MULTIMEDIA UNIVERSITY

ASSIGNMENT

TRIMESTER 1, 2023/2024

TCS3351

CRYPTOGRAPHY AND DATA SECURITY

Topic: **RSA CRYPTOSYSTEM**

NO.	NAME	STUDENT ID
1	NUR AYU AMIRA BINTI IDRIS	1201200722
2	DANIEL IMTIYAZ BIN FAISAL	1201201743
3	MUHAMMAD DHIYAUL NAUFAL BIN ZAINUDDIN	1201201537
4	MANNOJ SAKTHIVEL	1221303085

Evaluation: (40%)

Written Report	20%	
Algorithm Simulation	10%	
Oral Presentation	10%	
TOTAL	**40%**	

Table of Contents

1 ABSTRACT

This project presents a thorough analysis of the RSA Cryptosystem, emphasizing the use of prime numbers in its implementation, encryption, and decryption processes. It explores the mathematical underpinnings of RSA, such as modular arithmetic, prime numbers, and the role that public and private keys play in enabling secure communication. Through toy simulations and comparisons with symmetric encryption techniques like S-AES, the paper further emphasizes the practical applications of the RSA algorithm, showcasing RSA's resilience in securing digital communications against cryptanalysis. The team's research revealed important new information about the relationship between cryptography theory and practice, highlighting the critical role that RSA plays in modern secure communications.

2 INTRODUCTIONS

There are numerous approaches to encrypt messages to conceal information from view. Although cryptography has been around for thousands of years, its development and research have advanced significantly in recent years. To encrypt and decrypt messages, a variety of techniques are available; the most popular is the RSA algorithm.

In the realm of cryptography, where the safeguarding of digital information is paramount, the RSA algorithm stands as a beacon of security. Developed in the late 1970s by Ron Rivest, Adi Shamir, and Leonard Adleman, RSA represents a fundamental shift in cryptographic methodology with its innovative use of asymmetric encryption.

At its core, RSA relies on the mathematical challenge of factoring large prime numbers, a task believed to be computationally infeasible for sufficiently large numbers. This forms the basis of its security: while it is relatively easy to compute the product of two large prime numbers, finding those prime factors from the product is significantly more difficult, particularly as the size of the numbers increases.

The RSA algorithm operates with two distinct keys: a public key and a private key. The public key is freely distributable and used for encryption, while the private key is kept secret and used for decryption. This dichotomy allows for secure communication over unsecured channels, as anyone can encrypt a message using the recipient's public key, but only the recipient possessing the corresponding private key can decrypt it.

Key generation in RSA involves selecting two large prime numbers, computing their product to obtain a modulus, and then selecting an additional integer to serve as the public exponent. The private exponent is then derived using modular arithmetic and number theory principles. This process ensures that the resulting keys possess the desired cryptographic properties and cannot be easily compromised.

RSA's significance extends far beyond its role as a cryptographic tool. It underpins the security of countless digital transactions, from online banking and e-commerce to secure messaging and data transmission. Its robustness and reliability have made it a cornerstone of modern cryptography, standing as a testament to the power of mathematical principles in the realm of information security.

3 BACKGROUNDS OF STUDY / PRELIMINARIES

To provide a brief mathematical overview of the RSA method and its main features, we will go over some of the number theory and cryptography principles that are employed in it.

3.1 GCD AND PRIME NUMBERS

A big part of the RSA algorithm is prime numbers. Prime numbers are natural numbers greater than 1 that can't be written as the sum of smaller natural numbers. Examples of prime numbers are 2, 3, 5, 6, 11, 13, 17, and so on. A prime number is an integer bigger than 1 that can only be divided by itself and one other positive integer.

Primes are very important in modern cryptographic systems, and they have a lot of useful features in the field. Remember that 1 is not a prime number. For the RSA algorithm to work, prime numbers are used to generate keys. In fact, the whole algorithm is built on prime numbers.

The largest positive integer that divides each of the two or more integers is their greatest common divisor, or GCD. For instance, 53 and 59 have a GCD of 1. $gcd(a, b) = 1$ indicates that a is coprime to b. As an illustration, $gcd(3, 59) = 1$ shows that 3 is coprime to 59, which thus implies that 59 is coprime to 3.

Using the Euclidean algorithm, we can quickly find the largest common divisor of a group of big numbers. It turns out that $d = gcd(a, b)$ can always be written as $d = sa + tb$, where s and t are both positive integers. It has been shown to be very slow to find the greatest common divisor of two integers by simply using their prime factorization. It takes a long time to do this way of finding prime numbers, so the Euclidean method is often used for big numbers because it finds the greatest common divisor more quickly.

