Pseudo Random Numbers

Presentation by:
V. Balasubramanian
SSN College of Engineering

Objectives

- The concepts of randomness and unpredictability with respect to random numbers
- Pseudorandom Generators
- Pseudorandom Functions

Perfect Secrecy

- Consider the Vigenere cipher over the lowercase English alphabet, where the key length can be anything from 8 to 12 characters. What is the size of the key space for this scheme?

Answer

- $26^8 + 26^9 + 26^{10} + 26^{11} + 26^{12}$

Question

- Consider the Vigenere cipher over the lowercase English alphabet, where the key has length 8. For which of the following message spaces will this scheme be perfectly secret?

Answers

- The set of all 7 / 8-character strings of lowercase English letters.

Question

- Say we have a scheme with a claimed proof of security with respect to some definition, based on some assumption. The scheme was successfully attacked when used in the real world. What are possible reasons for this?

Answer

- The proof might be incorrect.
  The assumption may be false.
  The definition of security may not correctly capture the real-world threat model.

Q4

- In the definition of perfect secrecy, what threat model is assumed?

Answer

- The attacker can eavesdrop on a single ciphertext.

Q5

- Consider the Vigenere cipher over the lowercase English alphabet, where the key can have length 1 or length 2, each with 50% probability. Say the distribution over plaintexts is Pr[M=’aa’] = 0.4 and Pr[M=’ab’] = 0.6. What is Pr[C=’bb’]?

Answer

- Given:
- Pr[M=’aa’] = 0.4
- Pr[M=’ab’] = 0.6.
- key can have length 1 or length 2, each with 50% probability.

Contd...

- Pr[C='bb']
- $\sum P(K).P(d_k(y)$

- $P(k_{length1}) = \frac{1}{26}$
- $P(K_{length2}) = \frac{1}{26^2}$
- $P(C = bb) = (P(K_{length1})P(M = ' aa') + P(K_{length2})P(M = ' aa') + P(K_{length2})P(M = ' ab')$
- $\left(\frac{1}{26} + \frac{1}{26^2}\right) * 0.5 * 0.4 + \left(0 + \frac{1}{26^2}\right) * 0.5 * 0.6$
- (0.03846+ 0.001479)*0.2 + (0.001479)*0.3
- 0.0084

Q5

- Consider the Vigenere cipher over the lowercase English alphabet, where the key can have length 1 or length 2, each with 50% probability. Say the distribution over plaintexts is Pr[M=’aa’] = 0.4 and Pr[M=’ab’] = 0.6. What is Pr[M=’aa’ | C=’bb’]?

Sol

- Given:
- Pr[M=’aa’] = 0.4
- Pr[M=’ab’] = 0.6.

- Computed P(C=bb)=0.0084
- Pr[M=’aa’ | C=’bb’]

$$\Pr[x \mid y] = \frac{\Pr[x] \times \Pr[y \mid x]}{\Pr[y]}$$

Contd...

- P(c=bb|M=aa) = $\left(\frac{1}{26}+\frac{1}{26^2}\right) * 0.5$
- P(M=aa|c=bb) = $P(M) * \frac{\mathrm{P(C|M)}}{\mathrm{P(C)}}$
- $= 0.4 * 0.3994 * \frac{0.5}{0.0084}$
- =0.9509

Q6

- Which of the following are true for obtaining perfect secrecy using the one-time pad, assuming the message space contains messages all of some fixed length?

Answer

- The key must be as least as long as the messages in the message space.
- The key should be shared between the two communicating parties, and kept secret from any potential attacker.
- The key should be chosen uniformly.

Q7

- Consider the one-time pad over the message space of 5-bit strings, where Pr[M=00100] = 0.1 and Pr[M=11011] = 0.9. What is Pr[C=00000]?

sol

- Given: Pr[M=00100] = 0.1
  Pr[M=11011] = 0.9
- Pr[C=00000] = P(K1)* Pr[M=00100]
  +P(k2) Pr[M=11011] = 0.1*1/32
  +0.9*1/32 = 1/32 = 0.3125

Q8

- The Vigenere cipher is perfectly secret if the length of the key is equal to the length of the messages in the message space.

Perfect Secrecy

- Requires that absolutely no information about the plaintext is leaked, even to eavesdroppers with unlimited computational power
  - Seems unnecessarily strong

Bounded Attackers

- Consider brute-force search of key space; assume one key can be tested per clock cycle
- Desktop computer $\approx 2^{57}$ keys/year
- Supercomputer $\approx 2^{80}$ keys/year
- Supercomputer since Big Bang $\approx 2^{112}$ keys
  - Restricting attention to attackers who can try 2^{112} keys is fine!
- Modern key space: 2^{128} keys or more…

Indistinguishabiity

- Define a randomized exp't $\text{PrivK}_{A,\Pi}$:
  1. A outputs $m_0, m_1 \in \mathcal{M}$
  2. $k \leftarrow \text{Gen}$, $b \leftarrow \{0,1\}$, $c \leftarrow \text{Enc}_k(m_b)$
  3. $b' \leftarrow A(c)$

  Adversary A *succeeds* if b = b', and we say the experiment evaluates to 1 in this case

Cryptosystem

- A *private-key encryption scheme* is defined by three PPT algorithms (Gen, Enc, Dec):
  - Gen: takes as input 1^n; outputs k. (Assume $|k| \geq n$.)
  - Enc: takes as input a key k and message $m \in \{0,1\}^*$; outputs ciphertext c

    $$c \leftarrow Enc_k(m)$$

  - Dec: takes key k and ciphertext c as input; outputs a message m or "error"

Encryption and Plain Text

- In practice, we want encryption schemes that can encrypt arbitrary-length messages
- In general, encryption does not hide the plaintext length
  - The definition takes this into account by requiring m_0, m_1 to have the same length
- But beware that leaking plaintext length can often lead to problems in the real world!
  - Obvious examples…
  - Database searches
  - Encrypting compressed data

Pseudorandomness

- Important building block for computationally secure encryption
- Important concept in cryptography

Random Numbers

- A number of network security algorithms and protocols based on cryptography make use of random binary numbers:
  - Key distribution and reciprocal authentication schemes
  - Session key generation
  - Generation of keys for the RSA public-key encryption algorithm
  - Generation of a bit stream for symmetric stream encryption

Random Number Generator

- Pseudorandom number generators (PRNGs) or deterministic random bit generators (DRBGs).
- The other strategy is to produce bits non-deterministically using some physical source that produces some sort of random output. True random number generators (TRNGs) or non-deterministic random bit generators (NRBGs).

