Random number generation: an introduction

Introduction to random numbers

The generation of random numbers is too important to be left to chance

Robert Coveyou, Oak Ridge National Laboratory

Introduction to random numbers

- we have seen that in many applications we need random number generators
- for example, we use random number generator to obtain session keys; to avoid key guessing
- Given an adversary with knowledge of a sequence of past session keys, the probability of correctly predicting the next session key should be negligible

note: good random number generator for cryptography is expensive

Generators of randomness

Two principal methods

1. Measure a physical phenomenon that is expected to be random, and compensate for possible biases in the measurement process (random number generator, RNG)
2. Use algorithms that produce long sequences of apparently random values, which are fully determined by an initial value (the seed or key)
(pseudorandom number generator, PRNG)

Some clarifications

Note that

a) deterministic computations cannot give true random numbers because they are inherently predictable
b) distinguishing a "true" random number from the output of a pseudo-random number generator is a very difficult problem
c) carefully chosen pseudo-random number generators can be used instead of true random numbers in many applications
d) rigorous statistical analysis of the output is often needed to have confidence in the algorithm

Generation

- use of systems data
- use of external information
- above data are used to initialize a pseudorandom number generator, a program that:
  - given an initial seed (random)
  - obtains a long sequence of numbers

- if an adversary sees a long sequence of numbers in output it should be computationally very expensive to guess next output number (even in a probabilistic sense)

Initial seed

different sources

- machine /network
  - clock
  - free space on disk
  - number of files on disk
  - info on operating system (I/O queues, buffer state etc.)
  - user information (e.g., windows and size of windows)
  - inter-arrival time of packets
  - ...
- user
  - keyboard & mouse timing

Initial seed: how many bits of randomness?

- in many cases, sources provide few bits of randomness (in fact much information can be easily guessed)
    - clock: we can use only low order digit (e.g., milliseconds: 10 bits of randomness)
    - day, hour and minute can be easily guessed and are not random
- it is advantageous to mix several sources of randomness using crypto functions

common mistakes

- using small initial seed
    - ex.: 16 bits seed give only 65536 different seeds, and attacker can try them all
- using current time
    - if clock granularity is 10 msec then if we approximately know the time (just the hour) there are 60 (minutes) × 60 (seconds) × 100 (hundredths of a second) =
      = 360000 different choices only
- divulging seed value
    - according to Kaufman, Perlman & Speciner, "an implementation" used the time of day to choose a per-message encryption key; the time of day in this case may have had sufficient granularity, but the problem was that the application included the time of day in the unencrypted header of the message

Given this source code...

```
/* Netscape 1.1  seeding & key
    generation (1996) */

global variable seed;


RNG_CreateContext()
 (seconds, microseconds) = time
    of day;
 /* Time elapsed since 1970 */
 pid = process ID; ppid = parent
    process ID;
 a = mklcpr(microseconds);
 b = mklcpr(pid + seconds + (ppid
    << 12));
 seed = MD5(a, b);
```

```
/* not cryptographically significant
    */
mklcpr(x)
 return ((0xDEECE66D * x +
    0x2BBB62DC) >> 1);


RNG_GenerateRandomBytes()
 x = MD5(seed);
 seed = seed + 1;
 return x;
```

MD5 broken since 1996

Assessing randomness in Netscape 1.1

- if attacker does not have an account on the attacked UNIX machine pid and ppid are unknown
- even though the pid and ppid are 15-bit quantities on most UNIX machines, the sum pid + (ppid << 12) has only 27 bits, not 30
- if the value of seconds is known, a has only 20 unknown bits, and b has only 27 unknown bits. This leaves, at most, 47 bits of randomness in the secret key
- in addition, ppid & pid can be hacked
    - ppid is often 1 (for example, when the user starts Netscape from an X Window system menu); if not, it is usually just a bit smaller than the pid
    - mail-transport agent sendmail generates Message-IDs for outgoing mail using its process ID; as a result, sending e-mail to an invalid user on the attacked machine will cause the message to bounce back to the sender; the Message-ID contained in the reply message will tell the sender the last process ID used on that machine. If the user started Netscape relatively recently, and that the machine is not heavily loaded, this will closely approximate Netscape's pid

Random number bug in Debian Linux (2006)

- announced in 2008, affecting Debian + ∗Ubuntu
  http://www.debian.org/security/2008/dsa-1571
- vulnerability in OpenSSL, caused by the removal of the following line of code from **md_rand.c**

