ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ

Katedra aplikované elektroniky a telekomunikací

BAKALÁŘSKÁ PRÁCE

Generování náhodných čísel

Jan Novák **2016**

Originál (kopie) zadání BP/DP

Abstrakt

Tato bakalářská práce je zaměřena na generování náhodných a pseudonáhodných čísel v oblasti výpočetní techniky. Jsou zde vysvětleny principy a problémy generování náhodných čísel. Dále jsou v práci vysvětleny rozdíly mezi generátory pseudonáhodných čísel a generátory skutečně náhodných čísel. Několik vybraných generátorů z obou skupin je zde podrobněji popsáno. Část práce obsahuje popis metod, které lze použít pro hodnocení generátorů. Vybrané generátory pseudonáhodných čísel jsou v poslední části práce implementovány a testovány.

Klíčová slova

RNG, TRNG, PRNG, generátor náhodných čísel, generátor pseudonáhodných čísel, testování generátorů náhodných čísel, statistické testy, lineární kongruentní generátor, lineární zpětnovazební registr, permutovaný kongruentní generátor, mersenne twister

Abstract

Main subject of this bachelor thesis is generating random and pseudorandom numbers in computer technology. The principles and problems of random number generating are explained. Also the differences between pseudorandom number generators and true random number generators are cleared up. Few selected generators from both categories are discussed in detail. Part of this paper is also description of methods suitable for evaluation of random number generators. In the last section, selected generators are implemented and tested.

Key words

RNG, TRNG, PRNG, random number generator, preudorandom number generator, testing random number generators, statistical tests, linear congruential generator, linear feedback shift register, permuted congruential generator, mersenne twister

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této diplomové práce.

Dále prohlašuji, že veškerý software, použitý při řešení této bakalářské práce, je legální.

.....

podpis

V Plzni dne 1.6.2016 Jan Novák

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petru Burianovi Ph.D za cenné profesionální rady, připomínky a metodické vedení práce.

Obsah

Seznam symbolů a zkratek

TRNG ..... True Random Number Generator

PRNG ..... Pseudo Random Number Generator

CSPRNG..... Cryptographic Secure Pseudo Random Number Generator

LFSR..... Linear Feedback Shift Register

MT ..... Mersenne Twister

LCG ..... Linear Congruential Generator

PCG ..... Permuted Congruential Generator

Úvod

Náhodná čísla si člověk běžně spojí třeba s tažením sportky nebo hodem kostkou, díky rozšíření počítačových systémů, kde se hojně využívají, se ale staly nedílnou součástí našeho každodenního života. Tato práce se zabývá problémem generování náhodných čísel právě pro počítačové a obecně elektronické výpočetní systémy.

V první kapitole se seznámíme s pojmem náhodná čísla a s významem těchto čísel pro počítačovou techniku. Jmenujeme hlavní problém generování náhodných čísel a jeho řešením dospějeme ke dvěma používaným druhům generování. Oba tyto druhy, generování pseudonáhodných čísel a generování skutečně náhodných čísel rozebereme po stránce využitých principů a také po stránce jejich předností a nedostatků

Druhá a třetí část je zaměřena na popis generátorů obou druhů. Generátory pseudonáhodných čísel jsou rozebírány ve druhé kapitole a nejdříve se seznámíme s jejich implementací v běžných programovacích jazycích. Dále jsou podrobněji popsány čtyři konkrétní pseudonáhodné generátory. Mezi tuto čtveřici patří jak dva velice jednoduché a staré generátory, tak dva relativně nové a složitější. Ve třetí kapitole prozkoumáme generátory skutečně náhodných čísel. Najdeme několik generátorů zmíněných a dva konkrétní podrobněji popsané. Zjistíme, jak lze využít hardwarové i softwarové vlastnosti počítačových systémů v náš prospěch a získat skutečně náhodná data.

Čtvrtá kapitola je zaměřená na metody hodnocení generátorů náhodných čísel. Dozvíme se nejobjektivnější způsoby hodnocení těchto generátorů, ale najdeme zde také shrnutí ostatních vlastností, jež mohou být důležité při výběru správného generátoru pro konkrétní aplikaci.

V poslední kapitole pak můžeme nalézt jednoduché popisy implementací vybraných pseudonáhodných generátorů včetně úryvků jejich zdrojových kódů. Tyto implementace jsou dále testovány vybranými testy. Výsledky těchto testů jsou v kapitole stručně vypsány a jejich vyhodnocení nalezneme v závěru.

1 Náhodná čísla

Pod pojmem náhodné číslo si každý dokáže jistě něco představit, ale jasná definice náhodnosti je složitá. Existuje více výkladů náhodnosti, avšak každý na toho téma pohlíží z jiného pohledu. Teorie popisující tzv. subjektivní náhodnost naznačuje výskyt samotného konceptu náhodnosti pouze uvnitř lidské mysli. Tento výklad považuje za zdroj náhodnosti lidskou ignoranci a nevědomost, nikoliv vnější svět. S tímto výkladem souvisí další možnost, jak na náhodnost pohlížet - předpověditelnost. Pokud můžeme předpovědět následující čísla ze sekvence, kterou již máme, nemůže být celková sekvence náhodná. Podle tohoto výkladu můžeme prakticky každou sekvenci určit jako náhodnou, závisí pouze na vědomostech hodnotitele. Jestliže vezmeme člověka, který nezná násobilku a její koncept, může být dle jeho názoru označena např. řada obsahující násobky tří jako náhodná. [1, 2]

Teorie složitosti také přispívá k náhledu na náhodná čísla. Vezměme předpoklad, kdy je daný konečný počet objektů X a konečný počet popisů těchto objektů Y. Dále určíme D jako funkci přiřazující každému objektu x z množiny X právě jeden popis y z množiny Y, tedy $D(y) = x$. Každý objekt má tak svůj popis, tento popis určíme jako řetězec konečné délky. Popisná složitost objektu je potom délka řetězce, která je nutná pro plné popsání objektu. Pokud vezmeme za daný objekt sekvenci čísel a její popisná složitost je stejně dlouhá jako sekvence samotná, pak můžeme sekvenci označit jako náhodnou. Pro ilustraci porovnejme dvě bitové sekvence:

100100100100100100100100

Tato sekvence může být popsána jednoduše jako 8 kopií trojice bitů 100.

110101110001101110110100 1

Druhá sekvence nemá žádný očividný jednoduchý popis kromě sekvence samotné. Popisná složitost je tedy stejně dlouhá a můžeme sekvenci prohlásit za náhodnou. Tímto se ale vracíme zpět k míře vědomostí použitých k popisu, je totiž možné, že jednodušší, méně zřejmý popis existuje. [2, 3]

K popisu náhodnosti čísel se obecně využívá především statistika. Ta nám totiž poskytuje uchopitelné výsledky. Pomocí těchto výsledků můžeme určit, jak moc se námi

testovaná čísla blíží číslům, jež se dají označit za dokonale náhodná. Pokud mluvíme o náhodných číslech z pohledu statistiky, musíme dále rozlišovat dva případy. V prvním případě máme jednotlivá náhodná čísla, ve druhém celou sekvenci náhodných čísel. Jedno číslo je náhodné, pokud patří do množiny všech možných hodnot, z nichž mají všechny stejnou pravděpodobnost výskytu (rovnoměrné rozložení). Daná sekvence čísel je náhodná, pokud jsou všechna obsažená jednotlivá čísla vzájemně statisticky nezávislá. [4]

1.1 Využití náhodných čísel v počítačové technice

Náhodná čísla nalezla v různých oblastech počítačové techniky velmi široké využití. Ve vědeckých a výzkumných aplikacích jsou vyžadována pro vytváření vzorků z velkého množství dat i pro mnoho analytických metod datových zpracování. Pokud jsou dané problémy příliš složité pro konvenční řešení, nebo by takové řešení trvalo příliš dlouho, přibližné výsledky mohou být získány pomocí technik využívajících náhodná čísla (např. metody Monte Carlo [5]). Metody Monte Carlo jsou velmi využívané také pro simulace složitých zkoumaných jevů. Aby simulace byly realistické, je použití náhodných čísel nezbytné. Mezi takové simulace můžeme zařadit např. modely městské dopravy [6], vývoje ekonomiky nebo simulace galaktických systémů [7]. Klíčový význam mají náhodná čísla pro kryptografii [3], veškeré šifrování v počítačové technice je na nich závislé. Kryptografické aplikace jsou dnes nedílnou součástí každodenních lidských činností, ať už se jedná o komunikaci pomocí telefonu, komunikaci po internetu, případně přístup do internetového bankovnictví. U kryptografických aplikací je velmi důležitá nepředpovědítelnost použitých čísel. Se zvyšující se šancí na správné předpovězení použitých čísel se oslabuje celková bezpečnost systému. Své využití najdou náhodná čísla také v počítačových hrách, kde může být vysoká kvalita čísel velmi důležitá (např. pro online hazardní hry) [2].

1.2 Generování (pseudo)náhodných čísel

Při vývoji počítačových systémů se ukázalo jako nezbytné zahrnout možnost generování náhodných čísel. Problémem je však získání náhodnosti z principiálně plně deterministického systému. Generování náhodných čísel v počítačové technice se rozdělilo do dvou kategorií, generování skutečně náhodných čísel a generování pseudonáhodných čísel. Zatímco prvně jmenovaná kategorie generuje taková náhodná čísla, jaká si pod pojmem náhodná představíme, druhá kategorie generuje sekvence čísel, která náhodná ve

skutečnosti nejsou. Tyto generované sekvence jsou výsledkem deterministických výpočetních operací, které jsou pečlivě navrženy pro vytvoření sekvencí čísel co nejvíce připomínajících náhodné. Tyto dvě kategorie generování čísel pracují na zcela odlišných principech.

Generátory pseudonáhodných čísel (PRNG) jsou algoritmy běžící na nějaké elektronické platformě. Jedná se většinou o softwarová řešení (mohou být ale realizována i hardwarově, např. jednoduché generátory LFSR). Prakticky se skládají z několika částí: vnitřní stav (state), samotný výpočetní algoritmus a výstup. Z uloženého vnitřního stavu vypočítává algoritmus následující výstupní číslo a následující vnitřní stav. Tato čísla mohou být pro některé generátory shodná, tzn. algoritmus vypočítá následující výstupní číslo a uloží si ho jako nový vnitřní stav. Samotná podoba těchto částí velmi závisí na jednotlivých generátorech a jejich složitosti. Pro zahájení činnosti generátoru je potřeba zadat výchozí vnitřní stav, ze kterého se bude celá generovaná sekvence odvíjet, tzv. semínko (seed). Semínko je třeba vybírat s ohledem na vlastnosti algoritmu. Pokud je zvoleno špatně, i dobrý generátor podá nedostatečné výsledky. Naopak při vhodně zvoleném semínku může i jednoduchý generátor produkovat sekvence obstojné kvality. Generovaná čísla tedy nemohou být skutečně náhodná, neboť jsou přesně určena vnitřním stavem a výpočetním algoritmem. Při znalosti generátoru a několika jeho výstupních hodnot lze obecně získat stejné generované sekvence čísel opakovaně. Kvalita generátoru pak také udává, jak se generovaná pseudonáhodná čísla statisticky blíží skutečně náhodným.

Generátory náhodných čísel (TRNG) pracují na zcela odlišném principu. Výsledná čísla získáváme měřením lidmi nepředpověditelných fyzických veličin a jejich interpretací pomocí elektronických systémů. Teoreticky bychom mohli předpovědět i tyto výsledky, pokud bychom znali všechny vstupní proměnné a chápali celý proces, ovšem prakticky jsou tyto předpovědi pro lidi a lidskou techniku nereálné. Výstupní sekvence těchto generátorů tak můžeme bez obav označit za skutečně náhodné, nepředpovědítelné. TRNG jsou obecně sestaveny ze tří částí: zdroj entropie, mechanismus sběru entropie a následné zpracování (postprocessing), které však nutně nemusí být přítomno. Díky získávání entropie z fyzikálních procesů je druhá část nutně hardwarová. Existují i softwarově implementované TRNG, např. generátory založené na pohybu myši uživatele nebo na

stisku kláves. Tyto generátory ale také obsahují fyzickou vrstvu, v těchto případech skenované klávesnice a myši.

2 Generátory pseudonáhodných čísel

Generátorů pseudonáhodných čísel je velké množství s mnoho vlastnostmi. Díky tomu lze poměrně pohodlně nalézt generátor odpovídající požadavkům konkrétní aplikace. Všechny rozšířené programovací jazyky obsahují implementované funkce pro generování pseudonáhodných čísel, většinou velmi jednoduché s nepříliš dobrými statistickými výsledky. Použití těchto implementovaných generátorů se kvůli nízké kvalitě generovaných čísel nedoporučuje, nejsou ani kryptograficky zabezpečené.