NIST Document

- SP 800-90A, B, C, 22

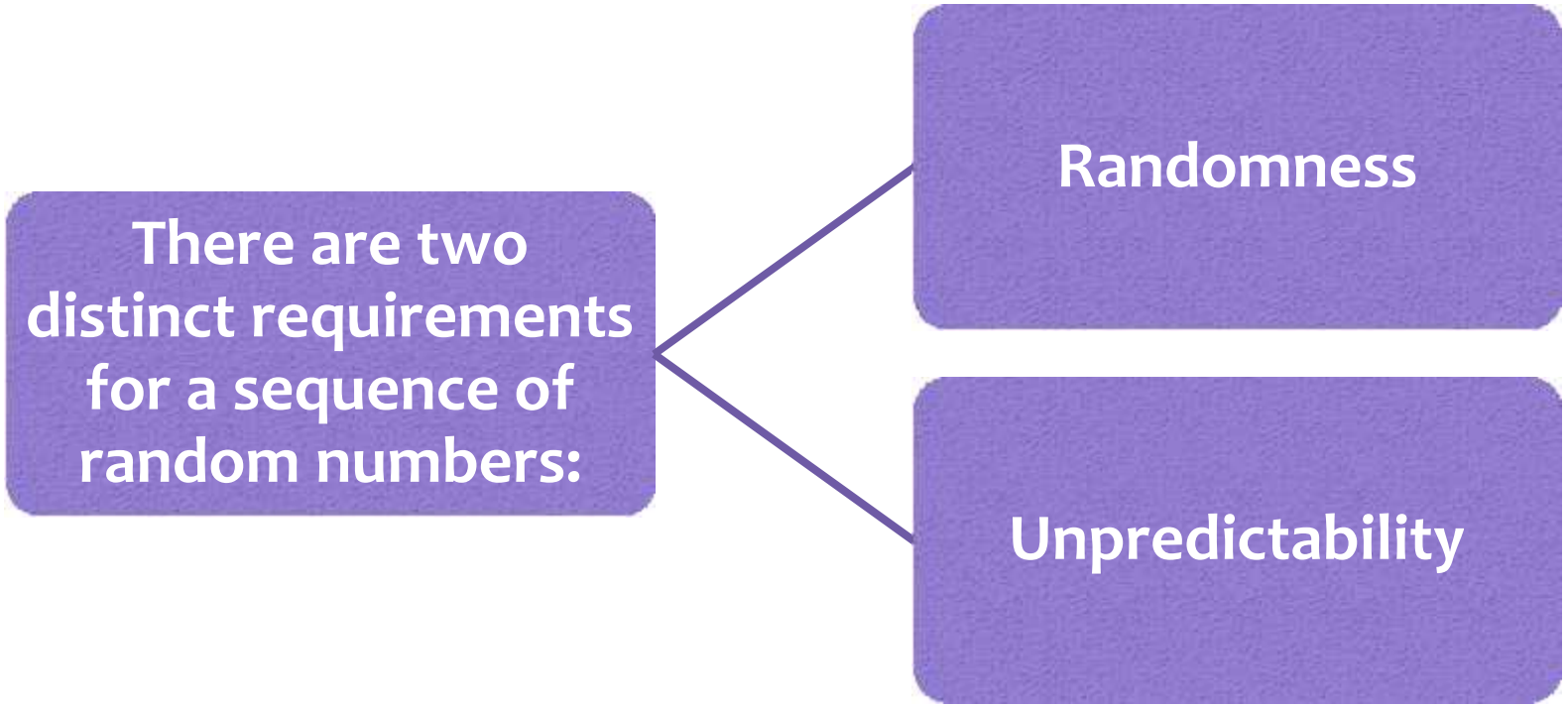
Use

- Nonces are used for handshaking to prevent replay attacks. The use of random numbers for the nonces frustrates an opponent's efforts to determine or guess the nonce, in order to repeat an obsolete transaction.
- Session key generation
- Generation of keys for the RSA public-key encryption algorithm
- Generation of a bit stream for symmetric stream encryption

Randomness

- The generation of a sequence of allegedly random numbers has been that the sequence of numbers be random in some well-defined statistical sense.
- **Uniform distribution:** The distribution of bits in the sequence should be uniform; that is, the frequency of occurrence of ones and zeros should be approximately equal.
- **Independence:** No one subsequence in the sequence can be inferred from the others.

Requirements



Randomness

- The generation of a sequence of allegedly random numbers being random in some well-defined statistical sense has been a concern

Two criteria are used to validate that a sequence of numbers is random:

Uniform distribution

- The frequency of occurrence of ones and zeros should be approximately equal

Independence

- No one subsequence in the sequence can be inferred from the others

Miller Rabin Theorem

- A number of effective algorithms exist that test the primality of a number by using a sequence of randomly chosen integers as input to relatively simple algorithms

Unpredictable

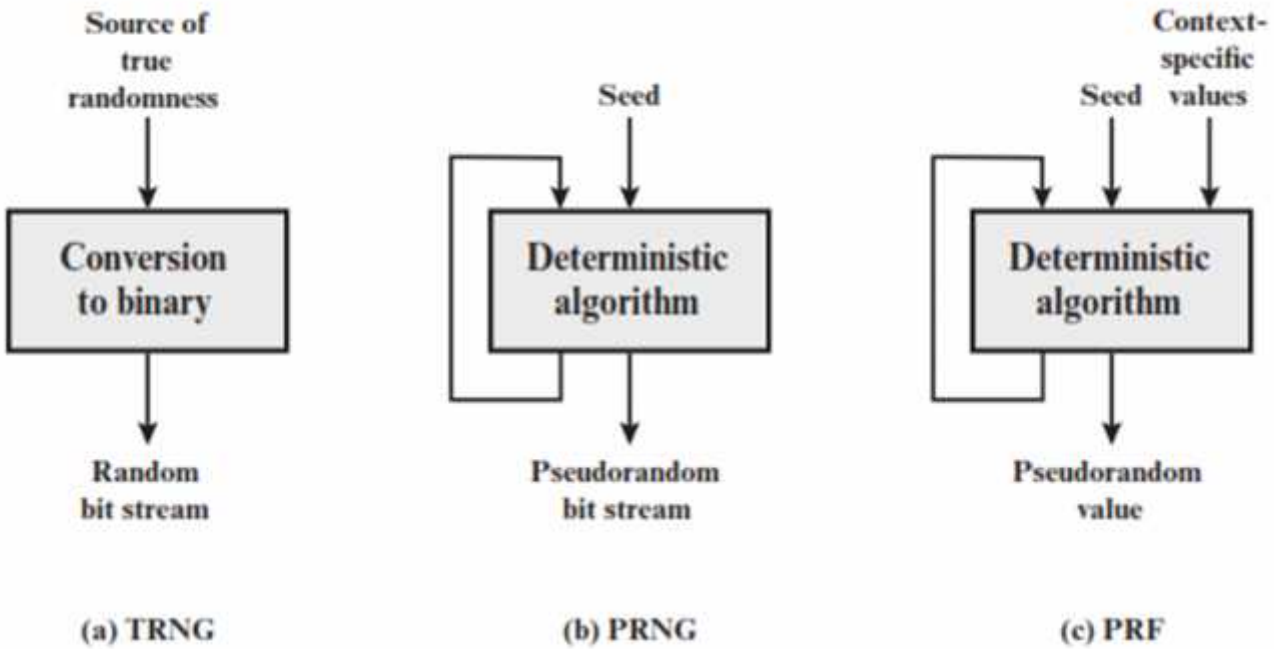
- Session key generation, and stream ciphers, the requirement is not just that the sequence of numbers be statistically random but that the successive members of the sequence are unpredictable.

