

PRIME AND DIFFIE-HELLMAN KEY EXCHANGE

IC G14 PJRP-003

PROJECT SOURCE CODE OVERVIEW

DIFFIE – HELLMAN ALGORITHM, MILLER – RABIN PRIMALITY TEST

The report is part of the first group project of the Introduction to Cryptography course, a module provided by the University of Natural Sciences, VNU-HCM.



HCMUS
Trường Đại học
Khoa học Tự nhiên,
ĐHQG-HCM

CQ2022/22, Introduction to Cryptography, University of Natural Sciences, VNU-HCM

INTRODUCTION TO CRYPTOGRAPHY

A PROJECT REPORT

IC G14 PJRP-003

RESPONSIBILITY

GROUP 14

Member 01: 22120257

Đinh Lê Gia Như

Member 02: 22120226

Lê Trọng Nghĩa

Member 03: 22120192

Nguyễn Đăng Long

FACULTY OF INFORMATION TECHNOLOGY

UNIVERSITY OF SCIENCE, VNU-HCM CITY UNIVERSITY

HO CHI MINH CITY, VIET NAM COUNTRY

October, 2024

Language: English, Vietnamese

Tóm tắt

Tài liệu IC G14 PJRP-003 này là tài liệu báo cáo thuộc Đề án Lab01 (Dự án nhỏ Lab01), khóa học Nhập môn mã hóa – mật mã, bộ môn Công nghệ trí thức, Trường Đại học Khoa học Tự Nhiên, Đại học Quốc Gia Thành phố Hồ Chí Minh, Việt Nam.

IC G14 PJRP-003 báo cáo các vấn đề tổng quan mã nguồn chương trình dự án Lab01, thuật toán Diffie-Hellman và phép thử kiểm tra số nguyên tố Miller-Rabin.

Các tài liệu, văn bản có liên quan đến Đề án Lab01 bao gồm: Báo cáo IC G14 PJRP-001, Báo cáo IC G14 PJRP-002, Báo cáo IC G14 PJRP-003 được lưu trữ trong thư mục project_01_report và nộp bài tập đề án trên trang môn học (Moodle) của Khoa Công nghệ thông tin, Trường Đại học Khoa học Tự nhiên, Đại học Quốc gia Thành phố Hồ Chí Minh, Việt Nam. Ngoài ra, mã nguồn của đề án đã thực hiện được lưu trữ trong thư mục project_01_source có kèm theo với các tài liệu báo cáo này khi kết thúc đề án.

Từ khóa có liên quan: diffie-hellman; key exchange; NIST DH; cryptography; miller-rabin; primality test; modular exponentiation; random; CSPRNG; c++; STL; function; variable; class; OOP; safe prime; sophia germain prime; OEIS; RFC; NIST DLMF; protocol; algorithm; cybersecurity; KEK

Lời cảm ơn

Người thực hiện bài báo cáo này, tôi là Lê Trọng Nghĩa, trân trọng ghi nhận đóng góp và đánh giá cao đến những đóng góp của các cá nhân Đinh Lê Gia Như, Lê Trọng Nghĩa, Nguyễn Đăng Long là thành viên của nhóm 14 thực hiện đồ án Lab01 có các nội dung liên quan đến các vấn đề được nêu có trong tài liệu này bao gồm các vấn đề về công nghệ thông tin, mật mã và bảo mật.

Tác giả bài báo cáo cũng xin cảm ơn các hướng dẫn, gợi ý về đồ án, và các giải đáp thắc mắc từ phía giảng viên phụ trách khóa học, người quản lý dự án có liên quan.

Nguồn tham khảo

Nguồn tài liệu tham khảo:

- a. Tiêu chuẩn hóa: RFC 2631, RFC 3526, RFC 2409, RFC 8268, NIST DH, NIST SP 800-56, NIST SP 800-131A Re.2, NIST SP 800-203, OEIS, OEIS: A005385, OEIS: A059327, OEIS: A059394, OEIS: A059395, OEIS: A059452, OEIS: A075705, OEIS: A075706, OEIS: A075707, OEIS: A227853, OEIS: A014233, cppreference
- [01] RFC 2631 <https://datatracker.ietf.org/doc/html/rfc2631>
- [02] RFC 3526 <https://datatracker.ietf.org/doc/html/rfc3526>
- [03] RFC 2409 <https://datatracker.ietf.org/doc/html/rfc2409#section-6.2>
- [04] RFC 8268 <https://datatracker.ietf.org/doc/html/rfc8268#section-3>
- [05] NIST DH <https://csrc.nist.gov/glossary/term/dh>
- [06] NIST SP 800-56: <https://csrc.nist.gov/files/pubs/sp/800/56/a/upd1/final/docs/sp800-56-draft-jul2005.pdf>
- [07] NIST SP 800-131A Re.2: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf>
- [08] NIST SP 800-203: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-203.pdf>
- [09] OEIS <https://oeis.org/>
- [10] OEIS: A005385 <https://oeis.org/A005385>
- [11] OEIS: A059327 <https://oeis.org/A059327>
- [12] OEIS: A059394 <https://oeis.org/A059394>
- [13] OEIS: A059395 <https://oeis.org/A059395>
- [14] OEIS: A059452 <https://oeis.org/A059452>
- [15] OEIS: A075705 <https://oeis.org/A075705>
- [16] OEIS: A075706 <https://oeis.org/A075706>
- [17] OEIS: A075707 <https://oeis.org/A075707>
- [18] OEIS: A227853 <https://oeis.org/A227853>
- [19] OEIS: A014233 <https://oeis.org/A014233>
- [20] cppreference: https://en.cppreference.com/w/cpp/standard_library
- [21] cppreference: <https://en.cppreference.com/w/cpp/numeric/random>
- [22] RFC 2630 <https://datatracker.ietf.org/doc/html/rfc2630>
- b. Diễn đàn: Wolfram MathWorld, Wikipedia, GeeksforGeeks, Omni Calculator, Linkedin, CryptoBook.Nakov, PlanetMath
- [23] Wolfram MathWorld: <https://mathworld.wolfram.com/Diffie-HellmanProtocol.html>
- [24] Wikipedia: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
- [25] GeeksforGeeks: <https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/>
- [26] GeeksforGeeks: <https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/>
- [27] GeeksforGeeks: <https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/>
- [28] GeeksforGeeks: <https://www.geeksforgeeks.org/introduction-to-primality-test-and-school-method/>

