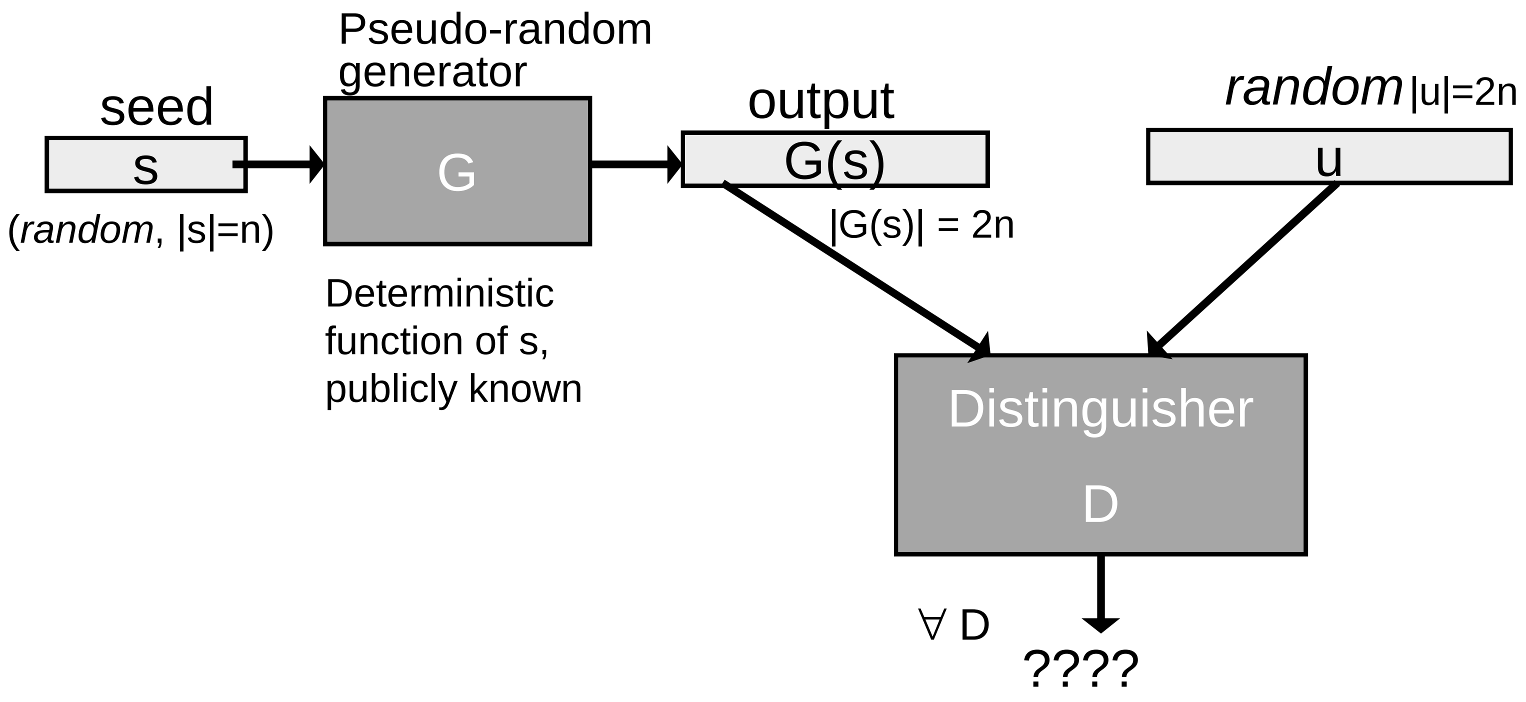
Introduction to Cryptography

Lecture 3

Benny Pinkas

Pseudo-random generator



Pseudo-random generators

- Pseudo-random generator (PRG)
  - G: $\{0,1\}^n \Rightarrow \{0,1\}^m$
    - A deterministic function, computable in polynomial time.
    - It must hold that m > n. Let us assume m=2n.
    - The function has only 2^n possible outputs.
- Pseudo-random property:
  - $\forall$ polynomial time adversary D, (whose output is 0/1) if we choose inputs $s \in_R \{0,1\}^n$, $u \in_R \{0,1\}^m$, (in other words, choose s and u uniformly at random), then

    it holds that D(G(s)) is similar to D(u)

    namely, | Pr[D(G(s))=1] - | Pr[D(u)=1] | is negligible

P vs. NP

- If P=NP then PRGs do not exist (why?)
- So their existence can only be conjectured until the P=NP question is resolved.

