Generátory náhodných čísel

- ideálny RNG, základné vlastnosti
- klasifikácia RNG (PRNG, TRNG, hybridné)
- princíp PRNG
- kryptograficky bezpečné PRNG
- štatistické testy FIPS a NIST
- využitie náhodných čísel v štandarde PKCS #1
- implementácia Intel RNG (*TBD)

Primárny zdroj informácii k dnešnej prednáške:

[1] prof. Ing. Dušan Levický, CSc.
APLIKOVANÁ KRYPTOGRAFIA
od utajenia správ ku kybernetickej bezpečnosti
Elfa, Košice, 2018 (str.113-119)

[2] Drutarovský, M.: Kryptografia pre vstavané kryptografické systémy. TUKE, 2017 (str.73-92)

Typické použitie v kryptografii

Náhodné čísla (**Random numbers**) sú pre kryptografiu a jej aplikácie v informačnej bezpečnosti mimoriadne dôležité. Ako **príklady použitia** náhodných čísel v kryptografických algoritmoch možno uviesť:

• generovanie **kľúčov relácií** v symetrickom šifrovaní,
• generovanie **kľúčov pre algoritmus RSA** v kryptografii s verejným kľúčom,
• generovanie **digitálneho podpisu** na báze algoritmu DSA (princíp algoritmu DSA bude opísaný v nasledujúcej prednáške),
• použitie **identifikátorov** v autentizačných krokoch distribúcie kľúčov,
• **vypĺňanie (padding)** blokov správ na zvýšenie bezpečnosti šifrovania.

Generovanie náhodných čísel má charakter generovania **postupností náhodných čísel** (sequence of random number), na ktoré sa kladú **dve základné požiadavky** a to:

• **náhodnosť** (randomness)
• **nepredikovateľnosť** (unpredictability).

Náhodnosť je vlastnosť, ktorá vyjadruje štatistické vlastnosti postupnosti náhodných čísel. Na posúdenie náhodnosti postupnosti náhodných čísel sa používajú dve kritéria:

• **rovnomernosť rozdelenia** (uniform distribution)
• **štatistická nezávislosť** (independence).

Rovnomernosť rozdelenia vyjadruje distribúcie náhodných čísel v postupnosti, ktorá by mala byť rovnomerná, čo znamená, že početnosť výskytu každého náhodného čísla v tejto postupnosti by mala byť rovnaká.

Štatistická nezávislosť znamená, že žiadne náhodné číslo v postupnosti nie je závislé od iného náhodného čísla v tejto postupnosti.

Je potrebné poznamenať, že pokiaľ **na vyhodnocovanie rovnomernosti rozdelenia** existujú **dobre prepracované testy**, testovanie **štatistickej nezávislosti** postupnosti náhodných čísel je **zložitý problém**.

Neexistujú testy, ktoré by jednoznačne potvrdili štatistickú nezávislosť!

Dokázať pomocou testovania, že nejaká postupnosť je **štatisticky nezávislá** (a teda náhodná), **je nemožné**. V praxi sa používajú testy, ktoré skôr dokazujú, že daná postupnosť nie je štatisticky nezávislá. Všeobecne používanou stratégiou je viacnásobne aplikovať tieto testy dovtedy, až sa dosiahne dostatočná hladina významnosti v tom zmysle, že daná postupnosť náhodných čísel je štatisticky nezávislá. Aj tento postup však má určité obmedzenia ...

Nepredikovateľnosť je vlastnosť, ktorá vyjadruje vlastnosť postupnosti náhodných čísel, že susedné čísla v tejto postupnosti nie sú predikovateľné, t. j. **zo znalosti jedného, resp. viacerých náhodných čísel** postupnosti **nemožno predpovedať ďalšie** náhodné číslo tejto postupnosti.

V štatisticky nezávislej náhodnej postupnosti (**true random sequence**) žiadne náhodné **číslo nie je závislé od iného čísla** v tejto postupnosti, preto táto postupnosť má vlastnosť **nepredikovateľnosti**.

Na generovanie postupností náhodných čísel sa používajú **generátory náhodných čísel** RNG (**Random Numbers Generators**).

Ideálny binárny RNG (Random Number Generator)
Typickým príkladom RNG, ktorý sa blíži k ideálnemu RNG je postupnosť bitov získaná hádzaním **nefalšovanej mince**. Jednotlivé výstupy (hody mincou) **sú na sebe nezávislé** a výsledné **rozloženie je rovnomerné** (oba výsledky experimentu sú rovnako pravdepodobné). Keďže opísaný RNG má **dva výstupné stavy**, hovoríme o **binárnom RNG**. Spojením *n* bitov môžeme získať *n*-bitové výstupné slová (*n*-bitové čísla) a získame tak RNG.

Z hľadiska spôsobu generovania náhodných čísel, možno generátory náhodných čísel rozdeliť do dvoch základných kategórií:

• **nedeterministické** RNG,
• **deterministické** RNG.

Nedeterministické generátory náhodných čísel sa tiež označujú termínom **skutočne náhodné RNG** a označujú sa skratkou **TRNG** (**True Random Numbers Generators**). Na generovanie náhodných čísel využívajú fyzikálne javy ako napr. termálny šum, šum polovodičových prvkov, polčas rozpadu rádioaktívnych látok atď. Výstupná postupnosť čísel, ktoré sa označujú ako **skutočne náhodné čísla** (**true random numbers**), je odvodená od týchto fyzikálnych javov. Nedeterministické generátory náhodných čísel sa **vyznačujú narastaním entrópie s narastajúcim časom**, ale sú tiež v praxi **výrazne pomalšie** ako deterministické generátory.

Deterministické generátory náhodných čísel používajú na generovanie náhodných čísel algoritmus, ktorý možno opísať pomocou **deterministickej funkcie** a **vnútorného stavu** generátora. Pretože **počet stavov** generátora je **konečný**, výstupná postupnosť je **periodická**, aj keď táto perióda je obvykle veľmi veľká. Generovaná postupnosť náhodných čísel nie je teda náhodná, ale pseudonáhodná a takto generované čísla sa označujú ako **pseudonáhodné čísla** (**pseudorandom numbers**). Deterministické generátory generujúce pseudonáhodné čísla sa označujú termínom **pseudonáhodné RNG**, teda **PRNG** (**Pseudorandom Number Generators**).

V súčasnosti používané PRNG generujú výstupné postupnosti s veľmi dobrými štatistickými charakteristikami a oproti TRNG sú PRNG v praxi **zvyčajne rýchlejšie**. V praktických aplikáciách sa často používajú **hybridné RNG**, ktoré sú **kombináciou TRNG a PRNG**. Hybridné RNG kombinujú výhodné vlastnosti PRNG ako sú rýchlosť a dobré štatistické vlastnosti výstupnej postupnosti s nedeterministickým charakterom výstupnej postupnosti TRNG, ktorý **zaručuje rast entrópie** s narastajúcim časom.

V ďalšej časti uvedieme **vybrané typy** generátorov PRNG.

Funkcia RAND v štandardných C knižniciach

Knižničná funkcia **rand()** je dostupná prakticky vo všetkých **distribúciách prekladačov jazyka C**, vrátane prekladačov pre MCU. Najčastejšie je v týchto knižniciach funkcia rand() implementovaná ako **lineárny kongruentný generátor**, v niektorých systémoch však existujú aj jej zložitejšie implementácie (napr. **aditívny PRNG**). V tejto časti opíšeme vybrané PRNG z obidvoch kategórií.

Spoločnou vlastnosťou týchto PRNG je **veľká rýchlosť generovania** výstupných hodnôt. Tieto typy PRNG však **nie sú vhodné pre kryptografické aplikácie**, keďže ich parametre a vnútorný stav sa dajú pomerne ľahko odvodiť z výstupných hodnôt.

Cieľom nasledujúcich **dvoch PRNG** je na **zdrojovom kóde** ukázať **jednoduchosť** ich implementácie (zvlášť jednoduchosť **vnútornej iteračnej slučky**), čo je primárnym dôvodom ich vysokej rýchlosti. V zdrojových kódoch bude tiež jasne viditeľná **počiatočná inicializácia stavu** opísaných PRNG.

Lineárny kongruentný generátor

Lineárny kongruentný generátor vytvára pseudonáhodnú postupnosť čísel X_1, X_2, $X_3 \ldots$, s využitím lineárnej rekurentnej rovnice

$$X_n = (aX_{n-1} + b) \bmod m, \;\; n \geqq 1, \tag{6.1}$$

kde

m – je modul, $m > 0$,

a – je koeficient, pričom $0 < a < m$,

b – je absolútny člen $0 \leqq b < m$,

X_0 – je počiatočná hodnota (angl. seed) $0 \leqq X_0 < m$.

