**I. MÔ TẢ THUẬT TOÁN**

**1. Giới thiệu chung**

AES - Advanced Encryption Standard là một chuẩn mã hóa dữ liệu tiên tiến, với bản chất là thuật toán mã hóa khối đối xứng đã được chính phủ Mỹ Hoa Kỳ áp dụng làm tiêu chuẩn mã hóa.

Mã hóa khối đối xứng:

* Khối: Thuật toán áp dụng xử lý đơn vị dữ liệu là các khối (data blocks). Cụ thể, các khối dữ liệu được xử lý có độ dài 128 bits, con số này là cố định.
* Đối xứng: Cả bên gửi và bên nhận sử dụng chung một khóa cho cả 2 giai đoạn mã hóa và giải mã.

**2. Các ký hiệu và định nghĩa**

**2.1. Đầu vào và đầu ra của thuật toán**

Là các đoạn dữ liệu độ dài 128 bits, được nhắc đến như các block data. Những block này được mã hóa / giải mã bằng khóa có độ dài 128/192/256 bit, ứng với các chuẩn AES 128/192/256. Tuy nhiên block size luôn luôn là cố định.

**2.2. Các Byte**

Đơn vị xử lý cơ bản của thuật AES là các byte. Đầu vào, ra và khóa của thuật toán được xử lý như những chuỗi các byte.

Mỗi byte này sẽ được biểu diễn như sau: {*b7, b6, b5, b4, b3, b2, b1, b0*}. Chúng được hiểu như những phần tử hữu hạn trong trường Galois (Finite Field), biểu diễn bởi đa thức:

*b7x7* + *b6x6* + *b5x5* + *b4x4* + *b3x3* + *b2x2* + *b1x1* + *b0x0* (2.2.1)

VD: {00101011} ứng với đa thức: *x5* + *x3* + *x1* + *x0*

Ngoài ra, byte còn có thể được biểu diễn dưới dạng hexa.

VD: {01100011} ứng với phần tử {63}.

**2.3. Các trạng thái**

Các giai đoạn xử lý bên trong của thuật toán AES được thực hiện trên mảng 2 chiều các byte, được gọi là **State** (Trạng thái). Các mảng này có kích thước 4x4, được tạo nên như sau:

A close up of a clock

Description generated with high confidence

Dãy text đầu vào có thể được chuyển thành mã ASCII tương ứng, mỗi kí tự hexa tương ứng với 4 bit. Sau đó, cứ mỗi cặp kí tự hexa này được copy vào 1 phần tử của mảng trạng thái, điền vào theo thứ tự hàng dọc chứ không phải hàng ngang. Giả dụ ta có 1 chuỗi văn bản đầu vào như sau: Two One Nine Two

Dãy mã ASCII tương ứng:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | w | o |  | O | n | e |  | N | i | n | e |  | T | w | o |
| 54 | 77 | 6F | 20 | 4F | 6E | 65 | 20 | 4E | 69 | 6E | 65 | 20 | 54 | 77 | 6F |

Dãy này sẽ được chuyển vào State để xử lý:

|  |  |  |  |
| --- | --- | --- | --- |
| 54 | 4F | 4E | 20 |
| 77 | 6E | 69 | 54 |
| 6F | 65 | 6E | 77 |
| 20 | 20 | 65 | 6F |

Thuật toán AES thực hiện trên từng khối dữ liệu 16 bit riêng lẻ. Vậy 16 bit dữ liệu tiếp theo sẽ được nạp vào State để thực hiện xử lý sau.

**3. Các phép toán liên quan**

**3.1. Phép cộng**

Phép cộng các phần tử trong trường Galois được thực hiện bởi việc “cộng” các hệ số tương ứng của đa thức, cụ thể là XOR các bit tương ứng với nhau. Phép trừ cũng được thực hiện tương tự. Ví dụ như sau:

(*x6* + *x4* + *x2* + *x* + 1) + (*x7* + *x* + 1) = *x7* +*x6* + *x4* + *x2* (biểu diễn đa thức)

{01010111}⊕{10000011} = {11010100} (biểu diễn nhị phân)

{57}⊕{83} = {d4} (biểu diễn thập lục phân)

**3.2. Phép nhân**

Phép nhân trong trường Galois được thực hiện bởi việc nhân đa thức thông thường, theo sau bởi phép chia lấy dư (modulo) cho đa thức không giảm bậc 8. Trong khuôn khổ thuật AES, đa thức này là:

*m*(*x*) *= x8 +x4 + x3 + x +* 1

Hoặc {01}{1b} trong khuôn khổ biểu diễn hexadecimal.

Ví dụ, {57}•{83} = {c1} vì:

(*x6* + *x4* + *x2* + *x* + 1) (*x7* + *x* + 1) = *x13* + *x11* + *x9* + *x8* + *x7* +

*x7* + *x5* + *x3* + *x2* + *x* + *x6* +

*x6* + *x4* + *x2* + *x* + 1

= *x13* + *x11* + *x9* + *x8* + *x6* + *x5* +

*x4* + *x3* + 1

Và lại có

*x13* + *x11* + *x9* + *x8* + *x6* + *x5* + *x4* + *x3* + 1 modulo (*x8 +x4 + x3 + x +* 1)

= *x7* + *x6* + 1

Việc chia cho đa thức đảm bảo bậc của kết quả sẽ luôn nhỏ hơn 8, từ đó có thể biểu diễn thành byte.

**3.3. Nhân với**

Đa thức 2.2.1 sau khi nhân với x ta sẽ thu được kết quả:

*b7x8* + *b6x7* + *b5x6* + *b4x5* + *b3x4* + *b2x3* + *b1x2* + *b0x*

Dễ thấy kết quả này có bậc lớn hơn 8, ta cần phải giản ước nó. Nếu *b7 =* 0 thì kết quả đã ở dạng rút gọn. Nhưng nếu *b7 =* 1 thì giản ước bằng cách trừ đi đa thức *m*(*x*).

