Đại học quốc gia TP.HCM
Trường Đại học Khoa học tự nhiên

Ngành khoa học máy

Môn học: Mã hóa ứng dụng

Đồ án cá nhân
Cài đặt thuật toán mã hóa RSA

Học viên:
Đỗ Quốc Thế – 19C11040

Giảng viên:
PGS.TS Nguyễn Đình Thúc

Mục lục

1 Giới thiệu

Đồ án cài đặt lại thuật toán mã hóa RSA với số p, q là các số nguyên tố 1024 bits bằng ngôn ngữ lập trình C++. Phần cài đặt này gồm 2 quá trình chính tương ứng với 2 quá trình chính trong mã hóa RSA: quá trình *phát sinh key* và quá trình *encrytp/decrypt*. Quá trình cài đặt sử dụng các thuật toán trong trong môn học như thuật toán nhân lấy phần dư, thuật toán lũy thừa lấy phần dư, thuật toán Bezout nhị phân, thuật toán số dư trung hoa và một số thuật toán khác.

Do đã thử qua một số thư viện số nguyên lớn được *public trên Github* và nhận thấy các thư viện khá chậm với số lớn và cồng kềnh nên đồ án tự cài đặt số nguyên lớn bằng các kĩ thuật lập trình căn bản. Qua nhiều *version* cải tiến, đồ án đã thực hiện được 2 quá trình chính kể trên với thời gian thực. Báo cáo này xin được trình bày 2 cách làm tốt nhất hiện đã đạt được.

Cách làm thứ nhất (cách 1) *code* lại hoàn toàn cách xử lý với số nguyên lớn mà không dùng thư viện hỗ trợ số nguyên lớn nào, cách làm này lưu trữ các chữ số trong dãy *bit nhị phân* dưới kiểu dữ liệu *char** cùng với các hàm thao tác nhanh nhất trong C/C++ như *memcpy, memset* và dùng các thuật toán trong chương trình học cùng với một số thuật toán tìm hiểu được từ các paper trên mạng.

Cách làm thứ hai (cách 2) cũng hoàn toàn tự *code* lại các thuật toán tuy nhiên có xử dụng kiểu dữ liệu nhiều bit hơn các kiểu dữ liệu thông thường có trong C++, cách này định nghĩa lại một kiểu số 4096 bits bằng thư viện Boost Multiprecision[(1)].

Cách làm thứ nhất thời gian thực thi các phép toán là không chậm nhưng cách làm thứ hai tối ưu hơn và đạt được thời gian xử lý rất nhanh. Kết quả chính của đồ án này là từ việc cài đặt theo cách 2. Tuy nhiên báo cáo cũng có sự so sánh giữa 2 cách cài đặt để rút ra được kết luận cuối cùng. Các phần tiếp theo trong báo cào sẽ nói về những bước cài đặt của đồ án, ở mỗi bước, báo cáo sẽ nói về cài đặt theo mỗi cách đã nêu ở trên.

Tuy đã cố gắng rất nhiều nhưng đồ án không tránh khỏi những thiếu sót nhất định, mong nhận được sự đóng góp của thầy để hoàn thiện hơn về kĩ năng và kiến thức cho những lần báo cáo tiếp theo. Cảm ơn thầy!

[(1)] https://t.ly/CmsG

2 Cài đặt tính toán cơ bản

2.1 Số nguyên lớn

Bước này là bước rất quan trọng hơn cả vì không cài đặt được cách lưu trữ số nguyên lớn thì không thể thực hiện các bước tiếp theo. Cài đặt số nguyên lớn chỉ phức tạp khi thực hiện theo cách 1.

Trong cách làm 1, các số được lưu dưới dạng dãy bit nhị phân, mỗi dãy bit là một con trỏ *char** và overload lại các toán tử cần dùng cho chương trình như *+; −; %; ≫, ≪, >, ==, >=*. Mỗi lần thực hiện các phép toán làm thay đổi độ dài của chuỗi bit nhị phân thì cần tạo lại vùng nhớ mới nên đồ án dùng các hàm có tốc độ xử lý cao như *memset*, *memcpy*.

