Chapter
Secure Random Number Generator

Jean-Louis Roch, Grenoble University, M2-SCCI/SECR

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

-- John Von Neumann, 1951

References:

- **NIST Special Publication 800-90**: « ***Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*** »,
  Elaine Barker, John Kelsey. March 2007
- **Handbook of Applied Cryptography**.
  Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone. August 2001
- + web refs.

Cryptographic Secure Pseudo-Random Number Generator

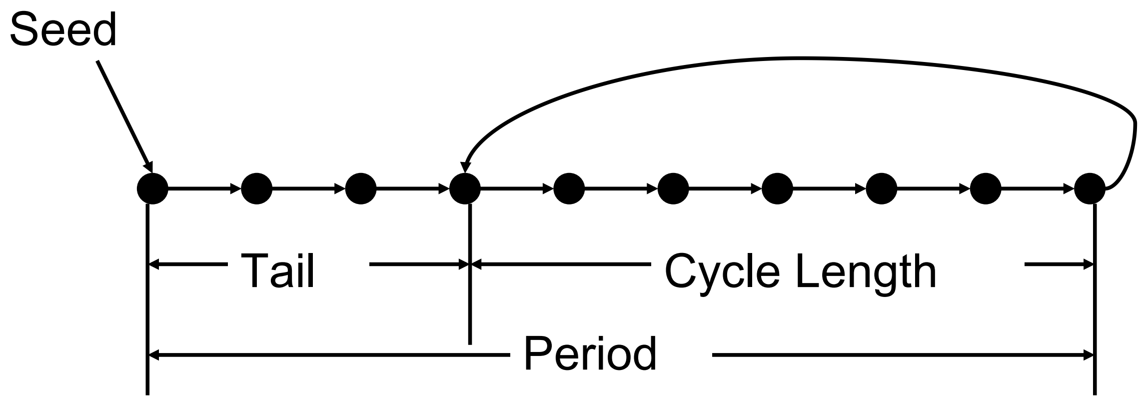
- RNG, PRNG and CSPRNG
  - Pseudorandom bit generation
  - Statistical tests
- De-skewing techniques PRNG
  - Example Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms
- Cryptographically secure pseudorandom bit generation
  - Security proof

Random Bit/Number Generator

- RBG: a device or algorithm which outputs a sequence of statistically independent and unbiased binary digits.
- Hardware-based
  - elapsed time between emission of particle during radioactive decay
  - thermal noise from a semiconductor diode or resistor;
  - the frequency instability of a free running oscillator;
  - air turbulence within disk drive which causes random fluctuations
  - drive sector read latency times
  - sound from a microphone or video input from a camera.
- Software-based
  - the system clock
  - elapsed time between keystrokes or mouse movement
  - content of input/output buffers
  - user input
  - operating system values such as system load and network statistics
- No physical RNG normalized in 2011 (but patents)

Pseudo Random Bit/Number Generator

- PRBG
  - Input: a ***seed*** i.e. a truly random input sequence of length k (the *seed*)
    - Use a physical RNG to initialize the ssinon 0 pts eed (human, date, pid, …)
  - Output: a **deterministic** sequence of length l >> k that "*seems random*"
    - An adversary cannot efficiently distinguish between sequences of PRBG and truly RBG of length l.



PRNG Iteration and random sequence

- S = finite set of states; r = #bits generated at each step.
- **ITERATION (secret)**
  f : S -> S

  RANDOM SEQUENCE (output)
  Bit extraction function **g: S -> {0,1}^{r}**

  – Seed s_0
    initial state = [user+ reseed]
  –
  – $s_1 := f(s_0)$ $r_1 := g(s_1)$
  – $s_2 := f(s_1)$ $r_2 := g(s_2)$
  – … …
  – $s_{i+1} := f(s_i)$ $r_{i+1} := g(s_{1+1})$
  – … …
- Element rank *k* in the sequence : $\mathbf{r_k := g (f^k (s_0))}$
- **Example** [BBS] : **S** = {0, …, n-1}
  – **f**(x) = x^2 mod n - **g**(x) = LSB(x) (i.e. x mod 2)

Pseudo Random Bit/Number Generator

- PRBG
  – Input: a ***seed*** i.e. a truly random input sequence of length k (the *seed*)
    - Use a physical RNG to initialize the seed (human, date, pid, …)
  – Output: a **deterministic** sequence of length l >> k that "*seems random*"
    - An adversary cannot efficiently distinguish between sequences of PRBG and truly RBG of length l.
- PRBG can be used to generate random numbers (ie PRNG)
  – Ex. :RNG of random integers in the interval [0; n] can be built from a RBG
    - Use RBG to generate $\lfloor \lg n \rfloor + 1$ bits and convert to integer (discard if >n)
- Example: Linear Congruential Generator LCG
  – Parameters: m and a, b, x_0 in {0, m-1}
    $x_{n+1} = a.x_n + b \bmod m$ (x_0 is the seed)

  – Eg: Unix PRNG: rand() with seed initialized by srand() ; rand48(), …)

Example: mid-square method

- proposed by von Neumann in the 1940's.
  - starts with a seed,
  - the seed is squared and the middle digits become the random number.
- **Example:**
  - $X_0 = 5497$
  - $X_0^2 = (5497)^2 = 30{,}217{,}009 \Rightarrow X_1 = 2170$
    - $R_1 = 0.2170$
  - $X_1^2 = (2170)^2 = 04{,}708{,}900 \Rightarrow X_2 = 7089$
    - $R_2 = 0.7089$
- Problems: difficult to assure that the sequence will not degenerate over a long period of time
  - zeros once they appear are carried in subsequent numbers (try 5197 as a seed).

