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

Mật mã và an ninh mạng (CO3069)

RSA ENCRYPTION IMPLEMENTATION

Học kỳ 232

GVHD: Nguyễn Cao Đạt
SV: Trần Mậu Thật 2112342
Huỳnh Tấn Đạt 2120015
Trần Anh Khoa 2111541
Võ Phạm Long Huy 1812438

TP. HỒ CHÍ MINH, THÁNG 4/2024

Mục lục

1 Giới thiệu

1.1 Đặt vấn đề

Ngày nay, vấn đề truyền tải thông tin qua mạng internet đã trở thành một phần không thể thiếu trong cuộc sống hiện đại, chúng ta có thể giao tiếp, truyền dữ liệu với khoảng cách xa, và tốc độ nhanh chóng. Có thể lưu trữ dữ liệu và truy cập dữ liệu với tốc độ cao và dung lượng lớn. Và dĩ nhiên, mạng internet là có thể truy cập vào bởi tất cả mọi người. Chính vì vậy, vấn đề bảo mật, riêng tư và toàn vẹn dữ liệu đã trở thành mối bận tâm hàng đầu của những cá nhân, tổ chức tham gia vào trong mạng. Từ đó, các giải thuật mã hoá dữ liệu đã được hình thành, bảo đảm tính bí mật, toàn vẹn và xác thực của thông tin. Tính bí mật đảm bảo rằng thông tin chỉ được truy cập bởi những người được ủy quyền và không bị lộ ra bên ngoài. Tính toàn vẹn đảm bảo rằng dữ liệu không bị sửa đổi trái phép trong quá trình truyền tải, đảm bảo rằng thông tin nhận được vẫn giữ nguyên giá trị và ý nghĩa ban đầu. Tính xác thực đảm bảo rằng người gửi và người nhận thông tin có thể xác định và xác minh danh tính của nhau. Đó là các chức năng cơ bản của việc mã hoá dữ liệu, bên cạnh đó các giải thuật mã hoá yêu cầu dễ mã hoá và dễ giải mã, nhưng lại khó để tấn công và bị giải mã bởi bên thứ ba nếu không có khoá. Ngày nay, có rất nhiều loại mã hoá dữ liệu với các mục tiêu, yêu cầu khác nhau, được phân thành các loại chủ yếu như sau:

- Mã hoá đối xứng (Symmetric encryption): Cả mã hoá và giải mã đều sử dụng cùng một khóa. Một số ví dụ phổ biến của mã hoá đối xứng bao gồm AES (Advanced Encryption Standard), DES (Data Encryption Standard), và 3DES (Triple DES).
- Mã hoá không đối xứng (Asymmetric encryption): Sử dụng cặp khóa: một khóa công khai và một khóa riêng tư. Khóa công khai được chia sẻ với mọi người và được sử dụng để mã hoá dữ liệu, trong khi khóa riêng tư được giữ bí mật và chỉ dùng để giải mã dữ liệu đã được mã hoá. Ví dụ nổi tiếng nhất là RSA (Rivest-Shamir-Adleman).
- Mã hoá băm (Hashing): Không thể giải mã ngược lại. Thường được sử dụng để tạo ra các giá trị băm duy nhất từ dữ liệu đầu vào, đại diện cho dữ liệu một cách duy nhất. Ví dụ: MD5, SHA-1, SHA-256. Mã hoá dựa trên điểm (Elliptic Curve Cryptography - ECC): Một loại mã hoá không đối xứng sử dụng các phép toán trên các điểm trên đường cong elip. ECC thường sử dụng khóa ngắn hơn so với RSA nhưng vẫn cung cấp mức độ bảo mật tương đương.
- Mã hoá đàn hồi (Homomorphic encryption): Cho phép các phép toán được thực hiện trực tiếp trên dữ liệu đã được mã hoá mà không cần giải mã nó. Điều này có ý nghĩa trong các ứng dụng mà quyền riêng tư cần được bảo vệ ngay cả khi dữ liệu được xử lý.

1.2 Phương hướng

Trong đó, mã hoá RSA là một trong những phương pháp mã hoá khóa công khai phổ biến nhất, được phát triển vào năm 1977 bởi Ron Rivest, Adi Shamir và Leonard Adleman.[**1**] RSA sử dụng cặp khóa công khai và khóa riêng tư để mã hoá và giải mã dữ liệu. Khóa công khai được chia sẻ với mọi người và được sử dụng để mã hoá dữ liệu, trong khi khóa riêng tư được giữ bí mật và chỉ dùng để giải mã dữ liệu đã được mã hoá. Phương pháp này cung cấp một cách thức an toàn và đáng tin cậy để truyền tải thông tin trên mạng và xác thực danh tính của người gửi. RSA được sử dụng rộng rãi trong các ứng dụng bảo mật, bao gồm truyền thông an toàn qua mạng, chữ ký điện tử và xác thực người dùng. Trong khuôn khổ bài tập lớn này, sẽ hiện thực mô phỏng mã hoá RSA trên dữ liệu, tìm cách để có thể tối ưu hơn, ngăn ngừa sự tấn công đơn giản với mã hoá RSA bằng cách tối ưu các số nguyên tố được chọn, để phòng ngừa sự phân tích thành các thừa số nguyên tố của số N thành công với các phương pháp toán học mạnh mẽ (sẽ được trình bày cụ thể hơn ở chương cơ sở lí thuyết).

Cụ thể, bài tập lớn này bao gồm các phần:

- Cơ sở lí thuyết: Bao gồm về khái niệm, nội dung, công thức, thuật toán của các phương pháp đã được nêu
- Phân tích và thiết kế: Bao gồm lên ý tưởng cho việc hiện thực, sử dụng các cơ sở lí thuyết để hiện thực

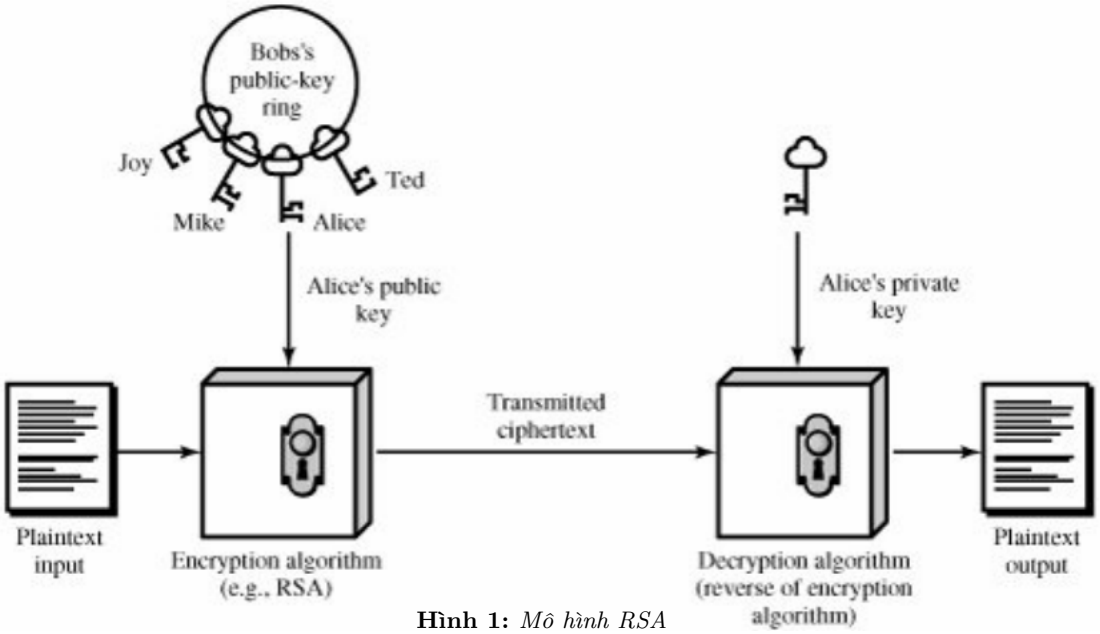
- Hiện thực và đánh giá: Sau khi hiện thực mô phỏng RSA, cải tiến, thực hiện so sánh và đánh giá mức độ hiệu quả, ưu nhược điểm
- Kết luận: Đánh giá kết quả đạt được, những vấn đề, khía cạnh tiếp tục nghiên cứu trong tương lai

2 Cơ sở lí thuyết

2.1 RSA Encryption

2.1.1 Giới thiệu RSA

RSA là một trong những thuật toán phổ biến nhất cho mật mã khóa công khai. Nó được công khai vào 1977 bởi ba nhà toán học Ronald Rivest, Adi Shamir và Leonard Adleman, theo tên viết tắt của họ. RSA dựa trên việc lựa chọn hai số nguyên tố lớn và tính toán một số nguyên dương khác từ chúng, được sử dụng làm các thành phần của cặp khóa. Đây là thuật toán đầu tiên phù hợp với việc tạo ra chữ ký điện tử đồng thời với việc mã hóa. Nó đánh dấu một sự tiến bộ vượt bậc của lĩnh vực mật mã học trong việc sử dụng khóa công cộng. RSA đang được sử dụng phổ biến trong thương mại điện tử và được cho là đảm bảo an toàn với điều kiện độ dài khóa đủ lớn.
Trong mật mã RSA, cả khóa chung và khóa riêng đều có thể mã hóa tin nhắn. Khóa ngược lại với khóa dùng để mã hóa tin nhắn sẽ được dùng để giải mã nó. Thuộc tính này là một lý do tại sao RSA đã trở thành thuật toán bất đối xứng được sử dụng rộng rãi nhất: Nó cung cấp một phương pháp để đảm bảo tính bảo mật, tính toàn vẹn, tính xác thực và tính không thoái thác của truyền thông điện tử và lưu trữ dữ liệu.



