Chapter 7

Pseudorandom Number Generation and Stream Ciphers

Dr. Shin-Ming Cheng

NTUST
CONNECTIVITY LAB

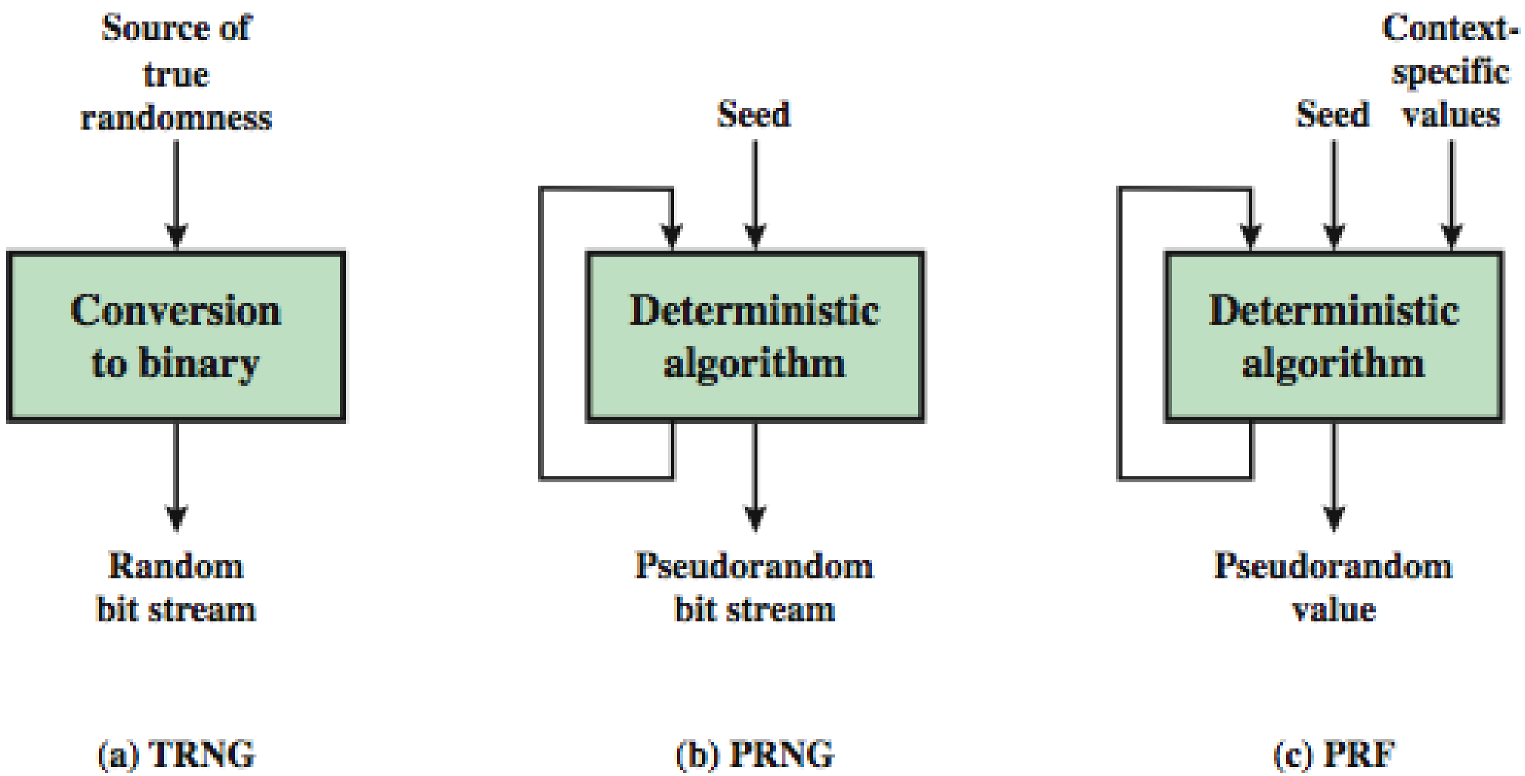
Random Numbers

- Usages in cryptography
  - nonces in authentication protocols to prevent replay
  - session keys
  - public key generation
  - keystream for a one-time pad
- Its critical that these values be
  - statistically random, uniform distribution, independent
  - unpredictability of future values from previous values
- True random numbers
- care needed with generated random numbers