Parameter m, ktorý definuje modulo redukciu medzivýsledku, je v prípade realizácie zvyčajne volený tak, aby operácia bola vykonaná automaticky bez nutnosti použiť dodatočné matematické operácie. Najčastejšie to sú hodnoty $m = 2^{16}$ resp. $m = 2^{32}$, pre ktoré je operácia modulo realizovaná implicitne, pokiaľ sú operácie v jazyku C realizované pri výpočtoch s presnosťou 16 resp. 32 bitov. Parametre a, b, m je možné zvoliť tak, aby perióda generovanej pseudonáhodnej postupnosti bola maximálne možná a rovná hodnote m. Vhodné hodnoty parametrov a, b, m môžeme nájsť aj vo forme tabuliek [34]. Výhodou tohto typu PRNG je jeho veľká rýchlosť, jednoduchý programový kód a minimálne pamäťové nároky. Konkrétne implementácie lineárneho kongruentného PRNG často obsahujú okrem implementácie vzťahu (6.1) aj extrakciu len špecifických bitov stavovej premennej. U tohto typu PRNG je stavová premenná tvorená obsahom X_n. Vzhľadom na väčšie závislosti medzi nižšími bitmi stavovej premennej sa preferuje využívanie viac významných bitov stavovej premennej. Napríklad funkcia `rand()` v knižnici DEV_C++ (overené pre verziu 5.11 TDM-GCC) využíva nasledujúcu implementáciu v jazyku C:

```
void srand (unsigned int seed) {
  holdrand = (long)seed;
}

int rand (void) {
  return(((holdrand = holdrand * 214013L + 2531011L) >> 16) &
      0x7fff);
}
```

Parametre tohto PRNG sú identické s parametrami PRNG označenom ako Microsoft Visual/Quick C/C++ v [34]. Tento PRNG využíva bity 30 – 16 stavovej premennej X_n. Súčasťou knižníc je aj funkcia `srand()`, pomocou ktorej môžeme definovať (meniť) počiatočnú hodnotu X_0. Je všeobecne známe, že pre kryptografické aplikácie sú lineárne kongruentné PRNG nevhodné, keďže z troch za sebou vygenerovaných výstupných hodnôt X_n, X_{n+1}, X_{n+2} môžeme pomocou riešenia sústavy lineárnych kongruentných rovníc určiť kompletnú štruktúru PRNG, vrátane neznámeho počiatočného stavu X_0.

https://en.wikipedia.org/wiki/Linear_congruential_generator

The following table lists the parameters of LCGs in common use, including built-in *rand()* functions in runtime libraries of various compilers.

Source	modulus *m*	multiplier *a*	increment *c*	output bits of seed in *rand()* or *Random(L)*
Numerical Recipes	2^{32}	1664525	1013904223	
Borland C/C++	2^{32}	22695477	1	bits 30..16 in *rand()*, 30..0 in *lrand()*
glibc (used by GCC)[9]	2^{31}	1103515245	12345	bits 30..0
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++ [10] C90, C99, C11: Suggestion in the ISO/IEC 9899 [11], C18	2^{31}	1103515245	12345	bits 30..16
Borland Delphi, Virtual Pascal	2^{32}	134775813	1	bits 63..32 of *(seed * L)*
Turbo Pascal	2^{32}	134775813 (0x8088405$_{16}$)	1	
Microsoft Visual/Quick C/C++	2^{32}	214013 (343FD$_{16}$)	2531011 (269EC3$_{16}$)	bits 30..16
Microsoft Visual Basic (6 and earlier)[12]	2^{24}	1140671485 (43FD43FD$_{16}$)	12820163 (C39EC3$_{16}$)	
RtlUniform from Native API[13]	2^{31} - 1	2147483629 (7FFFFFED$_{16}$)	2147483587 (7FFFFFC3$_{16}$)	
Apple CarbonLib, C++11's `minstd_rand0` [14]	2^{31} - 1	16807	0	see MINSTD
C++11's `minstd_rand` [14]	2^{31} - 1	48271	0	see MINSTD
MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407	
Newlib, Musl	2^{64}	6364136223846793005	1	bits 63...32
VMS's **MTH$RANDOM**,[15] old versions of glibc	2^{32}	69069 (10DCD$_{16}$)	1	
Java's java.util.Random, POSIX [in]rand48, glibc [in]rand48[_r]	2^{48}	25214903917 (5DEECE66D$_{16}$)	11	bits 47...16
`random0` [16][17][18][19][20] If X_n is even then X_{n+1} will be odd, and vice versa—the lowest bit oscillates at each step.	134456 = $2^3 7^5$	8121	28411	$\frac{X_n}{134456}$
POSIX[21] [jm]rand48, glibc [mj]rand48[_r]	2^{48}	25214903917 (5DEECE66D$_{16}$)	11	bits 47...15
POSIX [de]rand48, glibc [de]rand48[_r]	2^{48}	25214903917 (5DEECE66D$_{16}$)	11	bits 47...0
cc65[22]	2^{23}	65793 (10101$_{16}$)	4282663 (415927$_{16}$)	bits 22...8
cc65	2^{32}	16843009 (1010101$_{16}$)	826366247 (31415927$_{16}$)	bits 31...16
Formerly common: RANDU [4]	2^{31}	65539	0	

Mitchellov a Moorov aditívny PRNG

Vzhľadom na nízku kvalitu klasického lineárneho kongruentného PRNG existuje v niektorých prekladačoch možnosť zvoliť v čase linkovania kódu iný typ PRNG. Napríklad prekladač použitý v MDK ARM Keil umožňuje využiť Mitchellov a Moorov aditívny generátor ([35], s. 26–29) navrhnutý v roku 1958. Tento PRNG na aktualizáciu stavu (tvorí ho 55 hodnôt $X_n, \ldots, X_{n-54}$) využíva vzťah

$$X_n = (X_{n-24} + X_{n-55}) \bmod 2^{31}. \tag{6.2}$$

Vo vzťahu (6.2) je použitá len operácia súčtu, preto sa tento PRNG nazýva **aditívny**, a operácia modulárnej redukcie faktorom 2^{31}. V systémoch, kde je operácia násobenia výrazne pomalšia ako operácia súčtu (typický stav v päťdesiatych rokoch minulého storočia), môže byť tento aditívny PRNG rýchlejší ako lineárny kongruentný PRNG. Mitchellov a Moorov aditívny PRNG je použitý[1] v prípade, ak v MDK nezvolíme použitie úspornej knižnice Microlib. Microlib používa kód lineárneho kongruentného PRNG práve pre jeho minimálne pamäťové nároky.

Funkcia `rand()` je v prípade aditívneho PRNG tvorená nasledujúcimi kódmi v jazyku C:

[1] Overené experimentálne pre MDK verziu z roku 2016 na základe informácie dostupnej v hlavičkovom súbore `<stdlib.h>`. MDK knižnice nie sú dostupné v zdrojových kódoch.

```
static unsigned _random_number_seed[55] = {
 0x00000001, 0x66d78e85, 0xd5d38c09, 0x0a09d8f5, 0xbf1f87fb,
 0xcb8df767, 0xbdf70769, 0x503d1234, 0x7f4f84c8, 0x61de02a3,
 0xa7408dae, 0x7a24bde8, 0x5115a2ea, 0xbbe62e57, 0xf6d57fff,
 0x632a837a, 0x13861d77, 0xe19f2e7c, 0x695f5705, 0x87936b2e,
 0x50a19a6e, 0x728b0e94, 0xc5cc55ae, 0xb10a8ab1, 0x856f72d7,
 0xd0225c17, 0x51c4fda3, 0x89ed9861, 0xf1db829f, 0xbcfbc59d,
 0x83eec189, 0x6359b159, 0xcc505c30, 0x9cbc5ac9, 0x2fe230f9,
 0x39f65e42, 0x75157bd2, 0x40c158fb, 0x27eb9a3e, 0xc582a2d9,
 0x0569d6c2, 0xed8e30b3, 0x1083ddd2, 0x1f1da441, 0x5660e215,
 0x04f32fc5, 0xe18eef99, 0x4a593208, 0x5b7bed4c, 0x8102fc40,
 0x515341d9, 0xacff3dfa, 0x6d096cb5, 0x2bb3cc1d, 0x253d15ff
};

static int _random_j = 23, _random_k = 54;

int rand_C() {
unsigned int temp;
  temp = (_random_number_seed[_random_k] += \
          _random_number_seed[_random_j]);
  if (--_random_j < 0)
    _random_j = 54, --_random_k;
  else if (--_random_k < 0)
    _random_k = 54;
  return (temp & 0x7fffffff);                  /* mod 2^31 */
}

void srand_C(unsigned int seed) {
int i;
_random_j = 23;
_random_k = 54;
  for (i = 0; i<55; i++) {
    _random_number_seed[i] = seed + (seed >> 16);
    seed = 69069*seed + 1725307361;             /* mod 2^32 */
  }
}
```

Stav generátora je uložený v poli **unsigned** `_random_number_seed[55]`, má teda $55 \times 4 \times 8 = 1\,760$ bitov. Teoreticky tak môže mať počiatočná neurčitosť (entropia) generovanej postupnosti až 1 760 bitov. V praxi je však pole inicializované konštantnými hodnotami (viď zdrojový kód), alebo je možné pole **unsigned** `_random_number_seed[55]` inicializovať pomocou funkcie `srand_C()`. Táto funkcia využíva na inicializáciu klasický lineárny kongruentný PRNG, takže reálne využiteľná počiatočná neurčitosť, ktorú môžeme pri reinicializácii využiť je len 32 bitov (veľkosť premennej **unsigned int**). Ak v aplikácii použijeme preddefinované hodnoty **unsigned** `_random_number_seed[55]`, bude počiatočná neurčitosť, samozrejme, nulová. Perióda tohto PRNG nie je presne známa, závisí od počiatočných hodnôt stavu a nie je menšia ako 2^{55}.

Z pohľadu kryptografie je tento PRNG bezpečnejší ako lineárne kongruentné PRNG, pričom je však porovnateľne rýchly. Využíva však podstatne viac pamäte na uloženie stavu, čo je jeho výrazná nevýhoda. V prípade potreby realizovať tzv. kryptograficky bezpečné PRNG sú, z pohľadu bezpečnosti, k dispozícii lepšie prepracované realizácie PRNG. Sú však podstatne pomalšie.