Jazyky C a C++ obsahují funkce *rand()* a *random()*, které jsou implementovány jako jednoduchý lineární kongruentní generátor (viz. 2.3). Volně přístupný kód samotné implementace lze najít v [8]. Inicializační funkce *srand(unsigned int)* a *srandom(unsigned int)* nastavují semínko generátoru, v případě nepoužití těchto funkcí se generátor spouští z přednastavené hodnoty 1. Pro použití těchto funkcí je třeba zahrnout do programu knihovnu *stdlib,* generovaná čísla se nacházejí v rozsahu 0 – 2147483646. [9]

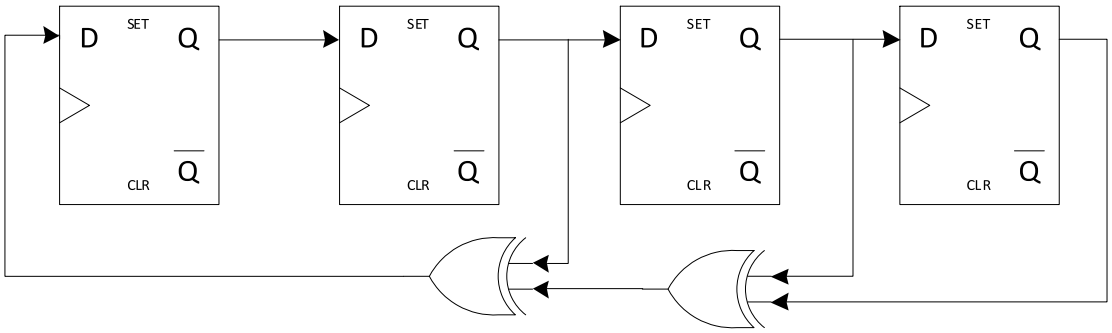
Jazyky C# a C++ využívající .NET Framework obsahují třídu *Random*, jejíž implementace vychází z modifikované verze subtraktivního algoritmu pro generování pseudonáhodných čísel, jehož autorem je Donald E. Knuth. Tento algoritmus je jednou z verzí lineárního kongruentního generátoru (viz. 2.3), kód implementace se nachází zde [10]. Generátor lze inicializovat dodáním 32 bitového semínka jako *Random(int),* nebo jako *Random()*, kdy hodnotu semínka dodají systémové hodiny. Voláním *Next(int, int)* dostaneme pseudonáhodné číslo v rozsahu určeném zadanými hodnotami int. [11]

Jazyk Java obsahuje třídu *Random*, která je velice podobná třídě *Random* v .NET Framework. Implementace je založená na stejném generátoru, ale jako semínko zadáváme 64 bitovou hodnotu long. Dále Java obsahuje funkci *Math.random*, která generuje hodnoty double v rozsahu (0 – 1>. Tato funkce ale pouze interně pracuje s funkcí *Random()*. [12]

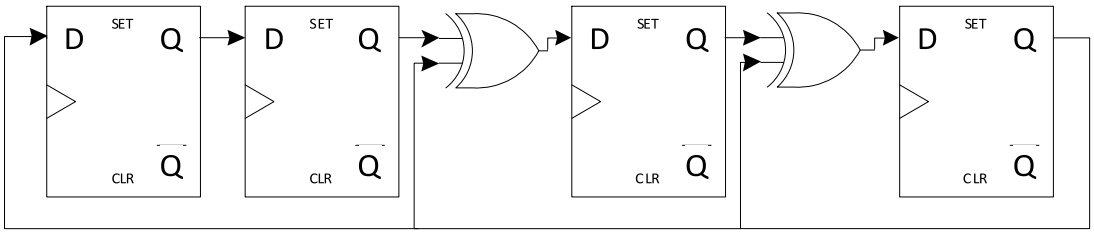
Pro tuto práci byly vybrány čtyři generátory pseudonáhodných čísel, které jsou dále podrobněji popsány, implementovány a nakonec testovány a porovnány.

2.1 Lineární zpětnovazební posuvný registr (LFSR)

Tyto generátory jsou jednou z nejjednodušších forem generování pseudonáhodných čísel. V podstatě se jedná o posuvný registr, jehož vstupní bit je lineární funkcí předchozího stavu uloženého v samotném registru. Tvořeny hardwarově, obecně se skládají z klopných obvodů typu D, které realizují paměťové buňky posuvného registru, a hradel XOR, která zajišťují požadovanou vstupní funkci. Umístění hradel XOR rozlišujeme na interní a externí. Při interním se hradla nachází mezi jednotlivými paměťovými buňkami, tato implementace se také nazývá Galoisova implementace (*Obr. 2.2*). Při externím umístění jsou hradla v samotné funkční zpětnovazebné smyčce, tato verze se nazývá Fibonacciho implementace (*Obr. 2.1*). [13, 14]



Obr. 2.1 Příklad jednoduché Fibonacciho implementace 4-bitového LFSR.



Obr. 2.2 Příklad jednoduché Galoisovy implementace 4-bitového LFSR.

Vektor bitů paměťových buněk nám při *n* bitovém registru dává *n* bitový vnitřní stav generátoru a při využití všech bitů pro výstup také *n* bitové generované číslo. Zpětnovazebná funkce se vyjadřuje jako polynom koeficientů dvourozměrného (bitového) pole, např. jednoduchý LFSR z *Obr. 2.1* můžeme vyjádřit pomocí polynomu $x^4+x^3+x^2+1$. Při vhodně zvoleném polynomu a tedy zvolené funkci dostáváme maximální možnou periodu generovaných čísel generátorem 2^n-1. Z periody 2^n odečítáme zakázaný stav těchto

generátorů. Při zakázaném stavu obsahuje registr samé nuly, to je vzhledem k realizování funkce pomocí hradel XOR problém. Pokud takový případ nastane, na všech vstupech hradel XOR se již neobjeví jiné než nulové hodnoty a na jejich výstupech tudíž také ne. Požadavky na semínko zde nejsou vysoké, musí prakticky splňovat pouze jedinou podmínku - nebýt nulové a tím neznemožnit funkci generátoru už při jeho inicializaci. [13, 14]

2.2 Mersenne Twister (MT)

Mersenne twister (dále jen MT) je generátor pseudonáhodných čísel, prakticky modifikace TGFSR (Twisted Generalized Feedback Shift Register), která využívá jako délku svojí periody Mersennovo prvočíslo. To je takové prvočíslo, které je o jedna menší, než celočíselná mocnina čísla dvě, dá se vyjádřit následujícím vztahem dle [15].

$$M_p = 2^n \tag{2.1}$$

MT byl vytvořen v roce 1997 na japonské Univerzitě Keió jako v té době zřejmě nejlepší pseudonáhodný generátor, tvůrci byli Makoto Matsumoto a Takuji Nishimura, přičemž prvně jmenovaný stál také za zrodem TGFSR. Později bylo vytvořeno velké množství různých verzí, např. SIMD-oriented Fast Mersenne Twister (SFMT) v roce 2006 nebo kryptograficky bezpečný CryptMT, MT totiž sám o sobě kryptograficky zabezpečený není. Jednoduchou lineární transformací maticí T^{-1} (temperovací matice, viz. Fáze temperování) dostaneme z výstupu generátoru lineární rekurentní sekvenci. Potom už lze poměrně snadno za pomoci dostatečného počtu výstupních dat získat momentální vnitřní stav a tedy i následující generovaná čísla. [16, 17]

V této práci se budeme zajímat o původní verzi MT19937, což je generátor s periodou 2^{19937}-1 (M_p) a s 623-rozměrným rovnoměrným rozdělením do 32-bitové přesnosti (k-rozdělení do v-bitové přesnosti – viz. konkrétní test, dodat číslo). Pro realizaci periody M_p se využívá tzv. nekompletní řady (algoritmus počítá s řadou bitových slov, která obsahuje také nekompletní bitové slovo, pouze část jeho bitů – viz. popis algoritmu). Tento generátor byl vytvořen pro generování uniformních reálných čísel, u kterých je speciální pozornost na MSB, jako odpověď na některé generátory, jež právě s MSB vykazovaly problémy. MT využívá výhod cache a pipeline a vyhýbá se složitým matematickým

funkcím násobení a dělení, je také poměrně úsporný na paměť, při velké periodě 2^{19937}-1 spotřebovává pro svoji práci místo pro 624 32-bitových slov. [16, 17]

Samotný algoritmus generátoru můžeme rozdělit na dvě fáze: fáze rekurentní a fáze temperování:

2.2.1 Fáze rekurentní

V první fázi je generována sekvence vektorů bitových slov, které považujeme za uniformní pseudonáhodná celá čísla v rozsahu *0 až* 2^w*-1*, kde w je počet složek vektoru bitového slova (LSB vpravo). Tato fáze je formou LSFR (Linear Feedback Shift Register), bity vnitřního stavu vznikají rekurzí a stejně tak vznikají tedy výstupní bity z rekurentních bitů vnitřního stavu. Posuvný registr je složen ze 623 32-bitových prvků a jednoho prvku 1-bitového, celkem tedy obsahuje 19937 bitů (M_p). [16]

Sekvence je generována lineární rekurentní rovnicí:

$$x_{k+n} := x_{k+m} \oplus (x_k^u \mid x_{k+1}^l)A \qquad (2.1)$$

Kde:
- n — konstanta, míra rekurzivity
- m — konstanta v rozsahu $1 \leq m < n$, zajišťující posun v rekurzivním vztahu sekvence x
- x^l_{k+1} — spodních *r* bitů ve slově x_{k+1} (váhově, od LSB)
- x^u_k — horních *w-r* bitů slova x_k (zase váhově, od MSB)
- A — konstantní matice o rozměrech *w*w* vytvořená pro zjednodušení maticového násobení
- $\oplus$, | — bitová operace XOR, spojovací operace

Jako seed použijeme vektory bitových slov x_0, x_1, … x_{n-1}, pro které nám rekurentní rovnice při k=0,1,2,… vytvoří vektory x_n, x_{n+1}, x_{n+2}, … [16]

Nejdříve spojujeme spodních r bitů slova x_{k+1} s horními w-r bity slova x_k $(x_k^u \mid x_{k+1}^l)$, poté výsledný vektor násobíme zprava maticí A, přičemž při vhodně zvolené matici A můžeme toto násobení jednoduše provést jako bitový posuv a samotnou matici tak vůbec nekonstruovat.

Pro zvolenou matici :

$$A = \begin{bmatrix} 0 & 1 & & & \\ 0 & 0 & 1 & & \\ \cdots & \cdots & \cdots & \ddots & \\ & & & & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \cdots & a_0 \end{bmatrix} \tag{2.2}$$

pak vypadá funkce násobení jako:

$$xA = \begin{cases} x \gg 1, & x_0 = 0 \\ (x \gg 1) \oplus a, & x_0 = 1 \end{cases} \tag{2.3}$$

Kde: x vektor jednotlivých bitů slova $(x_{w-1}, x_{w-2}, \ldots x_0)$
a spodní řádek matice A $(a_{w-1}, a_{w-2}, \ldots a_0)$
>> operace bitového posuvu vpravo

Výsledek tohoto násobení poté spojíme pomocí funkce XOR se vektorem x_{k+m} a získáme tak výsledný vektor x_{k+n}. Pro celý výpočet rekurzní rovnice tedy stačí funkce bitového posuvu a bitové operace XOR, OR a AND. [16, 17]

2.2.2 Fáze temperování

Pro zlepšení *k*-rozložení do *v*-bitové přesnosti generátoru provádíme takzvané temperování. Každé vygenerované slovo je zprava vynásobeno regulární maticí T o rozměrech w*w, čímž vzniká samotný výsledek temperování matice x, $z := xT$. Matice T je stejně jako matice A vybrána tak, aby se mohla její konstrukce eliminovat a její násobení realizovat jako jednoduché bitové operace. Temperování potom probíhá ve čtyřech krocích:

$$\begin{aligned} y &:= x \oplus (x \gg u) \\ y &:= x \oplus ((y \gg s)\&b) \\ y &:= x \oplus ((y \gg t)\&c) \\ z &:= x \oplus (y \gg l) \end{aligned} \tag{2.4}$$

Kde: u,s,t,l konstanty udávající temperující bitové posuvy
b,c temperující bitové masky velikosti slova

<<, >>	operace bitového posuvu vlevo, vpravo
&	bitová operace AND

Výsledkem po temperování již máme sekvenci pseudonáhodných bitových slov připravených pro jejich použití. Díky tomu, že MT přepisuje celou vnitřní řadu naráz, obejde se bez funkce modulo n a může používat pouze rychlé bitové operace. Jak je již zřejmé z výpočtů rekurentní rovnice a temperování, MT patří mezi parametrizované generátory. Jeho funkci můžeme ovlivnit celkem **11 parametry:**

Parametry periody: w - velikost slova (word), n - stupeň rekurze, m - střední hodnota posunu, r - bod rozdělení jednotlivého slova, a - vektorový parametr (matice A)

Parametry temperování: číselné parametry l, u, s a t, vektorové parametry b, c [16, 17]

2.3 Lineární kongruentní generátor (LCG)

Lineární kongruentní generátory (dále LCG) jsou jednou z nejstarších technik pro generování pseudonáhodných čísel, poprvé byly představeny už v roce 1951. Vzhledem ke své jednoduchosti a z toho vyplývající jednoduché implementaci a rychlosti byly tyto generátory velice oblíbené, ovšem špatnými volbami právě v implementaci a volbě konstant nebyla „kvalita“ generovaných pseudonáhodných čísel příliš vysoká, například dříve velmi rozšířený generátor RANDU obsahoval nevydařený LCG a podával pochybné výsledky [18].

Algoritmus LCG vychází ze základní rovnice:

$$x_{n+1} = (ax_n + c) \bmod m, \qquad n \geq 0 \tag{2.5}$$

Kde:	m	modul funkce, $m > 0$
	a	násobící konstanta, $0 \leq a < m$
	c	přičítaná konstanta, $0 \leq c < m$

Pokud jsou konstanty a,c nenulové, mluvíme o LCG, pokud je nenulová pouze konstanta a, jedná se o tzv. MCG (multiplicative congruential generator), často nazývaný Lehmerův generátor.