Pseudorandom Number Generators (PRNGs)

- use deterministic algorithmic techniques to create random numbers
  - are not truly random!
  - can pass many tests of “randomness”
- known as pseudorandom numbers
- created by Pseudorandom Number Generators (PRNGs)

Random & Pseudorandom Number Generators



PRNG Requirements

- randomness
  - uniformity, scalability, consistency
- unpredictability
  - forward and backward unpredictability
  - use same tests to check
- characteristics of the seed
  - secure
  - if known adversary can determine output
  - so must be random or pseudorandom number

Linear Congruential Generator (1/3)

- common iterative technique using:

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

- given suitable values of parameters can produce a long random-like sequence
- $a = c = 1$
- $a = 7,\ c = 0,\ m = 32$, and $X_0 = 1$
  - Sequence $\{7, 17, 23, 1, 7, \text{etc.}\}$ only has 32 possible values and a period of 4
- $a = 5,\ c = 0,\ m = 32$, and $X_0 = 1$
  - Sequence $\{5, 25, 29, 17, 21, 9,13,1,5, \text{etc.}\}$ has the period of 8

Linear Congruential Generator (2/3)

- m should be very large
  - Nearly equal to the maximum representable nonnegative integer for a given computer, 2^{31}
- suitable criteria to have are:
  - function generates a full-period
    - All the numbers from 0 through $m-1$ before repeating
  - generated sequence should appear random
  - efficient implementation with 32-bit arithmetic
- If m is prime and $c=0$, then for certain values of a the period of the generating function is $m-1$, with only the value 0 missing.

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

Linear Congruential Generator (3/3)

- IBM 360 family of computers $a = 7^5 = 15807$
- Once the initial value X_0 is chosen, the remaining numbers in the sequence follow deterministically.
- An attacker can reconstruct sequence given a small number of values

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

  - With X_0, X_1, X_2, and X_3, a, c, and m can be solved.
- Includes clock value for making this harder

Blum Blum Shub Generator (1/2)

- Choose two large prime numbers, p and q such that

$$p \equiv q \equiv 3 (\bmod 4)$$

- Choose a random number s, such that s is relative prime to $n = p \times q$
  - Neither p nor q is a factor of s.
- use least significant bit from iterative equation:

$$x_0 = s^2 \bmod n$$
$$x_i = x_{i-1}^2 \bmod n$$
$$b_i = x_i \bmod n$$

Blum Blum Shub Generator (2/2)

- unpredictable, passes **next-bit** test
  - Given the first k bits of the sequence, there is not a practical algorithm that can even allow you to state that the next bit will be 1
- Security rests on difficulty of factoring n
  - Given n, we need to determine its two prime factors p and q
- Slow
  - very large numbers must be used
  - too slow for cipher use, good for key generation