- Definitions :
  - a (P)RBG passes all *polynomial-time statistical tests* if no poly algorithm can distinguish between output sequence and truly random sequence of the same length with probability significantly greater that ½
  - a PRBG is a CSPRBP iff it passes the *next-bit test,* i.e. Given first k bits in input, no polynomial-time algorithm can predict the $(k + 1)^{st}$ bit with probability significantly greater than ½
    - Also called right-unpredictable or forward unpredictable
    - Similarly previous-bit test, or left-unpredictable or backward-unpredictable

Statistical tests [FIPS 140-1]

- Why: impossible to give a mathematical proof that a generator is indeed a random bit generator;
  -> the tests help detect certain kinds of weaknesses the generator may have.

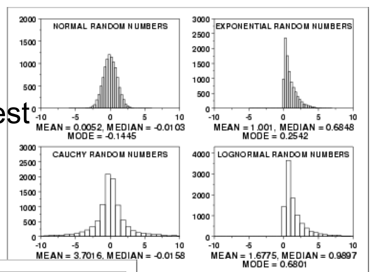
- How: by taking a sample output sequence of the generator and subjecting it to various statistical tests.
  - No risk “0”: “*accepted*” should be replaced by “*not rejected*”
  - Significance Level: α=type 1 error; β = type 2 error (eg = 0.001)

- Five Basic Test (Using Chi-square analysis)
  - Frequency Test: # of 0 and 1
  - Serial Test: # of 00, 01, 10, 11
  - Poker-k Test: # of each k-bit string
  - Run Test: comparing with expected run length
  - Autocorrelation test: correlations between s and shifted version

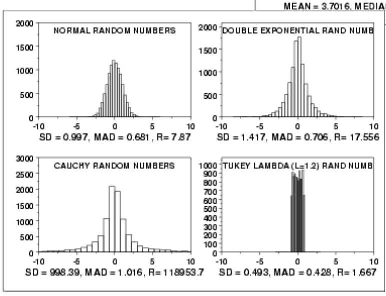
Common classical quantitative tests

See: Exploratory Data Analysis, *NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/* [http://www.itl.nist.gov/div898/handbook/eda/section3/eda35.htm]

- **Location**
  - Measures of Location
  - Confidence Limits for the Mean and One Sample t-Test
  - Two Sample t-Test for Equal Means
  - One Factor Analysis of Variance
  - Multi-Factor Analysis of Variance



- **Scale (or variability or spread)**
  - Measures of Scale
  - Bartlett's Test
  - Chi-Square Test
  - F-Test
  - Levene Test



- **Skewness and Kurtosis**
  - Measures of Skewness and Kurtosis

$$skewness = \frac{\sum_{i=1}^{N}(Y_i - \bar{Y})^3}{(N-1)s^3}$$

- **Randomness**
  – Autocorrelation
  – Runs Test

- **Distributional Measures**
  – Anderson-Darling Test
  – Chi-Square Goodness-of-Fit Test
  – Kolmogorov-Smirnov Test

- **Outliers**
  – Detection of Outliers
  – Grubbs Test
  – Tietjen-Moore Test
  – Generalized Extreme Deviate Test

- **2-Level Factorial Designs**
  – Yates Analysis

Some random number test suites

- NIST test suite of random number generators:
  [http://csrc.nist.gov/groups/ST/toolkit/rng/batteries_stats_test.html]

- Diehard tests [G. Marsaglia]
  [http://www.stat.fsu.edu/pub/diehard/]

- Dieharder [R. Brown, D. Eddelbuettel, D. Bauer,
  [http://www.phy.duke.edu/~rgb/General/dieharder.php]

- **TestU01**[P. L’Evuyer, R. Simard] 2009
  [http://www.iro.umontreal.ca/~simardr/testu01/tu01.html]
  – TestU01: A C Library for Empirical Testing of Random Number Generators,
  P. L'Ecuyer and R. Simard,
  ACM Transactions on Mathematical Software, Vol. 33, 4, article 22, 2007.