Jako seed potřebujeme hodnotu x_0 ($0 \leq x_0 < m$), z rovnice je také patrné, proč jsou generátory LCG tak úsporné na místo: jako jejich vnitřní stav (state) potřebujeme mít

uloženou pouze jednu předešlou generovanou hodnotu. Rychlost nám vyplývá z jednoduchosti rovnice, pro implementaci potřebujeme pouze funkce sčítání a násobení, velký dopad na rychlost má ale funkce modulo, která pokud není hardwarově implementovaná, generátor velice zpomaluje. Z tohoto důvodu se využívá volba konstanty m jako velikost bitového slova daného zařízení, funkce modulo je potom prováděna implicitně samotným hw a složitý výpočet odpadá. [19]

Pokud máme $m = 2^k$ (příhodně pro k-bitové stroje), maximální perioda LCG činí 2^k, u MCG je to 2^{k-2}. Při zvolení $m = 2^k$ ale vyvstává problém s nižšími bity, perioda bitu *číslo* ***b*** je 2^b. I když máme tedy celkovou periodu 2^k, pouze vyšší bity jsou „kvalitní“, nízké bity obsahují jasné opakující se vzorce. Statistické výsledky generátorů LCG tedy nejsou nijak povzbudivé, i když se používají techniky jako např. zahazování nižších bitů. Obecně ale zlepšujeme vlastnosti těchto generátorů zvětšováním počtu bitů, se kterými pracují, zvětšováním vnitřního stavu. Přidělením dostatečné operační paměti můžeme dostat dokonce i větší periodu a rovnoměrné k-rozložení výsledné sekvence než má MT. [19]

Mezi další výhody LCG mimo již zmiňovanou rychlost a úsporu paměti patří možnost jednoduché změny kompletně celé generované sekvence a to změnou přičítané konstanty c. Lze tak jednoduše vytvořit hned několik vygenerovaných sekvencí naráz (multiple streams). Další vlastností LCG je tzv. schopnost vyhledání (seekability), kdy lze pomocí jednoduché rovnice přeskočit libovolný počet ***i*** kroků v sekvenci bez nutnosti vypočítávat jednotlivé členy. [23]

$$x_{n+1} = \left(a^i x_n + \frac{c\left(a^i - 1\right)}{a - 1} \right) \bmod m \tag{2.6}$$

LCG se tedy mohou nabídnout jako poměrně dobrý základ pro generaci pseudonáhodných čísel, pokud následuje další úprava, která dokáže využít jejich předností a napravit nedostatky. Jak již bylo řečeno, samotná myšlenka za vznikem PCG je následné zpracování výsledků LCG pomocí funkce permutace. [19]

2.4 Permutovaný kongruentní generátor (PCG)

PCG jsou novým přírůstkem mezi generátory pseudonáhodných čísel, autorka Melissa E. O'Neill vytvořila v roce 2015 webové stránky pro zveřejnění své práce na toto téma a zpřístupnila zde také několik variant implementovaných PCG v podobě zdrojových kódů. PCG jsou generátory spojující generaci lineárními kongruentními generátory (LCG) a permutace funkce tzv. tuples ((podle definice uspořádaná množina hodnot – předpokládám jako u MT používané vektory bitových slov)), z toho vycházející název Permuted Congruential Generator. Představený princip permutace může být využit prakticky u každého generátoru pseudonáhodných čísel pro zlepšení jeho vlastností (samozřejmě při správné implementaci), autorka se však ve své práci zaměřila na vylepšení LCG, které mají vedle nedostatků i spoustu výhod a vzniklé PCG mají ambice být „nejlepšími" pseudonáhodnými generátory. Tato práce se bude zabývat několika představenými PCG. [19]

2.4.1 Permutace

Použitím funkce permutace chceme tedy samozřejmě upravit výstup tak, abychom dosáhli zlepšení jeho vlastností. Je však třeba volit tuto funkci opatrně, nemusíme totiž statistické vlastnosti našich pseudonáhodných čísel nijak změnit, nebo je můžeme dokonce ještě zhoršit. Vzhledem k povaze funkcí permutace, kdy měníme jeden výstup na jeden vstup, musíme dát pozor na jejich invertibilitu. Využijeme tedy variantu funkce k-do-1, kdy *k* vstupů může vést k jednomu výstupu. Pokud bychom proces obrátili, dostaneme pro každou naší vstupní hodnotu *k* výsledných možností. Při vysokém počtu *k* tak možnost inverze ztrácí na reálné proveditelnosti. Tyto funkce jsou široce využívané jako tzv. uniformní hashovací funkce. Místo neefektivního použití LCG a poté využití některého hashovacího algoritmu jsou ale u PCG vytvořeny funkce permutace přímo „na tělo". [19]

Základní myšlenkou je rozdělení bitů LCG výstupu na cílové bity a řídící bity. Řídící bity budou určovat konkrétní úpravu cílových bitů na konečný výstup generátoru. Možností těchto funkcí je opravdu mnoho a vzhledem k tomu, že je myšlenka tohoto využití mladá, dá se předpokládat, že budou vznikat nové a upravené verze PCG. Stačí dodržet pravidla pro tyto funkce, jak je stanovila autorka [19]:

- Bity jsou cenné, lze je zahodit, ale nelze vytvořit nové, ani duplikovat ty, které už máme.

- Rozdělení bitů do dvou nezávislých skupin, cílových bitů a řídících bitů.

- Řídící bity určují, jak naložit s cílovými bity. Co se s cílovými bity stane, to závisí pouze na vaší představivosti za předpokladu, že lze tuto operaci vrátit zpět, pokud ne, nejedná se o permutaci.

- Řídící bity nelze modifikovat, dokud plní řídící funkci.

- Funkce není omezená počtem průběhů, bity, které byly v předchozím průběhu použity jako řídící se mohou v dalším stát cílovými.

V originální práci bylo popsáno několik implementací permutačních funkcí pro tento generátor. Jedná se o implementace využívající náhodnou rotaci bitů (PCG-RR), náhodné bitové posuny (PCG-RS), náhodné bitové posuny s využitím bitové operace XOR (PCG-XSH), nebo kombinace předchozích (PCG-XSH-RR, PCG-XSH-RS) [19]. Pro podrobnější popis byla vybrána první jmenovaná implementace.

2.4.2 Náhodná rotace bitů PCG-RR (Random Rotation)

Využijeme vlastnosti LCG – nejvyšší bity jsou ty nejvíce náhodné. Použijeme tedy horních t bitů (řídící bity) pro určení rotace cílových bitů, tato rotace vytvoří spolu se zahozením spodních bitů výsledné pseudonáhodné číslo. Výsledná sekvence pseudonáhodných čísel tedy je vytvořena z mixu „náhodně“ rotovaných čísel vygenerovaných LCG. [19]

Výsledná čísla můžeme vyjádřit pomocí funkce:

$$f_c(n) = (n \twoheadrightarrow r) \circlearrowright c \qquad (2.7)$$

Kde:

n		cílové bity
c		počet rotovaných bitů, numerická hodnota daná řídícími bity t
$n \twoheadrightarrow r$		ponechání horních r bitů z n, zbytek zahozeno
$n \circlearrowright c$		rotace c bitů z n po směru hodinových ručiček

Touto permutací jsme vylepšili periodu všech bitů, která nyní činí 2^k (pro k-bitová čísla). Celková perioda generátoru potom bude 2^n, přičemž tato perioda proběhne *r*-krát a pro každý z těchto *r* průběhů bude jiná rotace, tudíž i kompletně jiná výsledná čísla. Výstupem generátoru jsou tedy *r*-bitová pseudonáhodná čísla. [19]

3 Generátory náhodných čísel

Generátorů náhodných čísel existuje také poměrně velké množství, i když ani zdaleka ne takové, jako generátorů pseudonáhodných čísel. Odlišují se nejen způsoby získávání entropie z fyzikálních procesů, ale i možnostmi jejího dalšího zpracování a samotného generování čísel. Mají široké využití v bezpečnostních aplikacích díky statistickým kvalitám výsledných čísel. Riziko u těchto generátorů se skrývá především v implementaci, špatná implementace dokáže významně zhoršit kvalitu výstupu celého generátoru. Získaná entropie musí být vhodně zpracována, aniž by byla znehodnocena samotnými funkčními obvody nebo vnějšími rušivými vlivy. Každé ovlivnění se projeví na náhodnosti generovaných čísel.

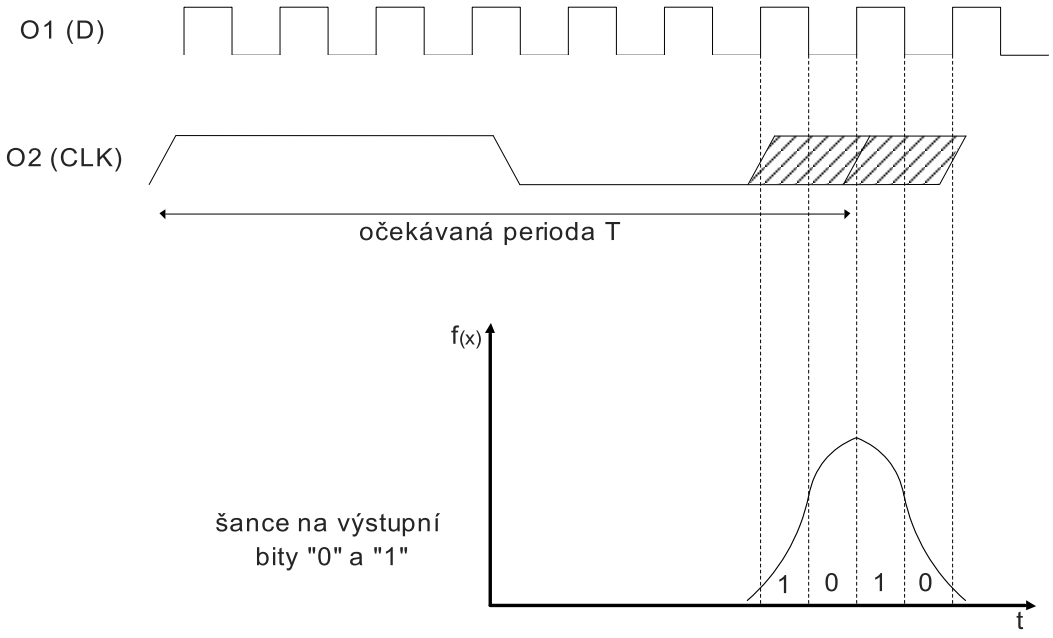
Za zmínku stojí generátory TRNG založené na principu nepředvídatelného rozpadu atomů a získávání entropie z uvolněného záření, nebo také generátory získávající entropii měřením atmosférického šumu. Mnoho aplikací (především šifrovacích) také využívá TRNG jako zdroje semínek pro další zpracování v PRNG. TRNG jsou totiž obecně pomalejší než rychlé výpočetní algoritmy PRNG a tak zkombinováním těchto dvou druhů generátorů můžeme získat velké množství vysoce kvalitních náhodných čísel. Pro generátory skutečně náhodných čísel tedy mluví především jejich kryptografická bezpečnost a celkově kvalita generovaných sekvencí, proti mluví jejich rychlost a také cena. Většinou je totiž využíván speciální hardware pouze k účelu generování čísel a to znamená náklady navíc.

Tato práce se zabývá podrobněji jedním plně hardwarovým TRNG, kde jako zdroj entropie slouží metastabilita číslicových klopných obvodů a jedním softwarovým TRNG, kde je entropie zastoupena chováním operačního systému a samotného procesoru, na kterém daný software běží.

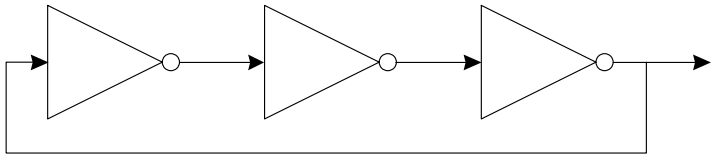
3.1 Generátor náhodných čísel založený na kruhových oscilátorech

Tyto generátory využívají jako zdroj entropie tepelný šum na oscilátoru, který způsobuje časové odchylky hrany signálu oscilátoru, tzv. jitter. Při správném navržení obvodu způsobí jitter v části elektroniky metastabilitu a tedy nepředvídatelné chování, hodnoty.

Základním principem je použití klopného obvodu (nejčastěji typu D), na jehož datový vstup je přiveden signál z rychlého oscilátoru (O1). Jako hodinový signál je na klopný obvod přiveden signál z druhého, pomalejšího oscilátoru (O2). Poměr rychlostí těchto oscilátorů je zásadní pro navození nepředvídatelného stavu na vstupu klopného obvodu. Rychlost O1 musí být taková, aby se jeho celá perioda vešla do časového okna, které tvoří samotný jitter oscilátoru O2. Pokud je tato podmínka splněna a střída signálu O1 je 50%, na datovém vstupu bude stejná šance získat logickou 1, jako získat logickou 0. Velmi dobře tento princip znázorňuje *Obr. 3.1*. [20]



Obr. 3.1 Průběhy signálů O1 a O2 se znázorněním jitteru O2 a graf funkce hustoty pravděpodobnosti vstupních bitů do D (překresleno z [20]*).*



Obr. 3.2 Jednoduchý třístupňový kruhový oscilátor sestavený z invertorů.

