KTH Computer Science
and Communication

Analys av pseudoslumptalsalgoritmer

Linjära kongruensgeneratorer, återkopplande skiftregister och AES-baserade pseudoslumptalsgeneratorer

HANNES SALIN

Examenrapport vid NADA
Handledare: Mikael Goldmann
Examinator: Mats Dam

Referat

Detta arbete handlar om pseudoslumptalsgenerering i ett jämförande och analyserande perspektiv. I fokus står tre olika typer av algoritmer: en linjär kongruensgenerator, ett linjärt återkopplande skiftregister samt en AES-baserad psuedoslumpgenerator. Av dessa tre har de två senare implementerats på egen hand i C++ och kongruensgeneratorn är en implementation funnen på Internet (upphovsman Robin Whittle[34]).

Denna jämförelse syftar till att undersöka huruvida olika pseudoslumptalsgeneratorer skiljer sig åt i prestanda och vilken grad av statistisk slumpmässighet dess utdata har. Detta för att påvisa att man bör välja rätt pseudoslumptalsgenerator till rätt sammanhang.

Prestandatestningen genomfördes genom att mäta tiden för respektive algoritm att generera 10^3, 10^6, 10^7 samt 10^9 heltal (dvs. ca 4 kB, 4 Mb, 40 Mb samt 4 Gb slumpdata) . En enklare statistisk testning gjordes med testbatteriet ENT som tillhandahåller fem olika statistiska test för mätning av slumpmässighet i tal/bit-sekvenser. Med de mätdata som följde gjordes en analys över skillnaderna algoritmerna emellan. Resultatet blev att av de tre testade algoritmerna anses AES-baserade PRNG vara den bästa algoritmen ur ett säkerhetsperspektiv då den hade bäst statistiska värden. Bortser man från säkerhetskrav krav anses det linjära återkopplande skiftregistret vara mest användbart ur ett prestandamässigt perspektiv då dess snabbhet och relativa statistiska egenskaper var mycket goda.

Abstract

This essay deals with pseudorandom number generation in a comparative and analytical perspective. The focus is on three different types of algorithms: a linear congruence generator, a linear feedback shift register and an AES-based psuedosrandom generator. Of these three, the latter two are implemented in C++ by the author and the linear congruence generator is an implementation found on the Internet (author Robin Whittle[34])

This comparison aims to investigate whether different pseudorandom number generators differ in performance and the degree of statistical randomness of its output. This is to demonstrate that one should choose the right pseudorandom number generator to the right context.

Performance testing was conducted by measuring the time for each algorithm to generate 10^3, 10^6, 10^7 and 10^9 integers (around 4 kB, 4 Mb, 40 Mb and 4 Gb random data). A simple statistical testing was done with the test battery ENT which provides five different statistical tests to measure the randomness of integer/bit-sequences. With the test result that followed, an analysis over the differences between the algorithms could be done. The result was that of the three tested algorithms the AES-based PRNG was considered to be the best algorithm from a security perspective when it had the best statistical values. If you disregard safety requirements the linear feedback shift register was considered to be the most useful algorithm from a performance perspective, because its speed and relative statistical properties were very good.

Innehåll

Introduktion

Inledning

Detta arbete inleds med en kortare bakgrund till problemformuleringen, följt av det avsedda syftet med arbetet. Fortsättningsvis följer en kortare introduktion till några relevanta matematiska begrepp som används längre fram i texten. Innan själva beskrivningen av implementationerna och analyserna samt vilket tillvägagångssätt som använts vid testningen, finns ett fortsatt introducerande avsnitt om vad slumptal och pseudoslumptal anses vara, hur de används idag och hur man kan testa den statistiska slumpmässigheten i utdatat från en pseudoslumptalsgenerator.

Avslutningsvis sammanfattas analyserna och ett resultat presenteras. Givetvis efterföljs detta av referenser som använts i arbetet samt en bilaga med källkod över implementationerna.

Bakgrund

Idag finns det många, och framförallt, breda användningsområden för slumpgenerering, inte minst pseudoslumptalsgeneratorer. I takt med datorernas framfart har kravet på just slump blivit allt större. Forskare behöver indata till olika modelleringar och simuleringar, spelbranschen bygger mer eller mindre på att mekanismerna är av slumpartad natur och framförallt återfinns ett enormt krav på slumptalens inneboende icke-determinism inom kryptologin. Det verkar som om vissa typer av generatorer lämpar sig bättre än andra för vissa tillämpningsområden, och att en pseudoslumpalgoritm inte kan konstrueras hursomhelst. Tex. anser Knuth[6] att många av dagens psedoslumpgeneratorer är alldeles för ineffektiva och saknar tillfredsställande statistisk slumpmässighet. Därför är det viktigt att man som utvecklare är medveten om att det kan finnas risker att använda sådana algoritmer, och att man dessutom vet vilken typ av generator som lämpar sig mest väl i det tänkta projektet.

Syfte

Detta arbete syftar till att undersöka huruvida tre utvalda typer av pseudoslumptalsgeneratorer står i konstrast till varandra ur både ett prestandamässigt och statistiskt tillfredsställande perspektiv. En grundlig analys av både snabbhet och statistiskt slumpmässiga egenskaper hos generatorerna leder till en diskussion om hur

och var dessa gör sig bäst lämpade inom den datatekniska sfären idag. Uppsatsen kommer att belysa kraven och validiteten för pseudoslumptalsgeneratorer och i slutskedet, med analysen som underbyggande material, är det författarens förhoppning om att läsaren inser att pseudoslumptalsgeneratorer inte kan användas hursomhelst, närsomhelst.

Del I

Teori

Inledande matematiska begrepp

Här presenteras en kortare inledning över de begrepp och notationer som läsaren kan komma att behöva bekanta sig med i den fortsatta läsningen, om denne såtillvida inte känner sig tillräckligt säker i grundläggande modulär aritmetik, primtalsfaktorisering, entropi och enkel boolesk algebra.

2.1 Entropi

Entropi i det här sammanhanget är ett begrepp som härstammar från informationsteorin (läran om överföring av information). För en informationskälla som genererar några slags symboler, mäts osäkerheten för att en sådan symbol kommer att erhållas från en s.k slumpmässig variabel, eller stokastisk variabel som man också kallar det. En stokastisk variabel X, är en variabel vars värde är okänt fram tills en viss händelse utspelas. Denna variabel är något av utfallen som finns i en mängd av potentiella utfall, efter att den slumpmässiga händelsen utspelats, tex. utfallen 1,2,3,4,5 eller 6 efter ett tärningskast. På så sätt är utfallsrummet $X = \{1, 2, 3, 4, 5, 6\}$.

Informellt kan man säga att entropi är måttet på den förväntade överraskningen av ett utfall från en slumpmässig variabel. Ofta exemplifierar man detta med slantsingling; krona eller klave. Har man ett matematiskt idealiskt mynt (lika stor chans att få krona som klave, dvs $\frac{1}{2}$ i båda fallen), och gör en serie kast är entropin maximal[36], dvs 1 bit i det fallet. Skulle man å andra sidan ha ett fejkmynt som visar klave på båda sidor är entropin noll eftersom man helt enkelt kan förutse ordningen för varje kast. Detsamma gäller för en genererad bitsekvens med upprepade nollor där enda utfallet är just en nolla, entropin är noll eftersom samma bitvärde repeterats och man därför kan förutspå varje nästa bit. Med fortsatt resonemang angående mynten kan man tänka sig att man har ett fejkmynt som har sannolikheten p för krona och sannolikheten q för klave. Om p och q inte är $\frac{1}{2}$, är inte entropin maximal ($= 1$) för det fallet längre eftersom det för varje singling är en större sannolikhet för den ena myntsidan att komma upp, än den andra. Entropin är givetvis inte heller noll eftersom det ändå finns viss osäkerhet för varje kast.

Enheten för entropi är bitar, och i det nyss nämda exemplet med ett matematiskt idealiskt mynt där entropin var maximal är således resultatet 1 bit. Rent matematiskt beskrivs entropi av *Shannon's generella formel för osäkerhet*[17]:

$$H(X) = -\sum_{i=1}^{M} p(x_i) \log_b p(x_i)$$

där M är antalet olika symboler i den totala mängden av möjliga symboler som informationskällan genererar ifrån, b är den logaritmiska basen (i vårt fall $b = 2$ eftersom man räknar med databitar i sammanhanget) och $p(x_i)$ är sannolikheten för att en symbol x_i genereras.

För en fullständig härledning av Shannon's formel hänvisas läsaren till Thomas D. Schnieders arbete *Information Theory Primer With an Appendix on Logarithms*, som finns att läsa på internet (se referens [17]).

2.2 Något om primtal

Ett primtal p är ett heltal som endast är delbart med 1 och sig självt. Alla heltal n som inte är primtal (även kallade sammansatta tal) kan *primtalsfaktoriseras*, vilket innebär att n skrivs om som produkten av ett eller flera primtal. Dessutom är denna faktorisering entydig, dvs. det finns bara ett sätt att primtalsfaktorisera varje enskilt sammansatt tal. Det finns oändligt många primtal, vilket Euklides också bevisade i sin berömda sats[43], även kallad *Euklides sats*. Som exempel är de första primtalen $2, 3, 5, 7, 11, 13, 17 \ldots$

Man har viss användning av primtal i datalogiska sammanhang (främst när man talar om pseudoslumptalsgeneratorer och kryptografi), eftersom mycket av datalogiska teorier bygger på talteori, där primtal utgör en av många viktiga delar. Det visar sig att många algoritmer och beräkningar som görs i dessa sammanhang kan vara direkt beroende på primtal och dess egenskaper. RSA-kryptering exempelvis, vilket är ett assymetriskt kryptosystem, bygger på primtal och primtalsfaktorisering.

2.3 Relativt prima

Två heltal n_1 och n_2 sägs vara *relativt prima* om och endast om[44] deras s.k *största gemensamma delare* är 1 ($sgd(n_1, n_2) = 1$,). Detta betyder att n_1 och n_2 inte har någon gemensam primtalsfaktor, för om så vore fallet är inte den största gemensamma delaren 1. Tag tex. 6 och 9, vilka kan primtalfaktoriseras till $2 \cdot 3$ samt $3 \cdot 3$; här är 3 den största gemensamma delaren och således är 6 och 9 inte relativt prima. Däremot är 6 och 25 ($2 \cdot 3$ samt $5 \cdot 5$) relativt prima med samma resonemang.

Denna egenskap är användbar för vissa pseudoslumptalsalgoritmer, exempelvis vid kraven för maximal periodlängd hos en linjär kongruensgenerator (mer om detta i avsnitt 4.1).

2.4 Räkning med modulo

Utan att gå in på djupet och nyttja teorem och bevis, följer här en mycket enkel förklaring till hur man räknar med modulo, eller *modulär aritmetik* som det också

kallas ibland. Om man har två heltal, n_1 och n_2, säger man att de är *kongruenta modulo m* om båda ger samma rest vid en division med m. Ett annat sätt att se på det, lite mer informellt, är att man begränsar talmängden man skall göra beräkningar inom till att bara innehålla alla tal mellan 0 och $m-1$. Således om en beräkning genererar ett tal som är större än $m-1$ (eller mindre än 0) "transformeras" detta tal till ett som finns i den givna talmängden. Detta sker givetvis inte utan ett visst naturligt mönster, och i det här fallet ser man det som en cyklisk representation av talen 0 till $m-1$. Låt säga att talmängden är satt till $\mathbb{Z}_3$ (alla heltal från 0 till 2), då är den cykliska representationen 0,1,2,0,1,2,0,1... Om man då tex. gör en addition, $2+1=3$, där 3 inte finns i talrymden, fortsätter man räkna frammåt i cykeln, så $3=0$ i $\mathbb{Z}_3$. På samma sätt är $2+2=4=1$ i $\mathbb{Z}_3$.