Hình 1: *Mô hình RSA*

Giải thuật RSA hoạt động như một thuật toán mã hóa khối, trong đó cả văn bản gốc và văn bản đã mã hóa đều là các số nguyên trong khoảng từ 0 đến n - 1, với n là một số nguyên lớn thường khoảng 1024 bit hoặc 309 chữ số thập phân. Điều này có nghĩa là n nhỏ hơn 2^{1024}.
Trong quá trình mã hóa RSA, một tin nhắn được mã hóa bằng cách sử dụng khóa công khai của người nhận và chỉ có thể được giải mã bằng cách sử dụng khóa riêng tư tương ứng. Sự an toàn của RSA phụ thuộc vào khó khăn của việc phân tích các số nguyên tố lớn thành các thừa số nguyên tố của chúng, một vấn đề trở nên không thể thực hiện tính toán với các số nguyên tố đủ lớn.

2.1.2 Tạo khóa

Mỗi người sẽ tạo cho mình một cặp khóa bao gồm khóa bí mật và khóa công khai theo các bước sau:

- **Bước 1:** Chọn 2 số nguyên tố lớn p và q với $p \neq q$, lựa chọn ngẫu nhiên và độc lập.
- **Bước 2:** Tính $n = pq$

- **Bước 3:** Tính giá trị hàm số Euler $\phi(n) = (p-1)(q-1)$
- **Bước 4:** Chọn một số tự nhiên e sao cho $1 < e < \phi(n)$ và là số nguyên tố cùng nhau với $\phi(n)$, $gcd(e, \phi(n)) = 1$
- **Bước 5:** Tính d sao cho d.e = 1 (mod $\phi(n)$)

Lưu ý:

- Các số nguyên tố thường được chọn bằng phương pháp thử xác suất.
- Các bước 4 và 5 có thể được thực hiện bằng giải thuật Euclid mở rộng
- Bước 5 có thể viết cách khác: Tìm số tự nhiên x, sao cho $d = \frac{x(p-1)(q-1)+1}{e}$ cũng là số tự nhiên.Khi đó sử dụng giá trị d mod (p-1)(q-1)
- Khóa công khai:
  - PU={e,n}
  - n: module
  - e : Số mũ công khai
- Khóa bí mật:
  - PR={d,n}
  - n : module
  - d: Số mũ bí mật

2.1.3 Mã hóa

Giải sử A muốn gửi một thông điệp M cho B, A sẽ nhận khóa công khai từ B và tiến hành mã hóa M thành C. (Chú ý rằng thông điệp M phải nhỏ hơn n)

- Lấy khóa công khai của người nhận là: PU={e,n}
- Tính toán: $C = M^e \mod n, 0 \leq M \leq n$

2.1.4 Giải mã

Sau khi nhận thông điệp đã được mã hóa bằng khóa công khai C từ A, B tiến hành dùng khóa riêng của mình để giải mã thông điệm C thành M.

- Dùng khóa riêng: PR={d,n}
- Tính toán: $M = C^d \mod n$

Quá trình giải mã hoạt động được là vì:

- Dựa trên định lý Euler: $a^{\phi(n)}$ mod n = 1 khi gcd(a,n) = 1
- Trong RSA ta có:
  - n = p.q
  - $\phi(n) = (p-1)(q-1)$
  - e.d = 1 mod $\phi(n)$ nên e.d = 1 + k.$\phi(n)$
- Vậy: $C^d = Med = M^{(1+k\phi(n))} = M^1.(M^{\phi(n)})^k = M^1 =$ M mod n

2.1.5 Ví dụ

1. Chọn các số nguyên tố: p=17 & q=11
2. Tính toán n = pq =17 x 11=187
3. Tính toán $\phi(n)$=(p–1)(q-1)=16 x 10=160
4. Chọn e sao cho gcd(e,160)=1: lấy e=7
5. Xác định d: d.e=1 mod 160 và d < 160 Tìm ra được d=23 vì 23x7=161=1x160+1
6. Công bố khóa công khai: PU=7,187
7. Giữ bí mật khóa riêng: PR=23,187
8. Thông điệp M = 88 (88)
9. Mã hóa:
   C = 887 mod 187 = 11
10. Giải mã:
    M = 1123 mod 187 = 88

2.1.6 Thuật toán Euclid

Giải thuật Euclid dùng để tính ước chung lớn nhất (gcd) của hai số tự nhiên a và b. Ước chung lớn nhất g là số lớn nhất chia được bởi cả a và b mà không để lại số dư và được ký hiệu là gcd(a, b)
Mã giả:

```
function gcd(a, b)
while b != 0
    t:= b
    b:= a mod b
    a:= t
return a
```

Phương pháp:

- Cho 2 số nguyên a và b
- Chia a cho b và lưu lại phần dư r
- Nếu r = 0, thì b chính là ước chung lớn nhất và kết thúc thuật toán
- Gán a = b và b = r, sau đó quay lại bước 2

2.1.7 Thuật toán Euclid mở rộng

Giải thuật Euclid mở rộng được sử dụng để giải một phương trình vô định nguyên (còn được gọi là phương trình Đi-ô-phăng) có dạng: ax + by = c

Trong đó a,b,c là các hệ số nguyên, x, y là các ẩn nhận giá trị nguyên.Điều kiện cần và đủ để phương trình này có nghiệm (nguyên) là gcd(a,b) là ước của c.Khẳng định này dựa trên một mệnh đề sau:

Nếu d = gcd(a,b) thì tồn tại số nguyên x, y sao cho ax + by = d

Giải thuật Euclid mở rộng kết hợp quá trình tìm gcd(a, b) trong thuật toán Euclid với việc tìm một cặp số x, y thoả mãn phương trình Đi-ô-phăng. Giả sử cho hai số tự nhiên a, b, ngoài ra a>b>0.Đặt r_0 = a, r_1 = b, chia r_0 cho r_1 được số dư r_2 và thương số nguyên q_1.Nếu $r_2 = 0$ thì dừng, nếu r_2 khác 0 chia r_1 cho r_2 được r_3,...Vì dãy r_i là giảm nên sau hữu hạn bước ta được số dư r_{m+2}
$r_0 = q_1 * r_1 + r_2, 0 < r_2 < r_1;$
$r_1 = q_2 * r_2 + r_3, 0 < r_3 < r_2;$
...;
$rm - 1 = q_m * r_m + rm + 1, 0 < rm + 1 < r_m$
$r_m = q_{m+1} * r_{m+1}$ trong đó số dư cuối cùng khác 0 là $r_{m+1} = d$. Bài toán đặt ra là tìm x, y sao cho

$a * x + b * y = r_{m+1} = d$

Để làm điều này, ta tìm x, y theo công thức truy hồi, nghĩa là sẽ tìm x_i và y_i sao cho:
$a * x_i + b * y_i = r_i vii = 0, 1, ...$

Ta có
$a * 1 + b * 0 = a = r_0$ và $a * 0 + b * 1 = b = r_1$ nghĩa là $x_0 = 1, x_1 = 0$ và $y_0 = 0, y_1 = 1$ (1)

Tổng quát, giả sử có $a * x_i + b * y_i = r_i$ với i = 0,1,...
$a * x_{i+1} + b * y_{i+1} = r_{i+1}$ với i = 0,1,...

Khi đó từ: $r_i = q_{i+1} * r_{i+1} + r_{i+2}$ suy ra
$r_i - q_{i+1} * r_{i+1} = r_{i+2}$
$(a * x_i + b * y_i) - q_{i+1} * (a * x_{i+1} + b * y_{i+1}) = r_{i+2}$
$a * (x_i - q_{i+1} * x_{i+1}) + b * (y_i - q_{i+1} * y_{i+1}) = r_{i+2}$

Từ đó chọn
$x_{i+2} = x_i - q_{i+1} * x_{i+1}$ (2)
$y_{i+2} = y_i - q_{i+1} * y_{i+1}$ (3)

Khi i = m - 1 ta có được x_{m+1} và y_{m+1}.Các công thức (1), (2), (3) là công thức truy hồi để tính x, y
Thuật toán:

```
    find x, y such that ax+by=GCD(a,b)
    With input a,b called from mod_Inverse, a <- e, b <- n => a and b are co-prime => GCD
    (a,b)=1
    We have ax+by = 1 => (ax+by) mod (b) = 1 => ax mod (b) = 1 => x is the Modular
    multiplicative inverse of number (a) modulo (b)
    Return x

```

2.2 Miller_Rabin

2.2.1 Giới thiệu

Trong ngành mật mã học, các thuật toán mã hoá công khai đều cần sử dụng các số nguyên tố lên tới hơn 512 bits. Có một số phương pháp để sinh số nguyên tố và các phương pháp đó đều dựa theo thuật toán kiểm tra tính nguyên tố của một số nguyên.

Các thuật toán kiểm tra số nguyên tố thường chia làm 2 loại:

- Thuật toán xác định chính xác: Có thể xác định chính xác ngay trong thuật toán nhưng tốn nhiều thời gian. (Độ phức tạp thuật toán lớn).
- Thuật toán xác định 1 phần: Có thể xác định 1 phần xem nó có thể là số nguyên tố hay không, không đúng hoàn toàn, nhưng tốn ít thời gian. (Độ phức tạp thuật toán nhỏ).

Thuật toán Miller-Rabin thuộc loại thuật toán xác định 1 phần.

2.2.2 Thuật toán

Sau đây là pseudo-code cho thuật toán Miller-Rabin:

```
TEST(p)
    Find k, q so that k> 0, q is odd and p = 2^k*q + 1.
    Choose a random number a in the range of [2, p - 1]
    If power(a,q) mod p = 1 Then
        return "p can be a prime number.";
    For j= 0 to k-1 do
        If power(a^(2^j)*q)mod p = -1 Then
            return "p can be a prime number.";
    return "p isn't a prime number.";
```

2.2.3 Ví dụ

- Ví dụ 1: Số 29
  $29 = 2^2 * 7 + 1 => k = 2, q = 7$
  - Nếu chọn $a = 10 : 10^7 mod 29 = 17$ do đó ta sẽ tiếp tục tính $(10^7)^2 mod 29 = 28$ thủ tục kiểm tra sẽ trả về "có thể là số nguyên tố".