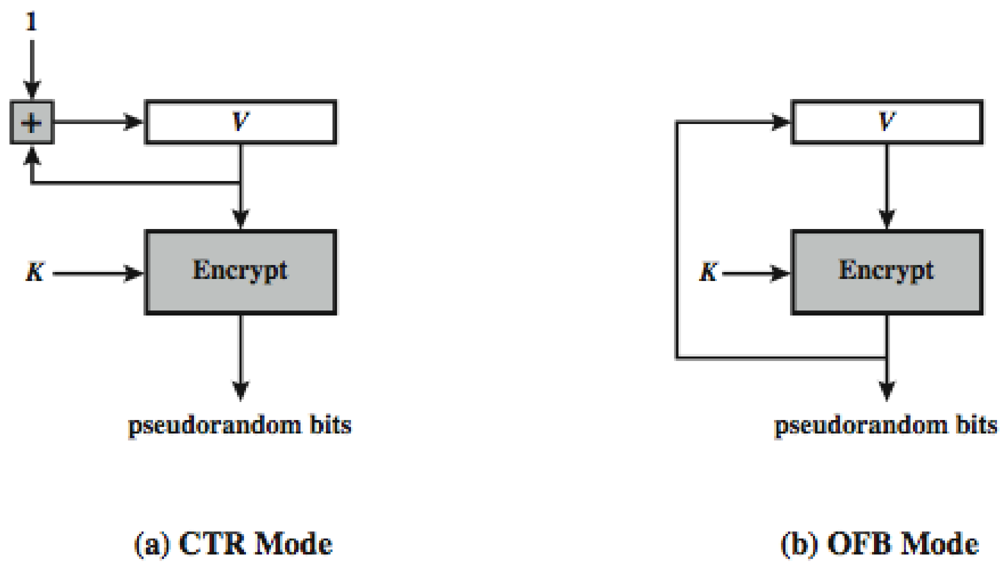
Using Block Ciphers as PRNGs

- For any block of plaintext, a symmetric block cipher produces an output block that is apparently random
- Often for creating session keys from master key
- CTR

$$X_i = E_K[V_i]$$

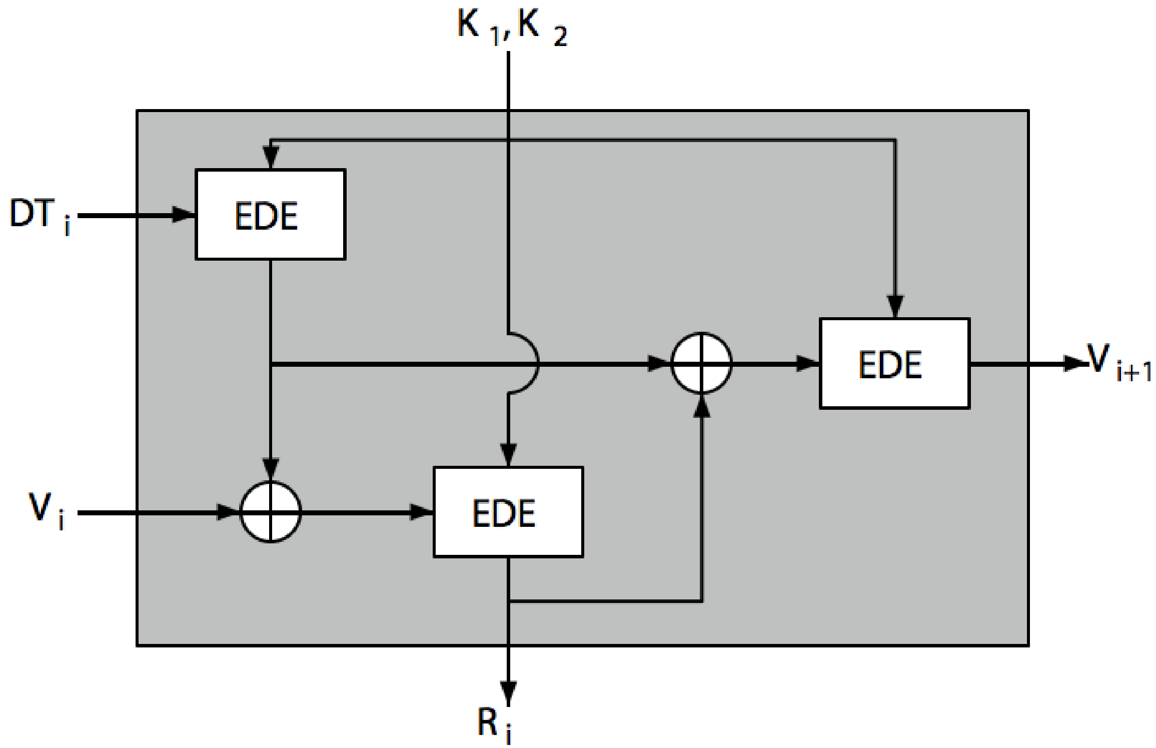
- OFB

$$X_i = E_K[X_{i-1}]$$



ANSI X9.17 PRG

› It uses date/time and seed inputs and 3 triple-DES encryptions to generate a new seed and random value

