50.042 FCS Summer 2024 Lecture 2 – Basic Encryption

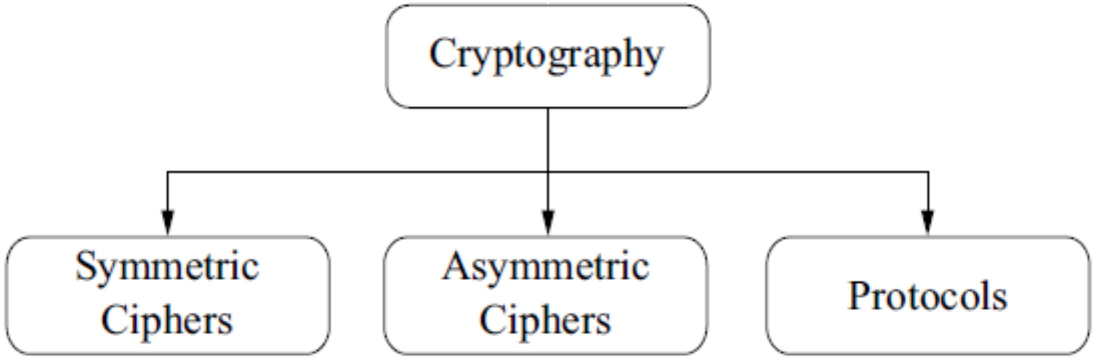
Felix LOH
Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

With selected materials adapted from: *Understanding Cryptography: A Textbook for Students and Practitioners, by C. Paar and J. Pelzl*

Overview of ciphers

- We will discuss the following ciphers/cryptoalgorithms in 50.042:
  - Symmetric ciphers
    - Substitution ciphers
    - Stream ciphers
    - Block ciphers
  - Asymmetric (or Public-key) ciphers
    - Diffie-Hellman key exchange (DHKE)
    - RSA
  - Hash functions



Substitution ciphers: basics

- These are historical ciphers; used until the middle of the last century
- Mono-alphabetic: both the plaintext and ciphertext are based on the alphabet (A-Z)
- Bijection (complete mapping) relationship between both alphabets
- Example: Caesar's cipher (or shift cipher)
  - Shift all characters by k in the alphabet
  - When k = 3:
  - x = 'SECURITY' $\rightarrow$ y = 'VHFXULWB'
  - To decrypt, shift backwards by k

Modulo operation (recap)

- Let $a, r, m \in \mathbb{Z}$ (i.e. a, r and m are members of $\mathbb{Z}$, the set of all integers), with $m > 0$

  We write $a \equiv r \bmod m$, if m divides $a - r$

  r is called the *remainder* and m is called the *modulus*

- Notes:
  - Strictly speaking, the remainder is **not** unique, because mathematically, it's **not** required that $0 \leq r < m$ (unlike Python)
  - So there are infinitely many values of r for any given a and m
  - E.g. $7 \equiv 7 \bmod 4$, $7 \equiv 3 \bmod 4$, $7 \equiv -1 \bmod 4$, and so on
  - However, by convention, we choose r such that $0 \leq r < m$, so in the above example, we pick $7 \equiv 3 \bmod 4$

Mathematical description of Caesar's cipher

- Let $x, y, k \in \mathbb{Z}_{26}$, where $\mathbb{Z}_{26}$ is the set of integers {0, 1, 2, 3, …, 24, 25}

  Then the encryption and decryption functions are as follows:

  encryption function.

  Encryption: $y = e_k(x) \equiv (x + k)\ mod\ 26$

  Decryption: $x = d_k(y) \equiv (y - k)\ mod\ 26$

$k = 3$

eg: $F = 15$

$e_k(F) = (F + 3) \bmod 26 = (15 + 3) \bmod 26 = 18 \bmod 26 = 18 = $ "I"

eg: $I = 18$

$d_k(I) = (18 - 3) \bmod 26 = 15 \bmod 26 = 15$

$k = 4$

eg. $Z = 25$

$e_k(Z) = (Z + 4) \bmod 26 = (29) \bmod 26 = 3 = $ "D"

$d_k(D) = (D - 4) \bmod 26 = (3 - 4) \bmod 26 = -1 \bmod 26 = 25$

Security assessment: Caesar’s cipher

- System model: Alice and Bob share the same key (k), no secure channel
- Attacker model: In possession of the ciphertext, does not have the key, wants to obtain the plaintext
- Requirements: Confidentiality of the plaintext, requires the key to decrypt
- How can the attacker break this cipher? What is the effort required?