Using a PRG for Encryption

- Replace the one-time-pad with the output of the PRG

- Key: a (short) random key $k \in \{0,1\}^{|k|}$.
- Message $m= m_1,\ldots,m_{|m|}$.
- Use a PRG $G : \{0,1\}^{|k|} \rightarrow \{0,1\}^{|m|}$
- Key generation: choose $k \in \{0,1\}^{|k|}$ uniformly at random.
- Encryption:
  - Use the output of the PRG as a one-time pad. Namely,
  - Generate $G(k) = g_1,\ldots,g_{|m|}$
  - Ciphertext $C = g_1 \oplus m_1,\ldots, g_{|m|} \oplus m_{|m|}$

- This is an example of a *stream cipher.*

Security of encryption against polynomial adversaries

- Perfect security (previous equivalent defs):
  - (indistinguishability) $\forall$ $m_0, m_1 \in M$, $\forall c$, the probability that c is an encryption of m_0 is equal to the probability that c is an encryption of m_1.
  - (semantic security) The distribution of m given the encryption of m is the same as the a-priori distribution of m.
- Security of pseudo-random encryption (equivalent defs):
  - (indistinguishability) $\forall$ $m_0, m_1 \in M$, no *polynomial time* adversary D can distinguish between the encryptions of m_0 and of m_1. Namely, $Pr[D(E(m_0))=1] \approx Pr[D(E(m_1))=1)$
  - (semantic security) $\forall$ $m_0, m_1 \in M$, a polynomial time adversary which is given $E(m_b)$, where $b \in_r \{0,1\}$, succeeds in finding b with probability $\approx$ ½.

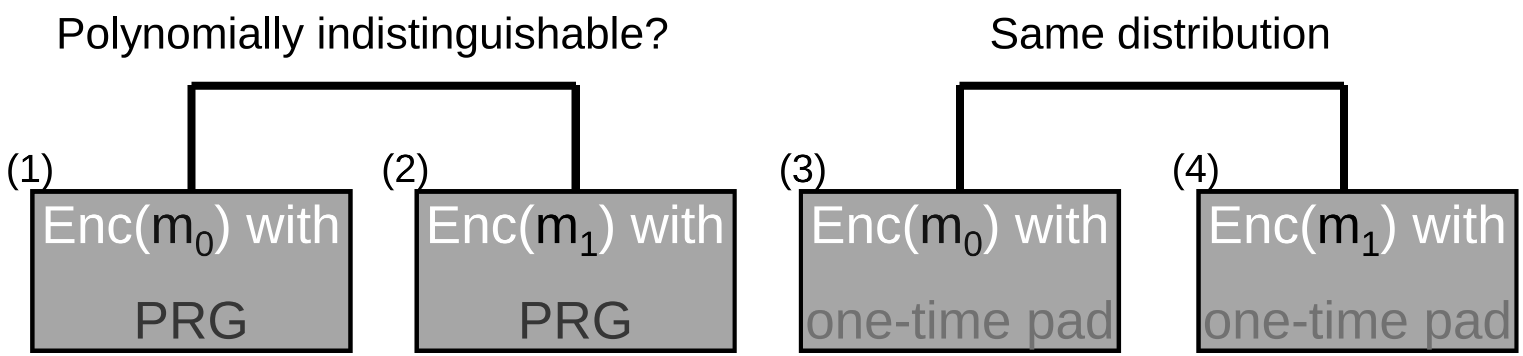
Proofs by reduction

- We don't know how to prove unconditional proofs of computational security; we must rely on assumptions.
  - We can simply assume that the encryption scheme is secure. This is bad.
  - Instead, we will assume that some low-level problem is hard to solve, and then prove that the cryptosystem is secure under this assumption.
  - (For example, the assumption might be that a certain function G is a pseudo-random generator.)
  - Advantages of this approach:
    - It is easier to design a low-level function.
    - There are (very few) "established" assumptions in cryptography, and people prove the security of cryptosystem based on these assumptions.

