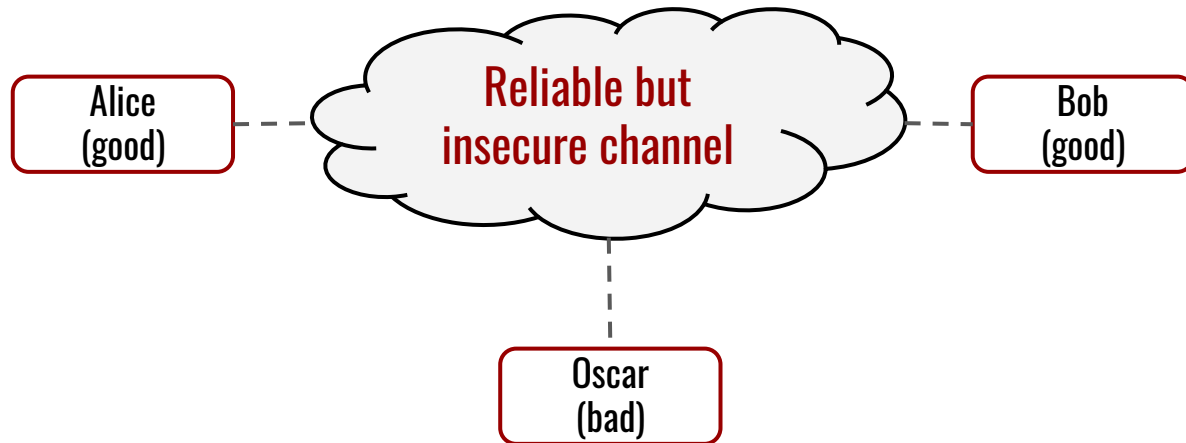


Symmetric Ciphers I

Computer and Network Security

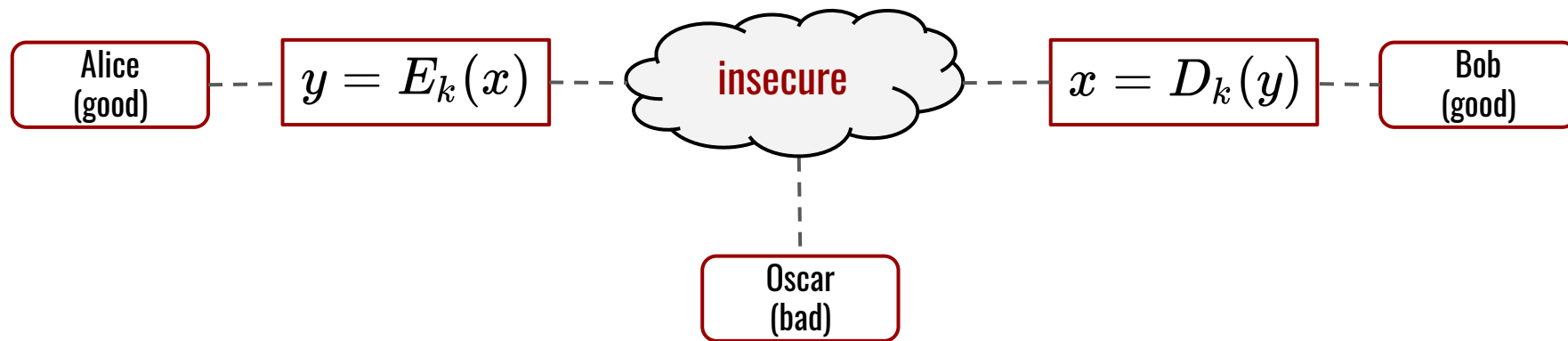
Emilio Coppa

Communication Model



How can we build a secure communication channel between Alice and Bob?

Secure Communication Channel using Symmetric Cipher



Q: why do we need a key? Cannot just we use E and D?

Remark. We assume that the symmetric key k is exchanged through a secure channel (see protocols for key exchange)

Example #1: shift cipher (Caesar's cipher)

Idea: shift letters in the alphabet by k position (if $k=13$ then is called ROT13)

$$y_i = E_k(x_i) = (x_i + k) \bmod 26$$

$$x_i = D_k(y_i) = (y_i - k) \bmod 26$$

right shift 23 positions

E.g., $k=23$

Plain: **ABCDEFGHIJKLMNOPQRSTUVWXYZ**

Cipher: **XYZABCDEFGHIJKLMNOPQRSTUVWXYZ**

Plaintext: THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD

Example #2: substitution cipher

Idea: map each letter to a different letter (k is the mapping, chosen randomly)

E.g.,

Plain: **ABCDEFGHIJKLMNOPQRSTUVWXYZ**

Cipher: ZEBRASCDFGHIJKLMNOPQTUVWXY

Plaintext: FLEE AT ONCE. WE ARE DISCOVERED!

Ciphertext: SIAA ZQ LKBA. VA ZOA RFPBLUAOAR!

Can we break these two ciphers?

Attacks:

- Brute force (or exhaustive key search):

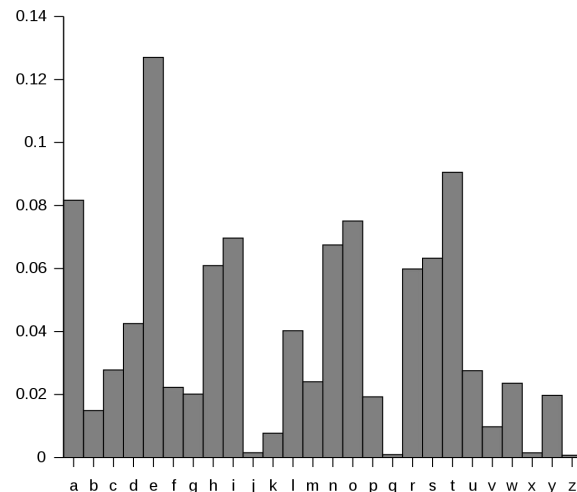
- Shift cipher: try values of k in $[0, 26]$
- Substitution cipher: try all $26!$ mapping = 2^{88}

⇒ trivial to break by hand

⇒ it seems to be secure....

- Letter Frequency Analysis:

- we exploit that the statistical property of the text are not affected by the substitution (e.g., on average we expect the most frequent letter in the cipher text to be the letter e in the plain text). Generalize this idea by looking at pair (e.g., an) and triples (e.g., “the”) to break the entire algorithm.



Can we break these two ciphers? **YES**

Attacks:

- Brute force (or exhaustive key search):
- Letter Frequency Analysis

Lessons learned:

1. key space should be large to avoid brute force.
2. ciphers should hide the statistical properties of the encrypted plaintext. The ciphertext symbols should appear to be random.

Perfect cipher

Given a plaintext space $= \{0, 1\}^n$, D known, and a ciphertext y then the probability that exists a key k such that $D_k(y) = x$ for any plaintext x is equal to the apriori probability that x is the plaintext:

$$Pr[x|y] = Pr[x]$$

In other words, **the ciphertext does not reveal any information on the plaintext.**

$$Pr[x|y] = Pr[x \wedge y] / Pr[y] \quad \text{(conditional probability)}$$

$$Pr[x \wedge y] = Pr[x|y] Pr[y] = Pr[y|x] Pr[x] \quad \text{(Bayes)}$$

$$Pr[x \wedge y] = Pr[x] Pr[y] \quad \text{(if x and y are independent)}$$

Then:

$$Pr[x] Pr[y] = Pr[y|x] Pr[x] \Rightarrow Pr[y] = Pr[y|x]$$

One-Time-Pad (OTP)

- Plaintext space: $\{0,1\}^n$
- Key space: $\{0,1\}^n$
- Symmetric scheme, key chosen using a True Random Number Generator (TRNG)
- key is only used once (i.e., two messages must be encrypted with two different keys)

$$y = E_k(x) = x \oplus k$$

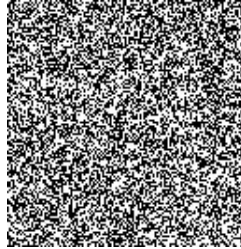
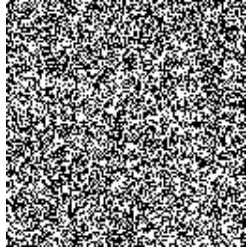
$$x = D_k(y) = y \oplus k = (x \oplus k) \oplus k = x \oplus (k \oplus k) = x \oplus 0$$

OTP is a perfect cipher!

