CISC 468: CRYPTOGRAPHY

LESSON 3: STREAM CIPHERS

Furkan Alaca

TODAY, WE WILL LEARN ABOUT...

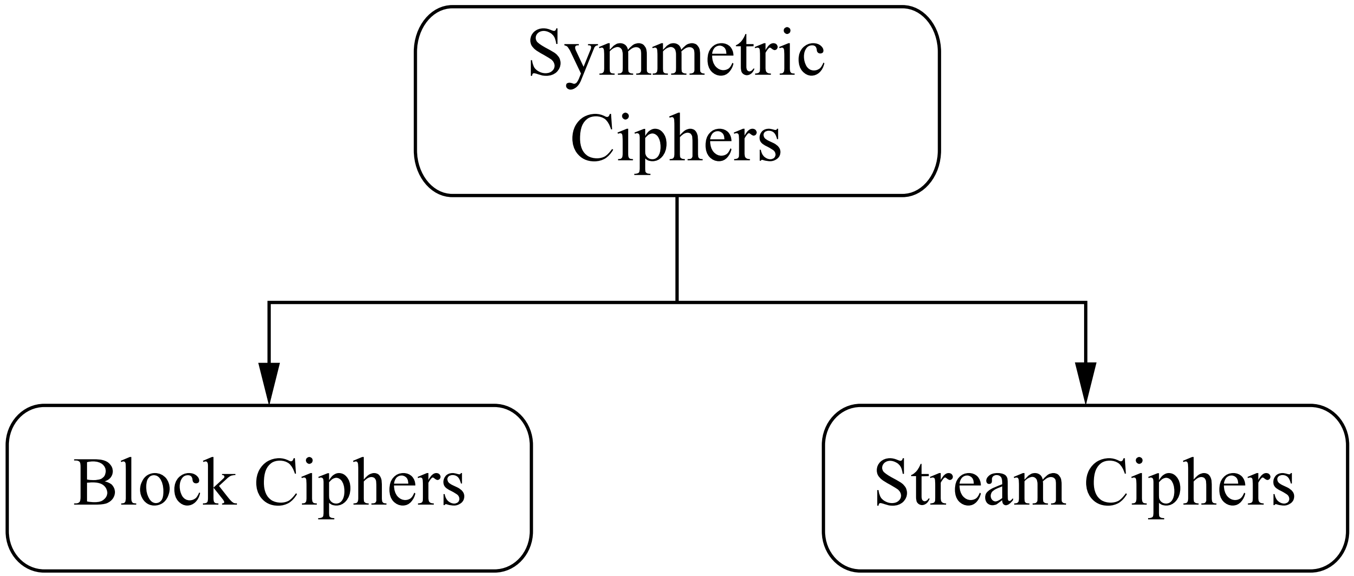
1. One of the two main categories of symmetric-key ciphers: Stream ciphers
2. An unbreakable stream cipher: The One-Time Pad