Using a PRG for Encryption: Security

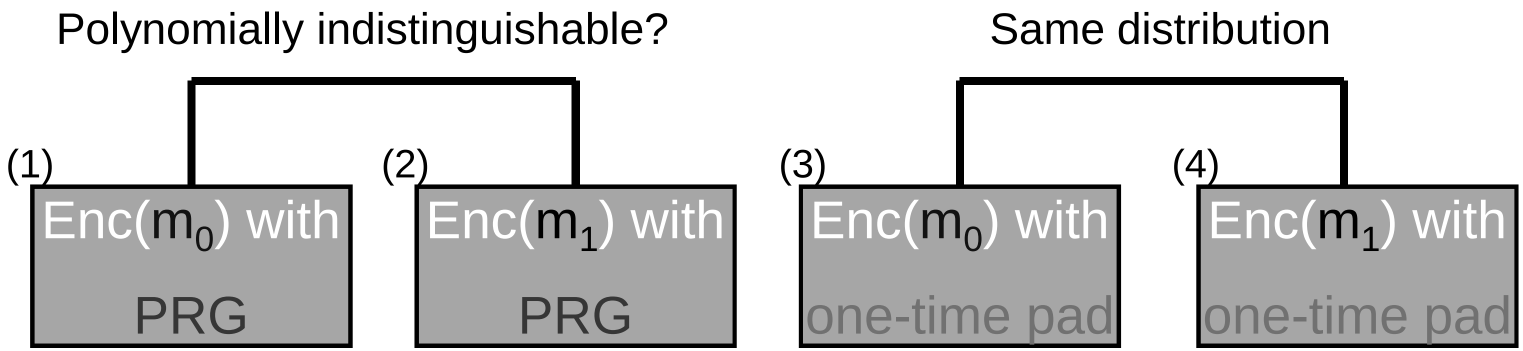
- The output of a pseudo-random generator is used instead of a one-time pad.
- Proof of security by reduction:
  - The assumption is that the PRG is strong (its output is indistinguishable from random).
  - We want to prove that in this case the encryption is strong (it satisfies the indistinguishability definition above).
  - In other words, prove that if one can break the security of the encryption (distinguish between encryptions of m_0 and of m_1), then it is also possible to break the security of the PRG (distinguish its output from random).

Proof of Security



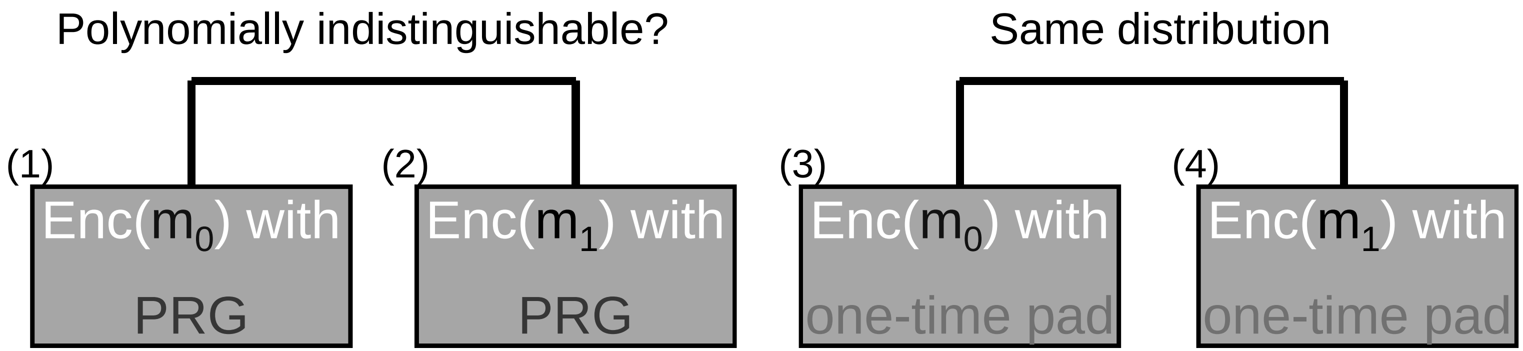
- Suppose that there is a distinguisher algorithm D'() which distinguishes between (1) and (2)
- We know that no D'() can distinguish between (3) and (4)
- We are given a string S and need to decide whether it is drawn from a pseudorandom distribution or from a uniformly random distribution
- We will use S as a pad to encrypt a message.

