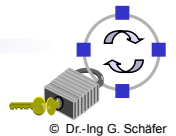


Network Security

Chapter 6

Random Number Generation

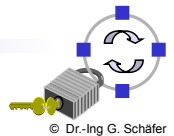


Tasks of Key Management (1)

- ❑ **Generation:**
 - ❑ It is crucial to security, that keys are generated with a truly random or at least a pseudo-random generation process (see below)
 - ❑ Otherwise, an attacker might reproduce the key generation process and easily find the key used to secure a specific communication
- ❑ **Distribution:**
 - ❑ Distribution of some initial keys usually has to be performed manually / out of band
 - ❑ Session key distribution is generally performed during an authentication exchange
 - ❑ Examples: Diffie-Hellman, Otway-Rees, Kerberos, X.509
- ❑ **Storage:**
 - ❑ Keys, especially authentication keys, should be securely stored:
 - either encrypted with a hard-to-guess pass-phrase, or better
 - in a secure device like a smart-card



- ❑ **Revocation:**
 - ❑ If a key has been compromised, it should be possible to revoke that key, so that it can no longer be misused (cf. X.509)
- ❑ **Destruction:**
 - ❑ Keys that are no longer used (e.g. old session keys) should be safely destroyed (cf. media security in lecture 1)
- ❑ **Recovery:**
 - ❑ If a key has been lost (e.g. defect smart-card, floppy, accidentally erased) it should be possible to recover it, in order to avoid loss of data
 - ❑ Key recovery is not to be mixed up with key escrow (see below):
- ❑ **Escrow:**
 - ❑ Mechanisms and architectures that shall allow government agencies (and only them) to obtain session keys in order to be able to eavesdrop on communications / to read stored data for law enforcement purposes
 - “If I can get my key back it’s key recovery,
if you can get my key back it’s key escrow...” :o)

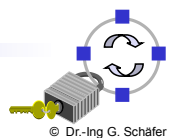


- ❑ **Definition:**

A *random bit generator* is a device or algorithm, which outputs a sequence of statistically independent and unbiased binary digits.
- ❑ **Remark:**
 - ❑ A random bit generator can be used to generate uniformly distributed random numbers, e.g. a random integer in the interval $[0, n]$ can be obtained by generating a random bit sequence of length $\lfloor \lg n \rfloor + 1$ and converting it into a number. If the resulting integer exceeds n it can be discarded and the process is repeated until an integer in the desired range has been generated.
- ❑ **Definition:**

A *pseudo-random bit generator (PRBG)* is a deterministic algorithm which, given a truly random binary sequence of length k , outputs a binary sequence of length $m \gg k$ which “appears” to be random.

The input to the PRBG is called the *seed* and the output is called a *pseudo-random bit sequence*.



□ Remarks:

- The output of a PRBG is not random, in fact the number of possible output sequences of length m is at most all small fraction $2^k / 2^m$, as the PRBG produces always the same output sequence for one (fixed) seed
- The motivation for using a PRBG is that it might be too expensive to produce true random numbers of length m , e.g. by coin flipping, so just a smaller amount of random bits is produced and then a pseudo-random bit sequence is produced out of the k truly random bits
- In order to gain confidence in the “randomness” of a pseudo-random sequence, statistical tests are conducted on the produced sequences

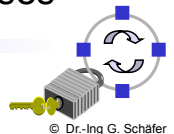
□ Example:

- A linear congruential generator produces a pseudo-random sequence of numbers y_1, y_2, \dots . According to the linear recurrence

$$y_i = a \times y_{i-1} + b \text{ mod } q$$

with a, b, q being parameters characterizing the PRBG

- Unfortunately, this generator is predictable even when a, b and q are unknown, and should, therefore, not be used for cryptographic purposes



□ Security requirements of PRBGs for use in cryptography:

- As a minimum security requirement the length k of the seed to a PRBG should be large enough to make brute-force search over all seeds infeasible for an attacker
- The output of a PRBG should be statistically indistinguishable from truly random sequences
- The output bits should be unpredictable for an attacker with limited resources, if he does not know the seed

□ Definition:

A PRBG is said to *pass all polynomial-time statistical tests*, if no deterministic polynomial-time algorithm can distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater than 0.5

- *Polynomial-time algorithm* means, that the running time of the algorithm is bound by a polynomial in the length m of the sequence



❑ Definition:

A PRBG is said to *pass the next-bit test*, if there is no deterministic polynomial-time algorithm which, on input of the first m bits of an output sequence s , can predict the $(m + 1)^{\text{st}}$ bit s_{m+1} of the output sequence with probability significantly greater than 0.5

❑ Theorem (universality of the next-bit test):

A PRBG passes the next-bit test

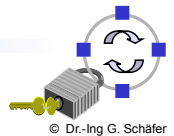


it passes all polynomial-time statistical tests

❑ For the proof, please see section 12.2 in [Sti95a]

❑ Definition:

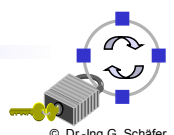
A PRBG that passes the next-bit test – possibly under some plausible but unproved mathematical assumption such as the intractability of the factoring problem for large integers – is called a *cryptographically secure pseudo-random bit generator (CSPRBG)*



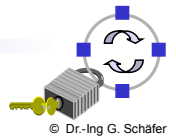
❑ Hardware-based random bit generators are based on physical phenomena, as:

- ❑ elapsed time between emission of particles during radioactive decay,
- ❑ thermal noise from a semiconductor diode or resistor,
- ❑ frequency instability of a free running oscillator,
- ❑ the amount a metal insulator semiconductor capacitor is charged during a fixed period of time,
- ❑ air turbulence within a sealed disk drive which causes random fluctuations in disk drive sector read latencies, and
- ❑ sound from a microphone or video input from a camera
- ❑ the state of an odd number of circular connected NOT gates

❑ A hardware-based random bit generator should ideally be enclosed in some tamper-resistant device and thus shielded from possible attackers



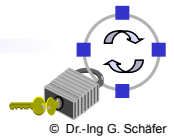
- ❑ Software-based random bit generators, may be based upon processes as:
 - ❑ the system clock,
 - ❑ elapsed time between keystrokes or mouse movement,
 - ❑ content of input- / output buffers
 - ❑ user input, and
 - ❑ operating system values such as system load and network statistics
- ❑ Ideally, multiple sources of randomness should be “mixed”, e.g. by concatenating their values and computing a cryptographic hash value for the combined value, in order to avoid that an attacker might guess the random value
 - ❑ If, for example, only the system clock is used as a random source, than an attacker might guess random-numbers obtained from that source of randomness if he knows about when they were generated



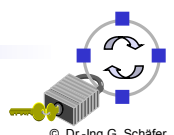
- ❑ *De-skewing:*
 - ❑ Consider a random generator that produces biased but uncorrelated bits, e.g. it produces 1's with probability $p \neq 0.5$ and 0's with probability $1 - p$, where p is unknown but fixed
 - ❑ The following technique can be used to obtain a random sequence that is uncorrelated and unbiased:
 - The output sequence of the generator is grouped into pairs of bits
 - All pairs 00 and 11 are discarded
 - For each pair 10 the unbiased generator produces a 1 and for each pair 01 it produces a 0
 - ❑ Another practical (although not provable) de-skewing technique is to pass sequences whose bits are correlated or biased through a cryptographic hash function such as MD5 or SHA-1



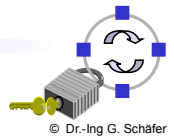
- ❑ The following tests allow to check, if a generated random or pseudo-random sequence inhibits certain statistical properties:
 - ❑ *Monobit Test*: Are there equally many 1's like 0's?
 - ❑ *Serial Test (Two-Bit Test)*: Are there equally many 00-, 01-, 10-, 11-pairs?
 - ❑ *Poker Test*: Are there equally many sequences n_i of length q having the same value with q such that $\lfloor m / q \rfloor \geq 5 \times (2^q)$
 - ❑ *Runs Test*: Are the numbers of *runs* (sequences containing only either 0's or 1's) of various lengths as expected for random numbers?
 - ❑ *Autocorrelation Test*: Are there correlations between the sequence and (non-cyclic) shifted versions of it?
 - ❑ *Maurer's Universal Test*: Can the sequence be compressed?
 - ❑ *NIST SP 800-22*: Standardized test suite, includes above & more advanced tests
- ❑ The above descriptions just give the basic ideas of the tests. For a more detailed and mathematical treatment, please refer to sections 5.4.4 and 5.4.5 in [Men97a]