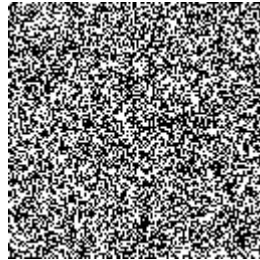
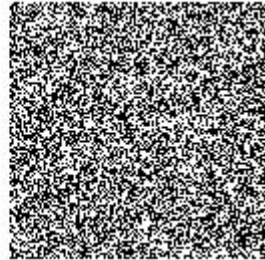
Practical problems: $|k| = |x|$, requires TRNG,
need to change key for each message.

Why cannot we use the same keystream twice in OTP?

SEND
CASH



SEND
CASH



Can we have a perfect cipher with $|k| < |x|$?

Shannon's Theorem: A cipher cannot be perfect if the size of its key space is less than the size of its message space.

Proof by contradiction.

$$2^{|k|} < 2^{|x|}$$

$$Pr[y_0] > 0 \quad (\text{ciphertext must exist})$$

$$S = \{D_k(y_0) : k \in K\} \quad (K \text{ is the set of all possible keys})$$

Then: $\exists x$ such that $x \notin S$

If we know: $\forall k \in K : E_k(x) \neq y_0 \Rightarrow Pr[y_0] = 0$

Symmetric Ciphers

Two main approaches:

- **stream ciphers:**
 - inspired by OTP
 - given a secret key, generate a byte of stream called **keystream** that has the same length as the message
 - encrypt/decrypt bits from x individually using a XOR operation with the keystream (as in OTP)
- **block ciphers:**
 - split the message x in blocks of fixed size
 - encrypt/decrypt each block
 - different “operation mode”: process blocks independently, chain blocks, etc.

Stream Ciphers

Given plaintext x (x_i is the i -th bit from x), keystream s (where $|x|=|s|$, s_i is the i -th bit from s):

- Encryption: $y_i = E_{s_i} = x_i \oplus s_i = x_i + s_i \bmod 2$
- Decryption: $x_i = D_{s_i} = y_i \oplus s_i = y_i + s_i \bmod 2$

A stream cipher is called:

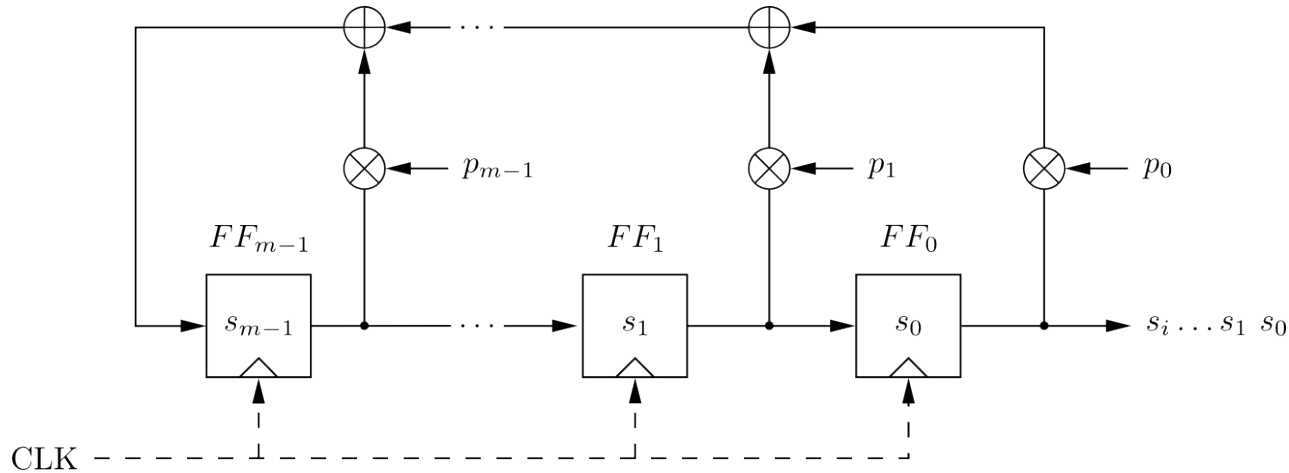
- **synchronous** when s_i is a function of the key
- **asynchronous** when is s_i a function of the key and previous bits of y

The cipher must provide the keystream generator.

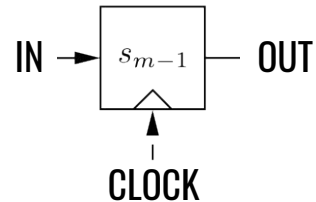
Notice that modulo 2 is equivalent to a bitwise XOR operation.

A5/1 (GSM) and LFSR

A5/1 is the cipher in GSM and is based on three **Linear Feedback Shift Registers (LFSR)**. Each LFSR:



Composed by Flip Flops (which can store 1 bit of information). Each FF: if CLK=1, then FF stores the input IN, emitting the stored value into OUT (even when CLK=0). p_i enable/disable feedback line (switch variable).



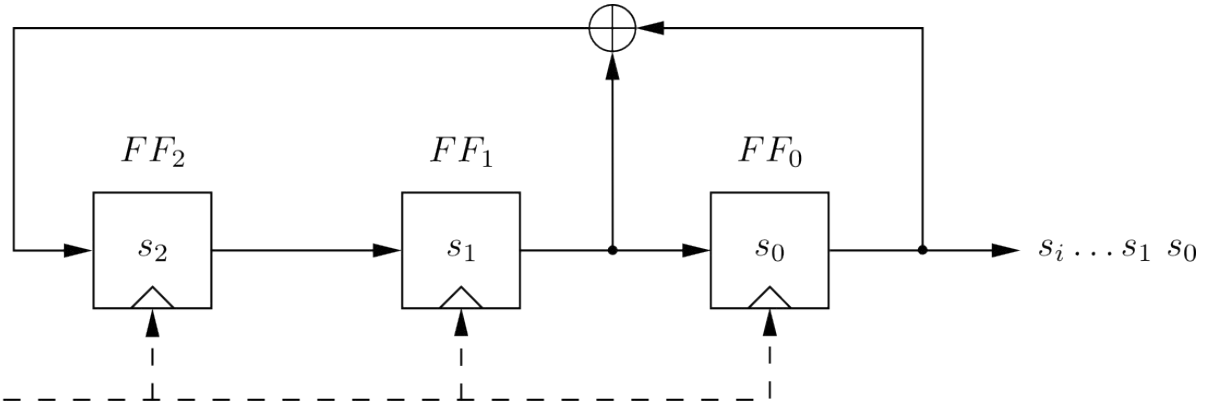
LFSR: $m=3$, $p_0=p_1=1$, $p_2=0$

clk	FF_2	FF_1	$FF_0=s_i$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0

initial state

shifted values

shifted values with effect of feedback



same as line 0 and 1,
it is a cycle and the
output will be: **0010111 0010111 0010111 ...**

LSFR

The previous LFSR can be described as:

Output length: $2^3 - 1$

$$s_{i+3} = s_{i+1} + s_i \bmod 2$$

More in general: $s_{i+m} \equiv \sum_{j=0}^{m-1} p_j \cdot s_{i+j} \bmod 2 \quad p_j \in \{0, 1\}, s_i \in \{0, 1\}$

Since to define a LSFR we just need to know the degree m and p_j then they are often described as a polynomial:

$$P(x) = x^m + p_{l-1}x^{m-1} + \dots + p_1x + p_0$$

The maximum period (or sequence length) by a LFSR of degree m is $(2^m - 1)$

Not all possible configuration for p_j give the maximum period. $P(x)$ s that give maximum period are called primitive and are irreducible. Choosing the value of p_j is part of the design of an algorithm based on LSFR. A5/1 has it own fixed values.

Single LFSR a stream cipher: attack

We assume that the adversary knows: (a) ciphertext y , (b) degree m , (c) first $2m-1$ bits of the plaintext. Steps:

- recover the first $2m-1$ bits of the keystream: $s_i = y_i + x_i \bmod 2$
- recover the other bits of the keystream:

$$\begin{array}{llll} i = 0, & s_m & \equiv & p_{m-1}s_{m-1} + \dots + p_1s_1 + p_0s_0 \quad \bmod 2 \\ i = 1, & s_{m+1} & \equiv & p_{m-1}s_m + \dots + p_1s_2 + p_0s_1 \quad \bmod 2 \\ \vdots & \vdots & \vdots & \vdots \\ i = m-1, & s_{2m-1} & \equiv & p_{m-1}s_{2m-2} + \dots + p_1s_m + p_0s_{m-1} \quad \bmod 2 \end{array}$$

These are m equations: we can solve this system (e.g., w/ gaussian elimination or matrix inversion) and recover p_j . He can now then generate other bits of the keystream.