Proof of Security



- Recall: we assume that there is a D'() which always distinguishes between (1) and (2), and which distinguishes between (3) and (4) with probability ½.
- Choose a random b∈{0,1} and compute $m_b \oplus S$. Give the result to D'().
  - if S was chosen uniformly, D'() must distinguish (3) from (4). (prob=½)
  - if S is pseudorandom, D'() must distinguish (1) from (2). (prob=1)
- If D'() outputs b then declare "pseudorandom", otherwise declare "random".
  - if S was chosen uniformly we output "pseudorandom" with prob ½.
  - if S is pseudorandom we output "pseudorandom" with prob 1.

Proof of Security



- Recall: we assume that there is a D'() which always distinguishes between (1) and (2), and which distinguishes between (3) and (4) with probability ½.
- Choose a random b∈{0,1} and compute $m_b \oplus S$. Give the result to D'().
  - if S was chosen uniformly, D'() must distinguish (3) from (4). (prob=½)
  - if S is pseudorandom, D'() must distinguish (1) from (2). (prob=½+δ)
- If D'() outputs b then declare "pseudorandom", otherwise declare "random".
  - if S was chosen uniformly we output "pseudorandom" with prob ½.
  - if S is pseudorandom we output "pseudorandom" with prob ½+δ.

Stream ciphers

- Stream ciphers are based on pseudo-random generators.
  - Usually used for encryption in the same way as OTP
- Examples: A5, SEAL, RC4.
  - Very fast implementations.
  - RC4 is popular and secure when used correctly, but it was shown that its first output bytes are biased. This resulted in breaking WEP encryption in 802.11.

- Some technical issues:
  - Stream ciphers require *synchronization* (for example, if some packets are lost in transit).

RC4

- Designed by Ron Rivest. Intellectual property belongs to RSA Inc.
  - Designed in 1987.
  - Kept secret until the design was leaked in 1994.
- Used in many protocols (SSL, etc.)
- Byte oriented operations.
- 8-16 machine operations per output byte.
- First output bytes are biased ☹

RC4 initialization

Word size is a single byte.

Input: $k_0;\ldots;k_{255}$ (if key has fewer bits, pad it to itself sufficiently many times)

1. $j = 0$
2. $S_0 = 0;\ S_1 = 1;\ldots\ ;\ S_{255} = 255$
3. Let the key be $k_0;\ldots;k_{255}$
4. For $i = 0$ to 255
   - $j = (j + S_i + k_i) \bmod 256$
   - Swap S_i and S_j

(note that S is a permutation of 0,…,255)

RC4 keying stream generation

An output byte B is generated as follows:

- $i = i + 1 \mod 256$
- $j = j + S_i \mod 256$
- Swap S_i and S_j
- $r = S_i + S_j \mod 256$
- Output: $B = S_r$

B is xored to the next byte of the plaintext.

(since S is a permutation, we want that B is uniformly distributed)

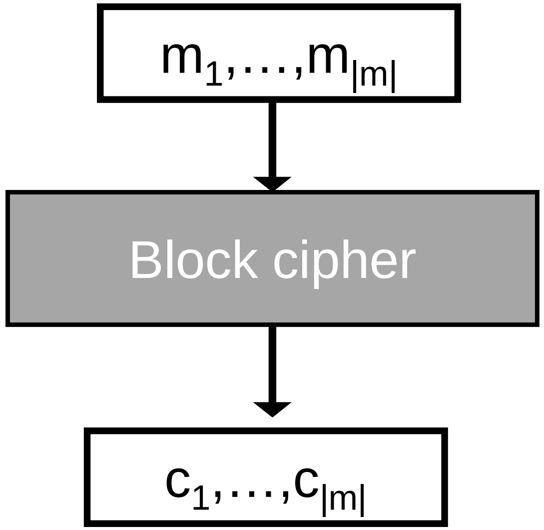
Bias: The probability that the first two output bytes are 0 is $2^{-16}+2^{-23}$

Block Ciphers

- Plaintexts, ciphertexts of fixed length, |m|. Usually, |m|=64 or |m|=128 bits.
- The encryption algorithm E_k is a *permutation* over $\{0,1\}^{|m|}$, and the decryption D_k is its inverse. (They *are not* permutations of the bit order, but rather of the entire string.)

- Ideally, use a *random* permutation.
  - Can only be implemented using a table with $2^{|m|}$ entries ☹
- Instead, use a *pseudo-random* permutation*, keyed by a key k.
  - Implemented by a computer program whose input is m,k.