READINGS

Chapter 2 (Stream Ciphers), Paar & Pelzl

- Section 2.1: Introduction
- Section 2.2.2: The One-Time Pad

SYMMETRIC CIPHERS

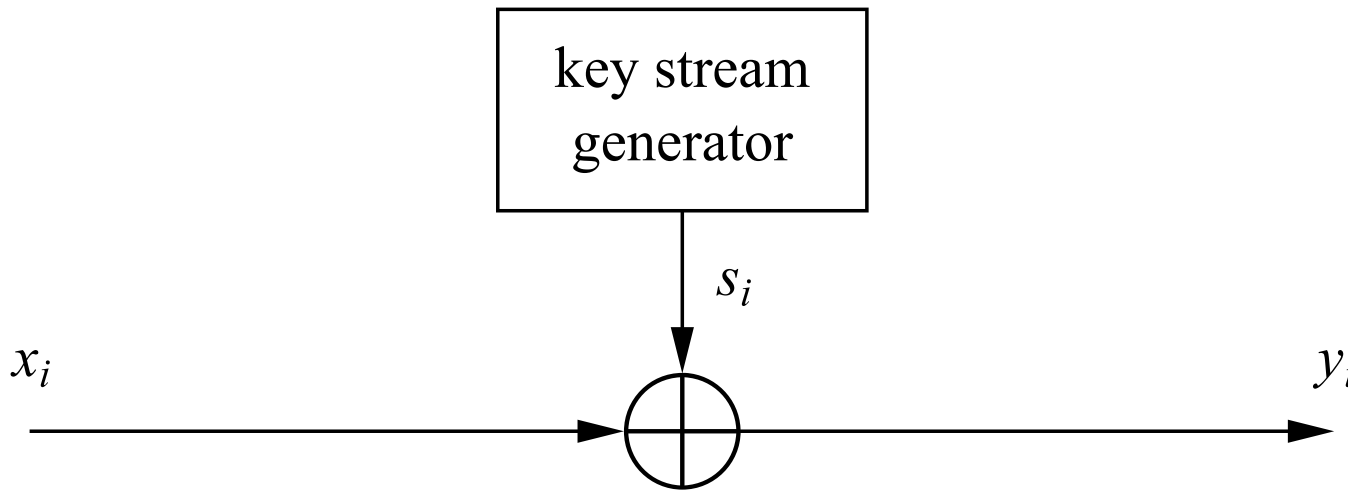


DATA REPRESENTATION

- Historical ciphers were designed to operate on letters
  - Computers did not exist
  - Consider the Caesar cipher: Both plaintext and ciphertext are represented as letters
- Modern cryptography is designed to work for any data that can be represented in a computer
  - Thus, they are designed to operate on binary data
  - Any data processed by a computer (text, images, videos, etc.) is represented in binary format

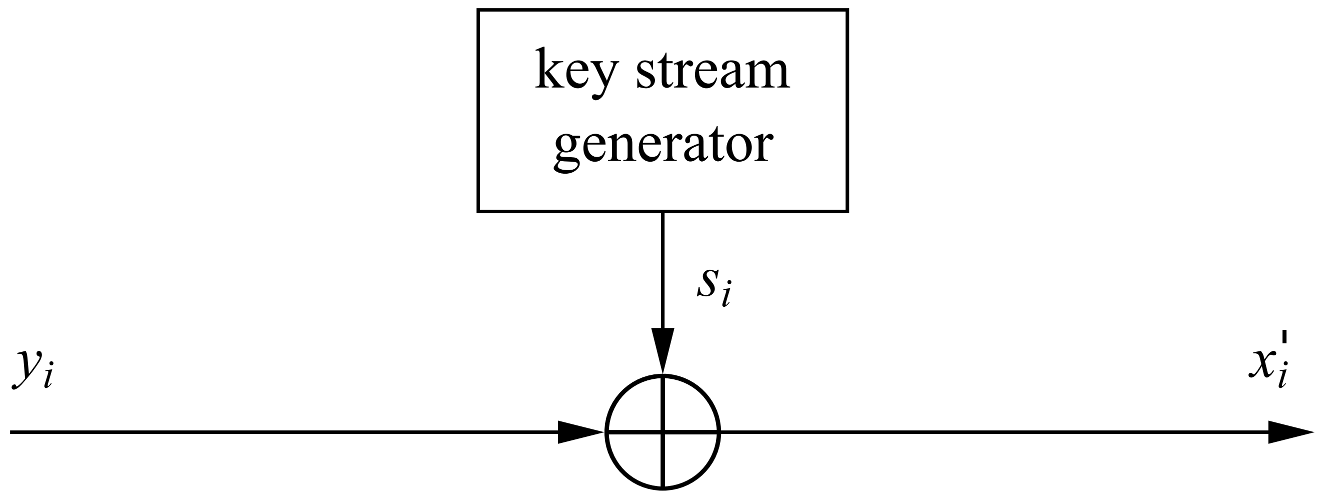
STREAM CIPHERS: ENCRYPTION

- *Stream ciphers* encrypt data sequentially one bit at a time
- For encryption: The sender bitwise-`XOR`s a stream of plaintext data with a *keystream* to generate a stream of ciphertext