Sammanfattningsvis så är den matematiska notationen för två heltal n_1 och n_2 som är kongruenta modulo m följande: $n_1 \equiv n_2 \pmod{m}$. Med exemplet ovan så är alltså $4 \equiv 1 \pmod{3}$.

2.5 Enklare boolesk algebra

Inom logiken har man uttryck för en förening mellan två påståenden som i sin tur ändras till ett nytt påstående, där betydelsen för det senare är något av de ursprungliga två påståendena. Dessa uttryck finns i några olika utföranden, några utav dem döpta till AND, OR och XOR. I datorsammanhang finns många viktiga och användbara tillämpningar av just sådana *operatorer*, inte minst för den fundamentala konstruktionen av datorer i allmänhet.

Operatorerna AND, OR och XOR betecknas $\wedge$, $\vee$ respektive $\oplus$. Nedanstående tabeller visar hur varje operator ändrar två bitvärden:

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

p	q	$p \oplus q$
1	1	0
1	0	1
0	1	1
0	0	0

Vad man gör med dessa operatorer rent praktiskt är att man ändrar en given databit eller sekvens av bitar, på olika sätt. Användningen av dem i detta arbete handlar mestadels om tillämpningen av dem i pseudoslumptalsalgoritmerna, främst kanske de linjära återkopplande skiftregistrerna och AES-krypteringen, där man har ett flertal beräkningar/operationer som jobbar med enskilda databitar och ändrar på dem (främst m.h.a XOR).

2.6 Kommutativ ring och Galoiskropp

En ring är en algebraisk struktur innehållandes en mängd S och två binära operationer addition (+) och multiplikation (*). Mängden S måste vara kommutativ under operatorn +, dvs. att för två element $a, b \in (S, +)$ gäller $a + b = b + a$. Kommutativiteten för addition ger att ringen innehåller additiva inverselement, men nödvändigtvis inte multiplikativa inverser. Vidare skall multiplikationen och additionen tillsammans vara distributiv, dvs. $a * (b + c) = (a * b) + (a * c)$ och $(b + c) * a = (b * a) + (c * a)$ för alla $a, b, c \in (S, *, +)$. Ringen av heltal $\mathbb{Z}$ är ett vanligt exempel på en ring.

En *kommutativ ring* är dock en ring med där ovanstående krav satisfieras samt att även multiplikationen över $(S, *)$ också är kommutativ, dvs. för alla $a, b \in (S, *)$ gäller att $a * b = b * a$. En *kropp* är en kommutativ ring där alla icke-noll element är multiplikativa inverser. En Galoiskropp GF är en ändlig kropp, dvs. en ändlig kommutativ ring där alla icke-noll element är multiplikativia inverser.

2.7 Chi-två test

Chi-två testning, eller χ^2-testning som det betecknas, är ett begrepp inom den matematiska statistiken. Vid statistisk testning använder man sig utav olika s.k *fördelningar*, dvs. uttryck för hur sannolika olika utfall av någon händelse ser ut. En av dessa fördelningar kallas χ^2-fördelning och är användbar vid vissa typer av statistiska undersökningar, bland annat i sammanhang där slump skall undersökas. Vad man gör vid ett χ^2-test är att man har ett antal observationer av den data man vill mäta och motsvarande förväntad frekvens av datat. Man tar sedan differensen av dessa och beräknar kvadraten av summan, och slutligen summerar man ihop alla kvadrerade differenser i den s.k testfunktionen:

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

där O_i är antalet observationer av data i och E_i är förväntat antal observationer av data i. Vad man nu kan se är huruvida den förväntade frekvensen (slumpmässigheten) beter sig.

Slump och pseudoslump

3.1 Slump

Läser man svenska Wikipedias beskrivning[35] av ordet slump, ser man att fenomenet beskrivs som en icke-kausal händelse, eller en verkan utan orsak som man också kan tolka det. Med det i åtanke kan man vidare diskutera huruvida en sådan icke-kausal händelse skall kunna verifieras - går det på något sätt att mäta eller avgöra om en verkan har en orsak eller ej? Är det dagens teknik eller människans hjärna som begränsar analysen av vad slump egentligen innebär? Dessa frågor i sig är intressanta, men hamnar ändå utanför ramen för denna uppsats även om det finns en stark koppling. Det är dock viktigt att på något sätt definiera bra slump och dålig slump, i det avseendet att man kan särkilja dem åt när man diskuterar olika datalogiska tillämpningars krav för slumpmässigt beteende.

Wikipedia anger två olika sätt på hur slump kan definieras: antingen anser man den vara *deterministisk*, fast i många fall med orsaker man idag inte kan mäta/-verifiera (inom den mikrokosmiska fysiken tex.), eller så anser man slumpen vara *indeterministisk* (eller icke-deterministisk). Detta innebär alltså att man underställer sig slumpen som ett rent, oförutsägbart fenomen vilket leder till att slumpen är en faktor inbyggd i naturlagarna så som vi uppfattar dem idag.

I det förra ställningstagandet kan man tänka sig att man ser på händelser, med idag oförklarliga orsaker, benämnas som slump som ett substitut i bristen på ett mera exakt begrepp. För börjar man fundera lite noggrannare på orsaker och verkan förstår man snabbt komplexiteten; orsakerna till livets uppkomst eller universums skapelse exempelvis. Här övergår diskusionen snabbt till spekulation och filosofiskt begrundande, således får slutsatsen i det här resonemanget bli att man tillsivdare inte kan konstatera om någon av definitionerna till slump är korrekta. I denna rapport kommer dock slump och slumpmässigt genererade tal att ses som icke-deterministiska, varför det också blir lättare att sedan referera till pseudoslump som deterministisk slump.

Ett annat sätt att definiera slump, enligt Chaitin[1], är att se det som ett strukturlöst fenomen, dvs. att man inte ser till varifrån någonting uppstod ifrån, utan om någonting tycks ha en logisk struktur. Vidare menar han att denna s.k logiska definition av slumpen definieras utifrån om den påstådda slumpmässiga händelsen kan beskrivas med hjälp av någon teori. Finns ingen sådan som beskriver händelsen, utan man enbart kan skriva ut den så som den är, anses den vara slumpmässig.

Ur ett mer matematiskt perspektiv skall ett tal som har blivit framslumpat, vara ett tal från en mängd X där sannolikheten att just detta tal väljes är lika stor som för vilket annat tal i X. Detta kallas att talen i mängden X har en likformig fördelning.

3.1.1 Slumptalens användningsområden

I datorsammanhang finns det en hel del användningsområden för slumpmässiga egenskaper, och allra främst slumpmässigt genererade tal. Vill man exempelvis modellera naturfenomen behövs någon sorts slumpdata för att kunna efterlikna vår värld på ett realistiskt sätt, så som kärn- och partikelfysik (slumpmässiga kollisioner) eller studiet av hur moleklyler rör på sig. På samma sätt behövs slumpmässigt genererade tal inom datorprogrammeringen för att kunna testa algoritmer, generera slump i vissa spelmekanismer eller tillhandahålla slumpmässig data vid skapandet av krypteringsnycklar och chiffertexter.

För bara ett århundrade sedan[14] använde forskare diverse enkla metoder så som slantsingling, kast med tärning och andra liknande handlingar när de behövde slumptal till sina arbeten. När datorerna började byggas på 1940-talet fick även utvecklingen av de matematiska metoderna för slumptalsgenerering, *slumptalsalgoritmer*, ett uppsving. Den första användningen[14] av sådana algoritmer på en dator, gjordes när beräkningen av neutroners förflyttningar under termonukleära explosioner skulle göras, under vätebombens utveckling.

När man insåg att datorer var kraftfulla verktyg för att implementera någon sorts slumptalsgenerering tog man det vidare till många andra användningsområden. Med tiden började man också involvera spelindustrin; de flesta spel som har någon typ av chans- eller sannolikhetsfunktion måste ha någon typ av slumptalsgenerator, allt från blandningen av spelkort åt ett nätpokerspel till slumpmekanismen hos enarmade banditer.

3.1.2 True Random Number Generator

Ett (kanske det enda) sätt att producera icke-deterministisk slump i datorsammanhang är med hjälp av såkallade *true random number generators* (TRNG). En sådan process använder sig av en icke-deterministisk källa[5], ett naturligt fenomen, och använder mätningar därifrån som indata till genereringsprocessen. Dessa källor kan exempelvis vara av termisk natur, kosmisk strålning eller radioaktivt sönderfall.

Oftast behövs speciell hårdvara (sensorer) för mätning av den icke-deterministiska källan, varför många företag tillverkar USB/PCI-komponenter eller andra typer av "svarta lådor" innehållandes dessa sensorer som antingen är fristående eller kan kopplas till dator.

Det finns en rad olika tekniker för att utvinna slumpmässig data från källorna, och vanligast tycks vara mätning av elektriska komponenters störningsbrus. Har man inte tillgång till sådan hårdvara får man försöka hitta andra lösningar, vilket man nu också gjort - pseudoslump. Idén är alltså att försöka efterlikna en

icke-deterministisk källa på bästa sätt till godtyckliga datorapplikationer. En vanlig variant är att programmeraren låter datorns inbyggda klocka generera slumptalen. Ett annat sätt, som används vid skapandet av nyckelpar till assymetrisk kryptering, är att låta användaren under en viss tid utföra en mängd slumpmässiga operationer på datorn, så som att trycka på tangenter eller röra muspekaren, för att skapa entropi till krypteringsalgoritmens indata. Det sistnämnda exemplet är diskutabelt huruvida det räknas som icke-deterministisk slump eller ej. Om en person står för det slumpmässiga beteendet kanske denne undermedvetet gör likartade rörelser/-tangentbordstryckningar varje gång och ett upprepande mönster uppstår. I detta arbete antas denna typ av slump ligga i gråzonen och anses mindre säkert jämförelsevis en hårdvara med sensorer för mätning av elektriskt brus eller liknande.

Att inte använda specifik hårdvara till TRNG medför risken att inte fullt ut kunna garantera icke-deterministisk slump, speciellt då graden av entropi kan variera starkt om källorna är för få eller deterministiska. Ian Goldberg och David Wagner[5] fann tex. att webbläsaren Netscape använde indata till sin slumpgenerering från bara tre olika källor: tiden, processens ID samt föräldrarprocessens ID. I teorin skulle man isådana fall kunna förutse utdatat om man kunde få tillgång till dessa källor på något sätt, vilket man kan tänka sig inte är en total omöjlighet.

Så problemet här är då återigen hur man skall definiera slump, skall exempelvis ett klockslag och några ID-nummer på processer i ett operativsystem räknas som slumpmässiga källor, även fast de kanske skulle kunna beräknas fram i teorin? Är det stor skillnad i entropi mellan hårdvarubaserade sätt kontra att låta en användare utföra en mängd slumpmässiga operationer på datorn och registrera dessa som slumpmässiga utfall? Man kan förmoda att det borde bero från fall till fall, hur viktigt det är med rent icke-determinsitisk slump. Kryptografi är ju ett område som naturligt behöver fullständiga TRNG-metoder (dock inte nödvändigtvis), medans ett ihopsnickrat kortspel i Java för hemmabruk, knappast har samma krav på slumptalsgenerering.

