

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ MINH
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH**



MẬT MÃ VÀ AN NINH MẠNG – CO3069

ĐỀ SỐ 3 - HIỆN THỰC RSA

GVHD: NGUYỄN CAO ĐẠT

—o0o—

Sinh viên 1: TRẦN ANH KHÔI – 2211694

Sinh viên 2: PHAN TUẤN KIỆT – 2211771

Sinh viên 3: ĐƯƠNG HẢI LÂM – 2211807

TP. HỒ CHÍ MINH, THÁNG 11 NĂM 2025

Danh sách thành viên và Phân chia công việc

Họ và tên	Mã số sinh viên	Công việc đảm nhận	Phần trăm đóng góp
Trần Anh Khôi	2211694	<ul style="list-style-type: none"> - Mở đầu và tổng kết - Hiện thực thuật toán lũy thừa modulo - Hiện thực hàm mã hóa và giải mã 	33.33%
Phan Tuấn Kiệt	2211771	<ul style="list-style-type: none"> - Phân tích và thiết kế hệ thống - Hiện thực thuật toán tìm ước số lớn nhất và Euclid mở rộng - Thực hiện cải tiến và tối ưu hóa 	33.33%
Dương Hải Lâm	2211807	<ul style="list-style-type: none"> - Tìm hiểu cơ sở lý thuyết - Hiện thực thuật toán kiểm tra tính nguyên tố Miller-Rabin - Hiện thực hàm tạo khóa và đảm bảo khóa an toàn 	33.33%

Bảng 1: Bảng phân chia công việc của các thành viên trong nhóm

Mục lục

1 Giới thiệu	1
1.1 Đặt vấn đề và mục tiêu	1
1.2 Các yêu cầu kỹ thuật cần đạt được	1
1.3 Cấu trúc bài báo cáo	1
2 Cơ sở lý thuyết	3
2.1 Tổng quan về Mật mã khóa công khai	3
2.1.1 Khái niệm và Đặc điểm cơ bản	3
2.1.2 Tính bảo mật của RSA	4
2.2 Cơ sở toán học của RSA	4
2.2.1 Phép toán Số học Modulo (Đồng dư thức)	4
2.2.2 Định lý Euler và Hàm $\phi(n)$	5
2.2.3 Nghịch đảo Modulo (Modular Multiplicative Inverse)	5
2.3 Thuật toán RSA	6
2.3.1 Tạo khóa (Key Generation)	6
2.3.2 Mã hóa (Encryption)	6
2.3.3 Giải mã (Decryption)	7
2.3.4 Chứng minh tính đúng đắn	7
2.4 Các thuật toán phụ trợ	7
2.4.1 Thuật toán Euclid và Euclid Mở rộng	7
2.4.2 Thuật toán Kiểm tra tính Nguyên tố Miller-Rabin	7
2.4.3 Thuật toán Lũy thừa Modulo Tối ưu (Square-and-Multiply)	8
2.4.4 Các Tiêu chí An toàn Bổ sung	8
3 Phân tích & Thiết kế	9
3.1 Phân tích yêu và lựa chọn công cụ	9
3.1.1 Yêu cầu và ràng buộc kĩ thuật	9
3.1.2 Lựa chọn và sử dụng công nghệ	9
3.2 Thiết kế cấu trúc module dữ liệu	9
3.2.1 Thiết kế Module Chức năng (Class Design)	9
3.2.2 Thiết kế Cấu trúc Dữ liệu Khóa (KeyPair Class)	10
3.3 Thiết kế Các thuật toán toán học cốt lõi (Utils.java)	10
3.3.1 Thiết kế Thuật toán GCD (Ước số chung lớn nhất)	10
3.3.2 Thiết kế Thuật toán Tính Lũy thừa Modulo (Modular Exponentiation)	10
3.3.3 Thiết kế Thuật toán Euclid Mở rộng và Nghịch đảo Modulo	11
3.4 Thiết kế quá trình tạo khóa	11
3.4.1 Thiết kế quá trình tạo khóa	11
3.4.2 Thiết kế hàm tạo số nguyên tố (PrimeGenerator.java)	11
3.4.3 Thiết kế cơ chế xác minh số nguyên tố (Prime Verification)	12
3.4.4 Thiết kế hàm tạo khóa công khai e và bí mật d	12
3.5 Thiết kế quá trình mã hóa và giải mã	12
3.5.1 Thiết kế hàm mã hóa	12
3.5.2 Thiết kế hàm giải mã	12
4 Hiện thực & Đánh giá	14
4.1 Môi trường và quy ước hiện thực	14
4.1.1 Môi trường phát triển và công cụ	14
4.1.2 Quy ước và tham số an toàn cụ thể	14
4.2 Hiện thực các thuật toán nền tảng	14
4.2.1 Hiện thực tính ước số chung lớn nhất (GCD)	14
4.2.2 Hiện thực thuật toán Euclid mở rộng	15
4.2.3 Hiện thực Tính Nghịch đảo Modulo d	15
4.2.4 Hiện thực Thuật toán Lũy thừa Modulo	16
4.2.5 Hiện thực hàm kiểm tra tính nguyên tố xác suất	16
4.3 Hiện thực quá trình tạo khóa	18
4.4 Hiện thực quá trình mã hóa và giải mã	19
4.4.1 Hàm mã hóa (encrypt)	19
4.4.2 Hàm giải mã (decrypt)	20
4.5 Các cải tiến và tối ưu hóa	20



4.5.1	Mã hóa an toàn với Padding OAEP (Optimal Asymmetric Encryption Padding)	20
4.5.2	Tối ưu hóa hiệu năng giải mã với CRT (Chinese Remainder Theorem)	22
4.5.3	Quy trình tạo khóa mạnh (Strong KeyPair Generation)	23
5	Kết luận	24
5.1	Tóm tắt kết quả đạt được	24
5.2	Ưu điểm và nhược điểm	24
5.3	Hướng phát triển	25

Danh sách hình vẽ

1	Mã hóa khóa công khai	3
2	OAEP Padding	22

Danh sách bảng

1	Bảng phân chia công việc của các thành viên trong nhóm	1
2	So sánh mật mã đối xứng và bất đối xứng	3

1 Giới thiệu

1.1 Đặt vấn đề và mục tiêu

Hệ mật mã RSA (Rivest–Shamir–Adleman) là một trong những hệ mật mã khóa công khai (Public-Key Cryptography) được sử dụng rộng rãi và có ảnh hưởng lớn nhất trong lịch sử mật mã học hiện đại. RSA đóng vai trò nền tảng trong các giao thức bảo mật Internet như TLS/SSL, bảo vệ việc truyền dữ liệu nhạy cảm và xác thực danh tính số. Tính bảo mật của RSA dựa trên sự khó khăn tính toán của bài toán phân tích thừa số nguyên tố (Factoring Problem) đối với các số nguyên lớn.

Mục tiêu chính của bài tập lớn này là tự hiện thực hoàn chỉnh hệ mật mã RSA bằng ngôn ngữ lập trình Java, tập trung vào việc tự xây dựng các thuật toán học nền tảng mà không sử dụng các hàm mật mã có sẵn trong thư viện. Mục tiêu này nhằm giúp người thực hiện nắm vững cơ chế toán học cốt lõi, bao gồm thuật toán tạo số nguyên tố lớn, thuật toán tìm ước số chung lớn nhất (GCD), thuật toán Euclid mở rộng, và thuật toán lũy thừa modulo, từ đó hiểu rõ hơn về tính đúng đắn và hiệu năng của hệ mật mã RSA.

1.2 Các yêu cầu kỹ thuật cần đạt được

Việc hiện thực hệ mật mã RSA phải tuân thủ nghiêm ngặt các yêu cầu kỹ thuật và ràng buộc sau:

1. Ràng buộc về sử dụng thư viện và độ lớn khóa:

- Yêu cầu cốt lõi là hiện thực phải được xây dựng từ các thuật toán cơ bản, nghiêm cấm sử dụng các phương thức mật mã có sẵn trong các thư viện (ví dụ: `BigInteger.gcd()`, `BigInteger.modPow()`, `BigInteger.modInverse()` của Java). Thay vào đó, việc xử lý số nguyên lớn phải dựa trên lớp `BigInteger` chỉ cho các phép toán số học cơ bản (cộng, trừ, nhân, mod).
- Các số nguyên tố P và Q được tạo ra phải có độ dài tối thiểu là 500 bits để đảm bảo tính an toàn mật mã của khóa modulus N. Ngoài ra, việc tạo số ngẫu nhiên cho các số nguyên tố phải sử dụng hàm ngẫu nhiên an toàn như `SecureRandom`.

2. **Tự hiện thực các thuật toán nền tảng:** Để xây dựng RSA, các thuật toán toán học sau phải được tự hiện thực:

- Tính ước số chung lớn nhất (GCD): Hiện thực thuật toán Euclid để tìm ước số chung lớn nhất của hai số nguyên lớn a và b .
- Tính khóa giải mã d (Nghịch đảo Modulo): Hiện thực thuật toán Euclid mở rộng (extendedGCD) để tính khóa bí mật d , là nghịch đảo modulo của khóa mã hóa e theo $\phi(n)$.
- Tính Lũy thừa Modulo: Hiện thực thuật toán Lũy thừa Modulo (`modPow`) để thực hiện phép toán $a^b \bmod n$ một cách hiệu quả.

3. Hiện thực chức năng tạo khóa và kiểm tra nguyên tố:

- Hiện thực thuật toán Miller-Rabin để kiểm tra tính nguyên tố xác suất của các ứng viên p và q .
- Chức năng tạo khóa có thể tự động tạo bộ khóa ngẫu nhiên, bao gồm: tính $n = p \cdot q$, tính $\phi(n) = (p-1)(q-1)$, chọn hoặc tạo giá trị e thỏa mãn $\gcd(e, \phi(n)) = 1$, và tính khóa giải mã d thỏa mãn $(e \cdot d) \bmod \phi(n) = 1$.