Security assessment: Caesar's cipher

- How can the attacker break this cipher? What is the effort required?
- Answer: The attacker can use a *brute force attack*, since the keyspace (the set of all possible keys) for this cipher is only 26 (including the case where there is zero shift, i.e. $k = 0$)
  - $k_1 = 0, k_2 = 1, k_3 = 2, \ldots, k_{26} = 25$ → 26 xn times
  - i.e. keyspace, $K = \{k_1, k_2, k_3, \ldots, k_{26}\} = \{0, 1, 2, \ldots, 25\}$ → standardized cipher so iterate from 0→25
- Try each of the 26 possible keys, the key that results in a legible plaintext is the correct key
- Very little effort required to break this cipher

Improving the substitution cipher

- The keyspace for Caesar's cipher is very small
- We can make the keyspace much larger by introducing random mappings between each character of the input alphabet and each character of the output alphabet
- E.g. A → X, B →E, C → T, etc. for one possible mapping
- Each of these possible mapping schemes is a key
- With this random mapping strategy, the number of possible different mappings (i.e. the size of the keyspace) is:
  - $26! \approx 2^{88}$
  - This is a very large keyspace, so brute force attacks are infeasible here
- But there is a better way to attack the substitution cipher…

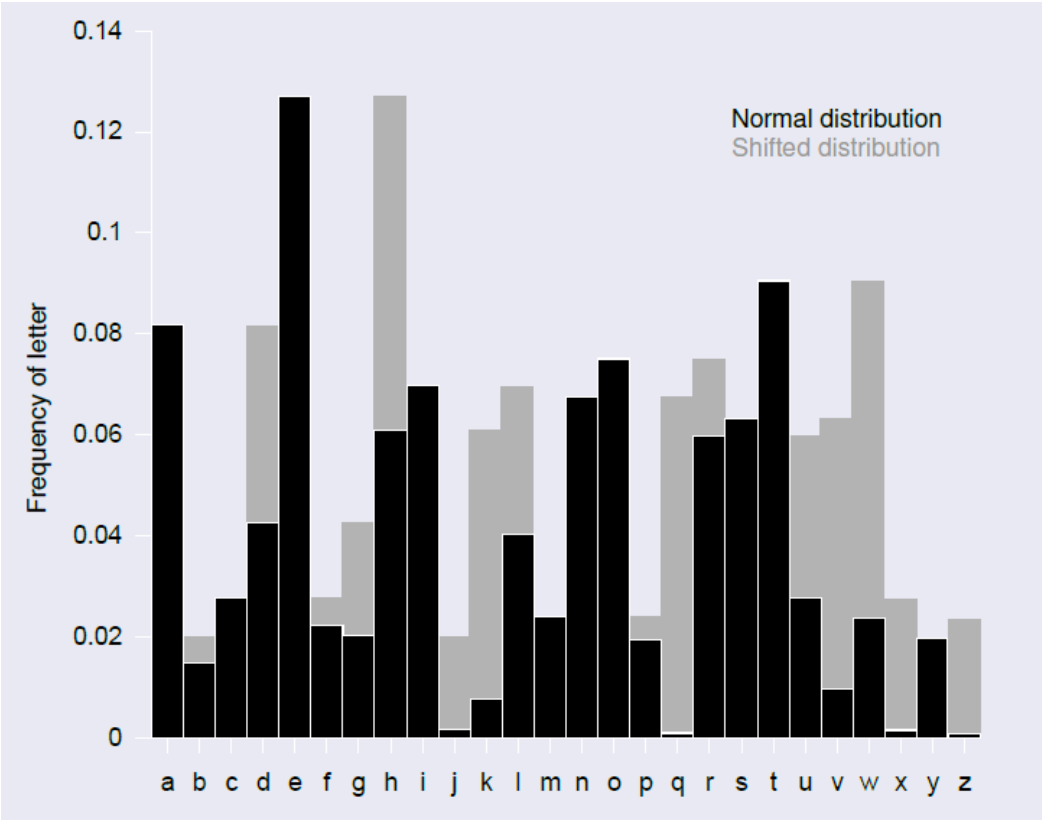
Frequency analysis of the substitution cipher