  - (*) will be explained shortly

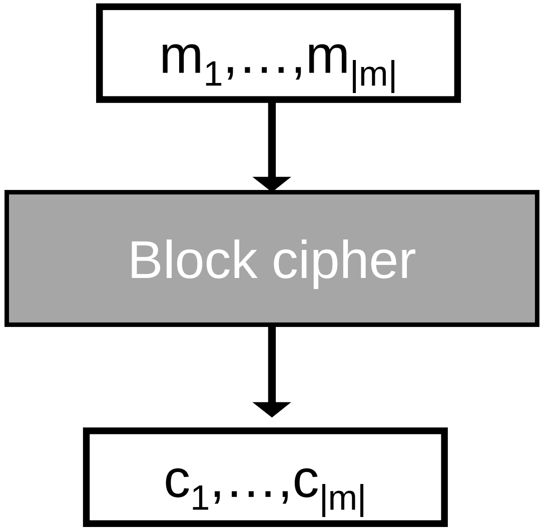


Block Ciphers

- Modeled as a pseudo-random permutation.

- Encrypt/decrypt whole blocks of bits
  - Might provide better encryption by simultaneously working on a block of bits
  - One error in ciphertext affects whole block
  - Delay in encryption/decryption
  - There was more research on the security of block ciphers than on the security of stream ciphers.

- Different *modes of operation* (for encrypting longer inputs)



Pseudo-random functions

- $F : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$
  - The first input is the key, and once chosen it is kept fixed.
  - For simplicity, assume $F : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$
  - $F(k,x)$ is written as $F_k(x)$

- F is pseudo-random if $F_k()$ (where k is chosen uniformly at random) is indistinguishable (to a polynomial distinguisher D) from a function *f* chosen at random from all functions mapping $\{0,1\}^n$ to $\{0,1\}^n$
  - There are 2^n choices of F_k, whereas there are $(2^n)^{2^n}$ choices for *f*.
  - The distinguisher D's task:
    - We choose a function G. With probability ½ G is F_k (where $k \in_R \{0,1\}^n$), and with probability ½ it is a random function *f*.
    - D can compute $G(x_1),G(x_2),\ldots$ for any $x_1,x_2,\ldots$ it chooses.
    - D must say if $G=F_k$ or $G=f$.
    - F_k is pseudo-random if D succeeds with prob ½+negligible..

Pseudo-random permutations

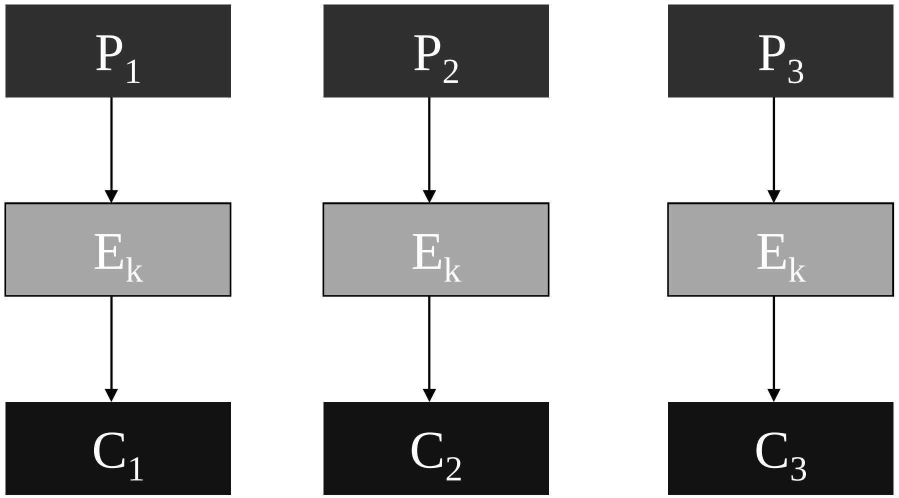
- $F_k(x)$ is a keyed permutation if for every choice of k, $F_k()$ is one-to-one.
  - Note that in this case $F_k(x)$ has an inverse, namely for every y there is exactly one x for which $F_k(x)=y$.

- $F_k(x)$ is a pseudo-random permutation if
  - It is a keyed permutation
  - It is indistinguishable (to a polynomial distinguisher D) from a permutation *f* chosen at random from all permutations mapping $\{0,1\}^n$ to $\{0,1\}^n$.
    - 2^n possible values for F_k
    - $(2^n)!$ possible values for a random permutation

Block ciphers

- A block cipher is a function $F_k(x)$ of a key k and an |m| bit input x, which has an |m| bit output.
  - $F_k(x)$ is a keyed permutation
- How can we encrypt plaintexts longer than |m|?
- Different modes of operation were designed for this task.