A5/1 used a combinations of 3 LSFRs, hence it is much stronger. However, there still attacks for it.

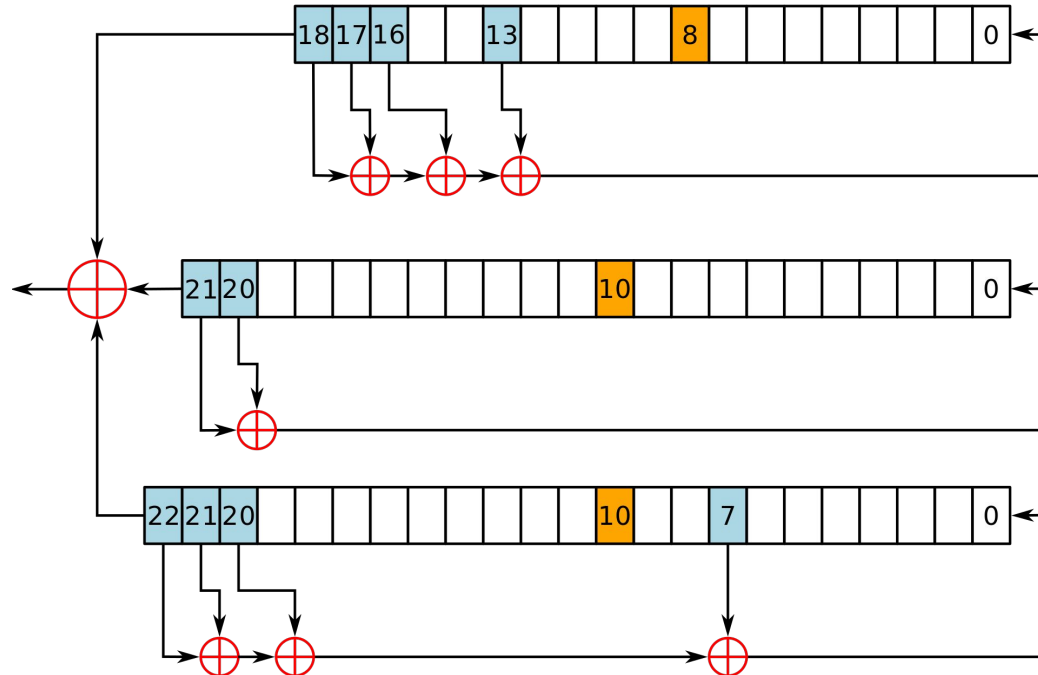
A5/1

LSFRs can be good PRNG but they are not (always) good for encryption.

Clocking bit is used to determine if CLK is enabled based on a majority rule.

Initial state of the registers is the key in A5/1.

LFSR number	Length in bits	Feedback polynomial	Clocking bit	Tapped bits
1	19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	8	13, 16, 17, 18
2	22	$x^{22} + x^{21} + 1$	10	20, 21
3	23	$x^{23} + x^{22} + x^{21} + x^8 + 1$	10	7, 20, 21, 22



RC-4: Ron's Code (or Rivest Cipher 4)

- Designed in 1987 by Ron Rivest, trade secret until 1994 when its description was leaked
- Synchronous, variable key length, very fast to compute
- Starting from the key it will apparently generate a random stream, eventually it will repeat the sequence. However, the period is very long ($> 10^{100}$).
- Several attacks along the years. From 2015, there is speculation that some state cryptologic agencies may possess the capability to break RC4 when used in the TLS protocol. IETF has published RFC 7465 to prohibit the use of RC4 in TLS; Mozilla and Microsoft have issued similar recommendations.

RC-4: Key Scheduling Algorithm (KSA)

S is an array of 256 integers.

```
int j = 0;
for(int i = 0; i < 256; i++)
    S[i] = i;
for(int i = 0; i < 256; i++) {
    j = (j + S[i] + key[i % len]) % 256;
    swap(&S[i], &S[j]);
}
```

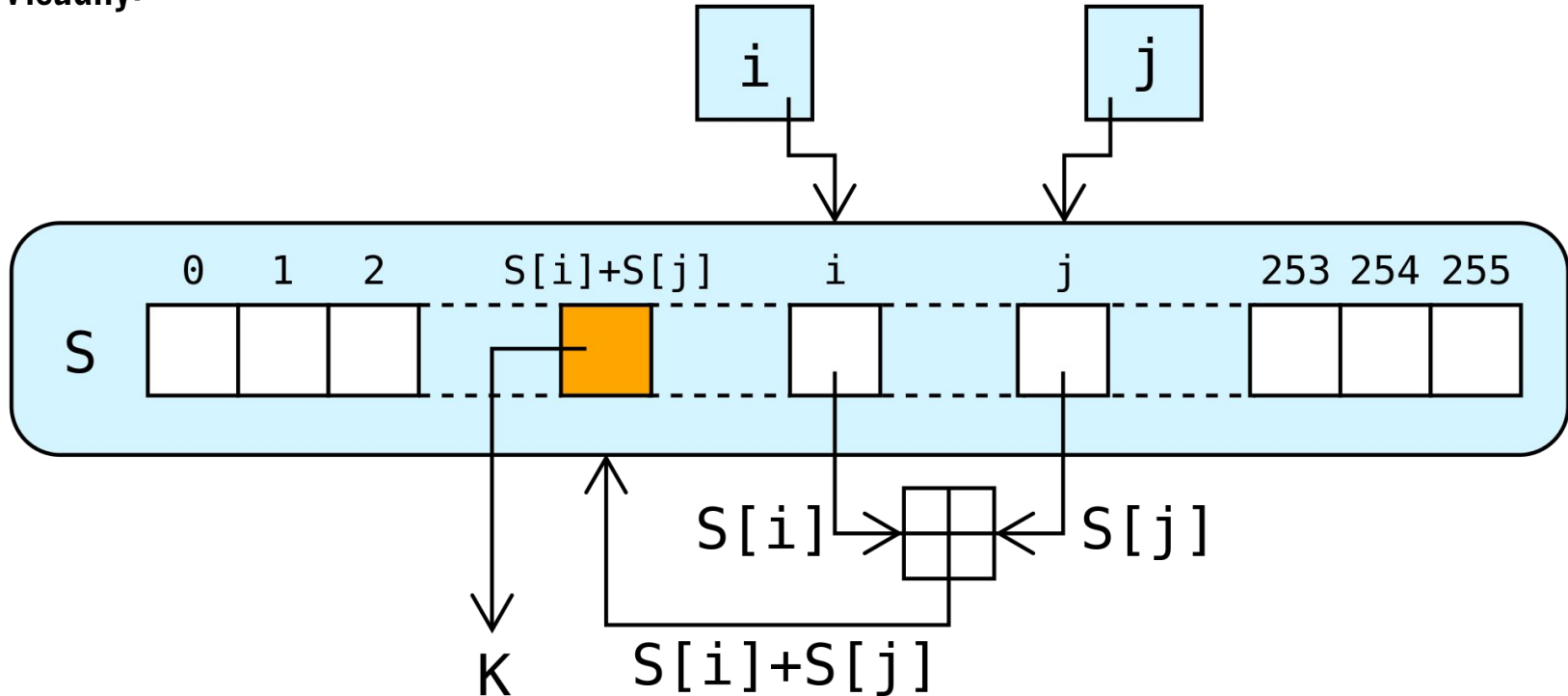
At the end S is a permutation of {0, ..., 255} generated based on the value of the key.

RC-4: Pseudo-Random Generation Algorithm (PRGA)

```
int i = 0, j = 0;
for(size_t n = 0, len = strlen(plaintext); n < len; n++) {
    i = (i + 1) % 256;
    j = (j + S[i]) % 256;
    swap(&S[i], &S[j]);
    int K = S[(S[i] + S[j]) % 256];    // keystream byte
    ciphertext[n] = K ^ plaintext[n]; // XOR encryption
}
```

RC-4: Pseudo-Random Generation Algorithm (PRGA)

Visually:



RC-4: how to encrypt different messages?

