

AES

10 rounds \rightarrow 128-bit Keys
12 rounds \rightarrow 192-bit Keys
14 rounds \rightarrow 256-bit Keys

AES Encryption

Function AES_Encryption (plaintext, Key)

Key_schedule = Key_Expansion (Key) // Generate round key

state = Convert_to_Matrix (plaintext)

state = AddRoundKey (state, Key_schedule[0])

for round = 1 to (Nr - 1) // Main round

state = SubBytes (state)

state = ShiftRows (state)

state = MixColumn (state)

state = AddRoundKey (state, Key_schedule[round])

// Final Round

state = SubBytes (state)

state = ShiftRows (state)

state = AddRoundKey (state, Key_schedule[Nr])

return Convert_To_Array (state)

End

function AES-Decrypt(cipher-text, Key)
 Key-schedule = Key-Expansion(Key)
 state = Convert-To-Matrix(cipher-text)
 state = AddRoundKey(state, Key-schedule[0])
 for round = (Nr - 1) down to 1:
 state = InvShiftRow(state)
 state = InvSubBytes(state)
 state = AddRoundKey(state, Key-schedule[round])
 state = InvMixColumn(state)

11 Final Round

state = InvShiftRow(state)
 state = InvSubBytes(state)
 state = AddRoundKey(state, Key-schedule [n])
 Return Convert-To-Array(state)

end

function Key-Expansion(Key)

expand the cipher Key into multiple round Keys using Rijndael's Key Schedule

Return Round-Keys.

end

function subBytes(state)

Replace each byte in the state using the S-box

Return new-state

end

function InvSubBytes(state)

Replace each byte in the state using the inverse S-box

return new-state

End

function ShiftRows(state)

Circularly shift rows of the state left

Return new-state

End

function InvShiftRows(state)

Circularly shift rows of the state right.

Return new-state

End

function MixColumn(state)

Apply matrix Multiplication in $\text{GF}(2^8)$ to each column

Return new-state

End

function InvMixColumn(state)

Apply Inverse matrix multiplication in $\text{GF}(2^8)$ to each column

return new-state

End

function AddRoundKey(state, round-key)

XOR state with round key

Return

End

Station-to-Station Protocol

- ① A & B agree on a large prime p & a generator g .
- ② A generates a private key a & computes public key
 $A\text{-pub} = g^a \pmod{p}$
- ③ B generates a private key b & computes public key
 $B\text{-pub} = g^b \pmod{p}$
- ④ $A \rightarrow B$: $A\text{-pub}$ (sends its public key to B)
- ⑤ B computes shared secret $s = A\text{-pub}^b \pmod{p}$
- ⑥ B signs $(A\text{-pub}, B\text{-pub})$ with its private key
 $\text{sig-}B = \text{sign}(B\text{-priv}, (A\text{-pub}, B\text{-pub}))$
- ⑦ B generates a random number R_B .
- ⑧ $B \rightarrow A$: $B\text{-pub}, \text{sig-}B, R_B$ (sends its public key, signature, random no.)
- ⑨ A computes shared secret $s' = B\text{-pub}^a \pmod{p}$
- ⑩ B signs A's verifier $\text{sig-}B$ using B's public key.
- ⑪ A signs $(A\text{-pub}, B\text{-pub}, R_B)$ with its private key
 $\text{sig-}A = \text{sign}(A\text{-priv}, (A\text{-pub}, B\text{-pub}), R_B)$
- ⑫ B generates a random num R_A
- ⑬ $A \rightarrow B$: $\text{sig-}A, R_A$ (sends its sign & nonce)
- ⑭ B verifies $\text{sig-}A$ using A's public key
- ⑮ Both A & B derive a session key H from B
- ⑯ A & B can now securely communicate

Elgamal Encryption

- ① Key generation (Done by the receiver)
- ② Encryption (Done by the sender)
- ③ Decryption (Done by the receiver)

① Key Generation

- ① Choose a large prime no. " p "
- ② Choose a generator ' g ' (primitive root of p)
- ③ Select a private key " n " (random integer such that $1 \leq n \leq p-2$)
- ④ Compute public key " y " as $y = g^n \pmod{p}$
- ⑤ Public Key: (p, g, y) .
- ⑥ Private Key: n

• Public key is shared with vendor

• Private key is kept secret by the receiver

② Encryption

- A vendor encrypts the msg M using the receiver public key
- ① Convert plaintext msg M into an integer ' m ' ($0 \leq m < p$)
 - ② Choose a random integer ' H ' ($1 \leq H \leq p-2$)
 - ③ Compute $c_1 = g^H \pmod{p}$
 - ④ Compute $c_2 = (m \times y^H) \pmod{p}$
 - ⑤ Find ciphertext (c_1, c_2) to receiver

- C_1 helps generate the shared secret.
- C_2 contains the encrypted msg.

