Blockchain

Nepal

Kerckhoff’s principles

Security by Obscurity

The security of the system cannot depend on the secrecy of the algorithm. It must be assumed that adversary always knows the encryption algorithm.

Advantages of Kerckhoff’s principles

- Secrecy of an algorithm can be broken by reverse engineering the implementation or by leaking out design documents (many examples)
- Published designs undergo public scrutiny
- It is better if security flaws are revealed by “white hat guys”
- Public designs allow for standards

Attack models & Complexity Measures

Complexity Measures

- data complexity
  - expected number of input data units required for the attack –
- storage complexity
  - expected number of storage units required –
- processing complexity
  - expected number of “basic operations” required to process input data and/or fill storage with data
  - parallelization may reduce attack time but not processing complexity!

Attack Models

- ciphertext-only attack
- known-plaintext attack
- (adaptive) chosen-plaintext attack
- (adaptive) chosen-ciphertext attack

Block Ciphers

Block Ciphers

a block cipher is a function

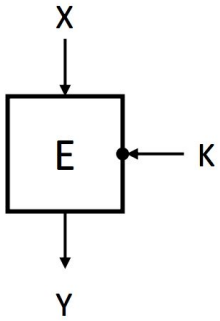
$$E: \{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$$

such that for each $K \in \{0, 1\}^k$

- E(K, .) is an invertible mapping from $\{0, 1\}^n$ to $\{0, 1\}^n$
- E(K, .) cannot be efficiently distinguished from a random permutation
  - » if K is unknown, the output is unpredictable (even parts of it, and even when some input-output pairs are known)
- E(K, X) is also written as $E_K(X)$ (and $E_K^{-1}(Y)$ as $D_K(Y)$)

terminology

- X – plaintext block
- Y – ciphertext block
- E – encryption/coding alg.
- D – decryption/decoding alg.
- K – key
- $\mathcal{K} = \{0, 1\}^k$ – key space



Application

- Confidentiality : encryption of data (of any size)
- Building block for
  - MAC functions
  - Hash functions
  - PRNGs (Pseudo-Random Number Generators)
  - key-stream generators for stream ciphers

Examples

- AES (Rijndael),
- DES,
- RC5,
- Blowfish,
- Skipjack,
- IDEA, ...

How to choose?

- Design assumptions vs. Application requirements
  - e.g. software implementations, hardware optimizations etc.
- Efficiency
  - Speed, Memory size, Code size (or number of gates)
- Security
  - key size (resistance to brute force exhaustive key search)
  - algebraic properties
  - complexity of best known attacks
  - openness of specification (security by obscurity vs. Kerckhoffs' principle)
- Patent issues

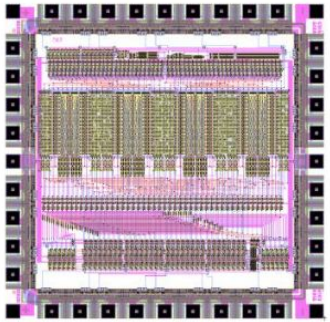
Data Encryption Standard (DES)

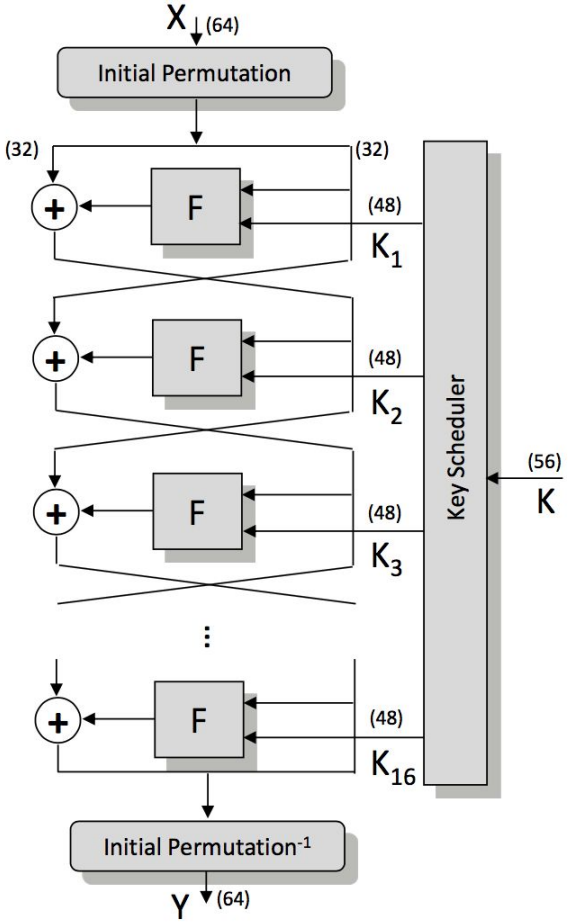
→ based on Lucifer, a cipher developed by IBM in the 70’

