A Symmetric Encryption Scheme $SE = (K, E, D)$ consists of three algorithms, as follows:

① The randomized key generation algorithm K returns a string k. We let $Keys(SE)$ denote the set of all strings that have non-zero probability of being output by K. The members of this set are called keys. We write $k \xleftarrow{\$} K$ for the operation of executing K and letting k denote the key returned.

② The encryption algorithm E, which might be randomized or stateful, takes a key $K \in Keys(SE)$ and a plaintext $M \in \{0,1\}^*$ to return a ciphertext $C \in \{0,1\}^* \cup \{\perp\}$. We write $C \xleftarrow{\$} E_K(M)$ for the operation of executing E on K and M and letting C denote the ciphertext returned.

③ The deterministic decryption algorithm D takes a key $K \in Keys(SE)$ and a ciphertext $C \in \{0,1\}^*$ to return some $M \in \{0,1\}^* \cup \{\perp\}$. We write $M \leftarrow D_K(C)$ for the operation of executing D on K and C and letting M denote the message returned.

The scheme is said to provide correct decryption if for any key $K \in Keys(SE)$ and any message $M \in \{0,1\}^*$

$$\Pr[C \xleftarrow{\$} E_K(M) : C = \perp \text{ OR } D_K(C) = M] = 1.$$

An encryption algorithm is randomized if it flips coins and uses those to compute its output on a given input K, M. Each time the algorithm is invoked, it flips coins anew. In particular, invoking the encryption algorithm twice on the same inputs may not yield the same response both times.

We say the encryption algorithm is stateful if its operation depends on a quantity called the state that is initialized in some pre-specified way. When the encryption algorithm is invoked on inputs K, M, it computes a ciphertext based on K, M and the current state. It then updates the state, and the new state value is stored.

The receiver, upon receiving a ciphertext C, will run the decryption algorithm with the same key used to create the ciphertext: $D_K(C)$. The decryption algorithm is neither randomized nor stateful.

The One-Time-Pad encryption scheme $SE = (K, E, D)$ is stateful and deterministic. The key generation algorithm simply returns a random k-bit string K, where the key length k is a parameter of the scheme, so that the key space is $Keys(SE) = \{0,1\}^k$. The encryptor maintains a counter ctr which is initially zero.

```
Algorithm Ek(M)
{ Let static ctr ← 0;
  Let m ← |M|;
  if (ctr + m > k)
  { return ⊥; }
  C ← M ⊕ K[ctr+1 ... ctr+m];
  ctr ← ctr + m;
  return ⟨ctr − m, C⟩;
}

Algorithm Dk(⟨ctr, C⟩)
{ Let m ← |M|;
  if (ctr + m > k)
  { return ⊥; }
  M ← C ⊕ K[ctr+1 ... ctr+m];
  return M;
}
```

Modes of operation : There are many ways in which block ciphers can be used for encryption. Some examples are :

1. Electronic Code Book (ECB) mode.
2. Cipher Block Chaining with Random Initial Vector (CBC$) mode.
3. Cipher Block Chaining with Counter (CBCC) mode.
4. Counter mode with a random starting point (CTR$).
5. Counter mode with a counter starting point (CTRC).

ECB mode : Let $E: K \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher. ECB is a symmetric encryption (stateless) scheme $SE = (K, E, D)$. The key-generation algorithm simply returns a random key for the block cipher.

```
algorithm Ek(M)
{ if(|M| mod n ≠ 0 or |M| = 0)
  { return ⊥; }
  Break M into n-bit blocks M[1]...M[m];
  for(i ← 1 to m)
  { C[i] ← Ek(M[i]); }
  C ← C[1]...C[m];
  return C;
}

algorithm Dk(C)
{ if(|C| mod n ≠ 0 or |C| = 0)
  { return ⊥; }
  Break C into n-bit blocks C[1]...C[m];
  for(i ← 1 to m)
  { M[i] ← Ek^-1(C[i]); }
  M ← M[1]...M[m];
  return M;
}
```