4. **Hiện thực chức năng mã hóa và giải mã:** Cuối cùng, hệ thống phải cung cấp các hàm chính để thực hiện quy trình mã hóa và giải mã:

- Mã hóa:** Chức năng mã hóa thông điệp m thành bản mã c bằng khóa công khai (e, n) theo công thức $c = m^e \bmod n$.
- Giải mã:** Chức năng giải mã bản mã c thành thông điệp gốc m bằng khóa bí mật (d, n) theo công thức $m = c^d \bmod n$.

1.3 Cấu trúc bài báo cáo

Báo cáo này được tổ chức thành năm chương chính nhằm trình bày một cách có hệ thống từ cơ sở lý thuyết đến kết quả hiện thực và đánh giá:



- **Chương 1 – Giới thiệu:** Trình bày bối cảnh, mục tiêu của bài tập lớn, các yêu cầu kỹ thuật cần đạt được và cấu trúc báo cáo.
- **Chương 2 – Cơ sở Lý thuyết:** Trình bày nền tảng toán học (Số học modulo, Định lý Euler) và các thuật toán mật mã quan trọng như RSA, Euclid mở rộng, Miller-Rabin và Lũy thừa Modulo
- **Chương 3 – Phân tích và Thiết kế:** Trình bày các bước phân tích yêu cầu kỹ thuật, lựa chọn công cụ và thiết kế cấu trúc module, luồng hoạt động của các thuật toán được hiện thực.
- **Chương 4 – Hiện thực và Đánh giá:** Trình bày chi tiết về môi trường phát triển, cách hiện thực các module chính, và kết quả kiểm thử chức năng và đánh giá hiệu năng của hệ thống.
- **Chương 5 – Kết luận:** Tóm tắt những kết quả đã đạt được, các hạn chế và hướng phát triển cho đề tài.

2 Cơ sở lý thuyết

2.1 Tổng quan về Mật mã khóa công khai

Mật mã khóa công khai (Public-Key Cryptography), hay còn gọi là mật mã bất đối xứng (Asymmetric Cryptography), là một bước tiến quan trọng trong lịch sử an toàn thông tin, giải quyết được vấn đề phân phối khóa vốn là điểm yếu của mật mã khóa bí mật (Symmetric Cryptography) [1, pp. 284 – 285].

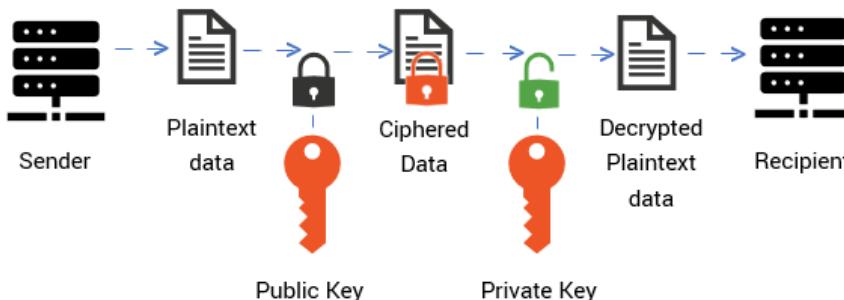
2.1.1 Khái niệm và Đặc điểm cơ bản

2.1.1.1 Mô hình mật mã khóa công khai [1, pp. 285 – 294]

Khác với mật mã đối xứng chỉ sử dụng một khóa duy nhất cho cả quá trình mã hóa và giải mã, mật mã khóa công khai sử dụng một cặp khóa (key pair) có quan hệ toán học chặt chẽ với nhau nhưng không thể suy diễn khóa này từ khóa kia

- Khóa công khai (Public Key - K_{pub}):** Khóa này được công bố rộng rãi cho tất cả mọi người. Bất kỳ ai cũng có thể sử dụng khóa này để mã hóa tin nhắn gửi cho chủ sở hữu khóa hoặc để kiểm tra chữ ký số.
- Khóa bí mật (Private Key - K_{pr}):** Khóa này được chủ sở hữu giữ bí mật tuyệt đối. Chỉ chủ sở hữu mới có thể sử dụng nó để giải mã tin nhắn nhận được hoặc tạo chữ ký số.

Public Key Encryption (Asymmetric)



Hình 1: Mã hóa khóa công khai

2.1.1.2 Sự khác biệt giữa khóa công khai và khóa bí mật

Đặc điểm	Mật mã khóa bí mật (Symmetric)	Mật mã khóa công khai (Asymmetric)
Số lượng khóa	1 khóa chung (Shared Secret Key).	2 khóa: 1 Công khai, 1 Bí mật.
Quản lý khóa	Khó khăn trong việc trao đổi khóa an toàn qua kênh không bảo mật.	Dễ dàng công bố khóa công khai mà không sợ lộ thông tin bảo mật.
Tốc độ	Rất nhanh, phù hợp mã hóa dữ liệu lớn.	Chậm hơn nhiều (khoảng 1000 lần), thường chỉ dùng để mã hóa khóa phiên hoặc chữ ký
Bảo mật	Phụ thuộc vào việc giữ bí mật khóa chung.	Phụ thuộc vào độ phức tạp tính toán của thuật toán (ví dụ: RSA, ECC).

Bảng 2: So sánh mật mã đối xứng và bất đối xứng

2.1.1.3 Ứng dụng thực tiễn

Mật mã khóa công khai phục vụ ba mục tiêu chính của an toàn thông tin [1, pp. 291 – 292]:

- Bảo mật (Confidentiality):**

- **Cơ chế:** Người gửi sử dụng K_{pub} của người nhận để mã hóa bản rõ (M) thành bản mã (C).
 - **Kết quả:** Chỉ có người nhận (người nắm giữ K_{pr} tương ứng) mới có thể giải mã để đọc nội dung. Điều này đảm bảo tính bí mật của thông tin trên đường truyền.
- **Xác thực (Authentication) & Chữ ký số (Digital Signature):**
 - **Cơ chế:** Người gửi sử dụng K_{pr} của chính mình để "ký" lên văn bản (hoặc mã băm của văn bản).
 - **Kết quả:** Bất kỳ ai cũng có thể dùng K_{pub} của người gửi để kiểm tra. Nếu khớp, điều này chứng minh văn bản thực sự xuất phát từ người nắm giữ khóa bí mật đó (Xác thực nguồn gốc) và văn bản chưa bị sửa đổi (Toàn vẹn dữ liệu).

2.1.2 Tính bảo mật của RSA

RSA (Rivest – Shamir – Adleman) là thuật toán mật mã khóa công khai phổ biến nhất hiện nay. Độ an toàn của RSA không dựa trên sự bí mật của thuật toán, mà dựa trên độ phức tạp của một bài toán số học nổi tiếng.

2.1.2.1 Bài toán phân tích thừa số nguyên tố (Integer Factorization Problem - IFP)

Cơ sở toán học của RSA dựa trên thực tế rằng việc thực hiện phép nhân hai số nguyên tố lớn rất dễ dàng, nhưng quá trình ngược lại - phân tích một hợp số lớn thành các thừa số nguyên tố - là cực kỳ khó khăn về mặt tính toán [2, p. 975].

- **Quá trình xuôi (Đề):** Chọn hai số nguyên tố lớn p và q . Tính n bằng công thức:

$$n = p \times q$$

Máy tính có thể thực hiện phép nhân này rất nhanh ngay cả khi p và q rất lớn.

- **Quá trình ngược (Khó):** Cho trước một số nguyên n rất lớn (là tích của hai số nguyên tố chưa biết), hãy tìm ra p và q . Đây là một bài toán "khó" (hard problem). Với các thuật toán và sức mạnh máy tính hiện tại, nếu n đủ lớn (ví dụ: 2048-bit hoặc 4096-bit), việc tìm ra p, q sẽ mất hàng trăm hoặc hàng nghìn năm.

2.1.2.2 Hàm cửa một chiều (Trapdoor One-way Function)

RSA hoạt động như một hàm cửa một chiều có "cửa sập" (trapdoor) [1, p. 293]:

- Hàm $f(x)$ dễ tính toán theo một chiều.
- Hàm ngược $f^{-1}(y)$ rất khó tính toán nếu không có thông tin đặc biệt.
- "Thông tin đặc biệt" (Trapdoor) ở đây chính là cặp số nguyên tố (p, q) hoặc khóa bí mật d . Nếu kẻ tấn công biết được p và q , họ có thể tính được khóa bí mật và phá vỡ hệ thống.

2.2 Cơ sở toán học của RSA

Độ an toàn và tính đúng đắn của RSA không dựa trên các thao tác bit phức tạp (như AES hay DES) mà dựa trên các nguyên lý của Số học (Number Theory). Phần này trình bày các định lý toán học cốt lõi giúp RSA hoạt động.

2.2.1 Phép toán Số học Modulo (Đồng dư thức)

Số học Modulo là hệ thống số học dành cho các số nguyên, trong đó các số "quay vòng" lại khi đạt tới một giá trị nhất định gọi là **modulus** (mô-đun) [3, p. 254].

2.2.1.1 Khái niệm đồng dư

Cho số nguyên dương n (gọi là modulus). Hai số nguyên a và b được gọi là đồng dư với nhau theo modulo n nếu chúng có cùng số dư khi chia cho n [1, p. 53].

Ký hiệu:

$$a \equiv b \pmod{n}$$

Điều này tương đương với việc hiệu của chúng chia hết cho n :

$$(a - b) \equiv k \cdot n \quad (\text{với } k \text{ là số nguyên})$$

2.2.1.2 Các tính chất cơ bản [1, p. 53]

Phép toán modulo có các tính chất quan trọng giúp đơn giản hóa việc tính toán với các số rất lớn trong RSA:

1. Tính chất cộng:

$$(a + b) \pmod{n} = [(a \pmod{n}) + (b \pmod{n})] \pmod{n}$$

2. Tính chất nhân (Quan trọng cho RSA):

$$(a \cdot b) \pmod{n} = [(a \pmod{n}) \cdot (b \pmod{n})] \pmod{n}$$

3. Tính chất lũy thừa:

$$a^b \pmod{n} = ((a \pmod{n}))^b \pmod{n}$$

Nhờ các tính chất này, quá trình mã hóa RSA ($c = m^e \pmod{n}$) có thể thực hiện hiệu quả mà không cần tính ra giá trị khổng lồ của m^e trước khi chia lấy dư.

