

Final Year Project Report

Full Unit – Interim Report

Prime numbers and Cryptosystems

Olugbenga Hakeem-Jimoh

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Luo, Zhiyuan



Department of Computer Science
Royal Holloway, University of London

December 12, 2024

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:4815

Student Name: Olugbenga Ayodele Fawaz Hakeem - Jimoh

Date of Submission: 12/12/2024

Signature: fawaz

Table of Contents

Abstract	3
Project Specification	4
Chapter 1: Introduction	6
1.1 Overview of Modern Cryptography	6
Chapter 2: Number Theory	8
2.1 Prime numbers and their properties	8
2.2 Importance of Large Prime Numbers	9
2.3 Greatest common Divisor and Extended Euclidean Algorithm	10
2.4 Modular Arithmetic	10
2.5 Fermat's Little theorem and Euler's Totient Function	11
Chapter 3: Prime Number Generation and Testing	13
3.1 Deterministic and Probabilistic Methods	13
3.2 Primality Testing Algorithm	13
Chapter 4: Cryptographic application of prime numbers	15
4.1 Diffie-Hellman Key Exchange	15
4.2 RSA Algorithm	16
4.3 Digital Signatures	18
Chapter 5: Security concerns	20
5.1 Computational Complexity and Scalability	20
Chapter 6: Project Code and Testing	22
6.1 Main code	22
6.2 Testing	23
6.3 Project Diary	25
Bibliography	27

Abstract

This project aims to explore the relationship between prime numbers and cryptographic systems by implementing fundamental routines for prime number handling and developing cryptographic applications using these principles. The primary focus will be designing and implementing cryptographic protocols, such as RSA (Rivest-Shamir-Adleman) and the Diffie Hellman key exchange, which rely heavily on number theory, specifically the intractability of certain problems involving prime numbers. These algorithms are critical in ensuring secure communication over public and insecure channels, making them essential for applications such as bank networks, secure communications, and data encryption. The initial phase of the project will focus on building basic cryptographic tools, such as prime number generation, RSA encryption/decryption routines, and key generation using standard data types in programming languages like Java or C++. A detailed report will accompany these deliverables, describing the RSA process and showcasing examples checked by the implemented program.

The final phase will focus on developing a full-fledged cryptographic application using an object-oriented design approach, incorporating a complete implementation life cycle that follows modern software engineering principles. The program will be capable of encrypting and decrypting messages and files using integers of arbitrary size, with a graphical user interface (GUI) to facilitate secure communication tasks, such as secure chat or file transfers. The project will also integrate RSA with a symmetric encryption scheme, where RSA will distribute the keys for the symmetric encryption.

This report will offer a comprehensive overview of modern cryptography, focusing on RSA and relevant number theory issues, such as primality checking and factorization. Additionally, it will cover the implementation challenges (e.g., choice of data structures, and numerical methods) and provide insights into the software engineering process followed during the project. Performance data on the running time of the cryptographic programs will also be presented.

Project Specification

Title

Prime Numbers and Their Cryptographic Applications

Objective

The project aims to:

1. Develop fundamental routines for working with prime numbers, including prime generation and testing.
2. Implement and analyse cryptographic protocols based on prime numbers, such as Diffie-Hellman Key Exchange and RSA encryption.
3. Explore the practical applications of these protocols in secure communication and digital signatures,

Scope

This project will cover the following:

1. Theoretical exploration of prime numbers and number-theoretic concepts critical for cryptography.
2. Implementation of algorithms for primality testing, large prime generation, and cryptographic routines.
3. Evaluation of the security, efficiency, and scalability of the implemented cryptographic applications.

Requirements

1. Functional Requirements

- Develop routines for:
 - Testing primality of numbers using deterministic and probabilistic methods.
 - Efficiently generating large prime numbers for cryptographic use.
- Implement cryptographic protocols:
 - RSA encryption for secure communication and digital signatures.

2. Non-Functional Requirements

- Algorithms should be computationally efficient and scalable to large inputs.
- Implemented routines must be secure against common attacks, such as brute force and mathematical vulnerabilities.

3. Technical Requirements

- Programming language: Java for implementing algorithms.
- Tools: version control (Git) and debugging tools.

Deliverables

1. Documented routines for prime number generation and primality testing.
2. Implemented and tested cryptographic algorithms (RSA).
3. Performance evaluation report highlighting the efficiency and security of the algorithms.
4. A final project report detailing the methodology, implementation, and findings.

Timeline

1. **Week 1-2:** Literature review and background research on prime numbers and cryptographic protocols.
2. **Week 3-4:** Implement Diffie-Hellman Key Exchange and RSA algorithms.
3. **Week 5-6:** Develop routines for primality testing
4. **Week 7:** Testing, performance evaluation, and security analysis.
5. **Week 8:** Documentation, report writing, and final presentation.

