IX1500 HT24
Project 2

Rasmus Norberg Thörn & Adalet Adiljan

September 2024

II Encryption

rasmusnt@kth.se
adalat@kth.se

1 Introduction

For this project, we learned about RSA encryption and decryption, which is one of the most widely used cryptographic methods for securing data transmission. RSA is named after its creators– Ron Rivest, Adi Shamir, and Leonard Adleman. The method is based of principles of number theory and modular arithmetic, using the difficulty of factoring very large composite numbers into primes. By understanding and implementing RSA in Python, it also highlights its importance in modern cybersecurity.
There are two keys that needs to be generated in the RSA encryption and decryption process: a public key for encryption and a private key for decryption, ensuring that only the intended recipient can acess the encrypted data.

2 Summary of the results

The following task was successfully accomplished with Python where the full code can be found here: Github Repo The final result of the RSA decryption process resulted with the plaintext message given in the explanatory diagram section. The reason that the numbers in the list seem to be of similar size can be explained by the mechanisms of the RSA algorithm and the properties of modular arithmetic used in the encryption process. The numbers in the ciphertext list are all of similar size because the plaintext message was divided into equally sized blocks (through padding and fixed block size), and each block of ciphertext is constrained within a specific range determined by the modulus n. Since n is typically a large number, it results in ciphertexts that have similar digit lengths. This process inherently produces ciphertext blocks that are numerically similar in size.

Moreover, this size consistency is reinforced by the bit length of n. For instance, if n n is 2048 bits, the corresponding ciphertexts will also be approximately 2048 bits in length when encrypted. Consequently, all ciphertexts will maintain a similar size.

Additionally, when plaintext messages are uniformly distributed and a consistent key and modulus are used, the ciphertexts generated will tend to exhibit similar numerical values. This further contributes to the impression that the numbers in the list are of the same size.

2.1 Task 1

In order for us to crack the message that Alice sends to Bob with the RSA encryption. The process begins with defining the public RSA parameters, including:

- Modulus n: the product of two large prime numbers p and q,
- Public exponent e: used for encryption
- Ciphertext: a list of encrypted ciphertext blocks that needs to be decrypted.

The first step involved to factorize n into the two prime numbers p and q. The factorization is used as following mentioned in (3) - Mathematical formulas and equations.

Once p and q are found, the second step involved to use the Euler's Totient Function covered in (4). This is a critical step in determining the unknown private key d, as its defines the number of integers less than n that are co-prime with n.

When we've obtained the important values, we then calculated the private key d, which is the modular inverse of e modulo $\Phi(n)$. Given the relation in (1), we're allowed to decrypt the cipher blocks, which corresponds to sympy's in-built function

called `mod_inverse()`, which corresponds to the relation in (5).

Finally, we decrypted each block of the cipertext using the private key d by using Python's built-in `pow()` function for efficient modular exponentiation. The decryption follows the RSA decryption formula covered in (1), where each decrypted block is converted into byte representation and then into a readable message.

3 Mathematical formulas and equations

3.1 Key Formulas

When decrypting the ciphertext, we use the following formula:

$$\text{plaintext} = c^d \mod n \tag{1}$$

This represents the RSA decryption process: plaintext is the original message that has been encrypted, c is the cipher text block (the encrypted message), d is the private key exponent, and n is the modulus (product of the two prime numbers p and q. The operation mod n ensures that the result remains within the range defined by n.

$$\text{plaintext} = \left\{(c_1^d \mod n), (c_2^d \mod n), \ldots, (c_k^d \mod n)\right\} \tag{2}$$

This notation expands the decryption formula to multiple blocks of cipher text: $(c_1, c_2, \ldots, c_k)$, where each cipher text block c_i is decrypted using the same private key d and modulus n.

This formula defines the modulus n in RSA encryption:

$$n = p * q \tag{3}$$

n is the product of two distinct large prime numbers p and q. The security of RSA relies on the difficulty of factorizing n back into its prime components.

