The RSA Algorithm: Implementation and Technical Analysis

Introduction

The RSA algorithm, named after Ron Rivest, Adi Shamir, and Leonard Adleman, is a widely used encryption and decryption algorithm in modern communication systems. It is a public-key cryptographic system that uses two different keys, namely the public key and the private key, to perform encryption and decryption operations. The RSA algorithm is considered one of the most secure and reliable cryptographic algorithms available, making it a popular choice for various applications, including secure online transactions, secure communication, and digital signatures.

In this report, we will explore the implementation and technical analysis of the RSA algorithm. We will discuss the various steps involved in implementing the algorithm, including key generation, encryption, and decryption. Furthermore, we will analyze the security and performance aspects of the algorithm, including its vulnerability to attacks and the computational complexity of its operations. Overall, this report aims to provide a comprehensive understanding of the RSA algorithm and its practical implications.

Motivation

The motivation behind the RSA algorithm was to create a secure cryptographic system that can provide confidentiality and integrity of data transmission over unsecured communication channels. Traditional symmetric key encryption algorithms use the same key for both encryption and decryption, which makes it necessary to share the key over a secure channel. The RSA algorithm overcomes this limitation by using a pair of public and private keys, enabling secure communication over an insecure channel.

One of the significant advantages of the RSA algorithm is its capability to provide secure digital signatures, which are used to verify the authenticity and integrity of digital documents. This is achieved by using the sender's private key to sign the document, which can later be verified using the sender's public key. This is a crucial aspect of secure online transactions and electronic commerce, where secure authentication and verification are critical.

Another application of the RSA algorithm is in the field of secure online communication, where sensitive information such as passwords, credit card numbers, and other personal information are transmitted over the internet. The RSA algorithm provides a secure mechanism for encrypting and decrypting such information, ensuring that only the intended recipient can access the information.

Moreover, the RSA algorithm is used in various cryptographic protocols such as SSL/TLS, SSH, and PGP. These protocols use the RSA algorithm to generate and exchange session keys, which are then used for secure communication

between two parties.

In summary, the RSA algorithm's motivation was to create a secure cryptographic system that can provide secure communication over unsecured channels and enable secure digital signatures. Its real-world applications include secure online transactions, electronic commerce, secure online communication, and various cryptographic protocols.

Key Idea

The key idea of the RSA algorithm relies on the difficulty of factoring large numbers into their prime factors, known as the Integer Factorization Problem. Given a large composite number N, finding its prime factors p and q is computationally infeasible for sufficiently large values of N. This forms the foundation of the RSA algorithm's security since the private key is generated from the prime factors of the public key.

The RSA algorithm generates a pair of public and private keys as follows:

- Choose two large prime numbers, p and q.
- Compute their product $N = pq$.
- Compute the totient function, $\varphi(N) = (p-1)(q-1)$.
- Choose an integer e, $1 < e < \varphi(N)$, such that $gcd(e, \varphi(N)) = 1$.
- Calculate the private key d, or e's multiplicative inverse, such that $de \equiv 1 \mod \varphi(N)$.
- The pair of keys (e, N) form the public key, which is made available to anyone who wants to send a message to the owner of the private key. The private key d is kept secret by the key owner.

To encrypt a message m using the recipient's public key (e, N), the sender first converts the message into a numerical representation using a predetermined encoding scheme. The sender then computes the ciphertext c using the following formula:

$$c \equiv m^e \mod N$$

To decrypt the ciphertext c using the recipient's private key d, the recipient applies the following formula:

$$m \equiv c^d \mod N$$

Function description and Pseudocode

`function string_to_ascii(string: str) -> int:` Convert a string to its corresponding ASCII code. Input: string - the string to be converted. Output: an integer representing the ASCII code of the input string

`function ascii_to_string(ascii: int) -> str`: Convert an ASCII code to its corresponding string. Input: ascii - an integer representing the ASCII code to be converted. Output: a string representing the input ASCII code

`function cook(no_digits: int = prime_length) -> tuple`: Generate two super large prime numbers. Input: no_digits - the desired number of digits in each prime number. Output: a tuple containing the two generated prime numbers

`function prepare() -> tuple`: Generate the public and private keys for RSA encryption. Output: a tuple containing the public key (n, e) and the private key (phi, d)

```
function prepare():
    p, q = cook()  # Obtain prime numbers p and q
    n = p * q  # Calculate the modulus n
    # Calculate Euler's totient function phi(n)
    phi = (p - 1) * (q - 1)
    # Generate a random prime number e within the range [2, phi - 1]
    e = generate_random_prime(2, phi - 1)
    # Calculate the modular inverse of e modulo phi
    d = modular_inverse(e, phi)
    # Return the public key (n, e) and private key (phi, d)
    return (n, e), (phi, d)
```