Cryptographic Secure Pseudo-Random Number Generator

- RNG, PRNG and CSPRNG
  – Pseudorandom bit generation
  – Statistical tests
- **De-skewing techniques PRNG**
  – Example Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms
- Cryptographically secure pseudorandom bit generation
  – Security proof

De-skewing techniques

- A PRNG may be defective:
  output bits may be biased or correlated
- De-skewing techniques: to generate "truly" random bit sequences from the output bits of a defective generator
  – To suppress the biais (von Neumann technique)
  – To decrease correlation (combination of 2 sequences) (eg Vitany (δ,ε)-decorrelation)
- **In practice**: to pass sequence whose bits are biased or correlated through
  – a hash function (eg SHA-1/2)
  – or a block cipher

Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms

Charles E. Leiserson, **Tao B. Schardl**, and Jim Sukha

MIT Computer Science and Artificial Intelligence Laboratory

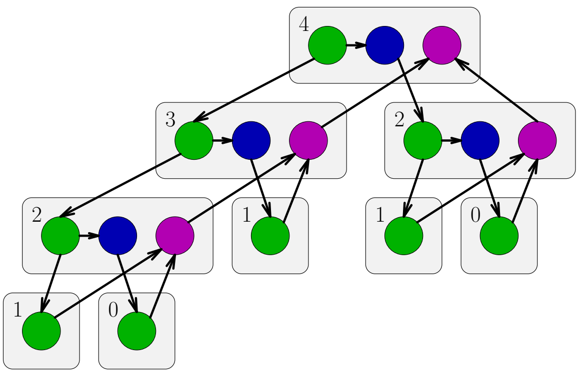
PPoPP 2012

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



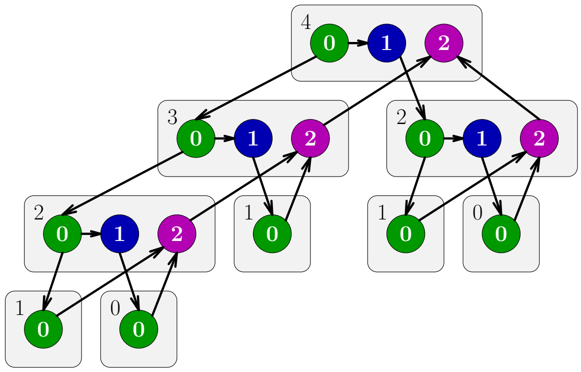
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

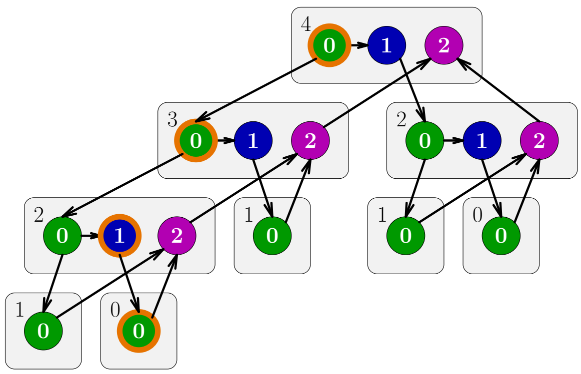
Pedigrees

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



$J = \langle 0, 0, 1, 0 \rangle$

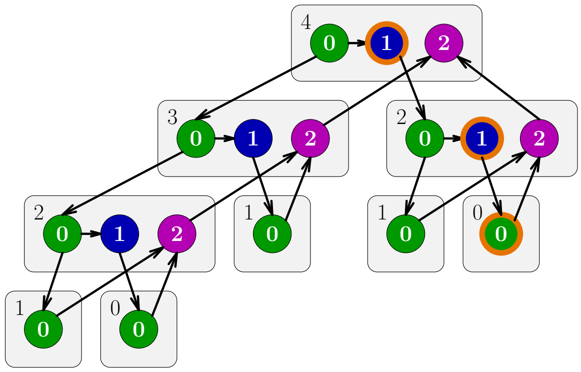
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



$J = \langle 1, 1, 0 \rangle$

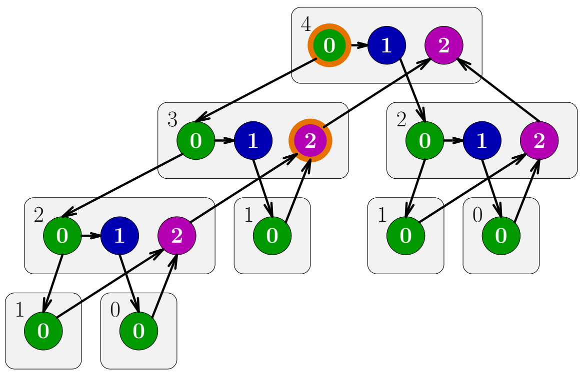
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



$J = \langle 0, 2 \rangle$

- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Outline

The DOTMIX DPRNG

DOTMIX hashes a pedigree in two stages.