Snadným způsobem, jak získat oscilátory požadovaných vlastností, je implementace kruhového oscilátoru. Tyto oscilátory vykazují dostatečné rychlosti a mají také saturovaný výstup, což zjednodušuje návrh celého obvodu. Nejjednodušší implementace kruhového oscilátoru je složena z jednoduchého třístupňového invertoru (*Obr. 3.2*). Důležitý je lichý počet prvků, oscilace je potom zajištěna zapojením, kdy se výstup posledního invertoru a tedy vstup prvního s každou periodou otáčí a budí tak další činnost. Tři členy jsou použity z důvodu rychlosti oscilátoru, každý člen zavádí do obvodu svoje pracovní zpoždění, které se označuje jako jednotka zpoždění, a prodlužuje tak čas celkové periody. Frekvence tohoto oscilátoru je tedy určená pracovním zpožděním jednotlivých invertorů a může být vyjádřena následující rovnicí. [20]

$$f_0 = \frac{1}{2Nt_p} \qquad (3.1)$$

Kde: N počet členů oscilátoru
t_p pracovní zpoždění jednoho členu

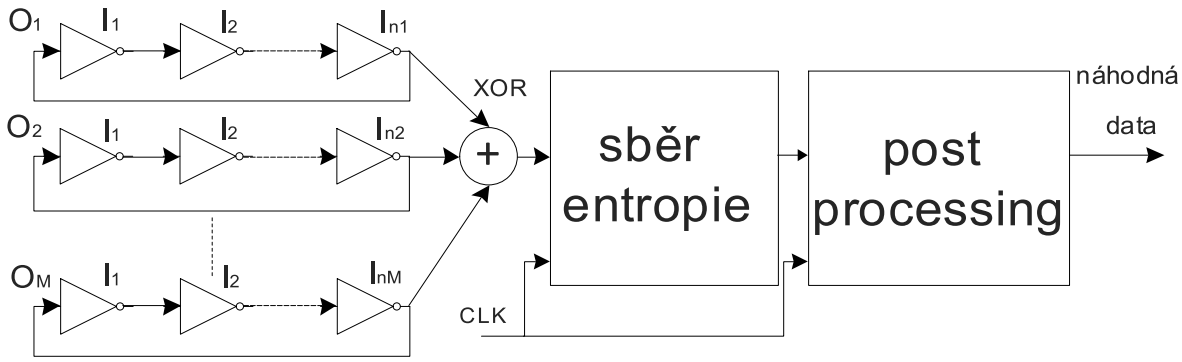
Dobu pracovního zpoždění jednotlivých invertorů t_p můžeme nahradit 69% jejich časové konstanty, dostaneme upravený vztah: [20]

$$f_0 = \frac{1}{2N * 0{,}69RC} \qquad (3.2)$$

Kde: R odpor sepnutého tranzistoru v invertoru
C celková kapacita uzlu (invertoru)

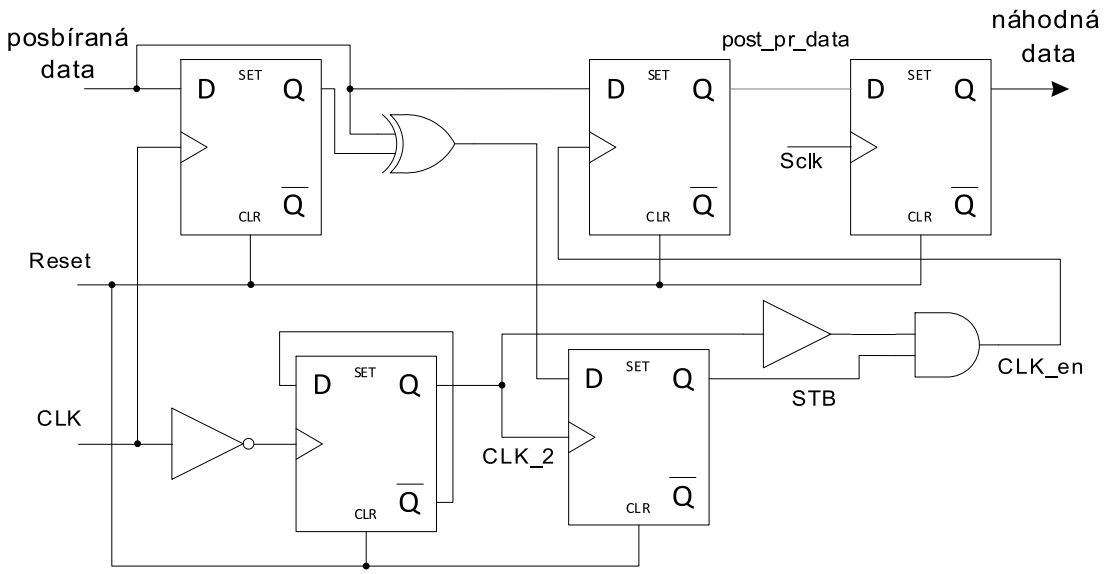
Složitější implementace využívající větší počet kruhových oscilátorů pro získání entropie se dají rozdělit do dvou hlavních kategorií podle délky (počtu prvků) použitých oscilátorů. První možností je použití nesoudělných délek jednotlivých oscilátorů a druhou možností je použití identických délek jednotlivých oscilátorů. Výstup všech M použitých oscilátorů se skládá dohromady pomocí stromové struktury z hradel XOR. Výstupem

struktury je potom signál obsahující požadovanou entropii (jitter), který je vzorkován nekorelovanou frekvencí pro získání náhodných bitů v bloku „sběr entropie“. Tento blok může být opět realizován např. pomocí klopného obvodu typu D. Výstupní signál XOR struktury obsahuje deterministické oblasti a přechodové zóny, ve kterých se objevuje jitter. K těmto zónám přispívá každý jednotlivý oscilátor a s jejich počtem entropie signálu roste. Získání maximální možné entropie je potom otázkou návrhu oscilátorů, návrhem struktury XOR a vhodného vzorkování. Náhodné bity získané tímto procesem jsou většinou ještě zpracovány algoritmy, které vylepší jejich statistické vlastnosti (blok „postprocessing“). Topologii této implementace zobrazuje *Obr. 3.3*. [21]

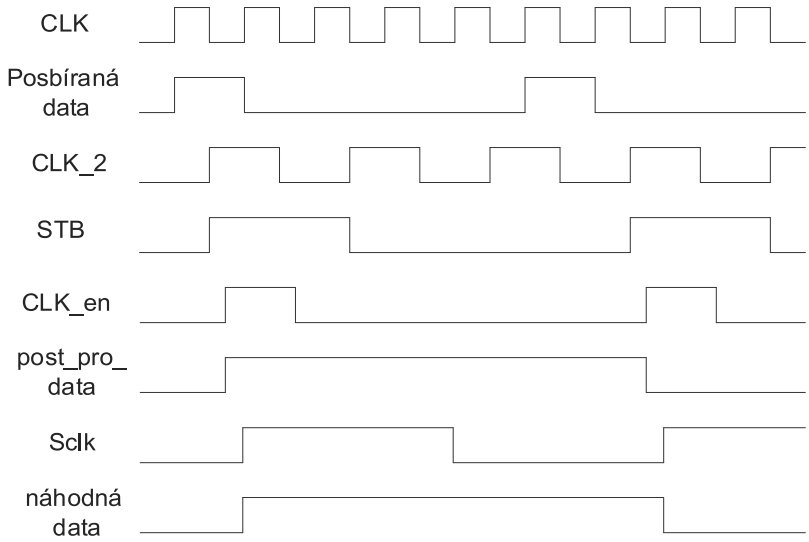


Obr. 3.3 Topologie TRNG s větším množstvím kruhových oscilátorů (překresleno z [21]*).*

Postprocessing vylepšující náhodnost dat může být zpracován například jako jednoduchý Von Neumanův korektor, jehož zapojení nalezneme na *Obr. 3.4*. Časování těchto obvodů potom zobrazuje ***Chyba! Nenalezen zdroj odkazů.***. Funkce tohoto korektoru spočívá ve vyřazování dvojic shodných bitů (tedy 00, 11) a převodu dvojic různých bitů na jediný bit (01 na 1, 10 na 0). V důsledku této funkce korektor přibližně čtyřikrát zmenšuje datový tok. Ze 4 vstupních bitů obecně generuje 1 výstupní bit. [21]



Obr. 3.4 Schéma jednoduchého Von Neumanova korektoru (překresleno z [21]*).*



Obr. 3.5 Časovací diagram jednoduchého Von Neumanova korektoru (překresleno z [20]).

3.2 Softwarový generátor náhodných čísel založený na podmíněném závodu funkcí

Tento plně softwarový generátor jehož autory jsou Adrian Coles, Radu Tudoran a Sebastian Banescu z Technické univerzity Cluj-Napoca, je představen jako kompaktní softwarové řešení bez potřeby specializovaného hardwaru, nebo využití externích hardwarových součástek, které představují riziko kompromitace generované náhodnosti a tedy snížení kryptografické bezpečnosti generátoru. [22]

Principem tohoto generátoru je současné nesynchronizované spuštění více výpočetních vláken programu, která všechny pracují s jednou sdílenou proměnnou. Tato proměnná je

nezávisle na sobě jednotlivými vlákny čtena, upravována a přepisována. Po dokončení výpočtů všech vláken je konečná hodnota uložená v proměnné a ta reprezentuje výsledné náhodné číslo. Díky nesynchronizovanému průběhu funkcí vláken jsou vytvořeny podmínky, kdy je výsledný stav proměnné nepředpověditelný a určený soupeřením jednotlivých vláken. Takové podmínky se běžně téměř nevyskytují a obecně jsou u programů nežádoucí. V tomto případě je ale aplikace navržena pro jejich podpoření a zvýšení jejich výskytu v co největší možné míře. [22]

Plánování výpočtů vláken při běhu programu je obvykle deterministické, existují však možnosti jak jej ovlivnit a získat rozdílné, náhodné výsledky při každém průběhu. Funkce každého vlákna se dá rozdělit na tři kroky. V prvním kroku přečteme hodnotu ze sdílené proměnné a uložíme jí do proměnné lokální, v druhém kroku hodnotu v lokální proměnné nějakým způsobem upravíme a ve třetím kroku zapíšeme nově upravenou hodnotu do sdílené proměnné. Podmínky, kdy nastává závod vláken, jsou dané tím, že pořadí vykonávání jednotlivých kroků se může průběh od průběhu programu lišit. Příkladem budiž dva rozdílné průběhy programu, který obsahuje dvě identická vlákna. Zatímco při prvním průběhu stihne první vlákno vykonat všechny tři kroky, než systém uvolní výpočetní prostor pro vlákno druhé, při druhém průběhu první vlákno nestihne vykonat zapsání nové hodnoty do sdílené proměnné. Druhé vlákno tak vykoná své tři kroky a až poté se vykoná poslední krok z prvního vlákna, kdy se do sdílené proměnné zapíše nová hodnota a výpočet druhého vlákna přepíše. Po těchto dvou průbězích tak bude v proměnné uložena rozdílná hodnota. Další kapitola se zabývá podmínkami pro tento nepředvídatelný závod. [22]

3.2.1 Získání náhodnosti

Jak již bylo řečeno, plánovač v počítači je deterministický, využívá algoritmus, který pro každý průběh programu vytvoří stejnou posloupnost vykonávaných kroků všech vláken ve stejném pořadí. Pro zjednodušení předpokládejme, že program obsahuje pouze dvě vlákna, která spolu budou závodit. Tato vlákna jsou z pohledu procesoru naprosto identická, mají stejnou prioritu a tak jim budou přiděleny stejně dlouhé časové úseky pro výpočty. Přepínání mezi jednotlivými vlákny probíhá vždy ve stejný čas, přesně určený plánovačem, tento čas je tedy také stejný pro každý průběh programu. Přes všechny tyto

deterministické vlastnosti program při každém průběhu vygeneruje náhodné číslo a to díky určitým náhodným faktorům, které můžeme rozdělit na hardwarové a softwarové. [22]

- **Softwarové faktory náhodnosti**

Nejdůležitějším softwarovým faktorem, který ovlivňuje průběh programu je prostředí, na kterém software běží, samotný operační systém. Plánovač sice za stejných podmínek vytvoří pokaždé stejnou vykonávanou sekvenci, ale v reálném prostředí spuštěného systému je obtížné, prakticky nemožné získat identické podmínky pro každý průběh. Autoři testovali generátor spuštěný jako jedinou uživatelskou aplikaci na operačním systému, i tady ale vnitřní procesy OS vždy přispěly k náhodnosti generovaných čísel. [22]

- **Hardwarové faktory náhodnosti**

Autoři v [22] identifikovali tři hardwarové faktory:

- Prvním faktorem je minutí mezipaměti (cache misses). Kdykoliv se mezipaměť netrefí při predikci následujících instrukcí, procesor je nucen data vytáhnout z dalších vrstev paměti a tím zabírá více hodinových cyklů. Tím pádem pro stejný čas přidělený jednotlivým vláknům může procesor vykonat různý počet instrukcí a jistá vlákna se opozdí oproti zbytku. Takto vytvořený závod vláken je nepředpovídatelný, ale také velmi závislý na prostředí spuštěného programu. Pokud bude procesor vykonávat pouze několik „málo“ procesů stále dokola, minutí mezipaměti se vyskytnou v malém množství, pokud vůbec. Jestliže se veškeré spuštěné procesy vejdou do mezipaměti, všechna minutí nakonec vymizí. V reálném OS je ale spuštěno větší množství procesů a minutí mezipaměti se budou objevovat. Můžeme je zavrhnout jako deterministické a přispějí k náhodnosti generovaných čísel.

- Druhým HW faktorem je způsob, jakým je ovlivněný tok instrukcí v pipeline při výskytu přerušení, které vede k přepnutí do jiného vlákna. V takové situaci pipeline obsahuje instrukce vlákna vykonávaného před přerušením. Tyto instrukce ale budou vykonány až po obnovení funkce tohoto vlákna. Nově načítané a vykonávané instrukce patří vláknu, na které přerušení vede. Předpokládejme případ, kdy první instrukce načtená po obnovení činnosti

prvního vlákna závisí na svojí předcházející instrukci. Tato předcházející instrukce byla vykonaná v pipeline před přerušením. Pro svoji práci potřebuje více hodinových taktů a tak by musela následná závislá instrukce čekat na celé její vykonání v případě, kdy by se přerušení neobjevilo. Při výskytu tohoto přerušení je ale tato předcházející dlouhá instrukce již vykonávána a její činnost bude dokončena v hodinových taktech, které již spadají do časového okna přiděleného pro vlákno spuštěné přerušením. Po obnovení činnosti prvního vlákna závislá instrukce již nepotřebuje čekat a ušetřené hodinové takty ovlivní závod.