- Frequency of occurrence of the letters of the English alphabet:

Letter	Frequency	Letter	Frequency
A	0.0817	N	0.0675
B	0.0150	O	0.0751
C	0.0278	P	0.0193
D	0.0425	Q	0.0010
E	0.1270	R	0.0599
F	0.0223	S	0.0633
G	0.0202	T	0.0906
H	0.0609	U	0.0276
I	0.0697	V	0.0098
J	0.0015	W	0.0236
K	0.0077	X	0.0015
L	0.0403	Y	0.0197
M	0.0241	Z	0.0007

Frequency analysis of the substitution cipher

- The language-specific character distribution can be used to deduce the magnitude of the shift (i.e. the key)
- E.g. Given the frequency plot on the right, as well as our knowledge of the distribution of the letters of the English alphabet, we can determine that *k* = 3 for this cipher, and break this cipher



Ways to counter frequency analysis

- Have several alternative replacements for the letter ‘e’, choose amongst these replacements randomly
- Intentionally misspell, or use a dialect
- Insert ‘red herring’ characters to impede frequency analysis
- Treat the word ‘the’ as a new single character and map it to a different character

A note on substitution operations

- Substitution operations are still a useful component of modern ciphers, but they must operate on alphabetical characters with uniform probability

Another substitution cipher: Vigenère cipher

- Published in 1553 by Giovan Battista Bellaso
- Changes the substitution mapping in a periodic pattern
- The key is a word that defines that periodic pattern
- E.g.

	a	b	c	d	e	f	…
A	a	b	c	d	e	f	…
B	b	c	d	e	f	g	…
C	c	d	e	f	g	h	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

plaintext:	d	e	a	d	b	e	e	f
key = "**CAB**":	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**
ciphertext:	f	e	b	f	b	f	g	f

Breaking the Vigenère cipher

- Direct frequency analysis will not work here, because the individual character frequencies are distributed
- However, there is a way to break this cipher, as the key has a fixed length and is repeated often:
  - For various key lengths *n*, compute the distribution for each *n*th character
  - With the right value of *n* (call this value $n_{correct}$), you will observe the characteristic distribution of the letters of the alphabet
  - Then there are basically $n_{correct}$ Caesar's ciphers that are used to encrypt the plaintext (one for each character of the key of length $n_{correct}$)
  - Break each character of the key separately, then break the cipher

ASCII character encoding

- In practice, the data used in modern digital logic devices is not directly represented by some alphabet
- Computer systems operate on *binary* data (recall from your Computation Structures class)
- The ASCII character encoding is typically used to represent text in computers
- E.g. The character ‘N’ is represented by 0x4E in ASCII, while ‘n’ is represented by 0x6E; ‘Hello’ is represented by 0x48656C6C6F
- From this point onwards, we will discuss ciphers in terms of binary data