- [29] Omni Calculator: <https://www.omnicalculator.com/math/power-modulo>
- [30] Wikipedia: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
- [31] Wikipedia: https://en.wikipedia.org/wiki/Modular_exponentiation
- [32] LinkedIn: <https://www.linkedin.com/pulse/cryptographically-secure-pseudo-random-number-csprng-ainapurapu-vfwbc/>
- [33] CryptoBook.Nakov: <https://cryptobook.nakov.com/secure-random-generators/secure-random-generators-csprng>
- [34] PlanetMath: <https://planetmath.org/SafePrime>

c. Tài liệu tham khảo:

- [35] An Introduction to Mathematical Cryptography Book.
- [36] X942: "Agreement Of Symmetric Keys Using Diffie-Hellman and MQV Algorithms", ANSI draft, 198.
- [37] FIPS-186: Federal Information Processing Standards Publication (FIPS PUB) 186, "Digital Signature Standard", 1994 May 19.
- [38] PKIX: Housley, R., Ford, W., Polk, W. and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", [RFC 2459](#), January 1999.

Table of Contents

1	Tổng quan mã nguồn chương trình dự án	9
1.1	Lớp dữ liệu (BigInt512).....	9
1.1.1	Thuộc tính	9
1.1.2	Tính toán và xử lý.....	9
1.2	Hàm tính lũy thừa mô-đun	13
1.3	Các hàm sinh số nguyên tố ngẫu nhiên	13
1.3.1	Các hàm kiểm tra số nguyên tố	13
1.3.2	Hàm sinh số nguyên tố an toàn	13
1.4	Hàm sinh khóa riêng ngẫu nhiên	14
1.5	Trao đổi khóa Diffie-Hellman	15
2	Thuật toán Diffie-Hellman [IETF: RFC 2631].....	16
2.1	Tổng quan.....	16
2.2	Cơ sở thuật toán.....	16
2.2.1	Khóa chia sẻ bí mật chung.....	16
2.2.2	Tính toán KEK	17
2.2.3	Độ dài khóa cho các thuật toán phổ biến	17
2.2.4	Xác thực khóa công khai	17
2.3	Yêu cầu về khóa và tham số	18
2.3.1	Tham số nhóm.....	18
2.3.2	Xác thực tham số nhóm	20
3	Phép thử kiểm tra số nguyên tố Miller-Rabin.....	21
3.1	Cơ sở toán học	21
3.1.1	Phép thử xác suất.....	21
3.1.2	Thuật toán đơn định	22
3.2	Thuật toán Miller – Rabin.....	22
3.2.1	Tổng quát.....	22
3.2.2	Thuật toán	22
3.2.3	Độ phức tạp.....	23
4	Tổng kết tự đánh giá chương trình dự án	24
4.1	Phân tích yêu cầu	24
4.1.1	Độ phức tạp chương trình	24

4.1.2	Thử nghiệm chương trình	24
4.2	Hiệu suất.....	28
4.2.1	Không gian.....	28
4.2.2	Thời gian.....	28

1 Tổng quan mã nguồn chương trình dự án

1.1 Lớp dữ liệu (BigInt512)

1.1.1 Thuộc tính

- `Uint_t parts[8]`: Mảng 8 phần tử `uint64_t` để lưu trữ số 512 bit
 - Mỗi phần tử lưu trữ 64 bit
 - Tổng dung lượng: $8 \times 64 = 512$ bit

1.1.2 Tính toán và xử lý

1.1.2.1 Khởi tạo

- Constructor mặc định: `BigInt512()`: Khởi tạo tất cả các bit = 0
- Constructor từ số nguyên: `BigInt512(uint64_t num)`: Lưu vào phần tử đầu tiên
- Constructor từ chuỗi hex: `BigInt512(const std::string& hexStr)`: Chuyển đổi từng cặp ký tự hex thành bits

1.1.2.2 Toán tử

1.1.2.2.1 Phép tính cơ bản

- Cộng (+): Cộng từng phần với xử lý carry
- Trừ (-): Trừ từng phần với xử lý borrow
- Nhân (*): Nhân từng phần sử dụng `__int128`
- Chia (/): Dùng phương pháp dịch bit và trừ

1.1.2.2.2 Phép tính nâng cao

- Chia lấy dư (%): Sử dụng công thức $a = (a/b)*b + a\%b$

1.1.2.2.3 Phép gán

- Gán AND (&=): Gán kết quả phép AND vào đối tượng hiện tại

1.1.2.2.4 Phép so sánh

- Bằng (==): So sánh từng phần
- Khác (!=): So sánh từng phần
- Lớn hơn (>): So sánh từ phần cao nhất
- Lớn hơn hoặc bằng (>=): So sánh từ phần cao nhất
- Nhỏ hơn hoặc bằng (<=): So sánh từ phần cao nhất

1.1.2.2.5 Phép tính xử lý nhị phân

- AND (&): Thực hiện AND bit từng phần
- Dịch trái (<<): Dịch bit sang trái với xử lý overflow
- Dịch phải (>>): Dịch bit sang phải với xử lý underflow

1.1.2.3 Hàm thao tác

1.1.2.3.1 Lấy độ dài chuỗi

- `get_bit_length()`:
 - Duyệt từ phần cao nhất (7) xuống thấp nhất (0)
 - Khi tìm thấy phần khác 0, tìm bit 1 cao nhất trong phần đó
 - Trả về độ dài = (vị trí phần $\times 64$) + vị trí bit + 1
 - Nếu không tìm thấy phần khác 0, trả về 0Pseudo code:

FUNCTION `get_bit_length()` RETURNS INTEGER

// Duyệt từ phần tử cao nhất xuống thấp nhất

FOR $i = 7$ TO 0 STEP -1

IF `parts[i]` $\neq 0$ THEN

// Tìm vị trí bit 1 cao nhất trong phần này

`bit_pos` = 63

`part` = `parts[i]`

// Duyệt từng bit từ trái sang phải

WHILE `bit_pos` ≥ 0 AND NOT(`part` AND ($1 \ll \text{bit_pos}$)) DO

`bit_pos` = `bit_pos` - 1

END WHILE

// Tính tổng độ dài bit

RETURN $i * 64 + \text{bit_pos} + 1$

END IF

END FOR

// Trường hợp số = 0

RETURN 0

END FUNCTION

1.1.2.3.2 Sinh số ngẫu nhiên

- generate_random_number(int bits):
 - Sử dụng nhiều nguồn entropy để tạo seeds
 - Khởi tạo Mersenne Twister generator với seeds
 - Tạo số ngẫu nhiên trong khoảng [0, 5000] cho phần đầu
 - Các phần còn lại đặt = 0
 - Trả về số ngẫu nhiên dạng BigInt512
 - Pseudo code:

FUNCTION generate_random_number(bits) RETURNS BigInt512

// Khởi tạo kết quả rỗng

result = new BigInt512()

// Khởi tạo các nguồn entropy

random_device = GET_RANDOM_DEVICE()

time_seed = GET_CURRENT_TIME_MICROSECONDS()

thread_id = GET_CURRENT_THREAD_ID()

stack_address = GET_CURRENT_STACK_ADDRESS()

// Tạo mảng seeds từ các nguồn entropy

seeds = [

random_device.generate() \times 7 times,

```

        time_seed & 0xFFFFFFFF,
        time_seed >> 32,
        HASH(thread_id),
        HASH(stack_address)
    ]

    // Khởi tạo generator với seeds
    generator = NEW_MERSENNE_TWISTER(seeds)

    // Tạo phân phối đều trong khoảng [0, 5000]
    distribution = NEW_UNIFORM_DISTRIBUTION(0, 5000)

    // Chỉ điền vào phần tử đầu tiên
    result.parts[0] = distribution.generate(generator)

    // Đặt các phần còn lại về 0
    FOR i = 1 TO 7 DO
        result.parts[i] = 0
    END FOR

    RETURN result
END FUNCTION

```

1.2 Hàm tính lũy thừa mô-đun

Trong mã nguồn chương trình dự án, hàm tính lũy thừa mô-đun là hàm `modular_exponentiation` có 3 tham số đầu vào là `base`, `exponent`, và `modulus` có kiểu lớp dữ liệu `BigInt512` có kết quả trả về kiểu lớp dữ liệu `BigInt512`.

Cách hoạt động của mã nguồn chương trình, cũng như những thay đổi về tham số của mã nguồn chương trình đều được dựa trên nguyên lý của thuật toán đã được nêu ở Mục 1.2 Thuật toán, Lũy thừa Mô-đun, Tài liệu báo cáo IC G14 PJRP-002 có trong Bộ tài liệu liên quan đến Dự án Lab01 này.

1.3 Các hàm sinh số nguyên tố ngẫu nhiên

1.3.1 Các hàm kiểm tra số nguyên tố

Trong mã nguồn chương trình dự án, các hàm thực hiện kiểm tra số nguyên tố dựa trên phép thử kiểm tra số nguyên tố Miller-Rabin là hàm `millertest` và hàm `isPrime`.

Về nguyên lý hoạt động chung của hai hàm này, dựa trên Thuật toán Miller-Rabin được nêu tại [Mục 3.2.2 Thuật toán, Thuật toán Miller-Rabin, Phép thử kiểm tra số nguyên tố Miller-Rabin, IC G14 PJRP-003, Lab01.](#)

1.3.2 Hàm sinh số nguyên tố an toàn

Trong mã nguồn chương trình dự án, hàm sinh số nguyên tố an toàn là hàm `generate_safe_prime` có tham số đầu vào cố định là `bit_size` chỉ kích thước bit của số lớn cần thao tác và cho kết quả trả về là kiểu lớp dữ liệu `BigInt512`.

Nguyên lý hoạt động của hàm sinh số nguyên tố an toàn:

1. Khởi tạo danh sách những số nguyên tố đầu tiên (ví dụ: 2, 3, 5, 7,...)
2. Xác định số phép thử tối đa cần thiết trong phạm vi mà hệ thống thực thi được lý tưởng.
Trong mã nguồn chương trình dự án, số lần phép thử tối đa mặc định là 5.000.000
3. Thực hiện vòng lặp để thực hiện lần lượt từng phép thử cho đến khi cho ra kết quả mong muốn là một số nguyên tố an toàn.
4. Thực hiện từng lần thử trong vòng lặp 3., khởi tạo giá trị p, q ban đầu, p = 0, q là được khởi tạo ngẫu nhiên. Xét điều kiện q là số lẻ và kiểm tra đơn giản q có là số nguyên tố hay không? Nếu q là không là số nguyên tố, thực hiện lại Bước 4.
5. Kiểm tra q có là số nguyên tố hay không bằng hàm kiểm tra số nguyên tố isPrime được nêu có trong [Mục 1.3.1 IC G14 PJRP-003](#). Nếu q không là số nguyên tố, quay lại Bước 4.
6. Tính toán: $p = q * 2 + 1$
7. Thực hiện kiểm tra p cũng phải là số nguyên tố hay không? [Mục 1.3.1, IC G14 PJRP-003](#)
8. Nếu p cũng là số nguyên tố thì hàm trả về kết quả là số p cần tìm và kết thúc thuật toán.
Ngược lại, quay lại Bước 4.

1.4 Hàm sinh khóa riêng ngẫu nhiên

Trong mã nguồn chương trình dự án, hàm sinh khóa riêng ngẫu nhiên là hàm **generate_private_key** có tham số đầu vào là số nguyên tố lớn p và cho kết quả trả về là khóa riêng có kiểu lớp dữ liệu BigInt512.

Hàm **generate_private_key** tính toán khóa riêng theo công thức

$$private_key = (Random \% (p - 4)) + 2$$

,tức là *private_key* có giá trị trong khoảng $[2, p - 2]$

1.5 Trao đổi khóa Diffie-Hellman

Trao đổi khóa Diffie-Hellman trao đổi giữa hai bên là Alice và Bob (thử nghiệm) được thực hiện trong hàm main() của mã nguồn chương trình dự án Lab01.

