Confidentiality – Symmetric Key Ciphers

The first topic we face when we discuss about cryptography is *confidentiality*, that is the property for which only the allowed parties can have the confidential information, other ones have not to be able to retrieve any information about the encrypted data. For sure the confidentiality can be achieved by means of cryptography, so the parties whoo need a confidential conversation have to encrypt messages by means of *ciphers*. This kind of cryptography is based on the concept of secret key, which is an item that allows to read the confidential information. The model we study now for confidentiality uses a Symmetric Key scheme. Symmetric Key scheme means that the two parties that want a confidential conversation they both know the secret and can use it in order to encrypt/decrypt the messages. One thing that is required is that they pre-shared the secret in a secure way.

The model is the following:

Alice and Bob share

- An encryptor E
- A secret key K

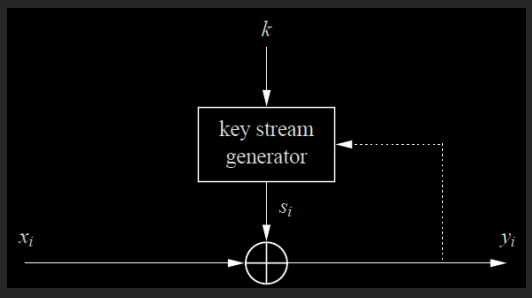
They communicate encrypting messages with E using the shared secret key K (which has to be exchanged in advance in a secure manner). The adversary, who cannot knows the key K, has not to be able to decrypt the messages in order to retrieve secret informations.

In order to encrypt data, two main cipher-approaches can be used and they are **stream ciphers** and **block ciphers**. We are going to see them.

STREAM CIPHERS

The idea of the stream ciphers is to simulate the *one time pad* cipher, which is a well-known PERFECT cipher. The scheme is the following: the secret key is used as a seed in order to start a key-stream, at each step of the computation a byte of the key stream is generated. From this we quickly understand that with a stream cipher we can generate a key of whatever length, no matter how big is our message, and this is a good point. There are two types of stream ciphers:

- Syncrhonous stream-ciphers: the i^{th} byte of the key is function of only the state of the cipher
- Asynchronous stream-ciphers: the i^{th} byte of the key is function of both the state of the cipher and the previous bytes of the ciphertext



Here is the scheme which describes a strem cipher. x_i is the i^{th} byte of the plaintext, s_i is the i^{th} byte of the key stream, y_i is the i^{th} byte of the ciphertext. Note the dotted feedback arrow. If it is present then we are talking about *asynchronous* stream cipher, otherwise we are talking about *synchronous* stream cipher.

In order to decrypt, the receiver must compute the stream cipher starting from the seed, which is the secret shared key, and then must XOR the ciphertext received with the key stream in order to obtain the plaintext.

Stream ciphers typically execute at a higher speed than block ciphers and have lower hardware complexity. However, stream ciphers can be susceptible to serious security problems if used incorrectly (stream cipher attacks). In particular, the same starting state (seed) must never be used twice. The security of a stream

cipher relies on a secret, that is used to start the seed and so the whole stream key. Knowing the secret, the adversary will be able to compute the stream key itself and so he will be able to decrypt every message.

RC-4

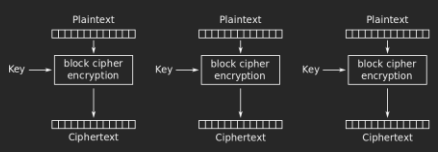
One of the most popular stream ciphers is RC-4, which is a synchronous stream cipher and it is very fast since one byte of key stream requires 8-16 instructions. It is composed by two parts:

- Key Scheduling Algorithm (KSA), which uses a variable-length key in order to perform a pseudo-random permutation of the first 256 natural numbers. After this phase, the permutation represents the initial state of the cipher, from which we will start to generate the bytes of the key stream
- Pseudo-Random Generation Algorithm (PRGA), which is a cycle that at each step modifies the 'state' of the stream cipher in order to generate the next byte of the key stream, which will be XORed with the corresponding byte of the plain text in order to get the corresponding byte of the cipher text. Note that the cycle is eventually endless, therefore it is a while(1), for which a key stream of any length can be generated.

