Stream Ciphers
LFSR Sequences

CSSE/MA479: Cryptography

Day 6

Cryptographically-Secure PRNG Examples

- Algorithms based on existing strong crypto
  - e.g. AES (blockcipher), SHA (hash function) [later]
  - Say, take least-significant bits of AES(s+1), AES(s+2), etc. for seed s.

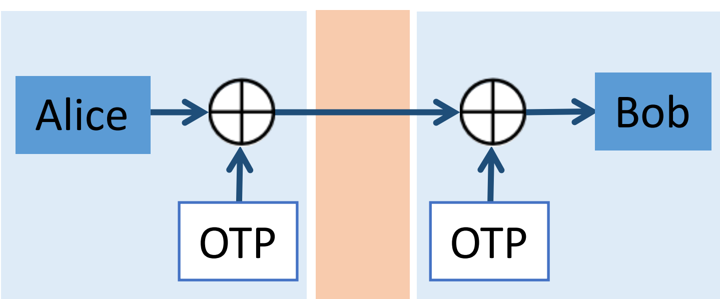
- Purpose-built algorithms (Better known as stream ciphers)
  - RC4 stream cipher – most popular, dates from 1987. Has security issues! Used in WEP, WPA; old versions of Microsoft Office, Adobe PDF; optionally in SSL (now prohibited)
  - Mathematical-proof-backed ciphers such as Blum-Blum-Shub (1986) – the authors proved its security assuming the computational difficulty of factoring
  - Modern ciphers such as those in the eSTREAM portfolio: set of 7 stream ciphers collected as part of a 2004–2008 effort “to promote the design of efficient and compact stream ciphers suitable for widespread adoption.” E.g., Salsa20, ChaCha

- Note: all PRNGs require inputting a “random seed”! (Use a true RNG?)

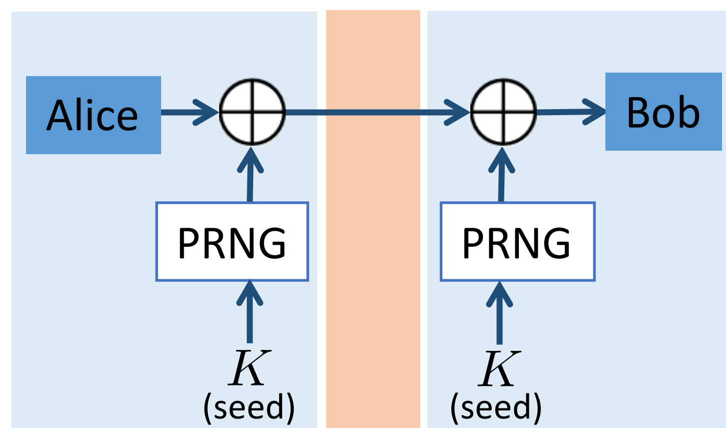
From PRNGs to Stream Ciphers

- As long as we’re not using true randomness…
- And in fact, our PRNGs are deterministic and require a seed…
- Why not use the seed as a key, and the PRNG output as a “non-ideal one-time pad”?

- This idea forms the basis of stream ciphers



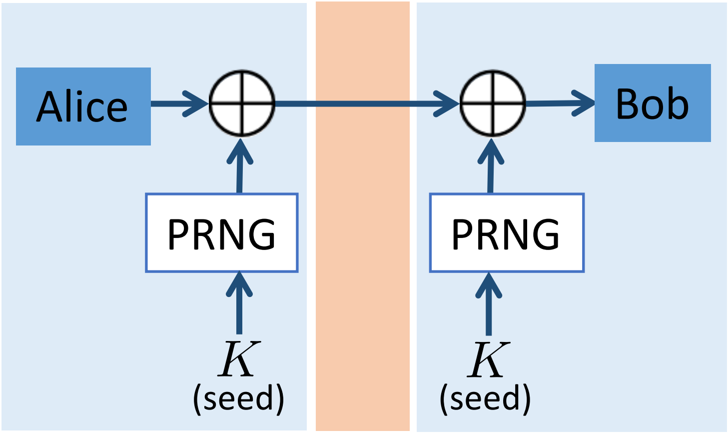
One-time pad



Stream cipher

Stream Ciphers

- Essentially, non-ideal one-time pad: plaintext bits XORed with a *pseudorandom* bitstream determined by a fixed-length key (often, the PRNG seed!)



- Alternative to (more common) block ciphers. Stream ciphers:
  - can execute faster
  - have lower hardware complexity
  - difficult to implement, prone to serious weakness if used incorrectly

RC4

- Ron Rivest 1987; leaked 1994
- Choose a key: bitstring of length between 40 and 256.
- Key scheduling: Outputs S, a permutation of [0, 1, 2, …, 255].