Kryptograficky bezpečné PRNG

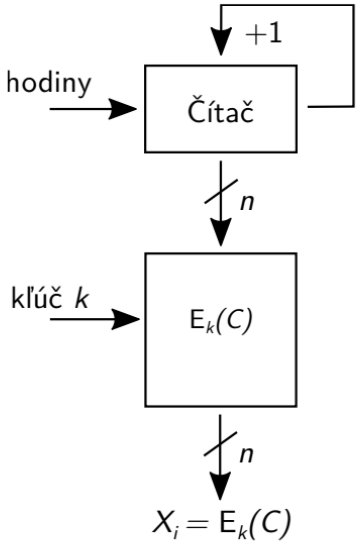
Kryptograficky bezpečné sú také PRNG, u ktorých, zjednodušene povedané, musí platiť:

- Aj napriek znalosti predchádzajúcich k bitov $b_1, b_2, \ldots, b_k$ vygenerovanej postupnosti nie je možné, s pravdepodobnosťou inou ako $\frac{1}{2}$, odhadnúť aký bude nasledujúci vygenerovaný bit b_{k+1}.
- V prípade, že bol odhalený (alebo správne odhadnutý) stav tohto typu PRNG, musí byť na základe stavu nemožné spätne rekonštruovať bity, ktoré boli generované pred kompromitáciou stavu (teda pred tým, než útočník získal alebo odhalil obsah stavovej premennej).

V ďalšej časti opíšeme príklady riešenia PRNG založených na **využití blokovej šifry** a tiež na **zložitosti faktorizácie veľkých čísel**. Pre tieto riešenia boli v predchádzajúcich kapitolách opísané všetky potrebné stavebné bloky vo forme funkcií v jazyku C, ktoré v prípade potreby vieme implementovať. Parametre týchto PRNG môžeme principiálne navrhnúť napr. aj v kalkulačke Magma.

PRNG s využitím čítača

Tento typ PRNG využíva blokový šifrovací algoritmus v zapojení zobrazenom na obr. 6.1. Šifrovací algoritmus $\mathtt{E}_k$ s tajným kľúčom k, ktorý šifruje postupne inkrementované hodnoty n-bitového čítača C. Výstup čítača C je vstupom blokovej šifry $\mathtt{E}_k$ a v i-tej iterácii je na výstupe PRNG hodnota $X_i = \mathtt{E}_k(C)$. Hodnota čítača C tvorí stav PRNG. Tajný kľúč k musí ostať utajený. Jeho kompromitácia vedie k okamžitému „nalomeniu“ PRNG a možnosti kompletnej rekonštrukcie všetkých v minulosti aj v budúcnosti vygenerovaných hodnôt. Pre n-bitový čítač a kvalitný blokový šifrovací algoritmus (napr. AES) má PRNG periódu 2^n. Rýchlosť PRNG s čítačom určuje rýchlosť implementácie blokového šifrovacieho algoritmu. Keďže vo viacerých moderných MCU (vrátane niektorých 8-bitových MCU) je k dispozícii HW koprocesor realizujúci šifrovanie pomocou algoritmu AES, je tento typ PRNG pomerne praktickou alternatívou riešenia kryptograficky bezpečného PRNG vo vstavaných systémoch na báze MCU. V prípade, že HW koprocesor pre šifru AES nie je v MCU k dispozícii, je možné použiť SW implementácie algoritmu AES, ktoré sme opísali v predchádzajúcej kapitole.



Obr. 6.1: Princíp PRNG s využitím čítača a blokovej šifry

Generátor BBS (Blum - Blum - Shub)

PRNG BBS v roku 1986 navrhli Lenore Blum, Manuel Blum a Michael Shub, podľa ktorých je pomenovaný. Bezpečnosť generátora BBS je založená na zložitosti faktorizácie veľkého čísla $n = pq$ na prvočísla p, q. Generátor je opísaný algoritmom 6.1.

Algoritmus 6.1 PRNG BBS

Input: dve veľké tajné náhodné (rôzne) prvočísla p a q, $p \equiv 3 \pmod 4$, $q \equiv 3 \pmod 4$, $n = pq$

Output: generovaná pseudonáhodná bitová postupnosť $b_1, b_2, \ldots, b_l$ dĺžky l

1: zvoľ náhodné celé číslo s (seed) z intervalu $[1, n-1]$ také, že gcd(s,n) = 1, a vypočítaj $X_0 \leftarrow s^2 \bmod n$
2: **for** i from 1 to l **do**
3: $\quad X_i \leftarrow X_{i-1}^2 \bmod n$
4: $\quad b_i \leftarrow$ najnižší významový bit čísla X_i
5: generovaná postupnosť je $b_1, b_2, \ldots, b_l$

Rýchlosť PRNG BBS je výrazne nižšia ako rýchlosť doteraz opísaných PRNG. Je to dané použitím veľkých hodnôt p a q, ktoré sú pre bezpečný PRNG BBS porovnateľné s veľkosťami používanými v šifrovacom algoritme RSA. Na jeho realizáciu môžeme využiť funkcie pre modulárnu aritmetiku s MP číslami.

*Doplnkový materiál k BBS generátoru

Nasledujúci príklad a informácie demonštrujú, ako je možné využiť Magma kalkulačku na hľadanie vhodných parametrov pre BBS PRNG.

Príklad 6.1 Pomocou kalkulačky Magma vygenerujme[2] modul n s veľkosťou 512 bitov, ktorý spĺňuje podmienky potrebné pre realizáciu generátora BBS. Pomocou nájdeného modulu, generátora BBS a počiatočného stavu BBS_SEED=123456 vygenerujeme 100 pseudonáhodných bitov.

Riešenie. Blumovo celé číslo $n = pq$, $p \equiv 3 \pmod 4$, $q \equiv 3 \pmod 4$, s veľkosťou 512 bitov pre generátor BBS môžeme v kalkulačke Magma vygenerovať priamo príkazom:

```
> N := BlumBlumShubModulus(512);
> N;
9190514327795770317947257998572999325978331417347243911451754660072188629 3\
5774217112816044014456522360711933293552040977207754261658891699334524605 1\
313869
> BBS_SEED:=123456;
> b := BlumBlumShub(N, BBS_SEED, 100);
> b;
[ 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0,
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1,
0, 0, 1, 1,0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0,
1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1,
1 ].
```

[2]PRNG implementovaný v Magme využíva 32-bitový počiatočný stav (seed). Je zrejmé, že pre reálne kryptografické aplikácie to je veľmi malá neurčitosť. V praktických aplikáciách je potrebné pre generované bity použiť napr. kvalitný TRNG. Následne je možné na prácu s takto získanými bitmi použiť Magma príkazy. Magmu tak môžeme využiť aj na hľadanie vhodných parametrov pre kryptografické vstavané systémy, nesmieme sa však spoliehať na pseudonáhodné bity z interného PRNG implementovaného v Magme.

V prípade potreby generovať vlastné prvočíselné hodnoty p a q, pre ktoré platia podmienky $p \equiv 3 \pmod 4$, $q \equiv 3 \pmod 4$ môžeme použiť v Magme napr. príkazy:

```
> SetSeed(123456);
> X := {1..2^256-1};
> repeat P:=Random(X);
repeat > until IsPrime(P) and (P mod 4) eq 3;
> P;
581463722571070883403111833298453334585788650667015290543092160524572513 05\
287.
```

Funkciu `IsPrime()` môžeme použiť aj na testovanie prvočíselnosti čísel generovaných z externého TRNG. Predpokladajme, že máme z výstupu TRNG k dispozícii veľké číslo s dostatočným počtom bitov, napr. číslo

```
X0 := 943212223451234512566719306754453795671239112539854759378262949572\
94028304 6100190,
```

ktoré je párne, takže určite nie je prvočíslom. Keďže prvočísel je nekonečne veľa, s využitím funkcie `IsPrime()` nájdeme vhodné prvočíslo napr. tak, že budeme postupne testovať všetky nepárne čísla $P \geqq X0$. Hľadané prvočíslo nájdeme napr. použitím nasledovných príkazov:

```
> X0 := 943212223451234512566719306754453795671239112539854759378262949572\
> 940283046100190;
> P := X0 + 1;
> cnt := 0;
> repeat P:=P+2;cnt:=cnt+1;
repeat > until IsPrime(P) and (P mod 4) eq 3;
> cnt;
146
> P;
943212223451234512566719306754453795671239112539854759378262949572940283 04\
6100483.
```

Po 146 pokusoch dostávame výsledné prvočíslo. Doba výpočtu v kalkulačke Magma bola 0,5 sekundy. V plnej verzii programu Magma môžeme testovať aj čísla uložené v externých súboroch a do súborov zapisovať výsledky. Tieto vlastnosti jasne potvrdzujú, že pomocou programu Magma môžeme v spojení s externým TRNG generovať parametre aj pre reálne kryptografické aplikácie.

Generátor RSA

Aj PRNG RSA je založený na probléme faktorizácie čísla n na prvočíselný rozklad a je opísaný algoritmom 6.2. Podobne ako u PRNG BBS, je rýchlosť tohto PRNG výrazne nižšia, ako rýchlosť lineárneho kongruentného PRNG. Aj na jeho realizáciu môžeme využiť funkcie pre modulárnu aritmetiku s MP číslami.

