Random and Prime Numbers

Computer and Network Security

Emilio Coppa

Random Number Generator (RNG)

Process which, through a (software or hardware) device, generates a sequence of numbers or symbols that cannot be reasonably predicted better than by a random chance.

Types:

1. True Random Number Generators (TRNG)

2. Pseudorandom Number Generators (PRNG)

3. Cryptographically Secure Pseudorandom Number Generators (CSPRNG)

True Random Number Generators (TRNG)

TRNG is a device that generates random numbers from a physical process, rather than by means of an algorithm. Such devices are often based on microscopic phenomena that generate low-level, statistically random "noise" signals, such as thermal noise, the photoelectric effect, involving a beam splitter, and other quantum phenomena. These stochastic processes are, in theory, completely unpredictable, and the theory's assertions of unpredictability are subject to experimental test.

E.g., coin flipping: if we flip 100 times a coin we can generate a “random” 100-bit number that is very hard to predict. Given bit i-th, there is no way to predict bit (i+1)-th.

Remark. TRNG requires hardware and thus are typically expensive.

Pseudorandom Number Generators (PRNG)

We start from physics and we look for random events

PRNG is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's **seed** (which may include truly random values).

$$s_0 = seed \qquad\qquad s_{i+1} = f(s_i)$$

- cheap compared to TRNG
- given a seed, they are deterministic: good in many application contexts, e.g., simulations
- output should still have good statistical properties, e.g., sequence of true random numbers
- not always suitable for cryptography

Linear Congruential Generator (LCG)

$$s_0 = seed \qquad s_{i+1} = a \cdot s_i + b \bmod m$$

`rand` function from ANSI C:

$$s_0 = 12345 \qquad s_{i+1} = 1103515245 \cdot s_i + 12345 \bmod 2^{31}$$

Remark. LCG are fast and simple. Do not use LCG in cryptography.

Mersenne Twister PRNG

By far the most used PRNG, designed in 1997 by Makoto Matsumoto and Takuji Nishimura. The period of a MT is a Mersenne prime, i.e., a prime number that is one less than a power of two. A common value for the period is $2^{19937}-1$, also called MT19937.

It is based on a "generalized" LFSR, i.e., a "twisted" generalized feedback shift register. The idea behind it is not "easy" to present but it can be implemented very efficiently (XOR, shift, OR, AND):

- pseudocode from Wikipedia
- discussion of the algorithm

It is used in many software/languages. It is not a good for cryptography.

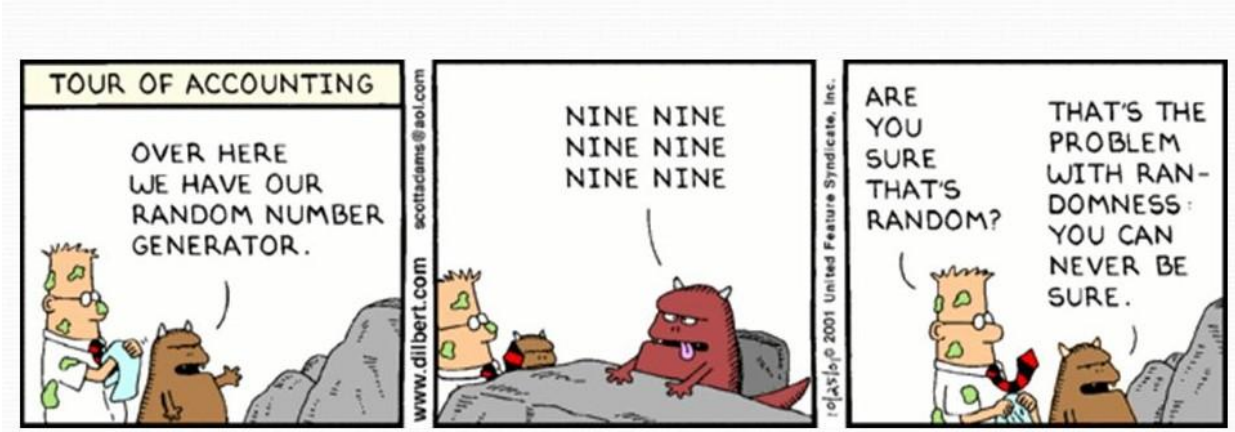
Cryptographically Secure Pseudorandom Number Generators (CSPRNG)

CSPRNG is a pseudorandom number generator (PRNG) with properties that make it suitable for use in cryptography. A CSPRNG is PRNG which is unpredictable: given a stream of n numbers generated by the CSPRNG is computationally infeasible to compute the subsequent numbers.