CBC $ mode : Let $E: K \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a block cipher. CBC mode with random Initial Vector (IV) yields a stateless symmetric encryption scheme, $SE = (K, E, D)$. The key generation algorithm simply returns a random key for the block cipher.

```
algorithm Ek(M)
{ if (|M| mod n ≠ 0 or |M| = 0)
  { return ⊥; }
  Break M into n-bit blocks M(1) ··· M(m);
  C(0) ← IV ←$ {0,1}^n;
  for (i ← 1 to m)
  { C(i) ← Ek( C(i-1) ⊕ M(i)); }
  C ← C(1) ··· C(m);
  return <IV, C>;
}
algorithm Dk(<IV, C>)
{ if (|C| mod n ≠ 0 or |M| = 0)
  { return ⊥; }
  Break C into n-bit blocks C(1) ··· C(m);
  C(0) ← IV;
  for (i ← 1 to m)
  { M(i) ← Ek^-1( C(i) ⊕ C(i-1)); }
  M ← M(1) ··· M(m);
  return M;
}
```

CBCC mode: Let $E: K \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher. CBC mode with counter IV yields a stateful symmetric encryption scheme, $SE = (K, E, D)$. The key generation algorithm simply returns a random key.

```
algorithm Ek(M)
{  static ctr ← 0;
   if (|M| mod n ≠ 0 or |M| = 0)
   { return ⊥; }
   Break M into n-bit blocks M(1)···M(m);
   if (ctr ≥ 2^n)
   { return ⊥; }
   C(0) ← IV ← [ctr]_n
   for (i ← 1 to m)
   {  C(i) ← Ek( C(i-1) ⊕ M(i)); }
   C ← C(1)···C(m);
   ctr ← ctr + 1;
   return <IV, C>;
}
algorithm Dk(<IV, C>)
{  if (|C| mod n ≠ 0 or |C| = 0)
   { return ⊥; }
   Break C into n-bit blocks C(1)···C(m);
   If (IV + m > 2^n)
   { return ⊥; }
   C(0) ← IV
   for (i ← 1 to m)
   { M(i) ← Ek^-1(C(i)) ⊕ C(i-1); }
   M ← M(1)···M(m);
   return M;
}
```

CTR$ mode: Let $F: K \times \{0,1\}^l \to \{0,1\}^L$ be a family of functions. Then CTR mode over F with a random starting point is a probabilistic, stateless symmetric encryption scheme, $SE = (K, E, D)$. The key-generation algorithm simply returns a random key for F.

```
Algorithm Ek(M)
{ n ← ⌈|M|/L⌉;
  R ←$ {0,1}^l;
  Pad ← Fk(R+1) || Fk(R+2) || ··· || Fk(R+n);
  Pad ← the first |M| bits of Pad;
  C' ← M ⊕ Pad;
  C ← R || C';
  return C;
}

Algorithm Dk(C)
{ if (|C| < l)
  { return ⊥; }
  Parse C into R || C' where |R| = l;
  n ← ⌈|C'|/L⌉;
  Pad ← Fk(R+1) || Fk(R+2) || ··· || Fk(R+n);
  Pad ← the first |C'| bits of Pad;
  M ← C' ⊕ Pad;
  return M;
}
```

CTR mode: Let $F: K \times \{0,1\}^{\ell} \to \{0,1\}^{L}$ be a family of functions. Operating it in CTR mode with a counter starting point is a stateful symmetric encryption scheme, $SE = (K, E, D)$. The key generation algorithm simply returns a random key for F.

```
algorithm Ek(M)
{ static ctr ← 0;
  m ← ⌈|M|/L⌉;
  if (ctr + m ≥ 2^ℓ)
  { return ⊥; }
  Pad ← Fk(ctr+1) || Fk(ctr+2) || ··· || Fk(ctr+m);
  Pad ← the first |M| bits of Pad;
  C ← M ⊕ Pad;
  ctr ← ctr + m;
  return <ctr − m, C>;
}

algorithm Dk(<i, C>)
{ m ← ⌈|C|/L⌉;
  Pad ← Fk(i+1) || Fk(i+2) || ··· || Fk(i+m);
  Pad ← the first |C| bits of Pad;
  M ← Pad ⊕ C;
  return M;
}
```