- With "true" random sequences each number is statistically independent of other numbers in the sequence and therefore unpredictable
  - True random numbers have their limitations, such as inefficiency, so it is more common to implement algorithms that generate sequences of numbers that appear to be random
  - Care must be taken that an opponent not be able to predict future elements of the sequence on the basis of earlier elements

Pseudo Randomness

- Cryptographic applications typically make use of algorithmic techniques for random number generation
- These algorithms are deterministic and therefore produce sequences of numbers that are not statistically random
- If the algorithm is good, the resulting sequences will pass many tests of randomness and are referred to as *pseudorandom numbers*

PRG



TRNG = true random number generator
PRNG = pseudorandom number generator
PRF = pseudorandom function

Figure 8.1 Random and Pseudorandom Number Generators

True Random Number Generator (TRNG)

- Takes as input a source that is effectively random
- The source is referred to as an *entropy source* and is drawn from the physical environment of the computer
  - Includes things such as keystroke timing patterns, disk electrical activity, mouse movements, and instantaneous values of the system clock
  - The source, or combination of sources, serve as input to an algorithm that produces random binary output
- The TRNG may simply involve conversion of an analog source to a binary output
- The TRNG may involve additional processing to overcome any bias in the source

Pseudorandom Number Generator (PRNG)

- Takes as input a fixed value, called the *seed*, and produces a sequence of output bits using a deterministic algorithm
  - Quite often the seed is generated by a TRNG
- The output bit stream is determined solely by the input value or values, so an adversary who knows the algorithm and the seed can reproduce the entire bit stream
- Other than the number of bits produced there is no difference between a PRNG and a PRF

Two different forms of PRNG

Pseudorandom number generator

- An algorithm that is used to produce an open-ended sequence of bits
- Input to a symmetric stream cipher is a common application for an open-ended sequence of bits

Pseudorandom function (PRF)

- Used to produce a pseudorandom string of bits of some fixed length
- Examples are symmetric encryption keys and nonces

PRNG Requirements

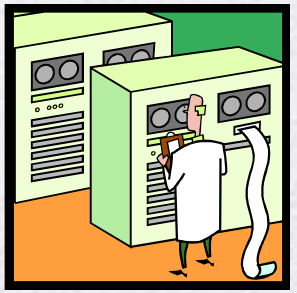
- The basic requirement when a PRNG or PRF is used for a cryptographic application is that an adversary who does not know the seed is unable to determine the pseudorandom string
- The requirement for secrecy of the output of a PRNG or PRF leads to specific requirements in the areas of:
  - Randomness
  - Unpredictability
  - Characteristics of the seed

Requireents

- A 128-bit seed, together with some context-specific values, are used to generate a 128-bit secret key that is subsequently used for symmetric encryption. Under normal circumstances, a 128-bit key is safe from a brute-force attack.
- randomness, unpredictability, and the characteristics of the seed

Randomness

- The generated bit stream needs to appear random even though it is deterministic
- There is no single test that can determine if a PRNG generates numbers that have the characteristic of randomness
  - If the PRNG exhibits randomness on the basis of multiple tests, then it can be assumed to satisfy the randomness requirement
- NIST SP 800-22 specifies that the tests should seek to establish three characteristics:
  - Uniformity
  - Scalability
  - Consistency



SP 800 22

- **Uniformity:** At any point in the generation of a sequence of random or pseudorandom bits, the occurrence of a zero or one is equally likely, that is, the probability of each is exactly 1/2. The expected number of zeros (or ones) is $n/2$, where n = the sequence length.
- **Scalability:** Any test applicable to a sequence can also be applied to subsequences extracted at random. If a sequence is random, then any such extracted subsequence should also be random. Hence, any extracted subsequence should pass any test for randomness.
- **Consistency:** The behavior of a generator must be consistent across starting values (seeds). It is inadequate to test a PRNG based on the output from a single seed or a TRNG on the basis of an output produced from a single physical output.

Vulnerability

- Wikileaks
- snowdown

Randomness Tests

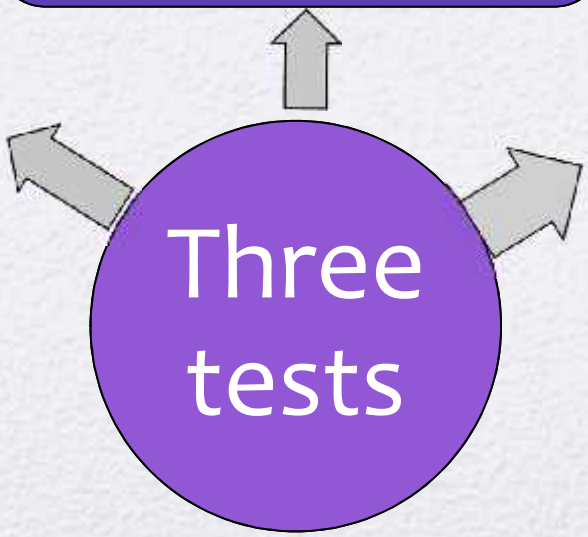
- SP 800-22 lists 15 separate tests of randomness

Frequency test

- The most basic test and must be included in any test suite
- Purpose is to determine whether the number of ones and zeros in a sequence is approximately the same as would be expected for a truly random sequence

Runs test

- Focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits bounded before and after with a bit of the opposite value
- Purpose is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence



Maurer's universal statistical test

- Focus is the number of bits between matching patterns
- Purpose is to detect whether or not the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random

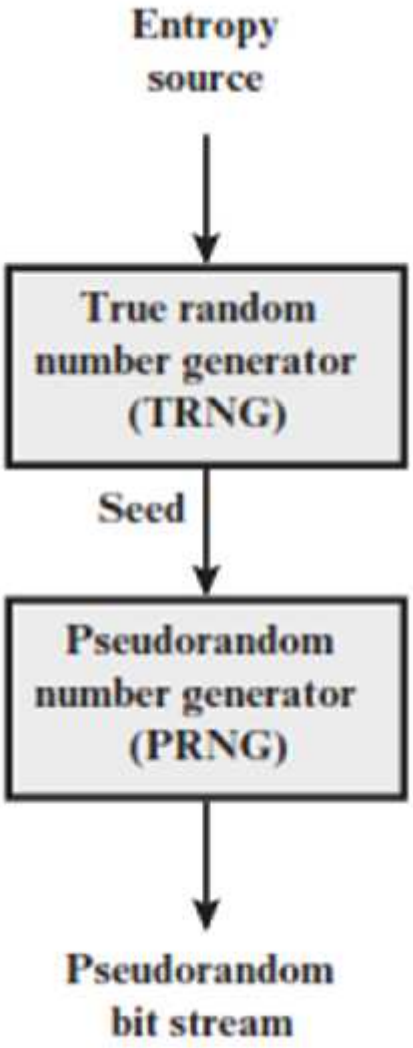
Unpredictability

- A stream of pseudorandom numbers should exhibit two forms of unpredictability:
- Forward unpredictability
  - If the seed is unknown, the next output bit in the sequence should be unpredictable in spite of any knowledge of previous bits in the sequence
- Backward unpredictability
  - It should not be feasible to determine the seed from knowledge of any generated values. No correlation between a seed and any value generated from that seed should be evident; each element of the sequence should appear to be the outcome of an independent random event whose probability is 1/2
- The same set of tests for randomness also provides a test of unpredictability
  - A random sequence will have no correlation with a fixed value (the seed)