Trình tự thực hiện:

1. Sinh số nguyên tố lớn p và phần tử sinh g
 - Sinh số nguyên tố lớn p bằng hàm `generate_safe_prime` đã nêu ở [Mục 1.3.2](#)
 - Đối với lựa chọn phần tử sinh g phù hợp, ở mã nguồn chương trình dự án lựa chọn giá trị $g = 2$, sau khi tham khảo từ RFC 3526 và Mục 6 RFC 2409 nêu rõ giải thích vì sao nên chọn $g = 2$. Xem thêm thuật toán tính toán giá trị phần tử sinh g tại [Mục 2.3.1.2](#).
2. Sinh khóa riêng của Alice và Bob, xem thêm tại [Mục 1.4](#).
3. Tính giá trị công khai của Alice và Bob, xem thêm tại Mục 1.1 Tổng quan, IC G14 PJRP-001.
4. Tính bí mật chung, xem thêm tại Mục 1.1 Tổng quan, IC G14 PJRP-001.
5. Hiển thị kết quả và xác minh rằng bí mật chung trùng khớp.
6. Kết thúc quá trình trao đổi khóa Diffie-Hellman.

2 Thuật toán Diffie-Hellman [IETF: RFC 2631]

2.1 Tổng quan

Thỏa thuận khóa Diffie-Hellman yêu cầu cả người gửi và người nhận tin nhắn đều phải có cặp khóa. Bằng cách kết hợp khóa riêng của một bên và khóa công khai của bên kia, cả hai bên đều có thể tính toán cùng một số bí mật được chia sẻ. Sau đó, số này có thể được chuyển đổi thành vật liệu khóa mật mã. Vật liệu khóa đó thường được sử dụng làm khóa mã hóa khóa (KEK) để mã hóa (gói) khóa mã hóa nội dung (CEK) sau đó được sử dụng để mã hóa dữ liệu tin nhắn.

2.2 Cơ sở thuật toán

Giai đoạn đầu tiên của quá trình thỏa thuận khóa là tính toán một số bí mật được chia sẻ, được gọi là ZZ. Khi cùng một cặp khóa công khai/riêng tư của người khởi tạo và người nhận được sử dụng, giá trị ZZ giống nhau sẽ được tạo ra. Sau đó, giá trị ZZ được chuyển đổi thành khóa mật mã đối xứng được chia sẻ. Khi người khởi tạo sử dụng cặp khóa công khai/riêng tư tĩnh, việc đưa vào giá trị ngẫu nhiên công khai đảm bảo rằng khóa đối xứng kết quả sẽ khác nhau cho mỗi thỏa thuận khóa.

2.2.1 Khóa chia sẻ bí mật chung

X9.42 định nghĩa rằng bí mật chung ZZ được tạo ra như sau:

$$ZZ = g^{(x_b * x_a)} \bmod p$$

Lưu ý rằng các bên riêng lẻ thực sự thực hiện các phép tính:

$$ZZ = (y_b^{x_a}) \bmod p = (y_a^{x_b}) \bmod p$$

trong đó ^ biểu thị phép mũ

y_a là khóa công khai của bên a; $y_a = g^{x_a} \bmod p$

y_b là khóa công khai của bên b; $y_b = g^{x_b} \bmod p$

x_a là khóa riêng của bên a

x_b là khóa riêng của bên b

p là số nguyên tố lớn

q là số nguyên tố lớn

$g = h^{\{(p-1)/q\}} \bmod p$, trong đó

h là bất kỳ số nguyên nào có $1 < h < p-1$ sao cho $h^{\{(p-1)/q\}} \bmod p > 1$

(g có bậc $q \bmod p$; tức là $g^q \bmod p = 1$ nếu $g \neq 1$)

j là số nguyên lớn sao cho $p = qj + 1$

(Xem [Mục 2.3](#) để biết tiêu chí cho khóa và tham số)

2.2.2 Tính toán KEK

Mỗi thuật toán mã hóa khóa yêu cầu một khóa có kích thước cụ thể (n). KEK được tạo ra bằng cách ánh xạ n -byte bên trái nhất của KM vào khóa. Đối với 3DES, yêu cầu 192 bit vật liệu khóa, thuật toán phải được chạy hai lần, một lần với giá trị bộ đếm là 1 (để tạo $K1'$, $K2'$ và 32 bit đầu tiên của $K3'$) và một lần với giá trị bộ đếm là 2 (để tạo 32 bit cuối cùng của $K3$). $K1'$, $K2'$ và $K3'$ sau đó được điều chỉnh chẵn lẻ để tạo 3 khóa DES $K1$, $K2$ và $K3$. Đối với RC2-128, yêu cầu 128 bit vật liệu khóa, thuật toán được chạy một lần, với giá trị bộ đếm là 1 và 128 bit ngoài cùng bên trái được chuyển đổi trực tiếp thành khóa RC2. Tương tự, đối với RC2-40, yêu cầu 40 bit vật liệu khóa, thuật toán được chạy một lần, với giá trị bộ đếm là 1 và 40 bit ngoài cùng bên trái được sử dụng làm khóa.

2.2.3 Độ dài khóa cho các thuật toán phổ biến

Một số thuật toán mã hóa khóa phổ biến có KEK có độ dài sau đây

3 khóa 3DES	192 bit
RC2-128	128 bit
RC2-40	40 bit

Độ dài khóa hiệu dụng RC2 bằng độ dài khóa thực RC2.

2.2.4 Xác thực khóa công khai

Thuật toán sau CÓ THỂ được sử dụng để xác thực khóa công khai đã nhận y.

1. Xác minh rằng y nằm trong khoảng $[2, p-1]$. Nếu không, khóa không hợp lệ.
2. Tính $y^q \bmod p$. Nếu kết quả $== 1$, khóa hợp lệ. Nếu không, khóa không hợp lệ.

Mục đích chính của xác thực khóa công khai là ngăn chặn một cuộc tấn công nhóm nhỏ vào cặp khóa của người gửi. Nếu chế độ Ephemeral-Static được sử dụng, kiểm tra này có thể không cần thiết. Xem thêm [IEEE: P1363] để biết thêm thông tin về xác thực Khóa công khai.

Lưu ý rằng quy trình này có thể phải tuân theo các bằng sáng chế đang chờ cấp.