2.2.2 Định lý Euler và Hàm $\phi(n)$

2.2.2.1 Hàm phi Euler $\phi(n)$

Hàm phi Euler, ký hiệu là $\phi(n)$, được định nghĩa là số lượng các số nguyên dương nhỏ hơn n và nguyên tố cùng nhau với n (ước chung lớn nhất bằng 1) [1, p. 65].

- Nếu p là một số nguyên tố, thì:

$$\phi(p) = p - 1$$

- Trường hợp đặc biệt trong RSA:** Nếu n là tích của hai số nguyên tố khác nhau p và q ($n = p \cdot q$), thì:

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

Lưu ý: Trong mã nguồn (KeyPair.java), giá trị này là bí mật quan trọng nhất; nếu lộ $\phi(n)$, kẻ tấn công có thể tính được khóa bí mật [4, p. 12].

2.2.2.2 Định lý Euler

Định lý Euler phát biểu rằng: Nếu a và n là hai số nguyên tố cùng nhau ($\gcd(a, n) = 1$), thì:

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

[1, p. 66]

Định lý này là cơ sở chứng minh tại sao quá trình giải mã RSA trả lại đúng thông điệp ban đầu.

2.2.3 Nghịch đảo Modulo (Modular Multiplicative Inverse)

2.2.3.1 Khái niệm

Số nguyên d được gọi là nghịch đảo modulo của e theo modulo $\phi(n)$ nếu:

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

Điều này có nghĩa là tồn tại một số nguyên k sao cho:

$$e \cdot d = 1 + k \cdot \phi(n)$$

Trong RSA:

- e : Số mũ công khai (Public Exponent).
- $\phi(n)$: Giá trị hàm Euler của modulus.
- d : Khóa bí mật (Private Key) cần tìm.

2.2.3.2 Điều kiện tồn tại

Nghịch đảo modulo d chỉ tồn tại khi và chỉ khi e và $\phi(n)$ nguyên tố cùng nhau:

$$\gcd(e, \phi(n)) = 1$$

Đây là lý do tại sao e thường được chọn là các số nguyên tố (phổ biến nhất là 65537) để đảm bảo điều kiện này dễ dàng thỏa mãn.

2.3 Thuật toán RSA

Hệ mã RSA bao gồm ba công đoạn chính: Tạo khóa (Key Generation), Mã hóa (Encryption) và Giải mã (Decryption).

2.3.1 Tạo khóa (Key Generation)

Quá trình tạo khóa là bước quan trọng nhất để thiết lập an ninh cho hệ thống. Do RSA hoạt động trên các số nguyên rất lớn, quá trình tạo khóa bao gồm 5 bước cơ bản sau đây [1, p. 302]:

1. **Bước 1: Chọn số nguyên tố.** Chọn ngẫu nhiên hai số nguyên tố lớn p và q ($p \neq q$). Yêu cầu độ dài bit của chúng phải đủ lớn (≥ 512 bits hoặc ≥ 1024 bits) để đảm bảo an toàn.
2. **Bước 2: Tính Modulus.** Tính tích của hai số nguyên tố:
$$n = p \cdot q$$

Giá trị n được dùng làm modulus cho cả khóa công khai và khóa bí mật.

3. **Bước 3: Tính hàm Euler.** Tính giá trị hàm $\phi(n)$:

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

4. **Bước 4: Chọn khóa công khai.** Chọn một số nguyên e sao cho:

$$1 < e < \phi(n) \quad \text{và} \quad \gcd(e, \phi(n)) = 1$$

Điều kiện $\gcd(e, \phi(n)) = 1$ (nguyên tố cùng nhau) đảm bảo e có nghịch đảo modulo $\phi(n)$.

Lưu ý: Trong thực tế, e thường được chọn là 65537 ($2^{16} + 1$) vì nó là số nguyên tố Fermat, giúp quá trình mã hóa diễn ra nhanh chóng nhờ cấu trúc nhị phân ít số 1 [2, p. 291].

5. **Bước 5: Tính khóa bí mật.** Tính d là nghịch đảo modulo của e :

$$d \equiv e^{-1} \pmod{\phi(n)}$$

Điều này tương đương với việc tìm d sao cho $(e \cdot d) \pmod{\phi(n)} = 1$.

Kết quả:

- **Khóa công khai (Public Key):** (e, n) .
- **Khóa bí mật (Private Key):** (d, n) .

2.3.2 Mã hóa (Encryption)

Để mã hóa một thông điệp M , trước hết chuyển đổi M thành số nguyên m ($0 \leq m < n$). Bản mã C được tính theo công thức [4, p. 6]:

$$C = m^e \pmod{n}$$

2.3.3 Giải mã (Decryption)

Để khôi phục thông điệp gốc từ bản mã C , người nhận sử dụng khóa bí mật d :

$$m = C^d \pmod{n}$$

2.3.4 Chứng minh tính đúng đắn

Ta cần chứng minh: $C^d \equiv M \pmod{n}$. *Chứng minh:* Từ thuật toán tạo khóa, ta có $e \cdot d \equiv 1 \pmod{\phi(n)}$, tức là $e \cdot d = 1 + k \cdot \phi(n)$ với $k \in \mathbb{Z}$. Thay vào biểu thức giải mã:

$$\begin{aligned} C^d &\equiv (m^e)^d \pmod{n} \\ &\equiv m^{e \cdot d} \pmod{n} \\ &\equiv m^{1+k \cdot \phi(n)} \pmod{n} \\ &\equiv m \cdot (m^{\phi(n)})^k \pmod{n} \end{aligned}$$

Theo Định lý Euler, $m^{\phi(n)} \equiv 1 \pmod{n}$, do đó:

$$m \cdot (1)^k \equiv m \pmod{n}$$

2.4 Các thuật toán phụ trợ

2.4.1 Thuật toán Euclid và Euclid Mở rộng

2.4.1.1 Thuật toán Euclid (Tính GCD)

Trong quá trình tạo khóa, e được chọn ngẫu nhiên và cần kiểm tra xem có nguyên tố cùng nhau với $\phi(n)$ hay không. Thuật toán Euclid để quyết định vấn đề này dựa trên đẳng thức [5, pp. 934 – 935]:

$$\gcd(a, b) = \gcd(b, a \pmod{b})$$

Quá trình lặp lại cho đến khi phần dư bằng 0, lúc đó số chia chính là ước chung lớn nhất.

2.4.1.2 Thuật toán Euclid Mở rộng (Tính nghịch đảo modulo)

Đây là thuật toán nền tảng để tính khóa bí mật d . Để tìm $d \equiv e^{-1} \pmod{\phi(n)}$, ta cần giải phương trình Bézout để tìm hai số nguyên x, y sao cho [5, pp. 946 – 950]:

$$e \cdot x + \phi(n) \cdot y = \gcd(e, \phi(n))$$

Nếu $\gcd(e, \phi(n)) = 1$, thì x chính là nghịch đảo cần tìm. Trong cài đặt thực tế, ta chỉ quan tâm đến giá trị x (tương ứng với d) và bỏ qua y .

2.4.2 Thuật toán Kiểm tra tính Nguyên tố Miller-Rabin

Việc sinh khóa RSA bắt đầu bằng việc tìm hai số nguyên tố lớn p và q . Do chi phí tính toán để chứng minh tính nguyên tố tuyệt đối là quá lớn, RSA sử dụng thuật toán kiểm tra xác suất Miller-Rabin [5, pp. 968 – 971].

Nguyên tắc: Thuật toán dựa trên việc kiểm tra hai tính chất số học của số nguyên tố n :

1. **Định lý nhỏ Fermat:** $a^{n-1} \equiv 1 \pmod{n}$.

2. **Căn bậc hai của đơn vị:** Phương trình $x^2 \equiv 1 \pmod{n}$ chỉ có nghiệm tầm thường $x = \pm 1$.

Nếu một số nguyên n thỏa mãn các kiểm tra này với nhiều cơ số a ngẫu nhiên, xác suất nó là hợp số là cực kỳ thấp (gần như bằng 0) [5, pp. 971 – 975].

2.4.3 Thuật toán Lũy thừa Modulo Tối ưu (Square-and-Multiply)

Các phép toán cốt lõi của RSA là $C = M^e \pmod{n}$ và $M = C^d \pmod{n}$. Với số mũ e, d lên tới 2048 bit, phép nhân lặp thông thường là bất khả thi về mặt thời gian và bộ nhớ.

Giải pháp: Thuật toán "Bình phương và Nhân" (Square-and-Multiply) giảm độ phức tạp từ $O(e)$ xuống $O(\log e)$ [2, p. 614].

Thuật toán dựa trên biểu diễn nhị phân của số mũ $e = \sum_{i=0}^k e_i 2^i$:

$$M^e = M^{\sum e_i 2^i} = \prod_{i=0}^k (M^{2^i})^{e_i} \pmod{n}$$

Điều này cho phép tính toán kết quả bằng cách thực hiện liên tiếp phép bình phương và phép nhân modulo, đảm bảo số liệu trung gian không bao giờ vượt quá n^2 .

2.4.4 Các Tiêu chí An toàn Bổ sung

Để đảm bảo an toàn trước các kỹ thuật thám mã, cặp số (p, q) sau khi sinh ra cần được kiểm tra thêm các điều kiện trong RSAPrimeVerifier.java [2, p. 291]:

- **Khoảng cách giữa p và q :** Cần đảm bảo $|p - q|$ đủ lớn. Nếu $p \approx q$, giá trị n có thể bị phân tích dễ dàng bằng Fermat Factorization ($n = x^2 - y^2$).
- **Tính trơn (Smoothness):** Các giá trị $p - 1$ và $q - 1$ không nên chỉ chứa các thừa số nguyên tố nhỏ để tránh tấn công Pollard's $p-1$.