If we can show that the greatest common divisors of a and b are the same as those of b and r, then $gcd(a, b) = gcd(b, r)$. This is because both pairs must have the same greatest common divisors. To give you an idea of how to use the Euclidean method to find the greatest common divisor of two integers:

$$662 = 414 \cdot 1 + 248$$
$$414 = 248 \cdot 1 + 166$$
$$248 = 166 \cdot 1 + 82$$

$$166 = 82 \cdot 2 + 2$$

$$82 = 2 \cdot 41$$

The math shows that the largest common divisor of (414, 662) is 2, since that is the last number left over.

3.2 MODULAR ARITHMETIC

In number theory, it is common for us to only be concerned with an integer's remainder when it is divided by another positive integer. There is also a particular notation for the remainders since we are usually just concerned with these. When an integer a is divided by another positive integer m, the remainder is represented by the notation $a \bmod m$.

Another related notation that is frequently used denotes the fact that when two integers are divided by another positive integer m, the remainders of the two integers are equal. In other words, if a and b are both integers and m is a positive integer, then $a \equiv b \pmod m$ if and only if $a \bmod m = b \bmod m$.

Since the modulo operator is necessary for both encryption and decryption techniques in the RSA algorithm, these modular arithmetic equations will be used repeatedly.

3.3 EULER'S THEOREM AND TOTIENT FUNCTION

Fermat's Little Theorem was previously introduced. It states that if a is not a multiple of p and p is a prime number, then $a^{p-1} = 1 \pmod p$. Euler's generalization of Fermat's theorem, however, shows that if a is relatively prime to m, then $a^{\emptyset(m)} = 1 \pmod m$, where Euler refers to $\phi(m)$ as a totient function.

This function, known as the totient function, will tally the number of positive integers that are smaller than m and relatively prime. The RSA encryption method uses Euler's theorem and two enormous prime numbers, p and q, which are kept private and $n = p \cdot q$, which is kept public. On the other hand, the private key d solves for a public key e, so that $de = 1 \pmod{\phi(n)}$. Together with calculating the public key e, it is possible to compute $\phi(n) = (p-1) \cdot (q-1)$ since the values of p and q are known.

When sending a message m with a public key e, such that $m^e \ (mod\ n)$, Euler's theorem is useful once more. Therefore, Euler's theorem is also used to compute the encrypted message, meaning that $(m^e)d \ = \ m^{\phi(n)} \ = \ m \ (mod\ n)$.

Euler's theorem and the totient function have numerous applications in number theory since they are frequently employed in primality testing, which determines whether a given integer is a prime or not. Modern cryptography heavily relies on the totient function, often known as the Phi function, which is frequently encountered in practical applications.

3.4 EXTENDED EUCLIDEAN ALGORITHM

As previously discussed, the Euclidean algorithm is used to find the greatest common divisors. As of right now, there is an extended Euclidean algorithm, which is just the Euclidean algorithm applied backwards. This algorithm extends the Euclidean algorithm by computing the coefficients of Bezout's identity, which are integers x and y such that $ax \ + \ by \ = \ gcd(a, b)$. Since the extended Euclidean algorithm computes the modular multiplicative inverse in public-key encryption and decryption methods, it is particularly helpful when the integers a and b are coprime. This algorithm is commonly utilized in current cryptography, particularly in the RSA algorithm.

The Euclidean algorithm can be used to find the numbers x and y by reversing the steps. Essentially, the algorithm starts with the greatest common divisor and works its way backwards recursively until it finds a value for the two integers, x and y.

4 METHODOLOGY

4.1 RSA ALGORITHM

Modern computers encrypt and decrypt messages using the asymmetric cryptographic method known as RSA (Rivest-Shamir-Adleman). Asymmetric cryptography, often known as public-key cryptography, uses two distinct keys for encryption and decoding. This is because one of the two keys can be distributed to anyone without jeopardizing the algorithm's security.

Two types of keys are used in the RSA algorithm: private and public. Since the public key is used to encrypt messages from plaintext to ciphertext, it is known to and available to everyone. However, only the matching private key will be able to decrypt messages that have been encrypted using this public key. Compared to other cryptographic algorithms, the RSA algorithm's high level of complexity during the key generation process is what gives it its current reputation for security and dependability.

4.2 KEY GENERATION

In contrast to symmetric algorithms like AES, public key methods necessitate computing the pair (K_{public}, $K_{private}$). The fact that these keys are computed mathematically rather than being created at random is what sets RSA apart from other encryption techniques. In contrast to most symmetric key algorithms, where the key generation step is not very difficult in terms of mathematical computations, the RSA algorithm's key creation step is highly central and significant. The RSA algorithm's key generation procedure entails the following five steps:

1. Select p and q, two large prime numbers.
2. Determine $n = p \cdot q$.
3. Compute $\Phi(n) = (p - 1) \cdot (q - 1)$.
4. If $1 < e < \Phi(n)$, then select an integer e, and:
    a) Verify that $\gcd(e, \Phi(n)) = 1$
    b) Ensure that $\Phi(n)$ and e are coprime.
5. Find the integer d such that it equals $e^{-1} \bmod \Phi(n)$.