Decryption

- The receiver decrypts the ciphertext using private key H .

① Compute the shared secret $s = C_1^H \mod p$

② Compute the modular inverse of $s^{-1} \mod p$ denoted as $s^{-1}_{\text{mod}} \text{ or } s_{\text{-inv}}$

③ Recover the message: $m = (C_2 \times s_{\text{-inv}}) \mod p$.

$$s = C_1^H = (g^H)^n = g^{Hn} \mod p$$

$$C_2 = m \cdot g^H = m \cdot g^{Hn} \mod p$$

Pseudo code

① Euclidean Algo

```
int gcd (int a,int b)
{
    if (b == 0) { return a; }

    gcd (b,a%b);
}
```

② Extended Euclidean Algo

```
int gcdExtended (int a,int b,int *n,int *y)
{
    if (a == 0)
    {
        *n = 0;
        *y = 1;
        return b;
    }

    int n1,y1;
    int gcd = gcdExtended(b%a,a,&n1,&y1);
    *n = y1 - (b/a)*n1;
    *y = n1;

    return gcd;
}
```

③ Affine cipher implementation

→ symmetric key cryptography

$$c = (H_1 \times M + H_2) \bmod 26$$

$$M = H_1^{-1}(c - H_2) \bmod 26$$

H_1 should be coprime with 26 or m
 $\& H_1^{-1}$ = Modular multiplicative inverse
of a modulo M .

Encryption

function affineEncrypt(plaintext, a, b, m):
 ciphertext ← ""

 for each letter P in plaintext:

 if P is a letter:

$$c \leftarrow (a \times (P - 'A') + b) \bmod m$$

$$\text{ciphertext} \leftarrow \text{ciphertext} + (c + 'A')$$

 else

$$\text{ciphertext} \leftarrow \text{ciphertext} + P$$

return ciphertext

Decryption

function affineDecrypt(ciphertext, a, b, m):

 plaintext ← ""

 a-inv ← modularInverse(a, m)

 for each letter c in ciphertext:

 if c is a letter:

$$p \leftarrow (\text{a-inv} \times (c - 'A' - b)) \bmod m$$

$$\text{plaintext} \leftarrow \text{plaintext} + (p + 'A')$$

 else

$$\text{plaintext} \leftarrow \text{plaintext} + c$$

return plaintext

finding modular Inverse

function modularInverse(a,m);

for n from 1 to m-1:

if $(axn) \bmod m == 1$:

return n;

return -1;

Cryptanalysis of Affine cipher

① Input a ciphertext

② for each a from 1 to 26-1:

if $\text{mod-inverse}(a, 26) != -1$:

for each b from 0 to 26-1:

} decrypt affine cipher (ciphertext, a, b):

y

return 0;

③ function decrypt-affine-cipher(ciphertext, a, b)

{, plaintext = ""

a-inv = mod-inverse(a, 26);

for each letter in ciphertext:

if (isAlpha(letter)):

pt ← $(a\text{-inv} \times (\text{letter} - 'A' - b)) \bmod m$

plaintext ← plaintext + (pt + ('A'))

else

plaintext ← plaintext + c

```

int invalid (char * plaintext) {
    for (int i=0; plaintext[i] != '\0'; i++) {
        if (isalpha (plaintext[i]) && plaintext[i] != ' ')
            return 0;
    }
    return 1;
}

```

Matrix Inverse

- ① Inverse the matrix
- ② Input the matrix
- ③ find the minor
- ④ find the Determinant of the matrix
- ⑤ find the cofactor of the matrix Do transpose.
- ⑥ Then if value is less than 0 & greater than 26
take mod 26

Multiply with Determinant inverse.

function determinant - 2x2 - Mod 26 (matrix):

$$\det = (\text{matrix}[0][0] \times \text{matrix}[1][1] - \text{matrix}[0][1] \times \text{matrix}[1][0]) \bmod 26$$

if $\det < 0$:

$$\det += 26;$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

return \det ;

function determinant - 3x3 - mod 26 (matrix):

$$\det = [a(eifh) + b(fg - dh) + c(dh - eg)] \bmod 26;$$

```
if det < 0:  
    det += 26  
return det
```

function modular-inverse(a, mod):

for i from 1 to mod-1:

if ($a \times i$) mod $\text{mod} == 1$

return i;

return -1;