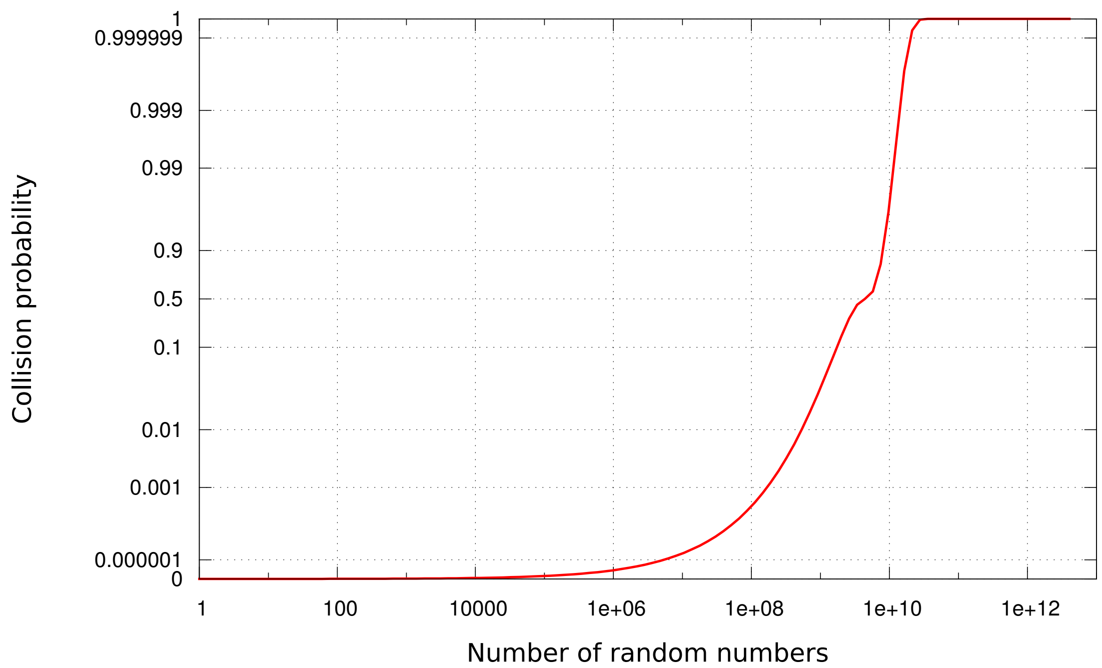
1. ***Compression***: Convert the pedigree into a single word while preserving uniqueness.
2. ***Mixing***: Remove correlation between the compressed pedigrees.

DotMix compression

Dot-product compression: Compute the dot product of the pedigree with a vector of random odd 64-bit integers.

Theorem: For any randomly chosen vector Γ of odd integers and any two distinct pedigrees J and J', the probability that $\Gamma \cdot J = \Gamma \cdot J'$ is at most $1/2^{63}$.

Efficacy of DotMix



DOTMIX mixing

DOTMIX(r) “randomly” permutes the result of the compression function using r iterations of the following “mixing” routine.

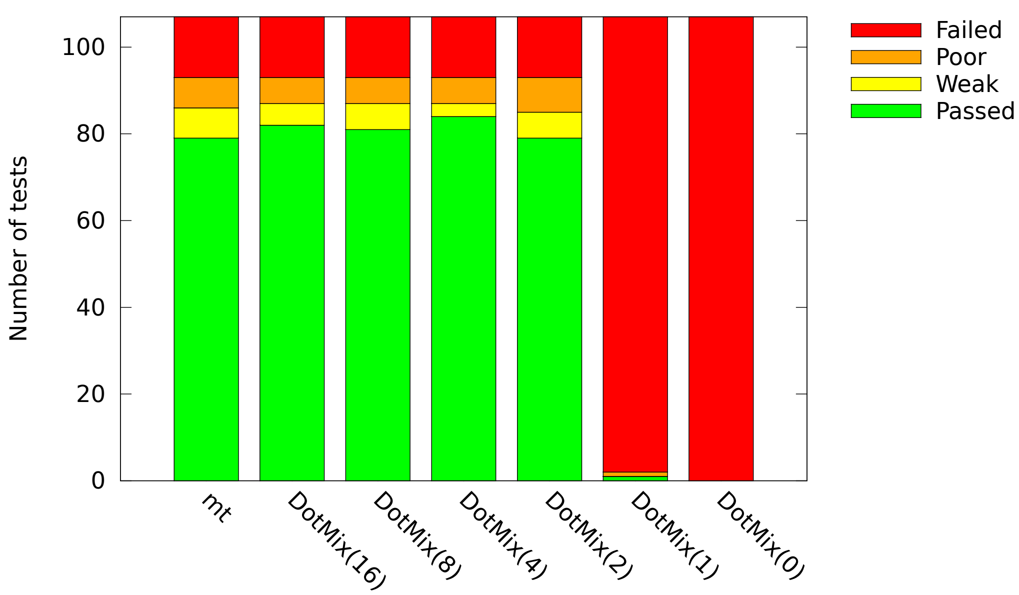
RC6 mixing: Let X_i designate the result of the ith round of mixing, where X_0 is the result of the compression function.

for (int $i = 0$; $i < r$; $++i$) {
  $Y = X_i \cdot (2X_i + 1) \bmod 2^{64}$;
  $X_{i+1} =$ swap left and right halves of Y;
}

One can show that this function is bijective [CRRY98], so mixing does not generate further collisions.