→ symmetric key block cipher

→ features:

- Feistel structure (same structure can be used for encoding and decoding)
- number of rounds: 16
- input block size: 64 bits
- output block size: 64 bits
- key size: 56 bits



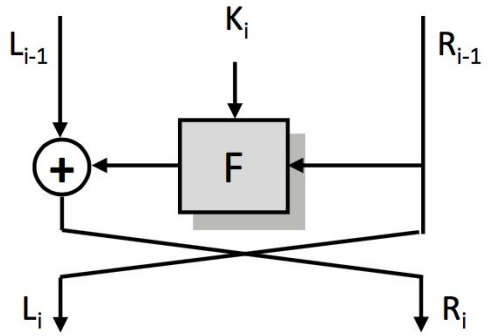


Properties of Feistel Cipher

- round i maps (L_{i-1}, R_{i-1}) into (L_i, R_i) as follows:

  $L_i = R_{i-1}$

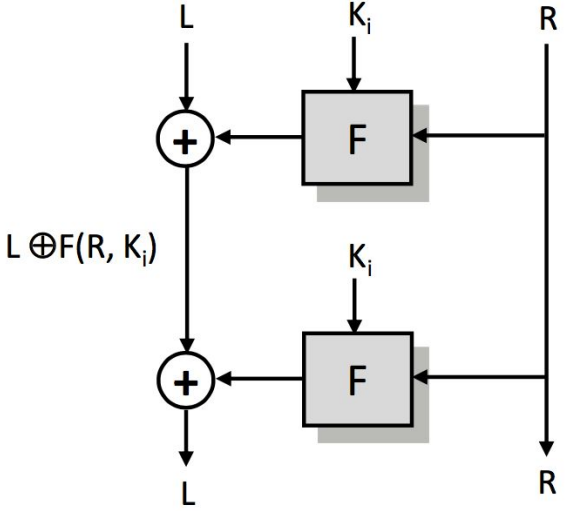
  $R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$



- a Feistel cipher is always invertible even if F is not invertible:

  $R_{i-1} = L_i$

  $L_{i-1} = R_i \oplus F(L_i, K_i)$



- decryption can be achieved using the same r-round process with the round keys used in reverse order (K_r through K_1)

DES in Java - Key

```
// 1. Create/generate a secret key

KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
SecretKey desKey = keyGenerator.generateKey();

// OR

SecretKeySpec desKey = new SecretKeySpec("password".getBytes(), "DES");

// 2. Message to encrypt

byte[] message = "Hello world!".getBytes();
```

DES in Java - Encryption

```
// Get cipher instance

Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

// Encrypt the message

desCipher.init(Cipher.ENCRYPT_MODE, desKey);

byte[] encryptedMessage = desCipher.doFinal(message);

System.out.println("Encrypted message:"+new String(encryptedMessage));
```

DES in Java - Decryption

```
// Get cipher instance

Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

// Decrypt the message

desCipher.init(Cipher.DECRYPT_MODE, desKey);

byte[] origMessage = desCipher.doFinal(encryptedMessage);

System.out.println("Plain message:"+new String(origMessage));
```