Seed Requirement

- The seed that serves as input to the PRNG must be secure. Because the PRNG is a deterministic algorithm, if the adversary can deduce the seed, then the output can also be determined.
- Seed must be unpredictable

Seed


Entropy source
True random number generator (TRNG)
Seed
Pseudorandom number generator (PRNG)
Pseudorandom bit stream

Seed Requirements

- The seed that serves as input to the PRNG must be secure and unpredictable
- The seed itself must be a random or pseudorandom number
- Typically the seed is generated by TRNG

Algorithm Design

- Algorithms fall into two categories:
  - Purpose-built algorithms
    - Algorithms designed specifically and solely for the purpose of generating pseudorandom bit streams
  - Algorithms based on existing cryptographic algorithms
    - Have the effect of randomizing input data

Three broad categories of cryptographic algorithms are commonly used to create PRNGs:
• Symmetric block ciphers • Asymmetric ciphers • Hash functions and message authentication codes

Random Means

- What does “uniform” mean?
- Which of the following is a uniform string?
  - 0101010101010101
  - 0010111011100110
  - 0000000000000000
- If we generate a uniform 16-bit string, each of the above occurs with probability 2^{-16}

Uniform

- "Uniformity" is not a property of a *string*, but a property of a *distribution*

- A distribution on *n*-bit strings is a function $D: \{0,1\}^n \rightarrow [0,1]$ such that $\Sigma_x D(x) = 1$
  - The *uniform* distribution on *n*-bit strings, denoted U_n, assigns probability 2^{-n} to every $x \in \{0,1\}^n$

Pseudorandom

- Informal: cannot be distinguished from uniform (i.e., random)
- Which of the following is pseudorandom?
  - 0101010101010101
  - 0010111011100110
  - 0000000000000000
- Pseudorandomness is a property of a *distribution*, not a *string*

Pseudorandomness

- Fix some distribution D on *n*-bit strings
  - $x \leftarrow D$ means "sample x according to D"
- Historically, D was considered pseudorandom if it "passed a bunch of statistical tests"
  - $Pr_{x \leftarrow D}[1^{st}$ bit of x is 1$] \approx ½$
  - $Pr_{x \leftarrow D}[$parity of x is 1$] \approx ½$
  - $Pr_{x \leftarrow D}[A_i(x)=1] \approx Pr_{x \leftarrow U_n}[A_i(x)=1]$ for i = 1, …, 20

Pseudorandomness

- This is not sufficient in an adversarial setting!
  - Who knows what statistical test an attacker will use?

- Cryptographic def'n of pseudorandomness:
  - D is pseudorandom if it passes all *efficient* statistical tests

Pseudorandom

- Let D be a distribution on *p*-bit strings

- D is (t, ε)-pseudorandom if for all A running in time at most t,

$$| \Pr_{x \leftarrow D}[A(x)=1] - \Pr_{x \leftarrow U_p}[A(x)=1] | \leq \varepsilon$$

Pseudorandomness (asymptotic)

- Security parameter *n*, polynomial *p*

- Let D_n be a distribution over *p(n)*-bit strings
- Pseudorandomness is a property of a *sequence* of distributions $\{D_n\} = \{D_1, D_2, \ldots\}$

Pseudorandomness (asymptotic)

- $\{D_n\}$ is *pseudorandom* if for all probabilistic, polynomial-time distinguishers A, there is a negligible function ε such that

$$\left| \Pr_{x \leftarrow D_n}[A(x)=1] - \Pr_{x \leftarrow U_{p(n)}}[A(x)=1] \right| \leq \varepsilon(n)$$

Linear Congruential Generator

- An algorithm first proposed by **Lehmer** that is parameterized with four numbers:

m	the modulus	$m > 0$
a	the multiplier	$0 < a < m$
c	the increment	$0 \leq c < m$
X_0	the starting value, or seed	$0 \leq X_0 < m$

- The sequence of random numbers $\{X_n\}$ is obtained via the following iterative equation:

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

- If m , a , c , and X_0 are integers, then this technique will produce a sequence of integers with each integer in the range $0 \leq X_n < m$
- The selection of values for a , c , and m is critical in developing a good random number generator

Example

- a=7, $c = 0$, $m = 32$, and $X_0 = 1$
- $X_{n+1} = (aX_0 + c) mod\ m$
- $X_1 = (7 * 1 + 0) mod\ 32 = 7$
- $X_2 = (7 * 7 + 0) mod\ 32 = 17$
- $X_3 = (7 * 17 + 0) mod\ 32 = 23$
- $X_4 = (7 * 23 + 0) mod\ 32 = 1$
- $X_5 = (7 * 1 + 0) mod\ 32 = 7$

Example

- This generates the sequence {7, 17, 23, 1, 7, etc.}
- Of the 32 possible values, only four are used; thus, the sequence is said to have a period of 4.
- If, instead, we change the value of *a to 5, then the* sequence is {5, 25, 29, 17, 21, 9, 13, 1, 5, etc. }, which increases the period to 8

Linear Congruential Generator

- *m to be very large,* maximum representable nonnegative integer for a given computer
- 32 bit computer 2^{31} is chosen
- Three tests

T_1: The function should be a full-period generating function. That is, the function should generate all the numbers from 0 through $m - 1$ before repeating.

T_2: The generated sequence should appear random.

T_3: The function should implement efficiently with 32-bit arithmetic.