  - Nếu chọn a = 2 : $2^7 mod 29 = 12$ do đó ta sẽ tiếp tục tính $(2^7)^2 mod 29 = 28$ thủ tục cũng sẽ trả về "có thể là số nguyên tố".
  - Vì vậy, nếu chỉ thử một vài giá trị a, ta chưa thể kết luận gì về tính nguyên tố của p. Tuy nhiên nếu thử hết các giá trị a từ 2 đến 28 ta đều nhận được kết quả "có thể là số nguyên tố". Vì vậy có thể chắc chắn rằng 29 là số nguyên tố.

- Ví dụ 2: Số 221
  $221 = 2^2 * 55 + 1 => k = 2, q = 55$
  - Nếu chọn $a = 5 : 5^{55} mod 221 = 112$ do đó ta sẽ tiếp tục tính $(5^{55})^2 mod 221 = 168$ thủ tục kiểm tra sẽ trả về "Không thể là số nguyên tố". Đúng vì 221 = 13*17.
  - Nếu chọn a = 21 : $21^{55} mod 221 = 200$ do đó ta sẽ tiếp tục tính $(21^{55})^2 mod 221 = 220$ thủ tục cũng sẽ trả về "có thể là số nguyên tố". Nghĩa là trong 1 số trường hợp thì thuật không thể xác định tính nguyên tố của số.

2.3 Miller_Rabin lặp

2.3.1 Cách làm

Người ta đã tính được xác suất để trong trường hợp p là hợp số, thuật toán Miller Rabin đưa ra khẳng định, "p không phải là số nguyên tố" là 75%. Trong 25% còn lại, Miller-Rabin không xác định được p nguyên tố hay hợp số. Do đó nếu chúng ta áp dụng thuật toán t lần (mỗi lần với các giá trị a khác nhau) thì xác suất không xác định (trong cả t lần) là 0.25^t . Với t bằng 10, xác suất trên là rất bé, nhỏ hơn 0.000001.

Tóm lại người ta kiểm tra tính nguyên tố của 1 số như sau:

- Thực hiện thuật toán 10 lần cho 10 số a khác nhau.
- Nếu cả 10 lần thuật toán cho ra kết quả "có thể là số nguyên tố", thì ta khẳng định p là số nguyên tố.
- Chỉ cần một lần thuật toán cho ra kết quả "không phải là số nguyên tố", thì ta khẳng định p là hợp số.

2.3.2 Ví dụ

- Ví dụ 1: số 41

a	$a^q mod p$	$a^{2q} mod p$	$a^{4q} mod p$
7	38	9	**40**
8	98	**40**	
9	9	**40**	
12	3	9	**40**
13	38	9	**40**
16	**1**		
24	14	32	**40**
25	**40**		
31	**40**		
37	**1**		

  Chúng ta có thể ngầm xác định 41 là số nguyên tố.

- Ví dụ 2: số 133

a	$a^q mod p$	$a^{2q} mod p$
11	**1**	
17	**83**	**106**

  Từ lần 2 đã xác định được rằng không phải số nguyên tố. Vì thế chúng ta kết luận đây không thể là số nguyên tố. (133=7*19)

2.4 RSA attack

2.4.1 Pollard's p - 1 algorithm

Nếu một trong các thừa số nguyên tố của n có tính chất đặc biệt thì đó là đôi khi dễ dàng phân tích n hơn. Ví dụ: nếu p|n và p 1 có thừa số nguyên tố "nhỏ", phương pháp sau đây sẽ có hiệu quả.

Chọn 1 số nguyên B. Cho 1 số nguyên n, Thuật toán này tìm số nguyên tố p sao cho p|n và p - 1 có thừa nguyên tố <= B.

Mã giả:

```
a = 2
for j = 2 to B
do a = a
j mod n
d = gcd(a-1, n)
if 1 < d < n
then return(d)
else return("failure")
```

Phương pháp:

- Giả sử p|n và q $\leq$ b B với mọi luỹ thừa nguyên tố q|p-1. Khi đó phải xẩy ra trường hợp p-1|B!.
- Sau khi kết thúc loop 1 ta có: $a \equiv 2^{B!} mod n$.
- Với p|n có: $a \equiv 2^{B!} mod p$
- với p-1|B! ta có: $a \equiv 1 mod p$=> p|a-1.

Ngoài ra, p|n => p|d; trong đó d = gcd(a - 1, n). d sẽ là ước số không tầm thường của n (trừ khi a = 1).

Cách chọn B:
B nhỏ thì thuật toán nhanh nhưng ít cơ hội thành công.
B lớn thì cơ hội thành công cao nhưng thuật toán chậm.
=> Chọn B sao cho đúng mục đích sử dụng.

Cách chặn tấn công p-1:
Chúng ta sẽ cần p-1 cũng có ước số nguyên tố lớn.
Ví dụ: p 100 chữ số thì chọn p0 cỡ 40 chữ số. Sau đó áp dụng kp0+1(k cỡ 60 chữ số) và dùng Miller-Rabin để kiểm tra tính nguyên tố. => Có được P. Áp dụng tương tự sẽ có được q. => Chặn được tấn công p-1.

2.4.2 Williams's p + 1 algorithm

Giải thuật William's P+1 là một phương pháp trong lĩnh vực mã hóa khóa công khai, được sử dụng để tăng cường tính an toàn của thuật toán RSA, một trong những thuật toán phổ biến nhất trong việc mã hóa và giải mã thông tin.

Trong thuật toán RSA, ta cần tìm số nguyên tố p và q lớn, rồi tính tích p và q để tạo ra khóa công khai và khóa bí mật. Vấn đề là, việc tìm các số nguyên tố lớn có thể mất rất nhiều thời gian, đặc biệt khi chúng ta muốn chúng có độ dài đủ lớn để đảm bảo tính an toàn của hệ thống mã hóa.

Giải thuật William's P+1 cố gắng tìm các số nguyên tố p bằng cách sử dụng một phương pháp cải tiến của thuật toán tìm ước số chung lớn nhất (GCD). Giải thuật này sử dụng ý tưởng về việc kiểm tra các điều kiện có thể tạo ra các số nguyên tố lớn từ các số nguyên tố nhỏ hơn. Đặc biệt, nó kiểm tra nếu (p-1) là một bội của một số nguyên tố nhỏ hơn (gọi là "n-1 smooth"). Nếu điều này xảy ra, nó tăng cường p bằng cách thêm một số nguyên vào p, và lặp lại quá trình này cho đến khi không thể tăng cường p nữa.

Mặc dù giải thuật này không đảm bảo rằng bạn sẽ tìm thấy số nguyên tố p ngay lập tức, nhưng nó có thể giúp giảm thời gian tìm kiếm so với việc sử dụng phương pháp tìm kiếm nguyên thủy.

Mã giả:

```
function William_P1(n, B):
a = 2
for i from 2 to B:
    a = (a ** i) % n
    gcd_value = gcd(a - 1, n)
    if 1 < gcd_value < n:
        return gcd_value
return FAILURE
```

Trong đó:

- n là số nguyên dương cần kiểm tra để xem có phải là số nguyên tố hay không.
- B là giới hạn trên của số mũ trong phép tính aî mod n.
- Hàm gcd(a, b) là hàm tính ước số chung lớn nhất của a và b.
- Hàm William_P1(n, B) thực hiện thuật toán William's P+1 để tìm ước số chung lớn nhất của (a-1) và n.
- Nếu tìm thấy một ước số chung lớn nhất trong khoảng từ 2 đến B, nó sẽ trả về giá trị đó. Nếu không tìm thấy, nó sẽ trả về FAILURE.

2.4.3 RSA cycling attack

Cuộc tấn công lặp là một phương pháp được sử dụng để khai thác tính tuần hoàn của quá trình mã hóa RSA. Nó liên quan đến việc lặp đi lặp lại việc mã hóa và giải mã một thông điệp cho đến khi phát hiện ra một chu kỳ. Cuộc tấn công này có thể xảy ra khi số mũ được sử dụng cho việc mã hóa và giải mã không nguyên tố cùng nhau với mô đun, điều này có thể dẫn đến một mẫu chu kỳ trong văn bản mã hóa. Bằng cách phát hiện chu kỳ này, một kẻ tấn công có thể khôi phục lại văn bản gốc mà không cần phải phân tích thừa số nguyên tố của mô đun.

