STREAM CIPHERS and PRGs
Stateful Generators

Initially, St is a random seed

Operation:

\[ X[1] X[2] X[3] \ldots \] is the output sequence and should be “pseudorandom”. 
\[(X[1] \ldots X[m], St) \leftarrow G(St, m)\]

means we

- Run \(G\) with starting state \(St\) for \(m\) steps
- Let \(X[1] \ldots X[m]\) be the output blocks produced
- Let \(St\) be the updated state
Usage for Encryption

Alice maintains a state $St_A$ and Bob maintains a state $St_B$. Initially: $St_A = St_B$ is a random seed.

\[
\mathcal{E}(M[1] \ldots M[m]) \\
(X[1] \ldots X[m], St_A) \leftarrow G(St_A, m) \\
\text{for } i = 1, \ldots, m \text{ do} \\
C[i] \leftarrow X[i] \oplus M[i]
\]

\[
\mathcal{D}(C[1] \ldots C[m]) \\
(X[1] \ldots X[m], St_B) \leftarrow G(St_B, m) \\
\text{for } i = 1, \ldots, m \text{ do} \\
M[i] \leftarrow X[i] \oplus C[i]
\]

Note that the states must be synchronized!
Usage for Pseudorandom Bit Generation

$G$ is initialized with a random seed and its outputs are then used coins for any purpose needing randomness, including:

- Keys
- IVs for block-cipher based encryption
- Nonces
- Simulations
Methods

• Linear Congruential Generators (LCGs)
• Linear Feedback Shift Registers (LFSRs)

These have
• Good statistical properties: \#1’s \approx \#0’s; Chi-square; \ldots
• But are predictable: Given some outputs can infer future ones

Predicatability can be exploited to break encryption privacy via a chosen-message attack.

Cryptographic constructs
• (Alleged)-RC4
• SEAL (1.0, 2.0)
INDR : Indistinguishability from random

- Pick a random seed $S_t$ and let $(X_1[1] \ldots X_1[m], S_t) \leftarrow G(S_t, m)$
- Pick $X_0[1] \ldots X_0[m]$ at random
- Pick a challenge bit $b$ at random

$X_b[1] \ldots X_b[m] \xrightarrow{A} b'$

$A$ is trying to compute $b$.

$G$ is secure if no practical $A$ has high advantage.
Let $G$ be a stateful generator with seed length $s$ and output-block length $n$.

**Game INDR$_G$**

**procedure Initialize**

$\text{St} \leftarrow \{0, 1\}^s; b \leftarrow \{0, 1\}$

**procedure Next($m$)**

$(X_1[1] \ldots X_1[m], \text{St}) \leftarrow G(\text{St}, m)$

$X_0[1] \ldots X_0[m] \leftarrow \{0, 1\}^{nm}$

return $X_b[1] \ldots X_b[m]$

**procedure Finalize($b'$)**

return $(b = b')$

The indr advantage of adversary $A$ is

$$\text{Adv}^{\text{indr}}_G(A) = 2 \Pr \left[ \text{INDR}^A_G \Rightarrow \text{true} \right] - 1$$
Let $E : \{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a block cipher, and define:

algorithm $G(\text{St})$

$(K, i) \leftarrow \text{St}$ // Parse St as $(K, i)$

$X \leftarrow E_K(i + 1)$; return $(X, (K, i + 1))$

- State has the form $(K, i)$ where $K$ is a key for $E$ and $i$ is an $n$-bit integer ($0 \leq i < 2^n$).
- Initial state is $(K, 0)$ where $K \leftarrow \{0, 1\}^k$. 

![Diagram]

$E$
Fact: If $E$ is a secure PRF, then $G$ is an INDR secure PRG.

Similarly, other modes of operation of block ciphers also give rise to PRGs.
Let $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a block cipher and define algorithm $G(\text{St})$

$$(K, S) \leftarrow \text{St} \quad \text{// Parse St as (K, S)}$$

$X \leftarrow E_K(S); S \leftarrow E_K(X); \text{return}(X, (K, S))$

- state has the form $(K, S)$ where $K$ is a key for $E$ and $S \in \{0,1\}^n$
- Initial state has both $K$ and $S$ chosen at random.

The standard uses $E = \text{DES}$ in 2-key EDE mode.

Analysis: [DHL02]
Other Standards, Implementations

- Two PRGs are specified in FIPS-186, based on DES or SHA-1
- NIST SP 800-90 specifies hash-based, HMAC-based, CTR-based, and ECC based generators.
- ANSI X9.31 and ANSI X9.62
- OpenSSL specifies a SHA-1 based PRG.
Suppose adversary obtains $S_t^2$. Then

- It can compute $X[3]X[4] \ldots$

Forward security requires that the answer to the latter question be “NO”.

Important in the face of exposure due to malware and system compromise.
Forward Security Failures

If adversary gets $(K, 2)$ then it can compute


Similar failures for ANSI X9.17.
Let $E : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a block cipher, and define:

algorithm $G(K)$

$X \leftarrow E_K(0)$; $K \leftarrow E_K(1)$; return $(X, K)$

- State is a key $K$ for $E$
- Initial state is a random $K \leftarrow \{0, 1\}^n$. 

\begin{center}
\begin{tikzpicture}
\node (k0) at (0,0) {$K_0$};
\node (e1) at (2,0) {$E$};
\node (e2) at (4,0) {$E$};
\node (x1) at (4,-2) {$X[1]$};
\node (k1) at (6,0) {$K_1$};
\node (e3) at (8,0) {$E$};
\node (e4) at (10,0) {$E$};
\node (x2) at (10,-2) {$X[2]$};
\node (k2) at (12,0) {$K_2$};
\draw[->] (k0) -- (e1);
\draw[->] (e1) -- (e2);
\draw[->] (e2) -- (x1);
\draw[->] (k1) -- (e3);
\draw[->] (e3) -- (e4);
\draw[->] (e4) -- (x2);
\end{tikzpicture}
\end{center}
In practice, random number generation (RNG) involves

- Seeding
- PRG

Failures in RNG are common: Netscape, Debian Linux, ...
They arise from failures in either component.
In principle, we know how to design good PRGs.
But seeding remains a problem.

Typical methods:

- Maintain an “entropy pool” based on system events, user keystrokes, ...
- Mix in more entropy as needed.

Example: OpenSSL
Alternative: Hardware RNGs