These five stages produce two asymmetric keys that can be used for further encryption and decryption: the private key is composed of d, and the public key is made up of n and e.

$$K_{public} = (n, e) \qquad K_{private} = (d)$$

The RSA algorithm typically employs large prime numbers during the key generation process, ensuring a minimum recommended key size of 2048 bits or more. Consequently, the product of these primes, denoted as n (n = p * q), results in a total key size of 2048 bits or more. This deliberate use of large prime numbers significantly bolsters the complexity of RSA encryption, heightening the difficulty for potential brute force attacks and other forms of cryptanalysis.

4.3 ENCRYPTION AND DECRYPTION

Now that all necessary variables for key generation have been computed, the RSA technique enables both encryption and decryption of messages. This capability stems from the establishment of the public key K_{public} comprising *n* and *e*. The encryption and decryption processes rely on straightforward formulas:

Encryption: $c \equiv m^e \pmod{n}$, where m represents the plaintext message.

Decryption: $m \equiv c^d \pmod{n}$, where c denotes the ciphertext.

To ensure that only the intended recipient can decrypt messages and that decryption necessitates the specific private key, senders must employ the recipient's public key. While the private key remains confidential, the public key is shared. When sending a message *M*, it undergoes conversion via a reversible protocol called a padding scheme, ensuring the resulting value is smaller than *n*. Subsequently, the encrypted ciphertext is transmitted to the recipient.

In RSA encryption, challenges arise when plaintext values are equal to 0 or 1. Encrypting $m=0$ invariably yields a ciphertext of 0, regardless of the chosen public key. Similarly, encrypting $m=1$ results in a ciphertext of 1. These deterministic encryption outcomes present vulnerabilities, as attackers can exploit the predictable nature of such ciphertexts. To mitigate this risk, padding schemes are integral to the RSA encryption process. These schemes introduce randomness and additional structure to the plaintext before encryption, reducing the predictability associated with certain plaintext values.

Moreover, beyond the concern with plaintext 0, similar issues arise when encrypting plaintext values of 1. Encrypting $m=1$ results in a ciphertext of 1, which poses security risks if left unaddressed. Padding schemes play a crucial role in handling such cases, ensuring that RSA encryption remains robust and resistant to attacks exploiting deterministic encryption outcomes.

Incorporating padding into the encryption process enhances the security of encrypted communications by preventing attackers from exploiting patterns in the ciphertexts.

The use of randomized padding in RSA encryption is customary to prevent algorithmic vulnerabilities. By ensuring that no insecure values are present in the message and introducing padding values that yield larger ciphertexts, the encryption complexity is increased, thereby thwarting dictionary attacks, and enhancing overall security.

Once the encrypted message reaches the recipient's end of the communication channel, the decryption process utilizes the private key. With *c* representing the ciphertext, decryption is achieved through $m \equiv c^d \pmod{n}$.

5 TOY SIMULATION / EXAMPLE

5.1 PRACTICAL EXAMPLE

Example 1:

1. Naufal wants to send an encrypted message to Farah using RSA encryption. He starts by selecting two prime numbers p and q such that $1 < p, q < 100$ and $p \neq q$. For this example, let's say Naufal has chosen $p = 37$ and $q = 31$ .Calculate $n = p \cdot q$
2. Calculate Euler's Totient Function $\phi(n) = (p-1)(q-1)$.
3. Choose an *e* such that *e* is coprime to $\phi(n)$ and $1 < e < \phi(n)$.
4. Calculate the modular multiplicative inverse *e* modulo $\phi(n)$, which will be ours d .
5. Encrypt the message "Hello" using the public key (n, e).
6. Decrypt the resulting ciphertext using the private key d .

Solution:

1. Calculate n :

$$n = p \cdot q = 37 \cdot 31 = 1147$$

2. Calculate $\phi(n)$:

$$\phi(n) = (p-1)(q-1) = 36 \cdot 30 = 1080$$

3. Choose an e :

Remember, e must be coprime and $1 < e < \phi(n)$. Assume we choose $e = 407$.
Since, 407 is coprime and $1 < 407 < 1080$.

4. Calculate d :

Using the Extended Euclidean Algorithm, d is calculated to be 743 . This calculation would be quite involved to do manually, as it requires several iterations to find the coefficients that satisfy $d \cdot e \equiv mod\ \phi(n)$.