Algoritmus 6.2 PRNG RSA

Input: dve veľké tajné prvočísla p a q, vypočítaj $n = pq$ a $\phi = (p-1)(q-1)$. Zvoľ náhodné celé číslo e, $1 < e < \phi$, také, že gcd(e,ϕ)=1

Output: generovaná pseudonáhodná bitová postupnosť $b_1, b_2, \ldots, b_l$ dĺžky l

1: zvoľ náhodné celé číslo X_0 (seed) z intervalu $[1, n-1]$
2: **for** i from 1 to l **do**
3: $\quad X_i \leftarrow X_{i-1}^e \bmod n$
4: $\quad b_i \leftarrow$ najnižší významový bit čísla X_i
5: generovaná postupnosť je $b_1, b_2, \ldots, b_l$

---

*Doplnkový materiál k RSA generátoru

Príklad 6.2 Pomocou kalkulačky Magma vygenerujme modul n vhodný pre použitie v generátore RSA s veľkosťou $n \approx 1024$ bitov a vhodný exponent e. Pre navrhnuté parametre n, e a hodnotu RSA_SEED=123456 vygenerujme pomocou PRNG RSA 100 výstupných pseudonáhodných bitov.

Riešenie. Zvolíme exponent e. Z hľadiska rýchlosti generovania je výhodné voliť čo najmenšiu hodnotu e, ktorá obsahujúce v binárnom rozvoji minimálny počet jednotiek. Zvolíme napr. druhé Fermatovo číslo $e = F_2 = 2^{2^2} + 1 = 17 = (10001)_2$. Pomocou nasledovných príkazov vygenerujeme požadované pseudonáhodné bity:

```
> E := 17;
> N := RSAModulus(1024, E);
> N;
109151186012559026525800422760694529300957321645487422776196708360434318825\
791592541750307711588456130079186542594017668188501688428710149761815759805\
374934389177655630149652156072324002430266222044897954792995083749300313\
284055386521725606621089257983063510290807561914289191925657379116297323242\
520421214661
> RSA_SEED := 123456;
> b := RandomSequenceRSA(N, E, RSA_SEED, 100);
> b;
[ 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0,
1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0,
0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
1 ].
```

Testovanie prvočíselnosti pre veľké čísla

Mnohé moderné kryptografické algoritmy a protokoly vyžadujú prvočísla. Aby sme si vytvorili predstavu o ich počte uveďme niektoré fakty. Odhaduje sa, že vo vesmíre je približne 10^{77} atómov. Existuje približne 10^{151} prvočísel dĺžky 512 bitov (čo je v praktických kryptografických aplikáciách bežne využívaná veľkosť[6]) alebo kratších.

Pravdepodobnosť, že náhodne zvolené číslo v blízkosti čísla n bude prvočíslo je približne 1 ku $\ln n$. Takže celkový počet prvočísel menších než n bude približne

$$\frac{n}{\ln n} \tag{1.45}$$

V moderných algoritmoch sa často vyskytujú dva typy otázok. Prvá: „ *Je číslo n prvočíslo*?“ je **neporovnateľne jednoduchšia** ako druhá: „*Aké prvočísla sú rozkladom čísla n*?“ Práve tento výrazný rozdiel v zložitosti odpovedí na tieto otázky je základom pre významné kryptografické algoritmy.

Pri generovaní prvočísel sa vychádza z (relatívnej) jednoduchosti odpovede na prvú otázku. Náhodne sa vyberie číslo n a niektorým zo známych testov prvočíselnosti sa zistí sa, či je prvočíslom. Pokiaľ sa zistí, že číslo n nie je prvočíslo, je možné zvoliť ďalšie náhodné číslo a test opakovať. Iná stratégia výberu napr. využíva voľbu najbližšieho[7] nepárneho čísla $n+2$ a opakovanie testu prvočíselnosti.

Na testovanie prvočíselnosti je možné využiť niektorý zo známych testov (napr. Solovay-Strassenov, Lehmannov, Rabin-Millerov a pod.).

---

[6] Tieto čísla jasne dokumentujú vysokú abstrakciu matematiky. Počet atómov vo vesmíre je pre mnohých ľudí ťažko predstaviteľná hodnota. Na druhej strane počítanie s 512 bitovými (a aj s podstatne väčšími!) číslami je s využitím bežne dostupných technických prostriedkov v podstate len vec rutiny. Naviac matematici pracujú s takýmito číslami často aj bez technických prostriedkov.

[7] Vzhľadom na veľkosť množiny prvočísel narazíme na prvočíslo veľmi rýchlo.

Základná klasifikácia testov prvočíselnosti

- **deterministické**
  - pre špeciálne čísla (LLT pre Mersennove čísla, …)
  - pre všeobecné čísla (AKT, ECPP, …)
- **pravdepodobnostné** (Lehmannov, Rabin-Millerov, …)

Poznámky:

AKT algoritmus (Agrawal-Kayal-Saxena primality test) – deterministický algoritmus pre všeobecné čísla **s polynomiálnou zložitosťou** (!!!), navrhnutý v roku 2002. Z teoretického pohľadu veľmi významný objav, z pohľadu rýchlosti však pre typické čísla používané v kryptografii významne pomalší ako praktické pravdepodobnostné testy. Ďalšie informácie – viď. napr. https://en.wikipedia.org/wiki/AKS_primality_test

ECPP algoritmus (Elliptic Curve Primality Prooving) – deterministický algoritmus pre všeobecné čísla. V súčasnosti jeden z najrýchlejších (pre praktické výpočty testovania veľkých prvočísel nad rámcec toho, čo používame v súčasnej kryptografii) a najpoužívanejších deterministických testov. Ďalšie informácie – viď. napr. https://en.wikipedia.org/wiki/Elliptic_curve_primality

LLT (Lucas-Lehmer Test) – deterministický algoritmus pre Mersennove čísla (čísla v tvare M=2^p-1). Ďalšie informácie – viď. napr. https://en.wikipedia.org/wiki/Lucas%E2%80%93Lehmer_primality_test

Koncepčným základom pre pravdepodobnostné testy je malá **Frematova veta**. Napr. **Frematov pravdepodobnostný tes**t je jej priamym využitím:

Algorithm 2.1: Fermat's test for primality

for $i = 0$ **to** $k - 1$ **do**
  Pick $a \in [2, ..., n - 1]$.
  $b \leftarrow a^{n-1} \mod n$.
  if $b \neq 1$ **then return** (Composite, a).
return "Probably Prime".

Lehmannov test

Tento jednoduchší test prvočíselnosti čísla p je realizovaný pomocou nasledujúcich krokov:

1) vyberieme náhodne číslo a menšie než p
2) vypočítame $a^{(p-1)/2} \bmod p$
3) ak je $a^{(p-1)/2} \not\equiv 1 \pmod p$ alebo $a^{(p-1)/2} \not\equiv -1 \pmod p$, potom p určite nie je prvočíslo
4) ak je $a^{(p-1)/2} \equiv 1 \pmod p$ alebo $a^{(p-1)/2} \equiv -1 \pmod p$, potom pravdepodobnosť toho, že p nie je prvočíslo nebude väčšia než 50%

Tento test zopakujeme t-krát (samozrejme vždy s iným náhodne vybraným číslom a). Ak bude test úspešný t-krát, potom **pravdepodobnosť** toho, že p nebude prvočíslom bude $1/2^t$.

Algorithm 1.2.1 (Lehmann's Primality Test)

INPUT: Odd integer $n \geq 3$, integer $\ell \geq 2$.
METHOD:

```
0     a, c: integer; b[1..ℓ]: array of integer;
1     for i from 1 to ℓ do
2         a ← a randomly chosen element of {1, ..., n − 1};
3         c ← a^((n−1)/2) mod n;
4         if c ∉ {1, n − 1}
5             then return 1;
6             else b[i] ← c;
7     if b[1] = ··· = b[ℓ] = 1
8         then return 1;
9         else return 0;
```

Rabin-Millerov test

Tento veľmi jednoduchý test je v podstate zjednodušenou formou algoritmu doporučovaného normou pre digitálne podpisy. Pre náhodne zvolené číslo p spočítame číslo b, pričom b je počet delení dvojčlenu $(p-1)$ číslom 2. (t.j. 2^b je najväčšia mocnina čísla 2, ktorá delí $(p-1)$). Potom určíme také m, pre ktoré bude platiť $p = 1 + 2^b * m$. Ďalej postupujeme pomocou nasledujúcich krokov:

1) zvolíme náhodné číslo a, tak aby platilo $a < p$
2) položíme $j = 0$ a $z = a^m \bmod p$
3) ak bude $z = 1$ alebo $z = p - 1$, potom p testom prejde a môže byť prvočíslom
4) ak bude $j > 0$ a $z = 1$, potom p prvočíslom nebude
5) položíme $j = j + 1$, ak bude $j < b$ a $z \neq p - 1$, položíme $z = z^2 \bmod p$ a vrátime sa ku kroku (4); ak bude $z = p - 1$, tak p testom prejde a môže byť prvočíslom
6) ak bude $j = b$ a $z \neq p - 1$, potom p nie je prvočíslom.

Algorithm 2.2: Miller–Rabin algorithm

```
Write n − 1 = 2^s · m, with m odd.
for j = 0 to k − 1 do
    Pick a ∈ [2, ..., n − 2].
    b ← a^m mod n.
    if b ≠ 1 and b ≠ (n − 1) then
        i ← 1.
        while i < s and b ≠ (n − 1) do
            b ← b^2 mod n.
            if b = 1 then return (Composite, a).
            i ← i + 1.
        if b ≠ (n − 1) then return (Composite, a).
return "Probable Prime".
```

Pravdepodobnosť toho, že testom prejde ako prvočíslo zložené číslo klesá v tomto teste rýchlejšie ako v predchádzajúcich a je rovná hodnote $1/4^t$, pričom t je počet iterácií.