3.1.1 Euler's Totient Function

Euler's Totient Function $\Phi(n)$ counts the number of integers less than n that are co-prime to n:

$$\Phi(n) = (p-1)(q-1) \tag{4}$$

$(p-1)$ and $(q-1)$ are the counts of integers that are co-prime (i.e. $gcd = 1$) to p and q, respectively while $\Phi(n)$ is crucial for calculating the private key d in the RSA algorithm.

3.2 Finding the private key

This formula is describes how to find the private key d:

$$d \times e \equiv 1 \pmod{\phi(n)} \tag{5}$$

The expression $d \times e \equiv 1 \pmod{\Phi(n)}$ means that d is the modular multiplicative inverse of e modulo $\Phi(n)$, where e is the public exponent used in encryption and d is obviously the private key we are trying to find.

4 Brief Discussion

Given the encryption and decryption process, we can realize that the RSA encryption provides strong security due to its difficulty of factoring large numbers. The downside of cracking an RSA code is its computational intensity; depending on

the size of the product n, the time required for decryption can vary significantly. Even in this task where n is very small compared to a proper encryption algorithm, it took a long time to factor into p and q. The complexity of the RSA decryption is primarily determined by the mathematical operations involved, particularly modular exponentiation. The time complexity for the modular exponentiation is logarithmic since it uses the method of exponentiation by squaring. Additionally, the larger the n, the more computational resources like CPU cycles and memory are required. Thus, it has become clear during the project that RSA is efficient for small-scale applications.

However, while successfully decrypting the message, implementing the decryption process in Python was a bit challenging at fist. Hard-coding the decryption logic required us to manually manage various aspects of the algorithm, such as handling large integers, modular arithmetic and byte conversion, all of demanded a deeper understanding of the underlying mathematics. While the decoding process was straight forward in Mathematica because of the clear instructions and examples in the project instruction, one of us encountered a significant decoding time delay for task 2. While the decoding process was quick for task 1 in mathematica, it instead caused a long delay in the python version. This was likely due to Python's more manual handling of large integers and modular arithmetic, which increased the time complexity of the operations. In Task 2, the growing complexity of the required operations remained still an issue. The increasing time complexity in Task 2 underscores the challenges that arise with more mathematically intensive problems, which led to our inability to solve it within the constraints (this time).

5 Explanatory Diagrams

The final result of the RSA decryption process from Alice to Bob was resulted with the following plaintext message:

> *"RSA Cryptography is the world's most widely used public-key cryptography method for securing communication on the Internet. Instrumental to the growth of e-commerce, RSA is used in almost all Internet-based transactions to safeguard sensitive data such as credit card numbers. Introduced in 1977 by MIT colleagues Ron Rivest, Adi Shamir, and Leonard Adleman, RSA -its name derived from the initials of their surnames- is a specific type of public-key cryptography, or PKC, innovated in 1976 by Whitfield Diffie, Martin Hellman, and Ralph Merkle. Intrigued by their research, Rivest, with Shamir and Adleman, developed cryptographic algorithms and techniques to practically enable secure encoding and decoding of messages between communicating parties."*

Each number in the cipher array corresponds to an encrypted word, which can be decrypted using the relation covered in (1). This process gives us (2), where each block (segment of text) in the array separated by a comma, corresponds to a sequence of characters that can form one or more words in the decrypted plaintext. The size of each block is determined by the key size which has a maximum of 2048 bits, translating to 256 bytes (assuming that no padding is applied during the encryption). Meaning, if paddings were applied then the effective size of the data in each block would be less than 256 bytes. These bytes are accumulated into a single byte string, which is then decoded using the UTF-8 encoding and finally reversed to preserve the original order of the message.

6 Conclusions

In this project, we successfully decrypted a message encrypted with RSA by factoring n, computing the private key d, and decrypting the ciphertext blocks. This

process highlights the reliance of RSA on the difficulty of factoring large numbers, making it a powerful tool for securing data in modern cryptography systems.