This is the output of the **Extended Euclidean Algorithm** using the numbers a=**1080** and b=**407**:

a	b	q	r	s1	s2	s3	t1	t2	t3
1080	407	2	266	1	0	1	0	1	-2
407	266	1	141	0	1	-1	1	-2	3
266	141	1	125	1	-1	2	-2	3	-5
141	125	1	16	-1	2	-3	3	-5	8
125	16	7	13	2	-3	23	-5	8	-61
16	13	1	3	-3	23	-26	8	-61	69
13	3	4	1	23	-26	127	-61	69	-337
3	1	3	0	-26	127	-407	69	-337	1080

Figure 5.1.1 Extended Euclidean Algorithm for example 1.

For encryption and decryption.

1. Encrypt the message ‘Hello’. The first character has been encrypted, and the same process happens once again for the remaining characters of the plaintext.:
    a. The ASCII value for ‘H’ is 72.
    b. The ASCII value for ‘e’ is 101.
    c. The ASCII value for ‘l’ is 108.
    d. The ASCII value for ‘o’ is 111.
        i. $c_H = 72^{407} \, mod \, 1147 = 875$
        ii. $c_e = 101^{407} \, mod \, 1147 = 529$
        iii. $c_l = 108^{407} \, mod \, 1147 = 860$
        iv. $c_o = 111^{407} \, mod \, 1147 = 851$

The entire plaintext has been encrypted and the final ciphertext is $875, 529, 860, 860, 851$. The ciphertext is sent to Farah and she decrypts the message using the same algorithm, followed by the same public key and the private key that Naufal used to encrypt the message. Farah is only able to decrypt the message because she’s aware of the private key, otherwise it would not be able to convert the ciphertext back to plaintext in terms of cryptanalysis and brute force attacks.

2. Decrypt the ciphertext:
    a. Farah obviously knows the private key $d = 743$ and uses it to decrypt the message she received from Naufal. Using the decryption formula, Farah computes $p = 875^{743} \, mod \, 1147 = 72$. The same process happens once again for the remaining blocks of the ciphertext, such that:
        i. 875: $875^{743} \, mod \, 1147 = 72 = H$
        ii. 529: $529^{743} \, mod \, 1147 = 101 = e$
        iii. 860: $860^{743} \, mod \, 1147 = 108 = l$
        iv. 851: $851^{743} \, mod \, 1147 = 111 = o$

Example 2:

1. Daniel wants to send an encrypted message to 'Aqilah using RSA encryption. He starts by selecting two prime numbers p and q such that $1 < p, q < 50$ and $p \neq q$. For this example, let's say Daniel has chosen $p = 19$ and $q = 23$ .Calculate $n = p \cdot q$
2. Calculate Euler's Totient Function $\phi(n) = (p-1)(q-1)$.
3. Choose an e such that e is coprime to $\phi(n)$ and $1 < e < \phi(n)$.
4. Calculate the modular multiplicative inverse e modulo $\phi(n)$, which will be ours d .
5. Encrypt the message "Goodbye" using the public key (n, e).
6. Decrypt the resulting ciphertext using the private key d .

Solution:

5. Calculate n :

$$n = p \cdot q = 19 \cdot 23 = 437$$

6. Calculate $\phi(n)$:

$$\phi(n) = (p-1)(q-1) = 18 \cdot 22 = 396$$

7. Choose an e :

Remember, e must be coprime and $1 < e < \phi(n)$. Assume we choose $e = 7$.
Since, 7 is coprime and $1 < 7 < 396$.

8. Calculate d :

Using the Extended Euclidean Algorithm, d is calculated to be 283 . This calculation would be quite involved to do manually, as it requires several iterations to find the coefficients that satisfy $d \cdot e \equiv mod\ \phi(n)$.

This is the output of the **Extended Euclidean Algorithm** using the numbers a=**396** and b=**7**:

a	b	q	r	s1	s2	s3	t1	t2	t3
396	7	56	4	1	0	1	0	1	-56
7	4	1	3	0	1	-1	1	-56	57
4	3	1	1	1	-1	2	-56	57	-113
3	1	3	0	-1	2	-7	57	-113	396

Figure 5.1.2 Extended Euclidean Algorithm for example 2.

For encryption and decryption.

1. Encrypt the message ‘Goodbye’. The first character has been encrypted, and the same process happens once again for the remaining characters of the plaintext.:
    a. The ASCII value for ‘G’ is 71.
    b. The ASCII value for ‘o’ is 111.
    c. The ASCII value for ‘o’ is 111.
    d. The ASCII value for ‘d’ is 100.
    e. The ASCII value for ‘b’ is 98.
    f. The ASCII value for ‘y’ is 121.
    g. The ASCII value for ‘e’ is 101.

    i. $c_G = 71^7 mod\ 437 = 174$
    ii. $c_o = 111^7\ mod\ 437 = 245$
    iii. $c_o = 111^7\ mod\ 437 = 245$
    iv. $c_d = 100^7\ mod\ 437 = 35$
    v. $c_b = 98^7\ mod\ 437 = 325$
    vi. $c_y = 121^7\ mod\ 437 = 26$
    vii. $c_e = 101^7\ mod\ 437 = 142$

The entire plaintext has been encrypted and the final ciphertext is 185 250 250 409 361 295 174 . The ciphertext is sent to ‘Aqilah and she decrypts the message using the same algorithm, followed by the same public key and the private key that Daniel used to encrypt the message. ‘Aqilah is only able to decrypt the message because she’s aware of the private key, otherwise it would not be able to convert the ciphertext back to plaintext in terms of cryptanalysis and brute force attacks.

