AES

10 rounds → 128-bit Keys
12 rounds → 192-bit Keys
14 rounds → 256-bit Keys

AES Encryption

```
Function AES_Encryption (plaintext, Key)
    Key_schedule = Key_Expansion (Key) // Generate round keys
    state = Convert_to_Matrix (plaintext)
    state = AddroundKey (state, Key_schedule[0])
    for round = 1 to (Nr-1) // main rounds
        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(ciphertext, Key)
    Key-schedule = Key-Expansion(Key)
    State = Convert_To_Matrix(ciphertext)
    State = AddRoundKey(State, Key-schedule[NR])
    for round = (NR-1) down to 1:
        State = InvShiftRows(State)
        State = InvSubBytes(State)
        State = AddRoundKey(State, Key-schedule[round])
        State = InvMixColumn(State)

    // Final Rounds
    State = InvShift Rows(State)
    State = Inv SubBytes(State)
    State = AddRoundKey(State, Key-schedule[0])
    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 GF(2^8) to each column
Return new_state
End

Function InvMixColumn(state)
Apply Inverse matrix Multiplication in 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
$A\text{-pub} = g^a \bmod p$
(3) B generates a private key b & computes public key $B\text{-pub} = g^b \bmod p$

(4) A → B : A-pub (sends its public key to B)
(5) B computes shared secret $s = A\text{-pub}^b \bmod p$
(6) B signs (A-pub, B-pub) with its private key
$sig\text{-}B = sign(B\text{-priv}, (A\text{-pub}, B\text{-pub}))$

(7) B generate a random number R-B.
(8) B → A : B-pub, sig_B, R_B (sends its public key, signature, random no.
(9) A computes shared secret $s = B\text{-pub}^a \bmod p$
(10) ~~B signs~~ A verifies sig-B using B's public key.

(11) A signs (A_pub, B_pub, R-B) with its private key
$sig\text{-}A = sign(A\text{-priv}, (A\text{-pub}, B\text{-pub}), R\text{-}B)$.

(12) A. generates a random nom R-A
(13) A → B: sign_A, R_A (sends its sign & nonce)

(14) B verifies sig-A using A's public key
(15) Both A & B derive a session key K from S
(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 "x" (random integer such that $1 \leq x \leq p-2$)
4. Compute public key "y" as $y = g^x \bmod p$
5. Public Key: (p, g, y)
6. Private Key: x

- Public key is shared with sender
- Private key is kept secret by the receiver

2. Encryption

A sender encrypts the msg M using the receiver public key

1. Converts plaintext msg M into an integer 'm' $(0 \leq m < p)$
2. Choose a random integer 'k' $(1 \leq k \leq p-2)$
3. Compute $c_1 = g^k \bmod p$
4. Compute $c_2 = (m \times y^k) \bmod p$
5. Find ciphertext (c_1, c_2) to sender

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

Decryption

- The receiver decrypts the ciphertext using private key x.

(1) Compute the shared secret $s = C_1 \wedge x \bmod p$

(2) Compute the modular inverse of $s \bmod p$ denoted as $s\text{-inv}$

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

$$s = C_1^{\,x} = (g^k)^x = g^{kx} \bmod p$$

$$C_2 = m \cdot g^{kx} = m \cdot g^{kx} \bmod p$$

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 * x, int * y)
{
  if (a == 0)
  {
    *x = 0;
    *y = 1;
    return b;
  }
  int x1, y1;
  int gcd = gcdExtended (b % a, a, &x1, &y1);
  *x = y1 - (b/a) * x1;
  *y = x1;
  return gcd;
}
```

(3) Affine Cipher Implementation

→ Symmetric Key Cryptography

$C = (K_1 \times M + K_2) \bmod 26$

$M = K_1^{-1}(C - K_2) \bmod 26$

K_1 should be coprime with 26 or m.