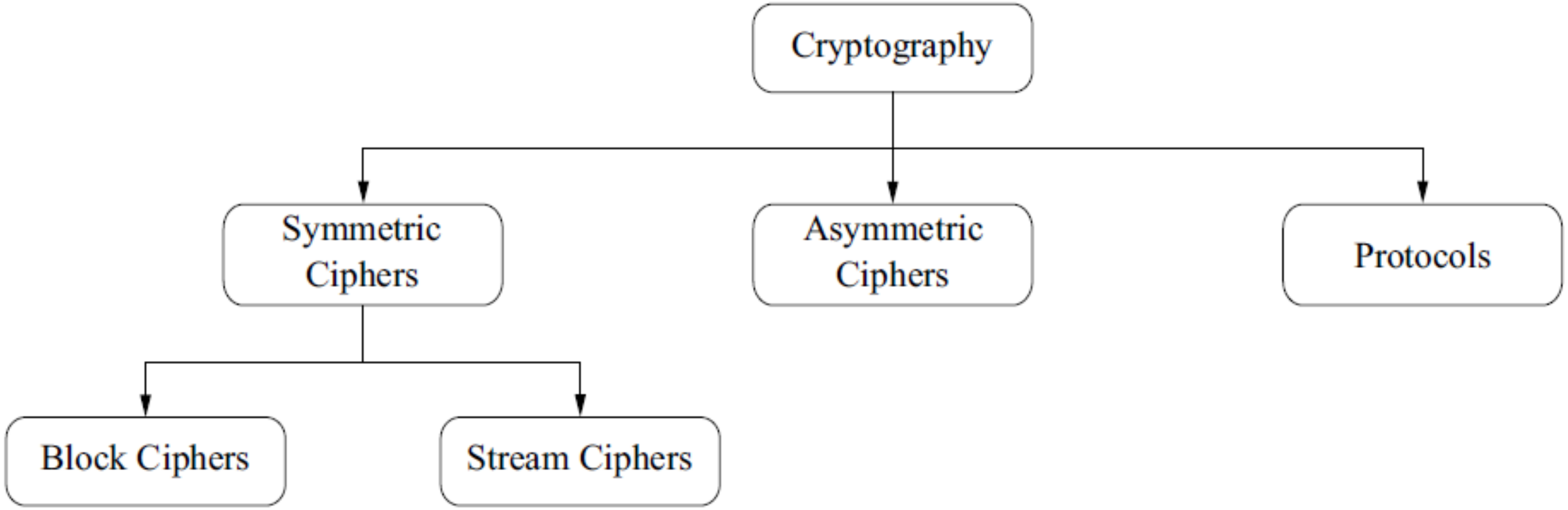
Substitutions on binary data

- How can we apply the substitution principle to binary data?
- Substitute one bit at a time:
  - 2 different keys are possible: 0 → 0 and 1 → 1 (buffer); 0 → 1 and 1 → 0 (inversion)
- Substitute two bits at a time:
  - 4! possible different keys
  - Reasoning: We can pick one out of 4 different 2-bit patterns to map to ‘00’: ‘00’, ‘01’, ’10’ or ‘11’
  - Then for ‘01’, we can pick one out of the three remaining 2-bit patterns, and then one out of the two remaining patterns for ‘10’

Substitutions on binary data

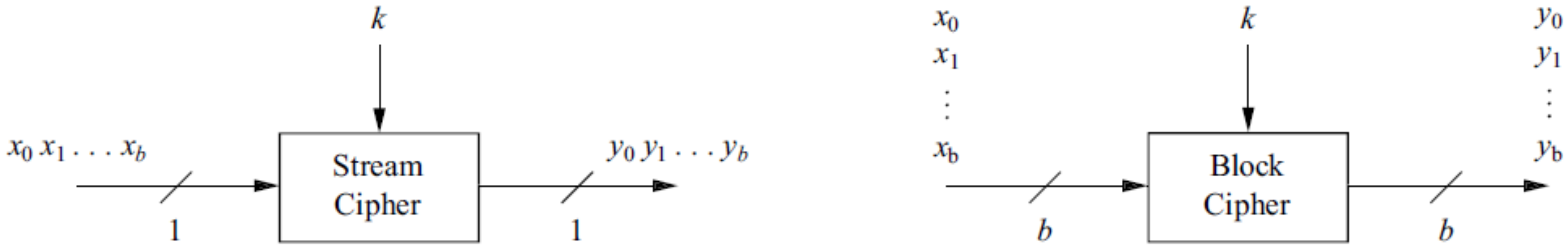
- Substitute *n*-bit blocks at a time:
  - $2^n!$ possible different keys
- Some blocks might occur more frequently than others, depending on the character encoding and *n*
- This would allow for attacks based on frequency analysis
- Ciphers based purely on substitution operations are insufficient

Overview of modern ciphers



- Symmetric ciphers – same secret key for both encryption and decryption
- Asymmetric (or Public-key) ciphers – different keys for encryption and decryption; one key is secret while the other is public

Overview of modern ciphers



1bit at a time

- Stream ciphers –operate on a single plaintext bit at a time
- Block ciphers – operate on a *fixed length* block of plaintext bits at a time (e.g. 128-bit or 256-bit blocks)

Overview of modern ciphers

- The basic operations of modern ciphers involve mostly the XOR operation, substitution and shift operations
  - For performance reasons
- Some ciphers use algebraic operations like multiplication and exponentiation
- These operations always operate within a *finite* set of integers (i.e. modulo operations, typically *mod* 2 or *mod* powers of 2)

