giant_step 2 8 11
```

1.1.5. Pollard's Rho

Factor a composite number n using Pollard's Rho. Optionally, specify the maximum number of retries.

Command:

```
python3 -m algorithms.pollards_rho <n> [--max_retries <number>]
```

Example:

```
python3 -m algorithms.pollards_rho 8051
```

1.1.6. Pollard's $p - 1$

Factor a composite number n based on a smoothness bound B.

Command:

```
python3 -m algorithms.pollards_p1 <n> [--B <bound>]
```

Example:

```
python3 -m algorithms.pollards_p1 10403 --B 20
```

1.1.7. Extended Euclidean Algorithm

Compute the greatest common divisor (GCD) of two integers a and b, along with coefficients x and y satisfying $ax + by = \text{GCD}(a, b)$.

Command:

```
python3 -m algorithms.extended_euclidean <a> <b>
```

Example:

```
python3 -m algorithms.extended_euclidean 240 46
```

1.1.8. Miller-Rabin Primality Test

Check whether a number n is probably prime. Optionally, specify the number of iterations.

Command:

```
python3 -m algorithms.miller_rabin <n> [--k <iterations>]
```

Example:

```
python3 -m algorithms.miller_rabin 101 --k 10
```

1.1.9. Fast Exponentiation

Calculate $\text{base}^{\text{exp}} \mod \text{mod}$ efficiently.

Command:

```
python3 -m algorithms.fast_exponentiation <base> <exp> <mod>
```

Example:

```
python3 -m algorithms.fast_exponentiation 3 200 13
```

1.2. Running Unit Tests

Testing the algorithms is straightforward. Each test file corresponds to a specific algorithm. To test all algorithms, navigate to the project root directory and run:

```
python3 -m unittest discover -s tests
```

This discovers and runs all test files in the `tests` folder. To test a specific algorithm, provide the test file name:

```
python3 -m unittest tests.test_blum_blum_shub
```

Replace `test_blum_blum_shub` with the appropriate test file name. The output indicates whether the tests passed or failed, along with details for any failures.

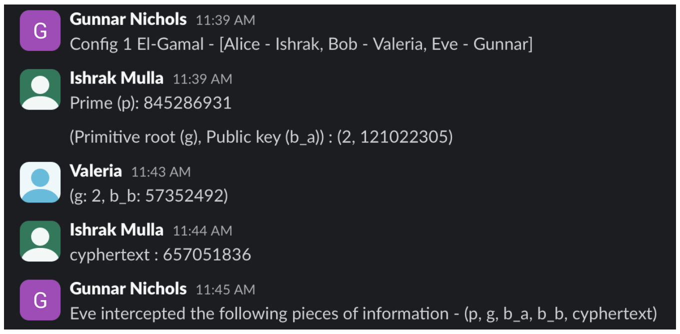
2. Exchanges

For our project, we utilized three configurations to represent cryptographic exchanges in both the El-Gamal and RSA protocols. Each configuration involved one member acting as **Alice (sender)**, another as **Bob (receiver)** and the third as **Eve (eavesdropper)**. This structure ensured that every member experienced all three roles. Below is the overview of the configurations and our approach:

- **Config 1**
  - Alice: Ishrak
  - Bob: Valeria (me)
  - Eve: Gunnar
- **Config 2**
  - Alice: Valeria (me)
  - Bob: Gunnar
  - Eve: Ishrak
- **Config 3**
  - Alice: Gunnar
  - Bob: Ishrak
  - Eve: Valeria (me)

For the randomness required in both protocols, I implemented the **Blum-Blum-Shub (BBS)** algorithm. BBS was chosen for its cryptographic strength and reliance on the hardness of factoring. All randomness in key generation and encryption was sourced from the BBS generator, which itself was seeded securely using Python's secrets library to mimic a **True Random Number Generator (TRNG)**.

2.1. Config 1: El-Gamal Protocol


Gunnar Nichols 11:39 AM
Config 1 El-Gamal - [Alice - Ishrak, Bob - Valeria, Eve - Gunnar]

Ishrak Mulla 11:39 AM
Prime (p): 845286931

(Primitive root (g), Public key (b_a)) : (2, 121022305)

Valeria 11:43 AM
(g: 2, b_b: 57352492)

Ishrak Mulla 11:44 AM
cyphertext : 657051836

Gunnar Nichols 11:45 AM
Eve intercepted the following pieces of information - (p, g, b_a, b_b, cyphertext)

Figure 1: Config 1 El-Gamal exchanges documentation

In the first configuration, I acted as **Bob**, while Ishrak was Alice, and Gunnar was Eve. The exchange began with Alice generating the key parameters. Using the **Blum-Blum-Shub (BBS)** algorithm, Ishrak generated a large prime modulus $p = 845286931$ and verified its primality using the **Miller-Rabin primality test**. A primitive root $g = 2$ was selected using the **Primitive Root Search algorithm**, which ensures that g generates the multiplicative group $\mathbb{Z}_p^*$.

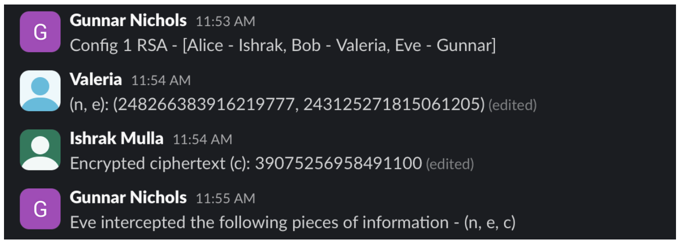
To generate her public key, Alice selected a private key r_a (randomly generated using BBS) and computed $b_a = g^{r_a} \mod p$ using the **Fast Exponentiation Algorithm**. She shared (p, g, b_a) with me.

As Bob, I received these parameters and generated my own private key r_b using BBS. With r_b, I calculated my public key $b_b = g^{r_b} \mod p$, again utilizing the **Fast Exponentiation Algorithm**. I shared b_b with Alice, completing the key exchange.

Alice then encrypted the message using my public key b_b. She calculated the shared secret $s = b_b^{r_a} \mod p$ with the **Fast Exponentiation Algorithm** and used it to encrypt her message m, producing the ciphertext $c = m \cdot s \mod p = 657051836$. This ciphertext was sent to me for decryption.

In this exchange, Gunnar, acting as Eve, intercepted p, g, b_a, b_b, c, as evidenced in Figure 1.

2.2. Config 1: RSA Protocol


Gunnar Nichols 11:53 AM
Config 1 RSA - [Alice - Ishrak, Bob - Valeria, Eve - Gunnar]

Valeria 11:54 AM
(n, e): (248266383916219777, 243125271815061205) (edited)

Ishrak Mulla 11:54 AM
Encrypted ciphertext (c): 39075256958491100 (edited)

Gunnar Nichols 11:55 AM
Eve intercepted the following pieces of information - (n, e, c)

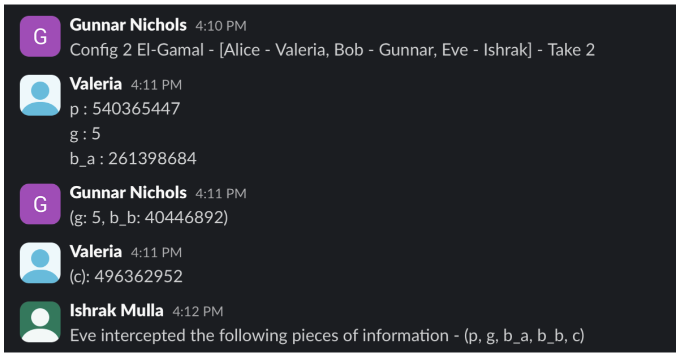
Figure 2: Config 1 RSA exchanges documentation