& K_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 ← (a × (P - 'A') + b) mod m
            ciphertext ← ciphertext + (C + 'A')
        else
            ciphertext ← 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 ← (a_inv × (C - 'A' - b)) mod m
            plaintext ← plaintext + (P + 'A')
        else
            plaintext ← plaintext + C
```

return plaintext

finding Modular Inverse

```
function modularInverse(a,m):
    for x from 1 to m-1:
        if (a x x) mod m == 1:
            return x;
    return -1;
```

Cryptanalysis of Affine cipher

① Input a ciphertext

② for each a from 1 to 26-1:
    if mod_inverse(a,26) != -1:
        for each b from 0 to 26-1:
            decrypt_affine_cipher(ciphertext,a,b):
        }
}
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)):
            p ← (a_inv x (letter - 'A' - b)) mod m
            plaintext ← plaintext + (p + 'A')
    else
        plaintext ← plaintext + c

```
int isvalid (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 maxix Do transpose.
6. Then if value is less than 0 & greater than 26 table mod 26.

→ Multiply with Determinant inverse

```
function determinant_2x2_mod26(matrix):
    det = (matrix[0][0] x matrix[1][1] - matrix[0][1] x matrix[1][0]) mod 26
    if det < 0:
        det += 26;
    return det;
```

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

```
function determinant_3x3_mod26 (matrix):
    det = [a(ei x fh) + b(fg - dxi) + c(dh - eg)];
                                    mod 26;
```

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

function modular_inverse(a, mod):
    for i from 1 to mod-1:
        if (a*i) % mod == 1
            return i;
    return -1;
function cofactor_3x3_matrix(matrix):
    cofactor[0][0] = (e*i - f*h) mod 26
    cofactor[0][1] = -(d*i - f*g) mod 26
    cofactor[0][2] = (d*h - e*g) mod 26
    cofactor[1][0] = -(b*i - c*h) mod 26
    cofactor[1][1] = (a*i - c*g) mod 26
    cofactor[1][2] = -(a*h - g*b) mod 26
    cofactor[2][0] = (b*f - e*c) mod 26
    cofactor[2][1] = -(a*f - d*c) mod 26
    cofactor[2][2] = (a*e - b*d) mod 26

    for i from 0 to 2:
        for j from 0 to 2:
            if cofactor[i][j] < 0:
                cofactor += 26
    return cofactor
```

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

```
function transpose(matrix, size):
    transposed = empty_matrix(size, size)
    for i from 0 to size-1:
        for j from 0 to size-1:
            transpose[i][j] = matrix[j][i]
    return transpa
```

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

```
function inverse_matrix_mod26(matrix, size):
    if size == 2:
    find det
find det_inv
    if det_inv == -1:
        return -1;
    inverse[0][0] = (matrix[1][1] x det_inv) mod 26
    inverse[0][1] = (-matrix[0][1] x det_inv) mod 26
    inverse[1][0] = (-matrix[1][0] x det_inv) mod 26
    inverse[1][1] = (matrix[0][0] x 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_3x3_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) % 26
        if inverse[i][j] < 0:
            inverse[i][j] += 26
return inverse
else
    return -1;
```

Hill cipher

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

~~C = HM~~ $C = KP \bmod 26$

$P = K^{-1}C \bmod 26$

$(K, 26)$ should be coprime to each other.
modular inverse of K 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
            number]
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] x vector[j]
        result[i] = sum mod 26
    return result.
```

Hill Cipher Cryptanalysis

```
function recover_Key_matrix(plaintext_samples, ciphertext_samples, size):
    plaintext_matrix = form_matrix_from_text(plaintext_samples, size);
    Ciphertext_matrix = form_matrix_from_text(ciphertext_samples, size);
    plaintext_inverse = inverse_matrix_mod26(plaintext_matrix, size);
    if plaintext_inverse is "No inverse exist":
        return "Key cannot be recovered";
    Key_matrix = multiply_matrices_mod26(ciphertext_matrix,
                                         plaintext_inverse, size);
    return Key_matrix
```