Chapter 1: Introduction

1.1 Overview of Modern Cryptography

Modern cryptography is the backbone of secure communication and data protection in the digital age. It is a discipline that combines mathematical theory, computer science, and engineering to create methods for secure data storage, authentication, and transmission over potentially insecure channels. Modern cryptography is built on rigorous mathematical principles and aims to provide four key security goals: **confidentiality, integrity, authentication, and non-repudiation.**

Key Principles of Modern Cryptography

1. **Confidentiality**
Ensures that information is accessible only to those authorized to access it. Encryption algorithms, like AES and RSA, are used to transform data into unreadable formats for unauthorized users.
2. **Integrity**
Verifies that data has not been altered during transmission. Hash functions, such as SHA-256, play a critical role in ensuring data integrity.
3. **Authentication**
Confirms the identity of the entities involved in communication. Techniques like digital signatures and certificates provide robust mechanisms for authentication.
4. **Non-Repudiation**
Ensures that a sender cannot deny sending a message or performing a transaction. This is achieved through public key cryptography and digital signatures.

Modern cryptography relies on two primary concepts:

1. **Computational Intractability:** Security is based on the difficulty of solving certain mathematical problems (e.g., factoring large integers, discrete logarithm problem).
2. **Key-Based Algorithms:** Cryptographic systems use keys to secure data. Without the key, decrypting or accessing the data is computationally infeasible.

Types of Cryptographic Algorithms

1. **Symmetric Key Cryptography**
 - Uses the same key for encryption and decryption.
 - Efficient for large data but requires secure key distribution.
 - Examples: AES (Advanced Encryption Standard), DES (Data Encryption Standard).
2. **Asymmetric Key Cryptography**

- Uses a pair of keys: a public key for encryption and a private key for decryption.
- Facilitates secure key exchange and digital signatures.
- Examples: RSA, Diffie-Hellman, Elliptic Curve Cryptography (ECC).

3. Hash Functions

- Convert data into a fixed-length hash value.
- Used for data integrity and digital signatures.
- Examples: SHA-256, MD5.

Applications of Modern Cryptography

1. Secure Communication

Ensures privacy and security in emails, instant messaging, and file transfers (e.g., SSL/TLS protocols).

2. Authentication Systems

Used in systems like two-factor authentication and digital certificates.

3. E-commerce and Online Banking

Protects transactions and secures payment systems through encryption and digital signatures.

4. Blockchain and Cryptocurrencies

Relies on cryptographic hashing and public-key cryptography to secure transactions and ensure trust.

5. Government and defence

Safeguards classified information and ensures secure communication between agencies.

Challenges in Modern Cryptography

1. Quantum Computing Threat

Quantum computers could potentially break current cryptographic algorithms like RSA and ECC. Post-quantum cryptography is being developed to address this.

2. Key Management

Securely generating, distributing, and storing cryptographic keys remains a challenge.

3. Implementation Vulnerabilities

Poor implementation of cryptographic algorithms can introduce security loopholes, despite strong underlying principles.

Chapter 2: Number Theory

2.1 Prime numbers and their properties

A prime number is a natural number greater than 1 that has no divisors other than 1 and itself. In other words, a prime number p is divisible only by 1 and p . For example, the numbers 2, 3, 5, 7, 11, and 13 are all prime numbers.

The opposite of a prime number is a composite number, which has more than two distinct divisors.

Fundamental Properties of Prime Numbers

1. **Divisibility**
A prime number p cannot be expressed as the product of two smaller natural numbers other than 1 and p itself.
2. **Smallest Prime Number**
The smallest prime number is 2, and it is also the only even prime number. All other prime numbers are odd because any even number greater than 2 is divisible by 2 and hence not prime.
3. **Infinite Set of Primes**
The set of prime numbers is infinite, as proven by Euclid around 300 BCE. This proof shows that there can always be a larger prime than any given finite list of primes.
4. **Prime Factorization**
Every integer greater than 1 can be uniquely expressed as a product of prime numbers, known as its **prime factorization**. This is a cornerstone of number theory and forms the basis of modern cryptographic systems.
5. **Relatively Prime Numbers**
Two numbers are said to be relatively prime if their greatest common divisor (GCD) is 1. Prime numbers often appear in such relationships, particularly in cryptographic applications.
6. **Density of Primes**
The density of prime numbers decreases as numbers grow larger. The **Prime Number Theorem** approximates the distribution of primes, stating that the number of primes less than a number n is roughly $n / \ln(n)$, where $\ln(n)$ is the natural logarithm of n .

Properties Related to Cryptography

Difficulty of Factorization

The foundation of many cryptographic systems, such as RSA, relies on the computational difficulty of factoring large composite numbers into their prime components.

Primality Testing

Algorithms for primality testing (e.g., Miller-Rabin or AKS) are used to identify prime numbers efficiently, particularly for generating large primes for cryptographic purposes.

Large Prime Generation

Large prime numbers are critical in public-key cryptography. Their generation typically involves probabilistic tests due to the inefficiency of deterministic tests for very large numbers.