- Třetím faktorem je časovač přerušení využívaný pro časování přepínání jednotlivých vláken. Hardwarové hodiny nejsou dokonalé a mohou se rozcházet s reálným časem každou sekundu až o konstantu $\rho << 1$. Díky tomu jednotlivé časové úseky přiřazené pro vlákna mohou mít rozdílnou délku, i když jsou matematicky shodné. Pokud nejsou využity stejné hardwarové hodiny pro generování přerušení a pro určení jednoho taktu procesoru, nebo nejsou tyto hodiny synchronizovány, může díky nedokonalosti HW hodin nastat následující případ. Pro každé vlákno budou v rámci jednoho časového úseku předěleného přerušením vykonány různé počty hodinových taktů procesoru.

Poslední dva faktory nejsou závislé na prostředí spuštěného programu. Úzce spolu souvisí vzhledem k jejich práci s časem, ovlivnění pipeline je nejspíše dokonce přímým následkem posledního faktoru, chyby hardwarových hodin. Ta se pohybuje pro každou sekundu v rozsahu $[-\rho, \rho]$, kde ρ nabývá hodnot 10^{-6} až 10^{-8} μs. Časový úsek τ přidělený pro výpočet vlákna je tak ve skutečnosti dlouhý něco v rozmezí $[\tau - \rho\tau, \tau + \rho\tau]$. [22]

4 Metody hodnocení kvality generátorů

Náhodná čísla se využívají v širokém spektru aplikací a existuje mnoho způsobů jejich generování. Pro každou aplikaci mohou být požadavky na generátory velmi odlišné a tak je otázkou, jak rozlišit „dobré“ generátory od těch „špatných“. Hlavním kritériem je samozřejmě kvalita (náhodnost) generovaných čísel, které jsou věnovány testy generátorů v kapitole *4.1*.

V mnoha případech ale může být žádoucí jiná vlastnost generátoru i na úkor kvality generované náhodnosti. Jedná se například o aplikace, kde potřebujeme rychle velké množství náhodných dat, která nemusí splňovat nejpřísnější statistické požadavky. Dalším takovým případem může být potřeba generovat náhodná čísla na hardwarově velice omezené platformě. Pak nám nezbývá nic než využít jednoduššího, méně HW náročného generátoru a to i za cenu méně kvalitních výstupních čísel. Na druhou stranu existují aplikace, ve kterých je co nejvyšší kvalita generovaných čísel prioritou. Vesměs se jedná o veškeré kryptografické aplikace, kde jde v prvé řadě o bezpečnost celého systému. Ostatní vlastnosti použitého generátoru jako je složitost, velikost a cena jsou potom podřízené kvalitě výstupu. Tyto ostatní vlastnosti generátorů nepodléhající přímému testování jsou rozebírány v kapitole *4.2*.

4.1 Testy generátorů náhodných čísel

Nejen generátorů, ale i jejich testů je veliké množství. V zásadě se dají rozdělit na dvě skupiny, empirické (statistické) testy (*4.1.1*) a teoretické testy (*4.1.2*). Empirické testy se zabývají zkoumáním generovaných sekvencí náhodných čísel bez znalosti vnitřní struktury RNG, které je generují. Teoretické testy se zabývají samotným procesem generování uvnitř RNG, aniž by bylo potřeba znát generovaná čísla. Teoretické testy jsou většinou lepší, pokud pro daný generátor existují. [2]

4.1.1 Empirické testy

Empirické testy obecně zkoumají části generovaných sekvencí a jsou tedy výborné pro zkoumání tzv. lokální náhodnosti. Důležitými dvěma testy jsou tzv. chi-kvadrát test a Kolmogorov-Smirnův test. Tyto dva testy se hodí pro použití v různých případech, často jsou však používané dohromady a společně tvoří základ pro všechny empirické testy.

Chi-kvadrát test (χ^2)

Původ tohoto testu se datuje až do roku 1900, kdy jej publikoval Karl Pearson. Ten využil při svém teoretickém popisu testu symbolu χ^2, z čehož vzniklo pojmenování. Předpokládáme *n* nezávislých pozorování, v našem případě *n* generovaných elementů (čísel) RNG. Každé z těchto *n* čísel patří do jedné z *k* kategorií. Jako p_s označíme pravděpodobnost, že číslo bude spadat do *s*-té kategorie. Jako Y_s označíme počet čísel, které do *s*-té kategorie skutečně spadnou. Pro velké počty generovaných čísel *n* potom dle [2] očekáváme:

$$Y_n \approx n * p_s \tag{4.1}$$

Zajímá nás statistická odchylka od této předpokládané hodnoty. Vyčíslená odchylka shrnující všech *n* čísel s přihlédnutím k pravděpodobnostem jejich zařazení p_s bude výsledkem tohoto testu. Označíme ji jako *V* a dle [2] odpovídá:

$$V = \frac{1}{n} \sum_{1 \le s \le k} \frac{{Y_s}^2}{p_s} - n \tag{4.2}$$

Nyní je třeba určit, jaká hodnota *V* je vyhovující a jaká nikoliv. Zavedeme proto stupeň volnosti *v:*[2]

$$v = k - 1 \tag{4.3}$$

Pomocí *v* určíme v kvantilových χ^2 tabulkách hladinu významnosti, které výsledek testu *V* odpovídá (nebo se jí nejvíce blíží). Pro ukázku vezměme případ, kdy *k = 8*, tedy *v = 7*. V *Tab. 4.1* potom vidíme, že by hodnota *V* měla být v 99% případů menší, než 18,48 (*V <18,48*). Větší hodnota *V* (*V >18,48*) má pouze jednoprocentní šanci na objevení. [2]

Tab. 4.1 Část kvantilové χ2 tabulky pro vybrané stupně volnosti.

	p = 0,01	*p = 0,05*	*p = 0,25*	*p = 0,5*	*p = 0,75*	*p = 0,95*	*p = 0,99*
v = 6	*0,872*	*1,635*	*3,455*	*5,348*	*7,841*	*12,590*	*16,810*
v = 7	*1,239*	*2,167*	*4,255*	*6,46*	*9,037*	*14,070*	*18,480*
v = 8	*1,646*	*2,733*	*5,071*	*7,344*	*10,220*	*15,510*	*20,090*
v = 9	*2,088*	*3,325*	*5,899*	*8,343*	*11,390*	*16,920*	*21,670*
v = 10	*2,558*	*3,940*	*6,737*	*9,342*	*12,550*	*18,310*	*23,210*

Čtením v tabulce jednoduše interpretujeme testované sekvence dle [23]:

- Pokud je *V* menší než hladina odpovídající 1%, nebo větší než hladina odpovídající 99%, považujeme testovanou sekvenci za „nenáhodnou“.
- Pokud je *V* v rozsahu hladin odpovídajících 1-5%, nebo v rozsahu odpovídající 95-99%, považujeme testovanou sekvenci za „podezřelou“.
- Pokud je *V* v rozsahu hladin odpovídajících 5-10%, nebo v rozsahu odpovídající 90-95%, považujeme testovanou sekvenci za „téměř podezřelou“.
- Pokud je *V* v rozsahu hladin odpovídajících 10%-90%, předpoklad náhodnosti není porušen.

Hodnota *n* musí být pro testování co největší, tedy co nejvíce zkoumaných čísel. Jestliže vyjde „nenáhodná“ sekvence, nebo pro vícero opakování testu „podezřelý“ výstup, generovaná sekvence není skutečně náhodná. Tento způsob hodnocení generátorů ale není vždy dostačující, rozšíření pomocí Kolmogorov-Smirnova testu je vhodnější. [2, 23]

Kolmogorov-Smirnův test

Kolmogorov-Smirnův (KS) test má své počátky v roce 1933 v práci A. N. Kolmogorova, jehož práci vylepšil v roce 1939 N. V. Smirnov. Nejdříve zavedeme distribuční funkci $F_X(x)$ pro náhodný jev X:

$$F_X(x) = pravděpodobnost, že\ (X \leq x) \tag{4.4}$$

Distribuční funkce $F_X(x)$ je v rozsahu <0,1> a bude vždy růst, vzhledem k pohybu x od - ∞ do + ∞ [23]. Prakticky stejně jako v χ^2 testu označíme generované hodnoty jako X (v χ^2 testu jsme označovali Y). Distribuční funkce u χ^2 testu $F_Y(y)$ by byla nespojitá, Y mohou nabývat jen určité diskrétní hodnoty. Pro použití KS testu naopak požadujeme distribuční funkci $F_X(x)$ spojitou. Řešíme rozdílnou situaci, kdy generovaná čísla smí nabýt jakoukoliv hodnotu v daném intervalu. Pro n nezávislých pozorování (čísel) dostaneme hodnoty X_1, X_2, ... X_n. Empirickou distribuční funkci $F_n(x)$ potom definujeme jako: [2]

$$F_n(x) = \frac{počet\ všech\ X, které\ jsou\ \leq x}{n} \tag{4.5}$$

KS test porovnává distribuční funkce $F_X(x)$ a $F_n(x)$ měřením jejich rozdílů. Pro dostatečně velké n očekáváme u skutečně náhodné sekvence velkou podobnost těchto funkcí. Měřené rozdíly definujeme: [2]

$$\begin{aligned} K_n^+ &= \sqrt{n}\ \max\big(F_n(x) - F_X(x)\big), \quad -\infty \leq x \leq +\infty \\ K_n^- &= \sqrt{n}\ \max\big(F_X(x) - F_n(x)\big), \quad -\infty \leq x \leq +\infty \end{aligned} \tag{4.6}$$

Kde: K_n^+ Největší odchylka při $F_X(x) < F_n(x)$
K_n^- Největší odchylka při $F_X(x) > F_n(x)$
$\sqrt{n}$ faktor odstraňující závislost K_n^+ a K_n^- na hodnotě n (podrobněji v [23])

Hodnoty K_n^+ a K_n^- jsou našimi výsledky, analogicky k V u χ^2 testu. Stejně tak je porovnáváme se statistickou tabulkou. Podle *Tab. 4.2* tak můžeme určit, že např. K_{11}^+ má 95% šanci být menší, než 1,1688. Vzhledem k tomu, že distribuce je stejná pro K_n^+ i K_n^-, můžeme to samé prohlásit o K_{11}^-. Na rozdíl od χ^2 testu nejsou v tabulce obsaženy přibližné hodnoty, ale přesné hodnoty (v rámci chyby zaokrouhlení). [2]

Tab. 4.2 Část tabulky kritických hodnot KS testu

	p = 0,01	*p = 0,05*	*p = 0,25*	*p = 0,5*	*p = 0,75*	*p = 0,95*	*p = 0,99*
n = 10	*0,02912*	*0,1147*	*0,3297*	*0,5426*	*0,7845*	*1,1658*	*1,4440*
n = 11	*0,03137*	*0,1172*	*0,3330*	*0,5439*	*0,7863*	*1,1688*	*1,4484*
n = 12	*0,03137*	*0,1193*	*0,3357*	*0,5453*	*0,7880*	*1,1714*	*1,4521*
n = 15	*0,03424*	*0,1244*	*0,3412*	*0,5500*	*0,7926*	*1,1773*	*1,4606*

Hodnotu *n* je třeba vhodně zvolit i u tohoto testu. Potřebujeme jí dostatečně velikou, abychom rozpoznali větší rozdíly v diferenčních funkcích $F_X(x)$ a $F_n(x)$. Zároveň ale nemůže být hodnota *n* příliš veliká, aby se nevyhladily lokální nenáhodné části. V [23] je popsána vhodná hodnota *n* jako 1000, přičemž je zde popsán i postup, kdy jsou výsledná statistická data K_{1000}^{+} znovu testovaná KS testem. Tím se dosáhne detekování lokálních i celkových nenáhodností. Hodnocení zkoumaných číselných sekvencí probíhá stejně, jako u χ^2 testu podle procentuálního zařazení výsledku. [2, 23]

Jak bylo řečeno na konci popisu χ^2 testu, jeho spojením s KS testem dostaneme mnohem lepší ohodnocení generátoru. Provedeme *m* χ^2 testů a zaznamenáme výsledné hodnoty $V_1, V_2, \dots V_m$. Ty následně podrobíme KS testu. Detaily procesu lze najít v [23].