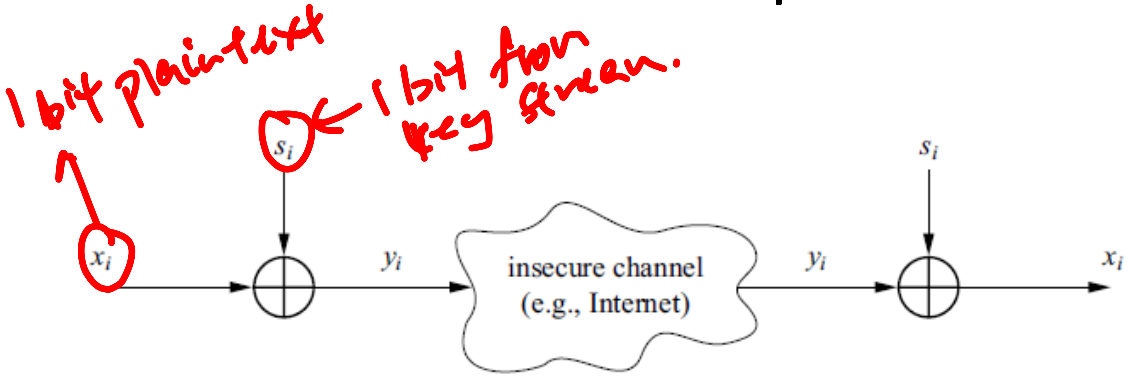
Stream ciphers vs. block ciphers

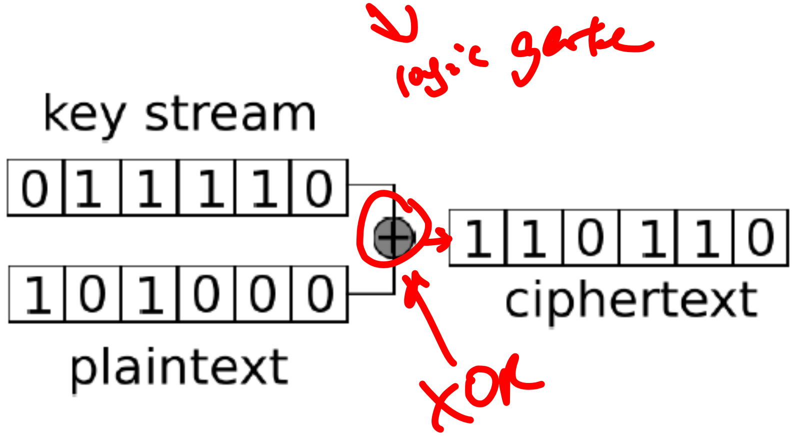
- Stream ciphers:
  - Operate on a single bit at a time
  - Suitable for encrypting audio signals
  - Pro: lower latency for inputs with low data rates
  - Con: low throughput for inputs with high data rates

- Block ciphers:
  - Operate on a *fixed length* block of bits at a time
  - Suitable for encrypting packet-based communication (like internet traffic)
  - Pro: high throughput, parallelization is possible
  - Con: the data needs to be padded (to fit blocks), not as efficient

Stream cipher encryption and decryption

- To encrypt the plaintext bits x_i of the plaintext message x, we just need to do the following:
- Do addition modulo 2 of a secret (random) key stream bit s_i to x_i, to obtain a corresponding ciphertext bit y_i
- Decryption uses the same operation: addition modulo 2 of the key stream bit s_i to y_i, obtain the corresponding plaintext bit x_i
- Addition modulo 2 is equivalent to the bitwise XOR operation





Mathematical description of stream ciphers

- Let the plaintext message x, the ciphertext message y, and the key stream s be composed of individual bits x_i, y_i, $s_i \in \{0, 1\}$ respectively

  Then the encryption and decryption functions are as follows:

  Encryption: $y_i = e_{s_i}(x_i) \equiv (x_i + s_i) \bmod 2$ (i.e. $y_i = x_i \oplus s_i$)

  Decryption: $x_i = d_{s_i}(y_i) \equiv (y_i + s_i) \bmod 2$ (i.e. $x_i = y_i \oplus s_i$)

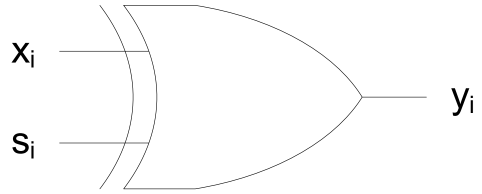
- *Notes:*
  - *The encryption and decryption functions are the same operation (XOR, i.e. addition modulo 2)*
  - *The key stream s is **not** the key k; rather, s is generated from k*

Why the XOR operation is a good encryption function

not same length as plaintext
it is generated dynamically using same function (pseudo random generator)

acts like a seed value.