Contd…

- For T1 test, it can be shown that if *m is prime and c = 0*
- $m = 2^{31} - 1$

$$X_{n+1} = (aX_n) \bmod (2^{31} - 1)$$

- Of 2 billion possible value for a, only few pass all tests. $a = 7^5 = 16807$
- IBM 360 computers

Linear Congruential Generator

If an opponent knows that the linear congruential algorithm is being used and if the parameters are known (e.g., $a = 7^5, c = 0, m = 2^{31} - 1$), then once a single number is discovered, all subsequent numbers are known. Even if the opponent knows only that a linear congruential algorithm is being used, knowledge of a small part of the sequence is sufficient to determine the parameters of the algorithm. Suppose that the opponent is able to determine values for X_0, X_1, X_2, and X_3. Then

$$X_1 = (aX_0 + c) \bmod m$$
$$X_2 = (aX_1 + c) \bmod m$$
$$X_3 = (aX_2 + c) \bmod m$$

These equations can be solved for a, c, and m.

the current clock value to each random number (mod *m)*

Blum Blum Shub (BBS) Generator

- Has perhaps the strongest public proof of its cryptographic strength of any purpose-built algorithm
- Referred to as a *cryptographically secure pseudorandom bit generator* (CSPRBG)
  - A CSPRBG is defined as one that passes the *next-bit-test* if there is not a polynomial-time algorithm that, on input of the first k bits of an output sequence, can predict the $(k + 1)$st bit with probability significantly greater than 1/2
- The security of BBS is based on the difficulty of factoring n

Blum Blum

- choose two large prime numbers, p and q, that both have a remainder of 3 when divided by 4.

$$p \equiv q \equiv 3 (\text{mod } 4)$$

This notation, explained more fully in Chapter 4, simply means that (p mod 4) = (q mod 4) = 3. For example, the prime numbers 7 and 11 satisfy $7 \equiv 11 \equiv 3 (\text{mod } 4)$. Let $n = p \times q$. Next, choose a random number s, such that s is relatively prime to n; this is equivalent to saying that neither p nor q is a factor of s. Then the BBS generator produces a sequence of bits B_i according to the following algorithm:

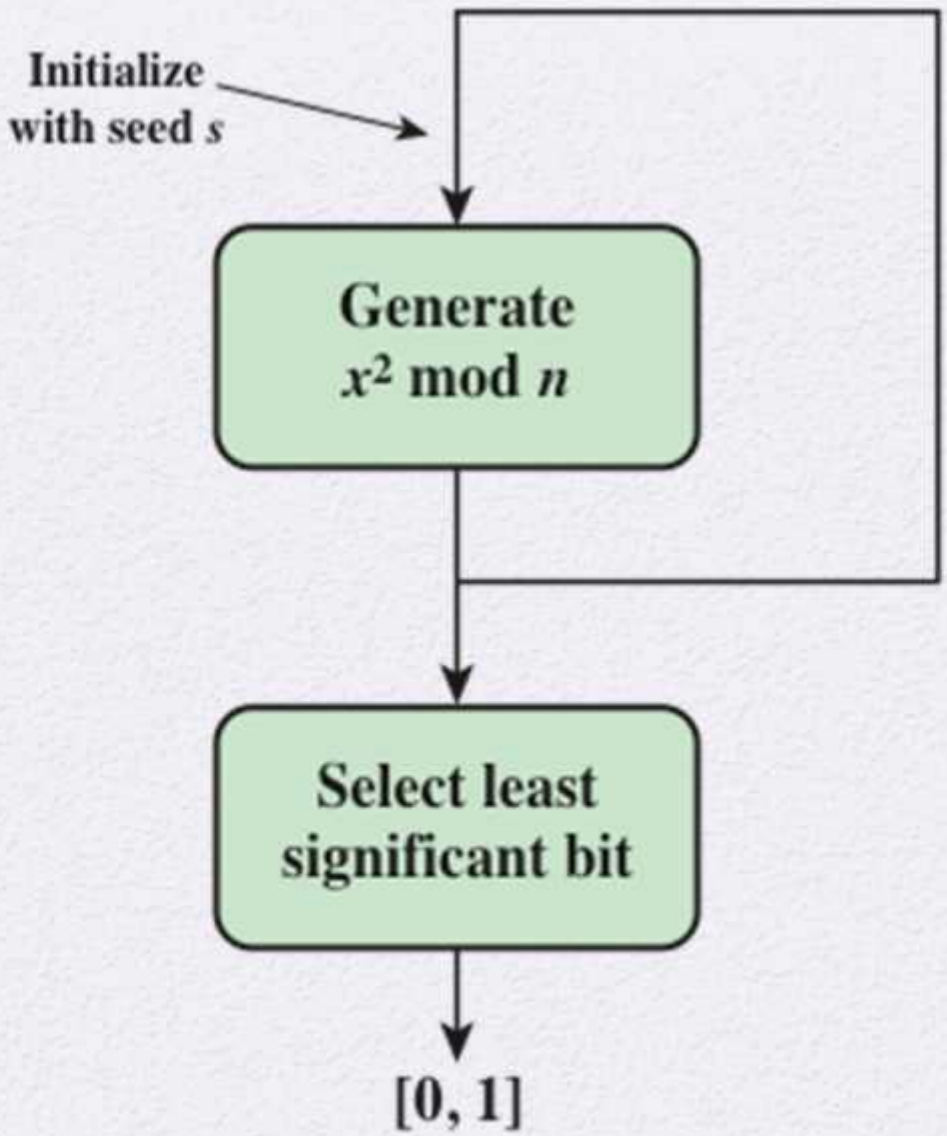


Figure 7.3 Blum Blum Shub Block Diagram

Example

- Here, n = 192649 = 383 * 503, and the seed s = 101355

$$X_0 = s^2 \bmod n$$

for $i = 1$ **to** ∞

$$X_i = (X_{i-1})^2 \bmod n$$

$$B_i = X_i \bmod 2$$

Example Operation of BBS Generator

i	X_i	B_i
0	20749	
1	143135	1
2	177671	1
3	97048	0
4	89992	0
5	174051	1
6	80649	1
7	45663	1
8	69442	0
9	186894	0
10	177046	0

i	X_i	B_i
11	137922	0
12	123175	1
13	8630	0
14	114386	0
15	14863	1
16	133015	1
17	106065	1
18	45870	0
19	137171	1
20	48060	0

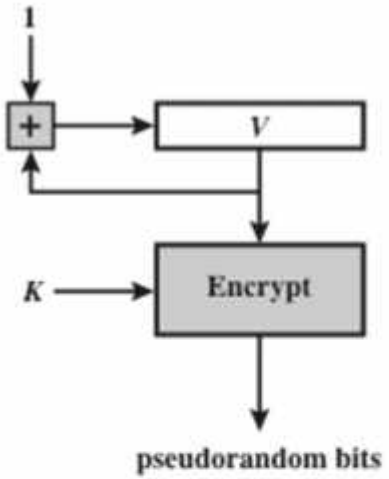
PRNG Using Block Cipher Modes of Operation

- Two approaches that use a block cipher to build a PNRG have gained widespread acceptance:
  - CTR mode
    - Recommended in NIST SP 800-90, ANSI standard X.82, and RFC 4086
  - OFB mode
    - Recommended in X9.82 and RFC 4086

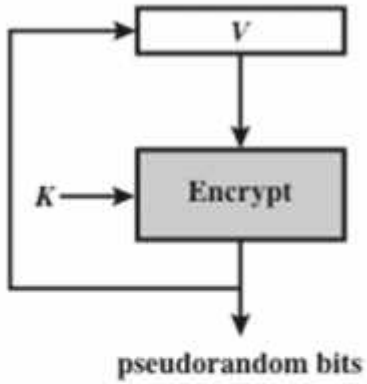
PRNG using Block Cipher

- A popular approach to PRNG construction is to use a symmetric block cipher as the heart of the PRNG mechanism. DES, AES algorithm
- A symmetric block cipher produces an output block that is apparently random.
- There are no patterns or regularities in the ciphertext that provide information that can be used to deduce the plaintext.
- Thus, a symmetric block cipher is a good candidate for building a pseudorandom number generator.