We cannot use the same key for two messages, otherwise the keystream could be recovered as in OTP. Hence, we need a way of “randomizing” each keystream:

Idea: combine the key with an initialization vector (IV) that acts as a randomizer

Problem: several algorithms are weak to attacks due to an unsecure combination of IV and the key

WEP (Wired Equivalent Privacy):

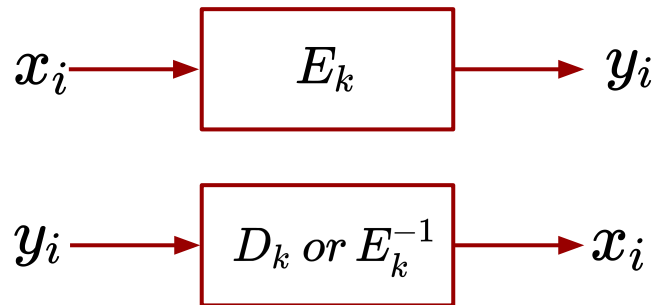
- used for WiFi
- IV is 24 bits, key is 40 bits
- **problem:** there is a 50% probability the same IV will repeat after 5,000 packets.
Also, the attacker is often able to “generate” traffic.

Block Ciphers

Given:

- a block x_i of the message x of h bits (h is fixed)
- a key k of n bits (n fixed)
- n can be different from h

Then:



Real-world ciphers: **DES/2-DES/3-DES**, IDEA, RC-2, RC-5, Blowfish, **AES**

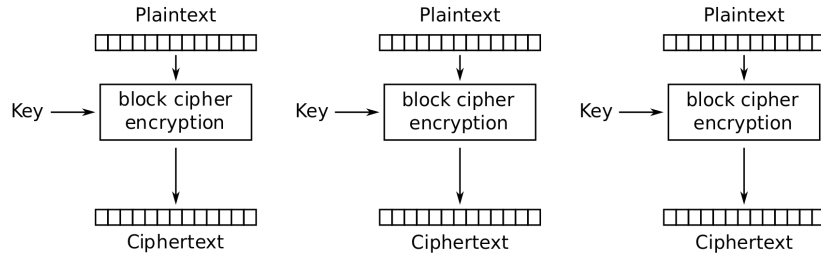
What if the message is larger than the block size?

Operation modes:

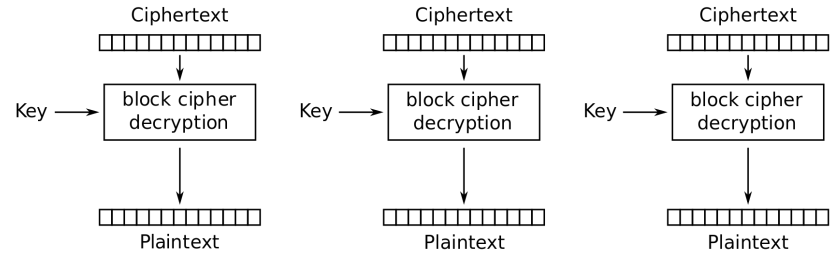
- Electronic Code Block (ECB)
- Cipher Block Chaining (CBC)
- Propagating Cipher Block Chaining (PCBC)
- Output Feedback Mode (OFB)
- Cipher Feedback Mode (CFB)
- Counter Mode (CTR)

Electronic Code Block (ECB)

Encryption:



Decryption:

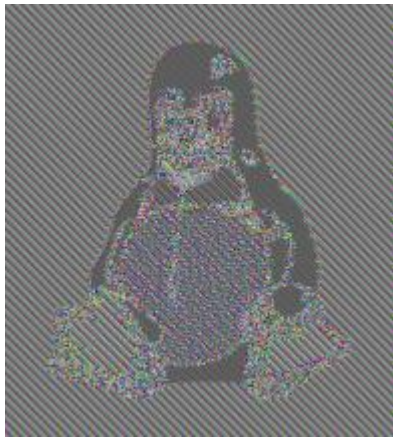


- Simple and efficient
- Parallel implementation

Electronic Code Block (ECB): plaintext patterns



plaintext



ciphertext



what we would like

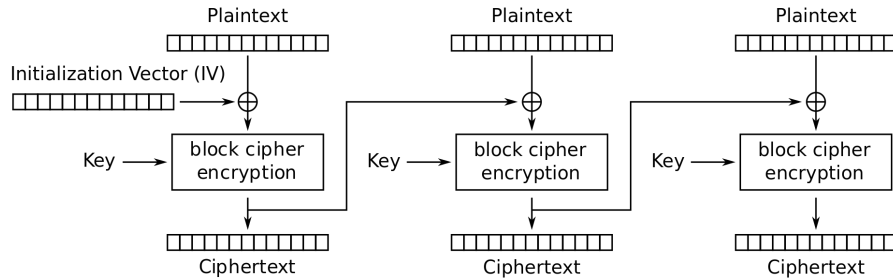
Does not conceal plaintext patterns

Electronic Code Block (ECB): attacks

- **Blocks reordering:** this can be a problem, e.g., in a contract.
- **Blocks can be replaced, removed, appended:** if the message is “Transfer \$10 to Oscar”, it could be manipulated to “Transfer \$10000 to Oscar”.
- Encrypting the same message twice will generate the same ciphertext

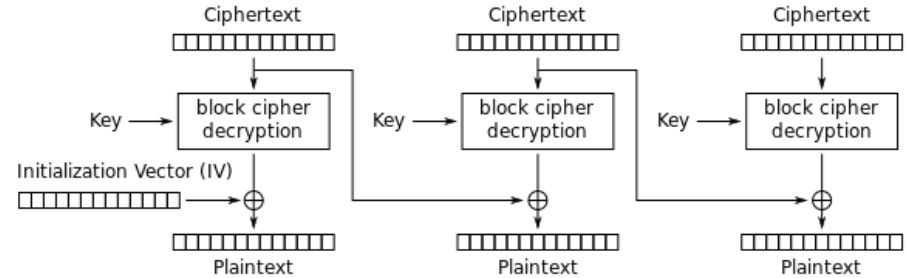
Cipher Block Chaining (CBC)

Encryption:



- block y_i depends on y_{i-1}
- encryption is randomized using an IV
- encryption: no parallelization
- decryption: parallelizable

Decryption:



- if one bit flipped in x_i then all subsequent blocks are affected
- if one bit is flipped in y_{i-1} then x_i is affected in an unpredictable manner, while x_i in a predictable manner. This could be exploited by an attacker. Hence use CRC/etc.

Cipher Block Chaining (CBC): IV

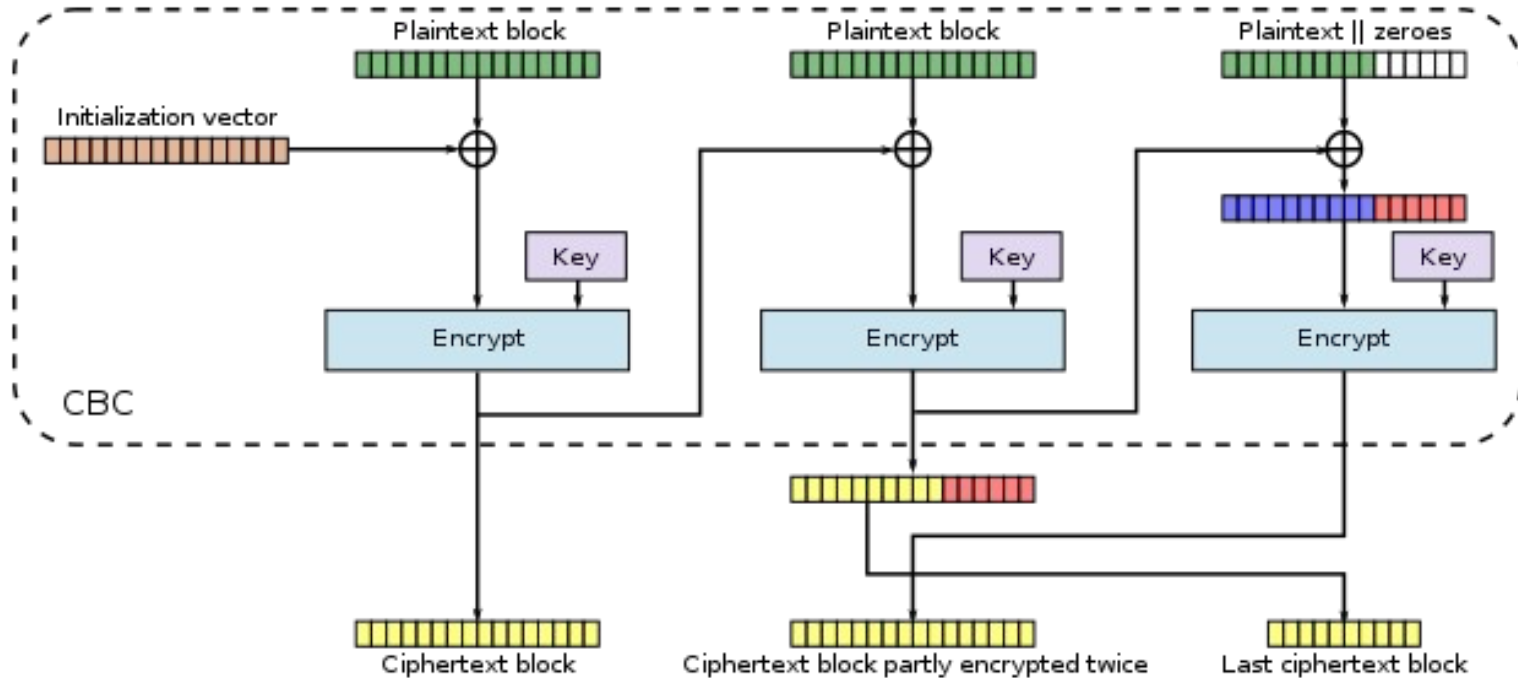
Initialization Vector:

- Should be different for each message, otherwise an attacker can understand when we are encrypting again the same message.
- Can be made public.

Cipher Block Chaining (CBC)

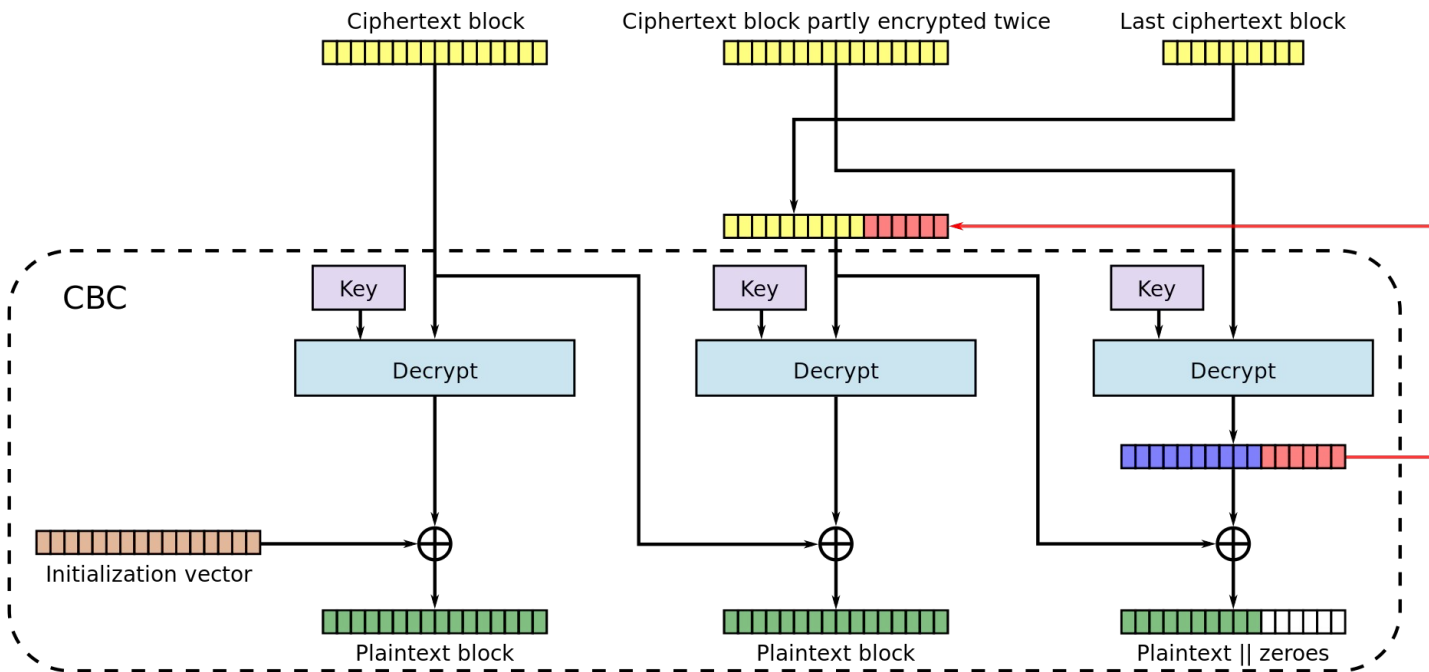
- CBC can be seen as an asynchronous stream cipher
- Message must be padded to a size multiple of the block size (a block cipher can deal only with a fixed block size). This was true even for ECB. Two strategies:
 - padding, which however increases the size (an example is given later)
 - **ciphertext stealing**.

Ciphertext Stealing: encryption



Swap the last cipher blocks, then truncate the ciphertext to the original length of the plaintext

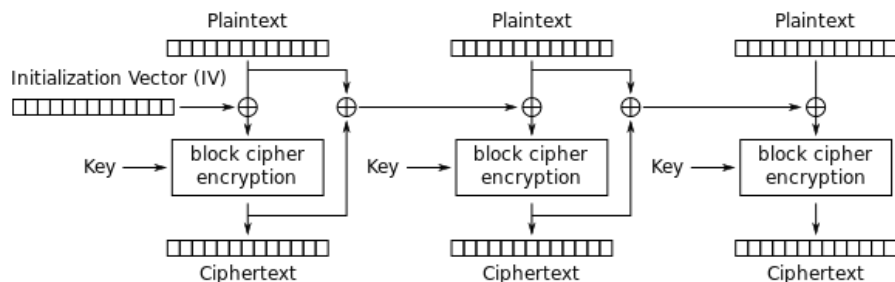
Ciphertext Stealing: decryption



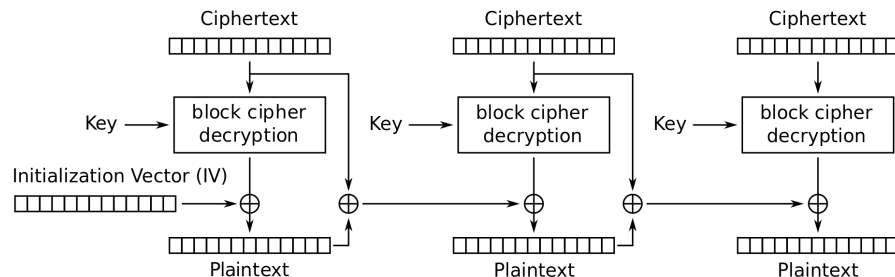
Swap the last cipher blocks, decrypt, then truncate the plaintext to the original length of the ciphertext

Propagating Cipher Block Chaining (PCBC)

Encryption:



Decryption:

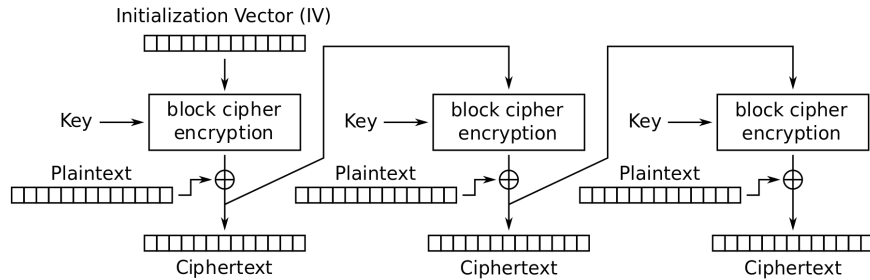


- Designed to propagate small changes to all subsequent blocks both during encryption and decryption