3 Phân tích & Thiết kế

3.1 Phân tích yêu và lựa chọn công cụ

3.1.1 Yêu cầu và ràng buộc kĩ thuật

Theo yêu cầu của bài tập lớn số 3, hệ thống RSA phải được hiện thực hoàn toàn thủ công trên Java/C/C++/Python với các ràng buộc sau:

- Không sử dụng các hiện thực RSA có sẵn: Không được dùng thư viện hoặc mã nguồn RSA từ web hay từ thư viện tiêu chuẩn của các ngôn ngữ.
- Cơ chế tự hiện thực (Self-implementation): Chỉ được sử dụng thư viện big-integer (BigInteger, NTL, GMP, ...) để lưu trữ số lớn và thực hiện các phép toán cơ bản (cộng, trừ, nhân, chia, mod). Các hàm như gcd, modPow, isProbablePrime, modInverse phải tự hiện thực.
- Sinh số ngẫu nhiên lớn: Được dùng các hàm sinh số ngẫu nhiên có sẵn như SecureRandom (Java), hoặc rand() / srand() (C/C++).
- Độ dài khóa: Hai số nguyên tố P và Q phải có độ dài tối thiểu 500 bits, đảm bảo modulus $N = P \times Q$ đạt khoảng 1000–1024 bits.
- Sinh viên phải tự xây dựng đầy đủ các thành phần của RSA: Sinh số nguyên tố lớn theo số bit yêu cầu, tự cài đặt thuật toán tìm GCD, tính khóa giải mã d từ e, P, Q , sinh bộ khóa RSA ngẫu nhiên, mã hóa với (e, n) , giải mã với (d, n) ,...

3.1.2 Lựa chọn và sử dụng công nghệ

Dựa theo các yêu cầu và ràng buộc từ đề bài, nhóm đưa ra quyết định kĩ thuật như sau:

- Ngôn ngữ lập trình:** Java (Phiên bản JDK 17+).
- Quản lý số lớn:** Sử dụng java.math.BigInteger chỉ với vai trò là cấu trúc dữ liệu để lưu trữ các số nguyên vượt quá giới hạn của long (64-bit). Chỉ các phép toán số học cơ bản sau được phép sử dụng: Cộng (add), Trừ (subtract), Nhân (multiply), Chia (divide), và Lấy dư (remainder/mod).
- Sinh số ngẫu nhiên:** Sử dụng java.security.SecureRandom. Đây là nguồn entropy an toàn về mặt mật mã (CSPRNG), đảm bảo các số nguyên tố được sinh ra không thể đoán trước, khắc phục điểm yếu của java.util.Random.

3.2 Thiết kế cấu trúc module dữ liệu

Hệ thống RSA được thiết kế theo hướng đối tượng, chia nhỏ các trách nhiệm thành các module (lớp) riêng biệt để đảm bảo tính bao đóng, dễ bảo trì và mở rộng. Dưới đây là phân tích chi tiết về cấu trúc module và dữ liệu.

3.2.1 Thiết kế Module Chức năng (Class Design)

Dựa trên source code, hệ thống được chia thành 5 lớp chính, mỗi lớp đảm nhận một vai trò cụ thể trong quy trình RSA:

- Utils.java:** Đây là lớp tiện ích nền tảng, cung cấp các thuật toán học số học cốt lõi cần thiết cho RSA. Nó chứa các phương thức tĩnh (static methods) để tính ước chung lớn nhất (GCD), nghịch đảo modulo, và lũy thừa modulo. Lớp này độc lập và không chứa trạng thái (stateless).
- PrimeGenerator.java:** Module này chuyên trách việc sinh các số ngẫu nhiên và kiểm tra tính nguyên tố. Nó đóng gói thuật toán kiểm tra nguyên tố Miller-Rabin để đảm bảo các số sinh ra là số nguyên tố với xác suất cao.
- KeyPair.java:** Đây là lớp trung tâm quản lý cấu trúc dữ liệu của một bộ khóa RSA. Nó chịu trách nhiệm điều phối việc sinh hai số nguyên tố p, q , tính toán modulus n , phi $\phi(n)$, khóa công khai e và khóa bí mật d .
- RSAUtils.java:** Lớp này cung cấp các giao diện cấp cao cho người dùng cuối để thực hiện hành động mã hóa và giải mã. Nó sử dụng các hàm toán học từ Utils và dữ liệu khóa từ KeyPair.

- RSAPrimeVerifier.java: Module này đóng vai trò như một lớp bảo mật bổ sung (security layer). Sau khi PrimeGenerator sinh ra các số nguyên tố, module này sẽ thực hiện các kiểm tra khắt khe hơn (như khoảng cách giữa p và q , độ tròn của $p - 1$) để chống lại các tấn công đã biết.

3.2.2 Thiết kế Cấu trúc Dữ liệu Khóa (KeyPair Class)

Lớp KeyPair được thiết kế để lưu trữ toàn bộ các thành phần của bộ khóa RSA.

Các thuộc tính (Attributes):

- private BigInteger p: Số nguyên tố bí mật thứ nhất.
- private BigInteger q: Số nguyên tố bí mật thứ hai.
- private BigInteger modulus: Modulus $n = p \times q$, là thành phần công khai dùng chung cho cả mã hóa và giải mã.
- private BigInteger encryptKey: Số mũ công khai e .
- private BigInteger decryptKey: Số mũ bí mật d .

Tính đóng gói (Encapsulation):

- Tất cả các thuộc tính đều được khai báo là private để ngăn chặn truy cập trực tiếp từ bên ngoài, đặc biệt là các thành phần bí mật p, q, d .
- Constructor của KeyPair là private, buộc người dùng phải sử dụng phương thức factory static generateRandomKeyPair. Điều này đảm bảo rằng một đối tượng KeyPair chỉ được tạo ra thông qua quy trình sinh khóa chuẩn, đảm bảo tính toàn vẹn của dữ liệu.
- Các phương thức getter được cung cấp để truy xuất các thành phần cần thiết (như khóa công khai để gửi đi, hoặc khóa bí mật để giải mã).

3.3 Thiết kế Các thuật toán học cốt lõi (Utils.java)

Các thuật toán toán học là trái tim của hệ thống RSA. File Utils.java hiện thực thủ công các thuật toán này thay vì phụ thuộc hoàn toàn vào thư viện có sẵn để thể hiện sự hiểu biết sâu sắc về cơ chế hoạt động.

3.3.1 Thiết kế Thuật toán GCD (Ước số chung lớn nhất)

Hàm gcd(BigInteger a, BigInteger b) được sử dụng để kiểm tra tính nguyên tố cùng nhau giữa a và b .

- **Thuật toán sử dụng:** Thuật toán Euclid đệ quy.
- **Nguyên lý:** $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$. Điều kiện dừng là khi $b = 0$, lúc đó $\text{gcd} = a$.

Giả mã (Pseudo-code):

```
1 function gcd(a, b):
2     if b == 0 then
3         return a
4     end if
5     return gcd(b, a mod b)
6 end function
```

3.3.2 Thiết kế Thuật toán Tính Lũy thừa Modulo (Modular Exponentiation)

Hàm modPow(BigInteger base, BigInteger exp, BigInteger mod) thực hiện phép tính $\text{base}^{\text{exp}} \bmod \text{mod}$. Đây là phép toán tốn kém nhất và quan trọng nhất trong RSA.

- **Thuật toán sử dụng:** Bình phương và Nhân (Square-and-Multiply), hay còn gọi là Exponentiation by Squaring.

- **Ưu điểm:** Giảm độ phức tạp từ $O(\exp)$ xuống $O(\log \exp)$, cho phép tính toán với các số mũ rất lớn (hàng nghìn bit) mà không bị tràn bộ nhớ hay quá thời gian.

Sơ đồ hoạt động:

1. Khởi tạo `result = 1`.
2. Duyệt các bit của số mũ `exp` từ bit cao nhất xuống thấp nhất.
3. Tại mỗi bước, bình phương `result`: `result = (result * result) % mod`.
4. Nếu bit hiện tại là 1, nhân `result` với `base`: `result = (result * base) % mod`.
5. Trả về `result`.

3.3.3 Thiết kế Thuật toán Euclid Mở rộng và Nghịch đảo Modulo

Để tính khóa bí mật d , ta cần tìm nghịch đảo modulo của e theo modulo $\phi(n)$.

- Hàm `extendedGCD(BigInteger a, BigInteger b)`:
 - Thực hiện thuật toán Euclid mở rộng để tìm các hệ số x, y sao cho $ax + by = \gcd(a, b)$.
 - Kết quả trả về là mảng `[d, x, y]`.
- Hàm `modMulInverse(BigInteger a, BigInteger phi)`:
 - Sử dụng kết quả từ `extendedGCD(a, phi)`.
 - Nếu $\gcd(a, \phi) \neq 1$, không tồn tại nghịch đảo (báo lỗi).
 - Nếu tồn tại, giá trị nghịch đảo chính là hệ số x .
 - **Xử lý số âm:** Kết quả x từ Euclid mở rộng có thể âm. Hàm thực hiện $(x \bmod \phi + \phi) \bmod \phi$ để đảm bảo kết quả luôn dương, phù hợp với định dạng khóa RSA.

3.4 Thiết kế quá trình tạo khóa

3.4.1 Thiết kế quá trình tạo khóa

Quá trình tạo khóa nằm trong `KeyPair.generateRandomKeyPair` và `PrimeGenerator`, đảm bảo các tham số an toàn mật mã.

3.4.2 Thiết kế hàm tạo số nguyên tố (PrimeGenerator.java)

Việc sinh số nguyên tố lớn là bước đầu tiên và quan trọng nhất.

- **Hàm `generatePrime(int bitLength)`:**
 - Sinh một số lẻ ngẫu nhiên có độ dài bit yêu cầu.
 - Sử dụng vòng lặp `while(true)` để liên tục sinh và kiểm tra cho đến khi tìm được số nguyên tố.
- **Thuật toán kiểm tra (Hàm `isProbablePrime`):**
 - Sử dụng **kiểm tra Miller-Rabin**. Đây là thuật toán xác suất.
 - Tham số `CERTAINTY = 10` (số vòng lặp kiểm tra) được thiết lập để cân bằng giữa hiệu năng và độ chính xác.
 - Quy trình: Viết $n - 1 = 2^k \cdot q$. Chọn ngẫu nhiên a . Kiểm tra xem

$$a^q \equiv 1 \pmod{n} \quad \text{hoặc} \quad a^{2^j \cdot q} \equiv -1 \pmod{n} \quad \text{với } j \in [0, k-1].$$

Nếu không thỏa mãn, n là hợp số.