PRNG



(a) CTR Mode



(b) OFB Mode

Figure 7.4 PRNG Mechanisms Based on Block Ciphers

Algorithm

```
while (len (temp) < requested_number_of_bits) do
        V = (V + 1) mod 2^128
        output_block = E(Key, V)
        temp = temp || output_block
```

The OFB algorithm can be summarized as follows.

```
while (len (temp) < requested_number_of_bits) do
        V = E(Key, V)
        temp = temp || V
```

Table 7.2

Output Block	Fraction of One Bits	Fraction of Bits that Match with Preceding Block
1786f4c7ff6e291dbdfdd90ec3453176	0.57	—
5e17b22b14677a4d66890f87565eae64	0.51	0.52
fd18284ac82251dfb3aa62c326cd46cc	0.47	0.54
c8e545198a758ef5dd86b41946389bd5	0.50	0.44
fe7bae0e23019542962e2c52d215a2e3	0.47	0.48
14fdf5ec99469598ae0379472803accd	0.49	0.52
6aeca972e5a3ef17bd1a1b775fc8b929	0.57	0.48
f7e97badf359d128f00d9b4ae323db64	0.55	0.45

Example Results for PRNG Using OFB

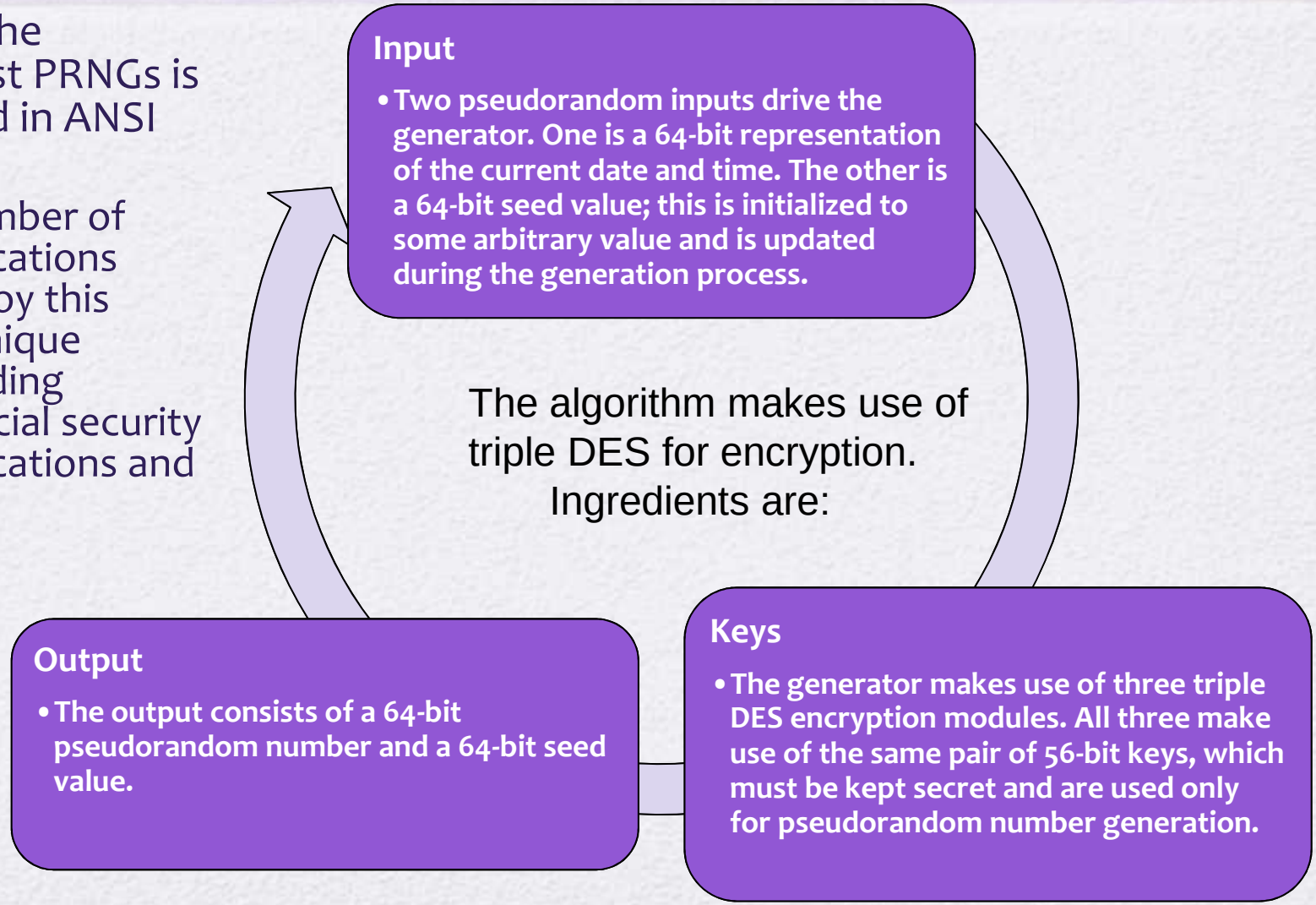
Table 7.3

Output Block	Fraction of One Bits	Fraction of Bits that Match with Preceding Block
1786f4c7ff6e291dbdfdd90ec3453176	0.57	—
60809669a3e092a01b463472fdcae420	0.41	0.41
d4e6e170b46b0573eedf88ee39bff33d	0.59	0.45
5f8fcfc5deca18ea246785d7fadc76f8	0.59	0.52
90e63ed27bb07868c753545bdd57ee28	0.53	0.52
0125856fdf4a17f747c7833695c52235	0.50	0.47
f4be2d179b0f2548fd748c8fc7c81990	0.51	0.48
1151fc48f90eebac658a3911515c3c66	0.47	0.45

Example Results for PRNG Using CTR

ANSI X9.17 PRNG

- One of the strongest PRNGs is specified in ANSI X9.17
  - A number of applications employ this technique including financial security applications and PGP