  MD_Update(&m,buf,j); /* purify complains */

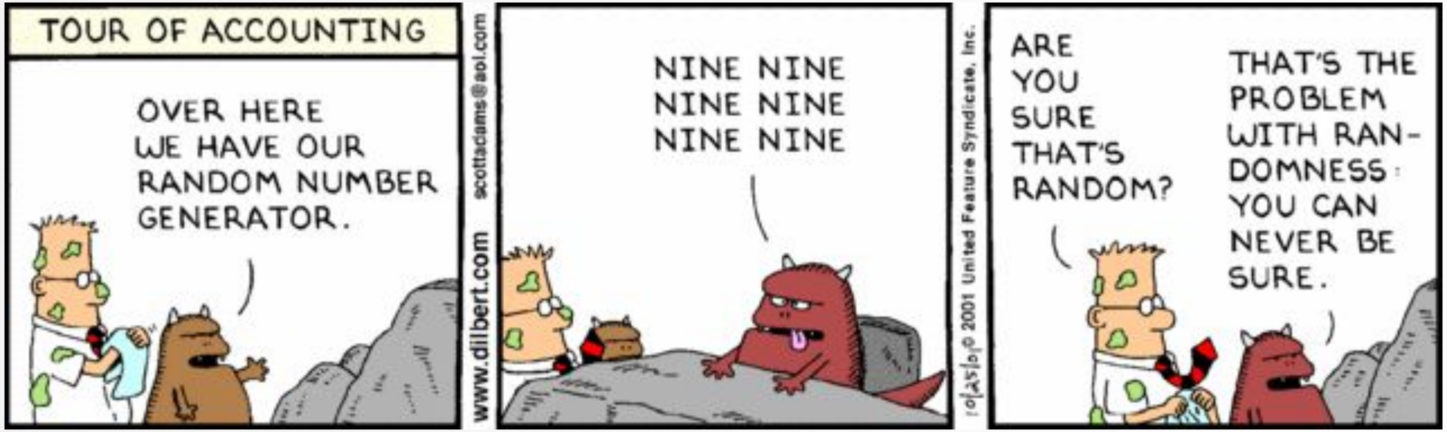
- line removed because it caused Valgrind (OS debugging tool) and Purify (a proprietary debugger, from IBM) to warn about use of uninitialized data in any code linked to OpenSSL
- instead of mixing in random data for the initial seed, the only used "random" value was the current process ID
  - on Linux platform, default maximum process ID is 32768, resulting in a very small number of seed values being used for all PRNG operations

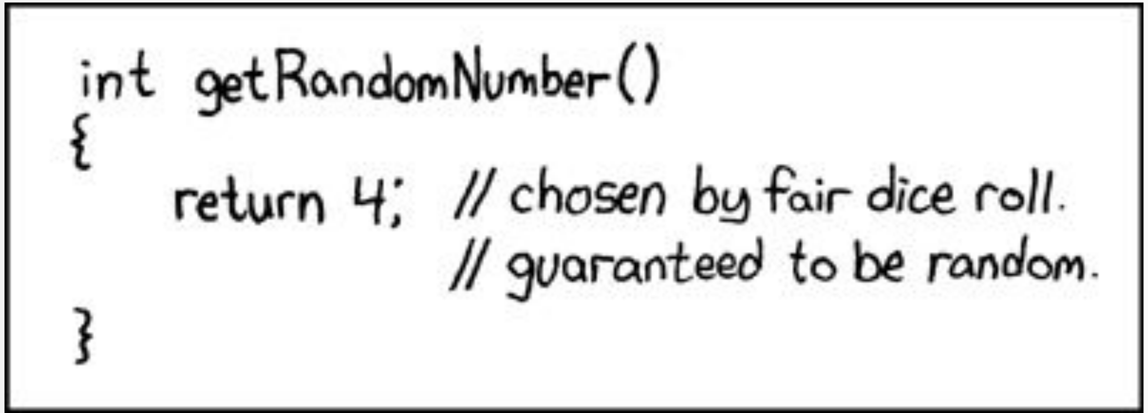
Bruce Schneier's commentary

from Crypto-Gram Newsletter, June 15, 2008

Random numbers are used everywhere in cryptography, for both short- and long-term security. And, as we've seen here, security flaws in random number generators are really easy to accidentally create and really hard to discover after the fact. Back when the NSA was routinely weakening commercial cryptography, their favorite technique was reducing the entropy of the random number generator.