Tóm lại việc nhân đa thức với (vì vốn là bậc 1, nên tương ứng với {00000010} hoặc {02}) có thể thực hiện bằng phép dịch trái 1 bit, tùy điều kiện mà triển khai thêm phép XOR với {1b} (đa thức *m*(*x*)). Công đoạn này được gọi là xtime().

Từ cơ sở này, phép nhân với bậc mũ bất kì có thể được cài đặt bằng việc lặp lại các xtime() và thêm vào các kết quả trung gian. Ví dụ như {57}•{13} = {fe} vì:

{57}•{02} = xtime({57}) = {ae}

{57}•{04} = xtime({ae}) = {47}

{57}•{08} = xtime({47}) = {8e}

{57}•{10} = xtime({8e}) = {07}

Do đó:

{57}•{13} = {57}•({01}⊕{02}⊕{10})

= {57}⊕{ae}⊕{07}

= {fe}

Cụ thể, ta tách thừa số thứ 2 thành các bội số của 2, nhân với thừa số thứ nhất được các kết quả trung gian rồi cộng (XOR) với nhau. Bằng cách này phép nhân với số bất kì có thể được cài đặt một cách dễ dàng.

**4. Đặc tả thuật toán**

Bảng đặc tả các tham số của AES:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Key Length**  ***(Nk words)*** | **Block Size**  ***(Nb words)*** | **Number of Rounds**  ***(Nr)*** |
| **AES-128** | 4 | 4 | 10 |
| **AES-192** | 6 | 4 | 12 |
| **AES-256** | 8 | 4 | 14 |

Số liệu trong bảng ứng với 3 chuẩn AES - 128, 192 và 256. Cả 3 chuẩn đều có chung kích thước data block xử lý (***Nb***), chỉ khác ở ***Nk*** và ***Nr***. Thuật AES tiến hành theo các vòng, số lượng vòng tùy thuộc vào chuẩn tương ứng. Hàm vòng được tạo nên bởi các chuyển đổi byte: Byte Substitution, Shift Rows, Mix Columns và Add Round Keys. Các phép này đều tác động lên đối tượng là từng byte của State.

**4.1. Quá trình mã hóa (Cipher)**

Đầu quá trình mã hóa, dữ liệu đầu vào được copy vào mảng trạng thái theo mô tả trong sec 2.3. Quá trình mã hóa trải qua ***Nr*** bước tất cả, với ***Nr*** phụ thuộc vào từng chuẩn AES. Trong đó, vòng đầu và vòng cuối khác với những vòng còn lại.

Pseudo Code của The Cipher như sau:

Cipher(byte in[4\*Nb], byte out[4\*Nb], word w[Nb\*(Nr+1)])

begin

byte state[4,Nb]

state = in

AddRoundKey(state, w[0, Nb-1])

for round = 1 step 1 to Nr–1

SubBytes(state)

ShiftRows(state)

MixColumns(state)

AddRoundKey(state, w[round\*Nb, (round+1)\*Nb-1])

end for

SubBytes(state)

ShiftRows(state)

AddRoundKey(state, w[Nr\*Nb, (Nr+1)\*Nb-1])

out = state

end

2 tham số đầu tiên của Cipher là mảng trạng thái đầu vào và đầu ra của quá trình mã hóa. Tham số cuối cùng là mảng các **Roundkey w[]** với tổng số ***Nb*** × ***(Nr + 1)*** phần tử được tạo ra từ key schedule, sẽ được mô tả sau. Các function được dùng trong Cipher là **SubBytes()**, **ShiftRows()**, **MixColumns()**, **AddRoundKey()** đều tác động lên đối tượng đó là State.

**4.1.1. SubBytes() function**

Chuyển hóa **SubBytes()** là một phép thay thế thực hiện trên từng byte của State. Mỗi byte này sẽ được thay thế bằng 1 byte khác, phụ thuộc vào chính giá trị của byte được thay thế đó. Byte thay thế được lấy ra từ một bảng được gọi là S-box.

A screenshot of a cell phone

Description generated with high confidence

Thông qua **SubBytes()**, mỗi byte của state sẽ được thay thế bằng 1 byte lấy từ S-box này. Quá trình được mô tả trực quan như sau:



Cách thay thế: Nếu byte cần được thay thế của state có giá trị hexa là {mn} thì sẽ được thay thế với giá trị của phần tử nằm ở hàng m cột n. Ví dụ: Nếu *s0,1* có giá trị {c9} thì *s’0,1* sẽ là giá trị ở hàng c cột 9: {dd}.

**4.1.2. ShiftRows() Transformation**

Chuyển hóa của **ShiftRows()** chỉ là xoay vòng các byte tương ứng trong cùng một hàng của state, được mô tả như sau:



Chỉ có các byte của dòng đầu tiên là được giữ nguyên vị trí.

**4.1.3. MixColumns() Transformation**

Chuyển hóa **MixColumn()** áp dụng trên State theo từng cột, coi mỗi cột là các hệ số của một đa thức bậc 3. Đa thức này sẽ được nhân modulo với một đa thức cố định .

*a*(*x*) = {03}*x3* + {01}*x2* + {01}*x* + {02}

Kết quả này có thể được viết dưới dạng phép nhân ma trận. Giả sử:

*s’*(*x*) = *a*(*x*) ⊗ *s*(*x*)

Thì ta sẽ có:

với 0 ≤ c < ***Nb***

Kết quả của 4 byte hàng dọc sẽ là:

*s’0,c* = ({02}•*s0,c*) ⊕ ({03}•*s1,c*) ⊕ ­*s2,c* ⊕ *s3,c*

*s’1,c* = *s0,c* ⊕ ({02}•*s1,c*) ⊕ ({03}•*s2,c*) ⊕ ­*s3,c*

*s’2,c* = *s0,c* ⊕ ­*s1,c* ⊕ ({02}•*s2,c*) ⊕ ({03}•*s3,c*)

*s’3,c* = ({03}•*s0,c*) ⊕ *s1,c* ⊕ ­*s2,c* ⊕ ({02}•*s3,c*)

Như ta đã biết phép nhân với một số bất kì trong trường Galois có thể thực hiện bằng việc thêm các kết quả trung gian và lặp lại các xtime(). Vậy function này cũng có thể cài đặt một cách dễ dàng.