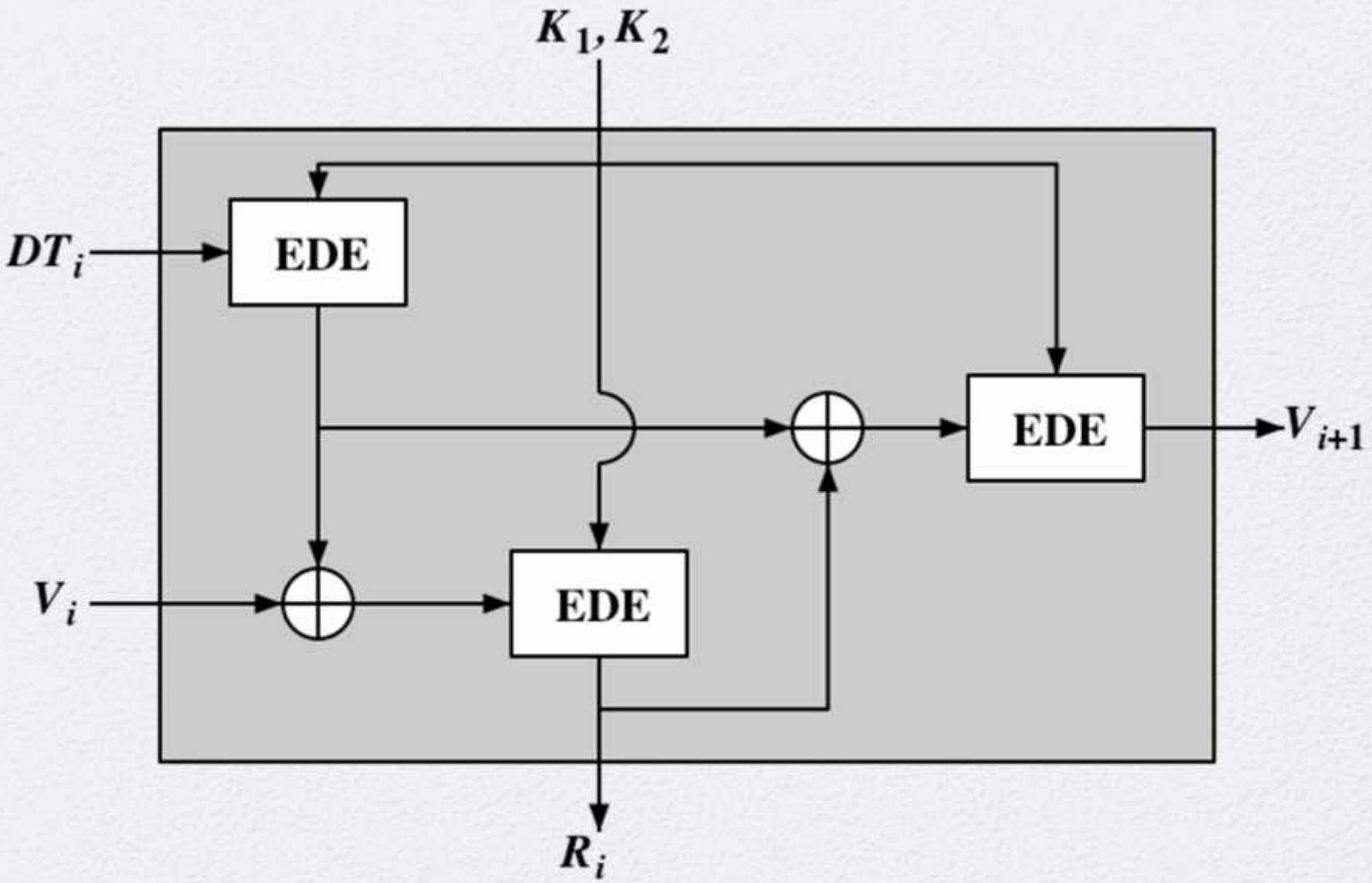


Figure 7.5 ANSI X9.17 Pseudorandom Number Generator

ANSI PRNG

- **Input:** Two pseudorandom inputs drive the generator. One is a 64-bit representation of the current date and time, which is updated on each number generation. The other is a 64-bit seed value; this is initialized to some arbitrary value and is updated during the generation process.
- **Keys:** The generator makes use of three triple DES encryption modules. All three make use of the same pair of 56-bit keys, which must be kept secret and are used only for pseudorandom number generation.

PRNG

- **Output:** The output consists of a 64-bit pseudorandom number and a 64-bit seed value.

Let us define the following quantities.

DT_i	Date/time value at the beginning of ith generation stage
V_i	Seed value at the beginning of ith generation stage
R_i	Pseudorandom number produced by the ith generation stage
K_1, K_2	DES keys used for each stage

Then

$$R_i = \text{EDE}([K_1, K_2], [V_i \oplus \text{EDE}([K_1, K_2], DT_i)])$$
$$V_{i+1} = \text{EDE}([K_1, K_2], [R_i \oplus \text{EDE}([K_1, K_2], DT_i)])$$

Stream Cipher

A typical stream cipher encrypts plaintext one byte at a time, although a stream cipher may be designed to operate on one bit at a time or on units larger than a byte at a time. Figure 8.7 is a representative diagram of stream cipher structure. In this structure, a key is input to a pseudorandom bit generator that produces a stream of 8-bit numbers that are apparently random. The output of the generator, called a **keystream**, is combined one byte at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation. For example, if the next byte generated by the generator is 01101100 and the next plaintext byte is 11001100, then the resulting ciphertext byte is

```
  11001100 plaintext
⊕ 01101100 key stream
  10100000 ciphertext
```

Decryption requires the use of the same pseudorandom sequence:

```
  10100000 ciphertext
⊕ 01101100 key stream
  11001100 plaintext
```