Prvočísla generované uvedenými pravdepodobnostnými testmi sa niekedy označujú tiež termínom **prvočísla priemyselnej kvality**. V niektorých kryptografických algoritmoch (napr. RSA) sa využíva číslo n, ktoré je súčinom dvoch veľkých prvočísel p a q. Niekedy sa od týchto čísel vyžaduje, aby to boli tzv. **silné prvočísla** (strong primes). Tieto prvočísla majú určité vlastnosti, ktoré sťažujú rozklad čísla n na prvočinitele s využitím špecifických postupov. Medzi doporučované vlastnosti patria najmä tieto:

- najväčší spoločný deliteľ čísel $(p-1)$ a $(q-1)$ má byť malý
- obe čísla $(p-1)$ a $(q-1)$ majú mať veľké prvočiniteľe p' a q'
- obe čísla $(p'-1)$ a $(q'-1)$ majú mať veľké prvočiniteľe
- obe čísla $(p'+1)$ a $(q'+1)$ majú mať veľké prvočiniteľe
- obidva čísla $(p-1)/2$ a $(q-1)/2$ majú byť prvočísla (táto podmienka zabezpečuje zároveň splnenie prvých dvoch podmienok).

PRNG s využitím hašovacej funkcie

Základný princíp

Jednocestnosť hašovacej funkcie umožňuje jednoduchým spôsobom realizovať PRNG. Možností konfigurácie je viacero, základom je spätná väzba z výstupu hašovacej funkcie späť na jej vstup s prípadným „zmiešaním“ napr. s počítadlom, napríklad takto (je to len principiálny pseudokód):

```
Initialization: state = hash(seed || cnt);
Iteration: output = state; ++cnt;
Reseeding: state = hash(state || cnt);
```

Na stránke:
https://www.stat.berkeley.edu/~stark/Java/Html/sha256Rand.htm

je príklad implementácie podobne vytvoreného PRNG.

Základné výhody PRNG na báze hašovacej funkcie sú:

- Implementácie hašovacích funkcií (ako napr. SHA-256) **sú dostupné** v mnohých programovacích jazykoch.
- Zadaním **správnej hodnoty** „seed” môže **ktokoľvek overiť**, že postupnosť čísel bola generovaná pomocou tohto typu PRNG,
- **Bez znalosti** hodnoty „seed” je postupnosť generovaných čísel pre vonkajšieho pozorovateľa **veľmi ťažko rozlíšiteľná** od nezávislej, rovnomerne rozloženej postupnosti náhodných čísel.

PRNG na báze hašovacích funkcií sú aj súčasťou štandardov, pozri napr. Hash_DRBG (HAsh Deterministic Random Bit Generator) v nasledujúcich dokumentoch:

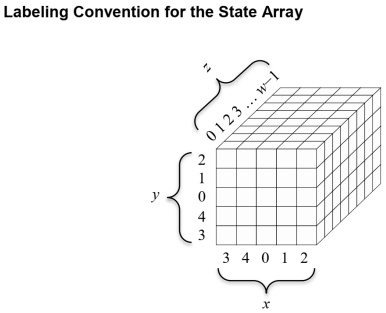
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf

https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/Hash_DRBG.pdf

https://github.com/greendow/Hash-DRBG

Z praktického pohľadu sú zaujímavé SHA3 varianty s voliteľnou veľkosťou výstupu spomenuté na minulej prednáške – **SHAKE128** a **SHAKE256**. Sú priamo štandardizované, pomerne rýchle a je ich možné použiť ako **kryptograficky kvalitné PRNG**.

Ich štruktúra je založená na **špongiovej konštrukcii** (**sponge construcion**) https://en.wikipedia.org/wiki/SHA-3, ktorá využíva **stav S** s veľkosťou **1600 bitov** (5 x 5 x 64)



a permutačnej funkcii f. **Permutačná funkcia f** je pomerne komplikovaná transformácia, ktorej úlohou je **„dokonale premiešať“** všetky bity stavu S.

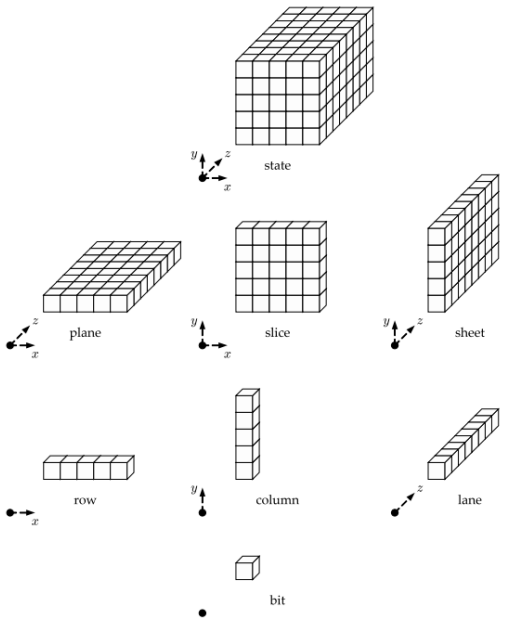


Figure 1: Parts of the state array, organized by dimension [8]

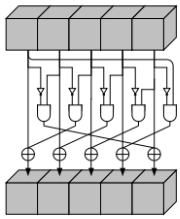


Figure 6: Illustration of χ applied to a single row [8]

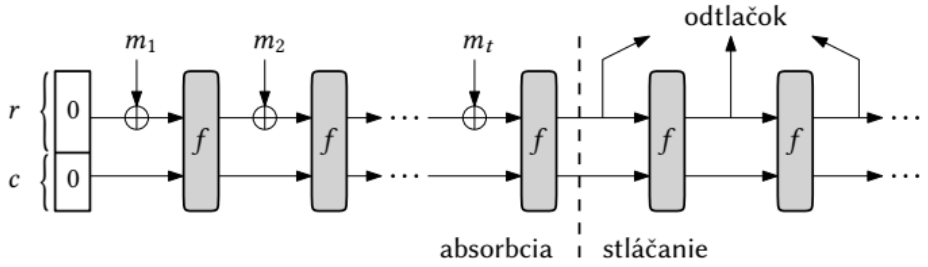
Detaily permutačnej funkcie f sú špecifikované v norme:
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

Pomerne prehľadný **pseudokód permutačnej transformácie** f a všeobecnej funkcie SHA3 (vrátane SHAKE128 a SHAKE256) sú dostupné na stránke autorov Keccak algoritmu:

https://keccak.team/keccak_specs_summary.html

Nasledujúci opis a **zjednodušený obrázok** je **prevzatý** z učebného textu (staršej, aktuálne voľne nedostupnej verzie):
Kryptológia - základy (http://www.dcs.fmph.uniba.sk/~stanek/K2_v2b.pdf) – aktuálny online dostupný učebný text o kryptológii využívaný na Katedre informatiky FMFI UK, autor **M. Stanek** (http://www.dcs.fmph.uniba.sk/~stanek/#vyuka), linku som dal aj www stránku predmetu.

V rokoch 2007 až 2012 prebehol verejný výber nového štandardu pre sadu hašovacích funkcií SHA-3. Víťazným algoritmom sa stal Keccak. Štandardizované by mali byť (podľa [68]) 4 hašovacie funkcie – SHA3-224, SHA3-256, SHA3-384, SHA3-512 a 2 funkcie s rozšíriteľným výstupom (angl. XOF – extendable-output function) – SHAKE128 a SHAKE256, ktoré umožňujú zvoliť potrebnú dĺžku výstupu. Z praktického hľadiska je dôležité spomenúť, že sa nepredpokladá migrácia z SHA-2 na SHA-3, ale vzájomná koexistencia týchto sád hašovacích funkcií. Návrh SHA-3 funkcií využíva tzv. špongiovú konštrukciu (angl. sponge), pozri obrázok 6.3. Táto konštrukcia využíva dve fázy, v prvej sa postupne do vnútorného stavu absorbujú bloky vstupného textu, v druhej sa vnútorný stav „stláča“ a formuje sa z neho výsledný odtlačok.



Obr. 6.3: Špongiová konštrukcia

Špongiová konštrukcia má parametre r (rýchlosť), c (kapacita). Rýchlosť r označuje veľkosť bloku vstupného textu, ktorý spracujeme v jednej iterácii. Hodnota $b = r + c$ sa nazýva šírka funkcie f a zodpovedá veľkosti vnútorného stavu v bitoch. Na začiatku je vnútorný stav inicializovaný na konštantnú hodnotu, napr. 0^b. Vstupný text sa najskôr zarovná tak, aby bola jeho dĺžka násobkom r. Na obrázku 6.3 sú bloky textu (po zarovnaní) označené $m_1, m_2, \ldots, m_t$. Funkcia f zobrazuje b-bitové vektory na b-bitové vektory. Pri stláčaní opätovne aplikujeme funkciu f kým nezískame požadovaný počet výstupných bitov odtlačku (z každej iterácie získame r bitov odtlačku).

Testovanie kvality náhodných generátorov

Na generátory náhodných čísel (RNG), realizované vo forme PRNG alebo TRNG sú kladené v kryptografii vysoké nároky. Typickou požiadavkou je, aby generovanú postupnosť nebolo možné odlíšiť od postupnosti generovanej z ideálneho zdroja náhodnosti. Vlastnosti ideálneho zdroja náhodnosti sú pritom jasne definované. **Ideálny RNG** generuje:

- postupnosť bitov, ktoré sú štatisticky nezávislé,
- generované bity sú rovnako pravdepodobné.