Thanks to Ron Rivest for suggesting this mixing function.

Dieharder statistical tests



Examples of normalized PRNG

- ANSI X9.17 generator
  - Input: m, a random seed s, Triple-DES encryption key k.
  - Output: m pseudorandom 64-bit strings $x_1, x_2, \ldots, x_m$
    - Let $I = E_k(D)$ with D=64-bit date/time (finest available resolution)
    - For i=1.. m { $x_i \leftarrow E_k(I \oplus s)$; $s \leftarrow E_k(x_i \oplus I)$; };
    - Return($x_1, x_2, \ldots, x_m$)
- FIPS 186 for DSA
  - Input an integer m and a 160 prime number q
  - Output: m pseudorandom numbers $k_1, \ldots, k_m$ in {0, .., q-1}
  - Parameters: (b,G) = (160, DES) or (b,G) = (160..512, SHA1)
    - Let s be a secret random seed with b bits
    - Let t= 160 bits constant t = efcdab89 98badcfe 10325476 c3d2e1f0 67452301
    - For i=1.. m { $k_i \leftarrow G(t, s) \bmod q$; $s \leftarrow (1 + s + k_i) \bmod 2^b$; };
    - Return($k_1, \ldots, k_m$)

Cryptographic Secure Pseudo-Random Number Generator

- RNG, PRNG and CSPRNG
  - Pseudorandom bit generation
  - Statistical tests
- De-skewing techniques PRNG
  - Example Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms
- **Cryptographically secure pseudorandom bit generation**
  - **Security proof**

Some Provable CSPRNG

[Ben Lynn, http://crypto.stanford.edu/pbc/notes/crypto/prng.xhtml]

- RSA Generator :
  - Primes p, q; $n = p.q$ and $\Phi = (p - 1)(q - 1)$; e (3 or …)
  - $x_k = x_{k-1}^e \bmod n$; output: $b_k = x_k \bmod 2$ [ie $LSB(x_k)$]
- Blum-Micali Generator :
  - Prime p, g generator of Z/pZ*;
  - $x_k = g^{xk-1} \bmod p$; output: $b_k = 1$ if $x_k \geq (p-1)/2$; else 0 [ie $HSB(x_n)$]
- Blum-Blum-Shub (BBS) Generator:
  - Primes p, q of the form 4m+3 ; n=p.q
  - $x_k = x_{k-1}^2 \bmod n$; output: $LSB(x_k)$

Blum-Blum-Shub (BBS) CSPRNG

- Primes p, q of the form 4m+3; n=p.q
- seed s prime to n (why?); $x_0 = s^2 \bmod n$;
- $x_k = x_{k-1}^2 \bmod n$; output: $LSB(x_k) = x_k \bmod 2$

Table 5.2 Example Operation of BBS Generator

s	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

s	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

Security proof: example

- Theorem:
  If it is impossible to compute [… one way function …], then the PRNG is computationally secure
  - Proof of left-unpredicatbility (previous bit)
  - Proof of right-unpredicatbility (next bit)

  - By polynomial time reduction from computation of s
    - To inverse a one-way function by using an Oracle RightPrediction

---

- General scheme of a polynomial-time reduction
- AlgoReductionF (y) // outputs x such that y=F(x), where
  // F is conjectured one-way

```
{
  Let G=PRNG built from y ;
  for (b_0=0..1) // Speculation loop with fixed b_0: polynomial time log^O(1)|x|
   { … ;
     // Use oracle to predict log^O(1)|x| bits
     … b_i = OracleRightPrediction(b_0, …, b_{i-1}) ;
     x= … ; // compute x
     z= F(x) ;
      if (z==y) return x ;
    }
 }
```

- May be extended to O(loglog |x|) bits extracted :
  - #speculation loop=$2^{O(\log\log |x|)}$ = $O(\log^{O(1)}|x|)$: yet polynomial time
    Ex: BBS, RSA provable secure with O(loglog n) bits at each iteration
  - Constant of O() : matters a lot in practice!!
    =>Fine analysis of complexity required!

Example: Blum-Micali is CSPRNG

- **Blum-Micali: in F_p, with g primitive element mod p**
  $f(x) = g^x$ mod p ; hardcore bit: b = HSB(x)

  BM generator: x0 = seed (or reseed)
  $x_k = g^{x_{k-1}}$ mod p ;
  $b_k = 1$ if $x_{k-1} \geq (p-1)/2$; else 0 [ie HSB(x_{k-1})]
- **Theorem**: if there exists A, 1 <A<p, such that
  it is impossible to compute α such that g^α = A mod p
  then BM generator is resistant to right and left prediction.
- Proof: by reduction:
  DiscreteLog $\leq_P$ PreviousBitBM $\leq_P$ NextBitBM
- **Assumption** (f one-way permutation distinguishable in polynomial time):
  it exists $N = \log^{O(1)} p$ such that for all $s=(b_1, \ldots, b_N)$ in $\{0,1\}^N$,
  there exists an unique seed x that generates s.