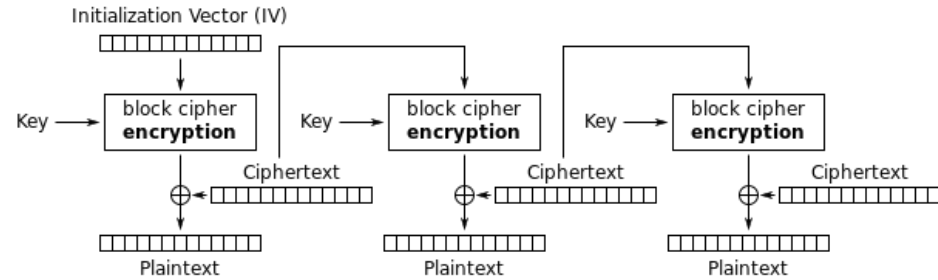
- However, if two adjacent ciphertext blocks are exchanged, subsequent decrypted blocks are not affected

Cipher FeedBack (CFB)

Encryption:



Decryption:

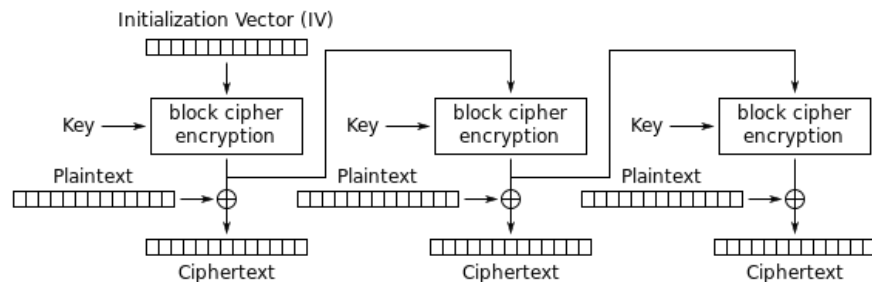


- asynchronous stream cipher
- error propagation in encryption
- encryption is not parallelizable
- encryption algorithm is both used in encryption and decryption

- decryption can be parallelized
- one bit error in ciphertext blocks, affect two plaintext blocks, other blocks are fine
- no need of padding

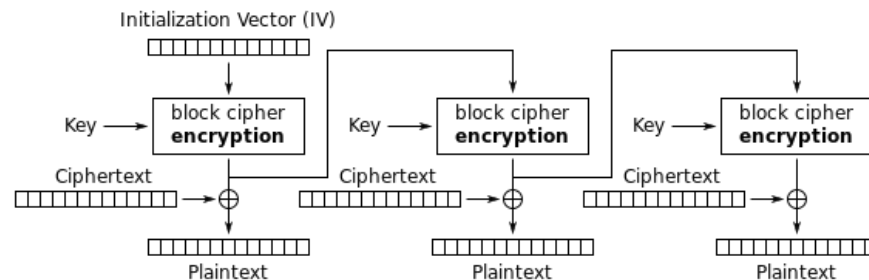
Output FeedBack (OFB)

Encryption:



- synchronous stream cipher
- one bit flip in ciphertext affect only one bit in the output (this helps using error correction codes)

Decryption:



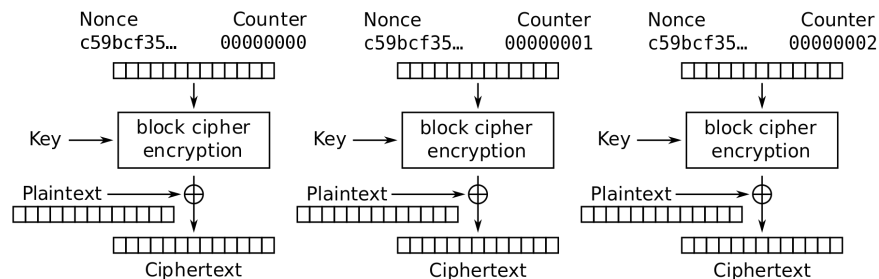
- encryption is not parallelizable
- decryption is not parallelizable
- encryption algorithm is both used in encryption and decryption

Output FeedBack (OFB): problems

- if encryption function and the key are known the adversary, then IV must be kept secret
- if the key is not known by the attacker, then IV can be sent in clear
- IV must be different for each message

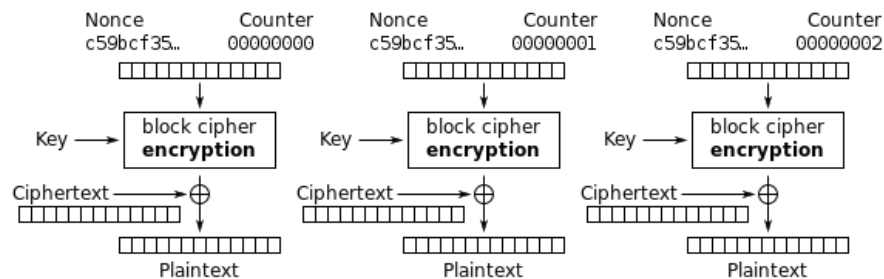
Counter Mode (CTR)

Encryption:



- synchronous stream cipher
- Besides IV (nonce), it uses a counter that is incremented for each block

Decryption:



- encryption is parallelizable
- decryption is parallelizable
- encryption algorithm is both used in encryption and decryption

Operation modes: summary

	ECB	CBC	CFB	OFB	CTR
	Electronic Code Book	Cipher Block Chaining	Output Feedback	Cipher Feedback	Counter
Information leakage	High	low	low	low	low
Encryption parallelizable	Yes	No	No	No	Yes
Decryption parallelizable	Yes	Yes	Yes	No	Yes
Ciphertext manipulation	Yes	No	No	No	No
Precompute	No	No	No	Yes	Yes
Encryption error propagation	No	Yes	Yes	No	No
Decryption error propagation	No	Partial (2 Blocks)	Partial (2 Blocks)	No	No

Initialization Vector and Operation Modes

Most modes (except ECB) require an initialization vector (IV):

- sort of "dummy block" to kick off the process for the first real block, and also to provide some randomization for the process.
- no need for the IV to be secret, in most cases, but it is important that it is never reused with the same key.
- for CBC and CFB, reusing an IV leaks some information about the first block of plaintext, and about any common prefix shared by the two messages.
- In CBC mode, the IV must, in addition, be unpredictable at encryption time (see TLS CBC IV attack)
- for OFB and CTR, reusing an IV completely destroys security.

CBC with PKCS#7 padding scheme

PKCS#7 padding scheme: the value of each added byte is the number of bytes that are added

E.g., block size is 8 bytes

if the message has 8 bytes, we add 8 bytes 0x8 at its end.

if the message has 7 bytes, we add 1 byte 0x1 at its end.

if the message has 6 bytes, we add 2 bytes 0x2 at its end.

...

if the message has 1 byte, we add 7 bytes 0x7 at its end.

Weaken a seemingly strong scheme by small modifications....

Is this approach still secure?

What if we assume than an application use this approach and when performing decryption of a ciphertext provided by a user, returns a “return code”:

- **KO**: the padding scheme is not respected by the ciphertext (after decryption)
- **OK**: the padding scheme is respected by the ciphertext (after decryption)

Notice that the application does not reveal the plaintext, it only provides feedback on whether after decryption the message is valid based on the padding scheme.

Can an attacker exploit this “feedback” from the application?

Padding oracle attack

The scenario where an application provides feedback about the validity of the padding for a message is often defined as the **padding oracle**.

Given a padding oracle, we can sometimes perform an attack able to reveal the plaintext or the secret key, even when the cipher is secure and the operation mode is secure.

We now consider an attack against CBC with PKCS#7 padding scheme, however there are many variants of this attack even for other modes/ciphers/padding (e.g., CBC/PKCS#5). This is an example of **Chosen Ciphertext Attack (CCA)**.

Padding oracle attack against CBC + PKCS#7

In CBC the encryption of a block x_i is performed as:

$$y_i = E_k(x_i \oplus y_{i-1})$$

while the decryption:

$$x_i = D_k(y_i) \oplus y_{i-1}$$

$$x_i = D_k(E_k(x_i \oplus y_{i-1})) \oplus y_{i-1}$$

Assuming the cipher is correct:

$$x_i = x_i \oplus y_{i-1} \oplus y_{i-1}$$

CBC and XOR

Changing a bit (or a byte) of y_{i-1} affects unpredictably the decryption for the block x_{i-1} but affects predictably the decryption of the block of x_i . In the padding oracle attack we exploit this property, designing a new ciphertext with two blocks:

- we do not care for decryption of the first ciphertext block as we cannot learn anything from it even if the padding oracle gives us some feedback
- we care for the decryption of the second ciphertext block as we know how to “bitwise” or “blockwise” manipulate it due to XOR and the feedback from the padding oracle can be valuable.