Đối với cách làm 2, chỉ cần dùng thư viện Boost khai báo kiểu số 4096 bit như code dưới dây:

```
namespace mp = boost::multiprecision;
typedef mp::number<mp::cpp_int_backend<4096, 4096, mp::signed_magnitude, mp::unchecked, void> >  number_t;
```

Như vậy ta có thể sử dụng kiểu dữ liệu mới là **number_t** với các phép toán built-in như đối với các phép toán trên số nguyên trong C++. Điều này giúp cải thiện rất nhiều chi phí tính toán.

2.2 Phép Mod

Phần phép tính quan trọng tiếp theo sau khi đã tìm được các biểu diễn số nguyên lớn là cài đặt phép toán modulo. Nếu dùng kiểu dữ liệu **number_t** như phần trước trình bày thì phần này không có gì đặc biệt vì chỉ cần dùng phép toán % của kiểu dữ liệu. Nếu cài đặt lại kiểu dữ liệu số nguyên lớn thì phần này cực kì quan trọng vì tất cả các bước xử lý trong các bước tiếp theo đều dùng tới phép modulo.

Thực nghiệm cho thấy cách tính modulo truyền thống như *Thuật toán 1* khi tính cho số nguyên lớn rất chậm, đồ án đã cài đặt thuật toán modulo theo bài báo [6], nội dung của thuật toán này thể hiện trong *Hình 1*.

Với việc cài đặt thuật tóan modulo cải tiến, tốc độ chương trình được cải thiện rất nhiều.

Thuật toán 1 Tính modulo

Input: BigInt a, BigInt b
Output: a % b

1: **procedure** MOD(a, b)
2: **while** $a > b$ **do**
3: $a \leftarrow a - b$
4: **return** $a = b \ ? \ 0 : a$

Compute: $X \ mod \ Y$

$Width(\alpha) = width \ of \ \alpha \ in \ terms \ of \ bits$
$Split(\alpha, w) = \{\sum_{i=0}^{w-1} \alpha_i 2^i, ..., \sum_{i=0}^{w-1} \alpha_{Width(\alpha)-w+i} 2^i\}$
$Num(\hat{\alpha}) = number \ of \ elements \ in \ \hat{\alpha}$
$T = \sum_{i=0}^{Width(T)-1} t_i 2^i, \ t_i \in \{0, 1\}$

1: $\hat{G} = Split(X, Width(Y))$
2: $N = Num(\hat{G}) - 1$
3: **while** $N > 0$ **do**
4: $T = \hat{G}[N]$
5: **for** $i = Width(Y) - 1$ *downto* 0 **do**
6: $T = T << 1$
7: **while** $t_{Width(Y)} = 1$ **do**
8: $t_{Width(Y)} = 0$
9: $T = T + (2^{Width(Y)} \ mod \ Y)$
10: $\hat{G}[N-1] = \hat{G}[N-1] + T$
11: **while** $\hat{G}[N-1]_{Width(Y)} = 1$ **do**
12: $\hat{G}[N-1]_{Width(Y)} = 0$
13: $\hat{G}[N-1] = \hat{G}[N-1] + (2^{Width(Y)} \ mod \ Y)$
14: $N = N - 1$
15: **while** $\hat{G}[0] > Y$ **do**
16: $\hat{G}[0] = \hat{G}[0] - Y$
return $\hat{G}[0]$

Hình 1: Thuật toán tính nhanh Mod trong [6]

2.3 Phép MulMod

Cũng như việc cài đặt phép tính %, việc cài đặt phép tính MulMod dùng kiểu dữ liệu **number_t** đã định nghĩa không có gì đặc biệt ngoài việc đảm bảo không bị tràn số, tuy nhiên với kiểu dữ liệu 4096 bits thì việc tràn số trong phép nhân với hai số 1024 bits không thể xảy ra vì phép nhân số n bits và số t bits cho ra kết quả nhiều nhất là $n + t + 2$ bits (*[1] Chương 14 trang 595*). Thực hiện phép MulMod với kiểu dữ liệu **number_t** là thực hiện phép nhân trước rồi phép modulo sau.