Prop. 1: PreviousBit_BM $\geq_P$ DiscreteLog

- OraclePreviousBitBM ($b_i, b_{i+1}, \ldots, b_k$) returns b_{i-1}.
  - From state=x, PLOG_HSB (x) returns 1 ***iff*** ($\text{DiscreteLog}_g\, x \geq (p-1)/2$).
  - **PLOG_HSB(x) $\leq_P$ PreviousBitBM**
    - AlgoReductionPLOG_HSB(x)
      { for (y_0 = x, i=1; i <= log p ; ++i) { $y_i = g^{y\_\{i-1\}}$; $b_i = (y_{i-1} \geq (p-1)/2$) ? 1 : 0 ; }
      return b_0= OraclePrevioustBitBM ($b_1, b_2, \ldots, b_{\log p}$) ; }
    - **Lower Bound:** PreviousBitBM ≥ BitPredictionBM(x) – O(log3 p)
- An Oracle for *BitPredictionBM* enables to compute α such that $A = g^\alpha$ mod p in polynomial time [thus breaks discrete log] :
  - AlgoReductionDiscreteLog(A)
    { for (k = $\log_2$ p , i = 0; i <=k; i+=1)
    { bi = OraclePLOG_HSB(A^{2^i} mod p); res = res + bi * $(p-1)/2^{i+1}$; }
    return α = res ; }
  - **Lower Bound**: PLOG_HSB ≥ $(\log_2 p)^{-1}$.DiscreteLog – O ($\log^2$ p)
- Thus: **DiscreteLog $\leq_P$ PLOG_HSB $\leq_P$ PreviousBitBM**
  Can be extended to randomized attack.

Prop. 2: NextBit_BM $\geq_P$ DiscreteLog

- Sketch of the Proof: *if Eve can predict the next bit, then she can compute the previous bit !*
- **PreviousBitBM $\leq_P$ NextBitBM**
  Note that OracleNextBitBM ($b_i, b_{i+1}, \ldots, b_k$) returns b_{k+1}.
  Proof by reduction:

```
AlgoReductionPreviousBitBM(b_i, b_{i+1}, … , b_k)
  {       // Returns b_{i-1} which is either 0 or 1: just speculate to find the good value !
          for (j=1; true ; j+=1 )
         {  b_{k+j}  = OracleNextBitBM(b_{i+j-1}, b_{i+j}, … , b_{k+j-1}) ;  // the correct value of  b_{k+j}
            hyp0 = OracleNextBitBM ( 0, b_i, b_{i+1}, …,  b_{k+j-1}) ; // value if previous bit = 0
            hyp1 = OracleNextBitBM ( 1, b_i, b_{i+1}, …,  b_{k+j-1}) ; // value if previous bit = 1
            if (hyp0 ≠ hyp1)  // Then we know the value of the previous bit b_{i-1} !
            {   if (b_{k+j} = hyp0) return 0; else return 1 ;
  }       }  }
```

- Finally:
  DiscreteLog $\leq_P$ PLOG_HSB $\leq_P$ PreviousBitBM $\leq_P$ NextBitBM

Remark: extracting, at each step, *loglog p* bits instead of 1 is provably secure. [since loglog p bits can be speculated in polynomial time]

Security of RSA Generator

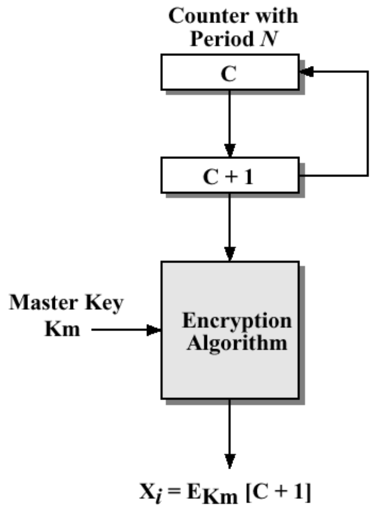
- RSA - PRNG:
  – Primes p, q; $n = p.q$ and $\Phi = (p - 1)(q - 1)$; e (3 or …)
  – x_0 = initial seed (prime to n)
  – $x_{k+1} = x_k^e \bmod n$; output: $b_{k+1} = x_{k+1} \bmod 2$ [ie LSB(x_k)]

- **RSA Hypothesis**. Let M proportional to $N^{2/e}$.
  For x in {1,…,M}, the distribution induced by $x^e \bmod n$ cannot be distinguished in polynomial time from the uniform distribution on {1, …, n}.

- Under RSA hypothesis,
  RSA-PRNG is cryptographically secure.

