# ECE 443/518 – Computer Cyber Security
# Lecture 03 Stream Ciphers

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

August 26, 2024

# Outline

# Reading Assignment

- This lecture: UC 2
- Next lecture: UC 3, 4 except 4.3, 5.1 – 5.1.5

# Outline

# Overview: The Substitution Cipher

- Large key space helps to resist brute-force attacks from computationally bounded passive adversaries.
- Effective cryptanalysis methods exist because ciphertext leaks statistics of plaintext.
- If a cipher could resist brute-force attacks from computationally unbounded passive adversaries, will it also resist any cryptanalysis method?
  - Including those cryptanalysis methods designed by someone really smart in future?
- Unconditional security
  - A.k.a. information-theoretically secure
  - If a cryptosystem cannot be broken even with infinite computational resources.

# Brute-Force Attack Revisited

Given $y$, $e()$, and $d()$, find $x$ and $k$ such that:

$$y = e_k(x), \text{ and } x = d_k(y).$$

- Key space $K$: the set of all possible keys
- For each $k \in K$, compute $x = d_k(y)$ and report $k$ if $x$ is meaningful.
- What does "meaningful" mean?
- What if there are many $k$'s such that $x = d_k(y)$ is meaningful?

# One-Time Pad (OTP)

- Plaintext: $x = x_0, x_1, \ldots$, where $x_j \in \{0, 1, \ldots, N-1\}$.
- Key: $k = k_0, k_1, \ldots$, where $k_j \in \{0, 1, \ldots, N-1\}$.
  - Choose a key that is of the same length as the message.
- Ciphertext: $y = y_0, y_1, \ldots$, where $y_j \in \{0, 1, \ldots, N-1\}$.
- $e()$: $y = e_k(x)$ where $y_j = (x_j + k_j) \bmod N$.
  - For $N$ being power of 2, e.g. bytes, using xor is also popular.
- $d()$: $x = d_k(y)$ where $x_j = (y_j - k_j) \bmod N$.
- Indistinguishable plaintext
  - For any $y = e_k(x)$, there exists $x'$ and $k'$ such that $x' = d_{k'}(y)$.
  - So the adversary cannot tell whether the actual plaintext is $x$ or $x'$.

# OTP and Unconditional Security

- For unconditional security, usually we prefer to choose a key, say $k'$, such that for $x' = d_{k'}(y)$, $x'$ is equally probable among all valid plaintexts.
  - Otherwise adversaries may learn that some plaintexts are more probable than others, eventually breaking the cryptosystem.
- For OTP, this implies the key $k$ should be chosen uniformly from the key space.
- <u>One-Time</u>
  - For different messages, when the key space is large enough, very unlikely you'll generate the same $k$ twice for uniform distribution.
  - If you reuse $k$ for the messages with the same length and the adversaries know that, then they can learn correlations among plaintext from correlations among ciphertext, potentially learning even more.

# Practical Considerations

- Key establishment
  - Need a random key for every message.
  - Size of each random key is the same as each message.
- If Alice and Bob have a secure channel to communicate these keys, why don't they just use it to send messages?
- Pre-shared random bits
  - Work for finite number of messages
- How to generate random bits?
- Can we generate more random bits from some random "seeds" deterministically?
  - So Alice and Bob can get more key bits from existing key bits?

# Outline

# True Random Number Generators (TRNG)

- True random number generators: output cannot be reproduced.
  - Via a random physical process, e.g. flipping a fair coin multiple times.
- Yes, computers can collect/generate true random bits.
  - Special TRNG devices: semiconductor noise, clock jitter, radioactive decay, etc.
  - Software measurements: delay variation between events, e.g. network packets and user inputs.
  - Concerns: speed, correlation between neighboring measurements.
- No, we can't generate more true random bits from some random "seeds" deterministically.
  - By definition of true random number.

# Pseudorandom Number Generators (PRNG)

- Pseudorandom number generators: generate sequences using a seed deterministically, usually via a function $f$,
$$s_0 = \text{seed}, s_{i+1} = f(s_i, s_{i-1}, \ldots).$$
  - Statistically similar to true random sequences.
  - Reproducible.
  - Widely used for simulation and testing.
- Most are predictable: one can derive the seed by observing a sub-sequence, and then predict what comes next.
  - Not suitable for use in cryptosystem where the seed should be a secret.
  - A major source of weakness for homebrew cryptosystems.
- Cryptosystem need to use unpredictable cryptographically secure pseudorandom number generators (CSPRNG).