Stream Ciphers

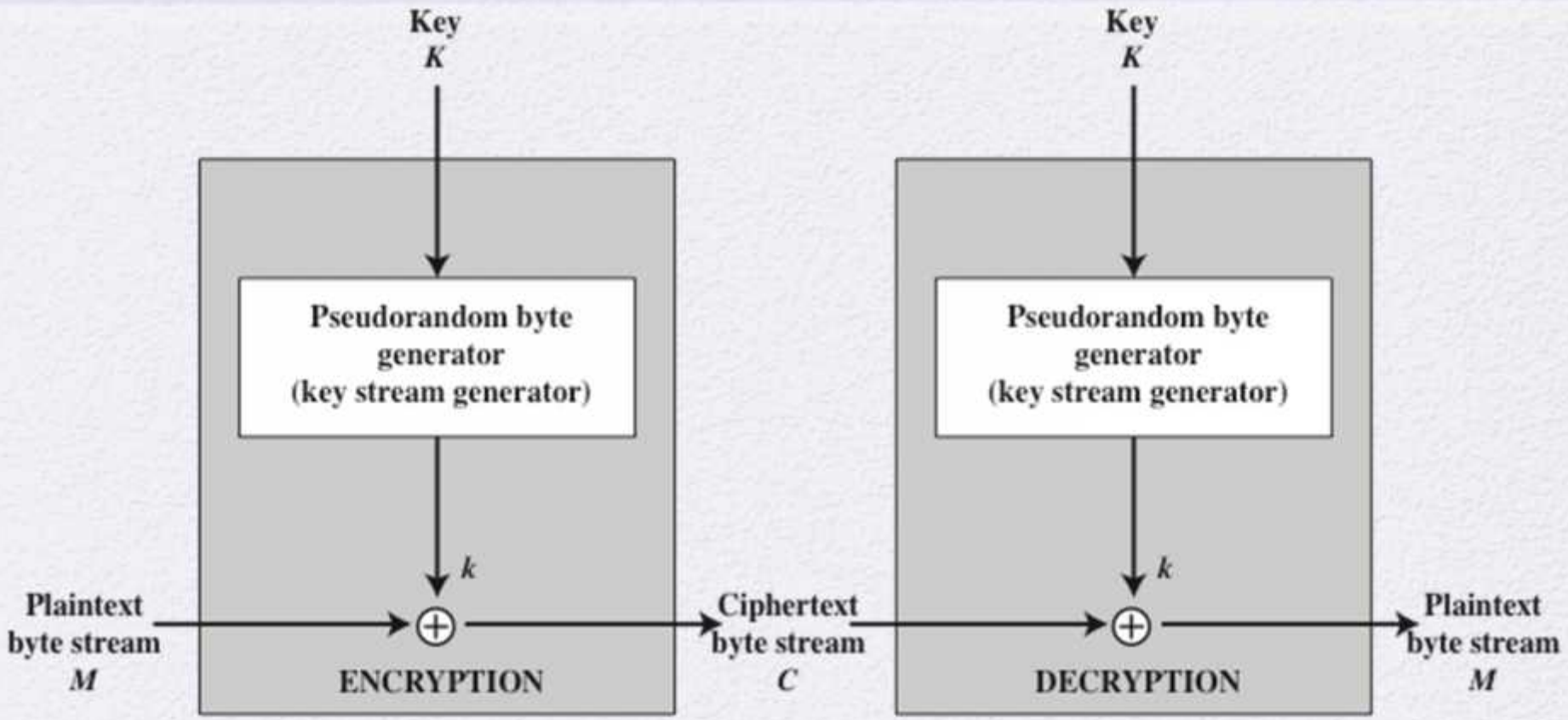


Figure 7.7 Stream Cipher Diagram

Stream Cipher Design Considerations

The encryption sequence should have a large period	• A pseudorandom number generator uses a function that produces a deterministic stream of bits that eventually repeats; the longer the period of repeat the more difficult it will be to do cryptanalysis
The keystream should approximate the properties of a true random number stream as close as possible	• There should be an approximately equal number of 1s and 0s • If the keystream is treated as a stream of bytes, then all of the 256 possible byte values should appear approximately equally often
A key length of at least 128 bits is desirable	• The output of the pseudorandom number generator is conditioned on the value of the input key • The same considerations that apply to block ciphers are valid
With a properly designed pseudorandom number generator a stream cipher can be as secure as a block cipher of comparable key length	• A potential advantage is that stream ciphers that do not use block ciphers as a building block are typically faster and use far less code than block ciphers

Key generation

- When describing algorithms, we assume access to uniformly distributed bits/bytes
- Where do these actually come from?

- *Random-number generation*

Random-number generation

- Precise details depend on the system
  - Linux or unix: /dev/random or /dev/urandom
  - **Do not use rand() or java.util.Random**
  - Use crypto libraries instead

Random-number generation

- Two steps:
  1. Continually collect a “pool” of high-entropy (i.e., “unpredictable”) data
  2. When random bits are requested, process this data to generate a sequence of uniform, independent bits/bytes
     - May “block” if insufficient entropy available

Step 1

- Collect a “pool” of high-entropy data
- Must ultimately come from some physical process (since computation is deterministic)
  - External inputs
    - Keystroke/mouse movements
    - Delays between network events
    - Hard-disk access times
    - Other external sources
  - Hardware random-number generation (e.g., Intel)

Min-entropy

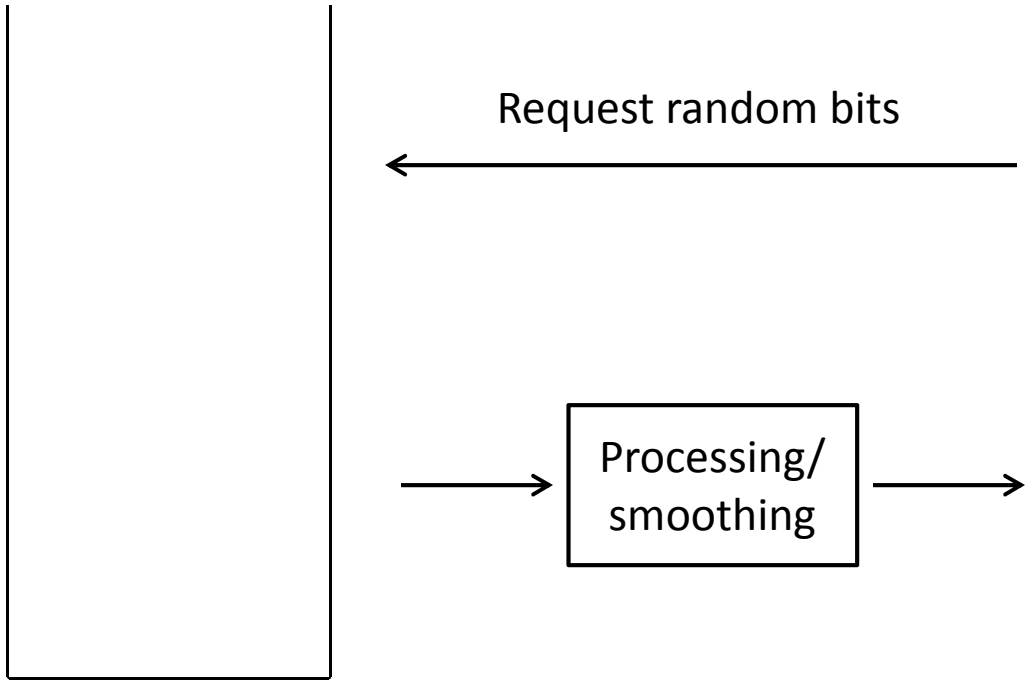
- I.e., "guessing entropy"
- The *min-entropy* of a random variable X is defined as

$$H_\infty(X) = -\log_2 \max_x \{ \Pr[X=x] \}$$

(in bits)

- If X ranges over n-bit strings, then $H_\infty(X) \leq n$
  - Equality iff X has uniform distribution

Random-number generation



Step 2: Smoothing

- Need to eliminate both *bias* and *dependencies*

- von Neumann technique for eliminating bias:
  - Collect two bits per output bit
    - 01 -> 0
    - 10 -> 1
    - 00, 11 -> skip
  - Note that this assumes *independence* (as well as constant bias)

Smoothing

- Can use *randomness extraction*
- Unkeyed extraction is possible for some input distributions; impossible for others
- Keyed extraction possible for all distributions
  - Extracted randomness is less than the input min-entropy
  - Where does the key come from?
- In practice, computational extraction is used

Key generation

- Read desired number of bytes from /dev/urandom
- See code

Encryption

- Plaintext = sequence of ASCII characters
- Key = sequence of hex digits, written in ASCII

- Read them; XOR them to get the ciphertext

Decryption

- Reverse encryption
- Read ciphertext and key; XOR them to recover the message