Cuộc tấn công tận dụng thực tế rằng trong mã hóa RSA, $c = m^e$ (mod n) và việc giải mã được thực hiện bằng cách $m = c^d$ (mod n)

Để thực hiện cuộc tấn công lặp, kẻ tấn công mã hóa một thông điệp văn bản được chọn nhiều lần bằng cách sử dụng khóa công khai (số mũ mã hóa e và mô đun n), sau đó giải mã văn bản mã hóa thu được bằng cách sử dụng khóa riêng (số mũ giải mã d và mô đun n). Nếu phát hiện ra một chu kỳ trong các thông điệp được giải mã, nó ngụ ý rằng số mũ mã hóa e và số mũ giải mã d chia sẻ một yếu tố chung với $\phi(n)$, Yếu tố chung này có thể được sử dụng để tính toán khóa riêng và cuối cùng là khôi phục lại thông điệp văn bản gốc.
Mã giả:

```
Input: n, starting values seed, start_0,...;
parameters par_0, par_1,..., a boundary B
Output: "success", p,q; or "fail"
set x_0 = Enc(x_i,.) mod n;
set start =|log n|;
repeat
set x_{i+1} = Enc(x_i,.) mod n;
until i >= start;
repeat
set x_{i+1} = Enc(x_i,.) mod n;
set test = gcd(x_{i+1} - x_{start}, n);
until test != 1 or i > B;
If test != 1 and test != n then Output("success",test, n:test);
else Output("fail");
```

Trong đó:
Enc(.,.) là hàm tổng quát RSA
Để ngăn chặn cuộc tấn công lặp:

- Sử dụng Kích thước Mô đun Lớn: Sử dụng kích thước mô đun lớn giúp làm cho việc phát hiện chu kỳ trở nên khó khăn hơn. Kích thước mô đun lớn hơn sẽ tăng cường độ an toàn của hệ thống mã hóa RSA bằng cách làm cho việc tìm các chu kỳ trở nên khó khăn hơn đối với kẻ tấn công.
- Lựa chọn số mũ mã hóa e và số mũ giải mã d là số nguyên tố cùng nhau.

2.5 Strong Prime Number

Chúng ta đã tìm hiểu về các phương thức cơ bản và hiệu quả để tấn công thông qua khoá công khai N, bằng cách dùng các giải thuật hiệu quả như Pollard's p-1 và William's p+1 và cả RSA cycling attack, chúng ta sẽ phân tích và áp dụng những phương pháp dùng để ngăn chặn 3 phương pháp tấn công trên và tạo được bộ khoá tối ưu hơn có thể.
Một số nguyên tố mạnh để cấu tạo nên hợp số N dùng làm khoá công khai, module trong hệ mã hoá RSA được đánh giá là số nguyên tố giúp cho số N khó có thể bị phân rã một cách dễ dàng. Để đáp ứng điều đó, số p được gọi là số nguyên tố mạnh thì phải thoả những yêu cầu thiết kế sau:

- p phải là số nguyên tố đủ lớn để việc phân rã N trở nên tốn kém thời gian và tài nguyên, gần như là điều không khả thi
- p - 1 phải được phân tích thành các thừa số nguyên tố mạnh hay nói cách khác p = a1*q1+1 trong đó (q1 lại là một số nguyên tố lớn) ngăn chặn Pollard p -1
- q1 - 1 phải được phân tích thành các thừa số nguyên tố mạnh hơn nữa, hay nói cách khác chính bản thân q1 = a2*q2+1 (trong đó q2 là một số nguyên tố lớn) ngăn chặn Cycling attack
- p + 1 phải được phân tích thành các thừa số nguyên tố mạnh, hay nói cách khác p = a3*q3 - 1 trong đó (q3 là một thừa số nguyên tố lớn) ngăn chặn William p + 1

Từ đó, chúng ta cần phải xây dựng hàm để tìm kiếm số nguyên tố p, q thoả mãn riêng bản thân chúng là những thừa số nguyên tố mạnh, đảm bảo N = p*q sẽ là một hợp số khó bị phân tích bởi các giải thuật thông thường.

3 Phân tích và thiết kế

3.1 Required Function

3.1.1 Hàm getBigIntegerPrimeNumber

Hàm getBigIntegerPrimeNumber sẽ lấy đầu vào là số n bit của số và trả về 1 con số nguyên tố lớn có độ dài bit nhị phân tối đa là n bits một cách ngẫu nhiên. Chúng ta có thể sử dụng một số hàm tạo số nguyên lớn ngẫu nhiên trong lớp BigInteger của Java, sau đó kiểm tra nếu như số nguyên này có phải là số nguyên tố thì trả về, nếu không thì thực hiện tạo số nguyên lớn ngẫu nhiên cho tới khi số nguyên đó là số nguyên tố

- Đầu vào: n(int) là số bit của số nguyên tố cần tạo
- Giải thuật: Sử dụng vòng lặp do while, trong phần kiểm tra điều kiện while, sử dụng hàm is Prime là hàm trả về kiểu bool
- Pseudocode:

```
getBigIntegerPrimeNumber(n bits) {
    do {
        choose random bignumber res with n bits
    } while (res isn't prime);
    return res;
}
```

- Đầu ra: Số nguyên tố có độ dài max = n bits.

3.1.2 Hàm isPrime

Hàm isPrime sẽ lấy đầu vào là một số nguyên lớn BigInteger n và trả về 1 giá trị bool là true/false ứng với số n có phải là số nguyên tố không. Hàm này sẽ dùng Miller-Rabin lặp với số lần lặp là 20 lần để kiểm tra tính nguyên tố của số.

- Đầu vào: Số lớn N.
- Pseudocode:

```
 boolean isPrime(n) {
if (n == 1) return false
if (n == 2) return true
if (n % 2 == 0) return false

d = n - 1,
while(d % 2 == 0): {d = d/2, r= r+1}
i=0
while(i<20)
{miller-rabin n, d, r;
if not pass miller-rabin test return false;
i++}
return true
}

```

- Đầu ra: Đúng/Sai

3.1.3 Hàm getGCD

Đây là hàm với chức năng tính toán và trả về ước chung lớn nhất của 2 số nguyên lớn, bằng cách sử dụng giải thuật Euclid để tim ra ước chung lớn nhất, thực hiện việc lặp lại trừ số lớn hơn cho số nhỏ hơn

- Đầu vào: số nguyên a, b
- Đầu ra: Ước chung lớn nhất của a, b

- Pseudocode:

```
        loop:
        If a is equal to 0, return b
        If b is equal to 0, return a
        If a is greater than b:
            Set a to a modulo b
        Else:
            Set b to b modulo a

```

3.1.4 Hàm findDecryptionKey_d

- Đầu vào: số nguyên e, p, q
- Đầu ra: Khóa giải mã d
- Giải thuật: Tính toán $\phi(n)$ dựa trên (p-1)*(q-1) và tính d từ hàm modulo nghịch đảo của e và $\phi(n)$ dựa trên công thức tạo khoá từ cơ sở lí thuyết của mã hoá RSA
- Pseudocode:

```
    Function findDecryptionKey_d(e, p, q):
    phi_n = (p - 1) * (q - 1)
    d = mod_Inverse(e, phi_n)
    Return d

```

3.1.5 generateRandomKeyPair

- Đầu vào: số nguyên p, q
- Đầu ra: Mảng chứa cặp khóa e,d
- Tương tự như hàm ở trên, nhưng ở đây hàm này sẽ trực tiếp số mũ mã hoá công khai đầu vào bằng cách tính số nguyên tố cùng nhau với $\phi(n)$ và rồi tìm số mũ mã hoá bí mật d
- Pseudocode:

```
    Function generateRandomKeyPair(p, q):
    phi_n = (p - 1) * (q - 1)
    random = Random()
    e = RandomBigInteger(phi_n.bitLength(), random)
    loop: (e <= 1 or e >= phi_n or GCD(e, phi_n) != 1):
        e = RandomBigInteger(phi_n.bitLength(), random)
        d = mod_Inverse(e, phi_n)
    Return [e, d]

```

3.1.6 Hàm Encrypt

Hàm Encrypt sẽ lấy giá trị m,e,n, trả về giá trị $m^e mod n$.

- Đầu vào: m: plain text ,e: số mũ mã hoá công khai,n: cơ số modulo
- Giải thuật: Sử dụng hàm modulo_Pow, sẽ được hiện thực ở phần hàm hỗ trợ
- Pseudocode:

```
        Encrypt(m,e,n) {
            return modulo_Pow(m, e, n);
        }

```

- Đầu ra: c(ciphered text)

3.1.7 Hàm Decrypt

Hàm Decrypt sẽ lấy giá trị c,d,n, trả về giá trị $c^d mod n$.

- Đầu vào: c: cipher text ,d: số mũ mã hoá bí mật, n: cơ số modulo
- Giải thuật: Sử dụng hàm modulo_Pow, sẽ được hiện thực ở phần hàm hỗ trợ
- Pseudocode:

```
        Decrypt( c,  d,  n) {
            return modulo_Pow(c, d, n);
        }

```

- Đầu ra: m(plain text)

3.2 Helper Function

3.2.1 Hàm Miller_Rabin

Hàm Miller_Rabin sẽ lấy đầu vào 2 số lớn n, d và 1 số nguyên r, trả về giá trị boolean.
Thứ tự tính như sau:

- Lựa a trong đoạn [2,n-2]
- Tính $n-1$ và $d*2^r$
- $x = a^d$ mod n, nếu x = 1 hoặc x = n-1 trả về true.
- Từ $j = 0$ tới $r-1$: $x = x^2$ mod n, nếu x = 1 hoặc x = n-1 trả về true.
- Nếu chạy xong vòng lặp không có true sẽ trả về false.

Mô tả về hàm:

- Đầu vào: 1 số n bất kì
- Pseudocode:(Đã nói ở phần cơ sở lý thuyết)
- Đầu ra: Chắc chắn không phải nguyên tố/Có thể là nguyên tố

3.2.2 Hàm modulo_pow

Hàm này sẽ lấy giá trị a,b,n và tính $a^b mod n$.