Còn với cách 1, cài đặt thuật toán MulMod cho kiểu số nguyên lớn tự định nghĩa được thực hiện theo *Thuật toán 2* (tham khảo giáo trình).

Thuật toán 2 Tính MulMod

Input: BigInt x, BigInt y, BigInt n
Output: x*y % n

```
1: procedure MulMod(x, y, n)
2:     p ← 0
3:     if y_0 = 1 then                          ▷ Bit cuối của y là 1
4:         p ← x
5:     for i ← 1, m do                          ▷ m là số bit của y
6:         x ← 2 * x % n                        ▷ x L-Shift 1 bit rồi % n
7:         if y_i = 1 then
8:             p ← (p + x) % n
9:     Return p
```

2.4 Phép PowerMod

Quá trình cài đặt phép PowerMod cần thiết cho cả hai cách làm vì nếu dùng **number_t** thực hiện phép lũy thừa thì có thể sẽ xảy ra tràn số. Phép toán PowerMod được cài đặt theo *Thuật toán 3* (tham khảo giáo trình).

2.5 Kiểm tra số nguyên tố

Đồ án dùng định lý Fermat nhỏ để kiểm tra số số giả nguyên tố mạnh với 4 cơ sở là bốn số tự nhiên. *Thuật toán 4* mô tả cách kiểm tra số nguyên tố (tham khảo giáo trình).

Thuật toán 3 Tính PowerMod

Input: BigInt a, BigInt p, BigInt n
Output: $a^p \ \% \ n$

```
1: procedure PowerMod(a, p, n)
2:     y ← 1
3:     if p = 0 then
4:         Return y
5:     A ← x
6:     if p_0 = 1 then
7:         y ← x
8:     for i ← 1, m do                          ▷ m là số bít của p
9:         A ← A^2 mod n
10:        if p_i = 1 then
11:            y ← A * y mod n
12:    Return y
```

Thuật toán 4 Kiểm tra số nguyên tố

Input: BigInt n
Output: True: là số nguyên tố, False: ngược lại

```
1: procedure IsPrime(n)
2:     for i in {2, 5, 7, 9} do
3:         if i^{n-1} % n ≠ 1 then
4:             Return false
5:     Return true
```

Lưu ý: Có một vài số không phải là số nguyên tố nhưng thõa mãn định lý Fermat nhỏ gọi là ***số Carmichael*** *ví dụ 3 số Carmichael nhỏ nhất là* ***561, 1105, 1729****. Tuy nhiên tỉ lệ gặp phải số Carmichael rất thấp*[(2)] *vì chỉ có khoảng* ***255 số Carmichael*** $< 10^8$ *và khoảng* ***20,138,200 số Carmichael*** $< 10^{21}$*, do đó có thể dùng phép kiểm tra số nguyên tố như trên.*

[(2)] https://t.ly/1PtR

3 Cài đặt tính toán cho RSA

3.1 Phát sinh số nguyên tố mạnh

Việc phát sinh số nguyên tố n bits thực hiện bằng việc chọn lựa ngẫu nhiên một số n bits và kiểm thử xem số vừa phát sinh có phải là số nguyên tố hay không thông qua hàm *ISPRIME* ở phần trên. Một đặc điểm đáng chú ý là các số nguyên tố đều có dạng chung[3] là $6n \pm 1$, do đó có thể dùng đặc tính này để hạn chế số phép thử bằng cách chỉ thử những số có dạng $6n \pm 1$.

Phần cài đặt phát sinh số nguyên tố n bits được thực hiện bằng cách phát sinh một số nguyên dương ngẫu nhiên t có $n-3$ bits *(vì 6 có 3 bits)* sau đó kiểm tra số $6n+1$ hoặc số $6n-1$ có phải số nguyên tố hay không, nếu một trong hai số trên là số nguyên tố thì chọn, ngược lại tăng t lên 1 đơn vị và lặp lại quá trình. *Thuật toán 5* mô tả cách tạo số nguyên tố n bits.