3.4.3 Thiết kế cơ chế xác minh số nguyên tố (Prime Verification)

Lớp RSAPrimeVerifier thực hiện các kiểm tra hậu xử lý để tránh các cặp số nguyên tố yếu:

1. Kiểm tra khoảng cách (`diff.bitLength() < MIN_BIT_DIFF_VALUE`):
 - Đảm bảo $|p - q|$ đủ lớn (ít nhất 50 bits trong cài đặt hiện tại). Nếu p và q quá gần nhau, n có thể bị phân tích dễ dàng bằng phương pháp Fermat.

2. Kiểm tra độ trơn (Smoothness Check - `isWeakSmooth`):
 - Kiểm tra xem $p - 1$ và $q - 1$ có chỉ chứa các thừa số nguyên tố nhỏ hay không.
 - Hàm `isWeakSmooth` thử chia n cho các số nhỏ (lên đến 1,000,000). Nếu sau khi chia hết mà kết quả về 1, tức là số đó "trơn" (smooth).
 - Mục đích: Chống lại tấn công Pollard's p-1.

3.4.4 Thiết kế hàm tạo khóa công khai e và bí mật d

Sau khi có p, q hợp lệ:

1. Tính $\phi(n)$:

$$\phi(n) = (p - 1) \times (q - 1).$$

2. Sinh khóa công khai e (`generateEncryptKey`):
 - Nếu người dùng không cung cấp e , hệ thống tự sinh ngẫu nhiên e trong khoảng $(1, \phi(n))$.
 - Điều kiện bắt buộc: $\text{gcd}(e, \phi(n)) = 1$ (sử dụng `Utils.gcd`). Nếu không thỏa, lặp lại việc chọn e .

3. Tính khóa bí mật d (`generateDecryptKey`):
 - Tính d sao cho

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

- Sử dụng: `d = Utils.modMulInverse(e, phi)`.

3.5 Thiết kế quá trình mã hóa và giải mã

Module RSAUtils thực hiện quy trình biến đổi dữ liệu dựa trên lý thuyết RSA.

3.5.1 Thiết kế hàm mã hóa

- **Input:** Thông điệp M (dưới dạng `BigInteger`), Khóa công khai (e, n) .
- **Kiểm tra:** $0 \leq M < n$. Nếu $M \geq n$, thông điệp không thể được mã hóa chính xác trong một khối (cần padding hoặc chia khối, nhưng trong phạm vi bài tập cơ bản này, ta giới hạn $M < n$).
- **Xử lý:**

$$C = \text{Utils.modPow}(M, e, n)$$

Tức là tính $M^e \pmod{n}$.

3.5.2 Thiết kế hàm giải mã

- **Input:** Bản mã C (dưới dạng `BigInteger`), Khóa bí mật (d, n) .

- **Xử lý:**

$$M = \text{Utils.modPow}(C, d, n)$$

Tức là tính $C^d \pmod{n}$.

- **Kết quả:** Theo định lý Euler,

$$C^d \equiv (M^e)^d \equiv M^{ed} \equiv M \pmod{n},$$

khôi phục lại thông điệp ban đầu.

4 Hiện thực & Đánh giá

4.1 Môi trường và quy ước hiện thực

Phần này trình bày môi trường đã được sử dụng để xây dựng và chạy hệ mã RSA, cùng với các quy ước và tham số an toàn đã được xác định trong mã nguồn.

4.1.1 Môi trường phát triển và công cụ

- **Ngôn ngữ lập trình:** Hệ mã RSA được hiện thực bằng ngôn ngữ Java.
- **Công cụ biên dịch và thực thi:** Quá trình biên dịch các tệp .java và thực thi chương trình chính (Main) được quản lý thông qua một shell script run.sh.
- **Quản lý số nguyên lớn:** Để xử lý các số có độ dài ≥ 500 bits, nhóm đã sử dụng lớp java.math.BigInteger. Việc sử dụng được giới hạn nghiêm ngặt chỉ ở các phép toán số học cơ bản (cộng, trừ, nhân, chia, mod) nhằm tuân thủ yêu cầu tự hiện thực các thuật toán mật mã.
- **Tạo số ngẫu nhiên an toàn:** Để đảm bảo tính ngẫu nhiên an toàn mật mã cho việc sinh số nguyên tố p, q , và e , mã nguồn sử dụng thư viện java.security.SecureRandom thay vì java.util.Random.

4.1.2 Quy ước và tham số an toàn cụ thể

Các tham số và hằng số sau đã được định nghĩa trong mã nguồn để kiểm soát độ an toàn, đặc biệt trong quá trình tạo khóa, đáp ứng khuyến nghị bảo mật hiện hành cho RSA:

- **Chiều dài Modulus tối thiểu:** Chiều dài bit tối thiểu cho modulus n được đặt là 1024 bits (MIN_BIT_LENGTH = 1024) trong lớp Main.java, đảm bảo p và q có độ dài ít nhất là 512 bits, thỏa mãn yêu cầu độ dài ≥ 500 bits.
- **Số vòng lặp Miller-Rabin:** Số vòng lặp kiểm tra tính nguyên tố (CERTAINTY) được đặt là 10 trong PrimeGenerator.java.
- **Kiểm tra an toàn bổ sung:** Module RSAPrimeVerifier.java đặt ra các tiêu chí nghiêm ngặt hơn cho việc chấp nhận p và q :
 - **Khoảng cách p và q :** Yêu cầu sự khác biệt bit tối thiểu giữa p và q là 50 bits (MIN_BIT_DIFF_VALUE = 50). Việc này nhằm ngăn chặn tấn công phân tích thừa số của Fermat (Fermat's Factorization Attack), một hình thức tấn công sẽ trở nên hiệu quả nếu p và q quá gần nhau.
 - **Kiểm tra Smoothness:** Kiểm tra $p - 1$ và $q - 1$ có các thừa số nguyên tố lớn hay không. Mục đích là để phòng ngừa tấn công $P - 1$ của Pollard (Pollard's $P - 1$ Factorization Attack), một thuật toán phân tích thừa số có thể tìm ra p hoặc q nhanh chóng nếu $p - 1$ hoặc $q - 1$ chỉ có các thừa số nguyên tố nhỏ.

4.2 Hiện thực các thuật toán nền tảng

4.2.1 Hiện thực tính ước số chung lớn nhất (GCD)

Để tính ước chung lớn nhất của hai số lớn BigInteger, nhóm sử dụng thuật toán Euclid đệ quy. Đây là phương pháp hiệu quả và dễ cài đặt nhất.

Logic cài đặt: Hàm gcd(a, b) kiểm tra nếu b bằng 0 thì trả về a . Ngược lại, nó gọi đệ quy chính nó với tham số $(b, a \% b)$.

Code hiện thực:

```
1 public static BigInteger gcd(BigInteger a, BigInteger b) {
2     if (b == BigInteger.ZERO) return a;
3     else return gcd(b, a.mod(b));
4 }
```

4.2.2 Hiệu thực thuật toán Euclid mở rộng

Thuật toán Euclid mở rộng không chỉ tìm GCD mà còn tìm hai hệ số Bezout x và y sao cho:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Logic cài đặt: Hàm extendedGCD trả về một mảng 3 phần tử $[d, x, y]$.

- **Trường hợp cơ sở:** Nếu $b = 0$, trả về $[a, 1, 0]$ vì $a \cdot 1 + 0 \cdot 0 = a$.
- **Bước đệ quy:** Gọi đệ quy để lấy $[d, x1, y1]$ từ $(b, a \% b)$. Sau đó cập nhật lại x, y theo công thức:

$$x = y1, \quad y = x1 - \left\lfloor \frac{a}{b} \right\rfloor \cdot y1$$

Code hiện thực:

```
1 private static BigInteger[] extendedGCD(BigInteger a, BigInteger b) {  
2     if (b == BigInteger.ZERO) {  
3         return new BigInteger[] { a, BigInteger.ONE, BigInteger.ZERO };  
4     } else {  
5         BigInteger[] vals = extendedGCD(b, a.mod(b));  
6         // vals[0] = gcd, vals[1] = x', vals[2] = y'  
7         BigInteger d = vals[0];  
8         BigInteger x1 = vals[1];  
9         // y = x' - (a/b) * y'  
10        BigInteger y1 = vals[1].subtract((a.divide(b)).multiply(vals[2]));  
11        return new BigInteger[] { d, x1, y1 };  
12    }  
13}
```

4.2.3 Hiệu thực Tính Nghịch đảo Modulo d

Mục tiêu là tìm d sao cho

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

Điều này tương đương với việc giải phương trình

$$e \cdot d + \phi(n) \cdot y = 1.$$

Ta sử dụng kết quả từ thuật toán Euclid mở rộng.

Logic cài đặt: Hàm modMulInverse(a, phi):

1. Gọi extendedGCD(a, phi) để lấy mảng $[gcd, x, y]$.
2. Kiểm tra gcd: Nếu khác 1, ném ngoại lệ (không tồn tại nghịch đảo).
3. Lấy x làm kết quả. Tuy nhiên, x có thể âm.
4. Chuẩn hóa x về dương bằng công thức:

$$(x \bmod \phi + \phi) \bmod \phi.$$

Code hiện thực:

```
1 public static BigInteger modMulInverse(BigInteger a, BigInteger phi) {  
2     BigInteger[] vals = extendedGCD(a, phi);  
3     BigInteger d = vals[0];  
4     BigInteger x = vals[1];  
5     if (!d.equals(BigInteger.ONE)) {  
6         throw new IllegalArgumentException("No modular inverse exists for " + a  
+ " and " + phi);  
7     } else {  
8         return x.mod(phi);  
9     }  
10}
```

```
7     } else {
8         return (x.mod(phi).add(phi)).mod(phi);
9     }
10 }
```

4.2.4 Hiệu thực Thuật toán Lũy thừa Modulo

Để thực hiện phép toán lũy thừa modulo $a^b \bmod n$ một cách hiệu quả, nhóm đã tự hiện thực thuật toán Bình phương và Nhân Modulo (Square) trong hàm modPow(base, exp, mod) thuộc tệp Utils.java.