The results of this project underscored the efficiency of RSA for small data blocks, while also giving us the insight to the performance limitations in more complex scenarios. Given the real world scenario where we might have quantum computing in the near future, it becomes more important to evaluate and address these limitations that might occur. Quantum computing could greatly affect the security of RSA encryption, which means that we might need to actively look for other cryptographic methods that can resist these advancements.

Despite the challenges we faced, this project provided us with a deeper understanding of the encryption and decryption processes, equipping us with the knowledge necessary to tackle similar tasks in the future.

7 Code

```
import sympy

n = 126456119090476383371855906671054993650778797793018127
e = 7937

cipher = [7181325669394092429689407793421456117281087971247441 1,
94488222878280906469948648507373969381938292074762 91,
88668970435389288697377439396925326741948237682465270,
86506877126882849406016686638047102838609248170576618,
16709897999784737136957685475437549241701090506782283,
11208215095364487980886240620532479008762312664404057 3,
10130087002194592854313267155705027991809648965123930 0,
32937734818698596498554567892462717857351635451752837,
10325079564956169693399399619102665858815655800906362 6,
99446883997415524776158640105790361842457834118830 57,
11902358336688274379804393189054393684294549871847606 8,
80592157601474287498990443067778705256100803395677817,
10238050865311702888261990312482738625712667434936127 4,
29811966446563529123471275226007901312118773446042793,
92762330649448230399110375463713210616117461806861915,
52785580108219931044518308758110100269607232524031605,
96630768452430661169900035905564353166088443035700946,
10467516534820543370699962368328541763954364395250232 4,
40951632727878687548912007343839372258522783062745255,
34396485788419603319314775862549362529268070611841 28,
92627296356479225180584868594165589614134261562166537,
17702026915107984931197975326852130481340232863388490,
35046202376732485019333999169687110582305137751148612,
77294680692381954105730803435472597801358963832333113,
58483888921987241464318109604079587034521404720554634,
36276638436400152414964124035009391520390748123243684,
51639523466890776909441678913110130114797820309131676,
88728872239148972759884018820709080618086999507011767,
45676147252256101340528647372987947783315245082701701,
23720650117296688653687823869949231140410366974406435,
11687390979684202854321680927805788864742167555283362 4,
48366928605018172969920839968881382332820063246862564,
35425491594411738404916586785616696655411948001887947,
40450505118769549506412191479348341185611602935328569,
10741827078383170866369938021902715291677951378869770 2,
```

```
10120067335931080114508426779816420944861857835311695,
65616489296627251359500608540483019164860755372512518,
11847413450576524199351895472796724862275584777010578,
27312179155400713716614474364846068772702007779234 64,
10599418784042349226543806726994624123223235946860821]

#factor n using factorint() to find p,
# q and find phi(n) using the Euler's Totient
Function (p - 1) * (q - 1)
factors = sympy.factorint(n)
print("sympy.factorint(n): ", factors)
primes = list(factors.keys())
print("list(factors.keys()): ", primes)
if len(primes) != 2:
    raise ValueError("n is not a product of two primes.")
p, q = primes
phi_n = (p - 1) * (q - 1)

#find the private key d using modular inverse of e mod phi(n)
d = sympy.mod_inverse(e, phi_n)

#decrypt each ciphertext block using calculated d
def decrypt_block(c, d, n):
    return pow(c, d, n)

#decrypt all blocks in cipher going backwards
#one block at a time using the decrypt_block() function above

#[::-1] goes backwards in the cipher
decrypted_message = [decrypt_block(c, d, n) for c in cipher[::-1]]

plaintext_bytes = b''

// 8 #+7 to round up the number of bytes if needed
block_size = (n.bit_length() + 7)
for m in decrypted_message:
    m_bytes = m.to_bytes(block_size, 'big')
    #accumulate the byte strings from each decrypted block
    plaintext_bytes += m_bytes


#decode the accumulated bytes to text
try:
    plaintext = plaintext_bytes.decode('utf-8')
except UnicodeDecodeError:
    plaintext = plaintext_bytes.decode('latin-1')

print("Decrypted message:")
plaintext = plaintext[::-1] #to reverse order of text
print(plaintext)
```