LFSR ~~Cryptanalysis~~ Construction 15-4-1
15-5-10

```
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(plaintext, 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 (seed length)
    n ∈ 15 (keystream length)
    seed ∈ [0,1,0,1]
    coeff ∈ [1,0,0,1]
    keystream ∈ [0]*n
    plaintext ∈ [0,0,1,---]
    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 seedlength. Then also find the Keystream & check if the character in earlier Keystream are matching with the current Keystream. if matching then ok, 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, generate_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;
    }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 feistel_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 (ciphertext)
    Split ciphertext_permuted into L & R (each 32 bits)
    for round 1 to 16:
        temp = Ri
        R = L XOR feistel_function (R, subkey (16-round+1))
        L = temp
    plaintext = final_Permutation (R+L)
    Return plaintext
End function
```

```
Function Key_Schedule(Key)
    Apply Permuted_Choice_1(Key) to get 56-bit Key
    Split Key into Left (C) & Right (D) (each 28 bits)
    for round = 1 to 16:
        Perform Left_Circular_Shift (C, D) based on Shift_Schedule
                                            (round)
        SubKey[round] = Permuted_Choice_2(C+D)
    Return SubKeys
End Function

Function Feistel_Function(R, SubKey)
    expanded_R = Expansion_Permutation (R)
    XOR_Result = expanded_R XOR SubKey
    S_box_Output = Apply_S_Boxes (XOR_Result)
    permuted_output = P_Permutation (S_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
```

ECDSA

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 "F_p".
2. Select a base point 'G' (of order n) on the curve.
3. Choose a private key d (random integer such that $0 < d < n$)
4. Compute public key $Q = d \times G$ (scalar multiplication on the curve).
5. Public Key : (E, p, G, n, Q)
6. Private Key : d

- The private key (d) is kept secret
- The public key (Q) is shared with verifiers.

Step 2 Sig Generation

The signer generates a signature for message M.

1. Compute hash $e = H(M)$ (using a cryptographic hash function like SHA-256).
2. Choose a random k $(0 < k < n)$

(3) the compute point $R = H \times G$.
(4) Compute $r = x\text{-}R \bmod n$ (where $x\text{-}R$ is the x-coordinate of R)
(5) If $r = 0$, choose another H
(6) compute $d = H^{-1} \times (e + d \times r) \bmod n$
(7) If $d = 0$ choose another H.
(8) signature = (r, d)

- r ensures randomness & prevents signature reuse.
- d binds the signature to the private key

Signature Verification

(1) Compute hash $e = H(M)$
(2) compute $w = d^{-1} \bmod n$
(3) Compute $u1 = e \times w \bmod n$ & $u2 = r \times w \bmod n$
(4) Compute $P = u1 \times G + u2 \times Q$
(5) Compute $v = x\text{-}P \bmod n$ (where $x\text{-}P$ is the x-coordinate of P).
(6) If $v == r$ the Sig. is valid, otherwise reject it

Schnorr Digital Signature Algorithm

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

Step1 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 Z_p
3. Choose a private key 'x' (random integer such that $0 < x < q$)
4. Compute public key $y = g^x \bmod p$
5. Public key: (p, q, g, y)
6. Private key: x

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

Step2 Signature Generation

- The signer generates a signature for message M.

~~1. choose a plaintext~~

1. choose a random nonce k $(0 < k < q)$
2. Compute commitment $r = g^k \bmod p$

(3) Compute hash $e = H(M \| r)$ & q (H is a cryptographic hash function)

(4) Compute the signature component $s = (H - x * e) \bmod q$.

(5) Signature = (r, s)

r acts as a commitment to randomness & ensures the signature is linked to the private key.

Step 3 Sig. Verify

(1) Compute hash $e = H(M \| r) \bmod q$

(2) Compute verification value $v = g^s \times y^e \bmod p$.

(3) If $v == r$ the signature is valid; otherwise reject it.