Cơ chế hiện thực:

Thuật toán này chuyển đổi số mũ exp sang dạng nhị phân và thực hiện các phép toán nhân modulo liên tiếp:

1. **Khởi tạo:** Biến result được khởi tạo với giá trị 1.
2. **Duyệt bit:** Duyệt qua từng bit của số mũ exp từ bit có trọng số lớn nhất đến bit có trọng số nhỏ nhất (từ trái sang phải). Với mỗi lần duyệt, thực hiện các bước sau:
 - (a) **Bước bình phương:** Biến result được bình phương, sau đó modulo với giá trị tham số mod. Đây là bước bắt buộc trong mọi vòng lặp, đảm bảo độ phức tạp tính toán được giữ ở mức $O(\log \exp)$
 - (b) **Bước nhân:** Nếu bit hiện tại của số mũ exp là 1, biến result được gán giá trị như sau: `result = (result * base) mod mod`

Code hiện thực

```
1 public static BigInteger modPow(BigInteger base, BigInteger exp, BigInteger mod)
2 {
3     BigInteger result = BigInteger.ONE;
4     for (int i = exp.bitLength() - 1; i >= 0; i--) {
5         result = (result.mod(mod).multiply(result.mod(mod))).mod(mod);
6         if (exp.testBit(i)) {
7             result = (result.mod(mod).multiply(base.mod(mod))).mod(mod);
8         }
9     }
10 }
```

4.2.5 Hiệu thực hàm kiểm tra tính nguyên tố xác suất

Để thực hiện chức năng kiểm tra tính nguyên tố xác suất cho các số nguyên tố lớn p và q , nhóm đã tự hiện thực Thuật toán Miller-Rabin trong hàm isProbablePrime(BigInteger n, int millerRabinRounds) thuộc lớp PrimeGenerator.java.

Cơ chế hiện thực của hàm isProbablePrime tập trung vào việc đảm bảo tốc độ và độ chính xác:

1. Xử lý các trường hợp cơ bản và số nguyên tố nhỏ:
 - Hàm kiểm tra nhanh các trường hợp $n < 2$, $n = 2$, $n = 3$, và các số chẵn (trừ 2) để loại bỏ các hợp số đơn giản.
 - Thuật toán tiến hành kiểm tra chia hết cho một danh sách các số nguyên tố nhỏ (SMALL_PRIMES) trước khi chuyển sang các bước tính toán phức tạp hơn. Việc này giúp loại bỏ nhanh chóng phần lớn các hợp số trong giai đoạn đầu, tiết kiệm thời gian đáng kể.
2. Phân tách cơ sở Miller-Rabin:
 - Số nguyên $n - 1$ được phân tách thành dạng $2^k \cdot q$, với q là số lẻ.

- Quá trình tính k và q được tối ưu bằng cách sử dụng các phép toán Bitwise (`getLowestSetBit()` và `shiftRight()`) thay vì phép chia modulo truyền thống, giúp tăng tốc quá trình phân tách này

3. Vòng lặp kiểm tra Miller-Rabin (Testing Loop):

- Thuật toán tiến hành lặp lại quá trình kiểm tra với số lần lặp xác định (`millerRabinRounds`) để tăng độ chính xác xác suất.
- Trong mỗi vòng lặp, một cơ số ngẫu nhiên a được chọn trong khoảng $[2, n - 2]$.
- Tính toán giá trị khởi tạo $x = a^q \bmod n$ bằng cách sử dụng hàm (`Utils.modPow`) (Lũy thừa Modulo đã tự hiện thực).
- Kiểm tra điều kiện tiên quyết: Nếu $x \equiv 1 \pmod{n}$ hoặc $x \equiv -1 \pmod{n}$ (tức là $x = n - 1$), n có khả năng là số nguyên tố và thuật toán chuyển sang cơ số a tiếp theo.
- Vòng lặp Bình phương: Nếu điều kiện trên không được thỏa mãn, thuật toán lặp lại phép bình phương $x = x^2 \bmod n$ trong $k - 1$ lần. Nếu x đạt giá trị $n - 1$, n vượt qua bài kiểm tra. Nếu x đạt giá trị 1 trước khi đạt $n - 1$, n là hợp số.

Nếu n vượt qua tất cả các vòng kiểm tra với các cơ số ngẫu nhiên, nó được coi là số nguyên tố xác suất cao.

Code hiện thực

```
1 private static boolean isProbablePrime(BigInteger n, int millerRabinRounds) {
2     // Handle simple cases
3     if (n.compareTo(BigInteger.TWO) < 0)
4         return false;
5     if (n.equals(BigInteger.TWO) || n.equals(BigInteger.valueOf(3)))
6         return true;
7     if (n.mod(BigInteger.TWO).equals(BigInteger.ZERO))
8         return false;
9     if (!n.testBit(0))
10        return false;
11
12    // check small primes first
13    for (int p : SMALL_PRIMES) {
14        BigInteger bigP = BigInteger.valueOf(p);
15        if (n.equals(bigP))
16            return true;
17        if (n.mod(bigP).equals(BigInteger.ZERO))
18            return false;
19    }
20
21    BigInteger nMinus1 = n.subtract(BigInteger.ONE);
22    int k = nMinus1.getLowestSetBit();
23    BigInteger q = nMinus1.shiftRight(k);
24
25    SecureRandom random = new SecureRandom();
26
27    for (int i = 0; i < millerRabinRounds; i++) {
28        BigInteger a = uniformRandom(BigInteger.TWO, n.subtract(BigInteger.TWO),
29                                     random);
30        BigInteger x = Utils.modPow(a, q, n);
31
32        if (x.equals(BigInteger.ONE) || x.equals(nMinus1))
33            continue;
34
35        boolean isComposite = true;
36        for (int j = 0; j < k - 1; j++) {
37            x = Utils.modPow(x, BigInteger.TWO, n);
38
39            if (x.equals(nMinus1)) {
40                isComposite = false;
41                break;
42            }
43        }
44    }
45 }
```

```
42         if (x.equals(BigInteger.ONE)) {
43             return false;
44         }
45     }
46
47     if (isComposite)
48         return false;
49     }
50
51     return true;
52 }
```

4.3 Hiệu thực quá trình tạo khóa

Để thực hiện chức năng tạo khóa cơ bản, hàm `generateRandomKeyPair(int bitLength)` trong lớp `KeyPair.java` được thiết kế để tạo ra cặp khóa RSA hợp lệ, đồng thời áp dụng các kiểm tra an toàn tối thiểu theo yêu cầu của bài tập lớn.

Quá trình tạo khóa này diễn ra trong một vòng lặp `do-while` nhằm đảm bảo tính hợp lệ của cặp khóa được sinh ra:

1. Sinh p và q cơ bản:

- Hai số nguyên tố p và q được sinh ra bằng cách gọi hàm `PrimeGenerator.generatePrime(bitLength / 2)`, với độ dài bit bằng một nửa độ dài khóa modulus `bitLength`.
- Trong một vòng lặp `while` lồng bên trong, thuật toán kiểm tra để đảm bảo $p \neq q$.

2. Tính toán modulus n và $\phi(n)$:

- Modulus n được tính bằng $n = p \cdot q$.
- Giá trị $\phi(n)$ (hàm phi Euler) được tính bằng $\phi(n) = (p - 1)(q - 1)$.

3. Lựa chọn khóa công khai e : Khóa công khai e được cố định là 65537.

4. Kiểm tra khóa cuối cùng: Vòng lặp `do-while` ngoài cùng kiểm tra hai điều kiện quan trọng:

- Kiểm tra an toàn bổ sung: Cặp (p, q) phải vượt qua các kiểm tra an toàn cơ bản (bao gồm kiểm tra khoảng cách và tính smoothness yếu của $p - 1$ và $q - 1$) thông qua `RSAPrimeVerifier.verifyPrimeForRSA(p, q)`.
- Tính nguyên tố cùng nhau: Khóa e phải là nguyên tố cùng nhau với $\phi(n)$ ($\gcd(\phi(n), e) = 1$).

5. Tính toán khóa bí mật d :

- Sau khi các điều kiện trên được thỏa mãn, khóa giải mã d được tính bằng cách gọi hàm `generateDecryptKey(e, phi)`, hàm này sử dụng thuật toán Euclid mở rộng (`Utils.modMulInverse`) để tìm nghịch đảo modulo $d \equiv e^{-1} (\bmod \phi(n))$.

Phương thức này đảm bảo cặp khóa được tạo ra tuân thủ các yêu cầu kỹ thuật tối thiểu và đã tích hợp các biện pháp phòng vệ cần thiết.

Code hiệu thực

```
1 public static KeyPair generateRandomKeyPair(int bitLength) {
2     BigInteger p;
3     BigInteger q;
4     BigInteger e;
5     BigInteger phi;
```

```
6     BigInteger n;
7     do {
8         // Generate two large primes p and q
9         p = PrimeGenerator.generatePrime(bitLength / 2);
10        q = PrimeGenerator.generatePrime(bitLength / 2);
11
12        // Ensure p != q
13        while (p.equals(q)) {
14            q = PrimeGenerator.generatePrime(bitLength / 2);
15        }
16
17        // Compute n = p * q
18        n = p.multiply(q);
19
20        // Compute phi(n) = (p - 1)(q - 1)
21        phi = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));
22
23        // Generate e
24        e = BigInteger.valueOf(65537);
25
26    } while (!RSAPrimeVerifier.verifyPrimeForRSA(p, q) || !Utils.gcd(e, phi).
27        equals(BigInteger.ONE));
28
29    // Generate d
30    BigInteger d = generateDecryptKey(e, phi);
31
32    return new KeyPair(p, q, e, d, n);
}
```

4.4 Hiệu thực quá trình mã hóa và giải mã

4.4.1 Hàm mã hóa (encrypt)

Để thực hiện chức năng mã hóa cơ bản RSA theo công thức $c = m^e \text{ mod } n$ một cách hiệu quả và tuân thủ các ràng buộc kỹ thuật, hàm `encrypt(BigInteger message, BigInteger e, BigInteger n)` trong `RSAUtils.java` gọi trực tiếp hàm lũy thừa modulo đã tự hiện thực.