Uvedená, veľmi jednoduchá definícia ideálneho RNG, umožňuje používať na testovanie defektov v postupnostiach generovaných pomocou reálnych PRNG alebo TRNG tzv. **štatistické testy**. Pomocou štatistických testov môžeme zistiť či testovaná postupnosť bitov (konečnej dĺžky) má nejaké štatistické odchýlky oproti postupnosti bitov generovanej pomocou ideálneho RNG. Určite však nemôžeme štatistickými testami potvrdiť, že segment testovanej postupnosti je výstupom „ideálneho“ RNG. Dokonca existuje aj určitá malá pravdepodobnosť, že niektorý segment postupnosti generovaný ideálnym RNG určitým štatistickým testom nemusí byť správne vyhodnotený. Tieto súvislosti musíme pri interpretácií výsledkov štatistických testov vždy zobrať do úvahy.

Existuje viacero špecializovaných testovacích balíkov (napr. medzi najznámejšie patria [38], [39]), ktoré obsahujú viac ako 10 štatistických testov pomocou ktorých sa snažia defekty (odchýlky) v testovaných postupnostiach odhaliť. Niektoré z testov sú univerzálne, iné sú optimalizované na testovanie špecifických typov PRNG. Napríklad test lineárnej zložitosti (angl. linear complexity test) zisťuje, či generovaná postupnosť nebola vytvorená pomocou LFSR. Špecializované balíky štatistických testov často vyžadujú na realizáciu kompletného testu pomerne veľké množstvo dát (minimálne desiatky MB, často aj podstatne viac). Medzi najjednoduchšie základné štatistické testy patria frekvenčný test (angl. frequency test, monobit test), sériový test (angl. serial test, two-bit test), pokerový test (angl. poker test), test rovnakých reťazcov (angl. runs test) a autokorelačný test [14].

6.3.1 Frekvenčný test

Testom kontrolujeme, či je počet výskytu hodnôt 1 a 0 v testovanej postupnosti $s = \{s_1 s_2 \dots s_n\}$ približne zhodný tak, ako by to bolo v prípade postupnosti generovanej ideálnym RNG. Ak označíme počet 0 a počet 1 v testovanej postupnosti s ako n_0 a n_1, testovanou štatistikou je náhodná premenná

$$X_1 = \frac{(n_0 - n_1)^2}{n}, \tag{6.3}$$

ktorá pre dostatočne veľké n má χ^2 rozdelenie s 1 stupňom voľnosti.

6.3.2 Pokerový test

Nech m je kladné celé číslo pre ktoré platí $\lfloor \frac{n}{m} \rfloor \geqq 5\,(2^m)$, a nech $k = \lfloor \frac{n}{m} \rfloor$, pričom n je dĺžka testovanej postupnosti $s = \{s_1 s_2 \dots s_n\}$. Rozdeľme postupnosť s na k neprekrývajúcich sa reťazcov, z ktorých má každý dĺžku m bitov. Nech n_i označuje počet výskytov i-teho reťazca dĺžky m bitov, takže $1 \leqq i \leqq 2^m$. Pokerovým testom kontrolujeme, či reťazce dĺžky m bitov sa v postupnosti s vyskytujú približne rovnako často tak, ako by to bolo v prípade postupnosti generovanej ideálnym RNG. Testovanou štatistikou je náhodná premenná

$$X_2 = \frac{2^m}{k} \left(\sum_{i=1}^{2^m} n_i^2 \right) - k, \tag{6.4}$$

ktorá pre dostatočne veľké n má χ^2 rozdelenie s $2^m - 1$ stupňami voľnosti. Pokerový test je zovšeobecnením frekvenčného testu, ktorý z pokerového testu dostaneme nastavením $m = 1$.

6.3.3 Sériový test

Testom kontrolujeme, či je počet výskytu reťazcov 00, 01, 10 a 11 v testovanej postupnosti $s = \{s_1 s_2 \dots s_n\}$ približne zhodný tak, ako by to bolo v prípade postupnosti generovanej ideálnym RNG. Ak označíme počet 0 a počet 1 v testovanej postupnosti s ako n_0 a n_1, podobne počet výskytov reťazcov 00, 01, 10, 11 v testovanej postupnosti ako n_{00}, n_{01}, n_{10}, n_{11}, platí $n_{00} + n_{01} + n_{10} + n_{11} = (n-1)$, keďže reťazce sa môžu prekrývať. Testovanou štatistikou je náhodná premenná

$$X_3 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1, \tag{6.5}$$

ktorá pre dostatočne veľké n má χ^2 rozdelenie s 2 stupňami voľnosti.

6.3.4 Test rovnakých reťazcov

Testom kontrolujeme, či počet reťazcov rovnakých znakov (bitov) rôznej dĺžky v postupnosti $s = \{s_1 s_2 \dots s_n\}$ je približne zhodný tak, ako by to bolo v prípade postupnosti generovanej ideálnym RNG. Očakávaný počet reťazcov $11\dots1$ a $00\dots0$ dĺžky i v postupnosti $s = \{s_1 s_2 \dots s_n\}$ dĺžky n je $e_i = (n-i-3)/2^{i+2}$. Nech k je rovné najväčšiemu celému číslu i, pre ktoré platí $e_i \geqq 5$. Nech B_i (G_i) je počet reťazcov s dĺžkou i, ktoré sú zložené zo samých jednotiek (núl) v postupnosti s, pričom platí $1 \leqq i \leqq k$. Testovanou štatistikou je náhodná premenná

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}, \tag{6.6}$$

ktorá pre dostatočne veľké n má χ^2 rozdelenie s $2k-2$ stupňami voľnosti.

6.3.5 Autokorelačný test

Týmto testom kontrolujeme koreláciu medzi postupnosťou $s = \{s_1 s_2 \dots s_n\}$ a (necyklicky) posunutou verziou tej istej postupnosti. Nech d je celé číslo, pre ktoré platí $1 \leqq d \leqq \lfloor \frac{n}{2} \rfloor$. Počet bitov postupnosti s, v ktorých sa líši od jej od d bitov posunutej verzie je $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$. Testovanou štatistikou je náhodná premenná

$$X_5 = 2\left(A(d) - \frac{n-d}{2}\right)/\sqrt{n-d}, \tag{6.7}$$

ktorá pre dostatočne veľké hodnoty $(n-d)$ má $N(0,1)$ rozdelenie. Keďže malé hodnoty $A(d)$ sú rovnako nepravdepodobné ako veľké hodnoty $A(d)$, pri vyhodnocovaní musíme použiť dvojstranný test.

6.3.6 Štatistické testy FIPS

Aj keď vzťahy (6.3)–(6.7) pôsobia z technického hľadiska „značne neprakticky“, sú základom na praktické využívanie. Tri z týchto testov boli súčasťou normy FIPS 140-2 [40]. Ich parametre (relatívne malá veľkosť testovanej množiny a pevne prednastavené hodnoty prahov pre jednotlivé testy) boli nastavené tak, aby sa dali potenciálne použiť aj vo vstavaných kryptografických zariadeniach.

V norme [40] je pevne nastavená dĺžka testovanej postupnosti na $n = 20\,000$ bitov a sú určené prahy pre jednotlivé testy pomocou ktorých je možné jednoducho rozhodnúť, či testovaná postupnosť $s = \{s_1 s_2 \dots s_{20000}\}$ vykazuje štatisticky významné odchýlky od postupnosti generovanej ideálnym RNG. Prahy sú spočítané pre určitú hladinu štatistickej významnosti tak, že napr. 99 %[5] ideálnych postupností bude testom vyhodnotených ako správne. Zvyšok (1 %) bude testom vyhodnocovaný chybne.

[5] Uvedené percento je možné pri určovaní prahov vo všeobecnosti meniť, v norme [40] je však použitá pevná hodnota.

Frekvenčný test FIPS140-2 (angl. monobit test)

1. Spočítaj počet jednotiek v testovanej postupnosti $s = \{s_1 s_2 \dots s_{20000}\}$. Označ ich počet ako X_1.

2. Test je úspešný ak $9\,725 < X_1 < 10\,275$.

Pokerový test FIPS140-2 (angl. poker test)

1. Rozdeľ testovanú postupnosť $s = \{s_1 s_2 \dots s_{20000}\}$ na 5 000 segmentov dĺžky 4 bity, ktoré sa neprekrývajú. Spočítaj počet výskytu všetkých 16-tich kombinácií 4-bitových hodnôt. Označ ich počet ako $f(i)$, pre $0 \leqq i \leqq 15$.

2. Spočítaj hodnotu

$$X_2 = \frac{16}{5000} \left(\sum_{i=0}^{15} (f(i))^2 \right) - 5000.$$

3. Test je úspešný ak $2,16 < X_2 < 46,17$.

Test rovnakých reťazcov FIPS140-2 (angl. runs test)

1. Definujme rovnaký reťazec dĺžky L ako maximálnu postupnosť rovnakých hodnôt (samé jednotky alebo samé nuly), ktorá sa vyskytuje v testovanej postupnosti $s = \{s_1 s_2 \dots s_{20000}\}$. Spočítaj výskyty rovnakých reťazcov dĺžky L v testovanej postupnosti a ulož ich.

2. Test je úspešný, ak výskyty rovnakých reťazcov dĺžky $1 \leqq L \leqq 6$ sú v rámci limitov špecifikovaných v tab. 6.1. Podmienka musí byť splnená pre rovnaké reťaze núl aj jednotiek (t. j. pre všetkých 12 testovaných hodnôt). V teste sú všetky rovnaké reťazce dĺžky $L > 6$ započítavané ako reťazce dĺžky $L = 6$.