ECB Encryption Mode (Electronic Code Book)

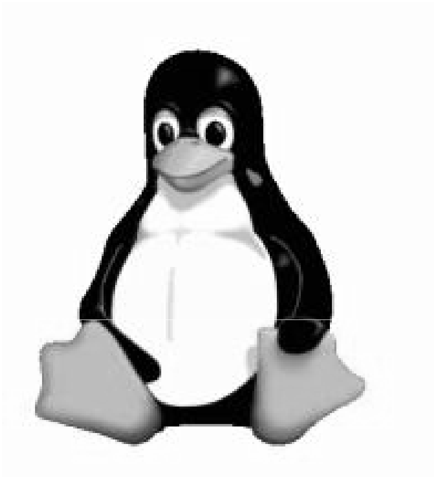


Namely, encrypt each plaintext block separately.

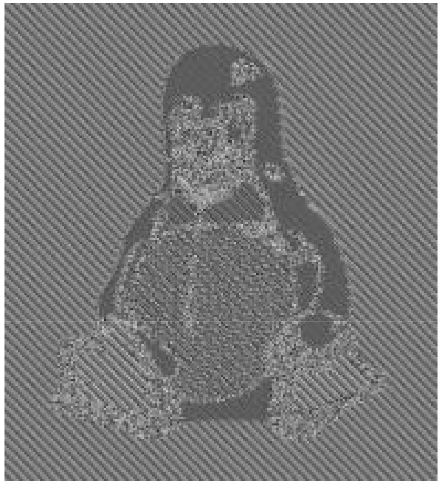
Properties of ECB

- Simple and efficient ☺
- Parallel implementation is possible ☺
- Does not conceal plaintext patterns ☹
  - Enc(P_1, P_2, P_1, P_3)
- Active attacks are easy ☹ (plaintext can be easily manipulated by removing, repeating, or interchanging blocks).