Johnson brus

Alla elektriska kretsar har en viss slumpmässig signalstörning[37], även kallat brus. Denna störning uppkommer bland annat från den termiska rörelsen hos laddningsbärare (elektroner) i elektriskt ledande material, vilket också uppstår oberoende av tillsatt spänningsstyrka eller typ av ledande material. Om så är fallet kallas det oftast för Johnson brus eller Johnson-Nyquist brus (uppkallat efter elektroingenjörerna som upptäckte och förklarade fenomenet). Eftersom det är den termiska rörelsen, dvs. temperaturen, som avgör hur mycket brus som uppstår kan man för vissa känsliga kretsar behöva kyla ner dem med flytande kväve.

Ett annat sätt som elektriskt brus uppstår på är genom *"shot noise"*[13], vilket är slumpmässiga variationer på elektronernas kvantifieringar, som strömmar genom något ledande material.

Då bruset uppstår som ett slumpmässigt fenomen är det väldigt användbart som källa till en TRNG. Vanligast[50] är att man förstärker bruset som uppkommer

från resistorer eller dioder och matar in det i en komparator eller s.k *Schimttrigger*. Dessa komponenter är olika typer av jämförare som ger en viss utdata beroende på en jämförelse av indatat. Dessa utdata kan sedan omvandlas till databitar eller heltal.

Tillgänglig hårdvara

Protego Information AB[46] är ett svenskt företag med säte i Lund som tillverkar hårdvarubaserade slumpgeneratorer. En av produkterna som saluförs, SG100, är en liten enhet som ansluts med en 9-pinnars serieport och erbjuder TRNG. Enligt tillverkarens hemsida använder man ett par olika källor till sina produkter: Zeenerdioder och termoddynamiskt brus (Johnson brus). Den förstnämnda källan är en typ av diod[38] som kan låta ström flöda i båda riktningarna, vilket en vanlig diod inte kan göra.

Figur 3.1. SG100, hårdvarubaserad TRNG, Protego Information AB, pris. 249 euro.

Det finns en mängd andra företag som också sysslar med hårdvarubaserad slumpgenerering, och oftast är det i form av USB-stickor eller vanliga PCI-kort. För att nämna några: IDQ[29] (från Schweiz), använder sig av kvantfysikaliska processer som källa, Simtec Electronics[51] registrerar kvantum-tunnling av elektroner som källa och LETech[30] (från Japan) som gör PCI-kort med termiskt brus som källa.

3.2 Pseudoslump

Då de flesta (tex. Teukolsky, Vetterling och Flannery[22]) i rådande stund inte anser sig inte kunna skapa fullkomligt slumpmässigt genererade tal med hjälp av vare sig algoritmer eller matematiska modeller, talar man därför om pseudoslump eller psudoslumptal (med då innebörden att talen är “pseudoslumpmässigt” genererade). Dessa är alltså till synes slumpmässiga tal, genererade genom nyss nämnda metoder, men är tillika i grund och botten deterministiska. Någonstans har man en variabel eller ett uttryck som startat en kedja av operationer som slutligen producerat ett tal, vilken alltid blir detsamma om startvilkoret också alltid är detsamma. Detta implicerar alltså att en ren slump inte kan vara den bakomliggande orsaken. En

informell definition av pseudoslump skulle kunna vara att det pseudoslumpmässiga utdatat skall var mycket "svårt" att skilja från äkta, icke-deterministisk slump. John von Neumann lär ha sagt *"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."*[24]

Ett resonemang[18] som också kan rättfärdiga varför en algoritm, impementerad på en dator, inte skulle kunna generera ren slump är att alla datorer är fullkomligt deterministiska och stoppar man in ett värde i en dator spottar den ut ett annat värde producerat av mekaniska beräkningar. Stoppar man in samma värde tillsammans med samma algoritm i annan dator blir utdatat detsamma som i den första. En dator har dessutom endast ett ändligt antal tillstånd den kan anta, och följden av detta är att datorn är periodisk i det avseendet att den efter ett tag (väldigt lång tid dock,) om man går igenom alla unika tillstånd, så kommer man slutligen stöta på ett och samma tillstånd igen. Detta innebär i sin tur att en förmodad slumptalsalgoritm också bara kan anta ett ändligt antal tillstånd och är således periodisk. Något som är periodiskt är också förutsägbart, och något som är förutsägbart är absolut inte slumpmässigt! Så genom ett motbevis har man nu insett att en förmodad slumptalsgenerator i en dator ej kan existera (om man inte talar om tidigare nämnda TRNG och någon implementation av sådan förstås).

En *pseudoslumpsgenerator* (eng. Pseudo Random Number Generator - PRNG) matas med ett s.k *frö* (eng. seed) som indata och så som beskrivet ovan så är det detta värde som står som grund för algoritmens utdata. Fröet skall vara ett slumpmässigt värde och ändras varje gång algoritmen körs, eftersom samma värde ger samma utdata oberoende av hur många gånger man kör den. På så sätt kan man se det som att algoritmen är en förlängning av fröet, ett slumptal som genererar ett nytt eller flera andra slumptal. Det finns en mängd olika sätt att ta fram dessa frön, exemplevis genom avläsning av användarens mus- och tangentbordsanvändning eller aktuell tid mätt i millisekunder och liknande.

När man talar om pseudoslumptalsgenerering, får man skilja på olika benämningar av pseudoslump. Ibland menar pratar man flyt- och heltalsgenerering och ibland om pseudoslumpbitsgenerering. I grund och botten är alla typer av PRNG faktiskt pseudoslumpbitsgeneratorer eftersom man räknar med databitar. Ett tal i en dator har alltid en binär representation, varför också ett pseudoslumpat tal egentligen är en bitsekvens av ettor och nollor. På så sätt matas en pseudoslumpsalgoritm med en viss bitsekvens, fröet, och producerar en ny, oftast längre bitsekvens. En formell beskrivning av en PRNG beskrivs på följande sätt av Stinson[20]:

*Låt k,l vara positiva heltal så att $l \geq k + 1$. En k, l-bits generator är då en funktion $f : (\mathbb{Z}_2)^k \to (\mathbb{Z}_2)^l$ som kan beräknas i polynomisk tid (som en funktion av k). Indatat $s_0 \in (\mathbb{Z}_2)^k$ är **fröet** och utdatan $f(s_0) \in (\mathbb{Z}_2)^l$ är den **genererade bitsekvensen**.*

En teoretisk iaktagelse är att den nyss beskrivna generatorn endast kan ha en ändligt stor utfallsrymd, dvs. systemet som generatorn implementeras i begränsar antalet unika utdata. Eftersom generatorn är beroende av systemets minne (storleken på in- och utdata) och att antalet möjliga utdatasekvenser är ändliga, kommer

utdatat att börja upprepa sig efter många beräkningar. En annan aspekt av upprepande utdata är en algoritms s.k *periodlängd*, vilket är den cykliska permutationen av algoritmens alla genererade pseudoslumptal. Ju längre periodlängd, dessto bättre generator eftersom upprepningen blir allt mer sällan. I fallet med de linjära kongruensgeneratorerna finns vissa matematiska villkor som måste satisfieras för att periodlängden skall bli så lång som möjligt, mer om detta i avsnitt 4.1.

Knuth beskriver[7] en intressant observation angående konstruktionen av PRNG, nämligen den att det är svårt att konstruera en sådan generator på ett helt slumpmässigt sätt. I ett försök där Knuth skapar "Algoritm K", utan något egentligt mönster, visar det sig att algoritmen snabbt hamnar i relativt korta cykler och antar ett repetetivt beteende. Poängen är alltså att konstruktionen av (bra) pseudoslumptalsgeneratorer i allmänhet kräver en hel del arbete och förståelsen för dess begränsningar är vital.

Pseudoslump kan tänkas behöva krav på sig, framförallt säkerhetstekniska sådana med tanke på att ett av de större användningsområdena är just inom säkerhet och kryptering. Även om en potentiell angripare eller annan ondsint person har tillgång till algoritmens källkod och t.o.m viss vetskap om den insamlade entropin som används för att mata frön, skall generatorn kunna vara säker. Tre grundläggande krav för pseudoslumptalsgenerering, finns att läsa om i Pinkas & Guttermans analys[4] av Linux inbyggda PRNG:

1. *Pseudorandomness*, generatorns utdata skall se slumpmässig ut för en utomstående observatör.

2. *Forward security*, även om en angripare får tillgång till generatorns tillstånd vid en viss tidpunkt skall det inte vara möjligt att kunna beräkna tidigare genererad utdata.

3. *Backward security*, även om en angripare får tillgång till generatorns tillstånd vid en viss tidpunkt skall denne inte kunna beräkna framtida genereringar av utdata, förutsatt att tillräcklig entropi finns för att mata frön och uppdatera generatorns tillstånd.

Om ovanstående krav är uppfyllda anses generatorn säker (vilket f.ö. inte Linux inbyggda generator tycks vara enl. författarna).

Fördelarna med PRNG i allmänhet är deras effektivitet och egenskapen att snabbt kunna generera många "slumpmässiga" tal. Samtidigt kan det också vara bra att algoritmerna är deterministiska, då man tillåts kunna generera samma sekvenser vid olika tidpunkter vid behov. Periodiciteten är dock en nackdel, men generatorer idag[48] har så långa perioder att det ändå inte spelar så stor roll.

Random.org, en webbsida som för övrigt tillhandahåller en TRNG baserad på atmosfäriskt brus, har sammanställt en tabell över olika användningsområden för både TRNG's och PRNG's:

Användningsområde	Mest lämpliga generatortyp
Lotteriverksamhet	TRNG
Spel och Hasardspel	TRNG
Slumpmässigt urval	TRNG
Simulation och modellering	PRNG
Säkerhet	TRNG
Konst	Varierar

Det skall förstås understrykas att tabellen ovan anger mest lämpade generatorn och nödvändigtvis inte mest använda. Att tex. spel och hasardspel är angiven med TRNG betyder inte att det alltid är implementerat så.

Har man verkligen någon nytta av PRNG om det ändå finns tillgång till TRNG? Svaret är att man absolut har stor nytta av pseudoslump, till och med en efterfrågan av effektiva och stabila generatorer finns. TRNG har inte några periodiciteter och man kan därför inte på något sätt förberäkna kommande tal om man läser av föregående. Men för att erhålla sådana egenskaper i en sekvens uppstår kompromissen som gäller med de flesta TRNG - tid och pris. Det kostar väsentligen mycket mer att köpa specifik hårdvara och installera detta i ett system än att bara ha programkod som på några få millisekunder kan generera miljontals heltal eller bitsekvenser. Man måste avgöra hur mcyket man kan tillåta sig kompromissa; är den aktuella implementationen i ett starkt behov på ett fullständigt slumpmässigt beteende? Ja, då kommer man med största sannolikhet att få leva med en inte alltför snabb lösning. Den förmodligen största marknaden där man kan tänka sig att TRNG ändå i många fall kan vara avgörande bör vara säkerhetsindustrin.

3.3 Statistisk testning av slump

Kan man verkligen avgöra huruvida en bitsekvens eller följd av tal är slumpmässigt genererade? Läser man de kapitel om slump- och pseduoslumptalsgenerering i en av Knuths böcker[10] får man ganska snart uppfattningen om att testning av slumpmässigt beteende inte verkar vara trivialt. Människor tenderar till att försöka avläsa mönster och upprepningar för att bilda sig en uppfattning om slumpmässigt beteende. Om sådant förekommer i en sekvens av tal eller databitar (eller vilken mängd av information som helst) vill man gärna avvisa det som slumpmässigt. Knuth skriver vidare, att ungefär vart tionde siffra i en slumpmässig följd kommer att vara likadan som den föregående siffran.

Även webbsidan *random.org*[49] medger att statistisk testning inte alls behöver ge korrekta slutsatser om en godtycklig mängds slumpmässighet. Däremot, liksom Knuth också förespråkar, bör man ha ett flertal olika statistiska tester som genomförs. Vissa av dessa kan komma att ge resultat som inte tyder på någon slumpmässighet, men ju fler tester som påvisar någon sorts likformig fördelning - desto säkrare kan man vara på att den testade sekvensen ändå är slumpmässigt genererad.