Algorithm 1 RC4 Key Scheduling Algorithm

for i from 0 to 255 **do**
    $S[i] := i$
$j := 0$
for i from 0 to 255 **do**
    $j := (j + S[i] + \text{key}[i \bmod \text{keylength}]) \bmod 256$
    $\text{swap}(S[i], S[j])$

- Main generation: two shifting indices, more swaps, output is a 2-phase lookup into the array S

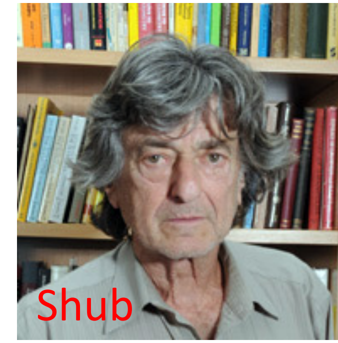
Algorithm 2 RC4 Pseudorandom Generation Algorithm (PRGA)

$i := 0$
$j := 0$
while GeneratingOutput: **do**
    $i := (i + 1) \bmod 256$
    $j := (j + S[i]) \bmod 256$
    $\text{swap}(S[i], S[j])$
    output $S[(S[i] + S[j]) \bmod 256]$

RC4 Weaknesses

- Biases in output! Mantin and Shamir showed that
  - The second byte in the output is 0 with probability 2/256 instead of 1/256.
  - The first two bytes are both 0 with probability $3/256^2$ instead of $1/256^2$.
- Biases in the state S that is output by the Key Scheduling Algorithm!
  - The probability that S[0] = 1 is about 37% larger than 1/256
  - The probability that S[0] = 255 is about 26 less than 1/256.
- Because of these, often a version called RC4-drop[n] is used
  - the first n bits are dropped before starting the keystream.
- Although any key length from 40 to 255 bits can be chosen, the low end of this range is susceptible to a brute-force attack.
- Must avoid nonrandom or related keys!

Blum-Blum-Shub



- 1986, Lenore Blum, Manuel Blum, Michael Shub
- "Quadratic residue generator"

1. Generate two large primes p, q, both congruent to 3 (mod 4).
2. Set n = pq, choose random integer x coprime to n.
3. To initialize, set initial seed $x_0 \equiv x^2 \pmod{n}$.
4. Produce a sequence of pseudorandom bits $b_1, b_2, \ldots$:

$$x_j \equiv x_{j-1}{}^2 \pmod{n}$$

b_j is the least significant bit of x_j.

- Unpredictability has been proven assuming the difficulty of factoring
- Quite slow, due to mathematical operations

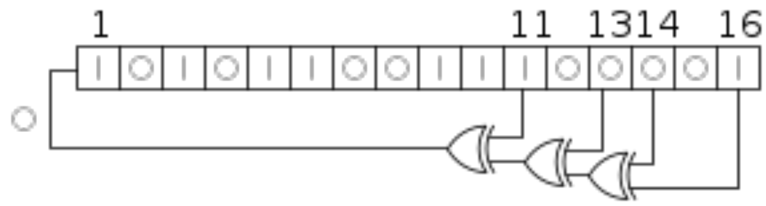
Linear Feedback Shift Register (LFSR)

- Common (insecure, on its own) building block: *linear feedback shift register* (LFSR) sequences

- *LFSR can be thought of as the "mother" (or maybe more like the sick great-uncle) of all pseudorandom generators.*

  --*Boaz Barak*

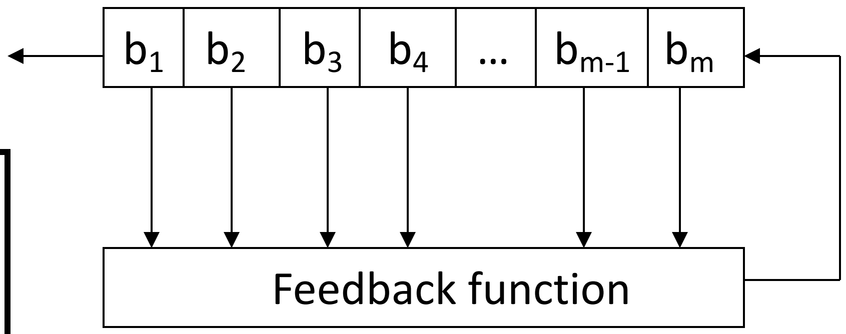
- Name comes from hardware implementation (can also be implemented in software)



Linear Feedback Shift Register (LFSR) Sequences

[Name comes from hardware implementation]

Generated "keystream"



Shift register "cascade of flip-flops"

To encrypt plaintext of length n, generate an n-bit sequence and XOR with the plaintext.