2.3 Yêu cầu về khóa và tham số

X9.42 yêu cầu các tham số nhóm phải có dạng $p = jq + 1$ trong đó q là số nguyên tố lớn có độ dài m và $j \geq 2$. Thuật toán để tạo số nguyên tố có dạng này (được suy ra từ các thuật toán trong FIPS PUB 186-1 [FIPS-186] và [X942]) có thể được tìm thấy trong phụ lục A.

X9.42 yêu cầu khóa riêng x phải nằm trong khoảng $[2, (q - 2)]$. x phải được tạo ngẫu nhiên trong khoảng này. Sau đó, y được tính bằng cách tính $g^x \bmod p$. Để tuân thủ bản ghi nhớ này, m PHẢI có độ dài ≥ 160 bit, (do đó, q PHẢI dài ít nhất 160 bit). Khi sử dụng mã hóa đối xứng mạnh hơn DES, có thể nên sử dụng m lớn hơn. p phải dài tối thiểu 512 bit.

2.3.1 Tham số nhóm

2.3.1.1 Tham số p, q

Thuật toán này tạo ra một cặp p, q trong đó q có độ dài m và p có độ dài L .

1. Đặt $m' = m/160$ trong đó $/$ biểu diễn phép chia số nguyên làm tròn lên. Tức là $200/160 = 2$.
2. Đặt $L' = L/160$
3. Đặt $N' = L/1024$
4. Chọn một chuỗi bit SEED tùy ý sao cho độ dài của SEED $\geq m$
5. Đặt $U = 0$
6. Đối với $i = 0$ đến $m' - 1$

$$U = U + (\text{SHA1}[\text{SEED} + i] \text{ XOR } \text{SHA1}[(\text{SEED} + m' + i)]) * 2^{(160 * i)}$$

Lưu ý rằng đối với $m=160$, điều này sẽ giảm xuống thuật toán [FIPS-186]

$U = \text{SHA1}[\text{SEED}] \text{ XOR } \text{SHA1}[(\text{SEED}+1) \bmod 2^{160}]$.

5. Tạo q từ U bằng cách tính $U \bmod (2^m)$ và đặt bit có ý nghĩa nhất (bit $2^{(m-1)}$) và bit có ý nghĩa nhỏ nhất thành 1. Về mặt phép toán boolean, $q = U \text{ HOẶC } 2^{(m-1)} \text{ HOẶC } 1$. Lưu ý rằng $2^{(m-1)} < q < 2^m$

6. Sử dụng thuật toán tính nguyên tố mạnh mẽ để kiểm tra xem q có phải là số nguyên tố hay không.

7. Nếu q không phải là số nguyên tố thì chuyển đến 4.

8. Cho $\text{counter} = 0$

9. Đặt $R = \text{seed} + 2^m + (L' * \text{counter})$

10. Đặt $V = 0$

12. Đối với $i = 0$ đến $L'-1$ thì

$V = V + \text{SHA1}(R + i) * 2^{(160 * i)}$

13. Đặt $W = V \bmod 2^L$

14. Đặt $X = W \text{ OR } 2^{(L-1)}$

Lưu ý rằng $0 \leq W < 2^{(L-1)}$ và do đó $X \geq 2^{(L-1)}$

15. Đặt $p = X - (X \bmod (2*q)) + 1$

6. Nếu $p > 2^{(L-1)}$ sử dụng một kiểm tra tính nguyên tố để kiểm tra xem p có phải là số nguyên tố không. Nếu không, hãy chuyển đến 18.

17. Nếu p là số nguyên tố, hãy xuất p , q , hạt giống, bộ đếm và dừng.

18. Đặt bộ đếm = bộ đếm + 1

19. Nếu bộ đếm $< (4096 * N)$ thì chuyển đến 8.

20. Xuất "thất bại"

Lưu ý: Một bài kiểm tra tính nguyên tố mạnh mẽ là bài kiểm tra mà xác suất một số không phải là số nguyên tố vượt qua bài kiểm tra là nhiều nhất 2^{-80} . [FIPS-186] cung cấp một thuật toán phù hợp, cũng như [X942].

2.3.1.2 Tham số g

Phần này đưa ra một thuật toán (được lấy từ [FIPS-186]) để tạo g.

1. Cho $j = (p - 1)/q$.
2. Đặt h = bất kỳ số nguyên nào, trong đó $1 < h < p - 1$ và h khác so với bất kỳ giá trị nào đã thử trước đó.
3. Đặt $g = h^j \bmod p$
4. Nếu $g = 1$, hãy chuyển sang bước 2

2.3.2 Xác thực tham số nhóm

ASN.1 cho khóa DH trong [PKIX] bao gồm các phần tử j và các Parm xác thực CÓ THỂ được người nhận khóa sử dụng để xác minh rằng các tham số nhóm đã được tạo đúng. Có thể thực hiện hai lần kiểm tra:

1. Xác minh rằng $p = qj + 1$. Điều này chứng minh rằng các tham số đáp ứng tiêu chí tham số X9.42.
2. Xác minh rằng khi quy trình tạo p,q của [FIPS-186] Phụ lục 2 được theo sau bằng hạt giống 'seed', thì p được tìm thấy khi 'counter' = pgenCounter.

Điều này chứng minh rằng các tham số được chọn ngẫu nhiên và không có dạng đặc biệt.

Việc các tác nhân có cung cấp thông tin xác thực trong chứng chỉ của họ hay không là vấn đề cục bộ giữa các tác nhân và CA của họ.

3 Phép thử kiểm tra số nguyên tố Miller-Rabin

3.1 Cơ sở toán học

Thuật toán Rabin-Miller là phiên bản mở rộng và mạnh hơn của phép thử Fermat. Thuật toán dựa vào nhận xét sau:

“Với mọi số nguyên dương x , ta tìm được duy nhất hai số tự nhiên k, m sao cho $x = 2^k * m$ và m lẻ.”

Do đó, xét số n , ta có thể phân tích $n - 1$ thành $2^k * m$, với m là số lẻ.