Problematiken vid statistisk testning av slump och pseudoslump, är att man förväntar sig en likformig fördelning. Man kan felaktigt ta för givet att tal i en sekvens alltid måste vara olika eller ha en viss spridning mellan varje förekomst av ett och samma tal. Låt säga att en TRNG skulle kunna generera en sekvens där låt säga åtta förekomster av talet 23 kommer i följd: 432432323232323234235667... Detta skulle ju onekligen se väldigt misstänkt ut! Men faktum är att en icke-deterministisk slumpgenerator mycket väl skulle kunna generera en sådan följd, annars vore det ju inte fullständigt slumpmässigt om man på något sätt skulle kunna förberäkna just den delsekvensen. Förmodligen är det nog inte så väldigt stor sannolikhet att en sådan lång och upprepande sekvens genereras (åtminstone inte för en kort sekvens), men låter man en generator producera väldigt mång tal under väldigt lång tid så måste ju möjligheten finnas; det skall ju vara lika stor chans för varje siffra att förekomma - varje gång.

Sammanfattningsvis måste man underkasta sig det faktum att oavsett vilken eller vilka tester man kommer att göra på en given sekvens, kommer man aldrig att kunna få ett helt korrekt svar om huruvida sekvensen var icke-deterministiskt genererad eller ej. Säkerligen kan man få bra uppskattningar och kanske värden som kan indikera på att det är slumpmässigt genererada sekvenser, men paradoxen ligger ju i att man inte *skall* kunna få ett resultat som säger att en viss sekvens är slumpmässig eller ej, för då vore den inte slumpmässigt genererad. Man kan aldrig avgöra om det var slump eller ej helt enkelt.

3.3.1 Testning med ENT

ENT, eller *Entropy calculation and analysis of putative random sequences* är en samling statistiska test implementerade av John "Random" Walker i mitten på åttiotalet och har under åren uppdaterats med nya och förbättrade testmetoder (senast 2008). Testningsmjukvaran är skriven i C och finns att ladda ner på skaparens hemsida[28]. ENT läser av innehållet i en fil och utför 5 olika tester: Entropi-, χ^2-, aritmetiskt medelvärdes-, Monte-Carlo värde för Pi- och korrelationskoeffecienttestning. ENT läser byte för byte alternativt bit för bit när den utför sina tester.

Entropitestet mäts i bitar per symbol, dvs bitar per siffra i heltalssekvensen i detta fall. Ju högre detta värde är, dessto mer informationstäthet är det i den testade sekvensen (tyder på högre slumpmässighet), medans sekvenser med väldigt få olika symboler/informationsenheter ger ett lägre entropivärde. Mer om vad entropi är och hur det fungerar kan läsas i avsnitt 2.1.

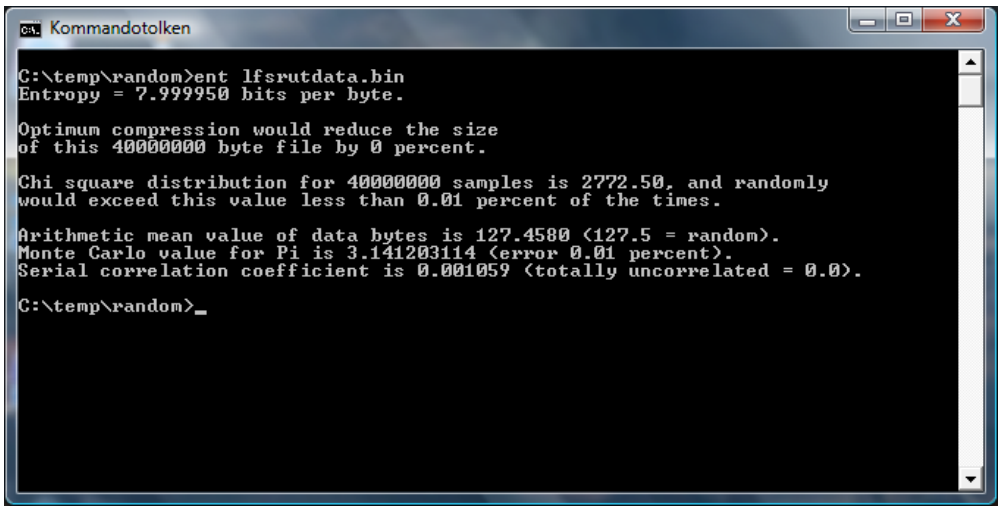
Enligt ENT's hemsida[28] är χ^2*-testet* ett av de vanligare statistiska testerna man gör vid mätning av slumpdata. Utdatan för det här testet är ett absoluttal och en procentsats som anger hur stor sannolikhet det är att en slumpmässig sekvens genererar ett värde högre än det framräknade absoluttalet. Enligt skaparen av ENT skall utdatat avläsas på så sätt att om procentsatsen är mer än 99% eller mindre än 1% är det med största sannolikhet inte slumpmässigt genererad data. Som referens anger Walker vidare på sin hemsida följande utdata för en förbättrad Park-Miller generator och en TRNG, baserad på radioaktivt sönderfall:

- (Park-Miller) Chi square distribution for 500000 samples is 212.53, and randomly would exceed this value 97.53 percent of the times.
- (TRNG) Chi square distribution for 500000 samples is 249.51, and randomly would exceed this value 40.98 percent of the times.

Dåliga generatorer eller annan icke slumpmässigt genererad data kan ligga runt 99.99%. Mer om χ^2-testning finns i avsnitt 2.7.

Det *aritmetiska medelvärdet* är summan av bytes, med tolkningas som 8-bitars heltal, (eller enskilda bitar tolkade som talen 0 eller 1 om man flaggar programmet med -b) delat på filens längd. Om datat är nära slumpmässigt genererat skall siffran ligga runt 127.5 (och 0.5 om man använder -b).

Monte-Carlo testet för Pi är ett test där man beräknar talet pi med hjälp av slumpmässiga punkter i en geometrisk figur i planet. Man har en cirkel med radien R inrutad i en kvadrat med sidlängd 2R. Proportionen av arean i cirkeln till arean i kvadraten är $\frac{\pi}{4}$. Om man nu väljer n slumpmässiga punkter innanför kvadraten kan man approximera att det hamnar $n\frac{\pi}{4}$ punkter i cirkeln. Låt sedan m vara antalet punkter som inte hamnar i cirkeln, då räknas sedan värdet på pi ut genom $\pi = \frac{4m}{n}$. För den nyss nämnda TRNG som användes som referens, erhölls ett värde av 0.06% felmarginal vid en sådan beräkning.



Figur 3.2. Typisk utskrift vid användning av ENT

Det sista testet tittar på huruvida varje byte i datat beror på den föregående byten. Värdet skall ligga så nära 0 som möjligt och icke slumpmässiga sekvenser ligger runt 0.5.

Det finns ett annat testbatteri som är större än ENT i antal tester, nämligen NIST's[11] *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Den givna tidsramen för detta arbete gjorde att jag valde bort detta test eftersom det är väldigt omfattande och kräver mycket inläsning av hur respektive test fungerar och hur de skall tolkas.

Del II

Algoritmerna

Valda algoritmer och implementation

Det här arbetet kommer att behandla tre olika typer av PRNG, prestandatesta dem och utsätta varje generator för statistisk testning med testbatteriet ENT. Varje generator producerar ett eller flera 32-bitars heltal (inte hela sanningen; den linjära kongruensgeneratorn generar 31-bitars heltal). Samtliga matas med 32-bitars frön, utom den AES-baserade PRNG'n som använder 128-bitars frön. Detta innebär att den AES-baserade PRNG'n kommer att vara avsevärt långsammare än de andra algoritmerna och att utdatat är 128 bitar vid varje körning. Detta är ett medvetet val eftersom jag ej ville göra om AES-algoritmen till att använda 32 bitar då den verkligen skulle baseras på AES, som använder 128-bitars block som lägst[12]. Det blir givetvis en orättvis jämförelse i prestandamätningen, men det är lika intressant att se hur mycket långsammare en 128-bitars algoritm är jämförelsevis en 32-bitars och om den kompromissen ger sämre eller bättre statistiska egenskaper.

Den statistiska testningen tittar på sekvenser *bytevis* respektive *bitvis*, varför det inte spelar någon roll hur många bitar algoritmen är baserad på.

De tre typerna av PRNG som presenteras här är *Linjära kongruensgeneratorer*, *Linjära återkopplande skiftregister* och en *AES-baserad PRNG (counter mode)*.

Två av algoritmerna har implementerats på egen hand: LFSR och den AES-baserade PRNGn. Båda gjordes i C++ i Microsoft Visual Studio 2010. Den tredje algoritmen är tagen från Internet och fanns redan implementerad av en annan person (mer detaljer om det i 4.1.2). Nedan följer en kortare beskrivning av respektive algoritmtyp och hur den har använts i detta arbete.

4.1 Linjära kongruensgeneratorer

Den första typen av PRNG som presenteras här är de s.k linjära kongruensgeneratorerna (LKG). Namnet kommer från de matematiska modeller man baserar beräkningarna på, dvs. linjära differensekvationer modulo m. Detta betyder att varje element i bitsekvensen är en unik linjär funktion modulo m av det föregående elementet i sekvensen. Enligt Wikipedia[39] sägs linjära kongruensgeneratorer vara en av de äldsta och mest kända pseudoslumptalsalgoritmerna. LKG är relativt enkla att förstå, då algoritmen inte består av speciellt många olika (egentligen bara en) beräkningar i allmänhet. Formellt definieras LKG på följande sätt[21]:

$$X_{n+1} \equiv (aX_n + c)\ (mod\ m)$$

där m är talet som algoritmen skall räkna modulo med, c ett inkrementationstal, a multipliceraren och X_0 fröet. Samtliga parametrar är heltal.
En viktig inneboende egenskap i de linjära kongruensgeneratorerna är de s.k *periodlängderna*. Detta är måttet på när sekvensen av tal återupprepas i utdatasekvensen. Eftersom generatorn beräknar modulo m, kan periodlängden som längst vara m tal lång på grund av modulo operatorns natur. Om periodlängden skall vara just m, måste följande vilkor satisfieras:[8]:

1. c och m är relativt prima.
2. $a - 1$ är delbart med alla primtalsfaktorer av m.
3. $a - 1$ är en multipel av 4 om m är en multipel av 4.

Man inser att algoritmens enkelhet tillsammans med periodegenskapen medför att vissa uppenbara begränsningar uppstår. LKG-algoritmer är alltså väldigt beroende av parametervalen för m, a och c. Exempelvis om m väljs till ett litet tal begränsas antalet element som kan permuteras i cykeln; periodlängden minskar. Således finns argument för att man bör välja stora m, och kanske sådant så att m är av typen 2^n då datorer jobbar och lagrar tal med sådana potenser.

Engelska Wikipedia[39] skriver att de mest effektiva LKG-algoritmerna använder $m = 2^{32}$ alternativt $m = 2^{64}$ av den anledningen att modulo operationen trunkerar alla bitar förutom de 32 eller 64 minst signifikanta bitarna.