- Need initial conditions (bits in register) and a function to generate more terms.
- Example:

  $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0;$

  $x_{n+5} = x_n + x_{n+2} \pmod 2$

- What mathematical concept does this relate to?

Linear Feedback Shift Register (LFSR) Sequences

- A recurrence relation!
  - Specify initial conditions and coefficients, for example:
  - $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0;$
  - $x_{n+5} = 1x_n + 0x_{n+1} + 1x_{n+2} + 0x_{n+3} + 0x_{n+4} \pmod 2$
- In general,

$$\boxed{x_{n+m} = \sum_{i=0}^{m-1} c_i x_{n+i}}$$

- How long until it repeats? (the *period* of the sequence)
  - 10 bits (initial 01000, coeffs 10100) generates ____ bits
  - Sage demo

```
F=GF(2); o=F(0); l=F(1) % so we work mod 2
c=[l,o,l,o,o]           % coefficients
k=[o,l,o,o,o]           % initial vector
n=50                    % generate first 50 bits
lfsr_sequence(c,k,n)    % built-in Sage functionality
```

Long periods

- LFSR *can* generate sequences with long periods
  - Like Vigenere with long key: hard to decrypt!
  - Potentially, lots of “bang for the buck”! But it depends on the key
- Good example: $x_{n+31} = x_n + x_{n+3} \pmod 2$
  - How many bits do we need to represent this recurrence?
    - 62 bits
  - How long is the period?
    - $2^{31} – 1$ (over 2 billion!)
      - Why “– 1”?
    - Why is this maximal?
- See http://www.ece.cmu.edu/~koopman/lfsr/index.html for a list of maximal-period generators
  - **Theorem**: *A LFSR produces a maximal-period sequence if and only if its characteristic polynomial is a primitive polynomial.* [More info]()

Example from size-31 list: 40000004

Implicit x_{n+0} term

```
   Hex:    4   0   0   0   0   0   0   4
Binary:01000000000000000000000000000100 1
```

x_{n+31} x_{n+3}

$x_{n+31} + x_{n+3} + x_n = 0 \pmod 2$

Attacking LFSR

- Security downside: Linear! Highly vulnerable to known plaintext attacks.
- Plaintext XOR ciphertext = portion of key

- Say, 011010111100
- Guess key length, say 3
  - key: $x_{n+3} = c_0 x_n + c_1 x_{n+1} + c_2 x_{n+2}$

$$\underbrace{\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}}_{M_3} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Can't solve this one for c_i's:
$\det(M_3) = 0$.
So, wrong key length!

Attacking LFSR

- Determine key length by computing determinants
  - Theorem: If N is the length of the shortest recurrence that generates the sequence, then
    - $\det(M_N) = 1 \pmod 2$
    - $\det(M_n) = 0 \pmod 2$ for all $n > N$.
  - E.g. determinants: 1 1 0 0 1 0 0 0 0 0 0 0 0 … → length is probably 5
- Use the key length to solve for the recurrence.
- Verify solution by using it to generate the whole key
- Demo

Attacking LFSR: Demo

```
% first import example "L100"
F=GF(2)
o=F(0); l=F(1)
L100= [l, o, o, l, l, o, o, l, o, o, l, l, l, o, o, o, l, l, o, o, o, l,
       o, l, o, o, o, l, l, l, l, o, l, l, o, o, l, l, l, l, l, o, l, o,
       l, o, l, o, o, l, o, l, l, o, l, l, o, l, o, l, l, o, o, o, o, l,
       l, o, l, l, l, o, o, l, o, l, o, l, l, l, l, o, o, o, o, o, o, o,
       l, o, o, o, l, o, o, l, o, o, o, o];

% Try to find length:
for Adim in range(2,20):
    A = matrix([L100[j:j+Adim] for j in range(Adim)])
    print(Adim, A.det())

% Try to solve:
Adim = 8
A = matrix([L100[j:j+Adim] for j in range(Adim)])
b = vector(L100[Adim:(Adim+Adim)])
sol = A.solve_right(b); print(sol)

% Then check the answer!
lfsr_sequence(list(sol),L100[:Adim],100) == L100
```

LFSRs in Practice

- To avoid such attacks, we can combine LFSRs nonlinearly.
- Example. A5/1 stream cipher.
  - Provides over-the-air communication privacy in the GSM (2G) cellular telephone standard
  - Uses three LFSRs
  - Somewhat insecure—has a feasible known plaintext attack
- In 3G networks, the cipher has been replaced with the block cipher KASUMI in a stream cipher mode of operation. Also insecure

The registers are clocked in a stop/go fashion: a register is clocked if the clocking bit (orange) agrees with the majority bit. Hence at each step at least two or three registers are clocked, and each register steps with probability ¾.