Mô hình diễn tả biến đổi **MixColumns()**:



**4.1.4. AddRoundKey() Transformation**

Trong biến đổi **AddRoundKey()**, một Round Key được cộng vào State bằng phép XOR đơn giản, trong đó mỗi Round Key độ dài ***Nb*** word sinh ra từ key schedule (giới thiệu sau):

[*s’0,c*, *s’1,c*, *s’2,c*, *s’3,c*] = [*s0,c*, *s1,c*, *s2,c*, *s4,c*] ⊕ [*wround\*Nb* + *c*] với 0 ≤ c < ***Nb***

[*wi*]là Round Key, round là thứ tự vòng mã hóa có giá trị 0 ≤ round ≤ ***Nr***.

Hình dưới diễn tả quá trình này, trong đó l = round × ***Nb***.

A close up of a clock

Description generated with high confidence

**AddRoundKey()** XOR mỗi cột của State với một word từ key schedule

**4.2. Key Expansion**

Thuật toán AES sử dụng khóa K (Khóa mã hóa) để thực hiện quá trình sinh khóa (Key Expansion). Quá trình này sinh ra tổng cộng ***Nb*** × (***Nr*** + 1) word: Thuật toán cần 4 word đầu, sau đó mỗi round trong tổng cộng ***Nr*** round lại cần ***Nb*** word dữ liệu khóa. Kết quả của quá trình này là một mảng các phần tử kích thước 4 byte, các [*wi*], với i nằm trong khoảng 0 ≤ i ≤ ***Nb*** × (***Nr*** + 1).

Pseudo-code của giải thuật như sau:

KeyExpansion(byte key[4\*Nk], word w[Nb\*(Nr+1)], Nk)

begin

word temp

i = 0

while (i < Nk)

w[i] = word(key[4\*i], key[4\*i+1], key[4\*i+2], key[4\*i+3])

i = i+1

end while

i = Nk

while (i < Nb \* (Nr+1)]

temp = w[i-1]

if (i mod Nk = 0)

temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]

else if (Nk > 6 and i mod Nk = 4)

temp = SubWord(temp)

end if

w[i] = w[i-Nk] xor temp

i = i + 1

end while

end

Hàm **SubWord()** lấy một word (4 byte) và thực hiện S-box với mỗi byte để tạo ra một word mới. Hàm **RotWord()** lấy một word [*a0*, *a1*, *a2*, *a3*] và biến đổi thành [*a1*, *a2*, *a3*, *a0*]. Hằng số vòng (round constant) **Rcon[i]** chứa các giá trị được cho bởi [*xi-1*, {00}, {00}, {00}] với *xi-1* là lũy thừa của *x* (*x* tương đương với {02}) trong trường GF(28).

Có thể thấy, ***Nk*** word đầu tiên của bảng khóa mở rộng được điền vào bởi khóa ban đầu. Mỗi word **w[i]** tiếp theo bằng XOR của word **w[i-1]** trước đó và word **w[i - *Nk*]**. Với những word ở vị trí là bội của ***Nk***, một phép biến đổi được áp dụng lên **w[i-1]** trước khi XOR, sau đó là XOR với **Rcon[i]**. Phép biến đổi bao gồm một phép biến đổi byte tuần hoàn trong một word (**RotWord()**), theo sau là áp dụng S-box lên tất cả 4 byte của word đó (**SubWord()**).

Ghi nhớ rằng quá trình Key Expansion cho AES 256-bit khác một chút so với AES 128-bit và 192-bit. Nếu ***Nk*** = 8 và i-4 là bội của ***Nk***, thì **SubWord()** được áp dụng vào **w[i-1]** trước khi XOR.

**4.4. Quá trình giải mã (Inverse Cipher):**

Quá trình Cipher có thể được đảo ngược và cải đặt theo thứ tự ngược lại nhằm tạo ra quá trình giải mã cho thuật toán AES. Những biến đổi riêng lẻ sử dụng trong Inverse Cipher - **InvShiftRows()**, **InvSubBytes()**, **InvMixColumns()**, và **AddRoundKey()** - thực hiện trên State và được miêu tả sau.

Pseudo Code của Inverse Cipher:

InvCipher(byte in[4\*Nb], byte out[4\*Nb], word w[Nb\*(Nr+1)])

begin

byte state[4,Nb]

state = in

AddRoundKey(state, w[Nr\*Nb, (Nr+1)\*Nb-1])

for round = Nr-1 step -1 downto 1

InvShiftRows(state)

InvSubBytes(state)

AddRoundKey(state, w[round\*Nb, (round+1)\*Nb-1])

InvMixColumns(state)

end for

InvShiftRows(state)

InvSubBytes(state)

AddRoundKey(state, w[0, Nb-1])

out = state

end

**4.4.1. InvShiftRows() Transformation**

**InvShiftRows()** là phép đảo ngược của biến đổi **ShiftRows()**. Tương tự như nguyên bản của nó, 3 dòng cuối cùng của State được đảo ngược theo vòng, với số offset khác nhau.