Nyní se stanovenými základy v podobě KS a χ^2 testů se můžeme pustit do samotných empirických testů. Z důvodu rozsahu práce je jich zde popsáno pouze několik a stručně. Začneme redefinováním sekvencí, které jsou testům podrobeny:

$$\langle U_n \rangle = U_0,\ U_1,\ U_2, \dots\ U_n \tag{4.7}$$

$$\langle Y_n \rangle = Y_0,\ Y_1,\ Y_2, \dots Y_n$$
$$\textit{definováno: } Y_n = \lfloor dU_n \rfloor \tag{4.8}$$

Sekvence *(4.7)* je sekvence reálných čísel nezávisle a rovnoměrně rozdělených mezi nulou a jedničkou. Sekvence *(4.8)* je sekvence celých čísel (integer) nezávisle a rovnoměrně rozdělených mezi 0 a *d*-1. Hodnota *d* se volí podle potřeby. [23]

Test rovnoměrného rozložení (frekvenční test)

Jedná se o nejjednodušší empirický test. Abychom zjistili splnění podmínky rovnoměrného rozložení sekvence *(4.7)*, aplikujeme KS test, kdy: $F_X(x) = x$ pro $0 \leq x \leq 1$. Jako alternativu můžeme zvolit vhodné d (např. 64 nebo 128 pro binární počítač) a poté k testování využít *(4.8)*. Pro každé celé číslo r ($0 \leq r \leq d$) zjistíme počet $Y_j = r$ pro $0 \leq j < n$. Pak aplikujeme χ^2 test pro $k = d$ a pravděpodobnost každé kategorie $p_s = 1/d$. [23]

Sériový test

Rovnoměrné nezávislé rozdělení nechceme jen pro individuální elementy sekvence, ale také pro páry po sobě následujících čísel. Pro např. binární sekvenci by tak měl být výskyt párů (0,0), (0,1), (1,0) a (1,1) stejně pravděpodobný. Využijeme *(4.8)* a spočítáme počet případů, kdy $(Y_{2j}, Y_{2j+1}) = (q, r)$ pro $0 \leq j < n$. Tento součet zjistíme pro každý jednotlivý pár (q, r) přičemž $0 \leq q$ a $r < d$. Poté aplikujeme χ^2 test pro $k = d^2$ a pravděpodobnost každé kategorie $p_s = 1/d^2$. Na co si musíme dát pozor je to, že χ^2 test vyžaduje n nezávislých hodnoty. Abychom testu dodali n nezávislých párů, musí mít naše sekvence celkem $2n$ jednotlivých elementů. Tento test nemusí být využit pouze k testování párů, lze testovat i trojice, čtveřice atd. Je ale potřeba opatrnosti při volení hodnoty d a nezapomenout na délku sekvence násobku n kvůli χ^2 testu. [2, 23]

Test mezer

Test zkoumá mezery mezi jednotlivými výskyty U_j (použijeme *(4.7)*) v zadaném rozsahu. Pokud jsou α a β dvě reálná čísla při $0 \leq \alpha < \beta \leq 1$, sledujeme délku po sobě jdoucích subsekvencí U_{j+1}, U_{j+2}, … U_{j+r} tak, že subsekvence U_{j+r} leží mezi α a β a vyplňuje celý tento prostor svojí délkou. Dále provedeme χ^2 test, přičemž použijeme různé délky mezer jako jednotlivé kategorie testu k. Pravděpodobnosti pro test získáme pomocí výpočtu: [2, 23]

$$p_0 = p, \quad p_1 = p(1-p), \quad p_2 = p(1-p)^2, \quad \ldots \quad p_k = p(1-p)^k, \tag{4.9}$$

Kde: $p = \beta - \alpha$ pravděpodobnost, že se jakékoliv U_j nachází mezi α a β

Poker test

Rozdělíme sekvenci do n skupin po pěti následujících celých číslech (využíváme tedy *(4.8)*) $\{Y_{5j}, Y_{5j+1}, \ldots Y_{5j+4}\}$ pro $0 \leq j < n$. Každá z těchto skupin patří do jedné z kategorií:

1. Všechny rozdílné (abcde)
2. Jeden pár (aabcd)
3. Dva páry (aabbc)
4. Tři stejné (aaabc)
5. Full house (aaabb)
6. Čtyři stejné (aaaab)
7. Pět stejných (aaaaa)

Obecně můžeme uvažovat n skupin k po sobě jdoucích celých čísel, přičemž skupina obsahuje r různých čísel (v našem případě $k = 5$ a např. pro 2. kategorii $r = 4$). Skupiny podrobíme χ^2 testu, pravděpodobnost pro použití v testu získáme vztahem: [2, 23]

$$p_r = \frac{d(d-1)\ldots(d-r+1)}{d^k}\left\{{k \atop r}\right\} \tag{4.10}$$

Další testy společně s podrobnějšími popisy všech testů, včetně teorie, která je podkládá, lze nalézt v [23].

4.1.2 Teoretické testy

Ačkoliv je vždy možné RNG testovat pomocí empirických testů, mnohem lepší je provést testování pomocí testů teoretických. Teoretické výsledky nám s předstihem poví, jak empirické testy dopadnou. Dají nám mnohem lepší představu o samotném generování než empirické testy zkoumající pouze výsledek stylem pokus a omyl. Zabývají se vnitřními algoritmy a vlivem, jaký na ně mají jejich parametry. Teoretické testy obecně vždy zkoumají celou periodu generátoru, vynikají tedy ve zkoumání globální náhodnosti. Vytvořit teoretický test však není nic jednoduchého. Teoretické testy jsou mnohem složitější než empirické testy a nejsou univerzálně použitelné na všechny generátory. Zde těmito testy podrobně zabývat nebudeme. Za zmínku ovšem stojí minimálně spektrální test. Dle [23] se jedná o zatím nejmocnější nástroj pro hodnocení RNG. Test se zaměřuje

na špičky diskrétní Fourierovy transformace zkoumané sekvence. Smyslem je nalezení periodických rysů (opakujících se vzorců, které od sebe nejsou daleko). Tyto periodické vlastnosti indikují odchylku od předpokládané náhodnosti. [23, 24]

4.2 Hodnocení ostatních vlastností generátorů

Mezi další důležité vlastnosti RNG patří především jejich hardwarové nároky, rychlost generování a kryptografická bezpečnost. Všechny jsou provázané a přirozeně závisí hlavně na teorii stojící za funkcí generátoru.

Hardwarové nároky

Nároky na hardware jsou velmi rozdílné pro PRNG a TRNG. Většina „true" generátorů se totiž neobejde bez speciálního HW určeného ke sběru a prvotnímu zpracování entropie. Nároky takových generátorů potom přímo odpovídají navrženým elektronickým obvodům.

Hardwarové nároky PRNG spočívají ve velikostech pamětí a procesorových výpočtech. Základním požadavkem na paměť je počet bytů, které zabírá samotný generátor v úložišti. Důležitým požadavkem je velikost pracovních registrů. Alogritmus generátoru může být například optimalizován pro práci se 32 bitovými proměnnými. Pokud daný hardware obsahuje pouze 16 bitové registry, nastávají komplikace v podobě zpomalení generátoru. Daná CPU musí data obsažená v proměnné generátoru rozdělovat do většího počtu registrů. U implementací zaměřených na rychlost generování může být toto zpomalení znatelné. Celková velikost pracovních dat generátoru v bytech je také důležitá, v extrémních případech by se tyto data nemusela do použitelných registrů vejít. Další hardwarové nároky PRNG můžeme pojmenovat jako nároky vykonávaných operací. Jednoduché bitové operace jako například bitový posuv jsou mnohem méně náročné, než aritmetické operace násobení a sčítání. Náročnost operací spočívá např. v potřebě použití pomocných odkládacích registrů pro dílčí mezivýsledky a v konečném součtu znamená nárůst počtu taktů CPU potřebných pro vykonání této operace. Růst těchto nároků znamená pokles rychlosti generátoru. Celkové hardwarové nároky PRNG můžeme tedy vyjádřit velikostí samotného generátoru (v bytech), velikostí pracovních dat generátoru (v bytech) a v počtu a typu potřebných operací pro vygenerování jednoho výstupního čísla.

Velikost generátoru většinou není problém a navíc je pro velkou část aplikací zabudován uvnitř jiného softwaru. Samotná velikost proto není pro ohodnocení generátoru příliš vypovídající.

Rychlost generování

Rychlost generování je většinou vyšší u PRNG než u TRNG. Obecně je udávaná jako schopnost generátoru vyprodukovat určité množství náhodných dat za jednotku času. Rychlost generátoru závisí na použitém hardwaru a složitosti generátoru. Pro TRNG použitý hardware přímo odpovídá složitosti generátoru. Získání entropie a její kvalitní zpracování vyžaduje čas, jenž se jen těžko dá snižovat. Snižováním potřebného času zjednodušováním hardwaru zasahujeme do kvality generovaných čísel, to může a nemusí být vhodné. Využití lepší hardwarové implementace pro zvýšení rychlosti není vždy možné. Pokud to možné je, pak mohou být překážkou náklady.

PRNG mají v tomto ohledu výhodu. Hardware CPU a jejich nejbližších periferií využívaný pro generování bude vždy rychlejší. Tyto obvody optimalizované pro práci na vysokých kmitočtech nepotřebují čekat na vnější zdroj entropie a mohou pracovat svojí plnou rychlostí. Rychlost generování je přímo závislá na hardwarových nárocích generátoru, resp. na počtu a náročnosti prováděných operací. Náročnější operace potřebují pro své vykonání větší počet taktů CPU a tím snižují celkovou rychlost. Implementace zaměřené na rychlost generování se snaží nahradit co největší počet složitých operací těmi jednoduššími. Delší implementace s větším počtem jednodušších operací potom může být mnohem rychlejší. Zvýšení rychlosti můžeme někdy dosáhnout optimalizací zdrojového kódu pro hardwarovou vrstvu. Například funkce dělení modulo je pro CPU velice náročná. Pokud jako hodnotu modulo však vezmeme délku bitového slova daného hardwaru, bude se tato funkce provádět implicitně a dojde k razantnímu zvýšení rychlosti.

Kryptografické zabezpečení

Pro použití RNG v šifrovacích aplikacích je důležité jejich kryptografické zabezpečení. Dokonale kryptograficky bezpečný generátor je takový, jehož výstupní čísla nelze žádným způsobem předvídat. Takovou podmínku nemohou PRNG nikdy splňovat, přesto se ale jisté implementace dají považovat za dostatečně zabezpečené. Takové generátory se označují jako CSPRNG. CSPRNG musí splňovat dvě podmínky. První

podmínkou je úspěšného složení testu příštího bitu, čímž je zaručeno úspěšné složení všech ostatních polynomiálních statistických testů náhodnosti [25]. Druhou podmínkou je odolnost vůči útokům, při kterých je odhalena (uhodnuta) část, nebo celý vnitřní stav generátoru. V takovém případě musí být nemožné rekonstruovat sekvenci generovanou před tímto útokem. Skutečně kryptograficky bezpečné mohou být pouze TRNG. Neplatí však, že každý „true“ generátor je bezpečný. Bezpečnost je nutno začlenit do návrhu hardwaru generátoru. Musí být odstraněny veškeré rušivé vlivy, které by mohly výstup generátoru ovlivňovat předvídatelným způsobem. Ve většině aplikací jsou využívány CSPRNG, které periodicky dostávají entropii získávanou TRNG, který nemusí být nutně kryptograficky bezpečný. CSPRNG však v případě používání nezabezpečeného zdroje entropie musí splňovat další podmínku. I při znalosti hodnot vstupujících do generátoru nesmí být možné předpovědět generované sekvence. Více informací k tématu kryptografie lze najít v [3].

5 Implementace a testování generátorů

V této kapitole jsou stručně popsané použité implementace PRNG a jejich následné testování. Implementace je možné nalézt v elektronické příloze a to jako spustitelnou konzolovou aplikaci (BP_RNGs.exe), nebo jako projekt se zdrojovými kódy. Konzolová aplikace byla využita k testování. Jejím standardním výstupem jsou generované hodnoty typu *integer*. Výběr požadovaného generátoru (LFSR, LCG, PCG) je realizován spouštěcím parametrem aplikace.

5.1 Implementace vybraných pseudonáhodných generátorů

Všechny PRNG v této kapitole s výjimkou v podobě MT jsou implementovány v jazyce C#. Mersenne Twister je realizován v jazyce C++.

5.1.1 Lineární zpětnovazební posuvný registr (LFSR)

Pro implementaci LFSR a intuitivnější práci s bitovými operacemi je zvoleno využití řad typu *bool* (tedy typ odpovídající binárním proměnným). Dále je využitá systémová třída *Convert* pro konverzi právě mezi řadami typu *bool* a použitými čísly typu *integer*. Tato konverze probíhá pomocí řetězců obsahujících znaky ‘1’ a ‘0’. Tato funkce je zobrazena i v ukázce kódu pod tímto odstavcem. Jako semínko lze využít čas, nebo uživatelem dodanou hodnotu (v podobě řetězce vyjadřující binární číslo). Algoritmus generátoru je vyjádřen ve funkci *Shift()* jako určení vstupního bitu do registru porovnáváním vybraných členů řady (bitů v registru) simulujících funkci XOR. Také obsahuje jednoduchou smyčku pro posun členů v řadě (tedy bitů v registru). Funkční polynom generátoru ($x^{32}+x^{30}+x^{26}+x^{25}+1$) byl vybrán z [26]. Funkce je také zobrazena v ukázce kódu pod tímto odstavcem, kompletní zdrojový kód viz elektronická příloha.

```
/* ---- Ukázka získání seed z času a jeho konverze do řady bool bits[] ---- */

      long time = DateTime.Now.Ticks;
      string seed = Convert.ToString(time, 2);

      for (int i = 0; i < seed.Length; i++)
      bits[i] = seed[i] == '1' ? true : false;
```

```
/* ---- Funkce algoritmu LFSR (32 bitový) ---- */


 public void Shift()
  {
    bool bnew1 = !(bits[bitsRNG - 1] == bits[bitsRNG - 3]); // xor bitů 32 a 30
    bool bnew2 = !(bits[bitsRNG - 7] == bits[bitsRNG - 8]); // xor bitů 26 a 25
    bool bnew = !(bnew1 == bnew2);                        // xor výsledných bitů

    for (int i = bits.Length - 1; i > 0; i--) // posun registru o jednu pozici
       {
         bits[i] = bits[i - 1];
       }
    bits[0] = bnew;              // nový bit na začátek registru
  }
```

5.1.2 Lineární kongruentní generátor (LCG)

Implementace jednoduchého LCG je opravdu velice krátká. Samotný algoritmus je vyjádřen pomocí jednoho řádku zdrojového kódu a obsahuje tři matematické operace. Těmito operacemi jsou násobení, sčítání a dělení modulo. Pro výpočet jednoho výstupního čísla generátoru je každá z těchto tří operací použita jednou. Hodnoty konstant byly použity dle [27]. Jako semínko lze využít čas, nebo uživatelem zvolenou hodnotu. Kompletní zdrojový kód viz elektronická příloha, zde jako ukázka zdrojového kódu použité konstanty a funkce pro vrácení nové generované hodnoty.

```
    /* ---- Nastavení konstant generátoru ---- */

             private const long m = 4294967291;
             private const long a = 1345659;
             private const long c = 1013904223;


    /* ---- Funkce vracející novou generovanou hodnotu ---- */

           private long state;

            public long Next()
            {
                state = ((a * state) + c) % m;
                return state;
             }
```