Trước khi thực hiện phép toán, hàm sẽ kiểm tra các điều kiện đầu vào để đảm bảo tính hợp lệ:

- Kiểm tra tính không âm: Nếu thông điệp `message` có dấu hiệu là số âm (`message.signum() == -1`), một ngoại lệ `IllegalArgumentException` sẽ được ném ra với thông báo "Message must be non-negative.".
- Kiểm tra kích thước: Nếu thông điệp `message` lớn hơn hoặc bằng modulus `n` (`message.compareTo(n) >= 0`), một ngoại lệ `IllegalArgumentException` sẽ được ném ra với thông báo "Message must be less than modulus n."

Quá trình mã hóa chính chỉ được thực hiện sau khi các kiểm tra này thành công.

Quá trình mã hóa chỉ đơn giản là gọi `modPow(base, exp, mod)` thuộc tệp `Utils.java` với các tham số:

- `base` là thông điệp m
- `exp` là khóa công khai e
- `mod` là modulus n

Code hiện thực

```
1 public BigInteger encrypt(BigInteger message, BigInteger e, BigInteger n) {
2     if (message.signum() == -1) {
3         throw new IllegalArgumentException("Message must be non-negative.");
4     }
5     if (message.compareTo(n) >= 0) {
```

```
6         throw new IllegalArgumentException("Message must be less than modulus n.");
7     }
8 }
9 }
```

4.4.2 Hàm giải mã (decrypt)

Để thực hiện chức năng giải mã cơ bản RSA theo công thức $m = c^d \bmod n$ một cách hiệu quả và tuân thủ các ràng buộc kỹ thuật, hàm decrypt(BigInteger cipher, BigInteger d, BigInteger n) trong RSAUtils.java gọi trực tiếp hàm lũy thừa modulo đã tự hiện thực.

Trước khi thực hiện phép toán, hàm sẽ kiểm tra các điều kiện đầu vào để đảm bảo tính hợp lệ:

- Kiểm tra tính không âm: Nếu bản mã cipher có dấu hiệu là số âm (cipher.signum() == -1), một ngoại lệ IllegalArgumentException sẽ được ném ra với thông báo "Ciphertext must be non-negative."
- Kiểm tra kích thước: Nếu bản mã cipher lớn hơn hoặc bằng modulus n (cipher.compareTo(n) >= 0), một ngoại lệ IllegalArgumentException sẽ được ném ra với thông báo "Ciphertext must be less than modulus n."

Quá trình giải mã chính chỉ được thực hiện sau khi các kiểm tra này thành công.

Quá trình giải mã chỉ đơn giản là gọi modPow(base, exp, mod) thuộc tệp Utils.java với các tham số:

- base là bản mã c
- exp là khóa bí mật d
- mod là modulus n

Code hiện thực

```
1 public BigInteger decrypt(BigInteger cipher, BigInteger d, BigInteger n) {
2     if (cipher.signum() == -1) {
3         throw new IllegalArgumentException("Ciphertext must be non-negative.");
4     }
5     if (cipher.compareTo(n) >= 0) {
6         throw new IllegalArgumentException("Ciphertext must be less than modulus
7             n.");
8     }
9 }
```

4.5 Các cải tiến và tối ưu hóa

Phần này mô tả chi tiết các cải tiến quan trọng đã được thực hiện để nâng cao bảo mật và hiệu suất của hệ thống RSA, cụ thể là OAEP và CRT.

4.5.1 Mã hóa an toàn với Padding OAEP (Optimal Asymmetric Encryption Padding)

OAEP là cơ chế padding tiêu chuẩn giúp RSA chống lại các tấn công dựa trên tính tất định (như Chosen Plaintext Attack) và tấn công dựa trên bản mã (Chosen Ciphertext Attack). Việc thực hiện padding sẽ giúp đảm bảo tính ngẫu nhiên của thông điệp, từ đó tăng cường bảo mật [1, pp. 307 – 308].

Quy trình Mã hóa OAEP từng bước (Step-by-Step):

Giả sử ta có thông điệp M , độ dài modulus là k bytes, và sử dụng hàm băm SHA-256 (độ dài output $h_{Len} = 32$ bytes).

Chuẩn bị tham số:

- Label L (thường là rỗng), hash L để có

$$l_{\text{Hash}} = \text{Hash}(L)$$

- Độ dài modulus: k bytes.

- Kiểm tra độ dài thông điệp:

$$\text{len}(M) \leq k - 2h_{\text{Len}} - 2$$

Tạo Data Block (DB):

- Tạo chuỗi padding PS gồm các byte 0x00 để lấp đầy.

- Kết hợp lại:

$$DB = l_{\text{Hash}} \parallel PS \parallel 0x01 \parallel M$$

với 0x01 là byte ngăn cách padding và thông điệp, và \parallel là phép nối chuỗi.

Tạo Seed ngẫu nhiên: sinh một chuỗi ngẫu nhiên dài h_{Len} bytes. Thông thường độ dài của seed sẽ bằng độ dài kết quả thuật toán hash trong MGF1, dựa vào đó để chúng ta có thể phân tách từng phần để giải mã.

Hàm hash MGF1: làm hàm sử dụng nhiều lần hash liên tiếp để từ một input đầu vào có thể tạo ra output với chiều dài byte cố định. Thông thường được hiện thực bằng cách Hash nhiều lần sau đó nối các chuỗi hash với nhau.

Masking (Che giấu) Data Block:

$$dbMask = \text{MGF1}(\text{seed}, \text{len}(DB)) \quad (1)$$

$$maskedDB = DB \oplus dbMask \quad (2)$$

Masking (Che giấu) Seed:

$$seedMask = \text{MGF1}(maskedDB, h_{\text{Len}}) \quad (3)$$

$$maskedSeed = seed \oplus seedMask \quad (4)$$

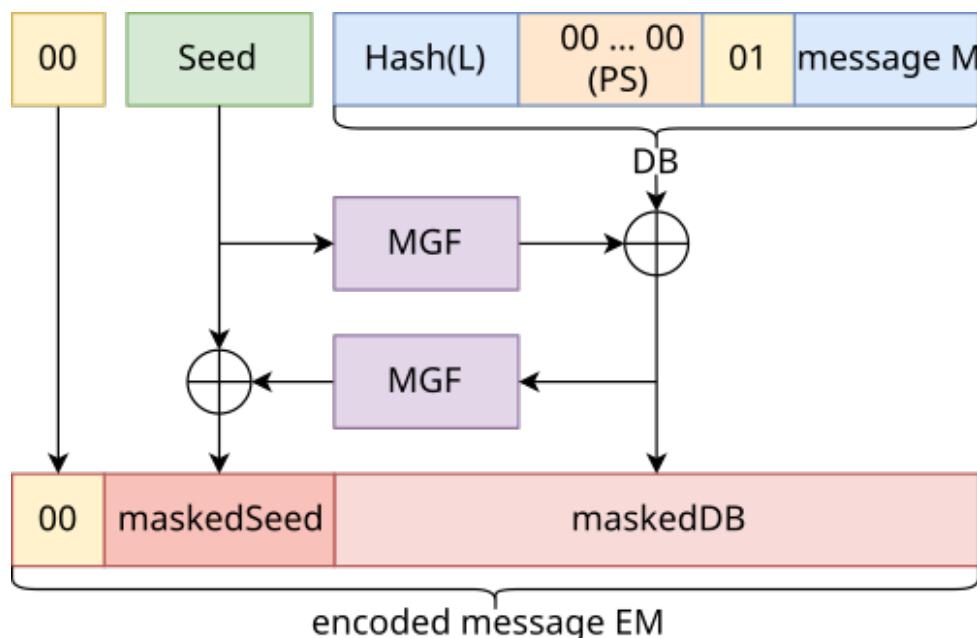
Tạo Encoded Message (EM):

$$EM = 0x00 \parallel maskedSeed \parallel maskedDB$$

với byte đầu tiên 0x00 đảm bảo $EM < n$ (modulus).

Mã hóa RSA:

$$C = EM^e \pmod{n} \quad (5)$$



Hình 2: OAEP Padding

4.5.2 Tối ưu hóa hiệu năng giải mã với CRT (Chinese Remainder Theorem)

Trong RSA tiêu chuẩn, quá trình giải mã yêu cầu thực hiện phép lũy thừa modulo trên số nguyên rất lớn: $M = C^d \pmod{n}$. Với độ dài khóa 2048-bit, thao tác này tiêu tốn tài nguyên tính toán đáng kể ($O(k^3)$).

Để khắc phục, hệ thống sử dụng **Định lý Số dư Trung Hoa (CRT)** để chuyển bài toán lớn modulo n thành hai bài toán con độc lập trên modulo p và q (với kích thước chỉ bằng một nửa). Phương pháp này giúp tăng tốc độ giải mã lên gấp 3-4 lần [1, pp. 300 – 301].

Giai đoạn chuẩn bị (Pre-computation) Các tham số hỗ trợ CRT được tính toán một lần duy nhất trong quá trình sinh khóa và lưu trữ bí mật cùng với khóa riêng (Private Key). Điều này giúp giảm tải tính toán trong thời gian thực (runtime).

Các giá trị cần tính trước bao gồm:

$$\begin{aligned} d_P &= d \pmod{p-1} && (\text{Số mũ rút gọn cho } p) \\ d_Q &= d \pmod{q-1} && (\text{Số mũ rút gọn cho } q) \\ q_{Inv} &= q^{-1} \pmod{p} && (\text{Hệ số nghịch đảo CRT}) \end{aligned}$$

Quy trình giải mã Thay vì tính trực tiếp trên n , quy trình giải mã sử dụng **Công thức Garner** (Garner's Formula) để tái tổ hợp kết quả từ hai thành phần p và q .