Công thức diễn giải của **InvShiftRows()** như sau:

*s’r,(c+shift(r,Nb))modNb = sr,c* với 0 < r < 4 và 0 ≤ c ≤ ***Nb***

**4.4.2. InvSubBytes() transformation**

**InvSubBytes()** tương tự như **SubBytes()**, chỉ khác ở chỗ byte thay thế được lấy từ Inverse S-box:

A receipt on a black background

Description generated with high confidence

**4.4.3. InvMixColumns() Transformation**

**InvMixColumns()** là phiên bản đảo ngược của **MixColumns()**, thao tác trên từng cột của State. Giả sử xét cột thứ c của State, ta có diễn giải như sau:

Với 0 ≤ c < ***Nb***

Kết quả của phép nhân này, 4 byte trong hàng được thay thế bằng:

*s’0,c* = ({0e} • *s0,c*) ⊕ ({0b} • *s1,c*) ⊕ ({0d} • *s2,c*) ⊕ ({09} • *s3,c*)

*s’1,c* = ({09} • *s0,c*) ⊕ ({0e} • *s1,c*) ⊕ ({0b} • *s2,c*) ⊕ ({0d} • *s3,c*)

*s’2,c* = ({0d} • *s0,c*) ⊕ ({09} • *s1,c*) ⊕ ({0e} • *s2,c*) ⊕ ({0b} • *s3,c*)

*s’3,c* = ({0b} • *s0,c*) ⊕ ({0d} • *s1,c*) ⊕ ({09} • *s2,c*) ⊕ ({0e} • *s3,c*)

**4.4.4. Đảo ngược của AddRoundKey() Transformation**

Phiên bản đảo ngược của **AddRoundKey()** là chính nó, bởi quá trình thực hiện chỉ gồm phép XOR.

**4.5. Mode of Operation**

Trong giải thuật mã hóa khối**, Mode of Operation** (chế độ thực hiện) diễn giải cách thực hiện lặp đi lặp lại quá trình mã hóa theo đơn vị từng khối, để truyền đi đơn vị dữ liệu lớn hơn khối một cách an toàn.

Hầu như tất cả các mode sử dụng một chuỗi nhị phân độc nhất, được gọi là vector khởi tạo (initialization vector - IV). IV phải không lặp lại, và, được khởi tạo ngẫu nhiên. Việc sử dụng IV đảm bảo rằng: Nếu một plaintext được encrypt nhiều lần với cùng một khóa bí mật, thì kết quả là khác nhau đối với mỗi lần.

IV không yêu cầu mức bảo mật như khóa, do đó IV không nhất thiết phải được giữ bí mật. Dù vậy, trong hầu hết các trường hợp, phải đảm bảo rằng một IV không bao giờ được dùng lại cho cùng một khóa.

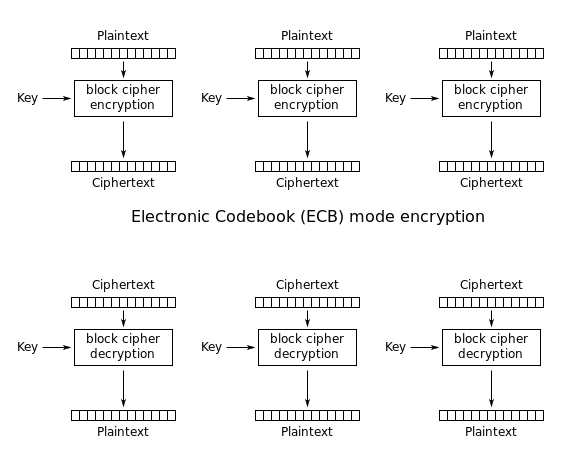
Một vài chế độ phổ biến đó là: ECB, CBC, PCBC, CFB, OFB, CTR. Mục đích của các chế độ này là làm ẩn (mask) các pattern xuất hiện trong dữ liệu đã được mã hóa, từ đó tăng tính bảo mật của ciphertext.

Các mode khác nhau có những cách mask dữ liệu cũng là khác nhau. Nhưng hầu hết đều có điểm chung là chúng trộn kết quả của lần cipher trước với đầu vào của lượt cipher sau. Ta có bảng mô tả chung các phương pháp:

A screenshot of a cell phone

Description generated with very high confidence

**4.5.1. Electronic Codebook (ECB)**

****

Là chế độ đơn giản nhất trong mã hóa khối. Chế độ này không sử dụng IV, thông điệp ban đầu được chia thành các đoạn nhỏ 16 bit để được xử lý độc lập. Chính vì không sử dụng IV nên đây cũng là chế độ tồn tại nhiều điểm yếu nhất, giả dụ như sự thiếu khả năng phân tán (lack of diffusion). Vì ECB xử lý các block dữ liệu giống nhau thành các ciphertext cũng là giống nhau, nên hoàn toàn thiếu đi khả năng che giấu data pattern. Giả dụ khi dùng ECB encrypt một bức ảnh, trong đầu ra ta sẽ thấy những vùng có màu giống nhau sẽ có cùng đặc điểm.

**Để cho đơn giản, trong khuôn khổ demo của bài tập này, chúng em sẽ sử dụng EBC Mode.**

**4.5.2. Cipher Block Chaining (CBC)**

Trong CBC mode, mỗi khối plaintext được XOR với đầu ra của khối trước đó trước khi được đưa vào encrypt. Điều này đảm bảo mỗi ciphertext block phụ thuộc vào tất cả plaintext đã được xử lý trước đó, vì vậy nên các đoạn text giống nhau vẫn sẽ cho kết quả khác nhau.

Để đảm bảo sự độc lập của các thông điệp, một IV phải được thêm vào ở mỗi block đầu các thông điệp đó.

A screenshot of a cell phone

Description generated with high confidence

Nếu block đầu có index 1, chế độ mã hóa này có thể được mô tả bằng công thức toán học:

**Ci = EK (Pi ⊕ Ci - 1),**

**Co = IV**

Trong khi công đoạn giãi mã là:

**Pi = DK (Ci) ⊕ C i-1,**

**Co = IV.**

(Pi Ci là plaintext và ciphertext thứ i tương ứng. DK và EK là decrypt và encrypt theo khóa K).