Example of PRNG based on block cipher

- Block cipher :
  – secret key and counter mode
  – The counter mode can be replaced by a RNG.



- Provable secure PRNG under the black box model

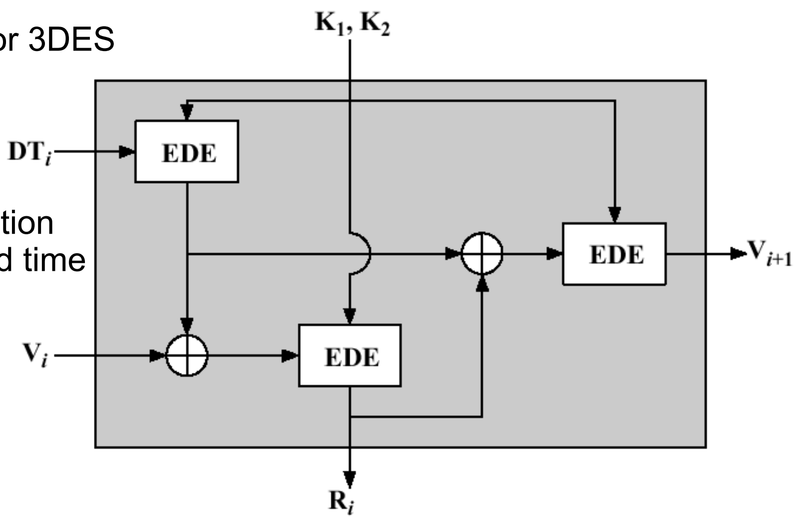
ANSI X9.17 CSPRNG

[Cadence / Document Number:I-IPA01-0087-USR, 2008]

•K_1 and K_2 are two keys for 3DES

•DT_i is a 64 bit representation of current system date and time

•V_i =initialization value (initially, V_0 =seed)

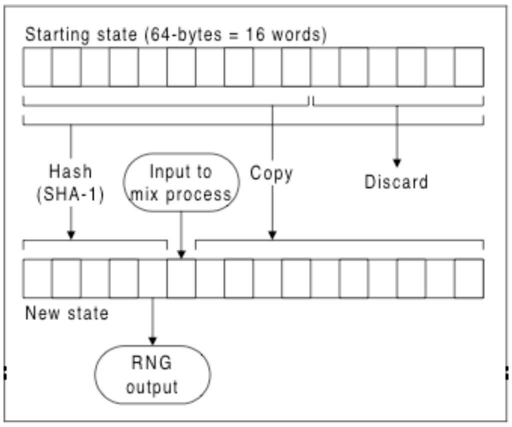


•R_i is the Random Number generated

•V_{i+1} is the initialization value for the next iteration

Intel Random Number Generator

- cf Intel Random Number Generator (B. Jun, P. Kocher, 1999)
  - Intel 80802 Firmware Hub chip included a hardware RNG
    - optional on 840 chipset, not included in current PCs
  - Uses two oscillators (hardware)
    - one fast, one slow, the slow is modulated by a thermal noise from two diodes)
  - Output debiaised using Von Neumann decorrelation step
  - Finally, mix process using SHA1:
    - 32 bits from the RNG are input to a SHA1 mixer, that provides the final 32 bits output.



Some readings

- RFC1750.txt Randomness Recommendations for Security (D. Eastlake, S. Crocker, J. Schiller, 1994)

> *Is there any hope for strong portable randomness in the future? There might be. All that's needed is a physical source of unpredictable numbers. A thermal noise or radioactive decay source and a fast, free-running oscillator would do the trick directly. This is a trivial amount of hardware, and could easily be included as a standard part of a computer system's architecture… All that's needed is the common perception among computer vendors that this small additional hardware and the software to access it is necessary and useful.*
>
> – Eastlake, Crocker, and Schiller , "RFC 1750: Randomness Recommendations for Security," *IETF Network Working Group*, December 1994.

Back slides

Consider this simple idea for constructing a PRNG: seed the state with some key and pick some encryption algorithm such as DES. Then each iteration, encrypt the current state and output a few bits of it. Intuitively, this seems like it should produce random-looking bits.

The Blum-Micali scheme mimics this process, but on a firm theoretical foundation, by using hardcore bits.

Let $f : \{0,1\}^n \to \{0,1\}^n$ be a permutation and $B : \{0,1\}^n \to \{0,1\}$ be a (t, ϵ)-hardcore bit of f. Define $G_{\mathrm{BM}} : \{0,1\}^n \to \{0,1\}^m$ as follows:

Pick random seed $S \in \{0,1\}^n$. For $i = 1$ to m 1. Output $h_i = B(S)$ 2. $S \leftarrow f(S)$

Theorem [Blum-Micali '81]: If B is (t, ϵ)-hardcore then G_{BM} is a $(t - m\,\mathrm{TIME}(f), \epsilon\, m)$-PRNG.