- Đầu vào: a: cơ số; b: số mũ, n: cơ số modulo
- Lặp b lần mỗi lần nhân với a và thực hiện modulo cho n
- Pseudocode:

```
    modulo_Pow( a,  b,  n) {
    res = 1
    For i = 1 to b:
        res = (res * a) mod n
    Return res
    }
```

- Đầu ra: Kết quả của phép modulo của $a^b mod n$

3.2.3 mod_Inverse

- Đầu vào: số nguyên e, n
- Đầu ra: d sao cho (e * d) mod n = 1
- Giải thuật: nhận vào số nguyên e và n, nếu như e và n không phải số nguyên tố cùng nhau, không thoả mãn yêu cầu của mã hoá RSA cũng như yêu cầu để tìm modulo nghịch đảo. Nếu như thoả mãn, sử dụng giải thuật Extended Euclidean để tính toán và tìm ra số d và trả về
- Pseudocode:

```
    Function mod_Inverse(e, n):
    if getGCD(e, n) != 1:
        Throw ArithmeticException("Please check to ensure that e and n are coprime
    integers")
    d = extendedEuclidean(e, n)
    loop: d < 0:
        d = d + n
    Return d

```

3.2.4 extendedEuclidean

- Đầu vào: số nguyên a, b
- Đầu ra: x là nghịch đảo nhân modular của a ,b
- Giải thuật: Sử dụng giải thuật Extended Euclidean để tìm ra số x và y sao cho ax+by=GCD(a,b). Như đã giải thích ở phần cơ sở lý thuyết, vì đầu vào gọi từ hàm mod_Inverse ta có a và b là 2 số nguyên tố cùng nhau (e,n) nên nói cách khác GCD(a,b)=1, hay đơn giản hơn ax+by = 1.
  Từ ax + by = 1 =>(ax + by) mod b = 1 => (ax) mod b = 1 => hay nói cách khác x chính là nghịch đảo modulo của số a trên cơ số modulo b
- Pseudocode:

```
    Function extendedEuclidean(a, b):
    x = 0
    y = 1
    lastX = 1
    lastY = 0
    loop: (b != 0):
        quotient = a // b
        remainder = a % b
        a = b
        b = remainder
        temp = x
        x = lastX - quotient * x
        lastX = temp
        temp = y
        y = lastY - quotient * y
        lastY = temp
    Return lastX

```

3.3 Improve RSA

3.3.1 findStrongPrime

- Đầu vào: n: số bit của số nguyên tố cần tính
- Đầu ra: res là số nguyên tố mạnh cần tìm
- Giải thuật: Chúng ta cần tìm một số nguyên tố thoả p-1 và p + 1 đều có thừa số nguyên tố lớn: p = a_1p_0 + 1 và p = a_2p_1 -1 với p_0 và p_1 là các số nguyên tố lớn. Trong đó bản thân p_0 cũng là một số nguyên tố mạnh

  - Tạo một số nguyên lớn res ngẫu nhiên với số bit là số bit yêu cầu đầu vào
  - Tạo 2 số nguyên tố p_0 và p_1, trong đó p_0 được tính là một số nguyên tố gần mạnh được lấy từ hàm findStrongPrime_p1 và p_1 được lấy từ hàm getBigIntegerPrimeNumber, cả p_0 và p_1 đều có số bit bằng 1/2 số bit của res.
  - Kiểm tra 2 số nguyên tố sinh ra có trùng nhau không, nếu có thực hiện việc sinh lại p_1
  - Tính increment = 2*p*q (Đây chính là bước nhảy để tìm ra số nguyên tố)
  - Ta tính inv_1, inv_2 lần lượt là nghịch đảo của modulo (p_1,p_0) và nghịch đảo modulo (p_0,p_1)
  - Ta gán $crt_1 = inv_1 * p_1$. Vậy crt_1 thoả: $crt_1 = 1 \bmod p_0$ và $crt_1 = 0 \bmod p_1$
  - Ta gán $crt_2 = (p_1 - inv_2) * p_0$. Vậy crt2 thoả: $crt_2 = -1 \bmod p_1$ và $crt_2 = 0 \bmod p_0$
  - Ta tính crt = crt1 + crt2 + increment. Suy ra lúc này crt thoả tính chất: $crt_2 = -1 \bmod p_1$ và $crt_2 = 1 \bmod p_0$
  - Ta biến đổi res = res + (crt - res % increment)% increment. Như vậy res hiện tại đã thoả mãn tính chất: $res = -1 \bmod p_1$ và $res = 1 \bmod p_0$.
  - Tuy nhiên res chưa thoả tính chất là số nguyên tố, vì vậy ta cần lặp và mỗi lần tăng res một khoảng increment là 4*p*q, như vậy res luôn giữ được tính chất $res = -1 \bmod p_1$ và $res = 1 \bmod p_0$ cho đến khi res là số nguyên tố.

- Pseudocode:

```
    function findStrongPrime(bit: integer): BigInteger
    two := 2
    rand := new Random()
    res := new BigInteger(bit, rand)
    p0 := findStrongPrime_p1(bit/2)
    p1 := getBigIntegerPrimeNumber(bit/2)
    while (p1 == p0) do
        p1 := getBigIntegerPrimeNumber(bit/2)
    end while
    p1 := p1 * two
    increment := p0 * p1
    inv1 := mod_Inverse(p1, p0)
    crt1 := inv1 * p1
    inv2 := mod_Inverse(p0, p1)
    crt2 := p1 - inv2
    crt2 := crt2 * p0
    crt := crt1 + crt2 + increment
    resmod := res mod increment
    res := res + (crt - resmod) mod increment
    increment := increment * two
    while (not isPrime(res)) do
        res := res + increment
    end while
    return res
end function


```

3.3.2 findStrongPrime_p1

- Đầu vào: n: số bit của số nguyên tố cần tính
- Đầu ra: res là số nguyên tố mạnh p_0 cần tìm cho hàm phía trên
- Giải thuật: Chúng ta cần tìm một số nguyên tố thoả p-1 có thừa số nguyên tố lớn: p = $a_1 p_0 + 1$ với p_0 số nguyên tố lớn.
  - Tạo một số nguyên lớn res ngẫu nhiên với số bit là số bit yêu cầu đầu vào
  - Tạo số nguyên tố ngẫu nhiên p_0 được lấy từ hàm getBigIntegerPrimeNumber, cả p_0 và p_1 đều có số bit bằng 1/2 số bit của res.
  - Tính increment = $2p^2$ (Đây chính là bước nhảy để tìm ra số nguyên tố)

- Ta tính crt là nghịch đảo modulo (1,p), thoả tính chất crt = 1 mod (p)
- Ta tính crt = crt increment. Suy ra lúc này crt thoả tính chất: $crt = 1 \bmod p$
- Ta biến đổi res = res + (crt - res % increment)% increment. Như vậy res hiện tại đã thoả mãn tính chất: $res = 1 \bmod p$
- Tuy nhiên res chưa thoả tính chất là số nguyên tố, vì vậy ta cần lặp và mỗi lần tăng res một khoảng increment là $4 * p^2$, như vậy res luôn giữ được tính chất $res = 1 \bmod p$ cho đến khi res là số nguyên tố.

- Pseudocode:

```
function findStrongPrime_p1(bit: integer): BigInteger
    two := 2
    rand := new Random()
    res := new BigInteger(bit, rand)
    p := getBigIntegerPrimeNumber(bit/2)
    p := p * two
    increment := p * p
    crt := mod_Inverse(BigInteger.ONE, p)
    crt := crt + increment
    resmod := res mod increment
    res := res + (crt - resmod) mod increment
    increment := increment * two
    while (not isPrime(res)) do
        res := res + increment
    end while
    return res
end function
```

3.4 Evaluate

3.4.1 normalRSA_Encryption

- Đầu vào: số ngẫu nhiên num_bits
- Đầu ra: in ra màn hình cặp khoá bí mật và công khai, quá trình mã hóa và giải mã văn bản
- Giải thuật:
  - Đầu tiên là tạo cặp số nguyên tố ngẫu nhiên p, q với num_bits là số lượng bit của mỗi số nguyên tố. Nếu như cặp số p và q bị trùng nhau thì thực hiện việc sinh lại số nguyên tố q.
  - Sử dụng hàm generateRandomKeyPair để sinh ra cặp khoá công khai e và bí mật d.
  - In ra màn hình cặp khoá công khai và bí mật
  - Tạo bản plain-text với số bit bằng số bit của n (numsbit * 2) ngẫu nhiên, sau đó modulo m cho n để đảm bảo m < n.
  - Thực hiện gọi hàm Encrypt, Decrypt để mã hoá và giải mã, in ra màn hình để kiểm chứng.

- Pseudocode:

```
   Function normalRSA_Encryption(num_bits):
    BigInteger q
    BigInteger p = getBigIntegerPrimeNumber(num_bits)
    do:
        q = getBigIntegerPrimeNumber(num_bits)
    while q == p
    [e, d] = generateRandomKeyPair(p, q)
    BigInteger n = p * q
    print: public key
    print: private key
    Random rand = new Random()
    BigInteger plain_text = new BigInteger(num_bits * 2, rand)
    plain_text = plain_text.mod(p * q)
    Print("Plain_text:")
```

```
    Print(plain_text)
    Print("Encryption:")
    BigInteger c = Encrypt(plain_text, e, n)
    Print("Decryption:")
    plain_text = Decrypt(c, d, n)
    assert(plain_text < p * q)
    Print(plain_text)