**II: DEMO GIẢI THUẬT**

**1: Tổng quan**

* Lý thuyết liên quan
  + Diffie-Hellman
    - Trao đổi khóa Diffie–Hellman (D-H) là một phương pháp trao đổi khóa được phát minh sớm nhất trong mật mã học. Phương pháp trao đổi khóa Diffie–Hellman cho phép hai bên (người, thực thể giao tiếp) thiết lập một khóa bí mật chung để mã hóa dữ liệu sử dụng trên kênh truyền thông không an toàn mà không cần có sự thỏa thuận trước về khóa bí mật giữa hai bên. Khóa bí mật tạo ra sẽ được sử dụng để mã hóa dữ liệu với phương pháp mã hóa khóa đối xứng.
  + MD5 Hashing
    - MD5 (viết tắt của tiếng Anh Message-Digest algorithm 5, giải thuật Tiêu hóa tin 5) là một hàm băm mật mã học được sử dụng phổ biến với giá trị Hash dài 128-bit. Là một chuẩn Internet (RFC 1321), MD5 đã được dùng trong nhiều ứng dụng bảo mật, và cũng được dùng phổ biến để kiểm tra tính toàn vẹn của tập tin. Một bảng băm MD5 thường được diễn tả bằng một số hệ thập lục phân 32 ký tự.
  + AES Encryption
* Thông tin kỹ thuật
  + Viết bằng ngôn ngữ C++, sử dụng thư viện open source kết hợp với class tự viết để xử lý phần mã hõa/giải mã

**2: Miêu tả demo**

* Demo được thiết kế dưới dạng server – client
* Server gửi các đoạn tin được mã hóa cho client giải mã
* Quá trình hoạt động
  + Server trao đổi bí mật chung với client bằng phương thức Diffie-Hellman
  + Server và client cùng băm MD5 bí mật chung để tạo khóa 128-bit phục vụ mã khóa/giải mã AES
  + Server gửi các đoạn tin được mã khóa AES cho client giải mã

**3: Chi tiết code**

AESCore.h

#include <iostream>

#include <string>

#include <GFCalc.h>

#include <Util.h>

using namespace std;

class simpleAES {

private:

//Side functions

static char sBox(char a) {

char b = GF::gfMulInv(a);

char B = 0x00;

for (int i = 7; i >= 0; --i) {

bool bi = Util::getKBit(b, i);

bi ^= Util::getKBit(0x63, i);

for (int j = 0; j < 4; ++j) {

int k = (i + 4 + j) % 8;

bi ^= Util::getKBit(b, k);

}

B |= (bi << i);

}

return B;

}

static char invSBox(char a) {

char A = 0x00;

a ^= 0x63;

for (int i = 7; i >= 0; --i) {

bool ai = Util::getKBit(a, (i + 7) % 8) ^

Util::getKBit(a, (i + 5) % 8) ^

Util::getKBit(a, (i + 2) % 8);

A |= (ai << i);

}

return GF::gfMulInv(A);

}

static void subWord(string &word) {

for (int i = 0; i < 4; ++i) {

word[i] = sBox(word[i]);

}

return;

}

static void rotWord(string &word) {

string tmp(4, '0');

for (int i = 0; i < 4; ++i) {

tmp[i] = word[(i + 1) % 4];

}

for (int i = 0; i < 4; ++i) {

word[i] = tmp[i];

}

return;

}

static char rcon(int i) {

if (i == 0) {

return 0;

}

char c = 1;

while (i != 1) {

char b = c & 0x80;

c <<= 1;

if (b == 0x80) {

c ^= 0x1b;

}

i--;

}

return c;

}

//Crypting components

static string keyExpansion(string key) {

int nKey = static\_cast<int>(key.length());

int rNum = nKey / 4 + 6;

string rKey = string(16 \* rNum + 16, '0');

int i = 0;

while (i < nKey) {

rKey[i] = key[i];

++i;

}

i = nKey;

while (i < 16 \* (rNum + 1)) {

string word = rKey.substr(i - 4, 4);

if (i % nKey == 0) {

rotWord(word);

subWord(word);

word[0] ^= rcon(i / nKey);

}

else if (nKey > 24 and i % nKey == 16) {

subWord(word);

}

for (int j = 0; j < 4; ++j) {

rKey[i + j] = rKey[i + j - nKey] ^ word[j];

}

i += 4;

}

return rKey;

}

static void addRoundKey(string &state, int round, string rKey) {

for (int i = 0; i < 4; ++i) {

for (int j = 0; j < 4; ++j) {

state[j + 4 \* i] ^= rKey[round \* 16 + 4 \* i + j];

}

}

return;

}

//Encrypting components

static void subBytes(string &state) {

for (int i = 0; i < 4; ++i) {

for (int j = 0; j < 4; ++j) {

state[i + 4 \* j] = sBox(state[i + 4 \* j]);

}

}

return;

}

static void shiftRows(string &state) {

for (int i = 1; i < 4; ++i) {

uint8\_t tmp[4];

for (int j = 0; j < 4; ++j) {

tmp[j] = state[i + 4 \* ((j + i) % 4)];

}

for (int j = 0; j < 4; ++j) {

state[i + 4 \* j] = tmp[j];

}

}

return;

}

static void mixColumns(string &state) {

for (int i = 0; i < 4; ++i) {

string tmp(4, '0');

tmp[0] = GF::gfMul(0x02, state[0 + 4 \* i]) ^ GF::gfMul(0x03, state[1 + 4 \* i]) ^ state[2 + 4 \* i] ^ state[3 + 4 \* i];

tmp[1] = GF::gfMul(0x02, state[1 + 4 \* i]) ^ GF::gfMul(0x03, state[2 + 4 \* i]) ^ state[0 + 4 \* i] ^ state[3 + 4 \* i];

tmp[2] = GF::gfMul(0x02, state[2 + 4 \* i]) ^ GF::gfMul(0x03, state[3 + 4 \* i]) ^ state[0 + 4 \* i] ^ state[1 + 4 \* i];

tmp[3] = GF::gfMul(0x02, state[3 + 4 \* i]) ^ GF::gfMul(0x03, state[0 + 4 \* i]) ^ state[1 + 4 \* i] ^ state[2 + 4 \* i];

for (int j = 0; j < 4; ++j) {

state[j + 4 \* i] = tmp[j];