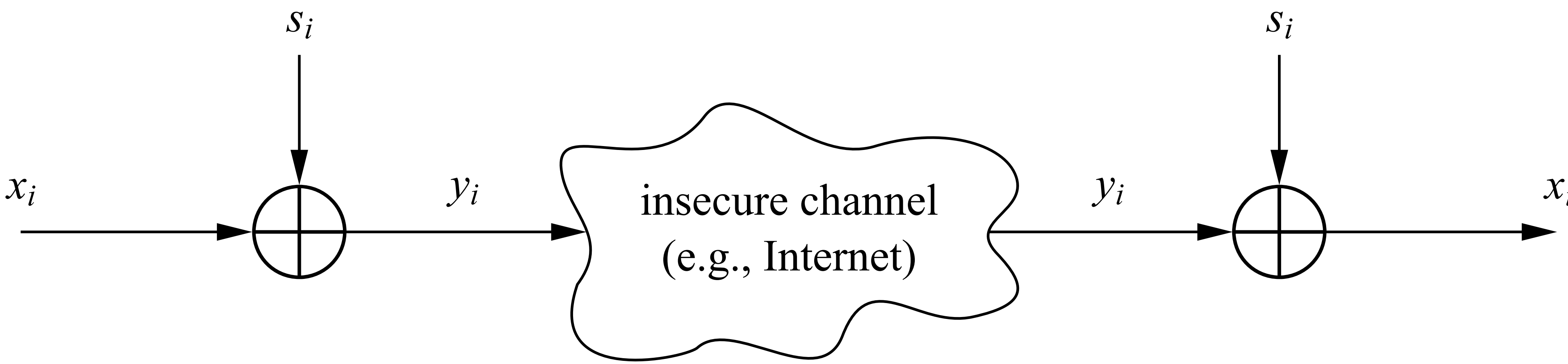
STREAM CIPHERS: DECRYPTION

- For decryption: The receiver bitwise-XORs the stream of ciphertext with a *keystream* to recover the original stream of plaintext data
- The receiver must be able to generate/retrieve the same keystream that the sender used for encryption



STREAM CIPHERS: MATHEMATICAL REPRESENTATION

- Let $x_i, y_i, s_i \in \{0, 1\}$ be individual bits of plaintext, ciphertext, and the keystream, respectively
- Encryption: $y_i = e_{s_i}(x_i) \equiv x_i + s_i \mod 2$
- Decryption: $x_i = e_{s_i}(y_i) \equiv y_i + s_i \mod 2$



Logical XOR (represented $\oplus$) is equivalent to addition $\mod 2$

STREAM CIPHERS: PROOF THAT DECRYPTION WORKS

$$
\begin{aligned}
d_{s_i}(y_i) &\equiv y_i + s_i \mod 2 \\
&\equiv (x_i + s_i) + s_i \mod 2 \\
&\equiv x_i + 2s_i \mod 2 \\
&\equiv x_i \mod 2
\end{aligned}
$$

STREAM CIPHERS: WHY XOR WORKS WELL

- Attacker cannot infer value of the plaintext or keystream by observing the ciphertext

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

EXAMPLE

Alice encrypts and sends the message `x=1000001` (ASCII code for the letter `A`) to Bob:

Alice		**Bob**
$x_0,\ldots,x_6 = 1000001$		$y_0,\ldots,y_6 = 1101101$
$\oplus$	$\xrightarrow{1101101}$	$\oplus$
$s_0,\ldots,s_6 = 0101100$		$s_0,\ldots,s_6 = 0101100$
$y_0,\ldots,y_6 = 1101101$		$x_0,\ldots,x_6 = 1000001$

Note that both Alice and Bob must agree in advance to use the same keystream `s=0101100` for Bob to be able to successfully recover the original message from the ciphertext `y=1101101`.