```

3.4.2 improvedRSA_Encryption

- Đầu vào: số ngẫu nhiên num_bits
- Đầu ra: in ra màn hình cặp khoá bí mật và công khai, quá trình mã hóa và giải mã văn bản
- Giải thuật:
  - Đầu tiên là tạo cặp số nguyên tố mạnh p, q từ hàm findStrongPrime với num_bits là số lượng bit của mỗi số nguyên tố. Nếu như cặp số p và q bị trùng nhau thì thực hiện việc sinh lại số nguyên tố q.
  - Sử dụng hàm generateRandomKeyPair để sinh ra cặp khoá công khai e và bí mật d.
  - Để đảm bảo tính mã hoá mạnh mẽ và khó giải mã, hàm này sẽ thực hiện bước kiểm tra đơn giản nếu như e và d không phải số nguyên tố cùng nhau sẽ thực hiện sinh lại cặp khoá e, d
  - In ra màn hình cặp khoá công khai và bí mật
  - Tạo bản plain-text với số bit bằng số bit của n (numsbit * 2) ngẫu nhiên, sau đó modulo m cho n để đảm bảo m $<$ n.
  - Thực hiện gọi hàm Encrypt, Decrypt để mã hoá và giải mã, in ra màn hình để kiểm chứng.
- Pseudocode:

```
function improvedRSA_Encryption(num_bits: integer)
  p := findStrongPrime(num_bits)
  q := findStrongPrime(num_bits)
  while (q == p) do
      q := findStrongPrime(num_bits)
  end while

  e := 0
  d := 0
  while (getGCD(e, d) != 1) do
      pair := generateRandomKeyPair(p, q)
      e := pair[0]
      d := pair[1]
  end while

  n := p * q

  print("IMPROVED RSA-ENCRYPTION")
  print("Public key:")
  print("\te: " + e)
  print("\tn: " + n)


  print("Private key:")
  print("\td: " + d)
  print("\tp: " + p)
  print("\tq: " + q)

  rand := new Random()
  plain_text := new BigInteger(num_bits*2, rand)
  plain_text := plain_text mod (p * q)
  print("Plain_text:")
  print(plain_text)

  print("Encryption:")
  c := Encrypt(plain_text, e, n)
```

```
    print(c)

    print("Decryption:")
    plain_text := Decrypt(c, d, n)
    assert(plain_text < p * q)
    print(plain_text)
end function


```

3.4.3 main

- Đầu vào: Không có
- Đầu ra: Chạy 2 giải thuật Normal_RSA và Improved_RSA, in ra màn hình thời gian chạy của mỗi giải thuật
- Giải thuật:
  - Start_Time: Lưu thời gian bắt đầu chạy vào biến này.
  - Gọi các giải thuật khác nhau (2 hàm ở trên) với số lượng bit khác nhau để kiểm tra
  - Lưu thời gian kết thúc vào end_Time
  - Tính toán thời gian chạy và in ra màn hình
- Pseudocode:

```
  function main(args: array of string) throws Exception
    startTime := getCurrentTimeMillis()
    normalRSA_Encryption(1024)
    improvedRSA_Encryption(1024)
    endTime := getCurrentTimeMillis()
    elapsedTime := endTime - startTime
    print("Running time of the RSA-Encryption: " + elapsedTime + " milliseconds")
end function


```

3.4.4 Đánh giá

Ngoài việc thực hiện chạy và đo thời gian nhiều lần của một giải thuật trên số lượng bits của khoá khác nhau, tính trung bình và tạo thành bảng để so sánh tương quan giữa 2 giải thuật tiêu tốn về mặt thời gian và hiệu suất.
Để đo về mặt hiệu quả an toàn của mã hoá RSA, chúng ta xem xét sử dụng công cụ Dcode, một công cụ phân tích thừa số nguyên tố lớn không giới hạn thời gian, không giới hạn độ lớn của số, kết hợp hybrid mạnh mẽ các thuật toán phân tích thừa số như sau:

- Classical iterative division: Phương pháp chia lặp cổ điển.
- Pollard rho algorithm: Thuật toán Pollard rho.
- Elliptic curves: Đường cong Elliptic.
- The quadratic sieve algorithm: Thuật toán sàng bậc hai.

Sau khi chạy nhiều lần trên khoá n được tạo ra từ hai giải thuật với số lượng bit thay đổi từ 16 (nhỏ nhất), 64, 128, 512, và lớn nhất là 1024 để kiểm tra về tính ưu việt hơn của Improved RSA trong việc gây khó khăn cho các công cụ phân tích thừa số nguyên tố thông thường.

4 Hiện thực và đánh giá

4.1 Required Function

4.1.1 Hàm getBigIntegerPrimeNumber

Hàm getBigIntegerPrimeNumber sẽ lấy đầu vào là số n bit của số và trả về 1 con số nguyên tố lớn có độ dài bit nhị phân tối đa là n bits một cách ngẫu nhiên.

```
    static BigInteger getBigIntegerPrimeNumber(int num_bits) {
        Random rand = new Random();
        BigInteger primeCandidate;
        do {
            primeCandidate = new BigInteger(num_bits, rand);
        } while (!isPrime(primeCandidate));
        return primeCandidate;
    }
```

4.1.2 Hàm isPrime

Hàm isPrime sẽ lấy đầu vào là BigInteger n và trả về 1 giá trị bool là true/false. Hàm này sẽ dùng Miller-Rabin lặp để kiểm tra tính nguyên tố của số.

```
    static boolean isPrime(BigInteger n) {
        if (n.compareTo(BigInteger.ONE) <= 0) {
            return false;
        }
        if (n.equals(BigInteger.TWO)) {
            return true;
        }
        if (n.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
            return false;
        }

        BigInteger d = n.subtract(BigInteger.ONE);
        int r = 0;
        while (d.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
            d = d.divide(BigInteger.TWO);
            r++;
        }

        int k = 20;
        for (int i = 0; i < k; i++) {
            if (!Miller_Rabin(n, d, r)) {
                return false;
            }
        }
        return true;
    }
```

4.1.3 Hàm Encrypt

Hàm Encrypt sẽ lấy giá trị m,e,n, trả về giá trị $m^e mod n$.

```
    static BigInteger Encrypt(BigInteger m, BigInteger e, BigInteger n) {
        return modulo_Pow(m, e, n);
    }
```

4.1.4 Hàm Decrypt

Hàm Decrypt sẽ lấy giá trị c,d,n, trả về giá trị $c^d mod n$.

```
    static BigInteger Decrypt(BigInteger c, BigInteger d, BigInteger n) {
        return modulo_Pow(c, d, n);
    }
```

4.1.5 Hàm getGCD

Hàm getGCD để tính ước chung lớn nhất (GCD) của hai số nguyên lớn sử dụng thuật toán Euclid. Hàm này chấp nhận hai số nguyên lớn a và b dưới dạng đối tượng BigInteger và trả về ước chung lớn nhất của chúng.

```
    static BigInteger getGCD(BigInteger a, BigInteger b) {
        while (true) {
            if (a.equals(BigInteger.ZERO)) return b;
            if (b.equals(BigInteger.ZERO)) return a;
            if (a.compareTo(b) == 1) {
                a = a.mod(b);
            } else {
                b = b.mod(a);
            }
        }
    }
```

4.1.6 Hàm findDecryptionKey_d

Hàm findDecryptionKey_d để tính khóa giải mã d dựa trên khóa mã hóa e và hai số nguyên tố lớn p và q. Hàm này chấp nhận ba đối tượng BigInteger: e, p, và q, và trả về khóa giải mã d.

```
    static BigInteger findDecryptionKey_d(BigInteger e, BigInteger p, BigInteger q) {
        BigInteger phi_n = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.
    ONE));
        BigInteger d = mod_Inverse(e,phi_n);
        return d;
    }
```

4.1.7 Hàm generateRandomKeyPair

Hàm generateRandomKeyPair để tạo ra cặp khóa ngẫu nhiên e và d dựa trên hai số nguyên tố lớn p và q. Hàm này chấp nhận hai đối tượng BigInteger: p và q, và trả về một mảng BigInteger chứa cặp khóa (e, d).

```
    static BigInteger[] generateRandomKeyPair(BigInteger p, BigInteger q) {
        BigInteger phi_n = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE)
    );
        Random random = new Random();
        BigInteger e = new BigInteger(phi_n.bitLength(), random);
        while (e.compareTo(BigInteger.ONE) <= 0 || e.compareTo(phi_n) >= 0 || !e.gcd(
    phi_n).equals(BigInteger.ONE)) {
            e = new BigInteger(phi_n.bitLength(), random);
        }
        BigInteger d = mod_Inverse(e,phi_n);
        return new BigInteger[]{e, d};
    }
```

4.2 Helper Function

4.2.1 Hàm Miller_Rabin

Hàm Miller_Rabin sẽ lấy đầu vào 2 số lớn n, d và 1 số nguyên r, trả về giá trị boolean.
Thứ tự tính như sau:

- Lựa a trong đoạn [2,n-2]
- Tính $n-1$ và $d*2^r$
- $x = a^d$ mod n, nếu x = 1 hoặc x = n-1 trả về true.
- Từ $j = 0$ tới $r-1$: $x < -x^2$ mod n, nếu x = 1 hoặc x = n-1 trả về true.
- Nếu chạy xong vòng lặp không có true sẽ trả về false.

```
    private static boolean Miller_Rabin(BigInteger n, BigInteger d, int r) {
        Random rand = new Random();
        BigInteger a = new BigInteger(n.bitLength(), rand);
        a = a.mod(n.subtract(BigInteger.TWO)).add(BigInteger.TWO);
        BigInteger x = modulo_Pow(a,d, n);
        if (x.equals(BigInteger.ONE) || x.equals(n.subtract(BigInteger.ONE))) {
            return true;
        }
        for (int i = 0; i < r - 1; i++) {
            x = modulo_Pow(x,BigInteger.TWO, n);
            if (x.equals(BigInteger.ONE) || x.equals(n.subtract(BigInteger.ONE))) {
                return true;
            }
        }
        return false;
    }
```

4.2.2 Hàm modulo_pow

Hàm này sẽ lấy giá trị a,b,n và tính $a^b mod n$.

```
    static BigInteger modulo_Pow(BigInteger a, BigInteger b, BigInteger n) {
        BigInteger result = BigInteger.ONE;
        a = a.mod(n);
        while (b.compareTo(BigInteger.ZERO) > 0) {
            if (b.and(BigInteger.ONE).equals(BigInteger.ONE)) {
                result = result.multiply(a).mod(n);
            }
            b = b.shiftRight(1);
            a = a.multiply(a).mod(n);
        }
        return result;
    }
```