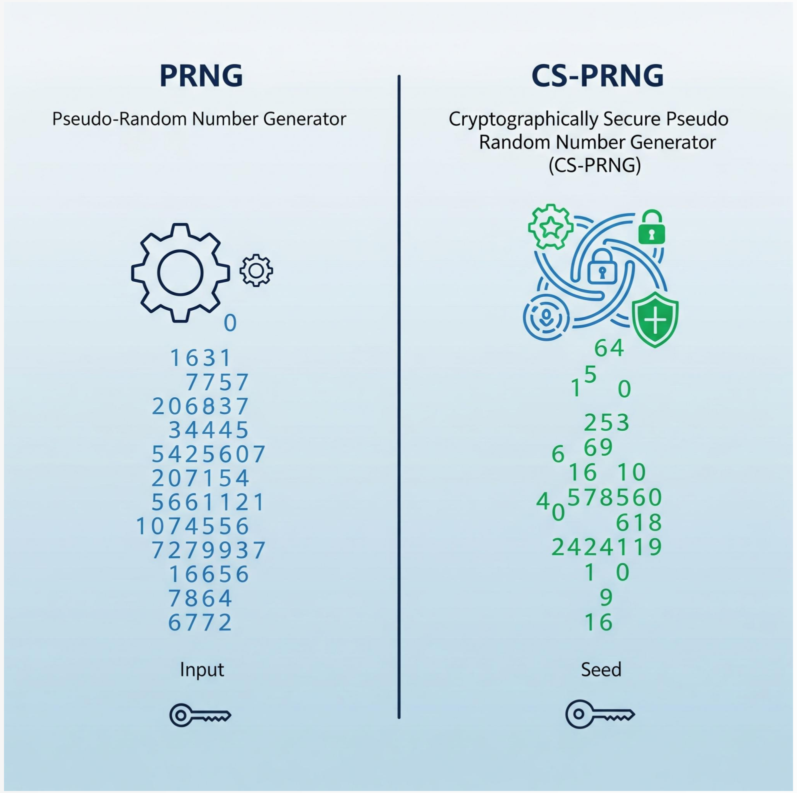
Random generators' folklore


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com scottadams@aol.com
10/25/01 © 2001 United Feature Syndicate, Inc.



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

PRNG & CS-PRNG


PRNG
Pseudo-Random Number Generator
0
1631
7757
206837
34445
5425607
207154
5661121
1074556
7279937
16656
7864
6772
Input
CS-PRNG
Cryptographically Secure Pseudo Random Number Generator (CS-PRNG)
64
1 5 0
253
6 69
16 10
4 0 578560
618
2424119
1 0
9
16
Seed

Introduction to CS-PRNG

- We have already seen that we can practically aim at pseudo-random number generators (PRNG)
- They can have a good implementation but are still cryptographically insecure
    - PRNGs are good for making numbers that look random (statistical property)
    - CS-PRNG are good for making numbers that stay secret and unpredictable

Comparison

Aspect	PRNG	CS-PRNG
Main goal	Simulations, modeling, or games	Sequences that are unpredictable and secure for cryptographic use
Randomness check	Mostly statistical: uniform distribution, independence, low correlation	Statistical and security: must **withstand cryptanalytic attacks**
Predictability	If the algorithm and state are known, all future outputs can be computed	With partial/total information, predicting other outputs must be computationally infeasible
Security properties	None guaranteed beyond statistical tests	**Provides forward secrecy and backward secrecy**
Applications	SW simulations and testing, procedural generation in games	Key generation, nonces, salts, …
Entropy source	Often seeded once, possibly from a fixed or predictable seed	Seeded from high-entropy sources, with periodic reseeding
Testing standards	May pass statistical test suites	Meets cryptographic standards

Challenge

Consider the following generator

```
hash function H
random initial seed $s$
    y = $s$
    for i = 1 to n do
        y = H(y)
        output(y)
```

Is it PRNG or CS-PRNG?

BSI evaluation criteria

- Bundesamt für Sicherheit in der Informationstechnik (Federal Office for Information Security)
- defines four criteria for quality of deterministic random number generators, aiming at CS-PRNG
    - K1 – Low probability of long identical runs
    - K2 – (See next slide.)
    - K3 – From any output subsequence, an attacker cannot deduce past/future outputs or the internal state
    - K4 – From any internal state, an attacker cannot deduce past outputs or past states
- **For cryptographic applications, only generators meeting the K4 standard are acceptable**

K2 class

- generated sequences are indistinguishable from "true random" numbers according to specified statistical tests
- monobit test (equal numbers of ones and zeros)
- poker test (a special instance of the X^2 (chi-square) test)
- runs test (counts the frequency of runs of various lengths)
- longruns test (runs of length 34 or greater in 20 000 bits of the sequence?)
- autocorrelation test
- these requirements are a test of how well a bit sequence
  - has zeros and ones equally often
  - after a sequence of n zeros (or ones), the next bit a one (or zero) with probability one-half
  - and any selected subsequence contains no information about the next element(s) in the sequence