Exempelvis kan en mycket enkel implementation av en LKG i C++ se ut på följande vis:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7     int i = 0;
8     int m = 5;
9     int a = 2;
10    int c = 3;
11    int X = 0;
12
13    cout<<"Enkel linjär kongruensgenerator"<<endl;
14    while(i < 10) {
15      X = (a * X + c) % m;
16      cout<<X<<" : ";
17      i++;
18    }
19
20    return 0;
21 }
```

Utskriften från ovanstående applikation genererar siffersekvensen $3:4:1:0:3:4:1:0:3:4$, och man ser snabbt att periodlängden är 4 för just detta parameterval. Beräkningen är alltså, med det givna fröet $X = 0$:

1. $X_{n+1} \equiv (2 * 0 + 3) \pmod 5 \equiv 3 \pmod 5$
2. $X_{n+1} \equiv (2 * 3 + 3) \pmod 5 = 9 \equiv 4 \pmod 5$
3. $X_{n+1} \equiv (2 * 4 + 3) \pmod 5 = 11 \equiv 1 \pmod 5$
4. $X_{n+1} \equiv (2 * 1 + 3) \pmod 5 \equiv 0 \pmod 5$
5. osv...

Eftersom algoritmen i sig inte är alltför komplex går det relativt snabbt att utföra beräkningarna, vilket gör algoritmen till en mycket snabb och minneseffektiv sådan. Nackdelen med detta är att kvalitén på pseudoslumptalen inte blir allt för hög, exempelvis om periodlängden är kort får man många upprepningar av tal. Man kan tänka sig att en praktisk implementation av LKG-algoritmer skulle kunna vara i de fall där man har mer krav på snabbhet än entropi, så som mindre inbyggda system med små internminnen eller i spel där man vill slumpa fram många tal snabbt.

Enligt Teukolsky mfl.[23] är många av programmeringsspråkens och olika datasystems egna slumpfunktioner (rand() tex.) relativt enkla linjära kongruensgeneratorer, vilket gör att man inte kan förlita sig på dem i alltför stor utsträckning. Sysslar man med krypteringsalgoritmer eller annan mjukvara som har höga krav på slumpmässiga egenskaper bör man använda andra externa pseudoslumptalsgeneratorer och algoritmer.

4.1.1 Lehmer-pseudoslumptalsgenerator

En variant av de linjära kongruensgeneratorerna är de s.k *multiplikativa kongruensgeneratorerna*, vilka har den egenskapen att $c = 0$. En *Lehmer-pseudoslumptalsgenerator* är en multiplikativ kongruensgenerator med ytterligare specifika restriktioner vid val av parametrar. Till att börja med måste fröet X_0 alltid vara relativt primt med modulo m, därtill skall a vara en primitiv rot till modulo m och således blir den maximala periodlängden[9] som kan existera $m - 1$. Parametern m skall vara ett primtal upphöjt till ett annat primtal, och Park & Miller[15] föreslog 1988 att m skulle väljas specifikt till $m = 2^{31} - 1$ och $a = 16807$. Vidare anger Park & Miller att det finns omkring 534 miljoner andra värden på a som tillsammans med det givna m'et ger maximal periodlängd. Den allmänna fomeln för en Lehmer-generator är:

$$X_{n+1} \equiv (aX_n) \pmod m$$

Enligt engelska Wikipedia[40] återfinns olika typer av Lehmer-generatorer i bland annat den gamla hemdatorn ZX Spectrum från 1982 ($m = 2^{16} + 1$), den amerikanska superdatorn Cray ($m = 2^{48}$) och i programmeringsspråket Forth ($m = 2^{32} - 5$).

4.1.2 Implementation

Den implementation av de linjära kongruensgeneratorerna som valts till denna uppsats är den s.k *Park-Miller pseudoslumptalsgeneratorn*, vilket är en Lehmer-generator med parametrarna m och a satta efter Park & Millers rekommendationer givna i föregående avsnitt. Implementationen är gjord av Robin Whittle[34] i C++, och är tillgänglig för publikt användande. Den publicerade generatorn finns tillgänglig både för flyttal och heltal, och endast heltalsvarianten används i det här arbetet. Källkoden är väldigt väldokumenterad, vilket syns i bilaga 1. Nedan finns två urklipp av de viktigaste delarna av koden, headerfilen och klassdefinitionen av själva generatorn (med viss bortplockad kommentering av utrymmesskäl):

```
1  /*
2   * Header for rand31-park-miller-int.c
3   */
4
5    unsigned int rand31pmc_next(void);
6    void               rand31pmc_seedi(long unsigned int);
7    unsigned int rand31pmc_ranlui(void);
8
9     long unsigned int seed31pmc  = 1;
```

I header-filen är även en kodrad borttagen, vilket är ett funktionshuvud som endast används i flyttalsvarianten.

```
1  // Robin Whittle  rw@firstpr.com.au     2005 September 21
2  // ...
3  // The following code is public domain ...
4
5  #include <iostream>
6  #include <string>
7  #include <cmath>
8  #include <time.h>
9  using namespace std;
10
11 class rand31pm {
12     unsigned int seed31;
13
14 public:
15     rand31pm() {seed31 = 1;}
16
17     void               seedi(unsigned int);
18     unsigned int ranlui(void);
19
20
21 private:
22
23                                     // Multiplier constant = 16807 =
                                           7^5.
```