4.2.3 Hàm mod_Inverse

Hàm Hàm mod_Inverse để tính nghịch đảo modulo của một số nguyên e theo modulo n. Hàm này chấp nhận hai đối tượng BigInteger: e và n, và trả về một số nguyên d sao cho (e * d) mod n = 1.

```
    static BigInteger mod_Inverse(BigInteger e, BigInteger n) {

    if (!getGCD(e, n).equals(BigInteger.ONE)) {
            throw new ArithmeticException("Please check to ensure that e and n are
    coprime integers");
        }
        BigInteger d = extendedEuclidean(e, n);
        while (d.compareTo(BigInteger.ZERO) < 0) {
            d = d.add(n);
        }
        return d;
    }
```

4.2.4 Hàm extendedEuclidean

Hàm extendedEuclidean để tính toán thuật toán Euclid mở rộng (Extended Euclidean Algorithm) giữa hai số nguyên a và b. Hàm này trả về các giá trị GCD(a,b), x và y sao cho ax + by = GCD(a,b).

```
    private static BigInteger extendedEuclidean(BigInteger a, BigInteger b) {
        BigInteger x = BigInteger.ZERO;
        BigInteger y = BigInteger.ONE;
        BigInteger lastX = BigInteger.ONE;
        BigInteger lastY = BigInteger.ZERO;
        BigInteger temp;
        while (!b.equals(BigInteger.ZERO)) {
            BigInteger quotient = a.divide(b);
            BigInteger remainder = a.mod(b);
            a = b;
```

```
            b = remainder;
            temp = x;
            x = lastX.subtract(quotient.multiply(x));
            lastX = temp;
            temp = y;
            y = lastY.subtract(quotient.multiply(y));
            lastY = temp;
        }
        return lastX;
    }
```

4.3 Improve RSA

4.3.1 Hàm findStrongPrime_p1

Hàm findStrongPrime_p1 được dùng để tạo ra một số nguyên tố mạnh cấp độ 2 dựa trên số bit được cung cấp.

```
    static BigInteger findStrongPrime_p1(int bit) {
        BigInteger two = new BigInteger("2");

        Random rand = new Random();
        BigInteger res = new BigInteger(bit,rand);
        BigInteger p = getBigIntegerPrimeNumber(bit/2);
        p = p.multiply(two);
        BigInteger increment = p.multiply(p);

        BigInteger crt = mod_Inverse(BigInteger.ONE, p);
        crt = crt.add(increment);

        BigInteger resmod = res.mod(increment);
        res = res.add(crt.subtract(resmod).mod(increment));
        increment = increment.multiply(two);

        while (!isPrime(res)) {
            res = res.add(increment);
        }
        return res;
     }
```

4.3.2 Hàm findStrongPrime

Hàm findStrongPrime được dùng để tạo ra một số nguyên tố mạnh dựa trên số bit được cung cấp:

- Tạo số nguyên tố mạnh q1 cấp 2: sử dụng hàm findStrongPrime_p1
- Tạo số nguyên tố ngẫu nhiên q": sử dụng hàm getBigIntegerPrimeNumber
- Trả về số nguyên tố mạnh p bằng phép tính: p = a1*q1 + 1 và p = a3*q3 - 1 (trong đó a1 và a3 là các số nguyên).
- Kiểm tra số tạo ra có phải số nguyên tố hay không bằng cách sử dụng hàm IsPrime.

```
    static BigInteger findStrongPrime(int bit) {
        BigInteger two = new BigInteger("2");
        Random rand = new Random();
        BigInteger res = new BigInteger(bit,rand);
        BigInteger p0 = findStrongPrime_p1(bit/2);
        BigInteger p1 = getBigIntegerPrimeNumber(bit/2);
        while (p1.compareTo(p0) == 0) {
            p1 = getBigIntegerPrimeNumber(bit/2);
        }
        p1 = p1.multiply(two);

        BigInteger increment = p0.multiply(p1);

        BigInteger inv1 = mod_Inverse(p1, p0);
        BigInteger crt1 = inv1.multiply(p1);
```

```
        BigInteger inv2 = mod_Inverse(p0, p1);
        BigInteger crt2 = p1.subtract(inv2);
        crt2 = crt2.multiply(p0);

        BigInteger crt = crt1.add(crt2).add(increment);
        BigInteger resmod = res.mod(increment);
        res = res.add(crt.subtract(resmod).mod(increment));
        increment = increment.multiply(two);

        while (!isPrime(res)) {
            res = res.add(increment);
        }
        return res;
    }
```

4.4 Demo

4.4.1 Hàm normalRSA_Encryption

Hàm normalRSA_Encryption để thực hiện mã hóa RSA với số nguyên tố ngẫu nhiên được tạo bằng cách sử dụng hàm tạo số nguyên tố ngẫu nhiên. Hàm này sẽ tạo ra một cặp khóa (public key và private key) và sau đó sử dụng chúng để mã hóa và giải mã một văn bản đơn giản.

```
    static void normalRSA_Encryption(int num_bits){
        // Create key
        BigInteger q;
        BigInteger p = getBigIntegerPrimeNumber(num_bits);
        do{
         q = getBigIntegerPrimeNumber(num_bits);}
        while (q.compareTo(p)==0);
        BigInteger [] pair = generateRandomKeyPair(p, q);
        BigInteger e = pair[0];
        BigInteger d = pair[1];
        BigInteger n = p.multiply(q);
        System.out.println("NORMAL RSA-ENCRYPTION");System.out.println("Public key");
        System.out.println("\te: "+e);
        System.out.println("\tn: "+n);
        System.out.println("Private key");
        System.out.println("\td: "+d);
        System.out.println("\tp: "+p);
        System.out.println("\tq: "+q);
        Random rand = new Random();
        BigInteger plain_text = new BigInteger(num_bits*2, rand);
        plain_text = plain_text.mod(p.multiply(q));
        System.out.println("Plain_text:");
        System.out.println(plain_text);
        System.out.println("Encryption:");
        BigInteger c = Encrypt(plain_text, e, n);
        System.out.println(c);
        System.out.println("Decryption:");
        plain_text = Decrypt(c, d, n);
        assert(plain_text.compareTo(p.multiply(q)) == -1);
        System.out.println(plain_text);
    }
```

4.4.2 Hàm improvedRSA_Encryption

Hàm improvedRSA_Encryption triển khai một phiên bản cải tiến của thuật toán RSA sử dụng số nguyên tố mạnh. Đảm bảo rằng số nguyên tố mạnh được sử dụng trong thuật toán mã hóa RSA, làm tăng tính bảo mật của quá trình mã hóa. Nó cũng cung cấp một minh chứng cho việc mã hóa và giải mã sử dụng các khóa và văn bản gốc được tạo ra.

```
    // Create key
    BigInteger q;
    BigInteger e;
    BigInteger d;
    BigInteger p = findStrongPrime(num_bits);
    do{
```

```
     q = findStrongPrime(num_bits);}
    while (q.compareTo(p)==0);
    do{
    BigInteger [] pair = generateRandomKeyPair(p, q);
     e = pair[0];
     d = pair[1];
        } while (!getGCD(e, d).equals(BigInteger.ONE));
    BigInteger n = p.multiply(q);
    System.out.println("IMPROVED RSA-ENCRYPTION");System.out.println("Public key");
    System.out.println("\te: "+e);
    System.out.println("\tn: "+n);
    System.out.println("Private key");
    System.out.println("\td: "+d);
    System.out.println("\tp: "+p);
    System.out.println("\tq: "+q);
    Random rand = new Random();
    BigInteger plain_text = new BigInteger(num_bits*2, rand);
    plain_text = plain_text.mod(p.multiply(q));
    System.out.println("Plain_text:");
    System.out.println(plain_text);
    System.out.println("Encryption:");
    BigInteger c = Encrypt(plain_text, e, n);
    System.out.println(c);
    System.out.println("Decryption:");
    plain_text = Decrypt(c, d, n);
    assert(plain_text.compareTo(p.multiply(q)) == -1);
    System.out.println(plain_text);
    }
```

4.4.3 Hàm main

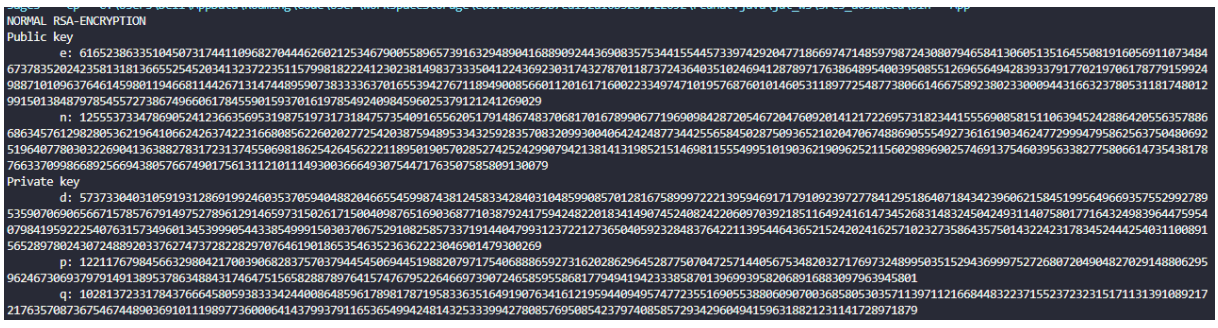
Thực hiện mã hóa RSA thông thường và RSA cải tiến với độ dài khóa lầ 1024 bit, tính và in ra thời gian thực thi của quá trình mã hóa.

```
    public static void main(String args[]) throws Exception {
        long startTime = System.currentTimeMillis();
        normalRSA_Encryption(1024);
        // improvedRSA_Encryption(1024);
        long endTime = System.currentTimeMillis();
        long elapsedTime = endTime - startTime;
        System.out.println("Running time of the RSA-Encryption: " + elapsedTime + "
    milliseconds");

    }
```