2.2 Importance of Large Prime Numbers

Large prime numbers play a critical role in modern cryptographic systems, particularly in public-key cryptography. The security of these systems is based on mathematical problems that become computationally infeasible when large prime numbers are used.

Why Large Prime Numbers?**Foundation of Cryptographic Algorithms:**

Prime numbers form the basis of essential cryptographic algorithms like RSA, Diffie-Hellman, and Elliptic Curve Cryptography (ECC).

Hardness of Key Problems:

Cryptographic security relies on the difficulty of solving problems such as:

Integer Factorization Problem (IFP): Factoring the product of two large primes ($n=p \times q$) is computationally infeasible without knowing p and q .

Discrete Logarithm Problem (DLP): Computing the exponent x in $g^x \bmod p$, where g is a generator and p is a large prime, is infeasible.

These problems underpin the security guarantees of algorithms like RSA and Diffie-Hellman.

Unpredictability and Uniqueness:

Large primes are hard to predict and randomly distributed, making them ideal for cryptographic keys.

Their unique properties ensure there are no shortcuts to solving the associated mathematical problems.

Resistance to Brute Force Attacks:

As prime numbers grow in size (e.g., 2048 bits or more), the time and resources needed to attack the system increase exponentially.

Even with advanced computing, including supercomputers, breaking such systems is practically infeasible.

2.3 Greatest common Divisor and Extended Euclidean Algorithm

Greatest Common Divisor (GCD)

The Greatest Common Divisor (GCD) of two integers a and b (not both zero) is the largest positive integer that divides both a and b without leaving a remainder.

For example:

$\text{GCD}(48, 18) = 6$, because 6 is the largest number that divides both 48 and 18.

Key Properties of GCD

1. **Divisibility Property**

If $d = \text{GCD}(a, b)$, then $d \mid a$ and $d \mid b$.

2. **Associative Property**

$\text{GCD}(a, b) = \text{GCD}(b, a)$.

3. **Relation to Linear Combinations**

For any integers a and b , there exist integers x and y such that:

$$\text{GCD}(a, b) = ax + by$$

This property is critical in number theory and forms the basis of the **Extended Euclidean Algorithm**.

4. **Recursive Property**

If $b \neq 0$, then $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$.

2.4 Modular Arithmetic

Euclidean Algorithm for Finding GCD

The Euclidean Algorithm is a highly efficient method for computing the GCD of two numbers. It uses the recursive property of GCDs:

$$\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$$

The process repeats until $b = 0$, at which point a is the GCD.

Steps of the Euclidean Algorithm:

Divide a by b , and find the remainder r :
 $r = a \bmod b$

Replace a with b and b with r .

Repeat step 1 until $b = 0$.

The GCD is the last non-zero b .

Example: Find GCD (48,18):

$$48 \div 18 = 2 \text{ remainder } 12 \text{ (r = 12)}$$

$$18 \div 12 = 1 \text{ remainder } 6 \text{ (r = 6)}$$

$$12 \div 6 = 2 \text{ remainder } 0 \text{ (r = 0)}$$

The GCD is 6.

Extended Euclidean Algorithm

The Extended Euclidean Algorithm extends the Euclidean Algorithm to find integers x and y such that:

$$\text{GCD}(a,b) = ax + by$$

These coefficients x and y are useful in many applications, such as modular inverses in cryptography.

Steps of the Extended Euclidean Algorithm:

- Start with the Euclidean Algorithm to find the GCD of a and b .
- Track the linear combinations of a and b during each step.
- Work backwards using the remainders to express the GCD as a linear combination.

Example: Solve $48x + 18y = \text{GCD}(48,18) = 6$

From the Euclidean Algorithm:

$$48 = 2 \cdot 18 + 12$$

$$18 = 1 \cdot 12 + 6$$

$$12 = 2 \cdot 6 + 0$$

Back-substitute to express 6 as a linear combination:

$$6 = 18 - 1 \cdot 12$$

$$6 = 18 - 1 \cdot (48 - 2 \cdot 18)$$

$$6 = 3 \cdot 18 - 1 \cdot 48$$

Thus, $x = -1$, $y = 3$.

2.5 Fermat's Little theorem and Euler's Totient Function

Fermat's Little Theorem

Statement:

If p is a prime number and a is an integer not divisible by p , then:

$$a^{p-1} \equiv 1 \pmod{p}$$

An alternative form states that for any integer a :

$$a^p \equiv a \pmod{p}$$

This theorem is a fundamental result in number theory and forms the basis for many cryptographic algorithms, such as RSA.

Applications of Fermat's Little Theorem:

Primality Testing

Fermat's Little Theorem can be used to test whether a number p is prime. If the theorem fails for a given a , then p is not prime. However, if it holds, p is likely prime, but not guaranteed (e.g., Carmichael numbers are exceptions).