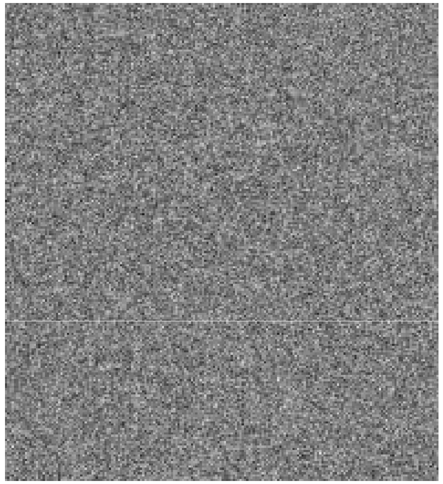
Encrypting bitmap images in ECB mode



original

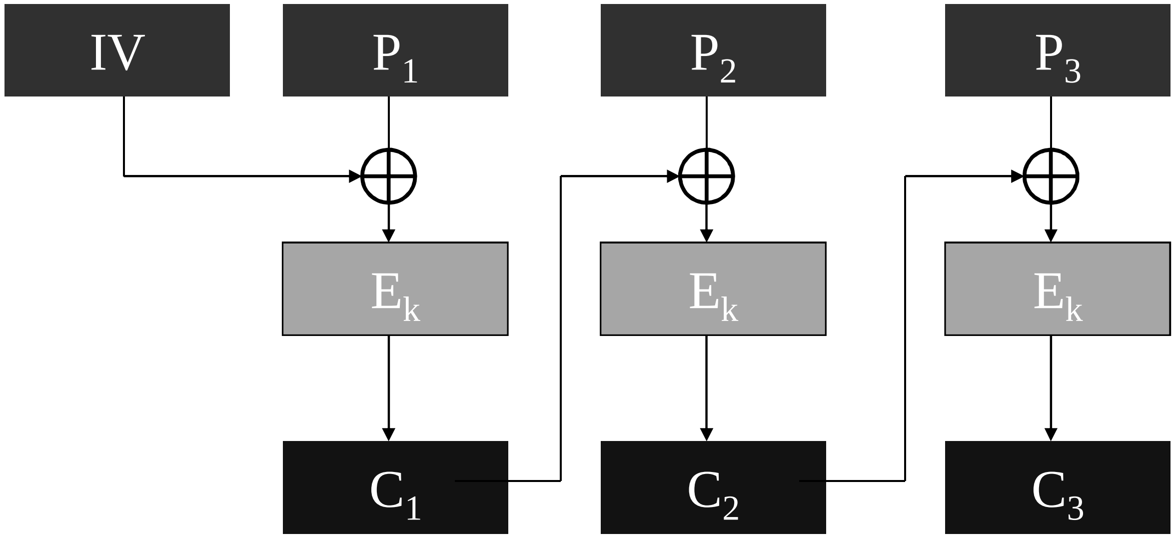


encrypted using ECB mode



encrypted using a secure mode

CBC Encryption Mode (Cipher Block Chaining)



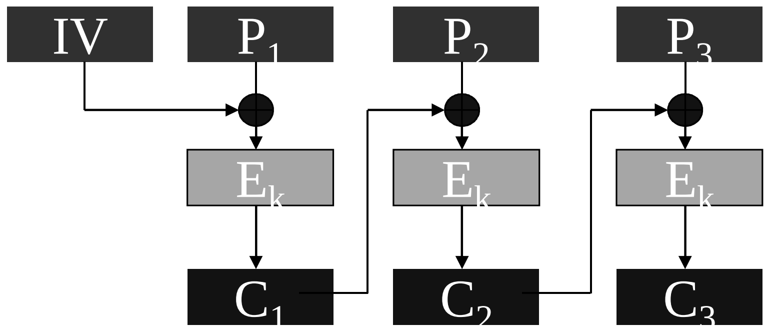
Previous *ciphertext* is XORed with current *plaintext* before encrypting current block.

An initialization vector IV is used as a “seed” for the process.

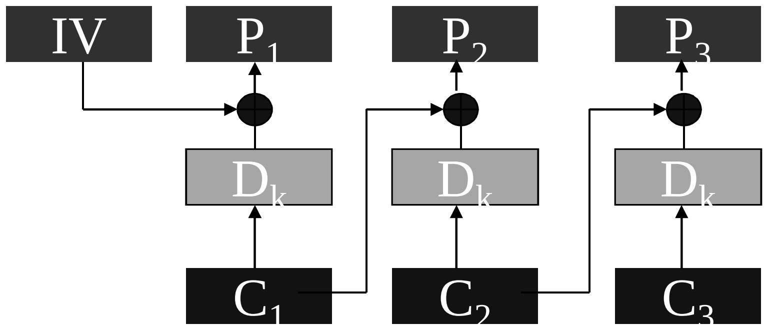
IV can be transmitted in the clear (unencrypted).

CBC Mode

Encryption:



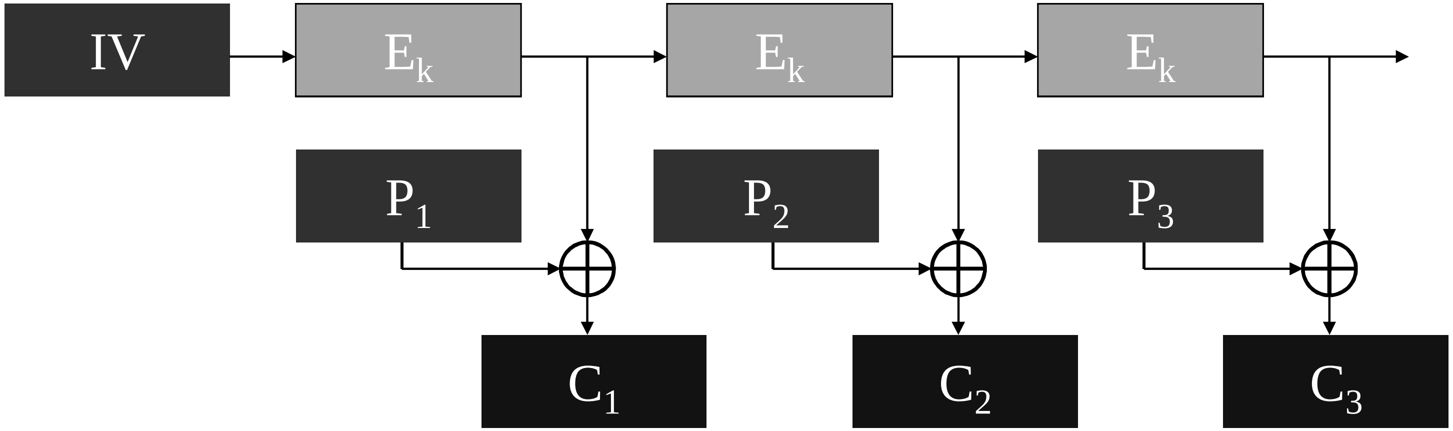
Decryption:



Properties of CBC

- Asynchronous: the receiver can start decrypting from any block in the ciphertext. ☺
- Errors in one *ciphertext* block propagate to the decryption of the next block (but that's it). ☺
- Conceals plaintext patterns (same block $\Rightarrow$ different ciphertext blocks) ☺
  - If IV is chosen at random, and E_K is a pseudo-random permutation, CBC provides chosen-plaintext security.
  - But if IV is fixed, CBC does not even hide not common *prefixes.*
- No parallel implementation is known ☹
- Plaintext cannot be easily manipulated ☺
- Standard in most systems: SSL, IPSec, etc.

OFB Mode (Output FeedBack)



- An initialization vector IV is used as a “seed” for generating a sequence of “pad” blocks
  - $E_k(IV)$, $E_k(E_k(IV))$, $E_k(E_k(E_k(IV)))$,…
- Essentially a stream cipher.
- IV can be sent in the clear. Must never be repeated.

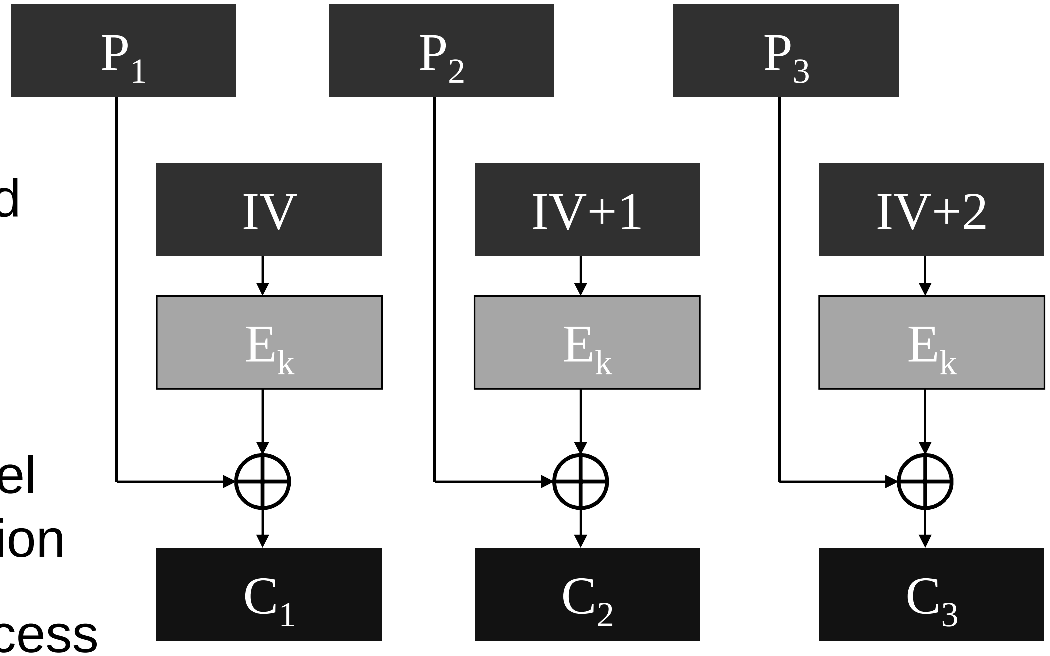
Properties of OFB

- Essentially implements a synchronous stream cipher. I.e., the two parties must know s_0 and the current bit position.
  - A block cipher can be used instead of a PRG.
  - The parties must synchronize the location they are encrypting/decrypting. ☹

- Conceals plaintext patterns. If IV is chosen at random, and E_K is a pseudo-random permutation, CBC provides chosen-plaintext security. ☺

- Errors in ciphertext do not propagate ☺
- Implementation:
  - Pre-processing is possible ☺
  - No parallel implementation is known ☹
- Active attacks (by manipulating the plaintext) are possible ☹

CTR (counter) Encryption Mode



IV is selected as a random value

- easy parallel implementation
- random access
- preprocessing