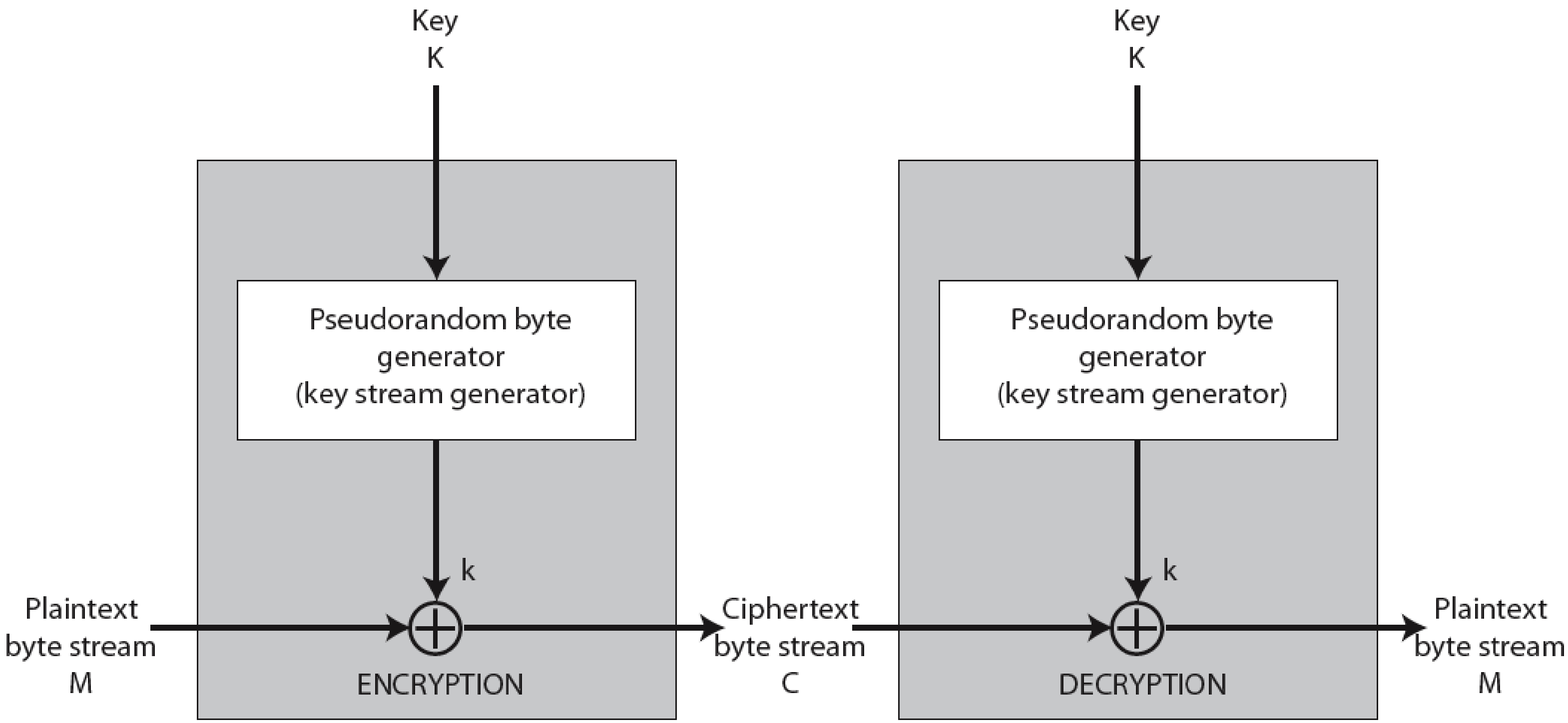


AB

Stream Ciphers

- process message bit by bit (as a stream)
  - have a pseudo random **keystream**
  - combined (XOR) with plaintext bit by bit
- randomness of stream key completely destroys statistically properties in message
  - $C_i = M_i \text{ XOR } StreamKey_i$
- but must never reuse stream key
  - recover messages