Padding oracle attack against CBC + PKCS#7 (2)

Assume that the attacker intercepts a ciphertext y and there is padding oracle. Let us define $y_i[j]$ as the j -th byte of y_i .

Then to decrypt block y_i with $i > 0$, he builds a new ciphertext y' with two blocks:

$$y' = y'_0 || y_i$$

He now chooses a random value for y'_0 and calls 256 times the padding oracle, setting each time the last byte y'_0 to a different value in $[0, 255]$. Internally, the oracle will compute:

$$x'_i = D_k(y_i) \oplus y'_0 \qquad x'_i[j] = D_k(y_i)[j] \oplus y'_0[j]$$

and check whether the padding scheme is respected. Notice that it will decrypt both blocks but we do not care for the first one.

Padding oracle attack against CBC + PKCS#7 (3)

Since we are trying all the possible values for the last byte of y_0 and due to how XOR works, we can expect that there is at least one value where the oracle will give OK since we will have:

$$x'_i[j] = 1$$

with **j equal to the last byte of block**. Hence, we get a decrypted block that is valid for the padding scheme (the last byte is one, hence meaning that there is one byte padding and no other bytes must be checked except the last one).

Remark. The oracle may give OK even when: $x'_i[j-1] = 2$ $x'_i[j] = 2$
or when: $x'_i[j-2] = 3$ $x'_i[j-1] = 3$ $x'_i[j] = 3$
or with other similar patterns. We consider these cases later.

Padding oracle attack against CBC + PKCS#7 (4)

Now, the attacker can compute $x_i[j]$:

$$x'_i[j] = D_k(y_i)[j] \oplus y'_0[j]$$

$$x'_i[j] = (x_i \oplus y_{i-1})[j] \oplus y'_0[j]$$

$$x'_i[j] = x_i[j] \oplus y_{i-1}[j] \oplus y'_0[j]$$

$$x_i[j] = x'_i[j] \oplus y_{i-1}[j] \oplus y'_0[j]$$

We have that $x'_i[j] = 1$ (after brute force), $y_{i-1}[j]$ is known (a block from the ciphertext!), and $y'_0[j]$ is known (decided by the attacker!). Hence he can compute $x_i[j]$.

At this point the attacker knows the last byte of the plaintext block x_i and he can iterate this process to get other bytes in the same block.

Padding oracle attack against CBC + PKCS#7 (5)

To get the value of $x_i[j-1]$, the attacker has to build a ciphertext such that the last two bytes of the plaintext are equal to 2, i.e., $x_i'[j] = 2$ and $x_i'[j-1] = 2$.

He builds a new ciphertext y' with two blocks:

$$y' = y'_0 || y_i$$

Hence, he needs to find a y'_0 such that $x_i'[j] = 2$ and $x_i'[j-1] = 2$. Using CBC, we have that:

$$x_i'[j] = x_i[j] \oplus y_{i-1}[j] \oplus y'_0[j]$$

$$2 = x_i[j] \oplus y_{i-1}[j] \oplus y'_0[j]$$

$$y'_0[j] = 2 \oplus x_i[j] \oplus y_{i-1}[j]$$

Hence the values of $y'_0[j]$ can be derived since $x_i[j]$ is known after the previous step. To guess the right value for $y'_0[j-1]$, he tries 256 values.

Padding oracle attack against CBC + PKCS#7 (6)

For one possible assignment of $y_0'[j-1]$, the oracle will confirm that the padding scheme is respected. Now, the attacker can derive the value of $x_i[j-1]$:

$$x'_i[j-1] = D_k(y_i)[j-1] \oplus y'_0[j-1]$$

$$x'_i[j-1] = (x_i \oplus y_{i-1})[j-1] \oplus y'_0[j-1]$$

$$x'_i[j-1] = x_i[j-1] \oplus y_{i-1}[j-1] \oplus y'_0[j-1]$$

$$x_i[j-1] = x'_i[j-1] \oplus y_{i-1}[j-1] \oplus y'_0[j-1]$$

where $x'_i[j-1] = 2$, $y_{i-1}[j-1]$ is known (a block from the ciphertext!), and $y'_0[j-1]$ is known (decided by the attacker!). The process can be repeated to recover other bytes in the same block.

Padding oracle attack against CBC + PKCS#7 (7)

Remark. When trying to discover $x_i[j]$, the oracle could return OK when, e.g.,:

$$x'_i[j - 1] = 2 \quad x'_i[j] = 2$$

This is unlikely if the cipher is robust and we choose a random value for y_0' . However, after computing the last byte (as seen before), we can check whether our assumption on $x'_i[j]$ was correct: it is enough to XOR the last second byte with 0x1 and check if the padding scheme is still valid (if it is not valid than $x'_i[j]$ is different from one and our assumption is wrong, we need to test another value with the padding oracle).

Padding oracle attack against CBC + PKCS#7 (8)

Can we decrypt the first block?

If the IV is sent as the first block of the ciphertext without any authentication (see other lectures in the course) we can still play the attack. However, in practice this is unlikely and we cannot expect to recover the first block as this attack requires to control a ciphertext with at least two blocks.

Padding oracle attack against CBC + PKCS#7 (9)

How to prevent this attack?

The best solution is to not provide a padding oracle, i.e., no feedback after decryption!

Block ciphers: common operations' properties

Shannon suggested two operations for a cipher:


- **Confusion**: each binary digit (bit) of the ciphertext should depend on several parts of the key, obscuring the connections between the two. It hides the relationship between the ciphertext and the key and makes it difficult to find the key from the ciphertext and if a single bit in a key is changed, the calculation of the values of most or all of the bits in the ciphertext will be affected. One way of achieving confusion is substitution (as in AES and DES).
- **Diffusion**: changing a single bit of the plaintext, then (statistically) half of the bits in the ciphertext should change, and similarly, if we change one bit of the ciphertext, then approximately one half of the plaintext bits should change. Since a bit can have only two states, when they are all re-evaluated and changed from one seemingly random position to another, half of the bits will have changed state. The idea is to hide statistical properties of the plaintext. One way of achieving diffusion is bit permutation (done in DES).

Block ciphers: common operations' properties (2)

A concept often connected to diffusion and confusion is the **Avalanche effect**:

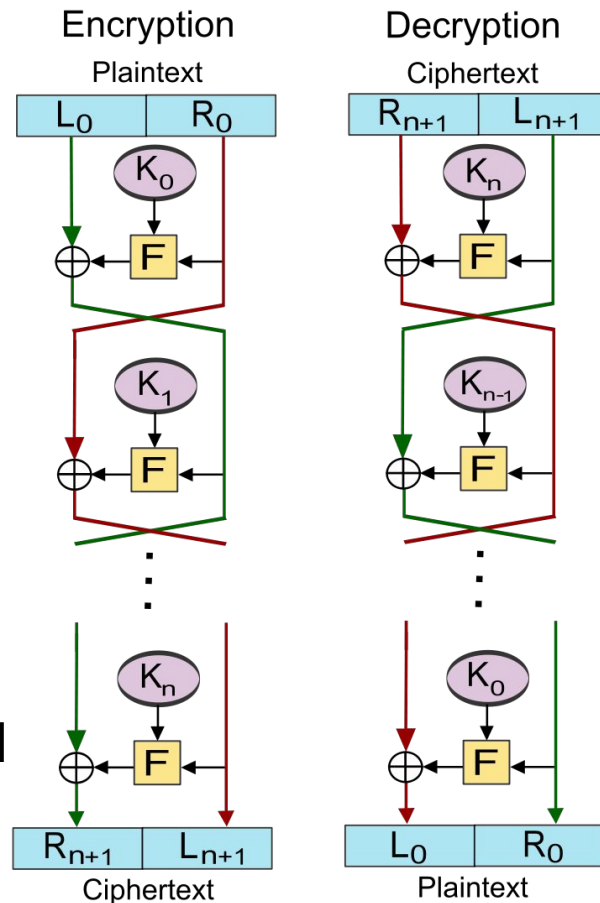
- [plaintext] very small changes to the plaintext lead to a big changes in the ciphertext.
- [key] very small changes to the key lead to a big changes in the ciphertext.