Modular Arithmetic

The theorem simplifies the computation of large powers modulo p . For instance, instead of calculating $a^{1000} \bmod 7$, you can reduce the exponent modulo 6 (since $p - 1 = 6$ for $p = 7$):

$$a^{1000} \bmod 7 = a^{1000 \bmod 6} \bmod 7$$

Euler's Totient Function

Definition:

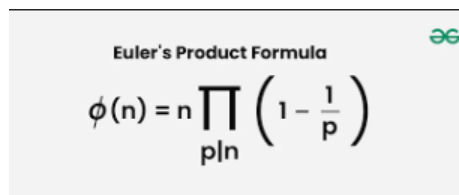
Euler's Totient Function, denoted as $\phi(n)$ counts the number of integers from 1 to n that are relatively prime to n (i.e., integers k such that $1 \leq k \leq n$ and $\text{GCD}(k, n) = 1$).

For example:

- $\Phi(9) = 6$, because the integers 1, 2, 4, 5, 7, 8 are relatively prime to 9.

Formula for Euler's Totient Function:

If n has the prime factorization $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$, then:



The diagram shows the formula for Euler's Totient Function as a product over prime factors. It includes the title 'Euler's Product Formula' and a small infinity symbol in the top right corner.

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

For specific cases:

- If $n = p^e$, where p is a prime number, then $\phi(n) = p^e - p^{e-1}$.
- If $n = p_1 p_2$ (product of two distinct primes), then $\phi(n) = (p_1 - 1)(p_2 - 1)$.

Properties of Euler's Totient Function:

1. Multiplicative Property

If a and b are coprime ($\text{GCD}(a, b) = 1$), then:

$$\Phi(ab) = \phi(a) \cdot \phi(b)$$

2. Relation to Primes

If p is prime, $\phi(p) = p - 1$ because all integers from 1 to $p - 1$ are coprime to p .

Euler's Theorem

Euler's Theorem generalizes Fermat's Little Theorem to any modulus n :

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

where a is an integer coprime to n .

For $n = p$ (a prime), $\phi(p) = p - 1$, and Euler's Theorem reduces to Fermat's Little Theorem.

Chapter 3: Prime Number Generation and Testing

Generating and testing prime numbers are critical tasks in number theory, particularly for cryptographic applications. Cryptographic protocols such as RSA and Diffie-Hellman rely on large prime numbers, typically 1024 bits or larger, to ensure security.

3.1 Deterministic and Probabilistic Methods

Deterministic Methods:

Deterministic methods guarantee an accurate result (whether a number is prime or composite) but may be computationally expensive for large numbers.

Trial Division:

The simplest method, which tests divisibility by all prime numbers up to \sqrt{n} . It is computationally infeasible for large numbers.

AKS Primality Test:

A deterministic polynomial-time algorithm that conclusively determines if n is prime. While theoretically important, it is not practical for very large numbers due to high overhead.

Probabilistic Methods:

Probabilistic methods are faster and widely used in practice. These algorithms determine whether a number is “probably prime” with a high degree of confidence.

Fermat Test:

Based on Fermat’s Little Theorem, it tests whether $a^{n-1} \equiv 1 \pmod{n}$ for a random base a . However, it can falsely identify some composite numbers (Carmichael numbers) as primes.

Miller-Rabin Primality Test:

An improvement over the Fermat test, Miller-Rabin is a probabilistic algorithm that reduces the chance of errors. It is widely used for large numbers in cryptography.

3.2 Primality Testing Algorithm

Miller-Rabin Primality Test

The **Miller-Rabin test** determines whether a given number n is composite or “probably prime.”

For an odd integer n ,

$n = 2^s d$ where,

s is a positive integer and d is an odd positive integer
an integer a and make it the base.

a is coprime to n .

coprimes are numbers which are both primes and only share 1 as a common factor

Then n is said to be a strong probable prime to base a if one of these congruence relations hold:

$$a^d \equiv 1 \pmod{n} \text{ or } a^{(2^r)d} \equiv -1 \pmod{n} \text{ for } 0 \leq r < s$$

so first checking $a^d \equiv 1 \pmod{n}$ then checking $a^{(2^r)d} \equiv -1 \pmod{n}$ for successive values of r .

If n is not a strong probable prime to base a , a is called a witness for the compositeness of n . No composite number is a strong pseudoprime to all bases at the same time (unlike fermats test which has Carmichael numbers).

Selecting a without selecting a witness is quite difficult. The miller test is used to find witnesses more efficiently.

The run time for this algorithm is also $\tilde{O}(k \log 2n)$

The average accuracy of this is 4^{-k} and it improves for larger numbers

Key Features:

- If n passes the test for several bases a , it is "probably prime."
- The probability of incorrectly classifying a composite number as prime decreases exponentially with more iterations.

Chapter 4: Cryptographic application of prime numbers

4.1 Diffie-Hellman Key Exchange

The **Diffie-Hellman Key Exchange** (DHKE) is one of the foundational protocols in cryptography, allowing two parties to establish a shared secret over an insecure communication channel. This shared secret can then be used as a symmetric key for encryption, ensuring secure communication.