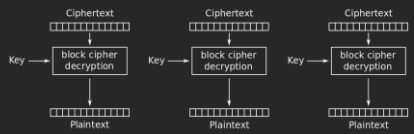
BLOCK CIPHERS

Block ciphers are kinds of ciphers which make use of an encryptor algorithm E_K that encrypts blocks of a certain length. The idea is the following, assume the encryptor works with blocks of h bits, then divide the message in blocks of h bits (eventually add padding to the last block) and then encrypt them with a secret shared key by means of the block encryptors. Note that the length of the block and the length of the key are fixed, but not necessarily of the same length. Examples of encryptors are DES, 3-DES, Blowfish, AES and so on and so forth. Please note that encryptors and block ciphers are different things, because an encryptor is the function that encrypts a single block of h bit, while a block cipher (which makes us of encryptors) is the whole scheme that computes all the encryptions of all the blocks of the plaintext. More than on the encryptors, we are interested on the several modes of operation designed for the block ciphers, which allow block ciphers to provide confidentiality for messages of arbitrary length. For each one of them, we will be studying certain properties that the specific mode of operation can satisfy or not, and they are *error propagations*, *preprocessing* and *parallel implementation*. Let's see the several modes of operation…

ECB (Electronic Code Block)



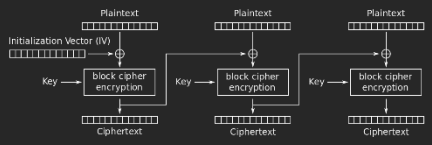
Electronic Codebook (ECB) mode encryption



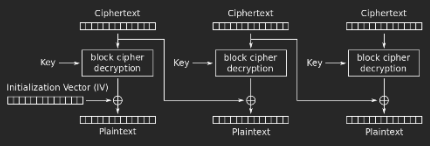
Electronic Codebook (ECB) mode decryption

It is the simplest one, simply divide the plaintext in blocks and encrypt each one of them separately. Error propagation does not occur for the independence of the blocks, parallel implementation is possible due to the independence of the blocks, precomputation isn't since we need the ciphertext before we can do any step. Even if it is the simplest and fastest, it is the weakest since it does not hide plaintext patterns (remember Linux penguin image). The decrypting counterpart is trivial. ECB is not used since it is too weak.

CBC (Cipher Block Chaining)



Cipher Block Chaining (CBC) mode encryption

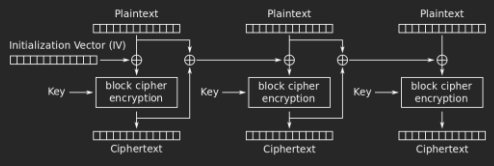


Cipher Block Chaining (CBC) mode decryption

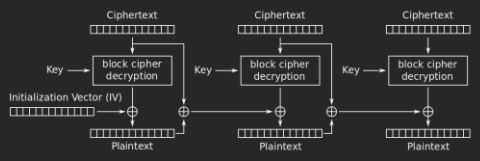
Similar to ECB, but here before encrypting the i^{th} plaintext block, we XOR it with $i\text{-}1^{th}$ ciphertext block. Since the first block does not have a previous block, an Initialization Vactor IV is xored with the first plaintext block. What is important to notice is that it is a good practice to change the IV for every encrypt session, because even if the file is the same, the ciphertext will be different and so the attacker cannot understand the file has not changed. For what concerns error propagation, if a block of ciphertext is received with a single bit wrong, it will affect the corresponding bit of the next plaintext and will completely damage the current plaintext, but no other blocks are affected, only two. For what concerns *parallel implementation*, it is not possible when encrypting (since every block needs the encryption of the previous one), but it is possible when encrypting (the current block needs the previous ciphertext block, but we have all of them). This is good because one expects that decryptions are much more frequent than encryptions. No precomputation can be performed.

There is a common practice used when the plaintext size is not a multiple of the size of the blocks, and here the *ciphertext stealing* tecnique is performed.