Proof: First we devise notation to record the reverse of a bit string. Define $G^R_{\mathrm{BM}}(S) = [G_{\mathrm{BM}}(S)]^R$, that is, if $G_{\mathrm{BM}}(S) = b_1 ... b_m$, then $G^R_{\mathrm{BM}}(S) = b_m ... b_1$.

Then note that if G^R_{BM} is a (t, ϵ)-PRNG, then G_{BM} is also a (t, ϵ)-PRNG.

Now suppose G^R_{BM} is not a $(t - m\,\mathrm{TIME}(f), m\,\epsilon)$-PRNG. Then there exists a $(t - m\,\mathrm{TIME}(f))$ algorithm A, and $0 \le i < m$ such that

$$\Pr[A(G^R_{\mathrm{BM}}(S)\,|_{1...i}) = G^R_{\mathrm{BM}}(S)\,|_{i+1}] \ge 1/2 + \epsilon$$

We shall build an algorithm A' that predicts $B(x)$ given $f(x)$.

Let $S \in \{0,1\}^n$ and define $y = f^{m-i}(S)$. Then

$$\begin{aligned} G^R_{\mathrm{BM}}(S)|_{1...i} &= [B(f^m(S)), B(f^{m-1}(S)), ..., B(f^{m-i+1}(S))] \\ &= [B(f^i(y)), ..., B(f(y))] \end{aligned}$$

Algorithm A' acts as follows. Given $z = f(y)$,

1. Compute $T(z) = [B(f^{i-1}(z)), ..., B(z)]$
2. Output $A(T(z))$.

Note that $\mathrm{TIME}(A') = t$. Then we wish to show that

$$\Pr[A'(f(y)) = B(y) \mid y \leftarrow \{0,1\}^n] \ge 1/2 + \epsilon$$

This follows since f is a permutation and hence the distribution $\{T_z \mid y \leftarrow \{0,1\}^n, z = f(y)\}$ is identical to the distribution $\{G^R_{\mathrm{BM}}(S)\,|_{1...i} \mid S \leftarrow \{0,1\}^n\}$.

This is a contradiction because B is (t, ϵ)-hardcore for f.

Examples of BM generators

- **Dlog generator:** $p = 1024$-bit prime, $g \in \mathbb{Z}_p^*$ a generator. Let $f : \{1, ..., p-1\} \to \{1, ..., p-1\}$, $f(x) = g^x \bmod p$. We know $\mathrm{MSB}(x) = \{0 \mathrm{if} x < p/2, 1 \mathrm{if} x > p/2$ is a (t, ϵ)-hardcore bit of f if no tn^3/ϵ-time discrete log algorithm exists. Thus we have a PRNG assuming Dlog is hard.
- **Blum-Blum-Shub (BBS):** $N = \mathrm{pq}$ 1024-bit, $p = q = 3 \mathrm{mod} 4$. Let $\mathrm{QR}_N = \{x \in \mathbb{Z}_N^* \mid x \mathrm{is\ QR}\}$. Then $f(x) = x^2 \bmod N$ is a permutation of QR_N.

  LSB(x) is (t, ϵ)-hardcore for f assuming no (tn^2/ϵ) factoring algorithm exists.

  1. $S \leftarrow \mathrm{QR}_N$
  2. Output $\mathrm{LSB}(S)$
  3. $S \leftarrow S^2 (\mathrm{mod}\, N)$
  4. Goto step 1

  [BBS not (t, ϵ)-hardcore implies LSB is not $(t, \epsilon/m)$-hardcore (where m is the number of output bits), which implies there exists a $t(n^m/\epsilon)^2$-time factoring algorithm (where $n = \lg N$).]

  Example: suppose no 2^{100}-time factoring algorithm exists for 1024-bit numbers, and that $m = 2^{20}$. Then we get that BBS is secure for $t/\mathrm{epsilon}^2 = 2^{40}$, e.g. BBS is a $(2^{20}, 2^{-10})$-PRNG, which is not secure.

Speeding up BM

We can output one bit per application of f. Can we output more?

For Dlog it turns out that for $i = 1, ..., n/2$ the msb_i(x) is a hardcore bit. But this is not enough. We need a notion of simultaneous security.

Definition: Let $f : \{0,1\}^n \to \{0,1\}$. Then bits $B_1, ..., B_k : \{0,1\}^n \to \{0,1\}$ are (t, ϵ)-simultaneously secure if $\{f(x), B_1(x), ..., B_k(x) \mid x \in \{0,1\}\}$ is (t, ϵ)-indistinguishable from $\{f(x), r_1, ..., r_k \mid r_1 ... r_k \leftarrow \{0,1\}^k$.

The Blum-Micali Theorem remains true for simultaneously secure bits.

Best result for Dlog [Shamir-Schrift]: $N = \mathrm{pq}$, $f(x) = g^x \bmod N$. Then the bits in the most significant half of x are (t, ϵ)-simultaneously secure for f assuming no $O(t(n/\epsilon)^3)$-time factoring algorithms exist.