5.1.3 Permutovaný kongruentní generátor (PCG)

V této práci byla využita mírně upravená C# implementace minimalistické verze PCG (původně v jazyce C). Kompletní implementace je přístupná zde [28], nebo v elektronické příloze zahrnuta do kompletního zdrojového kódu. Funkce generování následujícího čísla je zobrazena v kódu pod odstavcem. Obsahuje matematické operace násobení a sčítání, které vypočítají novou hodnotu vnitřního stavu ze stavu předchozího. Dále provádí permutační funkci rotace v podobě bitových posunů a funkce XOR. Permutační funkce je provedena na vnitřním stavu z minulého cyklu a neovlivňuje tak nově vytvořený vnitřní stav.

```
/* ---- Funkce generující následující psuedonáhodné číslo ---- */

    public uint Random32()
    {
           ulong oldState = m_state;

           m_state = oldState * 6364136223846793005UL + m_inc;

           uint xorShifted = (uint)(((oldState >> 18) ^ oldState) >> 27);
           int rot = (int)(oldState >> 59);

           return (xorShifted >> rot) | (xorShifted << ((-rot) & 31));
    }
```

5.1.4 Mersenne Twister (MT)

Vzhledem k velikému rozšíření a oblíbenosti generátoru MT není překvapením, že ho zvolená testovací knihovna PractRand [29] také obsahuje mezi vlastními implementovanými generátory. Pro využití implementace obsažené v testovací knihovně bylo rozhodnuto k získání reference pro vykonané testy na externě implementovaných generátorech. Jak už bylo řečeno na začátku kapitoly, stejně jako celá knihovna testů je tento MT19937 implementovaný v jazyce C++. Kompletní zdrojový kód lze nalézt v knihovně v [29] nebo v elektronické příloze této práce.

5.2 Testování implementovaných generátorů

Pro testování RNG se často využívají předpřipravené softwarové balíky statistických testů, většinou obsahujících také implementace vybraných generátorů. Nejznámějšími balíky testů jsou testy Diehard [30], NIST Statistical Test Suite [31], nebo TestU01[32]. Diehard byly dříve velmi využívané, dnes už jsou zastaralé a jejich použití pro kvalitní ohodnocení generátorů není moc vhodné. NIST STS je soubor testů zaměřený hlavně na kryptografickou bezpečnost testovaných generátorů. Knihovna TestU01 je pokročilejším souborem testů, vytvořena jako obecný soubor nástrojů pro testování RNG.

Pro testování generátorů v této práci je využitá testovací sada PractRand_0.92 [29]. Tato sada obsahuje nejen knihovnu testů a implementovaných generátorů, ale také předpřipravenou testovací aplikaci. Právě tato konzolová testovací aplikace byla nakonec zvolena pro vykonání testů. Kompletní dokumentace PractRand může být nalezena v [29] nebo v elektronické příloze práce.

5.2.1 Testovací aplikace

Testovací aplikace (RNG_test.exe) umožňuje testování vnitřně implementovaných generátorů, nebo externích generátorů, jejichž výstup je pomocí pipe propojen do standardního vstupu aplikace. Řízení aplikace probíhá pomocí spouštěcích parametrů, základním parametrem je název generátoru (při použití externího se nahrazuje parametrem *stdin*). Další parametry již nejsou nutně pro spuštění testu potřeba a slouží k nastavení testů a zobrazení výsledků. Seznam použitých parametrů při testování:

- *-tlfail* … Testovací aplikace přeruší testování po zobrazení mezivýsledků obsahujících neúspěšný výsledek.

- *-ttnormal* … Testování výsledných generovaných sekvencí generátorů za pomocí standardní sady testů.

- *-e 0.1* … Nastavení prahu testů. Nastavuje inteligentní hodnotu p = 0,1 (viz *4.1.1*) pro zobrazení očekávaného počtu výsledků testů odpovídajících nastavenému prahu.

- *-tf 0* … Nastavení skládání testů. Toto nastavení podrobuje testům pouze surová výstupní data generátoru (bez jejich transformací).

- *-tlmax 32GB* … Nastavení maximální délky testovaných dat (využito pouze pro testování interního generátoru mt19937).

Všechny parametry testovací aplikace včetně jejich popisu lze získat spuštěním testovací aplikace s parametrem *–help*. Seznam testů standardního setu využitého k testování, včetně stručného popisu [29]:

- *BCFN* … Kontrola lineárních korelací pro velké vzdálenosti (počítání bitů).

- *DC6* … Kontrola lineárních korelací pro malé vzdálenosti (počítání bitů)

- *Gap16* … Variace klasického testu mezer.

- *BRank* … Klasický test hodnosti binární matice (viz [23]).

- *FPF* … Test frekvence pohyblivé řádové čárky. Navzdory jménu pracuje tento test pouze s celými čísly. Kontroluje lineární korelace na velmi krátkých vzdálenostech (kratších než test DC6), zvláště pro korelace obsahujících hodně nulových bitů.

5.2.2 Výsledky testů

Testům byly podrobeny čtyři generátory. Tři jsou implementovány v samostatné aplikaci, jejíž výstup byl propojen se vstupem testovací aplikace. Pro ověření vstupu dat do testovací aplikace byly testy provedeny i způsobem, kdy byl výstup generátorů uložen do samostatného souboru a tento soubor následně tvořil vstup testovací aplikace. Čtvrtý generátor implementovaný uvnitř testovací aplikace tvoří jistou referenci k výsledkům testů ostatních generátorů. Pro ověření byly testy generátorů provedeny opakovaně, všechny se stálými výsledky.

Celé výsledky jsou obsaženy v přílohách. Generátory LFSR (Příloha 1) a LCG (Příloha 2) dle předpokladů testy neprochází. Testovací aplikace ukončí testování již po 8MB zpracovaných dat díky neúspěšnému složení většiny testů. Zajímavější výsledky by

měly dle předpokladů poskytnout testy generátoru PCG (Příloha 3), ovšem stejně jako první dva testy končí velice brzo neúspěšně. Testy jsou také ukončeny již po 8MB zpracovaných dat neúspěšným výsledkem. Testy vnitřně implementovaného generátoru MT19937 vykazují výsledky zcela odlišné. Pro zvolenou maximální velikost 32GB testovaných dat generátor testy úspěšně splňuje. Pouze několik dílčích testů vykazuje výsledky hodnocené jako neobvyklé, což znamená výsledky vymykající se očekávaným hodnotám, ale stále bezpečně uvnitř nastaveného prahu výsledků. Posouzení těchto testů včetně jejich vypovídající hodnoty v kapitole závěr.

Závěr

Cílem práce bylo analyzování technik pro generování náhodných a pseudonáhodných čísel a zvolení vhodné metody hodnocení kvality těchto generátorů. Jak jsme zjistili v první kapitole, až překvapivé množství činností v oblasti počítačové techniky náhodná čísla využívá. Zabezpečení těchto systémů by bez náhodných čísel nebylo stávajícím způsobem vůbec možné. V první kapitole jsme se také dozvěděli o rozdělení problému generování náhodných čísel na dvě kategorie, generování pseudonáhodných čísel a generování skutečně náhodných čísel. Druhá a třetí kapitola poté sloužila k bližšímu pohledu na tyto dvě rozdílné činnosti. Zjistili jsme podrobnosti vybraných generátorů pseudonáhodných i náhodných čísel a dověděli se o existenci většího počtu rozdílných způsobů generování těchto čísel. V další kapitole byly jako nejobjektivnější metoda hodnocení kvality generátorů stanoveny empirické (statistické) testy. Tyto testy lze využít na všechny dostupné generátory, ať už pseudonáhodných, nebo náhodných čísel. Hodnotí statistickou náhodnost generovaných sekvencí, dávají nám tak možnost vyčíslit množství generované náhodnosti. Kromě empirických testů jsme se v této kapitole dověděli také o ostatních vlastnostech generátorů, které mohou být důležité pro konkrétní aplikaci a zároveň podle nich nelze generátory jednoznačně a univerzálně hodnotit.

V poslední části jsou stručně popsané jednotlivé implementace vybraných generátorů. Tyto popisy také většinou obsahují úryvky zdrojových kódů jako doplnění k popisu. Část implementací je originálních, část implementací byla převzata z volně přístupných zdrojů dalších autorů. Tyto implementované generátory byly podrobeny empirickým testům, které slouží ke kvalitativnímu ohodnocení generátorů. Stručně vypsané výsledky testů se také v této kapitole dají nalézt, kompletní výsledky jsou potom obsaženy v přílohách. Vyhodnocení výsledků následuje dále.

Nyní k vyhodnocení implementovaných generátorů podle hardwarových nároků. Hodnocení implementovaných generátorů dle hardwarových nároků zohledňuje dvě vlastnosti. První vlastností je velikost funkční datové paměti generátoru a druhou počet a druh operací prováděných pro vygenerování jednoho výstupního čísla. Do hodnocení nejsou zahrnuty smyčky, jejich provedením je generováno více čísel. Implementace LFSR potřebuje pracovní paměť 115 bytů při započítání všech proměnných a konstant. Pro vygenerování jednoho čísla je třeba provést celkem 95 operací, z čehož je 37 aritmetických

operací. LCG potřebuje pro svoji práci pouze 32 bytů, přičemž vygenerování jednoho čísla znamená 3 operace. Všechny tyto tři operace jsou aritmetické, operace modulo je velmi náročná. Generátor PCG potřebuje paměť celkem 56 bytů. Pro vygenerování jednoho čísla je třeba 10 operací, z čehož je 8 jednodušších bitových operací a dvě náročnější aritmetické operace. Pro implementaci MT19937 obsaženou v souboru PractRand se mi nepodařilo zjistit přesný počet použitých operací, ani použitou paměť. U paměti se ovšem jedná minimálně o 2496 bytů při započítání velikosti vnitřního stavu tohoto generátoru. Z pohledu nároků na hardware vychází nejlépe generátor LCG, i přes použití složité operace modulo. Nejnáročnější je s jistotou implementace MT19937.

Kvalitativního hodnocení implementovaných generátorů je dosaženo prozkoumáním výsledků provedených empirických testů. Špatné výsledky generátorů LFSR a LCG jsou očekávané. Ovšem špatný výsledek generátoru PCG, který je srovnatelný právě s výsledky generátorů LFSR a LCG je podezřelý. Jako reference slouží výsledky testů generátoru MT implementovaného přímo v testovací aplikaci. Tento generátor testy pro nastavený rozsah dat úspěšně splňuje a to navzdory předpokládaným horším výsledkům oproti PCG. Vychází tedy jako kvalitativně nejlepší ze všech čtyřech generátorů. Velice podezřelá je přílišná korelace mezi výsledky testů externě implementovaných generátorů, viz přílohy. Výsledky jsou téměř totožné. Nabízí se tedy otázka věrohodnosti dosažených výsledků. Téměř s jistotou můžeme prohlásit, že výsledky jsou chybné. Silný vliv může mít zpracování externích generátorů. Problémy mohou být obsaženy v podobě formátu vstupních dat testovací aplikace a formátu výstupních dat generátorů. Vhodnější metodou testování těchto generátorů by byla implementace knihovny testů a implementace testovaných generátorů do společné aplikace. Jakýkoliv potenciální problém v komunikaci by byl vyřešen. Bohužel pro zadaný časový rámec práce již nebyla tato možnost časově realizovatelná a optimalizování komunikace mezi generátory a testovací aplikací nevedlo ke změně výsledků. Práce by se tedy dala rozšířit o implementaci testů a generátorů do společné aplikace. Dále provedení testů a získání nových výsledků, které by mohly zde obsažené velice podezřelé výsledky potvrdit nebo vyvrátit.

Seznam literatury a informačních zdrojů

[1] BENNETT, Deborah J. *Randomness*. B.m.: Harvard University Press, 1998. ISBN 9780674107465.

[2] BIEBIGHAUSER, Dan. Testing Random Number Generators. *University of Minnesota*. 2000, s. 15.

[3] GARRETT, Paul. *Making Breaking Codes An Introduction to Cryptography*. B.m.: Prentice Hall, 2000.

[4] HAAHR, Mads. *Introduction to Randomness and Random Numbers* [online]. Dostupné z: https://www.random.org/randomness/

[5] WEISSTEIN, Eric W. *Monte Carlo Method* [online]. [vid. 1. červen 2016]. Dostupné z: http://mathworld.wolfram.com/MonteCarloMethod.html

[6] SUN, Yi a Ilya TIMOFEYEV. Kinetic Monte Carlo simulations of 1D and 2D traffic flows: Comparison of two look-ahead potentials. 2013.