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

function cofactors_3x3-matrix(matrix):

cofactor[0][0] = $(e \times i - f \times h) \bmod 26$

cofactor[0][1] = $-(d \times i - f \times g) \bmod 26$

cofactor[0][2] = $(d \times h - e \times g) \bmod 26$

cofactor[1][0] = $-(b \times i - c \times h) \bmod 26$

cofactor[1][1] = $(a \times i - c \times g) \bmod 26$

cofactor[1][2] = $-(a \times h - g \times f) \bmod 26$

cofactor[2][0] = $(b \times f - e \times i) \bmod 26$

cofactor[2][1] = $-(a \times f - d \times i) \bmod 26$

cofactor[2][2] = $(a \times e - b \times d) \bmod 26$

for i from 0 to 2:

for j from 0 to 2:

if cofactor[i][j] < 0:

cofactor += 26

return cofactor

function transpose (matrix, size) :

 transposed = empty-matrix (size, size)

 for i from 0 to size - 1:

 for j from 0 to size - 1:

 transposed [i][j] = matrix [j][i]

 return transposed

function inverse_matrix_mod26 (matrix, size) :

 if size == 2:

 find det

 find det-inv

 if det-inv == -1:

 return -1;

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

 inverse [0][0] = (matrix [1][1] * det-inv) mod 26

 inverse [0][1] = (-matrix [0][1] * det-inv) mod 26

 inverse [1][0] = (-matrix [1][0] * det-inv) mod 26

 inverse [1][1] = (matrix [0][0] * det-inv) mod 26

 for i from 0 to 1:

 for j from 0 to 1:

 if inverse [i][j] < 0:

 inverse [i][j] += 26

 return inverse

else if size == 3:

 det = determinant - 3x3-mod26 (matrix);

 find det-inv = modular_inverse (det, 26);

 if det-inv == -1:

 return -1;

cofactor = cofactor 3×3 matrix (matrix)

adjoint = transpose (cofactor, 3)

for i from 0 to 2:

 for j from 0 to 2:

 inverse[i][j] = (adjoint[i][j] * det - inv) mod 26

 if inverse[i][j] < 0:

 inverse[i][j] += 26

 return inverse

else

 return -1;

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Hill cipher

$$C \leftarrow H P \quad C = HP \bmod 26$$

$$P \leftarrow H^{-1}C \bmod 26$$

$(H, 26)$ should be coprime to each other,
modular inverse of H should exist.

Encryption

function hill_cipher_encrypt(plaintext, key_matrix, size):

 plaintext_number = convert_text_to_number(plaintext)

 if length(plaintext_number) is not a mult of size:
 pad plaintext_number with 'x'

 ciphertext_number = []

 for i from 0 to length(plaintext_number) - 1

```
block = plaintext-number[i : i + size]
encrypted-block = multiply_matrix_vector_mod26
                  (key-matrix, block, size);
append encrypted-block to ciphertext-no.
ciphertext = convert-number-to-text(ciphertext-no.)
return ciphertext.
```

Decryption

```
function Hill cipher-decrypt (ciphertext, key-matrix, size) :
    ciphertext-number = convert-text-to-number(ciphertext)
    key-inverse = inverse-matrix_mod26(key-matrix, size)
    if key-inverse is "No inverse exist":
        return "Decryption Not Possible"
    plaintext-number = []
    for i from 0 to length(ciphertext-number) - 1
        block = ciphertext-number[i : i + size]
        decrypted-block = multiply-matrix-vector-mod26
                           (key-inverse, block, size)
```

append decrypted-block to plaintext-no.

plaintext = convert-number-to-text(plaintext)

return plaintext.

Helper function

function convert_text_to_no. (text):

return [ASCII-value(letter) - ASCII-value('A')
for each letter in text]

function convert_number_to_text (numbers):

return [character (n + ASCII-value('A'))] for each n in
numbers

function multiply_matrix_vector_mod26 (matrix, vector, size)

result = []

for i from 0 to size-1:

sum = 0

for j from 0 to size-1:

sum += matrix[i][j] * vector[j]

result[i] = sum mod 26

return result.