Stream Cipher Structure



Stream Cipher Properties

- Design considerations
  - long period with no repetitions
    - the longer the period of repeat the more difficult it will be to do cryptanalysis
  - statistically random
  - depends on large enough key
    - a key length of at least 128 bits is desirable
  - large linear complexity
- Comparing with block cipher
  - has same secure level with same size key
  - Cannot reuse key
  - simpler and faster

RC4

- Ron Rivest
- widely used
  - web SSL/TLS
  - wireless WEP/WPA
- variable key size (from 1 to 256 byes), byte-oriented stream cipher
- key forms random permutation of all 8-bit values
- uses that permutation to scramble input info processed a byte at a time

RC4 Key Schedule (1/2)

- starts with an array S of numbers: $0, 1, \ldots, 255$
  - S contains a permutation of all 8-bit numbers from through 0 through 255
  - If the length of the key K is 256 bytes, then K is transferred to T.
  - Otherwise, for a key of length *keylen* bytes, the first *keylen* elements of T are copied from K and then K is repeated as many times as necessary to fill out T.

```
for i = 0 to 255 do
   S[i] = i;
   T[i] = K[i mod keylen];
```

RC4 Key Schedule (2/2)

- Use T to produce the initial permutation of S
  - For each S[i], swapping S[i] with another byte in S according to a scheme dictated by T[i]

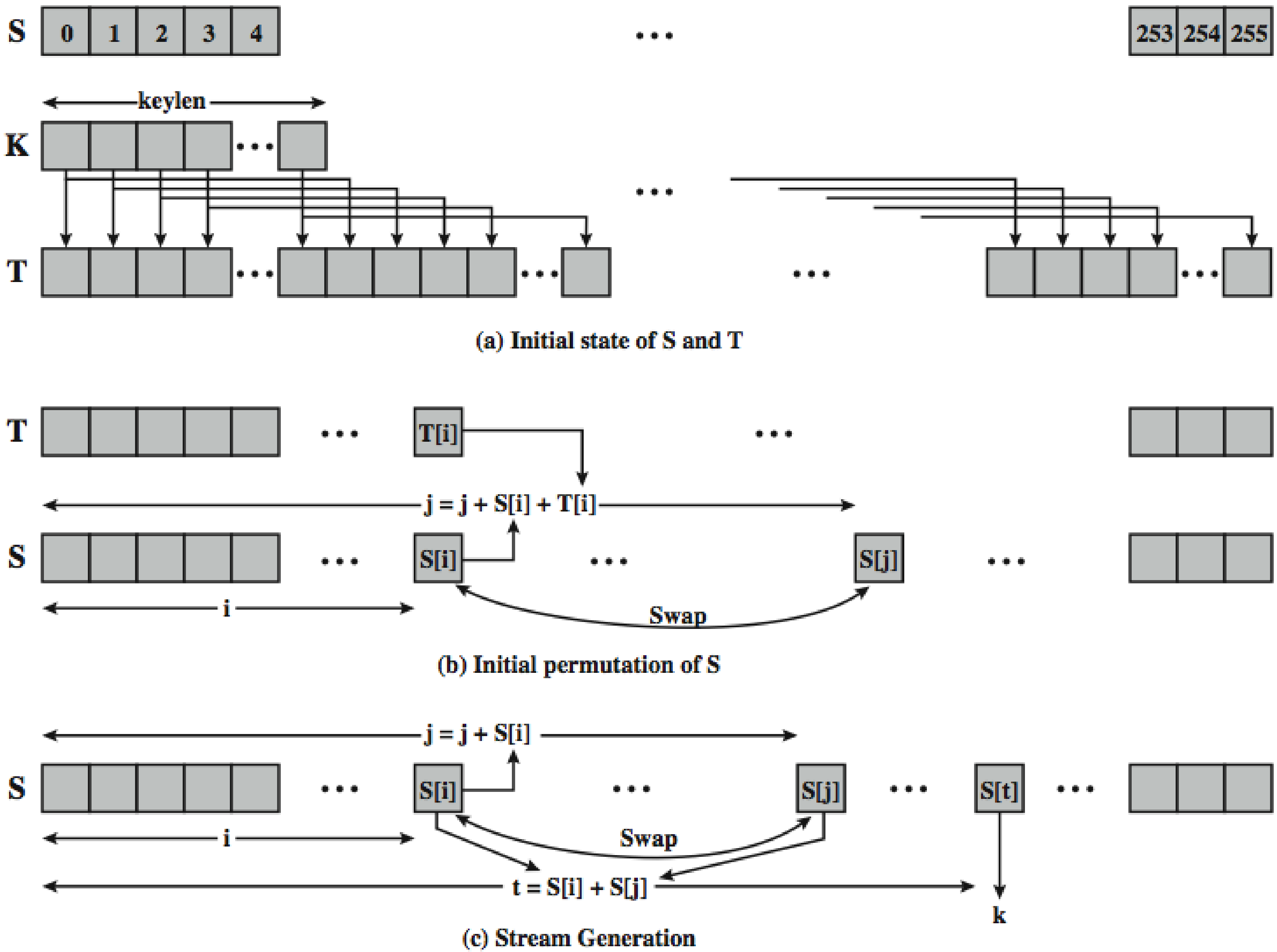
```
j = 0
for i = 0 to 255 do
    j = (j + S[i] + T[i]) (mod 256)
    swap (S[i], S[j])
```