THE KEYSTREAM: PRACTICAL CHALLENGES (1)

- The security of a stream cipher depends entirely on the keystream, and thus it *must not be revealed to the attacker*
- The keystream should be *indistinguishable from randomly-generated bits*; an attacker should be unable to reconstruct it

THE KEYSTREAM: PRACTICAL CHALLENGES (2)

- The sender (to encrypt) and receiver (to decrypt) both need the keystream; this dictates that either:
  - The entire keystream needs to be shared in advance over a secure channel

    or
  - The sender and receiver should be able to generate the keystream from a secret value

THE ONE-TIME PAD (OTP)

- OTP was used for diplomatic and military communication in the 1900s
- Sometimes referred to as the Vernam Cipher
- The sender and receiver must agree on a keystream consisting of *random, uniformly distributed* bits
  - Can be printed on a pad of paper, or burned onto an optical disc, and physically transported to destination
- When the sender has some bits of plaintext to send, it must be `XOR`d using an equal number of bits of the keystream
  - Those bits of the keystream are then "crossed out", **never to be used again**

OTP: CONSEQUENCES OF KEYSTREAM REUSE

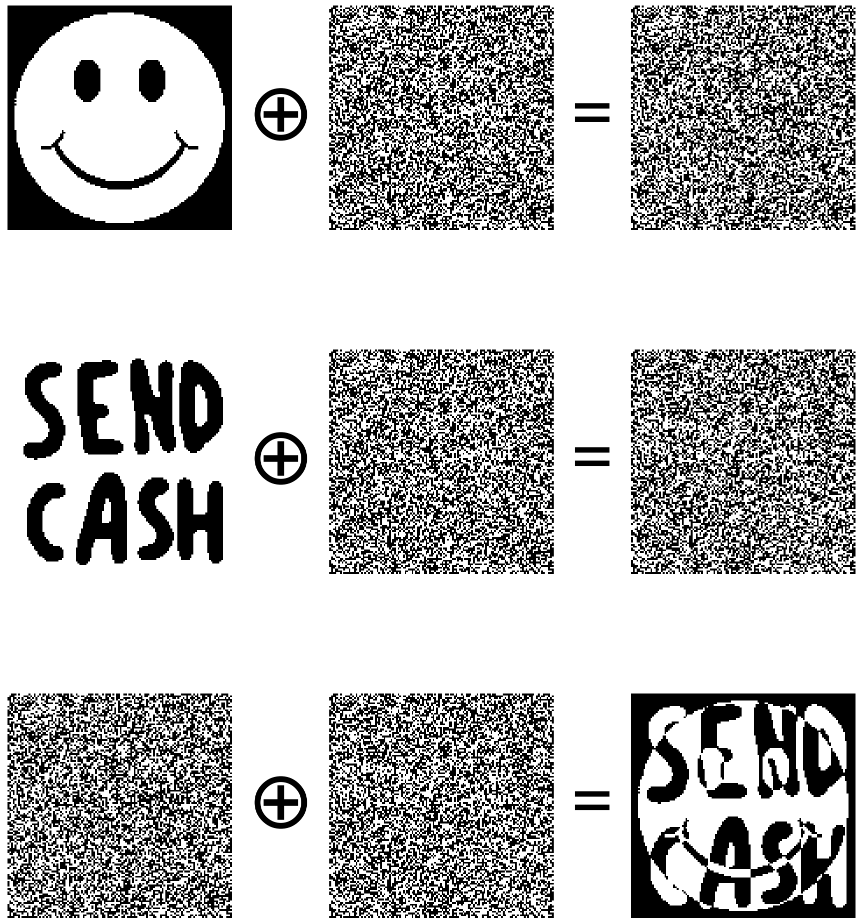
Suppose Alice encrypts two plaintexts x_1 and x_2 using the same keystream s:

$$e_s(x_1) = x_1 \oplus s = y_1$$
$$e_s(x_2) = x_2 \oplus s = y_2$$

If Oscar intercepts the two ciphertexts y_1 and y_2:

$$\begin{aligned} y_1 \oplus y_2 &= (x_1 \oplus s) \oplus (x_2 \oplus s) \\ &= x_1 \oplus x_2 \end{aligned}$$