Theo định lý Fermat nhỏ, nếu n là số nguyên tố thì với mọi a sao cho $\gcd(a, n) = 1$, ta có:

$$a^{n-1} \equiv 1 \pmod{n} \Leftrightarrow a^{2^k \cdot m} - 1 \equiv 0 \pmod{n} \Leftrightarrow (a^{2^{k-1}m} + 1)(a^{2^{k-2}m} + 1) \dots (a^m + 1)(a^m - 1) \equiv 0 \pmod{n}$$

Vì là số nguyên tố nên tồn tại ít nhất một trong các nhân tử của vế trái chia hết cho. Do đó, thay vì kiểm tra kết luận của định lý Fermat nhỏ, ta sẽ kiểm tra điều kiện sau:

- $a^m \equiv 1 \pmod{n}$
- $\exists l, 0 \leq l \leq k-1, \text{ sao cho } a^{2^l m} \equiv -1 \pmod{n}$

Nếu cả 2 điều kiện không được thỏa mãn thì chắc chắn n là hợp số.

Nhưng nếu cả hai điều kiện được thỏa mãn thì có phải số nguyên tố hay không ?

3.1.1 Phép thử xác suất

Để tăng tính chính xác của thuật toán ta có thể lặp lại bước kiểm tra với nhiều cơ số a , giống như phép thử Fermat. Hơn thế nữa, chứng minh được nếu n là hợp số, chỉ có $\approx 25\%$ số cơ số a trong đoạn $[2, n-1]$ thỏa mãn một trong hai điều kiện.

Nghĩa là với hợp số n bất kỳ, xác suất để thuật toán chứng minh được n là hợp số sau lần kiểm tra đầu tiên là $\geq 75\%$, lần thứ hai là $\geq 93,75\%$, lần thứ ba là $\geq 98,43\%$, lần thứ x là $\left(1 - \frac{1}{4^x}\right) * 100\%$.

Vì thế, để độ chính xác của thuật toán Miller-Rabin được chính xác nhất ta phải lấy số lần xác minh đủ lớn trong phạm vi bài toán cho phép.

Ta có thể thấy độ chính xác của thuật toán Miller-Rabin cao hơn nhiều so với phép thử Fermat.

3.1.2 Thuật toán đơn định

Phép thử xác suất có thể trở thành thuật toán bằng cách thay vì xét ngẫu nhiên, ta sẽ xét tất cả bị chặn bởi một hàm theo. Miller chứng minh được nếu Định đề Riemann tổng quát (GRH) là đúng thì ta chỉ cần kiểm tra $a \in [2, O(\ln^2 n)]$. Sau đó, Bach chứng minh được chỉ cần xét $a \in [2, 2 \ln^2 n]$. Với n đủ lớn thì vẫn có rất nhiều giá trị cần kiểm tra. Nhưng người ta cũng chứng minh được rằng.

- ❖ Nếu $n \leq 3 \cdot 10^9$, thì chỉ cần xét $a \in \{2; 3; 5; 7\}$.
- ❖ Nếu $n \leq 2^{64}$, thì chỉ cần xét $a \in \{2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37\}$

Thuật toán Miller-Rabin: Xem thêm tại nguồn tham khảo [29].

3.2 Thuật toán Miller – Rabin

3.2.1 Tổng quát

Thuật toán Miller-Rabin có thể được triển khai như sau:

1. **Phân tích dạng số:** Với $n - 1 = 2^s \cdot d$ (trong đó d là số lẻ), kiểm tra các điều kiện trên $a^d \bmod n$ hoặc $a^{2^r \cdot d} \bmod n = n - 1$.
2. **Kiểm tra nhiều cơ sở a :** Nếu không có cơ sở nào làm cho n không phải là số nguyên tố, thì có thể chấp nhận n là nguyên tố với xác suất cao.

3.2.2 Thuật toán

3.2.2.1 Thuật toán kiểm thử xác suất số nguyên tố

Thuật toán trả về *false* nếu n là một hợp số và trả về *true* nếu n có thể là một số nguyên tố. k là tham số đầu vào xác định mức độ chính xác, mặc định là 10. Giá trị k càng cao thì độ chính xác càng cao.

bool isPrime(type_t n , type_t k)

1. Xử lý các trường hợp cơ sở cho $n < 3$
2. Nếu n chẵn, trả về *false*
3. Tìm số lẻ d sao cho $n - 1$ có thể viết được dưới dạng $2^r \cdot d$
Lưu ý rằng vì n là số lẻ nên $(n - 1)$ phải là số chẵn và r phải là lớn hơn 0
4. Thực hiện k lần sau
Nếu ($\text{millerTest}(n, d) = \text{false}$) thì
trả về *false*
5. Trả về *true*

3.2.2.2 Thuật toán phép thử Miller

Thuật toán phép thử được gọi cho tất cả k lần thử. Nó trả về *false* nếu n là hợp số và trả về *true* nếu n có khả năng là xuất sắc. d là một số lẻ sao cho $2^d r = n - 1$ với một số $r \geq 1$

bool millerTest (type_t n , type_t d)

1. Chọn ngẫu nhiên một số a trong phạm vi $[2, n - 2]$
2. Tính toán: $x = a^d \bmod n$
3. Nếu $x = 1$ hoặc $x = n - 1$ thì trả về *true*
4. Thực hiện vòng lặp chạy $r - 1$ lần
Thực hiện các bước sau khi d không trở thành $n - 1$.
 - a) $x = (x * x) \% n$
 - b) Nếu $(x = 1)$ thì trả về *false*
 - c) Nếu $(x = n - 1)$ thì trả về giá trị *true*

3.2.3 Độ phức tạp

Thuật toán Miller-Rabin có độ phức tạp là $O(k \cdot \log^3(n))$, trong đó k là số lần kiểm tra với các giá trị khác nhau của a . Độ phức tạp này đến từ việc phải tính nhiều phép lũy thừa mô-đun để xác minh các điều kiện của thuật toán. Việc chọn k càng lớn sẽ tăng độ chính xác, tuy nhiên sẽ tăng thêm thời gian tính toán.

Thuật toán Miller-Rabin rất hiệu quả khi áp dụng cho các số lớn và ít gặp phải các số giả nguyên tố như trong thuật toán Fermat.

4 Tổng kết tự đánh giá chương trình dự án

Lưu ý: Chương trình thử nghiệm được thực hiện trong điều kiện giới hạn các giá trị tính toán với số nguyên tố lớn nằm trong khoảng từ 0 đến 5000 để dễ kiểm thử với nguồn tài liệu tham khảo (Wolfram MathWorld và OEIS) về số nguyên tố an toàn trong quá trình thực hiện kiểm nghiệm.