How Diffie-Hellman Works

The security of the Diffie-Hellman protocol relies on the **Discrete Logarithm Problem (DLP)**, which is computationally hard to solve in large cyclic groups.

1. Setup:

Publicly agree on two numbers:

- A **large prime** p .
- A **primitive root modulo** p (also known as the generator), g .
- Both p and g are publicly known and used by all participants.

2. Key Exchange Process:

Step 1: Each party selects a private key:

- Party A chooses a private key a (a random integer).
- Party B chooses a private key b (a random integer).

Step 2: Each party computes a public key:

- A : Computes $A_{\text{pub}} = g^a \bmod p$
- B : Computes $B_{\text{pub}} = g^b \bmod p$
- Both public keys (A_{pub} and B_{pub}) are exchanged over the insecure channel.

Step 3: Each party computes the shared secret:

- A : Computes $s = B_{\text{pub}}^a \bmod p = g^{ab} \bmod p$
- B : Computes $s = A_{\text{pub}}^b \bmod p = g^{ab} \bmod p$

Both parties arrive at the same shared secret s , but an eavesdropper cannot derive it easily.

Mathematical Foundation

The security of Diffie-Hellman rests on the **Discrete Logarithm Problem (DLP)**:

Given g , p , and $g^a \bmod p$, it is computationally infeasible to determine a for large p and a .

An eavesdropper who intercepts g , p , A_{pub} , and B_{pub} cannot compute $s = g^{ab} \bmod p$ without solving the DLP.

Limitations

Man-in-the-Middle (MITM) Attacks:

If an attacker intercepts the public keys during transmission, they can impersonate both parties unless authentication mechanisms are in place.

Computational Overhead:

Exponentiation operations in $g^a \bmod p$ can be resource-intensive for very large numbers.

Enhancements**Elliptic Curve Diffie-Hellman (ECDH):**

Uses elliptic curve groups instead of modular arithmetic, providing the same level of security with much smaller key sizes.

Authenticated Diffie-Hellman:

Combines Diffie-Hellman with digital signatures or certificates to protect against MITM attacks.

Example

Let $p=23$, $g=5$:

1. Party :
 - Private key: $a = 6$
 - Public key: $A_{\text{pub}} = g^a \bmod p = 5^6 \bmod 23 = 8$
 - Private key: $b = 15$
 - Public key: $B_{\text{pub}} = g^b \bmod p = 5^{15} \bmod 23 = 19$
2. Shared secret:
 - A: $s = B_{\text{pub}}^a \bmod p = 19^6 \bmod 23 = 2$
 - B: $s = A_{\text{pub}}^b \bmod p = 8^{15} \bmod 23 = 2$

Both parties now share the secret $s=2$.

4.2 RSA Algorithm

The **RSA algorithm** is one of the most widely used public-key cryptographic systems. It enables secure data transmission, encryption, and digital signatures. RSA is named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman, who introduced it in 1977.

How RSA Works

RSA is based on the mathematical properties of prime numbers and the intractability of the **integer factorization problem**.

Key Generation

Step 1: Select two large prime numbers, p and q .

These primes must be kept secret.

Step 2: Compute n , the modulus:

$$n = p \cdot q$$

n will be part of both the public and private keys.

Step 3: Compute Euler's Totient Function $\phi(n)$:

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

This represents the number of integers less than n that are coprime to n .

Step 4: Choose a public exponent e :

Select e such that $1 < e < \phi(n)$ and $\text{GCD}(e, \phi(n)) = 1$

A common choice for e is 65537, as it is small and efficient for encryption.

Step 5: Compute the private exponent d :

Find d , the modular multiplicative inverse of $e \bmod \phi(n)$:

$$E \cdot d \equiv 1 \pmod{\phi(n)}$$

This can be calculated using the **Extended Euclidean Algorithm**.

Public Key: (e, n)

Private Key: (d, n)

Encryption

To encrypt a message M (converted to a numerical value):

$$C = M^e \bmod n$$

Where:

- C is the ciphertext.
- e and n are part of the public key.

Decryption

To decrypt the ciphertext C :

$$M = C^d \bmod n$$

- d and n are part of the private key.

The original message M is recovered.

Mathematical Foundation

RSA's security relies on the difficulty of two mathematical problems:

1. **Integer Factorization Problem:**

Factoring n into its prime factors p and q is computationally hard, especially for large n (e.g., 2048 or 4096 bits).

2. **Modular Exponentiation Inverse:**

Reversing $C = M^e \bmod n$ knowing d is infeasible without the private key.

RSA in Practice

1. **Key Size Recommendations:**

- **2048 bits:** Minimum size for most applications today.
- **4096 bits:** For highly secure environments.

2. **Hybrid Encryption:**

RSA is often combined with symmetric encryption (e.g., AES) for efficiency. RSA encrypts the symmetric key, while AES handles the bulk data encryption.