Tabuľka 6.1: FIPS 140-2 limity pre test rovnakých reťazcov

L	Požadovaný interval
1	2 343–2 657
2	1 135–1 365
3	542–708
4	251–373
5	111–201
6+	111–201

Test rovnakých dlhých reťazcov FIPS140-2 (angl. long runs test)

1. Definujme dlhý reťazec ako súvislú postupnosť 26 a viac rovnakých hodnôt (samé jednotky alebo samé nuly).

2. Test je úspešný, ak sa v postupnosti $s = \{s_1 s_2 \dots s_{20000}\}$ dlhý reťazec nevyskytuje.

Na www stránke [4] je dostupný kód FIPS 140-2 (Federal Information Processing Standards) testov, ktorý implementuje všetky testy FIPS 140-2 v jazyku C. V zdrojovom kóde môžeme pomocou direktívy `#define` zvoliť jednu z troch verzií prahov, ktoré boli v norme FIPS 140-2 priebežne v rôznych vydaniach modifikované. Je zaujímavé, že sa tieto prahy priebežne v novších verziách menili a niektoré zmeny boli dokonca chybné. Naznačuje to, že problematika štatistických testov RNG nemusí byť celkom triviálna záležitosť. Na druhej strane z technického hľadiska je pomerne ľahko využiteľná.

4. DRUTAROVSKÝ, Miloš. *Kryptografia pre vstavané procesorové systémy: doplňujúce materiály a zdrojové kódy* [online] [cit. 2017-09-01]. Dostupné z: http://aplikovanakryptografia.fei.tuke.sk/

Štatistické testy NIST

V prípade potreby realizovať náročnejšie testy PRNG resp. TRNG môžeme využiť špecializované štatistické balíky. V súčasnosti patrí medzi najkvalitnejšie a najlepšie dokumentované verejne dostupné balíky aj súbor štatistických testov NIST (National Institute of Standards and Technology) [38]. Tento softvérový balík je poskytovaný v zdrojových kódoch v jazyku C [41] a je preto modifikovateľný aj koncovým užívateľom. Tento balík, okrem pomerne detailného opisu testovacích metód, poskytuje aj opis základnej metodiky na testovanie náhodných a pseudonáhodných generátorov. Využíva sa pre off-line testovanie RNG aj pri vývoji vstavaných systémov, vzhľadom na jeho funkcionalitu však nie je určený pre implementáciu vo vstavaných systémoch.

NIST aktuálne pracuje aj na špecifikácii testov vhodných pre testovanie zdrojov entropie použitých v TRNG implementáciách [42]. Tieto typy testov je možné využiť v on-line testovaní fyzikálnych zdrojov náhodnosti, ktoré sú využívané aj vo vstavaných systémoch. Podobné testy existujú aj v iných štandardoch, napr. v nemeckom AIS-31 [43].

Využitie náhodných čísel v štandarde PKCS #1

Cieľom nasledujúceho opisu je poukázať na nutnosť použitie náhodných dát v reálnej aplikácii - algoritme RSA. Zároveň budú využité aj hašovacie funkcie.

Algoritmus RSA tak, ako bol opísaný v jednej z predchádzajúcich prednášok, využíva pomerne **jednoduchú matematickú operáciu** – modulárne umocnenie. RSA tak má pomerne **jednoduchú algebraickú štruktúru**, ktorá sa tiež zvykne označovať ako „**Textbook RSA“**.

Takto využitý RSA nie je z pohľadu kryptografie bezpečný, pozri napr.
https://crypto.stackexchange.com/questions/1448/definition-of-textbook-rsa

Základný problém je v tom, že ak máme k dispozícii zašifrovaný text *C*, **vieme aj bez znalosti otvoreného textu** *M* vytvoriť **platný zašifrovaný text**

C‘= *C**2^*e* mod *n*,

ktorý zodpovedá správe 2*M* mod *n* .

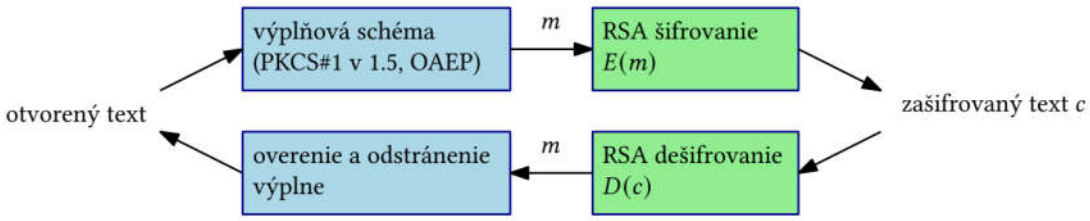
Skutočnosť, že vieme **modifikovať platné** šifrované dáta tak, že vzniknú **iné platné** šifrované dáta je možné veľmi jednoducho využiť **pri útokoch aktívnych útokoch** na „Textbook RSA“.

RSA sa preto **v praxi využíva** s vhodne definovaným algoritmom **vypĺňania** (**padding**) šifrovaných dát. Doporučované algoritmy sú súčasťou štandardov **PKCS** (**Public Key Cryptography Standards**) publikovaných firmou RSA. Výplňová schéma pre algoritmus RSA je súčasťou štandardu PKCS #1, aktuálne vo verzii 2.2 (pozri tiež RFC 8017):

http://mpqs.free.fr/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp_EMC_Corporation_Public-Key_Cryptography_Standards_(PKCS).pdf

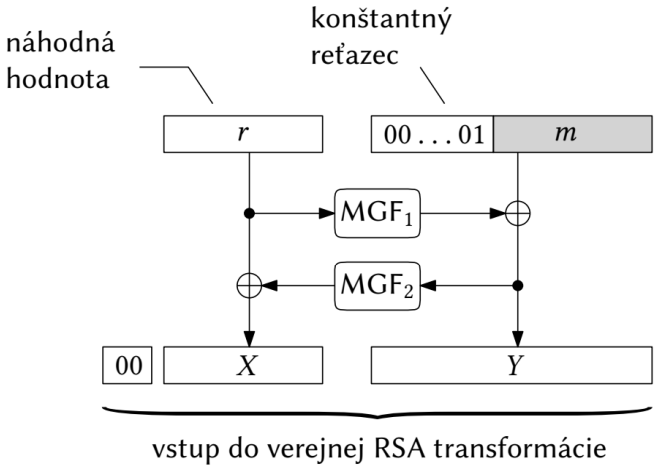
https://tools.ietf.org/html/rfc8017

Nasledujúci opis a **zjednodušený obrázok** je **prevzatý** z učebného textu (staršej, aktuálne nedostupnej verzie):
Kryptológia - základy (http://www.dcs.fmph.uniba.sk/~stanek/K2_v2b.pdf) - učebný text o kryptológii využívaný na Katedre informatiky FMFI UK, autor **M. Stanek.**



Obr. 6: Použitie RSA s výplňovou schémou

Napr. s RSA je doporučované využívať výplňovú schému **OAEP** (**Optimal Asymmetric Encryption Padding**), definovaná napr. v PKCS#1 v2.2), ktorá využíva pri spracovaní otvoreného textu kryptografické hašovacie funkcie. Z týchto sú konštruované **MGF** (**Mask Generation Function**), ktoré modelujú náhodné funkcie. Mierne zjednodušená OAEP je znázornená na obrázku 9.2.



Obr. 9.2: Výplňová schéma OAEP

Postup pri spracovaní vstupu m je nasledovný:

1. Zvolíme pseudonáhodný reťazec r a samostatne zreťazíme konštantný reťazec $00\ldots01$ s m, pričom r aj konštantný reťazec majú dostatočnú, vopred definovanú dĺžku.
2. Vypočítame $Y = \mathrm{MGF}_1(r) \oplus (00\ldots01 \,||\, m)$ a $X = \mathrm{MGF}_2(Y) \oplus r$, pričom MGF_1 a MGF_2 majú potrebnú dĺžku výstupu.
3. Výstupom je reťazec $00 \,||\, X \,||\, Y$, kde výstup má rovnakú dĺžku ako je dĺžka verejného modulu n v RSA. Nulový bajt na začiatku zaručuje, že výstup OAEP je zo $\mathbb{Z}_n$.

Výstup OAEP je následne spracovaný verejnou RSA transformáciou. Postup dešifrovania v RSA-OAEP je nasledovný:

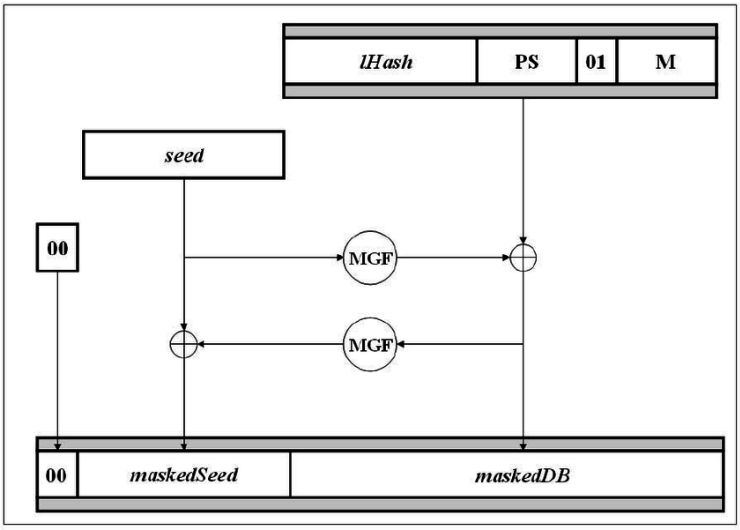
1. Šifrový text dešifrujeme s použitím súkromnej RSA transformácie a získaný medzivýsledok rozdelíme na časti X a Y (po kontrole nulového úvodného bajtu).
2. Vypočítame $r' = X \oplus \mathrm{MGF}_2(Y)$ a hodnotu $w = \mathrm{MGF}_1(r') \oplus Y$, pričom skontrolujeme zhodu začiatku reťazca w s konštantným reťazcom $00 \ldots 01$. Pokiaľ je kontrola úspešná, zvyšok reťazca m je otvorený text. V prípade, že je ktorákoľvek kontrola neúspešná, dešifrovanie skončí neúspechom – šifrový text je nekorektný.