2. Decrypt the ciphertext:
    a. ‘Aqilah obviously knows the private key $d = 283$ and uses it to decrypt the message she received from Daniel. Using the decryption formula, ‘Aqilah computes $p = 174^{283} \, mod \, 437 = 71$. The same process happens once again for the remaining blocks of the ciphertext, such that:
        i. $174: \; 174^{283} \, mod \, 437 = 71 = G$
        ii. $245: \; 245^{283} \, mod \, 437 = 111 = o$
        iii. $245: \; 245^{283} \, mod \, 437 = 111 = o$
        iv. $35: \; 35^{283} \, mod \, 437 = 100 = d$
        v. $325: \; 325^{283} \, mod \, 437 = 98 = b$
        vi. $26: \; 26^{283} \, mod \, 437 = 121 = y$
        vii. $142: \; 142^{283} \, mod \, 437 = 101 = e$

5.2 TOY SIMULATION

In this implementation, we have developed an RSA encryption and decryption mechanism in Python. The RSA algorithm enables secure data transmission and is based on the mathematical difficulty of factoring the product of two large prime numbers. Here's a breakdown of what each part of the code does:

1. **Function Definitions**:
    - **is_prime(number)**: Checks if a given number is prime. It returns **True** if the number is prime (only divisible by 1 and itself) and **False** otherwise.
    - **generate_prime(min_value, max_value)**: Generates a random prime number within a specified range. It uses the **is_prime** function to test if the randomly chosen number is prime.

- **mod_inverse(e, phi)**: Calculates the modular inverse of **e** modulo **phi**. The modular inverse is a number **d** such that **(d * e) % phi == 1**. This is used in calculating the private key in RSA.

2. **Key Generation**:
    - Two distinct prime numbers **p** and **q** are generated within a specified range. These primes are crucial for the security of RSA encryption.
    - **n = p * q**: **n** is the modulus for the public key and the private keys. It is used as part of the encryption and decryption processes.
    - **phi_n = (p - 1) * (q - 1)**: This calculates Euler's totient function (φ(n)), which is used in determining the modular inverse for the private key calculation.
    - A value **e** is chosen that is relatively prime to **phi_n** (i.e., **e** and **phi_n** share no common divisors other than 1). This **e** becomes part of the public key.
    - The modular inverse of **e** modulo **phi_n** is calculated, resulting in **d**, which is used as the private key.
3. **Encryption**:
    - The user is prompted to enter a message, which is then encoded into its ASCII values.
    - Each character of the message is encrypted using the formula **ciphertext = pow(c, e, n)**, where **c** is the ASCII value of a character, **e** is part of the public key, and **n** is the modulus. The **pow** function is used to perform modular exponentiation.
4. **Decryption**:
    - The encrypted message (ciphertext) is decrypted back to its original form using the private key **d** and the modulus **n** with the formula **decrypted_message = pow(ch, d, n)**, where **ch** is a character from the ciphertext.
    - The decrypted ASCII values are then converted back to characters to reconstruct the original message.

5. **Timing**:

    - The code also measures the time taken to encrypt and decrypt the message using **time.perf_counter()**, demonstrating the computational effort involved in the process.