# Outline

One-Time Pad

Random Number Generators

Stream Ciphers

# Stream Ciphers

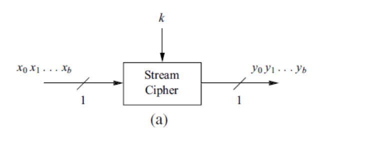


**Fig. 2.2** Principles of encrypting $b$ bits with a stream (a)

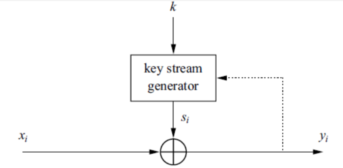


**Fig. 2.3** Synchronous and asynchronous stream ciphers

(Paar and Pelzl)

- Encode plaintext $x$ and ciphertext $y$ both as binary strings.
- Generate a key stream $s$ from the secret key $k$.
  - Synchronous: $s$ depends only on $k$.
  - Asynchronous: $s$ depends on both $k$ and $x$
- Usually use xor $\oplus$ to encrypt $x$ into $y$ using $s$.
  - Same function for both encryption and decryption.
  - Allow to process $x$, $y$, and $s$ as blocks of bits.

# (Synchronous) Stream Ciphers

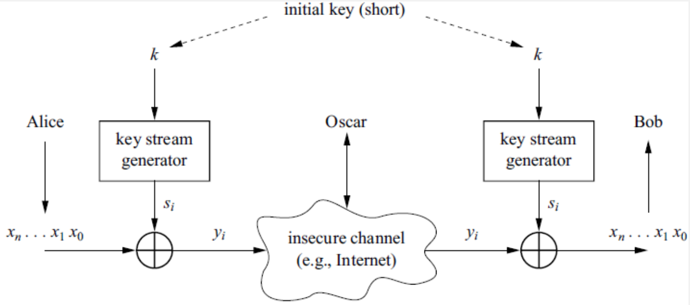


**Fig. 2.5** Practical stream ciphers

(Paar and Pelzl)

- What's the difference between stream ciphers and OTP?
- What's the danger to NOT use CSPRNG for the key stream generator?
- If Alice want to send a second message to Bob using the same key $K$, should she restart the key stream generator?

# Known-Plaintext Attack and CSPRNG

- Oscar may know some (but not all) bits of $x$
  - Packet headers, file headers, etc.
  - Or Oscar may even trigger Alice to send some information whose plaintext could be known.
- When the plaintext $x$ is encrypted with the key stream $s$ bit by bit via xor, for those known $x$ bits, adversaries may recover the corresponding bits in $s$.
- So the key stream generator must be CSPRNG – otherwise adversaries may predict all following bits of $s$, and then decrypt $y$ to obtain $x$.

# Linear Congruential Generator is NOT CSPRNG

$$
\begin{aligned}
S_0 &= \text{seed}, \\
&\dots \\
S_{i+1} &\equiv AS_i + B \pmod{m}, \\
S_{i+2} &\equiv AS_{i+1} + B \pmod{m}, \\
&\dots
\end{aligned}
$$

- A widely used software PRNG.
- $k = (\text{seed}, A, B)$: secret.
- $m$: known cryptosystem parameter.
- $S_i, S_{i+1}, S_{i+2}$: consecutive blocks of bits in the key stream
- Possible to solve for $A$ and $B$ if $S_i, S_{i+1}, S_{i+2}$ are obtained via known-plaintext attacks.

# LFSR is NOT CSPRNG

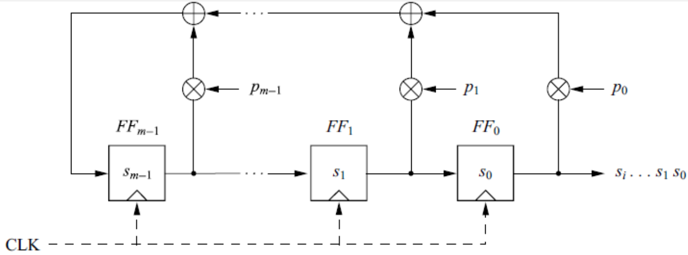


**Fig. 2.7** General LFSR with feedback coefficients $p_i$ and initial values $s_{m-1}, \dots, s_0$

(Paar and Pelzl)

$$s_{i+m} \equiv s_{i+m-1}p_{m-1} + \cdots + s_{i+1}p_1 + s_i p_0 \pmod 2.$$

- A widely used hardware PRNG: $\oplus$ for xor, $\otimes$ for and
- $k = (p_0, p_1, \dots, p_{m-1})$: secret.
- Possible to solve for $p_0, p_1, \dots, p_{m-1}$ if $2m$ consecutive bits of $s$ are obtained via known-plaintext attacks.

# How to design a CSPRNG?

- Can we prove that a PRNG is a CSPRNG?

# Summary

- One-time pad and unconditional security
- Stream ciphers and CSPRNG