

# Random Number Generation

CMSC 426 - Computer Security

Slides originally by Dr. Marron, modified by Robert Joyce

# Outline

- Properties of PRNGs
- LCGs
- Blum, Blum, Shub
- NIST SP 800-90A

# Random Number Uses

- Generation of symmetric keys
- Generation of primes ( $p$  and  $q$ ) for RSA
- Generation of secret keys for Diffie-Hellman
- Nonces for cryptographic protocols

# The “P” in “PRNG”

- Don't typically have access to a true random number generator (RNG).
- RNGs require some source of random noise, i.e. special hardware.
- Instead, use an algorithm that produces numbers that appear random - a **Pseudo-Random Number Generator** or **PRNG**.
- NIST documents also refer to a PRNG as a **Deterministic Random Bit Generator (DRBG)**.

# PRNG Requirements

- **Statistical Properties.** What does it mean to “appear random?”
  - Output of the PRNG should be *uniformly distributed*.
  - Outputs should appear *independent*. Can not infer a value from a previous or future value.
- **Unpredictability.** For cryptography, the statistics don't matter so much as that the values be unpredictable.

# A simple PRNG

- The **Linear Congruential Generator (LCG)** is perhaps the most commonly used PRNG.
- Given constants  $a$ ,  $c$ , and  $m$  and an initial seed  $X_0$ , generate numbers according to the formula

$$X_{n+1} = (a X_n + c) \bmod m$$

- The selection of the constants is important.

# LCG Examples

- Example:  $a = c = 1$ .
- Example:  $a = 7, c = 0, m = 32, X_0 = 1$ .
- Example:  $a = 5, c = 0, m = 32, X_0 = 1$ .

# Good LCGs?

- What would make an LCG good?
  1. Full-period generating — generates all values  $0 < X < m$ .
  2. Should appear random as determined by a battery of statistical tests.
  3. Efficient on current architectures (64 bit).

# LCG Parameters

- If  $n$  is a power of two, choose  $a$ ,  $c$  such that
  1.  $c$  is relatively prime to  $n$  (so  $c$  is odd).
  2.  $a - 1$  is divisible by 4.

Hull & Dobell, *Random Number Generators*, SIAM Review, Vol. 4, No. 3 (July 1962), pp. 230 - 254.

- Some examples from [Wikipedia](#):

	$n$	$a$	$c$
<b>glibc</b>	$2^{31}$	1103515245	12345
<b>MS Quick C</b>	$2^{32}$	214013	2531011

# LCGs are Weak

- Unfortunately, LCGs are not appropriate for cryptography.
- Python uses a PRNG called a *Mersenne Twister*, which is better than an LCG, but still not good enough for cryptography.

# Blum, Blum, Shub

- We've seen that a simple PRNG isn't suitable for cryptography (LCG)
- The Blum, Blum, Shub (BBS) generator is simple and secure — but has its own limitations.
- BBS is provably secure if used correctly; its security is based on the difficulty of factoring.

# BBS Parameters

- Construct a composite modulus  $M = p \cdot q$  with the following properties:
  - $p$  and  $q$  are primes of “cryptographic size” (at least 512 bits each)
  - $p$  and  $q$  are both congruent to 3 mod 4.
- Generate a *seed*  $x_0$ , a random positive integer less than  $M$  and relatively prime to  $M$ .

# BBS Generation

- The state of the generator is updated according to the rule:

$$x_{i+1} = x_i^2 \bmod M.$$

- From each  $x_i$ , extract the low-order bit. That is, the pseudo-random sequence is:

$$b_i = x_i \bmod 2, i = 1, 2, 3, \dots$$

- **Example:**  $p = 7, q = 11, x_0 = 17.$

# Security and Efficiency

- Given a sequence of  $b_i$  values, it is “difficult” to recover a state  $x_j$  (future or past).
- The difficulty is proven to be equivalent to a hard mathematical problem, which in turn is believed to be equivalent to factoring  $M$ .
- **So what is the downside?** Efficiency. We are computing one modular exponentiation for *each bit* of pseudo-random output.

# NIST SP 800-90A

- PRNG based on AES in CTR mode which *is* suitable for cryptographic applications.
- Note: NIST uses the term *Deterministic Random Bit Generator* (DRBG) rather than PRNG.
- The algorithm consists of separate *Initialization* and *Generation* phases.
- We'll see a simplified version of the standard using AES-128...

# Initialization

- The following steps initialize the PRNG:
  1. Obtain 256 bits of random "seed" data; the first 128 bits will be denoted  $(K_0)$ , and the remaining 128 bits will be denoted  $(V_0)$ .
  2. Initialize  $V$  and  $K$  to zero.
  3. Update  $V \leftarrow V + 1 \bmod 2^{128}$ .
  4. Encrypt  $V$  with key  $K$ ; save the output  $K'$ .
  5. Update  $V \leftarrow V + 1 \bmod 2^{128}$ .
  6. Encrypt  $V$  with key  $K$ ; save the output  $V'$ .
  7. Set  $K = K_0 \oplus K'$  and  $V = V_0 \oplus V'$ .

# Generation

- Generation of  $n$  blocks of pseudo-random data:
  1. Update  $V \leftarrow V + 1 \bmod 2^{128}$ .  
Encrypt  $V$  with key  $K$ ; save output as  $X$ .
  2. Update  $Output \leftarrow Concatenate(Output, X)$ .
  3. Repeat steps 1 - 3 a total of  $n$  times.
  4. Return  $Output$ .
- After generation,  $V$  and  $K$  are updated using steps 3 - 7 of the Initialization.
- A counter tracks the total number of pseudo-random bits produced; after some threshold, the PRNG must be re-initialized.

# Which PRNG to use?

- For *non-cryptographic* applications an LCG is usually sufficient.
- For *small volumes of critical pseudo-random bits*, BBS would be a reasonable choice, but there are few other practical uses
- For *large volumes of pseudo-random bits*, a PRNG from SP 800-90A will be secure and efficient.

There are many other PRNGS: this is just a sample!