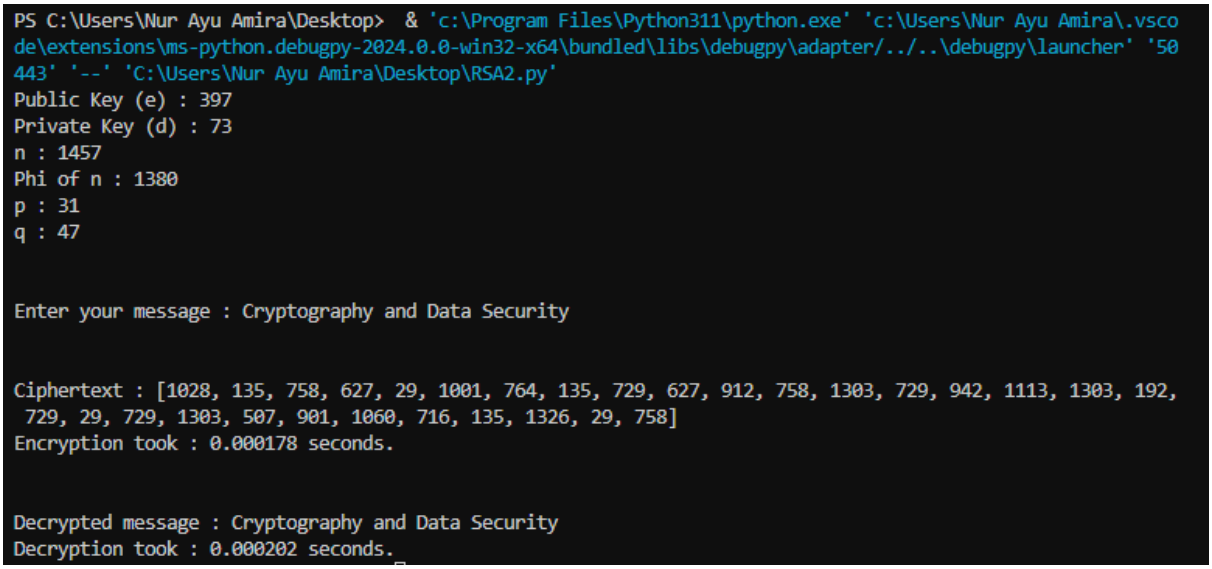

Figure 5.2.1 RSA Toy Simulation Output via Python.

The output from the code demonstrates the RSA algorithm in action, showcasing the encryption of the plaintext message into a series of numerical ciphertexts and then its successful decryption back to the original plaintext. This process also highlights the algorithm's efficiency, with the encryption and decryption times being recorded to show the algorithm's practicality for secure communications.

6 DISCUSSION / COMPARATIVE ANALYSIS

Message Size	RSA Encryption Time	RSA Decryption Time	S- AES Encryption Time	S- AES Decryption Time
10	0.000138 seconds.	0.000250 seconds.	0.000272 seconds	0.000186 seconds
20	0.000270 seconds.	0.000511 seconds.	0.000421 seconds	0.000327 seconds
30	0.000300 seconds.	0.000667 seconds.	0.000562 seconds	0.000464 seconds
40	0.000343 seconds.	0.000891 seconds.	0.000812 seconds	0.000731 seconds

Table 6.1: Data for Comparison Runtime Efficiency between RSA and S-AES.

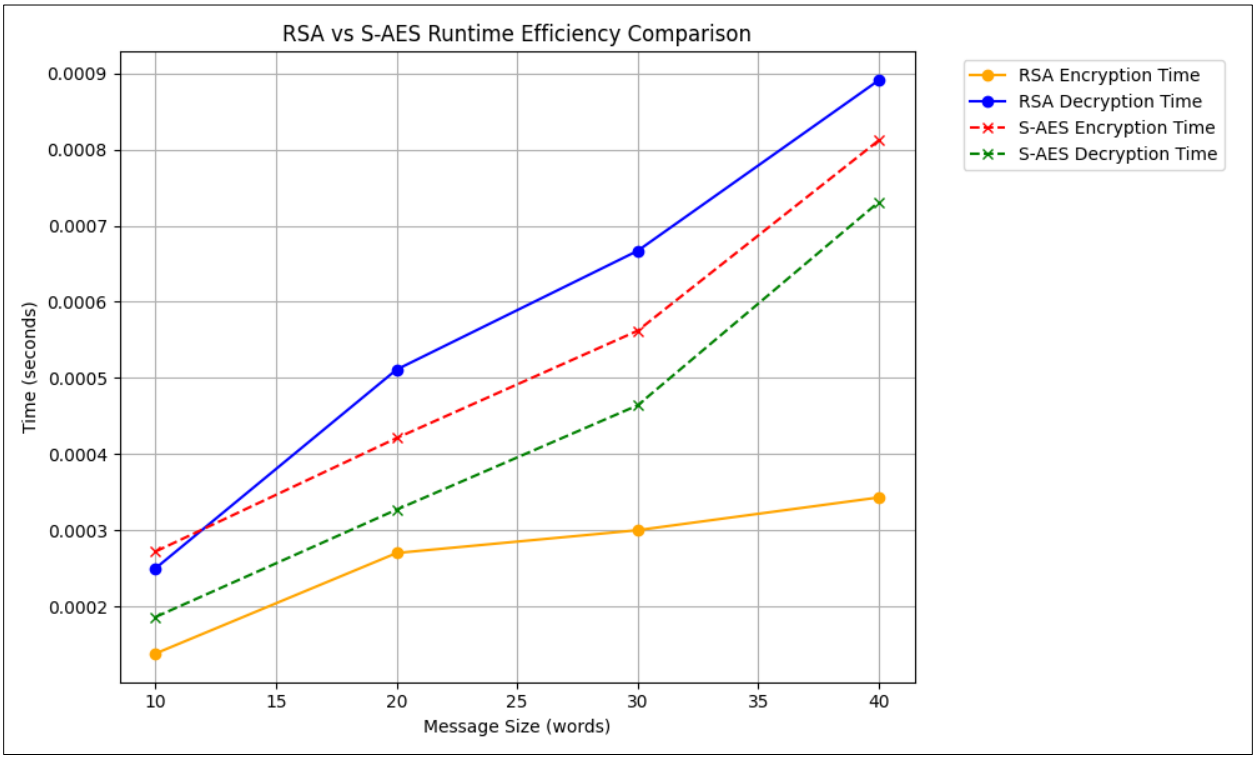

Figure 6.1: Graph for Comparison Runtime Efficiency between RSA and S-AES.