Implementations based on:

- cryptographic primitives, e.g., hash functions or ciphers
- mathematical problems, e.g., Blum Blum Shub exploits the quadratic residuosity problem
- special designs, e.g., Yarrow (/dev/random in Mac OS / iOS) evaluates quality of the inputs

Random Generator? Got it pretty easy!




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

Netscape 1.1

```
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);
}

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

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

Netscape 1.1 (2)

Problems:

- MD5 is broken: collision can be found “easily”
- pid and ppid are 15 bit on most UNIX systems
- the sum pid + (ppid << 12) gives only 27 bits of “randomness” not 30
- the value of seconds can be predicted
- pid can be leaked (e.g., local mail message on UNIX can leak pid)

Debian Bug: predictable random number generator in openssl

Security report: DSA-1571-1 openssl -- predictable random number generator

They removed two lines of code:

```
MD_Update(&m, buf, j);
...
MD_Update(&m, buf, j);
```

to avoid warnings on uninitialized data from Valgrind/Purify. This small change made the key generation phase predicatable as it was then only based on the current PID (15 bit of randomness).

CSPRNG: requirements

Requirement of PRNG plus:

- 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":
  - in the event that part or all of 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 CSPRNG state.

BSI evaluation criteria

German Federal Office for Information Security (BSI) defines four criteria for evaluation:

- K1: A sequence of random numbers with a low probability of containing identical consecutive elements (runs).

- K2:
  - monobit test: equal numbers of ones and zeros in the sequence
  - poker test: a special instance of the X^2 (chi-square) test
  - runs test: counts the frequency of runs of various lengths
  - longruns test: checks whether there exists any run of length 34 or greater in 20 000 bits of the sequence
  - autocorrelation test

BSI evaluation criteria (2)

- K3: it should be impossible for attacker to calculate/guess, from any given sub-sequence, any previous or future values in the sequence, nor any inner state of the generator

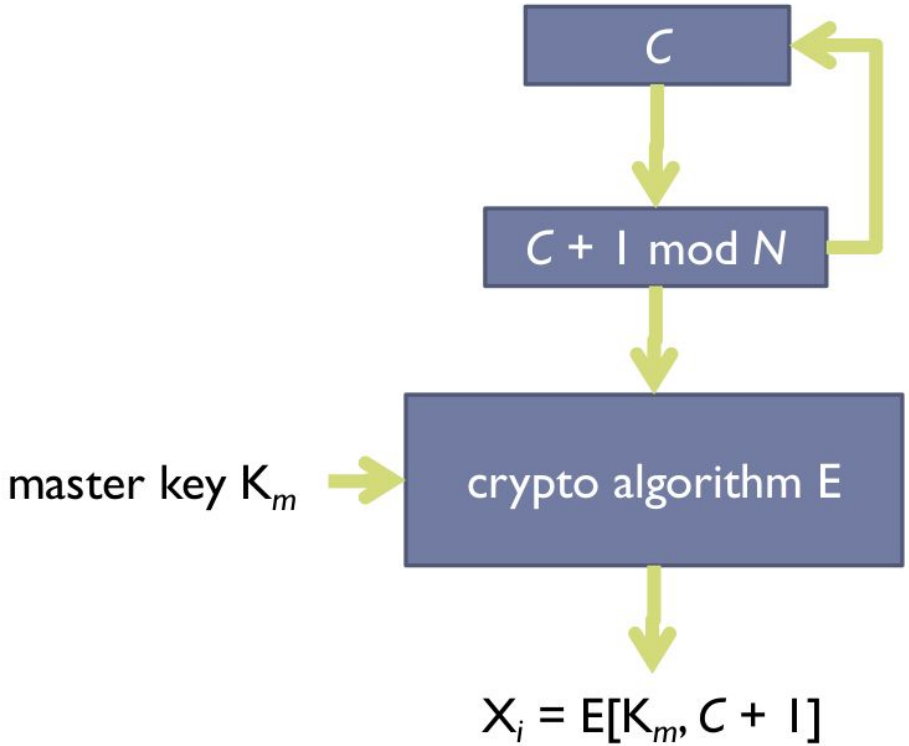
- K4: It should be impossible for attacker to calculate/guess from an inner state of the generator, any previous numbers in the sequence or any previous inner generator states

K4 is required for CSCPRNG.

NIST SP 800-22: Statistical Test Suite for RNG

This document from NIST describes a statistical test suite for evaluating the quality of RNG.

CSPRNG: cipher of a counter



CSPRNG: RNG based on RSA

Perform RSA setup: p, q, n, e, d

Generate numbers:

```
z = seed
while (1) {
    x = x^e mod n;
    // output lsb of x
}
```

CSPRNG: RNG based on 3DES-EDE (ANSI X9.17)

Given DES keys K, 64-bit seed s, current date/time D

Generate numbers as:

```
Y = 3DES_K(D)
while (1) {
    z = 3DES_K(Y xor s)
    s = 3DES_K(z xor Y)
    // output z
}
```

CSPRNG: Blum Blum Shub

Given:

- p and q prime numbers, N = pq
- s randomly chosen s.t. gcd(s, N) = 1

Generate numbers as:

$z = s^2 \bmod N$

```
while (1) {
    z = z^2 mod N
    b = z mod 2
    // output b
}
```

CSPRNG: how to choose the initial seed?

Most CSPRNG requires a (secret, random, unpredictable) initial seed to work. When we cannot obtain it using a TRNG (sometimes available but slow), we have to combine many unpredictable “sources” from a system.

Many OS internally defines a kernel entropy “pool”: random numbers are generated from it and it is replenished with entropy by the kernel.

Kernel entropy pool

When random numbers are generated from the pool the entropy of the pool is diminished (because the person receiving the random number has some information about the pool itself). So as the pool's entropy diminishes as random numbers are handed out, the pool must be replenished: this process is called stirring.

On Linux, a rough estimate of the number of bits of entropy in the pool can be obtain with:

```
> cat /proc/sys/kernel/random/entropy_avail
```

E.g., on my machine when writing this slide: 3724

Kernel entropy pool

Given a state of the entropy pool:

- How to generate new (pseudo-random) numbers?
  Use a cryptography function (e.g., SHA-1) to get the number, taking as input the bits from the pool, then manipulate the bits of the pool (e.g., increment by 1), keep track that the “entropy” is now reduced. In Linux, /dev/random will block until the “level” of entropy is “good enough”, while /dev/urandom will immediately return a number even when the entropy is low.

- How to perform stirring (“increase” entropy)?
  Use many “events” that are very hard to predict: keyboard timings, mouse movements, IDE timings, network timings. In some cases, stirring uses also TRNG.

Kernel entropy pool: why not use just a TRNG?

Several implementation details on /dev/random in Linux can be found in this 2016 paper (some changes have been made over time but still it is useful as a starting point).

Linux can use TRNGs as sources for the pool, for instance, it can use the RDRAND x86 instruction:

- Theodore Ts'o, the main Linux developer behind /dev/random, publicly said that we should not trust TRNG whose implementation cannot be audit (this is the case for RDRAND). There are concerns about the "pressure" that NSA and other GOV agencies may put on chip makers.

- Linux does not rely ONLY on RDRAND but uses it as an additional source (it should be safe starting from 3.13 even assuming an insecure RDRAND implementation). FreeBSD has removed the use of RDRAND *et similia* from the kernel entropy pool.

Finding large random primes

Several algorithms (e.g., RSA) requires to find large prime numbers. This is a complex problem. Notice that we are not interest in the factors but only in knowing whether it is prime. Primality tests are often probabilistic (their output is not 100% accurate but it is correct with high probability).

How common are primes?

The chance that a randomly picked odd integer p̃ is a prime follows from the famous prime number theorem:

$$Pr[\tilde{p} \text{ is prime}] \approx \frac{2}{ln(\tilde{p})}$$

E.g., if we want a prime number with 512 bits

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(2^{512})} = \frac{2}{512\ln(2)} \approx \frac{1}{177}$$

Hence we expect to test on average 177 random odd number before finding a prime.

Fermat Primality Test

Idea: Fermat's theorem holds for all primes. Hence if we find a number a such that $a^{p-1} \not\equiv 1$ then we know the candidate is not a prime number.

Algorithm:

```
1     FOR i = 1 TO s
1.1       choose random a ∈ {2, 3, ..., p̃ − 2}
1.2       IF a^(p̃−1) ≢ 1
1.3           RETURN ("p̃ is composite")
2     RETURN ("p̃ is likely prime")
```

Problem. There are composite numbers, e.g., Carmichael numbers such as 561, for which $a^{\tilde{p}-1} \equiv 1$ is true for many values of a.

Hence, Fermat Primality Test is not used in this form.

Euclid's Lemma

Theorem. If a prime p divides the product ab of two integers a and b, then p must divide at least one of those integers a and b.

E.g., p = 19, a = 133, b = 143, then ab = 133 * 143 = 19019

If we assume that p is prime, then p must divide either a or b: 19 divides 133 = 19 * 7

Miller-Rabin Primality Test (1)

$$a^{p-1} \equiv 1 \bmod p$$ (Fermat's Little Theorem)

$$a^{p-1} - 1 \equiv 0 \bmod p$$ (if we divide by p left side we get a remainder equal to 0)

$$a^{p-1} - 1 = (a^{(p-1)/2} - 1)(a^{(p-1)/2} + 1)$$ (since: $x^2 - y^2 = (x+y)(x-y)$)

$$(a^{(p-1)/2} - 1)(a^{(p-1)/2} + 1) \equiv 0 \bmod p$$

if (p-1)/2 is even then:

$$(a^{(p-1)/4} - 1)(a^{(p-1)/4} + 1)(a^{(p-1)/2} + 1) \equiv 0 \bmod p$$

if (p-1)/4 is even then repeat the same idea... and so on until we that $(p-1)/2^k$ is odd.

Miller-Rabin Primality Test (2)

Hence, we get something like:

$$(a^{(p-1)/2^k} - 1)(a^{(p-1)/2^k} + 1) \cdots (a^{(p-1)/2} + 1) \equiv 0 \bmod p$$

From the Euclid's Lemma, we know that if p is prime it should divide one term of the left side.
Hence, for each step j (from 0 to k), we need to check whether:

$$a^{(p-1)/2^j} \equiv -1 \equiv p - 1 \bmod p$$

And also:

$$a^{(p-1)/2^k} \equiv 1 \bmod p$$

If none of them is divisible by p, then p is not prime.

Miller-Rabin Primality Test (3)

However, if one of them is divisible by p, then p "could" be prime: it can be proved, that using this decomposition strategy, we get the correct result only 3 out 4 times, i.e., 75%.

As in the Fermat's Primality test, we can try different values of a:

- we test one value of a then ¼ to be wrong
- two values of then 1/16 to wrong
- ...and so on

If we try many values for a (e.g., 40) the probability to be wrong is very small.

Miller-Rabin Primality Test (3)

Theorem 7.6.1 *Given the decomposition of an odd prime candidate* $\tilde{p}$

$$\tilde{p} - 1 = 2^u r$$

where r *is odd. If we can find an integer* a *such that*

$$a^r \not\equiv 1 \bmod \tilde{p} \quad \text{and} \quad a^{r2^j} \not\equiv \tilde{p} - 1 \bmod \tilde{p}$$

for all $j = \{0, 1, \ldots, u-1\}$, *then* $\tilde{p}$ *is composite. Otherwise, it is probably a prime.*

Miller-Rabin Primality Test: algorithm (from wikipedia)

K is often called “security parameter”

```
Input #1: n > 3, an odd integer to be tested for primality
Input #2: k, the number of rounds of testing to perform
Output: “composite” if n is found to be composite, “probably prime” otherwise

write n as 2^r·d + 1 with d odd (by factoring out powers of 2 from n − 1)
WitnessLoop: repeat k times:
    pick a random integer a in the range [2, n − 2]
    x ← a^d mod n
    if x = 1 or x = n − 1 then
        continue WitnessLoop
    repeat r − 1 times:
        x ← x^2 mod n
        if x = n − 1 then
            continue WitnessLoop
    return “composite”
return “probably prime”
```

Miller-Rabin Primality Test: how many attempts?

It can be shown that the probability to wrong is even smaller with large primes: hence, the number of attempt s (security parameter) to get an error probability less than 2^{-80}

Bit lengths of $\tilde{p}$	Security parameter s
250	11
300	9
400	6
500	5
600	3

Credits

These slides are based on material from:

- Slides of Prof. D'Amore from CNS 2019-2020
- Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer. http://www.crypto-textbook.com/
- Wikipedia (english version)