Samozrejme, platný šifrový text je možné skonštruovať zašifrovaním nejakého otvoreného textu. Hlavná myšlienka a bezpečnostná črta OAEP je však v tom, že útočník nevie efektívne skonštruovať taký šifrový text, ktorý by bol platný – teda prešiel kontrolami pri dešifrovaní RSA-OAEP bez toho, aby poznal zároveň príslušný otvorený text (pravdepodobnosť zhody s reťazcom $00 \ldots 01$ po dešifrovaní je zanedbateľná). To znamená, že prístup k dešifrovaniu, ako napr. v IND-CCA2, je pre útočníka zbytočný a nedokáže ho zmysluplne využiť.

Poznámky:

- Opísaný algoritmus využíva **Feistelovu štruktúru**, ktorá je ľahko použiteľná aj v **spätnom smere**, teda pri RSA dešifrovaní.
- Náhodná hodnota *r* je **generovaná** pomocou vhodného **RNG.**
- **Konštantný reťazec 00...01** slúži na **kontrolu integrity** dešifrovanej správy *m.*
- Aj takto definovanú výplňovú schému je, **pri zlej implementácii**, možné využiť na **špecializované útoky**. Táto skutočnosť je už dlhodobo **známa** (pozri napr. diskusiu v http://crypto-world.info/klima/2001/chip-2001-11-172-175.pdf). Podrobnejšie o tejto problematike budeme rozprávať na konci prednášok.
- Obr. 9.2 **je zjednodušený**, štandard PKCS #1 využíva štruktúru výplňovej schémy zobrazenej na obr.10, ktorá využíva na vytvorenie **konštantného reťazca** hašovaciu funkciu a jej výstup ***lHash*** . PS je nulový reťazec, 01 je oddeľovací bajt.
- implementácia **kvalitného (bezpečného) kódu** je, aj napriek existencii vhodných štandardov, pomerne **náročná úloha**. Preto v praxi veľmi často **využívame** vhodné **kryptografické knižnice**. Na cvičení sme ukázali, ako je možné

s využitím knižnice **OpenSSL** (https://www.openssl.org/) **jednoducho šifrovať** dáta s algoritmom **RSA-OAEP**.



Obr.10 Výplňová schéma podľa štandardu PKCS #1, **konštantný reťazec** je tvorený **spojením** *lHash* || PS || 01. Hodnota *lHash* je výstup hašovacej funkcie vstupného reťazca – Hash(*label).* Ak hodnota *label* nie je definovaná, použije sa Hash **prázdneho reťazca**.

Intel Digital Random Number Generator (DRNG)

Novšie procesory Intel obsahujú **hardvérovo realizované RNG**, ktorých výstupy sú dostupné vo forme **inštrukcií RSRAND a RDSEED**.

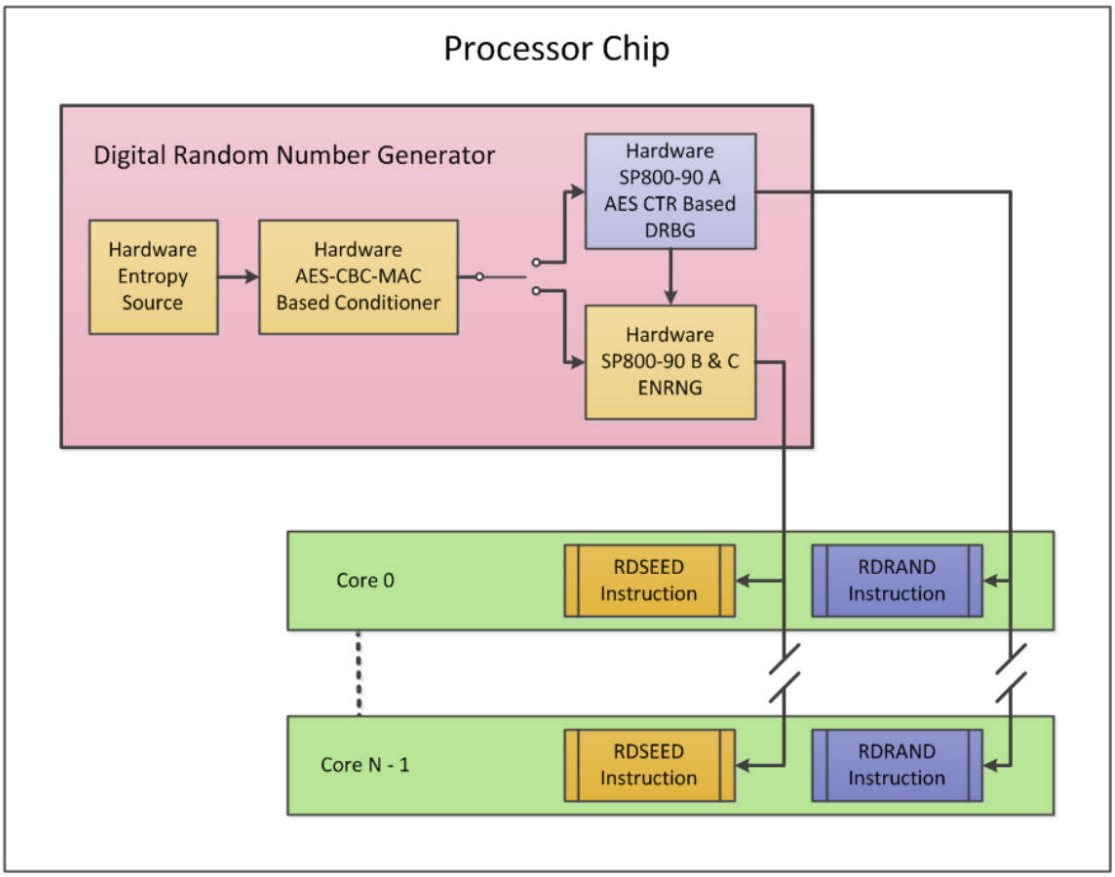


Figure 2. Digital Random Number Generator design

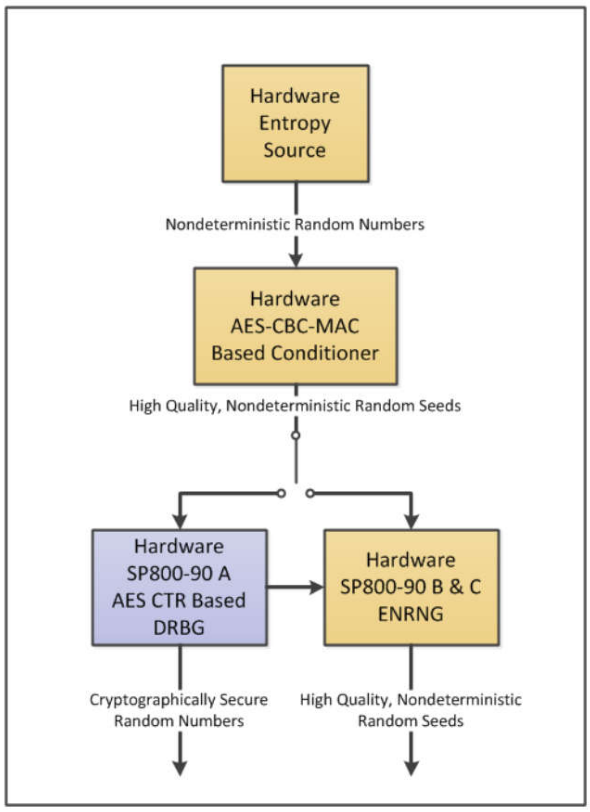


Figure 3. DRNG Component Architecture

The final two stages are:
1. A hardware CSPRNG that is based on **AES in CTR mode** and is compliant with SP800-90A. In SP800-90A terminology, this is referred to as a DRBG (Deterministic Random Bit Generator).
2. An ENRNG (Enhanced **Nondeterministic Random Number Generator**) that is compliant with SP800-90B and C.

Podrobnejšie informácie sú dostupné napr. v:
https://software.intel.com/sites/default/files/managed/98/4a/DRNG_Software_Implementation_Guide_2.1.pdf

Poznámka:
Na jednom z nasledujúcich cvičení využijeme **kód v jazyku C**, ktorý umožní otestovať **prítomnosť** inštrukcií **RSRAND a RDSEED** (niektoré staršie verzie procesorov podporujú len inštrukciu RSRAND, prípadne žiadnu) v konkrétnom **testovanom PC**.

Hardvérovo realizované RNG sú v súčasnosti často implementované aj v menších procesoroch (**mikrokontroléroch**), ktoré sú často využívané aj v **IoT zariadeniach**. Ako príklad môžeme uviesť blokovú schému populárneho čipu s **WiFi konektivitou ESP32** (viď. blok kryptografického HW akcelerátora).