RC4 Encryption

- encryption continues shuffling array values
- sum of shuffled pair selects "stream key" value from permutation
- XOR S[t] with next byte of message to en/decrypt

```
i = j = 0
for each message byte Mi
   i = (i + 1) (mod 256)
   j = (j + S[i]) (mod 256)
   swap(S[i], S[j])
   t = (S[i] + S[j]) (mod 256)
   Ci = Mi XOR S[t]
```

RC4 Logic



S
0
1
2
3
4
253
254
255
keylen
K
T
(a) Initial state of S and T
T
T[i]
j = j + S[i] + T[i]
S
S[i]
S[j]
i
Swap
(b) Initial permutation of S
j = j + S[i]
S
S[i]
S[j]
S[t]
i
Swap
t = S[i] + S[j]
k
(c) Stream Generation

RC4 Security

- Claimed secure against known attacks
  - have some analyses, none practical
- Result is very non-linear
- since RC4 is a stream cipher, must never reuse a key
- have a concern with WEP, but due to key handling rather than RC4 itself

True Random Number Generator (1/3)

- Uses a nondeterministic source to produce randomness
  - Best source is natural randomness in real world
  - Find a regular but random event and monitor
- Do generally need special h/w to do this
  - radiation counters, radio noise, audio noise, thermal noise in diodes, leaky capacitors, mercury discharge tubes

True Random Number Generator (2/3)

- Problems of bias or uneven distribution in signal
  - have to compensate for this when sample, often by passing bits through a hash function
  - best to only use a few noisiest bits from each sample
  - RFC4086 recommends using multiple sources + hash

True Random Number Generator (3/3)

- Published Collections of random numbers
  - Rand Co, in 1955, published 1 million numbers
    - generated using an electronic roulette wheel
    - has been used in some cipher designs cf Khafre
  - Tippett in 1927 published a collection
- Issues
  - these are limited
  - too well-known for most uses