Hill Cipher Cryptanalysis

function recover_key_matrix (plaintxt_samples, ciphertxt_samples, size):

plaintxt_matrix = form_matrix_from_text (plaintxt_samples, size) /

ciphertxt_matrix = form_matrix_from_text (ciphertxt_samples, size) /

plaintxt_inverse = inverse_matrix_mod26 (plaintxt_matrix, size);

if plaintxt_inverse is "No inverse exist":

return "Key cannot be recovered";

key_matrix = multiply_matrix_mod26 (ciphertxt_matrix,
plaintxt_inverse, size);

return key_matrix

IFDR Cryptanalysis Construction

IS-4-1
IS-5-10

function generateKeystream (seed, coeff, keystream, l, n) {
for i from 0 to l-1 Do:
keystream[i] = seed[i];
for i from 0 to n-l-1 Do:
feedback = 0
for j from 0 to l-1 Do:
feedback = feedback XOR (coeff[j] X
keystream[i+j])
keystream[i+l] = feedback MOD 2
}
}

function Encrypt (plaintxt, keystream, ciphertext, n) {
for i from 0 to n-1 Do:
ciphertext[i] = plaintext[i] XOR keystream[i];
}
}

function Decrypt (ciphertext, keystream, plaintext, n) {
for i from 0 to n-1 Do:
plaintext[i] = ciphertext[i] XOR keystream[i]
}
}

Main () {

l < 4 (key length)

n < 15 (keystream length)

seed < [0, 1, 0, 1]

coeff < [1, 0, 1, 1]

keystream < [0]^n

plaintxt < [0, 1, - -]

call generateKeystream (seed, coeff, keystream, l, n)

encrypt, Decrypt.

LFSR Cryptanalysis

- ① Input plaintext, ciphertext, seed length, Keystream size.
- ② Generate the Keystream using XOR operation b/w ciphertext & plaintext.
- ③ Find the coefficient of length equal to seed length.
Then also find the Keystream & check if the character in earlier Keystream are matching with the current Keystream. If matching then 0H, if not return -1;

```
void XOR (int plaintext[], int ciphertext[], int result[], int length) {  
    for (int i=0; i<length; i++)  
        result[i] = a[i] ^ b[i];  
}
```

```
int Coeff (int Keystream[], int l, int Keystream_size, int coeff[]){  
    int total = 1 << l = 2^l - 1;           → <<  
    for (int i=0; i<total; i++){  
        for (int j=0; j< l; j++)  
            coeff[j] = (i>>j) & 1;  
    }  
    int generated_Keystream [Keystream_size];  
    generateKeystream (coeff, l, Keystream_size, generated_Keystream);  
    int match = 1;  
    for (int i=0; i<Keystream_size; i++){  
        if (generated_Keystream[i] != Keystream[i]) {  
            match = 0;  
            break;  
        }  
    }  
    if (match){ return 1; }  
    else { return 0; }  
}
```

DES

function DES-Encrypt (plaintext, key)

Generate 16 subkeys using Key-Schedule (key)

plaintext-permuted = Initial-Permutation (plaintext)

split plaintext-permuted into L & R (each 32 bits)

for round = 1 to 16 :

temp = R

R = L XOR fiestel-function (R, subkey (round))

L = temp

ciphertext = Final-Permutation (R + L)

Return Ciphertext

End Function.

function DES-Decrypt (ciphertext, key)

Generate 16 subkeys using Key-Schedule (key)

ciphertext-permuted = Initial-Permutation (R ciphertext)

split ciphertext-permuted into L & R (each 32 bits)

for round 1 to 16 :

temp = R

R = L XOR fiestel-function (R, subkey (16-round+1))

L = temp

plaintxt = final-Permutation (R + L)

Return PlainText

End Function

Function KeySchedule(Key)

 Apply Permutated-Choice-1(Key) to get 56-bit Key
 split Key into Left (c) & Right (D) (each 18 bits)
 for round = 1 to 16:

 Perform left-circular shift (c,D) based on shift schedule
 (round)

 subKey[round] = Permutated-choice-2(c+D)

 Return subKeys

End Function.

Function Feistel-function(R, subKey)

 expanded_R = Expansion-Permutation(R)

 XOR-Result = expanded_R XOR subKey

 L-box-Output = Apply-L-Boxes(XOR-Result)

 permuted-output = P-Permutation(L-box-Output)

 Return permuted-output

End

Function Initial_Perm(block)

 Return Permute(block, IP-table)

End

Function Final_Perm(block)

 Return Permute(block, FP-table)

End

ECCDA

- ① Key Generation
- ② Signature Generation
- ③ Signature Verification.

Step 1. Key Generation

The signer generates a public-private key pair.