}

}

return;

}

//Decrypting components

static void invShiftRows(string &state) {

for (int i = 1; i < 4; ++i) {

string tmp(4, '0');

for (int j = 0; j < 4; ++j) {

tmp[j] = state[i + 4 \* j];

}

for (int j = 0; j < 4; ++j) {

state[i + 4 \* ((j + i) % 4)] = tmp[j];

}

}

return;

}

static void invSubBytes(string &state) {

for (int i = 0; i < 4; ++i) {

for (int j = 0; j < 4; ++j) {

state[i + 4 \* j] = invSBox(state[i + 4 \* j]);

}

}

return;

}

static void invMixColumns(string &state) {

for (int i = 0; i < 4; ++i) {

string tmp(4, '0');

tmp[0] = GF::gfMul(0x0e, state[0 + 4 \* i]) ^ GF::gfMul(0x0b, state[1 + 4 \* i]) ^ GF::gfMul(0x0d, state[2 + 4 \* i]) ^ GF::gfMul(0x09, state[3 + 4 \* i]);

tmp[1] = GF::gfMul(0x09, state[0 + 4 \* i]) ^ GF::gfMul(0x0e, state[1 + 4 \* i]) ^ GF::gfMul(0x0b, state[2 + 4 \* i]) ^ GF::gfMul(0x0d, state[3 + 4 \* i]);

tmp[2] = GF::gfMul(0x0d, state[0 + 4 \* i]) ^ GF::gfMul(0x09, state[1 + 4 \* i]) ^ GF::gfMul(0x0e, state[2 + 4 \* i]) ^ GF::gfMul(0x0b, state[3 + 4 \* i]);

tmp[3] = GF::gfMul(0x0b, state[0 + 4 \* i]) ^ GF::gfMul(0x0d, state[1 + 4 \* i]) ^ GF::gfMul(0x09, state[2 + 4 \* i]) ^ GF::gfMul(0x0e, state[3 + 4 \* i]);

for (int j = 0; j < 4; ++j) {

state[j + 4 \* i] = tmp[j];

}

}

return;

}

public:

static string encrypt(string input, string key) {

string rKey = keyExpansion(key);

cout << "\n\nKey expansion table (hexa): " << Util::asciiToHex(rKey) << "\n\n";

string output;

int rNum = static\_cast<int>(key.length()) / 4 + 6;

int nIn = static\_cast<int>(input.length());

int index = 0;

while (index < nIn) {

string state = input.substr(index, 16);

cout << "State " << index / 16 << ": " << "\"" << state << "\"\n";

cout << "State " << index / 16 << " (hexa): " << Util::asciiToHex(state) << "\n\n";

int round = 0;

cout << "Round 0:\n";

addRoundKey(state, round, rKey);

cout << "State " << index / 16 << " after addRoundKey() (hexa): " << Util::asciiToHex(state) << "\n\n";

for (round = 1; round < rNum; ++round) {

cout << "Round " << round << ":\n";

subBytes(state);

cout << "State " << index / 16 << " after subBytes() (hexa): " << Util::asciiToHex(state) << endl;

shiftRows(state);

cout << "State " << index / 16 << " after shiftRows() (hexa): " << Util::asciiToHex(state) << endl;

mixColumns(state);

cout << "State " << index / 16 << " after mixColumns() (hexa): " << Util::asciiToHex(state) << endl;

addRoundKey(state, round, rKey);

cout << "State " << index / 16 << " after addRoundKey() (hexa): " << Util::asciiToHex(state) << "\n\n";

}

cout << "Round " << rNum << ":\n";

subBytes(state);

cout << "State " << index / 16 << " after subBytes() (hexa): " << Util::asciiToHex(state) << endl;

shiftRows(state);

cout << "State " << index / 16 << " after shiftRows() (hexa): " << Util::asciiToHex(state) << endl;

addRoundKey(state, rNum, rKey);

cout << "State " << index / 16 << " after addRoundKey() (hexa): " << Util::asciiToHex(state) << "\n\n";

cout << "-------------\n";

output += state;

index += 16;

}

return output;

}

static string decrypt(string input, string key) {

string rKey = keyExpansion(key);

cout << "\n\nKey expansion table (hexa): " << Util::asciiToHex(rKey) << "\n\n";

string output;

int rNum = static\_cast<int>(key.length()) / 4 + 6;

int nIn = static\_cast<int>(input.length());

int index = 0;

while (index < nIn) {

string state = input.substr(index, 16);

cout << "State " << index / 16 << " (hexa): " << Util::asciiToHex(state) << "\n\n";

int round = rNum;

cout << "Round " << rNum << ":\n";

addRoundKey(state, round, rKey);

cout << "State " << index / 16 << " after addRoundKey() (hexa): " << Util::asciiToHex(state) << "\n\n";

for (round = rNum - 1; round > 0; --round) {

cout << "Round " << round << ":\n";

invShiftRows(state);

cout << "State " << index / 16 << " after invShiftRows() (hexa): " << Util::asciiToHex(state) << endl;

invSubBytes(state);

cout << "State " << index / 16 << " after invSubBytes() (hexa): " << Util::asciiToHex(state) << endl;

addRoundKey(state, round, rKey);

cout << "State " << index / 16 << " after addRoundKey() (hexa): " << Util::asciiToHex(state) << endl;

invMixColumns(state);

cout << "State " << index / 16 << " after invMixColumns() (hexa): " << Util::asciiToHex(state) << "\n\n";