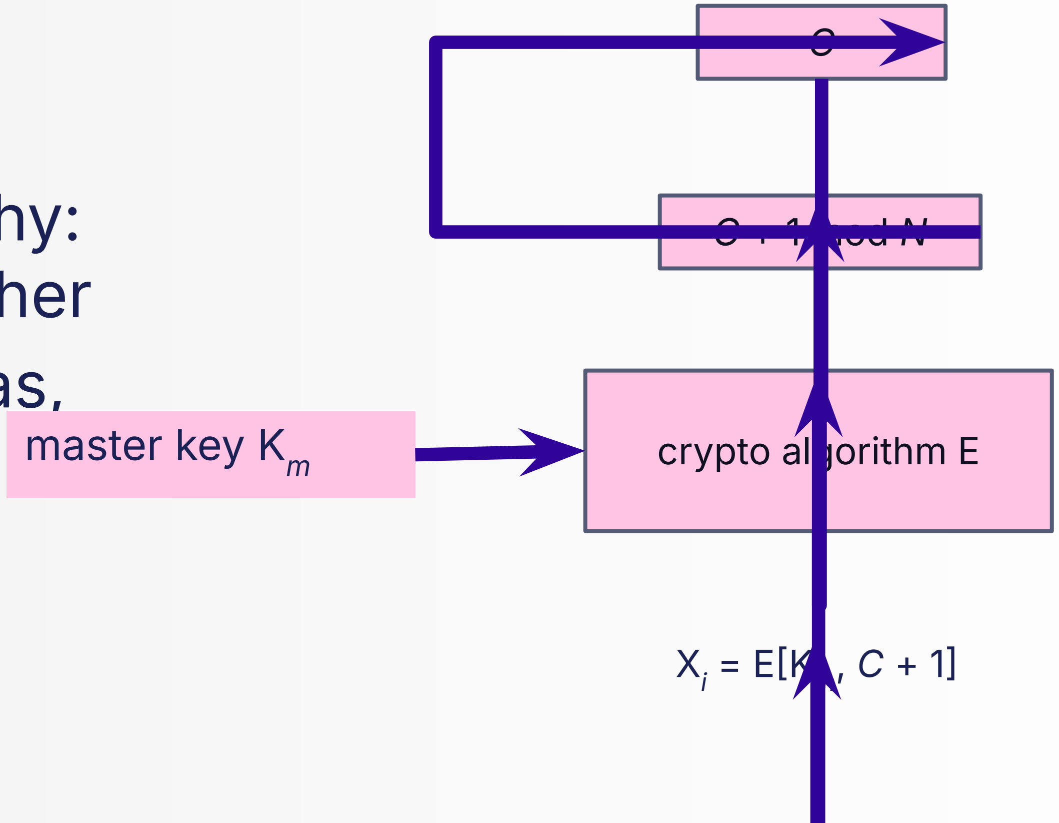
Cryptographically secure pseudorandom number generators (CS-PRGN)

- meet requirements of ordinary PRNG, and
- next-bit test
  - given the first k bits of a random sequence, there is no polynomial-time algorithm that can predict the (k+1)th bit with probability of success better than 50%
- withstand "state compromise extensions"
  - if part or all its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation
  - additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CS-PRNG state

Generator "cipher of a counter"

cyclic cryptography:
use counter + cipher

- Meyer and Matyas, 1982



RSA based generator

- prime numbers p, q
- $n = p \cdot q$
- integer e s.t. $\mathrm{GCD}(e, (p-1)\cdot(q-1)) = 1$
- z = seed
- loop

```
    $z_i = (z_{i-1})^e \bmod n$
    $i = i + 1$
```

- output: least significant bit of z_i

ANSI X9.31 RNG

- Deprecated in 2016
- Block cipher key K (AES or 3DES)
- Seed value V (secret)
- Loop
    - $DT = E_K(\text{time})$
    - $R_i = E_K(DT \oplus V_{i-1})$
    - $V_i = E_K(DT \oplus R_i)$

Output: R_i (or selected bits)

Blum Blum Shub (CS-PRNG)

- choose p, q big prime s.t. $p \equiv q \equiv 3 \pmod 4$
- $n = p \cdot q$
- randomly choose s s.t. $GCD(s, n) = 1$
- output the sequence of bits B_i

```
$X_0 = s^2 \bmod n$
for $i = 1$ to $\infty$
    $X_i = (X_{i-1})^2 \bmod n$
    $B_i = X_i \bmod 2$
    return $B_i$
```

CTR_DRBG (AES)

Maybe the current best (Counter mode Deterministic Random Bit Generator, 2012)

- State: Key K, counter V
- Block cipher: AES-128 / AES-256 in CTR mode
- Generate:
    - $V \leftarrow V + 1 \pmod{2^{\text{block size}}}$
    - out_block = $E_K(V)$
    - Repeat until enough bits produced
- Update: regenerate K and V by encrypting V+1, V+2, ... with current K and using the output blocks to form the new state
- Security: Forward & backward secure (if reseeded periodically)

More on randomness

- subroutines generally provided in programming languages for "random numbers" are not designed to be unguessable, but just designed merely to pass statistical tests
- hash together all the sources of randomness you can find, e.g., timing of keystrokes, disk seek times, count of packet arrivals, etc.
- for more reading about randomness, see RFC 1750