Upon examining the graph that compares the runtime efficiency between RSA and Simplified AES (S-AES) for both encryption and decryption, several insights can be derived.

For encryption, RSA starts with a lower time for small message sizes, indicating a higher initial efficiency. However, as message size increases, RSA's encryption time rises at a more gradual pace compared to S-AES. This suggests that RSA is more efficient for encrypting larger messages since its time increase is less steep relative to S-AES.

For RSA, the encryption time begins at 0.000138 seconds for a message size of 10 words and increases to 0.000343 seconds for a message size of 40 words. Similarly, the decryption time starts at 0.000250 seconds and grows to 0.000891 seconds for the same range of message sizes. The growth in time for both encryption and decryption are relatively linear, indicating a consistent increase in time as message size increases.

On the decryption side, a different trend is observed. S-AES consistently outperforms RSA across all message sizes, with decryption times for S-AES remaining below those for RSA. This consistent gap indicates that S-AES is more efficient for decryption tasks, taking less time to decrypt messages of equivalent sizes.

On the other hand, S-AES starts with an encryption time of 0.000272 seconds for a 10-word message, which is higher than RSA's initial encryption time. This time increases more significantly to 0.000812 seconds for a 40-word message. The decryption time for S-AES begins at 0.000186 seconds and increases to 0.000731 seconds for the 40-word message.

In essence, when considering the encryption process alone, RSA might be favoured, especially as the size of the message grows. For decryption, however, S-AES demonstrates superior efficiency.

Attributes	RSA	S-AES
Concept	Asymmetric cryptography algorithm for secure data transmission and authentication	Simplified variant of AES, operates on smaller block sizes for basic operations
Key Generation	RSA involves generating a pair of large prime numbers and deriving public and private keys from them	S-AES uses a simpler key generation process due to its educational nature and simplified design
Speed	Encryption is relatively fast, but decryption can be slow due to large key sizes	Generally faster than RSA for both encryption and decryption due to smaller key sizes and simplified algorithm
Key Size	Typically uses key sizes of 1024 bits for secure communication	Key sizes are significantly smaller, making it less secure but more performant for simple tasks
Advantages	Supports encryption, digital signatures, and is widely recognized for its security	Introduces the basics of AES in a more accessible way, good for understanding concepts in cryptography
Disadvantages	Requires complex key management, not suitable for all types of encryption due to computational cost	Not suitable for securing sensitive data due to lower security compared to full AES

Table 6.2: Comparative Analysis of RSA and S-AES Cryptographic Algorithms.

7 CONCLUSIONS

The RSA algorithm is a fundamental part of modern secure communication. This project has given us a clear understanding of how it encrypts and decrypts information using prime numbers. The strength of RSA lies in the difficulty of factoring large numbers, which is a neat application of a basic mathematical concept.

Throughout the implementation process, we have learned the importance of selecting appropriate prime numbers. The key takeaway is that larger primes increase security, making the encrypted message harder to crack. The hands-on experience of coding the RSA algorithm has been particularly rewarding. It was fascinating to transform the theoretical aspects of RSA into a working program. By applying programming skills to develop the encryption and decryption processes, we have gained a practical understanding that we could not have achieved through theory alone.

In conclusion, this project was not just about coding an algorithm, but also about appreciating the intricate balance between mathematical theory and practical application in the field of cryptography. The insights gained from this experience have been invaluable, and we look forward to applying them in future projects. Working through the challenges and successes of this assignment has been thoroughly enlightening.

8 REFERENCES

Zhou, X., & Tang, X. (2011). Research and implementation of RSA algorithm for encryption and decryption. *Proceedings of 2011 6th International Forum on Strategic Technology*, *2*, 1118–1121. doi:10.1109/IFOST.2011.6021216

Galla, L. K., Koganti, V. S., & Nuthalapati, N. (2016). Implementation of RSA. *2016 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, 81–87. doi:10.1109/ICCICCT.2016.7987922

Jung, A. (1987). Implementing the RSA cryptosystem. *Computers & Security*, *6*(4), 342–350. doi:10.1016/0167-4048(87)90070-8

Aufa, F. J., Endroyono, & Affandi, A. (2018). Security System Analysis in Combination Method: RSA Encryption and Digital Signature Algorithm. *2018 4th International Conference on Science and Technology (ICST)*, 1–5. doi:10.1109/ICSTC.2018.8528584

Minni, R., Sultania, K., Mishra, S., & Vincent, D. R. (2013). An algorithm to enhance security in RSA. *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 1–4. doi:10.1109/ICCCNT.2013.6726517

Al-Kaabi, S. S., & Belhaouari, S. B. (2019). Methods toward enhancing RSA algorithm: a survey. *International Journal of Network Security & Its Applications (IJNSA) Vol*, *11*.