}

cout << "Round 0:\n";

invShiftRows(state);

cout << "State " << index / 16 << " after invShiftRows() (hexa): " << Util::asciiToHex(state) << endl;

invSubBytes(state);

cout << "State " << index / 16 << " after invSubBytes() (hexa): " << Util::asciiToHex(state) << endl;

addRoundKey(state, 0, rKey);

cout << "State " << index / 16 << " after addRoundKey() (hexa): " << Util::asciiToHex(state) << "\n\n";

cout << "-------------\n";

output += state;

index += 16;

}

return output;

}

};

Server.cpp

#include "pch.h"

#include <conio.h>

#include <iostream>

#include <WinSock2.h>

#include <CkDh.h>

#include <CkCrypt2.h>

#include <string>

#include <vector>

using namespace std;

/\*Nhập dữ liệu vào string s, sau đó đệm thêm các ký tự ' ' (dấu cách) để đảm bảo độ dài chia hết cho 16. Độ dài dữ liệu không quá 1024 ký tự\*/

void setString(string &s) {

do {

cout << "Type input: ";

getline(cin, s);

int m = s.size() % 16;

if (m != 0) {

m = 16 - m;

}

for (int i = 0; i < m; ++i) {

s += ' ';

}

} while (s.size() > 1024);

}

int main() {

//----------

//----------

//----------

//----------

//----------

//Khởi tạo socket

WSADATA wsa;

char buff[1024];

cout << "Setting up socket... ";

if (WSAStartup(MAKEWORD(2, 2), &wsa)) {

cout << "WSAStartup failed. Error: " << WSAGetLastError() << endl;

return 1;

}

SOCKET listener = socket(2, SOCK\_STREAM, IPPROTO\_TCP);

if (listener == INVALID\_SOCKET) {

cout << "socket failed. Error: " << WSAGetLastError() << endl;

return 1;

}

sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

addr.sin\_port = htons(2511);

if (bind(listener, (sockaddr \*)&addr, sizeof(addr))) {

cout << "bind failed. Error: " << WSAGetLastError() << endl;

return 1;

}

if (listen(listener, 5)) {

cout << "listen failed. Error: " << WSAGetLastError() << endl;

return 1;

}

SOCKET client = accept(listener, NULL, NULL);

if (client == INVALID\_SOCKET) {

cout << "accept failed. Error: " << WSAGetLastError() << endl;

closesocket(listener);

WSACleanup();

return 1;

}

//----------

//----------

//----------

//----------

//----------

//Khởi tạo phương thức trao đổi bí mật chung Diffie-Hellman

cout << "Done.\n\nSetting up Diffie-Hellman protocol and shared secret:\n";

CkDh server;

bool success = server.UnlockComponent(":/");

if (!success) {

cout << server.lastErrorText() << "\r\n";

return -100;

}

//Khởi tạo số nguyên tố lớn p, căn nguyên thủy g và tính toán g ^ b mod p của server

cout << "Preparing server's prime, modulo and secret... ";

server.UseKnownPrime(2);

const char \*p = server.p(); /\*Số nguyên tố lớn p\*/

int g = server.get\_G(); /\*Căn nguyên thủy g\*/

string gs = to\_string(g);

const char \*eServer = server.createE(256); /\*g ^ b mod p của server\*/

cout << "Done.\nSending server's mixture to client... ";

string sendData = p;

sendData += '|' + gs + '|' + eServer; /\*Trộn p, g và e làm một, ngăn cách bởi dấu '|' để gửi sang client\*/

//Gửi dữ liệu trộn cho client

if (send(client, sendData.c\_str(), sendData.size(), NULL) == SOCKET\_ERROR) {

cout << "send failed. Error: " << WSAGetLastError() << endl;

closesocket(client);

WSACleanup();

return 1;

}

cout << "Done.\nReceiving client's mixture and shared secret... ";

//Nhận g ^ a mod p và bí mật chung đã được tính toán từ phía client dưới dạng dữ liệu trộn

string receiveData, eClient, kClient;

int res = recv(client, buff, sizeof(buff), 0);

if (res > 0) {

buff[res] = NULL;

receiveData += buff;

}

else if (res == 0) {

cout << "Connection closing...\n";

}

else {

cout << "recv failed. Error: " << WSAGetLastError() << endl;

closesocket(client);

WSACleanup();

return 1;

}

//Tách dữ liệu trộn từ client

vector<int> pos;

for (int i = 0; i < (int)receiveData.size(); ++i) {

if (receiveData[i] == '|') {

pos.push\_back(i);

}

}

eClient = receiveData.substr(0, pos[0]); /\*g ^ a mod p của client\*/

kClient = receiveData.substr(pos[0] + 1, receiveData.size() - pos[0] - 1); /\*Bí mật chung được tính toán từ phía client\*/

const char \*kServer = server.findK(eClient.c\_str()); /\*Bí mật chung được tính toán từ phía server\*/

cout << "Done.\nCreating key... ";

//Kiểm tra trùng khớp bí mật chung của hai phía

if (kServer != kClient) {

cout << "Keys are different. Quitting...\n";

closesocket(client);

WSACleanup();

}

else {

cout << "Keys are identical.\nDone setting up Diffie-Hellman.\n\nHashing key from the shared secret... ";

}

//----------

//----------

//----------

//----------

//----------

//Hash MD5 để tạo khóa 128 bit cho mã khóa AES

CkCrypt2 crypt;

success = crypt.UnlockComponent(":/");

if (success != true) {

std::cout << crypt.lastErrorText() << "\r\n";

return -100;