Thuật toán 5 Tạo số nguyên tố

Input: n: số bits
Output: Số nguyên tố n bits

```
1:  procedure GenPrime(n)
2:      t ← GenNumber(n − 3)                ▷ Tạo số nguyên n − 3 bits
3:      while true do
4:          t ← 6n + 1
5:          if IsPrime(t) then
6:              Return t
7:          t ← 6n − 1
8:          if IsPrime(t) then
9:              Return t
10:         t ← t + 1
```

Đồ án cũng có cài đặt thuật toán tạo số ***nguyên tố mạnh*** để tăng tính an toàn cho RSA. Số nguyên tố mạnh được định nghĩa[4]:

$$p \in \mathcal{P} : p - 1 = 2r,\ r \in \mathcal{P}$$

Để đơn giản, đồ án thực hiện phát sinh số nguyên tố mạnh được thực hiện dựa vào bài báo [3] của nhóm tác giả *R. L. Rivest and A. Shamir and*

[3] https://t.ly/vn5e
[4] Giáo trình

L. Adleman năm 1978 (trang 9). Cụ thể, phát sinh một số nguyên tố u, sau đó tìm số nguyên tố đầu tiên trong dãy số $i * u + 1,\ i = 2, 4, 6, ...$ *Thuật toán 6* thể hiện việc phát sinh số nguyên tố mạnh.

Thuật toán 6 Phát sinh số nguyên tố mạnh

Input: n: số bits
Output: Số nguyên tố mạnh $\approx n$ bits

```
1: procedure GenStrongPrime(n)
2:     u ← GenPrime(n − 2)                ▷ Tạo số nguyên tố n − 2 bits
3:     t ← 1
4:     while true do
5:         i ← 2t
6:         p ← i * u + 1
7:         if IsPrime(p) then
8:             Return p
9:         t ← t + 1
```

Như trình bày ở trên nhóm tác giả R. L. Rivest phát triển cách phát sinh số nguyên tố mạnh (an toàn) nhằm tăng tính bảo mật cho RSA, tuy nhiên trong một bài báo sau này [4] *(trang 1)* cũng của chính tác giả, tác giả có nói số nguyên tố mạnh không làm tăng tính bảo mật của RSA lên quá nhiều so với các số nguyên tố lớn thông thường, do đó không cần thiết phải dùng số nguyên tố mạnh trong RSA.

3.2 Tìm số e và d

Theo tham khảo tài liệu trên Iternet [(5),(6)] thì trong RSA thường chọn e là các số nguyên tố nhỏ trong các số Fermat [(7)] ($e \in \{3, 5, 17, 257, 65537\}$) rồi từ e ta chọn d sao cho $ed \equiv 1\ (mod\ \phi)$. Tuy nhiên, như trong giáo trình môn học, để an toàn hơn không nên chọn trước số e nhỏ rồi chọn d vì d có khả năng sẽ nhỏ, nên chọn các số $d \geq \sqrt[4]{n}$.

[(5)] https://t.ly/zGU9
[(6)] https://t.ly/LXCr
[(7)] https://t.ly/nieN

Ở đây đồ án thực hiện chọn e, d bằng cách chọn e là các số nguyên tố ≥ 63357, chọn d tương ứng với mỗi e được chọn sau đó kiểm tra điều kiện $d \geq \sqrt[4]{n}$ nếu d thỏa thì chọn cặp e, d. *Thuật toán 7* minh họa quá trình tạo e, d

Thuật toán tìm d khi biết e, ϕ được dùng là thuật toán Bezout nhị phân tham khảo từ tài liệu [5] *(Phần thuật giải 1.49, trang 26).*