4.1 Phân tích yêu cầu

4.1.1 Độ phức tạp chương trình

Hàm millerTest: $O(\log(n))$

Hàm isPrime: $O(\log(n)^2)$

Hàm generate_safe_prime: $O(\log(n)^2)$

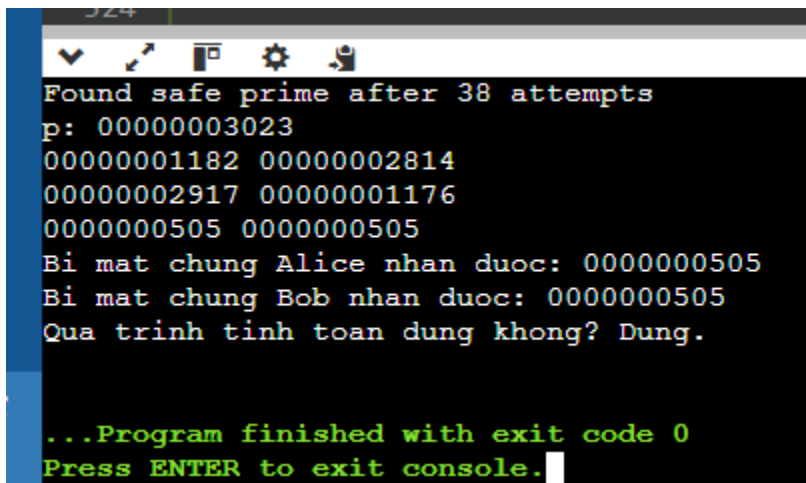
Hàm generate_private_key: $O(\log(n))$

Cả chương trình: $O(\log(n)^2)$

4.1.2 Thử nghiệm chương trình

Sau đây là tất cả 6 kết quả chạy thử nghiệm chương trình được thực hiện chạy trên môi trường lý tưởng (trong dự án này được thực hiện chạy thử nghiệm mặc định trên GDB online Debugger https://www.onlinegdb.com/online_c++_compiler và Visual Studio Code)

1. Thử nghiệm 1 (GDB online Debugger)

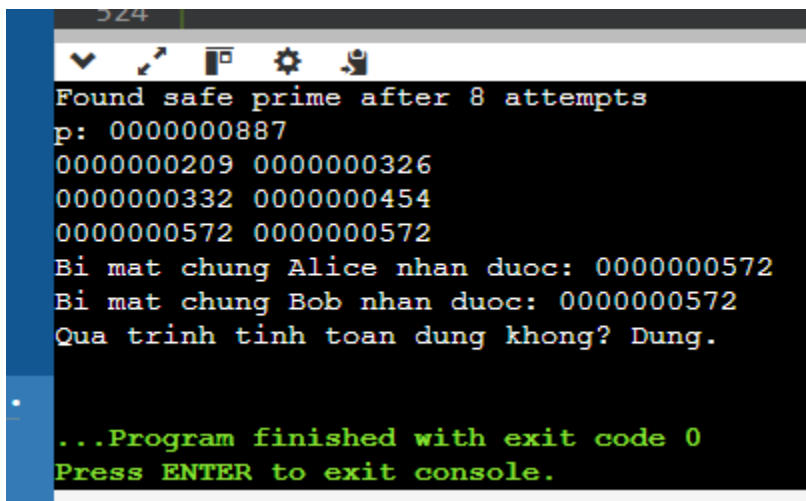


```
Found safe prime after 38 attempts
p: 00000003023
00000001182 00000002814
00000002917 00000001176
0000000505 0000000505
Bí mật chung Alice nhận được: 0000000505
Bí mật chung Bob nhận được: 0000000505
Qua trình tính toán đúng không? Đúng.

...Program finished with exit code 0
Press ENTER to exit console.
```

Giá trị $p = 3023$ khớp với nguồn tham khảo số nguyên tố an toàn OEIS: A075705 và OEIS: A075707.

2. Thử nghiệm 2 (GDB online Debugger)

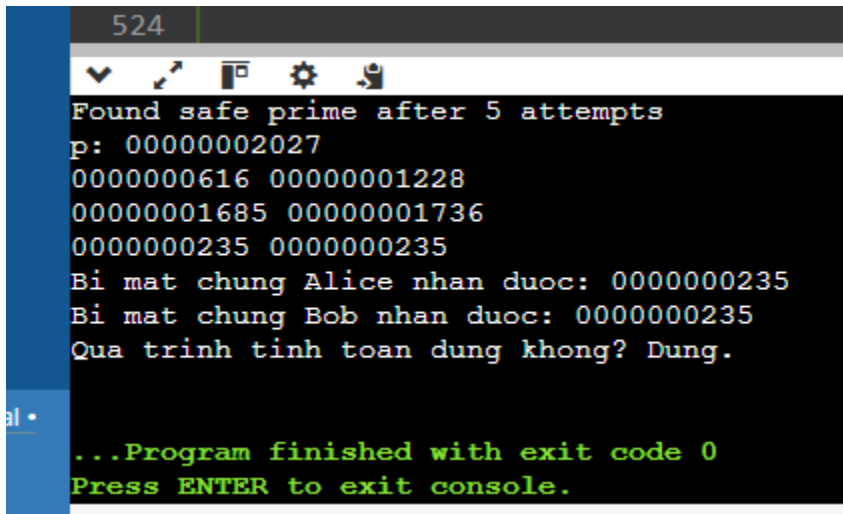


```
Found safe prime after 8 attempts
p: 0000000887
0000000209 0000000326
0000000332 0000000454
0000000572 0000000572
Bí mật chung Alice nhận được: 0000000572
Bí mật chung Bob nhận được: 0000000572
Qua trình tính toán đúng không? Đúng.

...Program finished with exit code 0
Press ENTER to exit console.
```

Giá trị $p = 887$ khớp với nguồn tham khảo số nguyên tố an toàn OEIS: A005385, OEIS: A059327, OEIS: A059452, OEIS: A075705.