PCBC (Propagating Cipher Block Chaining)



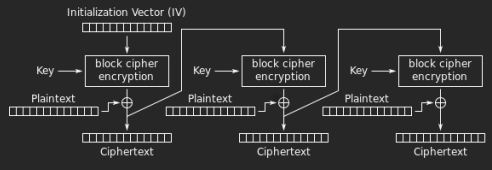
Propagating Cipher Block Chaining (PCBC) mode encryption



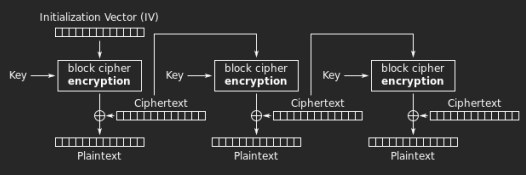
Propagating Cipher Block Chaining (PCBC) mode decryption

It is an extension of the CBC. We can see that the scheme is similar, what changes is that before the i^{th} block of plaintext is encrypted, it is XORed with the XOR between both $i\text{-}^{1th}$ plaintext and ciphertext block. In this manner, an error of a ciphertext block will be propagated till the end of the chain. What is happening here is that we can no longer decrypt in parallel as we did for CBC, in fact here we need the previous plaintext block in order to compute the current one. No precomputation can be performed. Actually it has been discovered a nice property, that states that if two adjacent blocks of ciphertext are exchanged than this does not affect the decryption of the subsequent blocks. For this reason PCBC is no longer used since Kerberos 5.

CFB (Cipher Feedback)



Cipher Feedback (CFB) mode encryption



Cipher Feedback (CFB) mode decryption

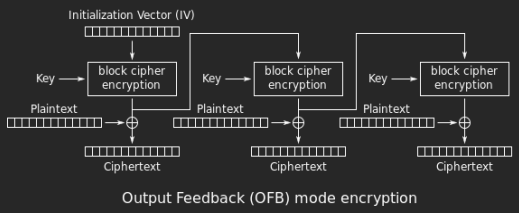
As we can see, the previous ciphertext becomes the input of the current block cipher. So the i^{th} block of the ciphertext is the result of the XOR between the i^{th} plaintext and the encryption of the $i\text{-}1^{th}$ block of the ciphertext. Whatching the decryption scheme, by construction in order to decrypt we use again the encryptor, because the i^{th} ciphertext is:

$$C_i = E_K(C_{i-1}) \oplus P_i$$

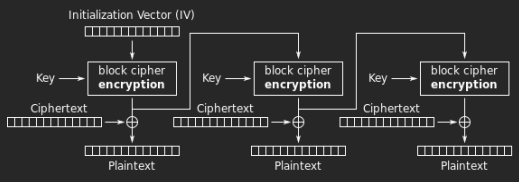
So in order to decrypt we have to XOR C_i with $E_K(C_{i-1})$ in order to retrieve P_i. Errors do not propagate, a single bit error only affects the corresponding bit of the corresponding block and completely damages the next block. Precomputation is not possible, parallel

encrypting is not possible, parallel decryption is. A common practice is using the CFB with shift register, in such a way we can encrypt a block of whatever size and so we do not need to add padding to the last block.

OFB (Output Feedback)



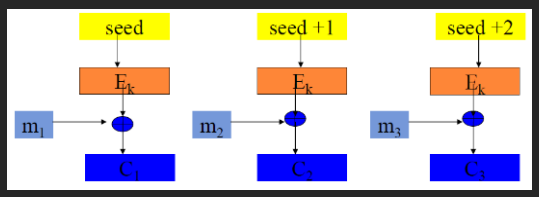
Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

The scheme is very similar to the previous one, what changes is that the input of the next encryptor is the output of the previous one. In this way we can precompute the "key stream" and after that, once received the ciphertext, decrypt in parallel. We can also encrypt in parallel once precomputed the "key stream". Error propagation does not happen, moreover it only affects the corresponding block. What er note here that this scheme corresponds to a *synchronous stream cipher* since the current byte of the key stream only depends on the previous one, while CFB has to be considered an *asynchronous* stream cipher. Warning: reusing the same IV twice in this mode of operation completely destroyes the security since same IV will produce the same stream key.

CTR (Counter Mode)



The idea is simple, we use a seed, and we generate the key stream by encrypting the consecutive values of the seed, that is considered a counter. Then for each encrypted seed, we XOR it with the corresponding block of the plaintext obtaining the ciphertext block.

It has many characteristics in common with OFB: errors do not propagate, and parallel implementation is possible after performing precomputation of the key stream, furthermore, reusing the same seed completely destroyes the security.