- ① choose an elliptic curve 'E' over a finite field " \mathbb{F}_p ".
- ② Select a base point ' G_1 ' (of order n) on the curve.
- ③ Choose a private key d (random integer such that $0 \leq d < n$)
- ④ Compute public key $\alpha = d \times G_1$ (scalar multiplication on the curve).
- ⑤ Public Key : (E, p, G_1, n, α)
- ⑥ Private Key : d

- The private key (d) is kept secret
- The public key (α) is shared with verifier.

Step 2 - sig Generation

The signer generates a signature for message M .

- ① Compute hash $c = h(M)$ (using a cryptographic hash function like SHA-256).
- ② choose a random k ($0 < k < n$)

- ③ the compute point $R = H \times G_1$.
- ④ Compute $\gamma = n - R \bmod n$ (where $n \cdot R$ is the n -coordinate of R)
- ⑤ If $\gamma = 0$, choose another H
- ⑥ Compute $d = H^{-1} \times (e + d \times \gamma) \bmod n$
- ⑦ If $d = 0$ choose another H .
- ⑧ Signature = (γ, d)
 - removes randomness & prevents signature reuse.
 - d binds the signature to the private key

Signature Verification

- ① Compute hash $e = h(m)$
- ② Compute $w^2 d^{-1} \bmod n$
- ③ Compute $U_1 = exw \bmod n$ & $U_2 = \gamma xw \bmod n$
- ④ Compute $P = U_1 \times G_1 + U_2 \times Q$
- ⑤ Compute $v = x - p \bmod n$ (where $x - p$ is the n -coordinate of P).
- ⑥ If $v = \gamma$ the sig. is valid, otherwise reject it

Lichten Digital signature Algorithm

- ① Key Generation (Done by the signer)
- ② Signature Generation (Done by the signer)
- ③ Signature Verification (Done by verifier)

Step 1 Key Generation

The signer generates their public & private keys

- ① choose a large prime p & a prime divisor q ($q \mid p-1$)
- ② select a generator ' g ' of order q in \mathbb{Z}_p
- ③ choose a private key ' n ' (random integer such that $0 < n < q$)
- ④ Compute public key $y = g^n \pmod{p}$
- ⑤ Public Key: (p, q, g, y)
- ⑥ Private Key: n

- Public Key is shared with Verifiers
- Private Key is kept secret by the signer.

Step 2 Signature Generation

- The signer generates a signature for message M .
- ① choose a plaintext
 - ② choose a random nonce H ($0 < H < q$)
 - ③ Compute commitment $x = g^H \pmod{p}$

- (3) Compute hash $e = h(m || r)$ & g (n is a cryptographic hash function)
- (4) Compute the signature component $s = (h - xr^e) \mod q$.
- (5) Signature = (s, r)

r acts as a commitment to randomness.

s ensures the signature is linked to the private key.

Step 3 Sig. Verify

- (1) Compute hash $e = h(m || r) \mod q$
- (2) Compute verification value $v = g^d \cdot x^{y^e} \mod p$.
- (3) If $v == r$, the signature is valid; otherwise reject it.

① Extended Euclidean for Polynomial

function ExtendedEuclidean(A, B)

$$d_0 = 1, d_1 = 0$$

$$T_0 = 0, T_1 = 1$$

while $B \neq 0$ do:

$$Q, R = \text{PolynomialDivision}(A, B)$$

// update A & B for the next iteration

$$A := B$$

$$B := R$$

// update Coefficients d & T

$$d := d_0 - Q \times d_1$$

$$T := T_0 - Q \times T_1$$

// shift values for next iteration

$$d_0 = d_1, d_1 = d$$

$$T_0 = T_1, T_1 = T$$

end while.

return (A, d_0, T_0) // gcd(A, B), Bezout coefficients

d_0 & T_0 .

② Multiplication

function multiply ($a, b, m, \text{deg_m}$):
 $\text{result} \leftarrow 0$

while ($b \neq 0$) do:

 if ($b \& 1$) then

$\text{result} \leftarrow \text{result} \text{ XOR } a$

$b \leftarrow b \gg 1$ // Shift b to right

 // Check if highest bit is set before shifting

a

 if ($(a \text{ AND } (1 \ll (\text{deg_m} - 1)))$) then:

$a \leftarrow (a \ll 1) \text{ XOR } m$ // Reduce
 using $m(x)$

 else:

$a \leftarrow a \ll 1$

Return result

Main:

 Declare deg-a, deg-b, deg-m