```
24                                      // Park and Miller in 1993
                                            recommend
25                                      // 48271.
26      #define consta 16807
27                                      // Modulus constant = 2^31 - 1 =
28                                      // 0x7FFFFFFFF.  Use .0 to deter
                                            compiler
29                                      // from complaining about a very
                                            large
30                                      // integer constant.
31      #define constm 2147483647.0
32
33      unsigned int nextrand()
34      {
35          double const a = consta;
36          double const m = constm;
37
38                                      // This is the linear congrentual
39                                      // generator:
40          seed31 = (fmod((seed31 * a), m));
41
42          return (seed31);
43      }
44  };
45
46
47  void rand31pm::seedi(unsigned int seedin)
48  {
49      if (seedin == 0) seedin = 1;
50      seed31 = seedin;
51  }
52
53  unsigned int rand31pm::ranlui(void)
54  {
55      return nextrand();
56  }
```

Källkoden är som till synes inte särskilt omfattande, vilket återspeglar de linjära kongruensgeneratorernas relativa enkelhet (i avseende på implementation). Egentligen återfinns kärnan i koden på rad 40: *seed31 = (fmod((seed31 * a), m));* vilket är själva beräkningen av nästa tal - kongruensgeneratorn! Resten av källkoden är bara en klassdefinition och en samling funktioner som gör programmet mer överskådligt och lättare att hantera.

Viktigt att notera här är också att källkoden är modifierad så att ingenting av originalets funktioner eller variabler ändrats, utan endast en tidtagningsmekanism är tillagd och viss irrelevant kod (flyttalsfunktionerna) är borttagen. Den fullständigt modifierade koden återfinns i bilaga 1.

Med det satta värdet $m = 0x7FFFFFFF = 2^{31} - 1$ kommer generatorn att producera en cykel av 2147483646 stycken 31-bitars heltal. Delar av sekvensen ser ut på följande sätt:

Värde	Antal steg i cykeln med fröet satt till 1
16807	1
282475249	2
1622650073	3
984943658	4
1144108930	5
470211272	6
101027544	7
1457850878	8
1458777923	9
2007237709	10
925166085	9998
1484786315	9999
1043618065	10000
1227283347	1000000
1808217256	2000000
1140279430	3000000
168075678	99000000
1209575029	100000000
941596188	101000000
1207672015	2147483643
1475608308	2147483644
1407677000	2147483645
1	2147483646

Park-Miller-algoritmen återfinns dessutom i en optimerad version i det avseende att istället för att göra divisionsberäkningar nyttjar man ett “trick” med additioner och modulo, upptäckt av David Carta[45]. Problematiken låg i att fröet$*$16807 ger ett 46-bitars tal och eftersom registren i en 32-bitars dator är 32 bitar stora måste man dividera ner produkten. Då division oftast är en multi-cykel operation i en dator, medans addition och multiplikation är enskilda cykeloperationer, fann alltså Carter ett sätt att kringgå detta.

4.2 Linjära återkopplande skiftregister LFSR

Ett skiftregister är en digital krets med ett antal vippor (digitala byggstenar som kan anta två olika tillstånd och därmed kan fungera som en minnescell/register) som serie- och/eller parallellkopplats på ett sådant sätt att de också delar samma klocka.

Databitar kan därför fortplanta sig genom skiftregistret och den sista vippan i serien ger en utdatabit. Systemet är en slags tillståndsmaskin med ett internt tillstånd bestående av bitarna i registret samt in- och utdatabitarna. Vid varje klockcykel triggas registret och antar ett nytt tillstånd. Dessa skiftregister kan också se ut och fungera på lite olika sätt, exempelvis kan det vara uppbyggt av flera parallella register så att det bildar ett flerdimensionellt system, bitarna kan skiftas antingen åt höger eller vänster och utdatabiten kan vara indatabiten för samma register. Det sistnämnda är också fallet för de s.k linjära återkopplande skiftregistren (LFSR från eng. *Linear Feedback Shift Register*). Informellt kan man säga att ett LFSR är en typ av skiftregister vars indatabit är en linjär funktion av registrets föregående tillstånd.

För att systemet skall vara linjärt återkopplande måste någon sorts funktion för detta beskrivas. Det som egentligen händer är att vissa specifika vippor, dvs. platser för en bit i registret, används för att tillsammans med andra specifika vippor, utföra en XOR operation som i sin tur ger en utdatabit. Beskrivningen av vilka specifika bitar som används för detta ändamål kallas *peksekvensen* (eng. *tap sequence*). I denna uppsats kommer notationen för vilka de specifika bitarna är ges av $[x_1, x_2, \ldots, x_i]$, så om exempelvis första och sista biten i ett 4-bitars återkopplande skiftregister är de specifika bitarna betecknas detta som $[1, 4]$. Sammantaget består det linjära återkopplande registrets tillstånd av dess bitar, peksekvensen samt in- och utdatabiten.

En mer formell beskrivning ges av Cusick & Stanica[2]: Ett LFSR av längd n består av n numrerade steg $0, 1, \ldots, n-1$, där varje sådant kan anta en etta eller nolla. Dessa kontrolleras av en klocka. En vektor med index $s_0, s_1, \ldots, s_{n-1}$ initierar skiftregistret och vid tidpunkt (klockcykel) i utförs följande operationer:

1. s_i (innehållet i steg 0) ändras;
2. Innehållet i steg i skiftas till steg $i-1$, för $1 \leq i \leq n-1$;
3. Det nya innehållet (den återkopplande biten) av steg $n-1$ erhålls genom att utföra XOR med värdena på en delmängd (peksekvensen) av ett givet antal steg.

Precis så som PRNG i allmänhet har beskrivits så finns paralleller med dessa återkopplande skiftregister, faktum är att ett LFSR faktiskt kan vara en pseudoslumptalsgenerator. $s_0, \ldots, s_{n-1}$ är fröet och det interna tillståndet i registret bestämmer deterministiskt utdatabiten. Detta innebär också att det endast kan finnas ett ändligt antal inre tillstånd för ett n-bitars register, nämligen $2^n - 1$ stycken. Alla bitar satta till noll räknas bort eftersom ett sådant tillstånd bara kommer att fortsätta generera noll-tillståndet[32]. På samma sätt som med en linjär kongruensgenerator så börjar samma tillstånd att upprepas efter en tid och bildar en cykel (med en given periodlängd). På grund av skiftregistrets natur[31] kommer varje unikt icke-noll tillstånd att passeras endast en gång i en cykelgenomgång för ett register med maximal peksekvens. En maximal peksekvens är en sådan peksekvens

som endast innehåller två tillståndsrymder: nollrymden (bara nollor i registret) och den maximala tillståndsrymden $2^n - 1$. Att välja maximal peksekvens är inte trivialt. Skiftregistrets period beror på hur peksekvensen väljs. Dock bör det noteras att det kan finnas flera maximala peksekvenser för ett visst n. Även fröet är direkt avgörande för hur många tillstånd som kommer att genereras.

Linjära återkopplande skiftregister beskrivs matematiskt med hjälp av de s.k *Galoiskropparna*, eller kommutativa ringar (se avsnitt 2.6). Alla beräkningar inom ringen görs modulo 2 eftersom det är databitar operationerna räknar på. I modulo 2 räkning är addition och subtraktion detsamma som XOR-operationen eftersom $1 + 1 = 0$, $0 + 0 = 0$, $0 + 1 = 1$ och $1 + 0 = 1$ (mod 2), vilket ger samma tabell som för XOR given i det inledande avsnittet 2.5. Multiplikation är en bitvis AND-operation. Matematiskt beskrivs det återkopplande skiftregistret genom ett polynom i Galoiskroppen GF(2):

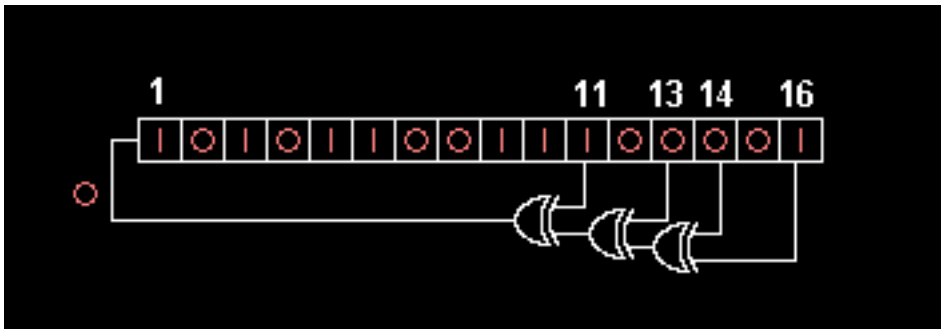
$$G(x) = g_m x^m + g_{m-1} x^{m-1} + \ldots + g_1 x^1 + g_0$$

Varje g_i är en koeffecient som antingen antar en etta eller nolla beroende på om x_i är en återkopplande bit eller ej ($g_i = 1$ om den återkopplar). Så för ett LFSR med peksekvensen $[1, 3] = [3, 1]$ så ser den matematiska beskrivningen ut på följande sätt: $G(x) = x^3 + x^1 + 1$, där konstanten $g_0 = 1$ alltid är ett då det representerar indatakopplingen till registret. För att nu kunna bestämma huruvida den maximala peksekvensen skall se ut använder man sig utav det polynomiska uttrycket. Detta görs genom att avgöra om det är ett primitivt polynom, dvs. om det ej kan faktoriseras. Om så är fallet, är den givna peksekvensen maximal.

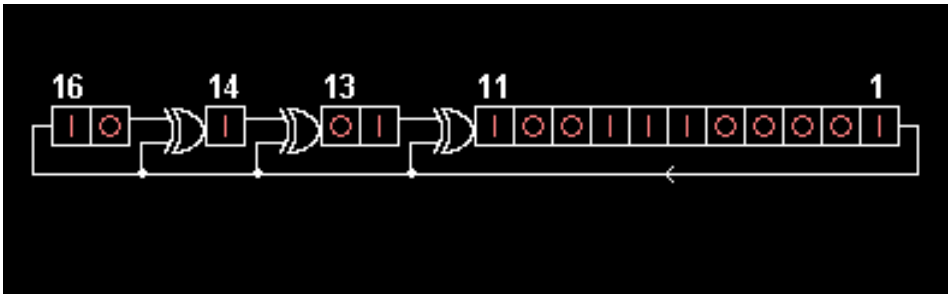
Givetvis är inte LFSR ett hårdvaruspecifikt fenomen, man kan också programmera algoritmer som beter sig som linjära skiftregister med hjälp av vanliga heltal och/eller arrayer. Den här uppsatsen använder sig utav en icke hårdvaruimplementerad LFSR, en simulation skriven i C++. Det är konceptet att linjärt utföra XOR-operationer enligt givna peksekvenser som är det essentiella i den här typen av bitsekvensgenerering.

4.2.1 Fibonacci och Galois

LFSR kan implementeras på två olika sätt[3], Fibonacci- eller Galoisimplementation. Den förra är den mest vanliga typen. Ett Fibonacci LFSR fungerar som det är beskrivet hittils i detta avsnitt, registret har en given peksekvens över n bitar och för varje klockcykel förskjuts registerbitarna från indatahållet mot utdatahållet. Utdatbiten är paritetsbiten given av de sekvensiella XOR-operationerna med peksekensernas bitar.



Figur 4.3. 16-bitars Fibonacci LFSR, källa: `http://en.wikipedia.org/wiki/LFSR`



Figur 4.4. 16-bitars Galois LFSR, källa: `http://en.wikipedia.org/wiki/LFSR`

Den andra typen, Galois LFSR, har samma egenskaper som ett Fibonacci LFSR, fast förskjutningen sker på ett lite annorlunda sätt. När förskjutningen sker kopieras de bitar som inte tillhör peksekvensen ett steg åt utdatahållet. Peksekvensbitarna XOR'as med utdatabiten från registret innan de kopieras åt utdatahållet. Figuren ovan illustrerar skillnaden mellan de två typerna.

4.2.2 Effektivitet

En effektivitetsmässig skillnad[3] på de två typerna av LFSR är att i en hårvaruimplementation är Galois LFSR något snabbare då den i allmänhet har en lägre grindfördröjning. I allmänhet är LFSR, liksom LKG, inte alltför komplexa att implementera. Åtminstone inte i fråga ur ett konceptuellt perspektiv. Egentligen bygger algoritmen på (höger)skift operationen och XOR, men skall det hårdvaruimplementeras måste ju utvecklaren förstås kunna den aktuella arkitekturen och dess assemblerspråk, vilket i sig inte nödvändigtvis är enkelt att hantera.

Liksom LKG så anses LFSR inte fungera optimalt (dvs. statistiskt tillfredsställande) som pseudoslumptals/bitsgenerator. Enligt Schlundt[16] visar det sig tydligen att de statistiska egenskaperna inte är särskilt bra. Han rekommenderar dock att om man vill generera ett n-bitars tal så bör man använda ett $(n+l)$-bitars LFSR

och bara plocka de n första bitarna från utdatat. Fortsättningsvis menar han också att man kan få bättre statistiska egenskaper genom att skifta flera gånger innan man utplockar ett pseudoslumptal från det återkopplande skiftregistret. Med detta sagt så är ändå ett LFSR relativt snabbt (främst om det implementeras med assembler före C) och enligt Schneier[19] är dessutom de linjära återkopplande skiftregistren den mest förekommande typen av skiftregister inom modern kryptografi.

Eftersom man alltid vill ha ett LFSR med maximal peksekvens måste man också hitta ett lämpligt primitivt polynom mod 2. Att göra det för större n är inte alltid lätt. Det enklaste sättet[19] enligt Schneier är att välja ett slumpmässigt sådant polynom och sedan testa om det är primitivt. Detta är dock komplicerat och i analogi med testning av slumpmässigt valda primtal. Däremot finns ett flertal matematiska mjukvarupaket som kan tillhandahålla sådan testning.

4.2.3 Implementation

Den valda implementationen till detta arbete blev en enkel LFSR skriven i C++ som kodades utifrån beskrivningen från New Wave Instruments hemsida[32].

LFSR Pseudokod

```
lfsr ← [000...001]
check ← false
tapseq ← n1, n2, n3, n4
while check == false do
  output ← lfsr[n1] xor lfsr[n2] xor lfsr[n3] xor lfsr[n4]
  rotateLFSR(lfsr)
  lfsr[0] ← output
  if lfsr == [000...001] then
    check ← true
  end if
end while
```

Ovanstående pseudokod visar hur konceptet enkelt kan kodas genom att endast utföra en rad XOR-operationer och sedan rotera registret/vektorn och stoppa in XOR-produkten som indata till första platsen. I implementationen har fyra stycken bitar använts till peksekvensen n_1, n_2, n_3, n_4. Dessa är inte valda att generera en maximal periodlängd. Den valda peksekvensen ger en periodlängd av 153391689 tal ($>$ 150 miljoner tal).

4.3 Advanced Encryption Standard

AES i sig är ingen renodlad pseudoslumpgenerator, men man skulle kunna tänka sig konstruera en sådan med algoritmen som grund. Då AES består av förhållandevis

komplexa beräkningar och manipulationer med enskilda bytes torde den kunna fungera som fundament åt en generator åtminstonde. Inledningsvis beskrivs ordinarie AES och dess interna funktioner och beräkningar, varefter den egna implementationen av en AES-baserad PRNG presenteras i detalj.

4.3.1 Bakgrund

Den 26 November 2001 fastslogs en ny standardiserad krypteringsalgoritm av amerikanska NIST (National Institute of Standards and Technology), som ersatte den tidigare algoritmen DES (Data Encryption Standard). Den nya krypteringsalgoritmen var resultatet av en flerårig process där ett flertal algoritmkandidater utvärderats och testats, för att slutligen avgöra den mest säkra, effektivaste (i minnes- och processorsnabb aspekt) och flexiblaste (implementationsmässigt) krypteringsalgoritmen. Vinnaren blev således Dr. Joan Daemen och Dr. Vincent Rijmens bidrag kallad *Rijndael*, vilket sedermera antogs som den nya standarden döpt till *Advanced Encryption Standard*, AES.

4.3.2 Symmetriska blockchiffer

Ett symmetrisk blockchiffer är en krypteringsalgoritm som delar upp ett givet meddelande m i ett antal block, $m = m_1, m_2, \ldots, m_n$ och utför någon typ av kryptering på respektive meddelandeblock. Oftast har man ett visst beroende mellan blocken sinsemellan då utdatat man får från det första blocket kan användas som indata till den krypterande delen av algoritmen, som krypterar nästkommande block.

När kryptering sker använder man en s.k *nyckel*, även den är ett block av viss storlek (inte nödvändigtvis samma som meddelandeblocken), som också består av en "text". Tillsammans med ett textblock och en mer eller mindre komplex krypteringsalgoritm mixas sedan ursprungsmeddelandet upp i en till synes slumpmässig oordning. Ofta sker detta genom tex. teckensubstitutioner och permutationer.

En vanlig design av blockchiffer är de s.k *iterativa* chiffren. Dessa bygger på att man har ett *nyckelschema* som genererar ett flertal nya subnycklar av den ordinarie krypteringsnyckeln, och att chiffret utför sina beräkningar i ett antal *rundor*. Idén är att man tillverkar lika många subnycklar som antal rundor som skall köras, en unik nyckel till varje runda.

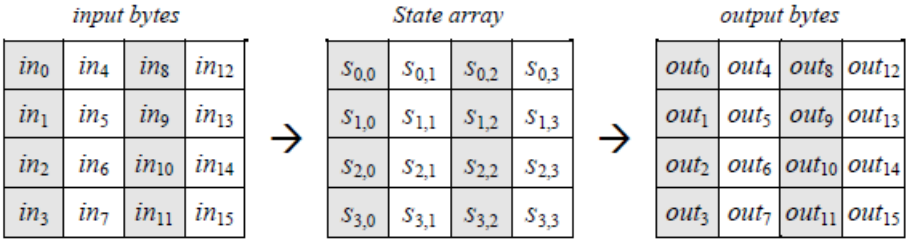
Blockchiffer kan använda olika operationstyper när meddelandeblocken bearbetas. De olika operationstyperna avgör huruvida meddelandeblocken skall krypteras självständigt från andra block, om de skall vara beroende av varandra och hur detta beroende skall se ut. Vanliga typer är Electronic Codebook (ECB), Cipher-block chaining (CBC), Cipher feedback (CFB) mfl.

4.3.3 Översikt av AES

AES är alltså ett iterativt blockchiffer och jobbar med 128-bitars block. Nyckelstorlekarna man kan välja mellan är 128, 192 och 256 bitar, vilket då refererar till AES-128, AES-192 samt AES-256. Algoritmen tar 128 bitar som indata och producerar

en lika stor utdata, och under rundorna som exekveras vid en kryptering/dekryptering görs beräkningar med enskilda bytes (ibland slussas de in i funktionanrop som words, dvs. 4 bytes åt gången). Antalet rundor varierar också beroende på nyckellängden, för 128-bitars nycklar körs 10 rundor, för 192-bitars nycklar - 12 rundor och för 256-bitars nycklar körs 14 rundor.

AES jobbar internt med en 4×4-matris innehållande bytes, vilken kallas för *tillståndet* S (eng. state). Varje rad i matrisen innehåller Nb bytes, där Nb är blocklängden dividerat med 32 ($\frac{128}{32} = 4$) bytes. Det finns implementationer av AES med större blockstorlekar, och då får S-matrisen fler kolumner eftersom Nb blir större. Initialt plockar algoritmen fram första meddelandeblocket på 128 bitar och kopierar över värdena till S-matrisen. På så sätt är det absolut första värdet på S samma som meddelandeblocket. Därefter utförs en rad olika operationer som ändrar värdena i S-matrisen, och slutligen kopieras värderna över till en utdatamatris. Man har således tagit 128 bitar meddelandetext och krypterat dessa till motsvarande 128 bitar chiffertext. Varje kolumn i S-matrisen motsvarar en array innehållande 4 bytes med data, en 32-bitars word med andra ord. I AES-dokumentationen[12] benämns varje word med $w_0 = (s_{0,0}, s_{1,0}, s_{2,0}, s_{3,0}), w_1 = (s_{0,1}, s_{1,1}, s_{2,1}, s_{3,1}), w_2 = (s_{0,2}, s_{1,2}, s_{2,2}, s_{3,2})$ samt $w_3 = (s_{0,3}, s_{1,3}, s_{2,3}, s_{3,3})$.



Figur 4.5. Inmatning av meddelandetext till S-matrisen som i slutskedet överför den krypterade meddelandetexten till en utdatamatris. Källa: *Federal Information Processing Standards Publication 197, NIST*

När AES läst in ett meddelandeblock till S-matrisen samt en krypteringsnyckel är algoritmen redo för kryptering.

En egen implementation av AES (krypteringsdelen) gjordes i C++, och nedan finns "skelettkoden" som gjordes i det första skedet. Samtliga funktionsanrop var alltså innehållslösa till en början och fylldes på efter hand. Men den här kodsnutten visar en övergripande bild av hur AES går tillväga, steg för steg. Den första for-loopen fyller S-matrisen *state*[][] med ett textblock á 16 bytes, den mellersta loopen utför alla rundor om den sista (vilket sker direkt efter) och den sista for-loopen fyller en 16 bytes stor utdata-matris med den krypterade texten.