}

crypt.put\_EncodingMode("hex");

crypt.put\_HashAlgorithm("md5");

string key = crypt.hashStringENC(kServer); /\*Khóa được tạo - chuỗi hexa\*/

key = Util::hexToAscii(key); /\*Đổi từ chuỗi Hexa sang chuỗi thường\*/

//Bắt đầu gửi dữ liệu mã hóa cho client

cout << "Done.\nStart sending encrypted message.\n\n";

string input;

while (true) {

//Nhập và mã hóa đoạn tin

setString(input);

sendData = simpleAES::encrypt(input, key);

//Dữ liệu mã hóa được in ra bằng hàm std::cout ngay sau khi được mã hóa nên đảm bảo sự trung thực (không cài văn bản gốc vào đó)

cout << "Encrypted message in hex: " << Util::asciiToHex(sendData) << "\n\n";

//Gửi dữ liệu mã hóa cho client

if (send(client, sendData.c\_str(), sendData.size(), NULL) == SOCKET\_ERROR) {

cout << "send failed. Error: " << WSAGetLastError() << '\n';

closesocket(client);

WSACleanup();

return 1;

}

}

closesocket(client);

WSACleanup();

}

Client.cpp

#include "pch.h"

#include <iostream>

#include <WinSock2.h>

#include <ws2tcpip.h>

#include <CkDh.h>

#include <CkCrypt2.h>

#include <string>

#include <vector>

using namespace std;

int main() {

Sleep(2000);

//----------

//----------

//----------

//----------

//----------

//Khởi tạo socket

WSADATA wsa;

char buff[1024];

addrinfo \*result;

sockaddr\_in addr;

cout << "Setting up socket... ";

if (WSAStartup(MAKEWORD(2, 2), &wsa)) {

cout << "WSAStartup failed. Error: " << WSAGetLastError() << endl;

}

SOCKET server = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (server == INVALID\_SOCKET) {

cout << "socket failed. Error: " << WSAGetLastError() << endl;

WSACleanup();

return 1;

}

if (getaddrinfo("127.0.0.1", "http", NULL, &result)) {

cout << "send failed. Error: " << WSAGetLastError() << endl;

WSACleanup();

return 1;

}

memcpy(&addr, result->ai\_addr, result->ai\_addrlen);

addr.sin\_port = htons(2511);

if (connect(server, (sockaddr\*)&addr, sizeof(addr)) == SOCKET\_ERROR) {

cout << "Unable to connect to server.";

WSACleanup();

return 1;

}

//----------

//----------

//----------

//----------

//----------

//Nhận dữ liệu trộn từ server

cout << "Done.\n\nReceiving server's mixture... ";

CkDh client;

bool success = client.UnlockComponent(":/");

if (!success) {

cout << client.lastErrorText() << "\r\n";

return -100;

}

string p, gs, eServer, receiveData;

int res = recv(server, buff, sizeof(buff), 0);

if (res > 0) {

buff[res] = NULL;

receiveData += buff;

}

else if (res == 0) {

cout << "Connection closed.\n";

}

else {

cout << "recv failed. Error: " << WSAGetLastError() << endl;

closesocket(server);

WSACleanup();

return 1;

}

//Tách dữ liệu trộn từ server

vector<int> pos;

for (int i = 0; i < (int)receiveData.size(); ++i) {

if (receiveData[i] == '|') {

pos.push\_back(i);

}

}

p = receiveData.substr(0, pos[0]); /\*Số nguyên tố lớn p\*/

gs = receiveData.substr(pos[0] + 1, pos[1] - pos[0] - 1); /\*Căn nguyên thủy g\*/

eServer = receiveData.substr(pos[1] + 1, receiveData.size() - pos[1] - 1); /\*g ^ b mod p của server\*/

int g = stoi(gs.c\_str());

success = client.SetPG(p.c\_str(), g);

if (success != true) {

cout << "P is not a safe prime\n";

return 1;

}

//Tính toán g ^ a mod p và bí mật chung

const char \*eClient = client.createE(256);

const char \*kClient = client.findK(eServer.c\_str());

//Trộn g ^ a mod p và bí mật chung làm một, ngăn cách bởi dấu '|' để gửi sang server

string sendData = eClient;

sendData += '|';

sendData += kClient;

//Gửi dữ liệu trộn cho server

if (send(server, sendData.c\_str(), sendData.size(), NULL) == SOCKET\_ERROR) {

cout << "send failed. Error: " << WSAGetLastError() << endl;

closesocket(server);

WSACleanup();

return 1;

}

//----------

//----------

//----------

//----------

//----------

//Hash MD5 để tạo khóa 128 bit cho mã khóa AES

cout << "Done.\n\nHashing key from the shared secret... ";

CkCrypt2 crypt;

success = crypt.UnlockComponent(":/");

if (success != true) {

std::cout << crypt.lastErrorText() << "\r\n";

return -100;

}

crypt.put\_EncodingMode("hex");

crypt.put\_HashAlgorithm("md5");

string key = crypt.hashStringENC(kClient); /\*Khóa được tạo - chuỗi hexa\*/

key = Util::hexToAscii(key); /\*Đổi từ chuỗi Hexa sang chuỗi thường\*/

//Bắt đầu giải mã dữ liệu mã hóa từ server

cout << "Done.\nStart receiving encrypted message.\n\n";

while (true) {

//Nhận dữ liệu mã hóa

string receive;

int res = recv(server, buff, sizeof(buff), 0);

if (res > 0) {

buff[res] = NULL;

receive += buff;

}

else if (res == 0) {

cout << "Connection closed.\n";

}

else {

cout << "recv failed. Error: " << WSAGetLastError() << endl;

closesocket(server);

WSACleanup();

return 1;

}

cout << "Encrypted message in hex: " << Util::asciiToHex(receive) << endl;

//Giải mã dữ liệu

string output = simpleAES::decrypt(receive, key);

cout << "Decrypted message: " << "\"" << output << "\"\n\n";

}

closesocket(server);

WSACleanup();

}