Thuật toán 7 Tạo key e, d

Input: p, q: Hai số nguyên tố lớn
Output: e, d sao cho $ed \equiv 1 \ (mod \ \phi) \ , \phi = (p-1)(q-1)$

```
1:  procedure GenKeys(p, q)
2:      n ← p * q
3:      φ ← (p − 1)(q − 1)
4:      t ← 10922                                   ▷ 6t + 5 = 65537
5:      while true do
6:          e ← 6t + 1
7:          if IsPrime(e) then
8:              d ← BinaryBezout(e, φ)
9:              if d ≥ ∜n then
10:                 break
11:         e ← 6t + 5
12:         if IsPrime(e) then
13:             d ← BinaryBezout(e, φ)
14:             if d ≥ ∜n then
15:                 break
16:         t ← t + 1
17:     Return e, d
```

3.3 Giải mã nhanh dùng CRT

Định lí số dư trung hoa có nội dung như sau: Với $n_i \in \mathcal{P}, n_i \neq n_j \ \forall i \neq j$ và $a_i \in \mathbb{N}$. Hệ phương trình sau có nghiệm duy nhất trong $\mathbb{Z}_{n_1 n_2 \ldots n_k}$

$$\begin{cases} x &= a_1 \ mod \ n_1 \\ x &= a_2 \ mod \ n_2 \\ &\vdots \\ x &= a_k \ mod \ n_k \end{cases} \tag{1}$$

Chứng minh của Quisquater & Couvreur trong [2] thể hiện việc áp dụng CRT trong RSA như sau: Với $p, q \in \mathcal{P}, p \neq q$, $\phi = (p-1)(q-1)$, $e, d \in \mathbb{N}, gcd(e, \phi) = 1, ed = 1 \ mod \ \phi$, $n = pq$, nếu $x = c^d \ mod \ n$ thì x cũng là nghiệm của hệ phương trình:

$$\begin{cases} x &= c^{d_1} \ mod \ p \\ x &= c^{d_2} \ mod \ q \end{cases} \tag{2}$$

$$\text{với } d_1 = d \ mod \ (p-1), d_2 = d \ mod \ (q-1)$$

Vận dụng *(2)*, quá trình *decrypt* được cải thiện rõ rệt vì thay vì tính c^d và modulo n thì kết quả có thể tính được bằng các phép tính tương tự nhưng với số bit của p, q nhỏ hơn nhiều so với số bit của d, n.

Cài đặt *decrypt nhanh* là cài đặt giải hệ phương trình CRT tổng quát *(1)* từ đó giải *(2)*.

Cho trước hai số nguyên a, b định lý Bezout phát biểu có thể tìm được x và y thỏa $ax + by = gcd(a, b)$. Trong *hệ phương trình (1)* ta thấy với $n = n_1 n_2 \dots n_k, n_i \in \mathcal{P}$ thì $gcd(n_i, \frac{n}{n_i}) = 1$. Như vậy bằng thuật toán Euclid mở rộng (hoặc *Bezout nhị phân* [5] như đã nêu ở phần trên) có thể tìm được r_i và s_i sao cho $n_i r_i + \frac{n}{n_i} s_i = 1$, từ đó ta có thể tìm được nghiệm duy nhất của *(1)* là $x = \sum_{i=1}^{k} a_i s_i \frac{n}{n_i}$. *Thuật toán 8* dùng để giải phương trình CRT tổng quát.

Thuật toán 8 Giải hệ phương trình CRT

Input: listA, listN: Danh sách a_i và n_i
Output: x: Nghiệm của hệ phương trình CRT

```
1:  procedure SolveCrt(listA, listN)
2:      len ← sizeof(listN)
3:      n ← 0
4:      sum ← 0
5:      for i in listN do
6:          n ← n * i
7:      for i ← 1, len do
8:          n_i ← listN[i]
9:          p ← n/n_i
10:         a_i ← listA[i]
11:         r, s ← BinaryBezout(n_i, p)
12:         sum ← a_i * s * p
13:     Return sum % n
```

Ngoài ra, nếu tính và lưu trước giá trị $s_1\frac{n}{p}$ và $s_2\frac{n}{q}$ cùng với việc tính và lưu e, d trong quá trình tạo keys của RSA thì khi *decrypt* sẽ không phải chạy lại thuật toán tìm s_i do đó thời gian *decrypt* có thể được rút ngắn hơn nữa.