Data Encryption Standard (DES)

- Developed by IBM in 1970s as Lucifer, classified design elements, based on **Feistel design**
- Approved as a federal standard in 1976, and published in 1977 as FIPS PUB 46
- Key length: 56 bits  very weak, can be broken in less than 24 hours using “linear attack”
- Block size 64 bits
- It is believed that NSA can easily break it from the 90s
- The US government NIST (National Inst. of standards and technology) announced a call for an advanced encryption standard (AES) in 1997 to replace DES.

Feistel Network

- This design allows encryption and decryption to be the same/similar. Hence, function F does not have to be invertible.
- It has been proved that if F is pseudorandom function with K_i used as seeds (subkeys derived from the secret key k) then sufficient to make it a "strong" pseudorandom permutation.
- DES is based on this design. We do see not its internal design (F and key derivation), see [slides](#) of chapter 3 of Book "Understanding Cryptography" for more details.



DES can be broken: what we do?

Two approaches:

1. We move to another cipher, e.g., AES:
 - a. **pros:** safe choice, likely to be ok for many years to come
 - b. **cons:** need to invest a lot of money in new software and hardware implementations (and then test them!)
2. We try to “fix” DES with different strategies.
 - a. **pros:** reuse of existing hardware and software implementations
 - b. **cons:** what if our “strategy” is not actually fixing the problem?

Can we make DES stronger by applying it several times?

Iterated ciphers:

- **EEE mode:** $y = E_{k_1}(E_{k_2}(E_{k_3}(x)))$
E.g. 3-DES
- **EDE mode:** $y = E_{k_1}(D_{k_2}(E_{k_1}(x)))$
Very common as it requires only two keys.

If two keys may be enough, what about 2-DES with two keys? What 2-DES with just one key?

2-DES with one key

$$y = E_{k_1}(E_{k_1}(x))$$

The brute-force complexity is not changed: still 2^{56} attempts are enough!
The only difference is that each attempt cost twice as one attempt for DES.

2-DES with two keys

$$y = E_{k_2}(E_{k_1}(x))$$

$$x = D_{k_1}(D_{k_2}(y))$$

Q. Is the attack complexity 2 times 2^{56} since we use two keys of 56 bits each?

A. NO, effective key strength is only 2^{57} and not 2^{112} .

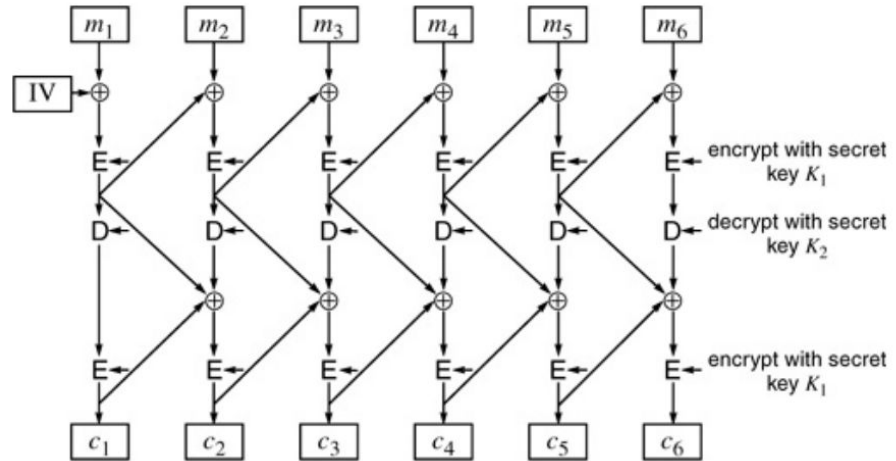
Meet-in-the-Middle Attack. Assuming to have a pair (x, y) :

- 2^{56} attempts: $A = \{ \forall k_1 : E_{k_1}(x) \}$
- 2^{56} attempts: $B = \{ \forall k_2 : D_{k_2}(y) \}$
- Find matching: (a, b) such that $a = b$ where $a \in A, b \in B$
- first key is the one that have generated a, second one is the one that generated b
- Total attempts: $2^{56} + 2^{56} = 2^{57}$

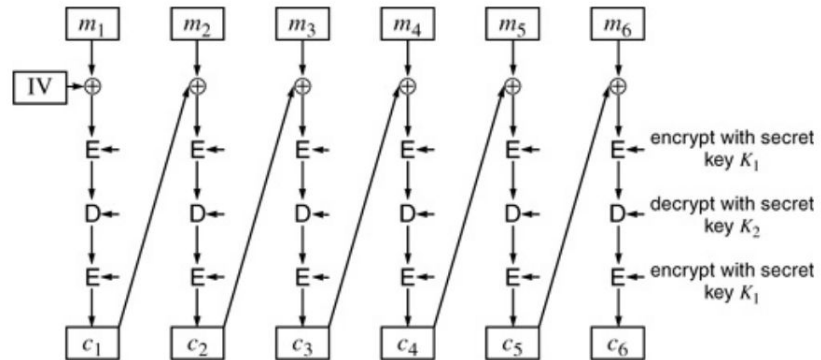
Remark: Using MITM, we can see that 3-DES thus has 2^{112} key strength

EDE:

- inside CBC



- outside CBC



CBC outside vs CBC inside

1. Bit Flipping

- CBC Outside: one bit flip in the ciphertext causes that block of plain text and next block garbled \Rightarrow Self-Synchronizing (i.e., after some garbled blocks, you get correct blocks)
- CBC Inside: one bit flip in the ciphertext causes more blocks to be garbled

2. Pipelining: more pipelining possible in CBC inside implementation

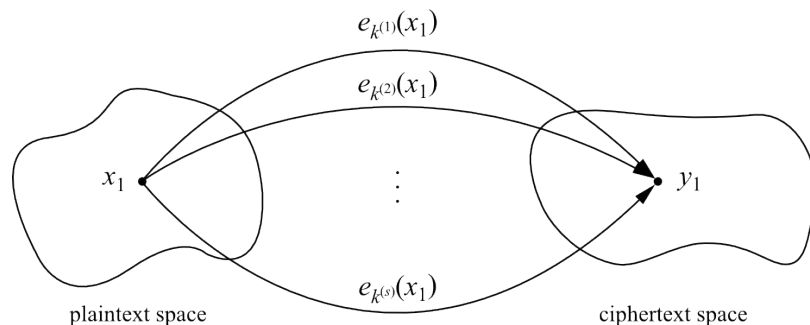
3. Flexibility of Change: in “CBC outside” we can easily replace CBC with other operation modes (ECB, CFB, ...)

Another idea to make a cipher stronger: key whitening

- Common form: xor-encrypt-xor (using a simple XOR with the key before the first round and after the last round of encryption)
- DES-X: $\text{DES-X}(x) = k_2 \oplus (\text{DES}_k(x \oplus k_1))$
 - Three keys (k 56 bits, k_1 64 bits, k_2 64 bits): 184 bits
 - However, effective key size is only 119 bits when the attacker can obtain enough (plaintext, ciphertext) pairs [paper]

False positives in brute force

Assume key space is larger than message space. A brute force can produce false positives:



Keys k_i that are found are not the one used for the encryption. The likelihood of this is related to the relative size of the key.

False positives in brute force (2)

Assume a cipher with a block width of 64 bit and a key size of 80 bit:

- If we encrypt x_1 under all possible 2^{80} keys, we obtain 2^{80} ciphertexts
- However, there exist only 2^{64} different ones
- If we run through all keys for a given (plaintext, ciphertext) pair, we find on average:
 $2^{80}/2^{64} = 2^{16}$ keys that perform the mapping $e_k(x_1) = y_1$

Given a block cipher with a key length of k bits and block size of n bits, as well as t (plaintext, ciphertext) pairs $(x_1, y_1), \dots, (x_t, y_t)$, the expected number of false keys which encrypt all plaintexts to the corresponding ciphertexts is:

$$2^{k-tn}$$

- In this example assuming two (plaintext, ciphertext) pairs, the likelihood is $2^{80-2 \cdot 64} = 2^{-48}$
- for almost all practical purposes two (plaintext, ciphertext) pairs are sufficient. As soon you have some plaintext/ciphertext pairs, brute force is very effective.

Credits

These slides are based on material from:

- Slides of Prof. D'Amore from CNS 2019-2020
- Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer. <http://www.crypto-textbook.com/>
- Wikipedia (english version)