`function encrypt(public_key: tuple, m: int) -> int`: Encrypt a message using RSA encryption. Input: `public_key` - a tuple containing the public key `(n, e)`, `m` - the message to be encrypted. Output: an integer representing the encrypted message (ciphertext)

```
function encrypt(public_key: tuple, m: int) -> int:
    # Extract the modulus n and exponent e from the public key
    n, e = public_key
    # Calculate the ciphertext using modular exponentiation
    ciphertext = pow(m, e, n)
    return ciphertext
```

`function decrypt(public_key: tuple, private_key: tuple, c: int) -> int`: Decrypt a ciphertext using RSA decryption. Input: `public_key` - a tuple containing the public key `(n, e)`, `private_key` - a tuple containing the private key `(phi, d)`, `c` - the ciphertext to be decrypted. Output: an integer representing the decrypted message (plaintext)

```
function decrypt(public_key, private_key: tuple, c: int) -> int:
    # Extract the private exponent d from the private key
    _, d = private_key
    # Extract the modulus n from the public key
    n, _ = public_key
    # Calculate the plaintext using modular exponentiation
```

```
    plaintext = pow(c, d, n)
    return plaintext  # Return the plaintext
```

`function brute_force_decrypt(public_key, encrypted_message)`: Attempt to decrypt the message using only public key. Input: `public_key` - a tuple containing the public key `(n, e)`, `private_key` - a tuple containing the private key `(phi, d)`, `c` - the ciphertext to be decrypted. Output: an integer representing the decrypted message (plaintext)

Asymptotic Complexity

According to the pseudocode in the previous section, we will analyze the complexity of each function:

`function string_to_ascii(string: str) -> int`: The time complexity of this function depends on the length of the input string. Let's denote the length of the string as n. The conversion requires iterating over each character in the string and performing a constant-time operation for each character. Therefore, the time complexity is O(n), where n is the length of the string.

`function ascii_to_string(ascii: int) -> str`: The time complexity of this function depends on the number of bits required to represent the input ASCII number. Let's denote the number of bits as m. The conversion requires iterating over each byte in the ASCII representation and performing a constant-time operation for each byte. Therefore, the time complexity is O(m), where m is the number of bits required to represent the ASCII number.

`function cook(no_digits: int=prime_length) -> tuple`: The time complexity of generating prime numbers is dependent on the size of the prime numbers. The specific implementation in the code utilizes the randprime function from the sympy library, which uses probabilistic primality testing algorithms. The time complexity of generating prime numbers with probabilistic primality testing is generally considered sublinear in the size of the numbers. Therefore, the time complexity of this function can be approximated as O(log^k(n)), where n is the size of the prime numbers and k is a constant.

`function prepare() -> tuple`: The time complexity of this function depends on the time complexity of the cook function, which generates the prime numbers. Therefore, the time complexity of this function is also approximately O(log^k(n)).

`function encrypt(public_key: tuple, m: int) -> int`: The time complexity of modular exponentiation using the pow function is logarithmic in the exponent. Therefore, the time complexity of this function is approximately O(log(e)), where e is the exponent.

`function decrypt(public_key, private_key: tuple, c: int) -> int`: The complexity of this function is also approximately O(log(d)), where d is the exponent.

Technology used

In this project, several technologies were used to implement the RSA algorithm and create a minimal website to serve as a client and server for encryption and decryption of messages.

To work with large prime numbers, I used the sympy library. This library provides efficient algorithms for generating large prime numbers, which are essential for the RSA algorithm. The sympy library is written in Python and is available for installation via pip.

The ascii library was used to convert between plain text and ASCII code. This library provides easy-to-use functions for converting strings to their corresponding ASCII code and vice versa.

To create a minimal website for serving the encryption and decryption of messages, the Flask framework was used to build the server. Flask is a lightweight web framework for Python that provides the necessary tools to create a web application quickly and efficiently. Flask is simple to use and provides a variety of extensions that can be used to extend its functionality.

The client-side of the website was built using plain JavaScript, which allowed for a lightweight and fast client that could communicate with the server to perform encryption and decryption of messages. I created a simple UI, but interactive enough for the user to try the encryption scheme with their own message.

Project Demo

Please refer to the source code in this Github repo for how to run the project. I also included a quick demo video there. Refer to README.md for how to navigate through the codebase.

Reference

- Milanov, Evgeny. "The RSA Algorithm." University of Washington, 3 June 2009, sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf.
- "RSA (cryptosystem)." Wikipedia, Wikimedia Foundation, 18 Apr. 2023, en.wikipedia.org/wiki/RSA_(cryptosystem).
- Aboud, Sattar. "Efficient Method for Breaking RSA Scheme." Ubiquitous Computing and Communication Journal, vol. NTMS, 2009, pp. 1-5.