4 Kết quả đạt được

Trong cách làm 1 (xây dựng lại class BigInt), dù đã cải tiến rất nhiều về cách code như chuyển từ lưu trữ chuỗi nhị phân bằng *std::string* sang *char**, chuyển từ dùng *strcpy* sang *memcpy*, dùng *memset*, ... chương trình đạt được tốc độ runtime cũng tương đối tốt tuy nhiên vẫn còn chậm hơn rất nhiều so với cách làm 2 (dùng *boost::multiprecision*). Dưới đây là bảng so sánh hai cách làm với số nguyên 512 bits và các kết quả khi thực hiện các phép tính với theo cách làm 2.

Cách làm lưu trữ số	**Mod**	**MulMod**	**PowerMod**
BigInt	63	970	3,814,117
boost::multiprecision	≈ 1	≈ 6	$\approx 4,500$

Bảng 1: Thực hiện các phép toán với số nguyên 512 bits (Đơn vị: μs)

Phép tính	Mod	MulMod	PowerMod
Thời gian	≈ 1	≈ 19	$\approx 42,185$

Bảng 2: Thực hiện các phép toán với số nguyên 1024 bits dùng cách lưu trữ số lớn với *boost::multiprecision* (Đơn vị: μs)

Số bit p, q	Số bit M	Encrypt	Decrypt $c^d \ mod \ N$	Decrypt CRT
1024	500	1,049	236,354	58,427
512	500	427	57,252	22,798

Bảng 3: Thời gian thực hiện *encrypt* và *decrypt* (Đơn vị: μs)

5 Kết luận

Đồ án đã thực hiện cài đặt lại và thực nghiệm thuật toán mã hóa RSA bằng 2 cách, nhìn chung cả hai cách đều không dùng thư viện số nguyên lớn mà tự cài đặt lại các phép toán. Cách 2 dùng cách lưu trữ số nguyên dựa vào thư viện *boost::multiprecision* cho thấy thời gian chạy tốt hơn, thời gian viết chương trình nhanh hơn trong khi tự cài đặt lại *class BigInt* tốn thời gian code (vì cần phải unittest để kiểm tra tính đúng đắn của code) mà thời gian chạy cũng không được nhanh, hơn nữa việc cài đặt này là không cần thiết vì *"reinvent the wheel"* không phải là cách hay.

Nhược điểm chủ yếu của việc viết lại lớp số nguyên lớn là yêu cầu phải xử lý số nguyên có dấu (vì các thuật toán như Bezout nhị phân hay Euclid mở rộng đều cần thao tác với số có dấu) tuy nhiên việc xử lý số có dấu tốn khá nhiều thời gian implement. Hơn nữa, với cách làm hiện tại thì việc chậm là hiển nhiên vì nếu số 1024 bits tức là 1024 *char* trong con trỏ $char* \rightarrow$ cần xử lý $8 * 1024 = 8192$ bits! Vậy nên để cải thiện tốc độ cần tối ưu thêm ở mức lưu trữ các bits số mà hiện tại *boost::multiprecision* đã thực hiện điều đó.

Tài liệu tham khảo

[1] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996.

[2] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for rsa public-key cryptosystem. *Electronics Letters*, 18, 1982.

[3] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21, 2 1978.

[4] Ron Rivest and Robert Silverman. Are 'strong' primes needed for rsa, 2001. rsilverman@rsasecurity.com 11352 received 30 Jan 2001.

[5] Bùi Doãn Khanh và Nguyễn Đình Thúc. Giáo trình mã hóa thông tin: Lý thuyết và ứng dụng, 2004.

[6] Mark A. Will and Ryan K. L. Ko. Computing mod without mod, 2014.