- Assume the key stream bit s_i has been chosen in a perfectly random manner; i.e. s_i has a 50% chance to be the value 0 and 50% chance to be the value 1
- Then from the XOR truth table, if the plaintext bit $x_i = 0$, the resulting ciphertext bit y_i has equal probability of being the value 0 or 1 (so y_i is unpredictable); likewise for the case where $x_i = 1$
- The XOR operation is a perfectly balanced function



$y_i = x_i \oplus s_i \equiv (x_i + s_i) \bmod 2$

x_i	s_i	y_i
0	0	0
0	1	1
1	0	1
1	1	0

XOR operations in modern cryptography

- The XOR operation plays a major role in modern cryptography
- We will see the XOR operation being utilized in other modern ciphers

Unconditional security vs. computational security

- Definition of unconditional security:
  - *A cryptosystem is unconditionally or information-theoretically secure if it cannot be broken even with infinite computational resources.*
- Definition of computational security:
  - *A cryptosystem is computationally secure if the best known algorithm for breaking it requires at least t operations.*
  - *t* is some arbitrarily large number

- As a matter of fact, all known *practical* cryptoalgorithms (stream ciphers, block ciphers, public-key algorithms, etc.) are **not** unconditionally secure
- The best we can hope is for these to be computationally secure

The One-Time Pad – an "ideal" stream cipher

- A stream cipher for which
    1. the key stream $s_0, s_1, s_2, \ldots$ is generated by a true random number generator, and
    2. the key stream is only known to the legitimate communicating parties, and
    3. every key stream bit s_i is <u>only used once</u>

  is called a one-time pad

The One-Time Pad – an “ideal” stream cipher

- The one-time pad (OTP) is unconditionally secure, brute force attacks will not work against it
- Reason: each ciphertext bit y_i represents a linear equation modulo 2 with two unknowns: $y_i \equiv (x_i + s_i) \bmod 2$ → this cannot be solved, assuming that each key stream bit s_i has been generated randomly and is not reused for other ciphertext bits (i.e. reused for any other linear equation)
- Main takeaway: The security of any stream cipher completely depends on the key stream

Reuse of the key stream

- It is not a good idea to use the key stream again, once it has been used to encrypt a message
- Reason: Suppose we have two plaintext messages x_1 and x_2, with key stream s to be reused. Then we have ciphertext messages $y_1 = e_s(x_1)$ and $y_2 = e_s(x_2)$
- Since $e_s(x) = x \oplus s$,