- ❑ There are a number of algorithms, that use cryptographic hash functions or encryption algorithms for generation of cryptographically secure pseudo random numbers
 - ❑ Although these schemes can not be proven to be secure, they seem sufficient for most practical situations
- ❑ One such approach is the ANSI X9.17 generator:
 - ❑ Input: a random and secret 64-bit seed s , integer m , and 3-DES key K
 - ❑ Output: m pseudo-random 64-bit strings y_1, y_2, \dots, Y_m
 - 1.) $q = E(K, \text{Date_Time})$
 - 2.) For i from 1 to m do
 - 2.1) $x_i = E(K, (q \oplus s))$
 - 2.2) $s = E(K, (x_i \oplus q))$
 - 3.) Return(x_1, x_2, \dots, x_m)
 - ❑ This method is a U.S. Federal Information Processing Standard (FIPS) approved method for pseudo-randomly generating keys and initialization vectors for use with DES



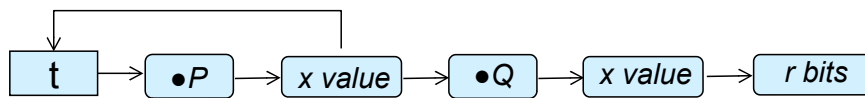
- The RSA-PRBG is a CSPRBG under the assumption that the RSA problem is intractable:
 - Output: a pseudo-random bit sequence z_1, z_2, \dots, z_k of length k
 - 1.) Setup procedure:
 - Generate two secret primes p, q suitable for use with RSA
 - Compute $n = p \times q$ and $\Phi = (p - 1) \times (q - 1)$
 - Select a random integer e such that $1 < e < \Phi$ and $\text{gcd}(e, \Phi) = 1$
 - 2.) Select a random integer y_0 (the seed) such that $y_0 \in [1, n]$
 - 3.) For i from 1 to k do
 - 3.1) $y_i = (y_{i-1})^e \text{ mod } n$
 - 3.2) $z_i =$ the least significant bit of y_i
 - The efficiency of the generator can be slightly improved by taking the last j bits of every y_i , with $j = c \times \lg(\lg(n))$ and c is a constant
 - However, for a given bit-length m of n , a range of values for the constant c such that the algorithm still yields a CSPRBG has not yet been determined



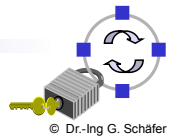
- The Blum-Blum-Shub-PRBG is a CSPRBG under the assumption that the integer factorization problem is intractable:
 - Output: a pseudo-random bit sequence z_1, z_2, \dots, z_k of length k
 - 1.) Setup procedure:
 - Generate two large secret and distinct primes p, q such that p, q are each congruent 3 modulo 4 and let $n = p \times q$
 - 2.) Select a random integer s (the seed) such that $s \in [1, n - 1]$ such that $\text{gcd}(s, n) = 1$ and let $y_0 = s^2 \text{ mod } n$
 - 3.) For i from 1 to k do
 - 3.1) $y_i = (y_{i-1})^2 \text{ mod } n$
 - 3.2) $z_i =$ the least significant bit of y_i
 - The efficiency of the generator can be improved using the same method as for the RSA generator with similar constraints on the constant c



- ❑ Dual Elliptic Curve Deterministic Random Bit Generator:
 - ❑ Based on the intractability of the elliptic curve discrete logarithm problem
 - ❑ Simplified version:



- ❑ State t is multiplied with a generator P , the x -value of the new point becomes t'
- ❑ Multiplied with a different point Q r bits of output can be generated, number of bits depend on curve (ranging between 240 and 504 bits)
- ❑ Part of NIST 800-90A standard
- ❑ Security:
 - It has been shown that if P is chosen to be eQ for a constant e then attackers can derive the state t
 - We do not know how the predefined points P and Q in NIST 800-90A are derived, so be careful 😊



CSPRNG security is a big thing!

- ❑ In September 2006 Debian was accidentally modified that only the process ID was used to feed the OpenSSL CSPRNG
 - ❑ Only 32,768 possible values!
 - ❑ Was not discovered until May 2008
- ❑ A scan of about 23 million TLS and SSH hosts showed that
 - ❑ At least 0.34% of the hosts shared keys because of faulty RNGs
 - ❑ 0.50% of the scanned TLS could be compromised because of low randomness
 - ❑ and 1.06% of the SSH hosts...
- ❑ Supervise your CSPRNG!
 - ❑ Do not generate random numbers right after booting your system
 - ❑ Use blocking RNGs, i.e. those that do not continue until having enough entropy