[7] MACGILLIVRAY, H. T. a R. J. DODD. Monte-Carlo simulations of galaxy systems. *Astrophysics and Space Science* [online]. 1982, roč. 86, č. 2, s. 437–452 [vid. 24. květen 2016]. ISSN 0004-640X. Dostupné z: doi:10.1007/BF00683347

[8] *GNU C library file* [online]. Dostupné z: https://github.com/lattera/glibc/blob/master/stdlib/random_r.c

[9] SELINGER, Peter. *The GLIBC random number generator* [online]. 2007. Dostupné z: http://www.mathstat.dal.ca/~selinger/random/

[10] *Microsoft Reference Source .NET Framework 4.6.1* [online]. Dostupné z: http://referencesource.microsoft.com/#mscorlib/system/random.cs

[11] *Microsoft Developer Network - Random Class* [online]. Dostupné z: https://msdn.microsoft.com/en-us/library/system.random.aspx

[12] *Random (Java Platform SE 8)* [online]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

[13] KLEIN, Andreas. Linear Feedback Shift Registers. In: *Stream Ciphers* [online]. London: Springer London, 2013 [vid. 1. květen 2016], s. 17–58. Dostupné z: doi:10.1007/978-1-4471-5079-4_2

[14] SALUJA, Kewal K. *Linear Feedback Shift Registers Theory and Applications.* 1987

[15] CALDWELL, Chris K. *Mersenne Primes: History, Theorems and Lists* [online]. Dostupné z: https://primes.utm.edu/mersenne/

[16] MATSUMOTO, Makoto a Takuji NISHIMURA. Mersenne twister: a 623-

dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* [online]. 1998, roč. 8, č. 1, s. 3–30. ISSN 10493301. Dostupné z: doi:10.1145/272991.272995

[17] JAGANNATAM, Archana. Mersenne Twister – A Pseudo Random Number Generator and its Variants [online]. 2008. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.175.9735&rep=rep1&type=pdf

[18] SAVORY, Paul. *Poor Statistical Qualities for the RANDU Random Number Generator* [online]. Dostupné z: http://demonstrations.wolfram.com/PoorStatisticalQualitiesForTheRANDURandomNumberGenerator/

[19] O'NEILL, Me. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. *Pcg-Random.Org* [online]. nedatováno, roč. V, č. 212, s. 1–46. Dostupné z: http://www.pcg-random.org/pdf/toms-oneill-pcg-family.pdf

[20] ROBSON, Stewart. A Ring Oscillator Based Truly Random Number Generator by. 2013.

[21] NING, Liao, Jiang DING, Bai CHUANG a Zou XUECHENG. Design and validation of high speed true random number generators based on prime-length ring oscillators. *The Journal of China Universities of Posts and Telecommunications* [online]. 2015, roč. 22, č. 4, s. 1–6. ISSN 10058885. Dostupné z: doi:10.1016/S1005-8885(15)60661-6

[22] COLEŞA, Adrian, Radu TUDORAN a Sebastian BÂNESCU. Software random number generation based on race conditions. *Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2008* [online]. 2008, č. October, s. 439–444. Dostupné z: doi:10.1109/SYNASC.2008.36

[23] KNUTH, Donald E. *The Art of Computer Programming, vol. 2*. 2. vyd. 1981.

[24] RUKHIN, Andrew, Juan SOTO, James NECHVATAL, Miles SMID, Elaine BARKER, Stefan LEIGH, Mark LEVENSON, Mark VANGEL, David BANKS, Alan HECKERT, James DRAY a San VO. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. *National Institute of Standards and Technology (NIST)*. 2010, roč. 800-22, č. Rev 1a, s. 131.

[25] YAO, Andrew C. Theory and Applications of Trapdoor Functions. nedatováno.

[26] WARD, Roy a Tim MOLTENO. Table of Linear Feedback Shift Registers. 2007.

[27] L 'ECUYER, Pierre. TABLES OF LINEAR CONGRUENTIAL GENERATORS OF DIFFERENT SIZES AND GOOD LATTICE STRUCTURE. 1999, roč. 68, č. 22599, s. 249–260.

[28] HARRIS, Kevin. *GitHub - PCG-Csharp code* [online]. 2015. Dostupné z: https://github.com/Tectorum/pcg-csharp

[29] *PractRand library* [online]. Dostupné z: http://pracrand.sourceforge.net/

[30] MARSAGLIA, George. *DIEHARD Battery of Tests of Randomness* [online]. 1995. Dostupné z: http://stat.fsu.edu/pub/diehard/

[31] *NIST - RANDOM NUMBER GENERATION* [online]. 2014. Dostupné z: http://csrc.nist.gov/groups/ST/toolkit/rng/index.html

[32] SIMARD, Richard. *TestU01* [online]. 2009. Dostupné z: http://www.iro.umontreal.ca/~simardr/testu01/tu01.html

Přílohy

Příloha 1 – Výsledky testů generátoru LFSR

```
D:\Data\BP_RNGs\BP_RNGs\bin\Debug>BP_RNGs lfsr | RNG_test stdin -tlfail
-ttnormal -e 0.1 -tf 0
RNG = RNG_stdin, PractRand version 0.92, seed = 0x57047fa4
test set = normal, folding = none

rng=RNG_stdin, seed=0x57047fa4
length= 8 megabytes (2^23 bytes), time= 2.6 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-5,T)                  R=+746455 p = 0           FAIL !!!!!!!!
  BCFN(2+1,13-5,T)                  R=+608579 p = 0           FAIL !!!!!!!!
  BCFN(2+2,13-6,T)                  R=+458938 p = 0           FAIL !!!!!!!!
  BCFN(2+3,13-6,T)                  R=+241110 p = 0           FAIL !!!!!!!!
  BCFN(2+4,13-6,T)                  R=+118670 p = 0           FAIL !!!!!!!!
  BCFN(2+5,13-7,T)                  R=+71158  p = 0           FAIL !!!!!!!!
  BCFN(2+6,13-8,T)                  R=+41820  p = 0           FAIL !!!!!!!!
  BCFN(2+7,13-8,T)                  R=+20745  p =  7e-5266    FAIL !!!!!!!!
  BCFN(2+8,13-9,T)                  R=+11858  p =  1e-2665    FAIL !!!!!!!!
  BCFN(2+9,13-9,T)                  R= +5896  p =  5e-1326    FAIL !!!!!!!!
  DC6-9x1Bytes-1                    R=+141193 p = 0           FAIL !!!!!!!!
  Gap-16:A                          R=+181050 p = 0           FAIL !!!!!!!!
  Gap-16:B                          R=+1178808 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:(0,14-2)              R=+303993 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(1,14-2)              R=+127713 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(2,14-3)              R=+87180  p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(3,14-4)              R=+106253 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(4,14-5)              R=+127701 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:all                   R=+355708 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:all2                  R=+57126395720 p = 0      FAIL !!!!!!!!

  FPF-14+6/16:cross                 R=+131444 p = 0           FAIL !!!!!!!!
  BRank(12):128(1)                  R= +1282  p~=  8.0e-387   FAIL !!!!!!!
  BRank(12):256(1)                  R= +2743  p~=  1.0e-826   FAIL !!!!!!!
  BRank(12):384(0)                  R= +2973  p~=  6.1e-896   FAIL !!!!!!!
  BRank(12):512(0)                  R= +4006  p~=  5e-1207    FAIL !!!!!!!!
```

Příloha 2 – Výsledky testů generátoru LCG

```
D:\Data\BP_RNGs\BP_RNGs\bin\Debug>BP_RNGs lcg | RNG_test stdin -tlfail
-ttnormal -e 0.1 -tf 0
RNG = RNG_stdin, PractRand version 0.92, seed = 0x2be55931
test set = normal, folding = none

rng=RNG_stdin, seed=0x2be55931
length= 8 megabytes (2^23 bytes), time= 2.2 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-5,T)                  R=+765283 p = 0          FAIL !!!!!!!!
  BCFN(2+1,13-5,T)                  R=+619440 p = 0          FAIL !!!!!!!!
  BCFN(2+2,13-6,T)                  R=+473674 p = 0          FAIL !!!!!!!!
  BCFN(2+3,13-6,T)                  R=+241949 p = 0          FAIL !!!!!!!!
  BCFN(2+4,13-6,T)                  R=+118686 p = 0          FAIL !!!!!!!!
  BCFN(2+5,13-7,T)                  R=+71158  p = 0          FAIL !!!!!!!!
  BCFN(2+6,13-8,T)                  R=+41820  p = 0          FAIL !!!!!!!!
  BCFN(2+7,13-8,T)                  R=+20745  p =  7e-5266   FAIL !!!!!!!!
  BCFN(2+8,13-9,T)                  R=+11858  p =  1e-2665   FAIL !!!!!!!!
  BCFN(2+9,13-9,T)                  R= +5896  p =  5e-1326   FAIL !!!!!!!!
  DC6-9x1Bytes-1                    R=+145394 p = 0          FAIL !!!!!!!!
  Gap-16:A                          R=+186716 p = 0          FAIL !!!!!!!!
  Gap-16:B                          R=+1215862 p = 0         FAIL !!!!!!!!
  FPF-14+6/16:(0,14-2)              R=+294193 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:(1,14-2)              R=+146810 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:(2,14-3)              R=+97777  p = 0          FAIL !!!!!!!!
  FPF-14+6/16:(3,14-4)              R=+111423 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:(4,14-5)              R=+133854 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:all                   R=+367874 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:all2                  R=+58658380929 p = 0     FAIL !!!!!!!!

  FPF-14+6/16:cross                 R=+121071 p = 0          FAIL !!!!!!!!
  BRank(12):128(1)                  R= +1769  p~=  1.8e-533  FAIL !!!!!!!
  BRank(12):256(1)                  R= +3717  p~=  5e-1120   FAIL !!!!!!!!
  BRank(12):384(0)                  R= +4006  p~=  5e-1207   FAIL !!!!!!!!
  BRank(12):512(0)                  R= +5384  p~=  9e-1622   FAIL !!!!!!!!
```

Příloha 3 – Výsledky testů generátoru PCG

```
D:\Data\BP_RNGs\BP_RNGs\bin\Debug>BP_RNGs pcg | RNG_test stdin -tlfail
-ttnormal -e 0.1 -tf 0
RNG = RNG_stdin, PractRand version 0.92, seed = 0xd62ec9ae
test set = normal, folding = none

rng=RNG_stdin, seed=0xd62ec9ae
length= 8 megabytes (2^23 bytes), time= 2.1 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+0,13-5,T)                  R=+761957 p = 0           FAIL !!!!!!!!
  BCFN(2+1,13-5,T)                  R=+619007 p = 0           FAIL !!!!!!!!
  BCFN(2+2,13-6,T)                  R=+472983 p = 0           FAIL !!!!!!!!
  BCFN(2+3,13-6,T)                  R=+241893 p = 0           FAIL !!!!!!!!
  BCFN(2+4,13-6,T)                  R=+118686 p = 0           FAIL !!!!!!!!
  BCFN(2+5,13-7,T)                  R=+71158  p = 0           FAIL !!!!!!!!
  BCFN(2+6,13-8,T)                  R=+41820  p = 0           FAIL !!!!!!!!
  BCFN(2+7,13-8,T)                  R=+20745  p =  7e-5266    FAIL !!!!!!!!
  BCFN(2+8,13-9,T)                  R=+11858  p =  1e-2665    FAIL !!!!!!!!
  BCFN(2+9,13-9,T)                  R= +5896  p =  5e-1326    FAIL !!!!!!!!
  DC6-9x1Bytes-1                    R=+145293 p = 0           FAIL !!!!!!!!
  Gap-16:A                          R=+186717 p = 0           FAIL !!!!!!!!
  Gap-16:B                          R=+1215878 p = 0          FAIL !!!!!!!!
  FPF-14+6/16:(0,14-2)              R=+294615 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(1,14-2)              R=+146242 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(2,14-3)              R=+97542  p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(3,14-4)              R=+111754 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:(4,14-5)              R=+133925 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:all                   R=+367831 p = 0           FAIL !!!!!!!!
  FPF-14+6/16:all2                  R=+58705972604 p = 0      FAIL !!!!!!!!

  FPF-14+6/16:cross                 R=+121625 p = 0           FAIL !!!!!!!!
  BRank(12):128(1)                  R= +1769  p~=  1.8e-533   FAIL !!!!!!!
  BRank(12):256(1)                  R= +3717  p~=  5e-1120    FAIL !!!!!!!!
  BRank(12):384(0)                  R= +4006  p~=  5e-1207    FAIL !!!!!!!!
  BRank(12):512(0)                  R= +5384  p~=  9e-1622    FAIL !!!!!!!!
```

Příloha 4 – Výsledky testů generátoru MT19937

```
D:\Data\BP_RNGs\BP_RNGs\bin\Debug>RNG_test mt1993
7 -tlfail -ttnormal -e 0.1 -tf 0 -tlmax 32GB
RNG = mt19937, PractRand version 0.92, seed = 0x1ac6ec56
test set = normal, folding = none

rng=mt19937, seed=0x1ac6ec56
length= 256 megabytes (2^28 bytes), time= 2.4 seconds
  no anomalies in 48 test result(s)

rng=mt19937, seed=0x1ac6ec56
length= 512 megabytes (2^29 bytes), time= 4.9 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R=  -4.0  p =1-6.6e-3   unusual
  ...and 49 test result(s) without anomalies

rng=mt19937, seed=0x1ac6ec56
length= 1 gigabyte (2^30 bytes), time= 9.9 seconds
  no anomalies in 53 test result(s)

rng=mt19937, seed=0x1ac6ec56
length= 2 gigabytes (2^31 bytes), time= 19.7 seconds
  Test Name                         Raw       Processed     Evaluation
  Gap-16:B                          R=  +4.1  p =  2.0e-3   unusual
  ...and 55 test result(s) without anomalies

rng=mt19937, seed=0x1ac6ec56
length= 4 gigabytes (2^32 bytes), time= 38.1 seconds
  no anomalies in 58 test result(s)

rng=mt19937, seed=0x1ac6ec56
length= 8 gigabytes (2^33 bytes), time= 76.9 seconds
  no anomalies in 61 test result(s)

rng=mt19937, seed=0x1ac6ec56
length= 16 gigabytes (2^34 bytes), time= 153 seconds
  no anomalies in 64 test result(s)

rng=mt19937, seed=0x1ac6ec56
length= 32 gigabytes (2^35 bytes), time= 295 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R=  -4.6  p =1-3.4e-3   unusual
  ...and 65 test result(s) without anomalies
```