Asjad, S. (12 2019). *The RSA Algorithm*.

9 APPENDICES

9.1 RSA2.py

```
import random
import math
import time

# The function checks if a number is prime.
def is_prime(number):
    if number < 2: # Any number less than 2 is not prime
        return False
    for i in range(2, int(math.sqrt(number)) + 1): # Loop from 2 to sqrt(number) + 1
        if number % i == 0: # If number is divisible by i, it's not prime
            return False
    return True  # If no divisors found, number is prime


# Generates a random prime number within a specified range.
def generate_prime(min_value, max_value):
    prime = random.randint(min_value, max_value) # Generate a random number within range
    while not is_prime(prime):  # Keep generating if not prime
        prime = random.randint(min_value, max_value)
    return prime # Return the prime number


# Calculates the modular inverse of e modulo phi.
def mod_inverse(e, phi):
    for d in range(3, phi): # Start loop from 3 to phi
        if (d * e) % phi == 1:
            return d # d is the modular inverse
    raise ValueError("Mod Inverse does not exist.")  # If not found, raise error

# Generate primes p and q
p, q = generate_prime(1, 100), generate_prime(1, 100)
while p == q:
    q = generate_prime(1, 100)

# Calculate n and phi(n)
n = p * q
phi_n = (p - 1) * (q - 1)

# Choose e
e = random.randint(3, phi_n - 1)
while math.gcd(e, phi_n) != 1:
```

```
    e = random.randint(3, phi_n - 1)

# Calculate d
d = mod_inverse(e, phi_n)

print("Public Key (e):", e)
print("Private Key (d):", d)
print("n:", n)
print("Phi of n:", phi_n)
print("p:", p)
print("q:", q)

# Prompt the user to enter a message
print("\n")
message = input("Enter your message: ")

start_time = time.perf_counter() # Start timing for encryption

# Encode and encrypt the message
message_encoded = [ord(c) for c in message] #converts each character to ASCII
ciphertext = [pow(c, e, n) for c in message_encoded]

end_time = time.perf_counter() # End timing
encryption_time = end_time - start_time # Calculate duration
print("\n")
print("Ciphertext:", ciphertext)
print(f"Encryption took {encryption_time:.6f} seconds.")

start_time = time.perf_counter() # Start timing for decryption

# Decrypt the message
message_decoded = [pow(ch, d, n) for ch in ciphertext]
decrypted_message = "".join(chr(ch) for ch in message_decoded)

end_time = time.perf_counter() # End timings
decryption_time = end_time - start_time # Calculate duration
print("\n")
print("Decrypted message:", decrypted_message)
print(f"Decryption took {decryption_time:.6f} seconds.")
```

9.2 S-AES.py

```
import time
import random


#This function encrypts a plaintext message using a simple symmetric encryption algorithm.
def saes_encrypt(plaintext, key):
    # Encrypts the plaintext by performing an XOR operation between
    # each character's ASCII value and the key, then converts it back to a character.
    # The key is expected to be a number between 0 and 255.
    encrypted = ''.join(chr(ord(c) ^ key) for c in plaintext)
    return encrypted

# This function decrypts a ciphertext message that was encrypted using the same key.
def saes_decrypt(ciphertext, key):
    # Decrypts the ciphertext by performing an XOR operation between
    # each character's ASCII value and the key, similar to the encryption process.
    # Since XOR is its own inverse, applying it again with the same key decrypts the message.
    decrypted = ''.join(chr(ord(c) ^ key) for c in ciphertext)
    return decrypted

# Automatically generate a numerical key (0-255)
key = random.randint(0, 255)
print(f"Generated Key: {key}")

plaintext = input("Enter your message: ")

start_time = time.perf_counter()
encrypted = saes_encrypt(plaintext, key)
end_time = time.perf_counter()
encryption_time = end_time - start_time

start_time = time.perf_counter()
decrypted = saes_decrypt(encrypted, key)
end_time = time.perf_counter()
decryption_time = end_time - start_time

print(f"Encrypted: {encrypted}")
print(f"Encryption Time: {encryption_time:.6f} seconds.")
print(f"Decrypted: {decrypted}")
print(f"Decryption Time: {decryption_time:.6f} seconds.")
```