In the same configuration, Ishrak remained as Alice, I acted as Bob and Gunnar continued as Eve. As Bob, I began by generating RSA keys. Using the **Blum-Blum-Shub (BBS)** algorithm, I selected two large primes p and q, verified their primality with the **Miller-Rabin Test**, and computed the modulus $n = p \cdot q = 248266383916219777$. I also calculated $\phi(n) = (p-1)(q-1)$ and chose a public exponent $e = 243125271815061205$, ensuring that $\gcd(e, \phi(n)) = 1$.

To find the private key d, I used the **Extended Euclidean Algorithm** to compute the modular inverse of $e \mod \phi(n)$. I then shared (n, e) with Alice.

Alice encrypted the message m using my public key by calculating $c = m^e \mod n$ with the **Fast Exponentiation Algorithm**. She sent the ciphertext $c = 39075256958491100$ to me for decryption. Gunnar, acting as Eve, intercepted n, e, c, which is documented in Figure **??**.

2.3. Configuration 2: El-Gamal Protocol



Gunnar Nichols 4:10 PM
Config 2 El-Gamal - [Alice - Valeria, Bob - Gunnar, Eve - Ishrak] - Take 2

Valeria 4:11 PM
p : 540365447
g : 5
b_a : 261398684

Gunnar Nichols 4:11 PM
(g: 5, b_b: 40446892)

Valeria 4:11 PM
(c): 496362952

Ishrak Mulla 4:12 PM
Eve intercepted the following pieces of information - (p, g, b_a, b_b, c)

Figure 3: Config 2 El-Gamal exchanges documentation

In the second configuration, I acted as **Alice**, Gunnar was Bob and Ishrak was Eve. The exchange began with me generating the key parameters. Using the **Blum-Blum-Shub (BBS)** algorithm, I generated a large prime modulus $p = 540365447$, verified its primality with the **Miller-Rabin Test** and determined a primitive root $g = 5$ using the **Primitive Root Search Algorithm**.

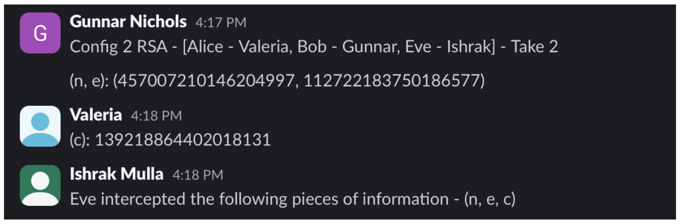
For my public key, I selected a private key r_a using the BBS generator and calculated $b_a = g^{r_a} \mod p = 261398684$ with the **Fast Exponentiation Algorithm**. I shared (p, g, b_a) with Gunnar.

As Bob, Gunnar used BBS to generate his private key r_b and computed his public key $b_b = g^{r_b} \mod p = 40446892$, which he shared with me. To encrypt the message, I calculated the shared secret $s = b_b^{r_a} \mod p$ using the Fast Exponentiation Algorithm. Using s, I encrypted the message m as $c = m \cdot s \mod p = 496362952$ and sent it to Gunnar for decryption.

In this exchange, Ishrak, acting as Eve, intercepted p, g, b_a, b_b, c. Figure 3 provides evidence of the intercepted parameters and ciphertext.

2.4. Configuration 2: RSA Protocol

For the RSA exchange in this configuration, I again took the role of **Alice**, Gunnar was Bob and Ishrak was Eve. Gunnar began the exchange by generating his RSA keys. Using the **Blum-Blum-Shub (BBS)** algorithm, he generated two large primes p and q, verified their primality using the Miller-Rabin Test, and computed the modulus $n = 457007210146204997$. He then calculated $\phi(n) = (p-1)(q-1)$ and selected a public exponent $e = 112722183750186577$, ensuring $\gcd(e, \phi(n)) = 1$.


Gunnar Nichols 4:17 PM
Config 2 RSA - [Alice - Valeria, Bob - Gunnar, Eve - Ishrak] - Take 2

(n, e): (457007210146204997, 112722183750186577)

Valeria 4:18 PM
(c): 139218864402018131

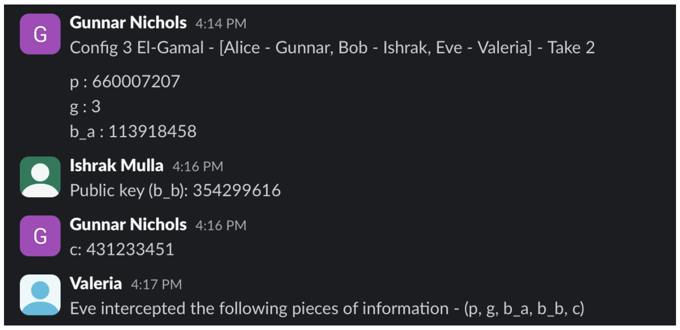
Ishrak Mulla 4:18 PM
Eve intercepted the following pieces of information - (n, e, c)

Figure 4: Config 2 RSA exchanges documentation

Using the **Extended Euclidean Algorithm**, Gunnar computed the private key d as the modular inverse of $e \mod \phi(n)$. He shared (n, e) with me as Alice.

To encrypt the message m, I calculated $c = m^e \mod n = 139218864402018131$ using the **Fast Exponentiation Algorithm** and sent the ciphertext c to Gunnar. Ishrak, as Eve, intercepted n, e, c, as shown in Figure 4.

2.5. Configuration 3: El-Gamal Protocol


Gunnar Nichols 4:14 PM
Config 3 El-Gamal - [Alice - Gunnar, Bob - Ishrak, Eve - Valeria] - Take 2