Bước 1: Giải mã thành phần (Partial Decryption)
Thực hiện hai phép lũy thừa trên các modulo nhỏ hơn:

$$m_1 = C^{d_P} \pmod{p} \quad (6)$$

$$m_2 = C^{d_Q} \pmod{q} \quad (7)$$

Nhận xét: Do p và q chỉ có độ dài bằng một nửa n , và số mũ d_P, d_Q cũng nhỏ hơn nhiều so với d , nên bước này cực kỳ nhanh.

Bước 2: Tái tổ hợp kết quả (Recombination)
Sử dụng hệ số q_{Inv} để kết hợp m_1 và m_2 thành bản rõ M ban đầu:

$$\begin{aligned} h &= ((m_1 - m_2) \cdot q_{Inv}) \pmod{p} \\ M &= m_2 + (h \cdot q) \end{aligned} \quad (8)$$

Phân tích Hiệu năng (Performance Analysis) Giả sử độ dài của n là k bit. Độ phức tạp của thuật toán lũy thừa modulo là $O(k^3)$.

- **RSA thường:** Tính toán trên k bit \Rightarrow Chi phí $\approx k^3$.
- **RSA-CRT:** Thực hiện 2 phép tính trên $k/2$ bit, về mặt lý thuyết sẽ nhanh hơn gần 4 lần:

$$2 \cdot \left(\frac{k}{2}\right)^3 = 2 \cdot \frac{k^3}{8} = \frac{k^3}{4}$$

4.5.3 Quy trình tạo khóa mạnh (Strong KeyPair Generation)

Để khắc phục các điểm yếu của việc sinh số nguyên tố ngẫu nhiên đơn thuần, nhóm áp dụng các tiêu chuẩn bảo mật nghiêm ngặt sau (được hiện thực trong `generateStrongKeyPair`):

1. Kiểm tra tính nguyên tố Miller-Rabin (Miller-Rabin Primality Test):

Hệ thống sử dụng số vòng lặp kiểm tra là $k = 40$. Theo lý thuyết xác suất, khả năng một hợp số vượt qua được bài kiểm tra này là $P(\text{error}) \leq 4^{-k}$. Với $k = 40$, xác suất lỗi là:

$$P(\text{error}) \leq 4^{-40} = 2^{-80} \approx 8.27 \times 10^{-25} \quad (9)$$

Con số này nhỏ đến mức có thể coi p, q là số nguyên tố tuyệt đối trong thực tế, đảm bảo nền tảng toán học vững chắc.

2. Chống tấn công phân tích Fermat (Anti-Fermat Factorization):

Nếu hai thừa số nguyên tố p và q quá gần nhau, kẻ tấn công có thể phân tích n dễ dàng bằng thuật toán Fermat. Hệ thống đảm bảo khoảng cách giữa p và q phải thỏa mãn ngưỡng an toàn:

$$\Delta = |p - q| > 2^{\frac{\text{bitLength}}{2} - 100} \quad (10)$$

Điều này ngăn chặn việc tìm ra p, q thông qua việc dò tìm xung quanh \sqrt{n} .

3. Ngăn chặn tấn công Pollard's $p - 1$ (Smoothness Check):

Thuật toán Pollard's $p - 1$ rất hiệu quả nếu $p - 1$ là số "trơn" (smooth) - tức chỉ chứa các ước số nhỏ. Hệ thống thực hiện phép chia thử (trial division) để loại bỏ các số p mà $p - 1$ chỉ bao gồm các ước số nhỏ hơn ngưỡng $Limit = 10^6$.

$$\text{Nếu } (p - 1) \text{ là } 10^6\text{-smooth} \implies \text{Loại bỏ và sinh lại } p. \quad (11)$$

Điều này đảm bảo $p - 1$ có ít nhất một ước số lớn hơn 10^6 , làm tăng độ phức tạp tính toán cho kẻ tấn công.

5 Kết luận

Chương này tổng kết công việc đã hoàn thành, đánh giá những ưu và nhược điểm của hệ thống RSA tự hiện thực, đồng thời đề xuất các hướng phát triển trong tương lai.

5.1 Tóm tắt kết quả đạt được

Bài tập lớn đã hoàn thành xuất sắc yêu cầu về việc tự hiện thực hệ mã RSA, đồng thời nâng cấp đáng kể về cả hiệu năng lẫn độ an toàn so với phiên bản cơ sở.

Những thành tựu chính:

- Hoàn thành Yêu cầu Cơ bản:** Đã tự hiện thực tất cả các thuật toán nền tảng cần thiết, bao gồm Thuật toán Euclid, Euclid Mở rộng, Bình phương và Nhân Modulo (ModPow), và Thuật toán Miller-Rabin.
- Nâng cấp Bảo mật (OAEP):** Đã tích hợp thành công OAEP (Optimal Asymmetric Encryption Padding). Sự bổ sung này loại bỏ lỗi hổng nghiêm trọng của RSA tròn (Textbook RSA), đảm bảo tính ngẫu nhiên của bản mã và cung cấp khả năng chống lại tấn công Chosen-Ciphertext Attack (CCA).
- Nâng cấp Hiệu năng (CRT):** Đã hiện thực Giải mã bằng Định lý Số dư Trung Hoa (CRT). Việc này cải thiện tốc độ giải mã đáng kể (lên đến 4 lần theo lý thuyết), được chứng minh qua demo so sánh hiệu suất.
- Chất lượng Khóa Cao:** Hàm generateStrongKeyPair được triển khai với độ chắc chắn cao hơn (certainty = 40) và kiểm tra khoảng cách tối thiểu giữa P và Q để tăng cường khả năng chống lại các tấn công phân tích thừa số đặc biệt.
- Xử lý dữ liệu thực tế:** Đã thử nghiệm thành công việc chuyển đổi chuỗi ký tự và chuỗi byte sang đối tượng BigInteger (sử dụng hàm khởi tạo BigInteger(int signum, byte[] magnitude)) và thực hiện mã hóa/giải mã chính xác cho các thông điệp có kích thước nhỏ hơn modulus n .
- Giải pháp Tối ưu:** Đã cung cấp phương thức decryptOAEP_CRT là giải pháp đề xuất, kết hợp tốc độ của CRT và tính bảo mật của OAEP.

Những điều chưa hoàn thành:

- Thiếu mã hóa phân đoạn (Block Chaining):** Hệ thống hiện tại chưa thể xử lý việc mã hóa và giải mã các chuỗi ký tự hoặc chuỗi byte có kích thước lớn hơn modulus n , do thiếu cơ chế chia thông điệp thành các khối nhỏ (segmentation) và mã hóa từng khối độc lập.

5.2 Ưu điểm và nhược điểm

Ưu điểm

- Mật mã An toàn Tuyệt đối:** Nhờ việc tích hợp OAEP, hệ thống đã chuyển từ RSA học thuật (không an toàn) sang một triển khai an toàn hơn, phù hợp với các tiêu chuẩn bảo mật hiện đại.
- Tốc độ Giải mã Vượt trội:** CRT làm tăng tốc độ giải mã đáng kể, đặc biệt quan trọng khi sử dụng các khóa lớn (2048 bits trở lên).
- Khóa Chắc chắn:** Các kiểm tra bổ sung trong RSAPrimeVerifier.java và generateStrongKeyPair đảm bảo các số nguyên tố được chọn có chất lượng cao hơn, chống lại các tấn công dựa trên đặc tính của số nguyên tố (Fermat, Pollard $P - 1$).

Nhược điểm

- Hiệu năng Java:** Mặc dù đã có CRT, hiệu năng tổng thể của các thuật toán tự hiện thực bằng Java vẫn kém hơn đáng kể so với các thư viện mật mã đã được tối ưu hóa ở cấp độ thấp (native code).
- Phụ thuộc Mã băm (Hash):** Tính an toàn của OAEP phụ thuộc trực tiếp vào tính an toàn của hàm băm được sử dụng (SHA-256).
- Hạn chế Dữ liệu Đầu vào:** Hệ thống vẫn chưa thể xử lý trực tiếp thông điệp dưới dạng văn bản hoặc tệp tin, cần thêm các bước xử lý dữ liệu (data serialization) ở bên ngoài.

5.3 Hướng phát triển

Để phát triển hệ mã RSA này thành một công cụ sử dụng được trong môi trường thực tế, nhóm đề xuất các hướng phát triển sau:

- Tối ưu hóa Hiệu năng Chuyên sâu:
 - Nghiên cứu và triển khai các thuật toán tối ưu hóa phép nhân số lớn hơn (ví dụ: Thuật toán Karatsuba) để cải thiện tốc độ của các phép toán nhân BigInteger, vốn là nền tảng của modPow.
 - Tích hợp thuật toán Montgomery Multiplication để tăng tốc độ cho toàn bộ các phép nhân modulo bên trong hàm modPow.
- Triển khai Mã hóa Phân đoạn (Block Chaining/Segmentation):
 - Phát triển module để chia các chuỗi ký tự hoặc tệp tin lớn thành các khối dữ liệu nhỏ hơn kích thước mã hóa tối đa, mã hóa từng khối độc lập, và ghép chúng lại thành bản mã cuối cùng.
- Tăng cường Khả năng Chống Tấn công Kênh bên:
 - Nghiên cứu và triển khai các biến thể của thuật toán lũy thừa modulo, ví dụ như Montgomery Ladder, để đảm bảo thời gian thực thi là hằng số, từ đó chống lại các tấn công timing attack tinh vi.
- Hỗ trợ Tùy biến Thuật toán Băm:
 - Mở rộng tính linh hoạt của OAEP để cho phép người dùng chọn thuật toán băm khác ngoài SHA-256 (ví dụ: SHA-512) thông qua tham số cấu hình.

Tài liệu

- [1] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Pearson, 2017.
- [2] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC press, 1996.
- [3] K. H. Rosen, *Discrete mathematics and its applications*. Maidenhead, England: McGraw Hill Higher Education, Jan. 2018.
- [4] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems.” <https://people.csail.mit.edu/rivest/Rsapaper.pdf>, 1978. [Accessed 30-11-2025].
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, London, England: MIT Press, 3 ed., July 2009.