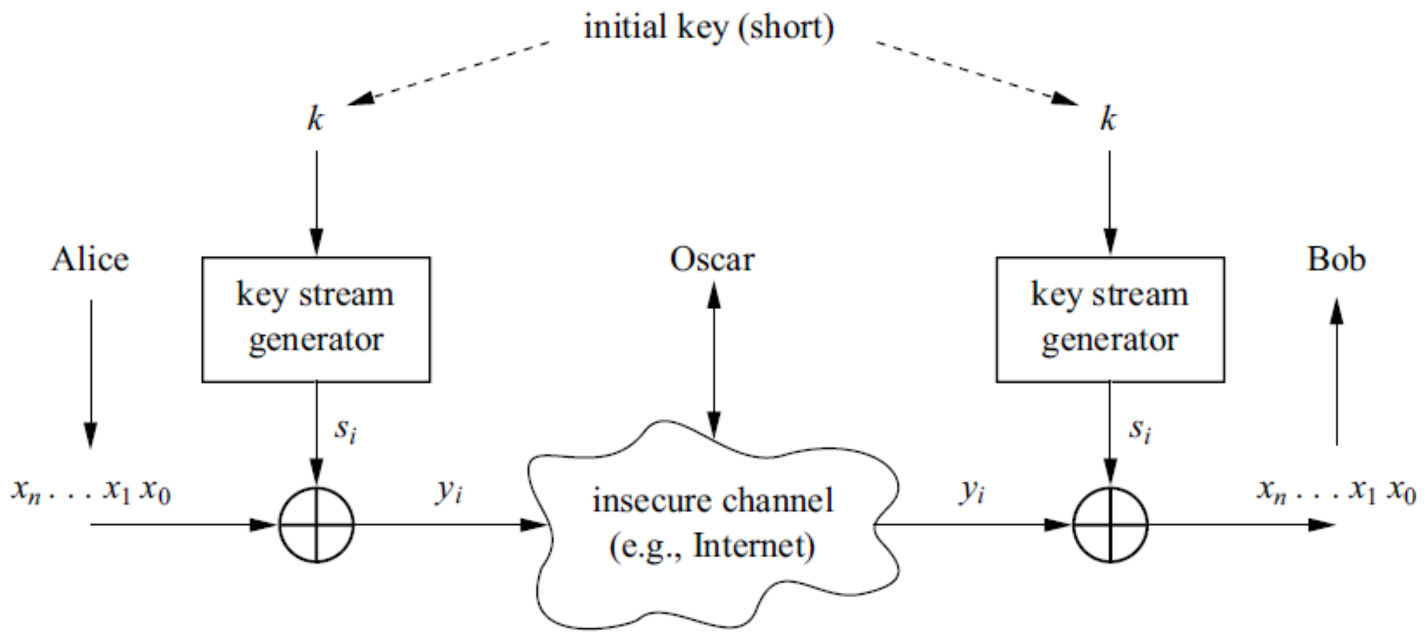
  $y_1 \oplus y_2 = e_s(x_1) \oplus e_s(x_2) = (x_1 \oplus s) \oplus (x_2 \oplus s) = x_1 \oplus x_2$
- So, if the alphabet used for the messages x_1 and x_2 have some frequency distribution, then frequency analysis attacks are possible
- Also, if either x_1 or x_2 is known to Oscar, then the other message can be derived easily

Problems with the One-Time Pad

- The one-time pad (OTP) is unconditionally secure, but it is not a practical cipher, for the following reasons:

1. It needs a true random number generator
2. The key stream s has to be transmitted over a secure channel
3. Since the *key stream bits* cannot be reused, we require one *key bit* for every bit of the plaintext; this means that the key is as long as the plaintext

Practical stream ciphers

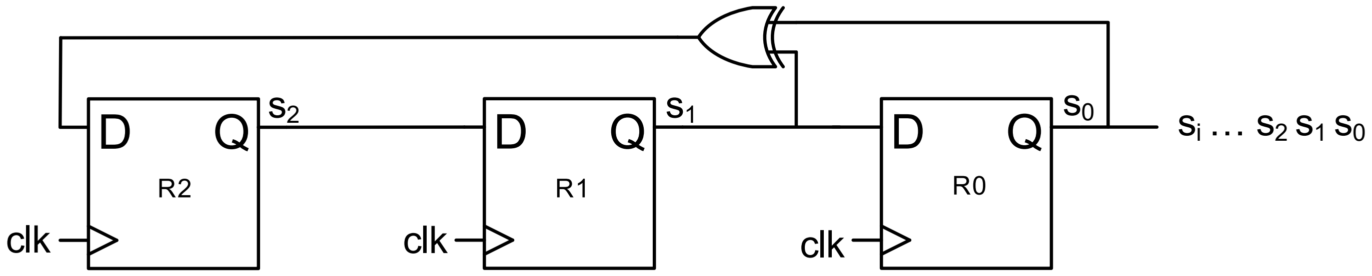


- Using a true random number generator to produce the key stream is difficult and impractical
- In practice, a pseudorandom number generator is used for the key stream, with key *k* used as the seed (key *k* is short compared to the key stream *s*)

Linear feedback shift register

- We can use a linear feedback shift register (LFSR) as a pseudorandom number generator
- E.g. the LFSR below with initial state or key of (s_2 = 1, s_1 = 0, s_0 = 0) will produce the output s_i = Q_{R0} as shown in the table on the right

clk	Q_{R2}	Q_{R1}	$Q_{R0} = s_i$
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



LFSR with starting values = $s_2\,s_1\,s_0$