p : 660007207
g : 3
b_a : 113918458

Ishrak Mulla 4:16 PM
Public key (b_b): 354299616

Gunnar Nichols 4:16 PM
c: 431233451

Valeria 4:17 PM
Eve intercepted the following pieces of information - (p, g, b_a, b_b, c)

Figure 5: Config 3 El-Gamal exchanges documentation

In **Configuration 3** of the El-Gamal protocol, I acted as **Eve**, while Gunnar was Alice and Ishrak was Bob. The exchange began with Gunnar generating the key parameters as Alice. He used the **Blum-Blum-Shub (BBS)** algorithm to generate a prime modulus $p = 660007207$, which was verified for primality using the **Miller-Rabin Test**. Gunnar then used the **Primitive Root Search Algorithm** to find $g = 3$, a primitive root of $\mathbb{Z}_p^*$.

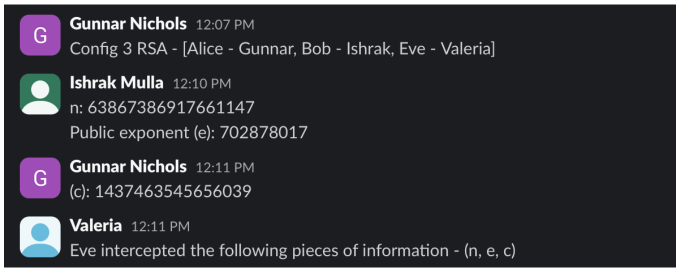
Gunnar selected a private key r_a using the BBS generator and computed his public key $b_a = g^{r_a} \mod p$ using the **Fast Exponentiation Algorithm**. He shared (p, g, b_a) with Bob (Ishrak).

As Bob, Ishrak generated his own private key r_b using BBS and calculated $b_b = g^{r_b} \mod p$, again using the **Fast Exponentiation Algorithm**. He shared b_b with Gunnar.

To encrypt the message, Gunnar computed the shared secret $s = b_b^{r_a} \mod p$ and encrypted the plaintext message m as $c = m \cdot s \mod p$. The ciphertext $c = 431233451$ was sent to Bob for decryption.

As Eve, my role was to intercept p, g, b_a, b_b, c and attempt to decrypt the ciphertext. To do so, I used the **Baby-Step Giant-Step Algorithm** to solve the discrete logarithm problem $b_a = g^{r_a} \mod p$ to recover Gunnar's private key r_a. Using r_a, I computed the shared secret $s = b_b^{r_a} \mod p$ with the **Fast Exponentiation Algorithm**. Finally, I calculated the modular inverse of s, s^{-1}, and decrypted the message as $m = c \cdot s^{-1} \mod p$. Evidence of the intercepted parameters is included in Figure 5.

2.6. Configuration 3: RSA Protocol



Gunnar Nichols 12:07 PM
Config 3 RSA - [Alice - Gunnar, Bob - Ishrak, Eve - Valeria]

Ishrak Mulla 12:10 PM
n: 63867386917661147
Public exponent (e): 702878017

Gunnar Nichols 12:11 PM
(c): 1437463545656039

Valeria 12:11 PM
Eve intercepted the following pieces of information - (n, e, c)

Figure 6: Config 3 RSA exchanges documentation

For the RSA exchange in **Configuration 3**, I again took the role of **Eve**, Gunnar acted as Alice and Ishrak as Bob. As Bob, Ishrak generated his RSA keys using the **Blum-Blum-Shub (BBS)** algorithm. He selected two large primes p and q, verified their primality using the **Miller-Rabin Test**, and computed the modulus $n = 63867386917661147$. He calculated $\phi(n) = (p-1)(q-1)$ and selected the public exponent $e = 702878017$, ensuring $\gcd(e, \phi(n)) = 1$. Ishrak then used the **Extended Euclidean Algorithm** to compute the private key d, the modular inverse of $e \mod \phi(n)$. He shared (n, e) with Gunnar.

As Alice, Gunnar encrypted the plaintext message m using Ishrak's public key, calculating $c = m^e \mod n$ with the **Fast Exponentiation Algorithm**. The resulting ciphertext $c = 1437463545656039$ was sent to Ishrak for decryption.

As Eve, I intercepted n, e, c and used **Pollard's Rho Method** to factor n into its prime components p and q. Pollard's Rho is an efficient probabilistic algorithm for integer factorization, which allowed me to find p and q quickly due to the relatively small size of n. With p and q determined, I calculated $\phi(n) = (p-1)(q-1)$ and derived the private key d using the **Extended Euclidean Algorithm**.

Finally, I decrypted the message by calculating $m = c^d \mod n$ using the **Fast Exponentiation Algorithm**. Evidence of the intercepted values and the decryption process is included in Figure 6.