3. Thử nghiệm 3 (GDB online Debugger)

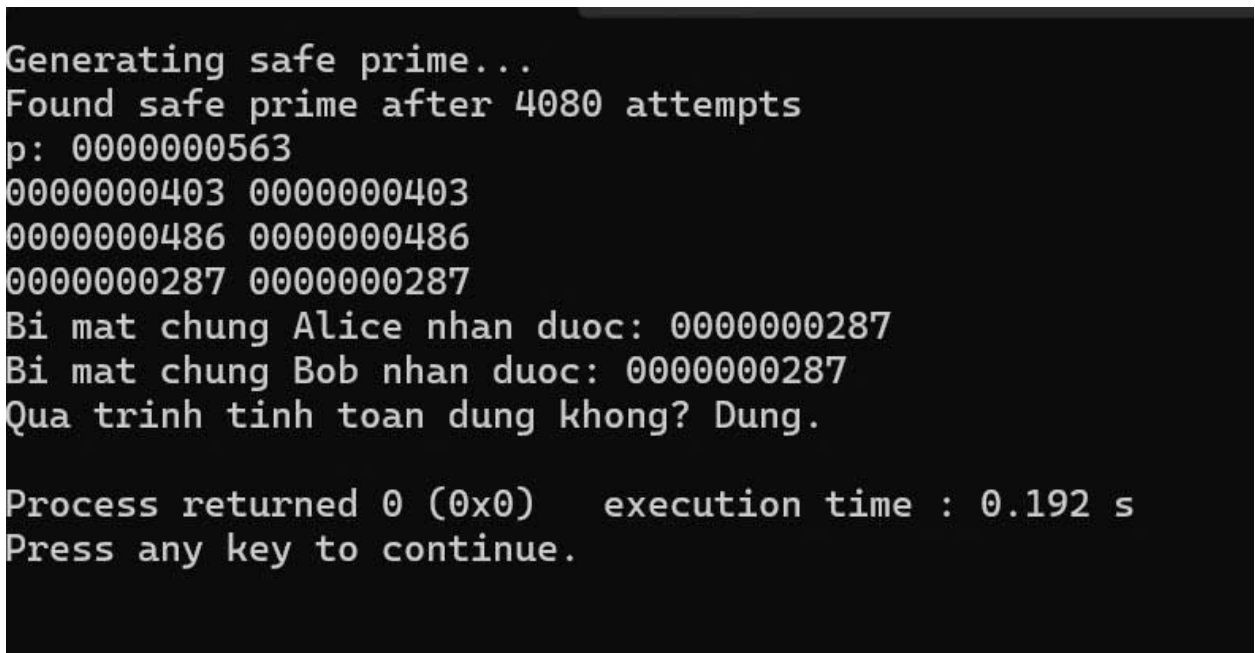
A screenshot of a GDB online Debugger console window. The window has a title bar with the number '524' and several icons. The console output is as follows:

```
Found safe prime after 5 attempts
p: 00000002027
0000000616 00000001228
00000001685 00000001736
0000000235 0000000235
Bí mật chung Alice nhận được: 0000000235
Bí mật chung Bob nhận được: 0000000235
Qua trình tính toán dừng không? Dừng.

...Program finished with exit code 0
Press ENTER to exit console.
```

Giá trị $p = 2027$ khớp với nguồn tham khảo số nguyên tố an toàn OEIS: A005385, OEIS: A059452, OEIS: A075705, OEIS: A075706.

4. Thử nghiệm 4 (Visual Studio Code)

A screenshot of a Visual Studio Code console window. The console output is as follows:

```
Generating safe prime...
Found safe prime after 4080 attempts
p: 0000000563
0000000403 0000000403
0000000486 0000000486
0000000287 0000000287
Bí mật chung Alice nhận được: 0000000287
Bí mật chung Bob nhận được: 0000000287
Qua trình tính toán dừng không? Dừng.

Process returned 0 (0x0)   execution time : 0.192 s
Press any key to continue.
```

Giá trị $p = 563$ khớp với nguồn tham khảo số nguyên tố an toàn OEIS: A005385, OEIS: A059452.

5. Thử nghiệm 5 (Visual Studio Code)

```
Generating safe prime...
Found safe prime after 1055 attempts
p: 00000001823
0000000243 0000000243
00000001158 00000001158
0000000200 0000000200
Bí mật chung Alice nhận được: 0000000200
Bí mật chung Bob nhận được: 0000000200
Qua trình tính toán đúng không? Đúng.

Process returned 0 (0x0)    execution time : 0.068 s
Press any key to continue.
|
```

Giá trị $p = 1823$ khớp với nguồn tham khảo số nguyên tố an toàn OEIS: A005385.

6. Thử nghiệm 6 (Visual Studio Code)

```
Generating safe prime...
Found safe prime after 1 attempts
p: 00000003947
000000053 000000053
00000001998 00000001998
0000000926 0000000926
Bí mật chung Alice nhận được: 0000000926
Bí mật chung Bob nhận được: 0000000926
Qua trình tính toán đúng không? Đúng.

Process returned 0 (0x0)    execution time : 0.015 s
Press any key to continue.
|
```

Giá trị $p = 3947$ khớp với nguồn tham khảo số nguyên tố an toàn OEIS: A059452

4.2 Hiệu suất

4.2.1 Không gian

Sử dụng kiểu lớp dữ liệu BigInt512 cho số nguyên lớn 512-bit thay thế cho chương trình mẫu cũ chạy kiểu dữ liệu int cho số nguyên lớn 8-bit. Mở rộng được không gian lưu trữ giá trị số nguyên lớn, đảm bảo được tính bảo mật cần thiết.

4.2.2 Thời gian

Đảm bảo được thời gian cần thiết để thực hiện trao đổi khóa Diffie-Hellman khi ứng dụng vào thực tiễn.

Xem xét thử nghiệm chương trình 4, 5, 6 tại Mục 4.1.2, cho thấy thời gian chạy chương trình trao đổi khóa rơi vào khoảng 0.015 đến 0.192s cho thử nghiệm chương trình ở quy mô nhỏ có giới hạn. Xem xét trong môi trường mạng thực tế, thời gian để mà thực hiện trao đổi khóa nếu quy mô lớn, mã nguồn chương trình dự án có thể có khả năng được đảm bảo.