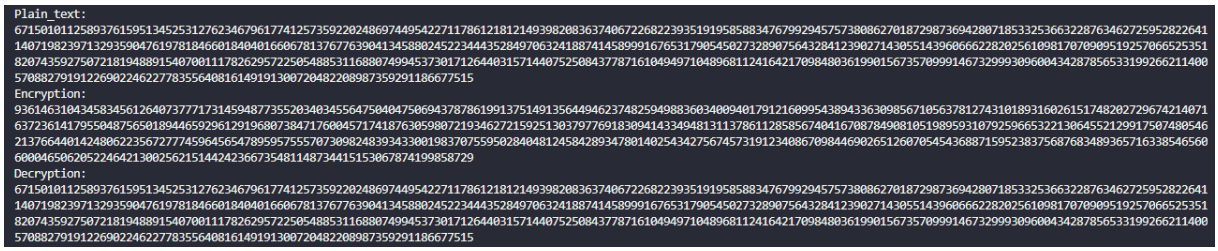
4.4.4 Kết quả chạy thử

Đầu tiên là với mã hoá RSA thông thường, dưới đây là cặp khoá công khai và khoá bí mật với độ dài 1024 bits được tạo ra



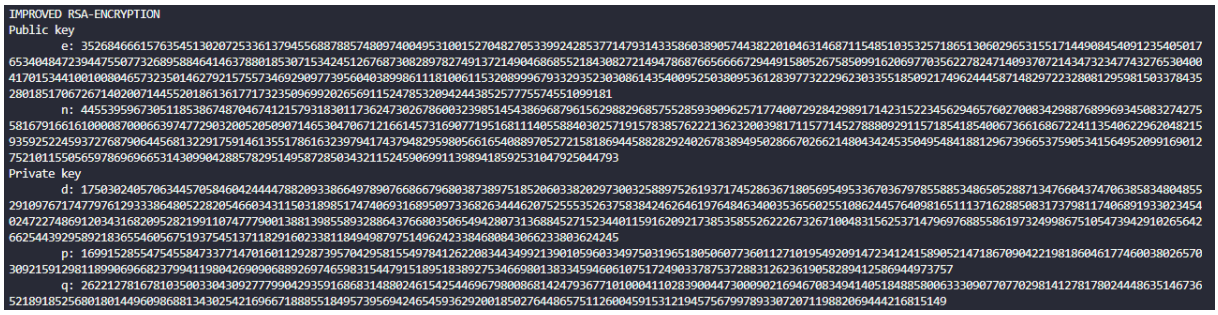
Hình 2: *Cặp khoá công khai - bí mật*

Kiểm thử tính đúng đắn trong việc mã hoá và giải mã thông điệp qua RSA thông thường



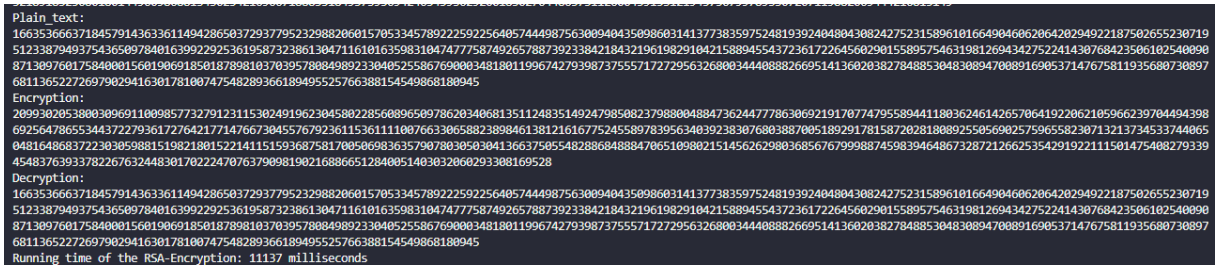
Hình 3: *Hình ảnh kiểm thử*

Tiếp theo là với mã hoá RSA được cải tiến với số nguyên tố mạnh, dưới đây là cặp khoá công khai và khoá bí mật với độ dài 1024 bits được tạo ra



Hình 4: *Cặp khoá công khai - bí mật*

Kiểm thử tính đúng đắn trong việc mã hoá và giải mã thông điệp qua RSA đã được cải tiến thêm



Hình 5: *Hình ảnh kiểm thử*

4.5 Evaluate

Number of bit	Running Time (Normal RSA)	Running Time (Improved RSA)	Defactoring Time (Normal RSA)	Defactoring Time (Improved RSA)
16 bits	24 ms	49 ms	0.4 ms	10 ms
64 bits	58 ms	75 ms	85 ms	1526 ms
128 bits	126 ms	146 ms	5 phút	43 phút
256 bits	380 ms	402 ms	Chưa thử được	Chưa thử được
512 bits	1004 ms	1329 ms	Chưa thử được	Chưa thử được
1024 bits	5002 ms	7468 ms	Chưa thử được	Chưa thử được

Bảng 1: *Bảng đánh giá kết quả*

Chúng ta có thể thấy được đơn giản qua bảng trên thì với số lượng bit tăng lên, thời gian cách biệt trong việc sử dụng mã hoá RSA thông thường và RSA cải tiến với đầu vào số nguyên tố mạnh là không lớn. Bên cạnh đó, công cụ dùng để kiểm thử là công cụ Prime Factors Decomposition của Dcode. Sử dụng kết hợp các thuật toán phân tích thừa số nguyên tố lớn:

- Classical iterative division: Phép chia lặp cổ điển
- Pollard rho algorithm: Thuật toán Pollard rho
- Elliptic curves: Đường cong Elliptic
- The quadratic sieve algorithm: Thuật toán sàng bậc hai

Chúng ta thấy được rõ ràng là thời gian để defactoring số N với N được tạo ra từ số nguyên tố mạnh thường tốn thời gian gần như xấp xỉ 10 lần so với N được tạo ra từ số nguyên tố thông thường. Chứng minh được tính hiệu quả cải thiện của thuật toán.

Như vậy, với số lượng bit từ 512 việc phá khoá RSA có thể tốn thời gian với con số lên tới hàng trăm năm, lẫn lượng tài nguyên tính toán khổng lồ, chính vì vậy chúng ta có thể tin tưởng được vào thuật toán RSA cải tiến.

5 Kết luận

Như vậy, thông qua bài tập lớn này, đã trình bày giới thiệu về mã hoá đối xứng RSA, các công thức liên quan đến RSA, cơ sở lí thuyết về thuật toán Euclid trong việc tìm ước chung lớn nhất, thuật toán Euclid mở rộng trong việc tìm modulo nghịch đảo cơ số n của số e. Tìm hiểu thuật toán Miller - Rabin và sử dụng Miller - Rabin lặp để xác định một số có phải là số nguyên tố thông qua xác suất với độ chính xác gần như tuyệt đối với 20 lần lặp.

Trong việc nghiên cứu tìm hiểu các giải thuật tấn công vào hệ mã RSA, bài báo cáo đã giải thích được 3 phương pháp tấn công cơ bản là William's p + 1; Pollard's p - 1, Cycling attac. Từ đó, tìm hiểu và tạo ra một số nguyên tố mạnh, cải thiện tính an toàn của RSA thông qua số nguyên tố mạnh đó. Cuối cùng, là sử dụng các công cụ phân tích thừa số nguyên tố để kiểm thử lại chính khả năng an toàn của RSA.

Và trong phần hiện thực, bài báo cáo đã trình bày chi tiết các phân tích thiết kết để thực hiện các hàm theo như lí thuyết đã nghiên cứu. Quan trọng hơn hết, bài báo cáo trong đó đã trình bày nội dung tìm hiểu được cách thức hiện thực hệ mã RSA mà không sử dụng một số hàm hiện thực sẵn trong Java, từ đó, hiểu được nguyên lí, giải thuật của việc tạo ra số nguyên tố lớn, tìm ra modulo nghịch đảo, định lí Euclid. Bên cạnh đó, hàm Improved RSA cố gắng ngăn chặn tấn công bằng các giải thuật William's p + 1 và Pollard's p - 1, Cycling attack, tìm ra được cách tạo ra một số nguyên tố mạnh, làm tăng thời gian cần phải tính toán để tấn công vào mã hoá RSA, tăng độ hiệu quả và an toàn trong việc mã hoá.

Ưu điểm rõ ràng có thể thấy của chương trình là tính đơn giản, dễ hiểu, độ an toàn bởi thuật toán, và tốc độ thực thi nhanh. Bài báo cáo đã trình bày khái quát nội dung nghiên cứu, giúp người đọc và sinh viên đã có được những kiến thức căn bản trong mã hoá RSA, một mã hoá nền tảng trong kiến thức về anh ninh mạng. Cuối cùng, bài báo cáo còn cải thiện được RSA cũng như kiểm chứng thông qua các công cụ. Đây là một trong những hướng đi tuy không mới nhưng khơi dậy nhiều hướng nghiên cứu tiếp theo cả trong việc mã hoá tin cậy và tìm lỗ hổng bảo mật.

Hướng đi tiếp theo của việc nghiên cứu và bài báo cáo này chưa đạt được, có thể mở rộng trong tương lai như việc hạn chế đoán được khoá giải mã d thông qua nắm bắt thông điệp được gửi đi, thông điệp được giải mã mà không cần phân tích thừa số nguyên tố n, hoặc tính toán d bằng một số phương pháp khác chưa được đề cập đến, tương tự như việc cố gắng tìm $\phi(n)$ thông qua n, và nhiều hướng nghiên cứu tấn công thông qua độ phức tạp, thời gian thực thi, năng lượng tiêu thụ để đoán được khoá giải mã, ...

References

[1] R.L. Rivest, A. Shamir, and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, 1977.