3. **Libraries and Standards:**

RSA is implemented in libraries like OpenSSL and follows standards like PKCS#1.

Limitations of RSA

- RSA is computationally intensive, especially for large data encryption.
- Symmetric algorithms like AES are faster for bulk data.
- Requires large keys (e.g., 2048 bits or more) for adequate security, leading to higher computational overhead.

- Shor's algorithm, when run on a sufficiently powerful quantum computer, could efficiently factorize n , breaking RSA.

Example of RSA Key Generation and Usage

1. Choose primes $p = 61$, $q = 53$:
 $n = 61 \cdot 53 = 3233$
 $\phi(n) = (61-1)(53-1) = 3120$.
2. Choose $e = 17$:
 $\text{GCD}(17, 3120) = 1$
3. Compute d :
 $d = 2753$ calculated as $e \cdot d \equiv 1 \pmod{3120}$

Public Key: $(e, n) = (17, 3233)$

Private Key: $(d, n) = (2753, 3233)$

Encrypt $M = 42$:

$$C = 42^{17} \bmod 3233 = 2557$$

Decrypt $C = 2557$:

$$M = 2557^{2753} \bmod 3233 = 42$$

4.3 Digital Signatures

A **digital signature** is a cryptographic mechanism used to verify the authenticity, integrity, and non-repudiation of digital messages or documents. It ensures that the message has not been altered and that it comes from a verified sender.

How Digital Signatures Work

Digital signatures use **asymmetric encryption** with two keys:

- **Private Key:** Used by the signer to generate the signature.
- **Public Key:** Used by the recipient to verify the signature.

The process involves two main steps: **signing** and **verification**.

Signing Process

The sender (signer) generates the signature using their private key:

Hashing the Message:

A cryptographic hash function (e.g., SHA-256) is applied to the message to create a fixed-size hash value (digest).

The hash ensures that even a small change in the message results in a completely different hash.

Encrypting the Hash:

The sender encrypts the hash with their private key, creating the digital signature.

The original message and the signature are sent to the recipient.

Verification Process

The recipient verifies the authenticity and integrity of the message:

Decrypting the Signature:

The recipient uses the sender's public key to decrypt the digital signature, obtaining the original hash value.

Recomputing the Hash:

The recipient computes a new hash from the received message using the same hash function.

Comparing the Hashes:

If the decrypted hash matches the newly computed hash, the signature is valid, confirming the message's authenticity and integrity.

Limitations of Digital Signatures**1. Private Key Security:**

If the private key is compromised, the signature can be forged.

2. Computational Overhead:

Signature generation and verification require significant computational resources for large-scale applications.

3. Quantum Vulnerability:

Quantum computers could potentially break widely used algorithms like RSA and ECDSA.

Example: RSA Digital Signature

- Generate a private key d and public key (e, n)
- Hash the message M to produce $H(M)$.
- Compute the signature $S = H(M)^d \bmod n$ using the private key.
- Decrypt the signature S using the public key: $H(M) = S^e \bmod n$
- Compare the result with the hash of the received message.

If they match, the signature is valid.

Chapter 5: Security concerns

5.1 Computational Complexity and Scalability

Cryptographic systems are designed to be secure and efficient, balancing robustness against attacks with practical usability. Computational complexity refers to the resources (time, space, or computational power) required to perform cryptographic operations, while scalability ensures these systems remain efficient as demands grow, such as increasing data volumes or network sizes.

Computational Complexity

Computational complexity in cryptography is measured by how the resource requirements of an algorithm grow with the size of its inputs (e.g., key size, plaintext length).

Categories of Complexity

Polynomial Time (P):

Problems solvable in $O(n^k)$ time, where n is the input size and k is a constant. Most cryptographic operations, such as AES encryption, operate in polynomial time.

Exponential Time (EXP):

Problems requiring $O(2^n)$ time, where n is the input size. Examples include brute-forcing a cryptographic key.

NP (Nondeterministic Polynomial):

Problems for which verifying a solution is possible in polynomial time but finding the solution may not be feasible.

Examples include factoring large integers (used in RSA) or solving the discrete logarithm problem (used in Diffie-Hellman).

Cryptographic security often relies on **hard problems** with exponential or super-polynomial complexity, such as:

- **Integer Factorization Problem (IFP):** Basis of RSA.
- **Discrete Logarithm Problem (DLP):** Basis of Diffie-Hellman.
- **Elliptic Curve DLP (ECDLP):** Basis of elliptic curve cryptography.

Efficient cryptographic algorithms aim to minimize computational overhead while maximizing security. For example:

- Symmetric encryption (e.g., AES) is computationally efficient, operating in polynomial time.
- Asymmetric encryption (e.g., RSA) is more computationally intensive due to reliance on hard mathematical problems.

Scalability

Scalability in cryptography refers to the ability of cryptographic systems to handle increased workloads, such as:

- Larger data sizes or higher transaction volumes.
- Greater numbers of users or devices.