KEYSTREAM REUSE ILLUSTRATED



Source

SECURITY OF THE ONE-TIME PAD

- The OTP is *information-theoretically secure*, i.e., impossible to break even with unlimited computational power
- Each ciphertext bit is computed from two unknowns, so it is mathematically impossible to solve

$$
\begin{aligned}
y_0 &\equiv x_0 + s_0 \mod 2 \\
y_1 &\equiv x_1 + s_1 \mod 2 \\
&\vdots
\end{aligned}
$$

- Trying to guess the keystream is futile: There exists a keystream that decrypts the ciphertext to *any possible plaintext* of the same length

OTP PRACTICAL CHALLENGE (1): SINGLE-USE SECRET

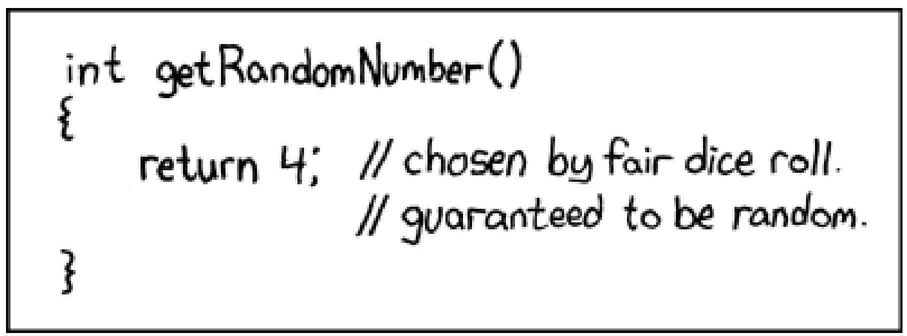
- The keystream must be at least the size of the plaintext
- What happens when the sender runs out of keystream bits?
  - Cannot reuse the keystream in whole or in part (we already saw why not)
  - Only secure solution is to generate and securely transport another one-time pad

OTP PRACTICAL CHALLENGE (2): KEYSTREAM TRANSPORT

- OTP solves the problem of communicating data by imposing a requirement to transport a pre-shared keystream of equal length
  - e.g., if you want to securely send 1GB of data, you must first securely generate and agree upon a 1GB keystream
  - Very cumbersome, so not practical

OTP PRACTICAL CHALLENGES (3): RANDOM NUMBER GENERATION

- Properly generating a large amount of bits is not easy
  - Certain statistical properties must be fulfilled, e.g., the bits should have a *uniform distribution* and should be *independent* of the plaintext



Source: xkcd

OTP: CONFIDENTIALITY VS. INTEGRITY

- OTP offers confidentiality, but not integrity
- What are the implications for a *passive* vs. *active* attacker?

RECAP

- Common feature of all stream ciphers:
  - The sender `XOR`s the message with a keystream to encrypt
  - The receiver `XOR`s the ciphertext with a keystream to decrypt
  - The keystream must be secret and must be generated in a way that fulfills certain statistical properties, in order to appear *indistinguishable from random*
- The One-Time Pad is a perfectly secure symmetric-key cipher, but it is impractical for real-world applications

COMING UP NEXT

Is it possible to build a stream cipher that allows the sender and receiver to agree on a *small* secret (e.g., 128 bits) and use that to *generate* a keystream that is long enough to encrypt/decrypt large amounts of data?

Would we have to compromise any security in doing so?