Factors Affecting Scalability

Key Size and Computational Overhead:

Larger key sizes provide stronger security but increase computational demands. RSA with 4096-bit keys is more secure than 2048-bit keys but significantly slower.

Algorithm Efficiency:

Modern algorithms like ECC achieve similar security to RSA with smaller keys, improving scalability.

Parallelization:

Some cryptographic operations can be parallelized to handle larger workloads efficiently (e.g., block ciphers like AES).

Network Latency:

In distributed systems, cryptographic protocols must minimize communication overhead for scalability.

Balancing Security and Efficiency

Cryptographic systems aim to balance security with practical performance:

Symmetric Cryptography:

Fast and suitable for large-scale data encryption.

Example: AES scales well for real-time applications like video streaming.

Asymmetric Cryptography:

Slower due to mathematical complexity.

Used for smaller-scale operations like key exchange or digital signatures.

Hybrid Systems:

Combine the speed of symmetric encryption with the security of asymmetric encryption.

Example: TLS uses asymmetric cryptography for key exchange and symmetric encryption for data transmission.

Quantum Computing and Complexity

The emergence of quantum computing threatens the scalability of traditional cryptographic systems:

Shor's Algorithm:

Reduces the complexity of integer factorization and discrete logarithms from exponential to polynomial, breaking RSA and ECC.

Grover's Algorithm:

Provides a quadratic speedup for brute-force attacks on symmetric encryption, requiring larger key sizes (e.g., AES-256 instead of AES-128).

Transition to **post-quantum cryptography** (e.g., lattice-based cryptography) with hard problems resistant to quantum attacks.

Chapter 6: Project Code and Testing

6.1 Main code

```
package primality2;

import java.math.BigInteger;
import java.io.*;
import java.math.*;

public class LargerPrimes {
    // returns (x^y) % p
    static BigInteger modpower(BigInteger x, BigInteger y, BigInteger p) {

        BigInteger res = BigInteger.ONE;

        // x cannot be larger than p
        x = x.mod(p);

        while (y.compareTo(BigInteger.ZERO) > 0) {
            // If y is odd, multiply x with result
            if (y.and(BigInteger.ONE).equals(BigInteger.ONE)) // y%2 == 1
                res = (res.multiply(x)).mod(p);

            // y must be even now
            y = y.shiftRight(1); // y = y/2
            x = (x.multiply(x)).mod(p);
        }

        return res;
    }

    // This function is called for all k
    // It returns false if n is composite and returns true if n is probably
    // prime.
    // d is an odd number such that (2^r)*d = n-1 for some r >= 1
    static boolean millerTest(BigInteger d, BigInteger n) {

        // Pick a random number in [2..n-2]
        // Corner cases make sure that n > 4
        BigInteger a = BigInteger.valueOf(2)
            .add(new
                BigInteger(n.subtract(BigInteger.valueOf(4)).bitLength(), new
                    java.util.Random()).mod(n.subtract(BigInteger.valueOf(4))));

        // Compute a^d % n
        BigInteger x = modpower(a, d, n);

        if (x.equals(BigInteger.ONE) || x.equals(n.subtract(BigInteger.ONE)))
            return true;

        // Keep squaring x while one of the following doesn't happen
        // d does not reach n-1
        // (x^2) % n is not 1
        // (x^2) % n is not n-1
        while (!d.equals(n.subtract(BigInteger.ONE))) {
            x = (x.multiply(x)).mod(n);
            d = d.shiftLeft(1); // d *= 2
        }
    }
}
```

```

        if (x.equals(BigInteger.ONE)) {
            return false;
        }
        if (x.equals(n.subtract(BigInteger.ONE))) {
            return true;
        }
    }

    // Return composite
    return false;
}

// It returns false if n is composite
// and returns true if n is probably prime.
// k is an input parameter that determines accuracy level.
// Higher value of k indicates more accuracy.
static boolean isPrime(BigInteger n, int k) {

    // Corner cases
    if (n.compareTo(BigInteger.ONE) <= 0 ||
n.equals(BigInteger.valueOf(4))) {
        return false;
    }
    // case for 2 and 3
    if (n.compareTo(BigInteger.valueOf(3)) <= 0) {
        return true;
    }
    // d=n-1
    BigInteger d = n.subtract(BigInteger.ONE);
    //make d odd
    while (d.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
        d = d.divide(BigInteger.TWO);
    }

    // Iterate given number of 'k' times
    for (int i = 0; i < k; i++)
        if (!millerTest(d, n))
            return false;

    return true;
}
}

```

6.2 Testing

```

package primality2;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.math.BigInteger;

class LargerPrimesTest {
    private LargerPrimes largerPrimesChecker;

    @BeforeEach
    public void setUp() {
        largerPrimesChecker = new LargerPrimes();
    }
}

```



```

@Test
public void testIsPrime_With_20digits_Prime() {
    // Test a known large prime number with 30 digits
    BigInteger largePrime1 = new
BigInteger("636093585352712707633611402517");
    assertTrue(largerPrimesChecker.isPrime(largePrime1,5),
"636093585352712707633611402517 should be prime.");
}

@Test
public void testIsPrime_With_40digits_Prime() {
    // Test a known large prime number with 40 digits
    BigInteger largePrime2 = new
BigInteger("3670373232436798233752740931578068777707");
    assertTrue(largerPrimesChecker.isPrime(largePrime2,5),
"3670373232436798233752740931578068777707 should be prime.");
}

@Test
public void testIsPrime_With_70digits_Prime() {
    // Test a known large prime number with 70 digits
    BigInteger largePrime3 = new
BigInteger("3306255143249535762725215783282813901982286552958066097386549
634901033");
    assertTrue(largerPrimesChecker.isPrime(largePrime3,10),
"3306255143249535762725215783282813901982286552958066097386549634901033
should be prime.");
}

@Test
public void testIsPrime_With_200digits_Prime() {
    // Test a known large prime number with 200 digits
    BigInteger largePrime4 = new
BigInteger("8448024844808145225766885636922507652022559134795074145769875
5694187073835274168220758444289342689715106769742554568945911355262656562
652414251609357801774840095947028519555119850032559423832765791073");
    assertTrue(largerPrimesChecker.isPrime(largePrime4,5),
"844802484480814522576688563692250765202255913479507414576987556941870738
3527416822075844428934268971510676974255456894591135526265656265241425160
9357801774840095947028519555119850032559423832765791073 should be
prime.");
}

@Test
public void testIsPrime_With_300digit_Prime() {
    // Test a known large prime number with 300 digits
    BigInteger largePrime5 = new
BigInteger("9661159765693672142705565449157699259405445301968783906698717
1765029893400866897259444021383232563657138734670221794643470746008858225
9540361194892910171986784380227412758863824090881265847423369932172527434
5661908391735560924723129083958535814804572364723032042202277181218210028
94701992133782057657");
    assertTrue(largerPrimesChecker.isPrime(largePrime5,40),
"966115976569367214270556544915769925940544530196878390669871717650298934
0086689725944402138323256365713873467022179464347074600885822595403611948
9291017198678438022741275886382409088126584742336993217252743456619083917
3556092472312908395853581480457236472303204220227718121821002894701992133
782057657 should be prime.");
}

@Test
public void testIsPrime_WithLargeCompositel() {

```

[illegible]

DEMO-link

6.3 Project Diary

PROJECT DIARY

1-6/12/2024:

research made and code completed:

- Researched and documented how rsa works
- Researched fermats and miller-rabins primality tests
- finished primality testing for up to 300 digits commits:

- committed to the documents folder, how rsa works and primality tests.
- committed final primality testing

6-8/11/2024:

meeting with supervisor:

- reminded of work that needs to be finished urgently
- set deadline of 8/11/2024 to finish research on numbers primes, rsa and diffie-hellman key exchange commits:
- committed to the documents folder, reviews on topics studied, number theory and primes, and rsa and DH key exchange

[16/10/2024:](#)

project session lecture:

- revised version control systems and how gitlab works
- looked through gitlab rules file to understand the expected repository structure

[11/10/2024:](#)

project plan submitted

[3/10/2024](#)

lecturer meeting:

- Completed the first meeting with supervisor Luo,zhiyuan
- Discussed where to find material to complete project plan
- Brief overview of the project and understanding what challenges to look forward to
- And updated git repository for the first time

Future work

- Implement prime number generation code
- Implement RSA testing in code
- Explore other cryptographic protocols I could add to improve RSA code e.g. padding
- Showcase security efficiency with user based privileges

Bibliography

- [1] Dave Cohen and Carlos Matos. *Third Year Projects – Rules and Guidelines*. Royal Holloway, University of London, 2013.
- [2] Katz, J. (2019). *Introduction To Modern Cryptography*. CRC Press.
- [3] T Estermann (2010). *Introduction to modern prime number theory*. Cambridge: Cambridge University Press.
- [4] Lake, J. (2021). *What is the Diffie–Hellman key exchange and how does it work?* [online] Comparitech.com. Available at: <https://www.comparitech.com/blog/information-security/diffie-hellman-key-exchange/>.
- [5] Wickramasinghe, S. (2023). *RSA Algorithm in Cryptography: Rivest Shamir Adleman Explained*. [online] Splunk-Blogs. Available at: https://www.splunk.com/en_us/blog/learn/rsa-algorithm-cryptography.html.
- [6] Wikipedia Contributors (2019). Miller. [online] Wikipedia. Available at: <https://en.wikipedia.org/wiki/Miller>
- [7] Wikipedia Contributors (2019). *Fermat primality test*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Fermat_primality_test.
- [8] Chaubey, N.K. and Prajapati, B.B. (2020). *Quantum cryptography and the future of cyber security*. Hershey, PA: IGI Global.