```
1  void Cipher() {
2
3    KeyExpansion();
4
5    int i,j;
6    for(i = 0; i < 4;i++) {
7      for(j = 0; j < 4; j++) {
8        state[j][i] = plaintext[4*i+j];
9      }
10   }
11
12   int r = 0;
13   AddRoundKey(r);
14
15   for(r=1; r<10; r++) {
16     SubBytes();
17     ShiftRows();
18     MixColumns();
19     AddRoundKey(r);
20   }
21
22   SubBytes();
23   ShiftRows();
24   AddRoundKey(10);
25
26   for(i=0;i<4;i++)
27     {
28          for(j=0;j<4;j++)
29          {
30               ciphertext[i*4+j]=state[j][i];
31          }
32     }
33
34 }
```

Första funktionen, *KeyExpansion()*, är en funktion som tar krypteringsnyckeln och "stretchar" ut den till ett flertal nya nycklar, närmare bestämt en till varje enskild runda som AES skall köra. För en 128-bitars nyckel producerar funktionen alltså 10 nycklar (originalnyckeln + 9 nya).

AddRoundKey() tar aktuell subnyckel och utför XOR med S-matrisens samtliga element (subnyckel $\oplus$ S).

SubBytes() är en mappningsfunktion för varje byte i S-matrisen. AES har en hårdkodad *S-box* (även kallad Rijndael S-box), vilket är en 16×16-matris med 256 olika tecken. Tanken med denna S-box är att varje byte som funktionen SubBytes() skickar in skall substitueras till ett specifikt tecken i S-boxen. S-boxen i sin helhet finns återgiven i bilaga 3.

MixColumns() är den sista funktionen som körs i varje runda (bortsett den allra sista) och syftar till att skapa s.k *diffusion*, vilket innebär att man försöker göra sambandet mellan nyckel och krypterad text så komplex som möjligt. Funktionen

ändrar om innehållet i varje kolumnvektor i S genom att utföra matrismultiplikationer i $GF(2^8)$ där varje kolumn motsvarar ett polynom i den givna ringen modulo 2^8. Den här funktionen var den svåraste att implementera. Dock visade det sig att man kan hårdkoda boxar, precis som S-boxen, med de produkter som förekommer i ringen. Detta underlättade implementationen avsevärt.

4.3.4 Implementation

Vid konstruktionen av den AES-baserade pseudoslumptalsgeneratorn har operationstypen *Counter mode* (CTR) använts. Detta innebär att man använder en intern räknare i algoritmen som krypteras och inkrementeras hela tiden.

Detta var användbart eftersom man på ett enkelt sätt kan fortsätta mata algoritmen med indata som manipuleras och producerar pseudoslumptal och som inte direkt beror på varje utdata. Det finns olika sätt att implementera en kryptoalgoritm med CTR, exempelvis kan räknaren konkateneras med ett initialvärde och ett s.k *nonce* (slumpmässig data) innan själva inkrementationen startar[33]. I implementationen som gjorts har dock inget nonce använts, utan först initieras räknaren och därefter påbörjas en loop som inkrementerar räknaren och matar varje värde till krypteringsalgoritmen. Den i sin tur genererar ett tillstånd S som innehåller 16 byte med pseudoslumptal. Tillståndet S delas upp i fyra enskilda 32-bitars tal innan de matas ut, eftersom de andra algoritmerna som används i uppsatsen genererar 32-bitars tal. Detta är kompromissen som gjorts för att inte få en totalt orättvis jämförelse med de andra algoritmerna.

AES-baserad PRNG Pseudokod

$key \leftarrow inputkey$
$counter \leftarrow [000 \ldots 001]$
$KeyExpansion()$
repeat
  $state \leftarrow counter$
  $r \leftarrow 0$
  $AddRoundkey(r)$
  for $r = 1$ to 10 **do**
    $Subbytes()$
    $ShiftRows()$
    $MixColumns()$
    $AddRoundkey(r)$
  end for
  $Subbytes()$
  $ShiftRows()$
  $AddRoundkey(r)$

output ← *state*
counter ← *counter* + 1
until counter = n

Respektive funktion är precis samma som i ordinarie AES-kryptoalgoritm. Den stora skillnaden är att räknaren fungerar som frö och för varje loop ökas det aktuella räknarvärdet med ett. Detta pågår tills räknaren når n, vilket innebär att $4n$ hetal genererats. Räknaren är en publik char-array och beroende på vilket n man vill ha anger man vissa specifika värden på några loopar i main-koden som styr antalet genererade heltal.

Nyckeln som används i implementationen är slumpdata från en TRNG, erhållen från `random.org`[47] vars generator avläser atmosfäriskt brus.

Anm. Om 10^x heltal skall genereras måste en intern räknare sättas till $\frac{10^{x-1}}{4}$ eftersom varje omgång ger fyra stycken heltal. I implementationen används en for-loop som skriver ut enskilda bytes från tillståndsmatrisen, i mängder om 4 bytes.

Del III

Metod

Tillvägagångssätt

Utvärderingen av algoritmerna delades upp i två delar, nämligen *prestandatestning* och *statistisk testning*. Prestanda i det här sammanhanget definieras som tiden det tar att generera en viss mängd heltal (på en viss typ av system). Den statistiska testningen bestod i att spara en mängd heltal till fil och sedan utföra olika test över dels talen representerade som heltal (bytevis) och dels som en ström av rena databitar.

För att få en mer jämförande bild gjordes också en mätning av standardbiblioteket stdlib.h med dess inbyggda pseudoslumpgenerator rand(). Enligt `cplusplus.com`[26] varierar olika implementationers största värde som rand()-funktionen kan generera, men vanligast är det relativt låga heltalet 32767, vilket var fallet för det biblioteket som användes vid mätningen.

5.1 Testmiljö

Prestandatestningen valdes att göras på tre olika system: en dator med Linux (Ubuntu) i en av KTH's datorsalar i D-huset, en Windowsdator i en av KTH's datorsalar i D-huset samt en typisk modern bärbar dator (MS Windows Vista). Valet av testmiljöerna är gjorda med tanke på att dels testa prestandan i två av de största operativsystemen i världen idag, Linux och MS Windows, men också att testa på en typisk bärbar dator som så många datorägare använder idag (med MS Windows). Teknikens framfart i kombination med att datorer i allmänhet inte kostar alltför mycket, gör förmodligen att fler och fler användare har råd med mindre och effektivare datorer. En slutsats av detta är att mer och mer mjukvara, som nyttjar pseudoslumpalgoritmer, återfinns på sådana system, varför det ansetts viktigt att göra prestandatestningen även där. De skoldatorer som använts har haft nästan identiska hårdvaruspecifikationer.

Respektive testmiljös specifikationer återfinns nedan:

Linuxdator (skoldator)

Typ	Specifikation
Processor	Intel(R) Core(TM)2 Quad CPU Q9550, 2.83 GHz
RAM	3.81 GB
Hårddisk	178.2 GB
Operativsystem	Ubuntu 10.09 (lucid), linux kernel 2.6.32-30-generic

Windowsdator (skoldator)

Typ	Specifikation
Processor	Intel(R) Core(TM)2 Quad CPU Q9550, 2.83 GHz
RAM	3.25 GB
Hårddisk	232 GB
Operativsystem	MS Windows XP professional (2002 SP 3) (32-bitars)

Bärbar dator

Typ	Specifikation
Processor	AMD Turion(tm) X2 Dual-Core Mobile RM.75 2.20 GHz
RAM	4.00 GB
Hårddisk	200 GB
Operativsystem	MS Windows Vista Home Premium (SP 2) (32-bitars)

5.2 Testning av prestanda

För att försöka hålla det så objektivt som möjligt mellan varje algoritms prestandatestning utfördes tre ommätningar för varje parametervärde, där fyra olika sådana värden testades: 10^3 , 10^6 , 10^7 och 10^9 genererade heltal. Utdatat sparades binärt och filstorlekarna för samtliga utdata, för varje algoritm blev detsamma: 4 kB, 3 907 kB, 39 063 kB respektive 3 906 250 kB.

I prestandatestningen var det oviktigt med maximala periodlängder eftersom testningen endast ser till hur *snabbt* generatorn kan generera tal. Varje generator fick ett frö som slumpades fram genom den webbaserade TRNG-tjänsten på `random.org`[47]. Under själva körningen av respektive algoritm skrevs ingenting ut på skärmen. Detta för att det skulle ta orimligt lång tid att göra mätningarna. Däremot sparades utdatat till fil i binärform.

Microsoft Visual Studio 2010 användes vid kodningen av samtliga algoritmer.

Varje algoritm utrustades med en tidtagningsfunktion för att kunna mäta snabbheten i heltalsgenereringen. Biblioteket *time.h* användes och tiden mättes i sekunder:

```
1  #include <time.h>
2  \\ ...
3  \\ ...
4  clock_t start = clock();
5  \\ ...
6  \\ ...
7  \\ generate some numbers...
8  \\ ...
9  \\ ...
10 cout<<"Run time: "<<( ( clock() - start ) / (double)CLOCKS_PER_SEC ) <<
       " seconds."<<endl;
11 cout<<"Last generated integer: "<<test1<<" , after loopcount "<<(
       unsigned int)loopcount<<endl;
```

Anm. Kompileringen gjordes med *debug* och inte med *release*. Detta för att inte låta kompilatorn göra optimeringar. Skulle release använts kanske kompilatorn optimerar någon generator mer/mindre än någon annan generator, vilket ev. skulle kunna leda till orättvis jämförelse vid prestandatestningen. Vid ett test med release vid kompileringen snabbades tiderna upp drygt 10 gånger snabbare än vid debug-kompileringen.

5.3 Testning av slumpmässighet

Så som beskrivet i avsnitt 3.3.1 har ENT's testpaket använts vid den statistiska testningen. Med ENT så har tre typer av test gjorts, dels har hela testpaketet applicerats över utdatat i form av heltal (bytevis), och dels över utdatat i form av enskilda databitar. Slutligen mättes frekvensen av databitar för respektive utdata.

Resultat

6.1 Erhållen data vid prestandamätningen

I nedanstående tabeller presenteras de tider som uppmättes för respektive system när algoritmerna fick generera 4 kB, 3 907 kB, 39 063 kB samt 3 906 250 kB utdata. Tidsmätningen är angiven i sekunder.

Linuxdator (skoldator)

Utdata	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	0.010	0.010	0.010	0.010
	0.010	0.010	0.010	0.010
	0.010	0.010	0.010	0.010
3 907 kB	0.160	0.090	1.060	0.090
	0.160	0.090	1.060	0.050
	0.160	0.080	1.060	0.090
39 063 kB	1.570	0.510	10.960	0.500
	1.590	0.500	10.690	0.480
	1.570	0.500	10.840	0.500
3 906 250 kB	158.770	44.830	1034.130	43.540
	159.160	44.500	1107.936	43.380
	158.890	44.800	1063.170	44.010

Windowsdator (skoldator)

Utdata	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	0.000	0.000	0.000	0.000
	0.000	0.000	0.000	0.000
	0.000	0.000	0.000	0.000
3 907 kB	0.438	0.344	2.593	0.328
	0.468	0.328	2.593	0.328
	0.484	0.312	2.609	0.343
39 063 kB	4.516	3.593	25.875	3.406
	4.749	4.249	25.905	4.687
	5.796	4.843	25.905	3.421
3 906 250 kB	494.759	317.658	2584.820	341.830
	449.604	324.300	2611.931	344.768
	450.682	321.814	2599.387	341.660

Bärbar dator

Utdata	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	0.070	0.012	0.013	0.009
	0.070	0.080	0.013	0.009
	0.050	0.011	0.080	0.007
3 907 kB	1.031	0.558	4.561	0.700
	1.321	0.510	4.454	0.832
	1.125	0.575	4.519	0.715
39 063 kB	8.550	5.565	41.817	5.656
	8.275	5.539	42.646	5.701
	8.561	6.054	39.422	5.855
3 906 250 kB	970.616	825.496	4663.620	649.800
	1059.900	1061.550	4812.670	649.088
	1143.890	1102.910	4598.271	636.072

6.2 Erhållen data vid en statistiska mätningen

Varje generator fick generera 39 063 kB utdata i binärfiler. Därefter utfördes först det statistiska testbatteriet sedan frekvensberäkningen. På samtliga system sparades filen till lokal disk. Av utrymmesskäl för tabellen definieras deltesten för testbatteriet på följande vis:

1. **test 1** - Entropitestet (bit/symbol)

2. **test 2** - χ^2-testet
3. **test 3** - Aritmetiska medelvärdestestet
4. **test 4** - Monte-Carlo värde för Pi-testet
5. **test 5** - Korrelationskoefficienttestet

För fullständig information om mätvärdena, se avsnitt 3.3.1. Nedanstående data erhölls:

ENT bytevis testning

Deltest	LKG	LFSR	AES-baserad PRNG	rand()
test 1	7.954358	7.999950	7.999996	4.879639
test 2	Överstiger 2504126.04 slumpmässigt 0.01 %	Överstiger 2442.50 slumpmässigt 0.01 %	Överstiger 239.73 slumpmässigt 74.55 %	Överstiger 253656.46 slumpmässigt 0.01 %
test 3	111.4820	127.4580	127.5058	47.7465
test 4	10.99 % felmarg.	0.01% felmarg.	0.02 % felmarg.	27.32 % felmarg.
test 5	-0.049155	0.001059	0.000012	-0.056873

Vid mätning av utdatafilen med datat representerat som databitar erhölls:

ENT bitvis testning

Deltest	LKG	LFSR	AES-baserad PRNG	rand()
test 1	0.999295	1.000000	1.000000	0.785566
test 2	Överstiger 312490.78 slumpmässigt 0.01 %	Överstiger 3424.84 slumpmässigt 0.01 %	Överstiger 1.60 slumpmässigt 20.61 %	Överstiger 90314.14 slumpmässigt 0.01 %
test 3	0.4844	0.4998	0.5000	0.2344
test 4	10.99 % felmarg.	0.01 % felmarg.	0.02 % felmarg.	27.32 % felmarg.
test 5	-0.000932	0.000392	0.000017	0.259853

Avslutningsvis testades bitfrekvensen för varje utdatafil. Resultatet sammanfattas i följande tabeller:

Bitfrekvens LKG

Bit	Antal	Förekomst
0	164999910	0.515625
1	155000058	0.484375

Bitfrekvens LFSR

Bit	Antal	Förekomst
0	160052795	0.500165
1	159947205	0.499835

Bitfrekvens AES-baserad PRNG

Bit	Antal	Förekomst
0	160011310	0.500035
1	159988690	0.499965

Bitfrekvens rand()

Bit	Antal	Förekomst
0	244998873	0.765621
1	75001127	0.234379

För att utföra ovanstående frekvenstester användes kommandot *ent -c -b "filnamn"*.

Analys

7.1 Analys av erhållen data

Av de uppmätta värdena sammanställdes en rad tabeller med dels medelvärden i tidsprestandamätningen, snabbaste/långsammaste systemen/algoritmerna samt jämförelser från den statistiska mätningen.

Medelvärden av snabbhet (i sekunder) att generera heltal - Windowsdator (skoldator)

Anta heltal	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	0.000	0.000	0.000	0.000
3 907 kB	0.463	0.328	2.598	0.333
39 063 kB	5.020	4.228	25.895	3.838
3 906 250 kB	465.015	321.257	2598.712	342.752

Medelvärden av snabbhet (i sekunder) att generera heltal - Linuxdator (skoldator)

Anta heltal	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	0.010	0.010	0.010	0.010
3 907 kB	0.160	0.086	1.060	0.076
39 063 kB	1.576	0.503	10.830	0.493
3 906 250 kB	158.940	44.743	1068.412	43.643

Medelvärden av snabbhet (i sekunder) att generera heltal - Bärbar dator

Anta heltal	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	0.063	0.034	0.035	0.008
3 907 kB	1.159	0.547	4.511	0.749
39 063 kB	8.462	5.719	41.295	5.737
3 906 250 kB	1058.140	996.652	4691.520	644.986

Från ovanstående tabeller kan man sedan utläsa vilken generator som varit snabbast, och på vilket system. Det mest intressanta är att jämföra de två skoldatorerna eftersom de har nästintill identisk hårdvara (skiljer 0,56 Gb i RAM-minne och 53,8 Gb hårddisk) men olika operativsystem. Min analys av detta är att de skillnader i tid för algoritmerna som uppstått mellan de två sistnämnda datorerna till största del beror på *hur* respektive operativsystem arbetar med minneshanteringen, processer etc.

Sammanställningen av de snabbaste algoritmerna respektive de långsammaste följer i tabellerna nedan (angiven med system och tid i sek.), *Linux* står för Linuxdatorn, *WinXP* för Windowsdatorn i skolan samt *WinVista* för den bärbara datorn.

Snabbaste algoritmerna - Systemvis

Anta heltal	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	(WinXP, 0.000 s)	(WinXP, 0.000 s)	(WinXP, 0.000 s)	(WinXP, 0.000 s)
3 907 kB				(Linux, 0.076 s)
39 063 kB				(Linux, 0.493 s)
3 906 250 kB				(Linux, 43.643 s)

Snabbaste algoritmerna (bortsett rand()) - Systemvis

Anta heltal	LKG	LFSR	AES-baserad PRNG
4 kB	(WinXP, 0.000 s)	(WinXP, 0.000 s)	(WinXP, 0.000 s)
3 907 kB		(Linux, 0.086 s)	
39 063 kB		(Linux, 0.503 s)	
3 906 250 kB		(Linux, 44.743 s)	

Långsammaste algoritmerna - Systemvis

Anta heltal	LKG	LFSR	AES-baserad PRNG	rand()
4 kB	(WinVista, 0.063 s)			
3 907 kB			(WinVista, 4.511 s)	
39 063 kB			(WinVista, 41.295 s)	
3 906 250 kB			(WinVista, 4691.520 s)	

Då referensgeneratorn rand() inte förekommer bland någon av de långsammaste generatorerna behövs ingen ytterligare jämförelse presenteras med enbart de tre implementerade generatorerna.

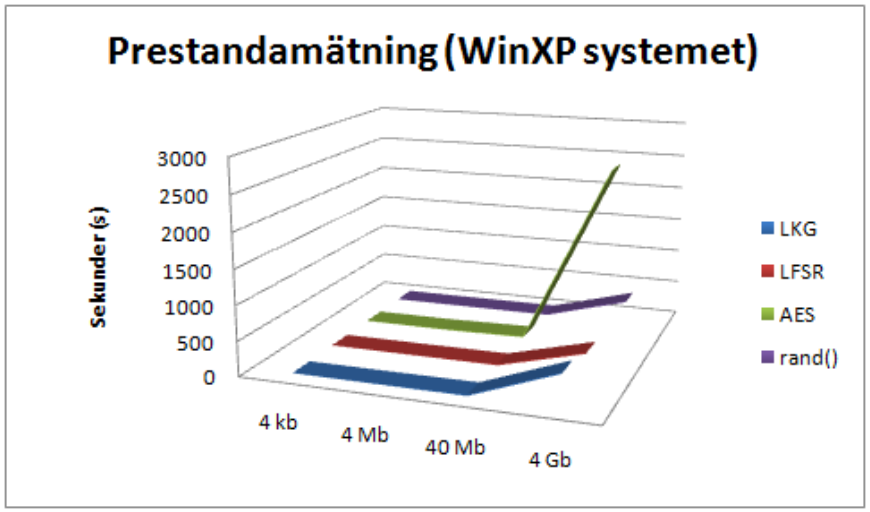
Den snabbaste generatorn var referensgeneratorn rand(). Den var relativt sett väldigt mycket snabbare än alla andra generatorer bortsett LFSR; de var nästan lika snabba i alla prestandatester - på alla system. Det intressanta är att då rand() är en LKG[23] hade den ändå mer prestandalikheter med LFSR-algoritmen än LKG-algoritmen.

Bortsett referensgeneratorn var den snabbaste algoritmen, oavsett hårdvara, LFSR implementationen. Detta var under samtliga körningar över samtliga mängder heltal. Resultatet är naturligt då algoritmen endast använder sig av XOR- och skiftoperationer medans LKG och AES-baserade PRNG använder mer krävande operationer ur ett tidskomplexitetsmässigt perspektiv. Givetivis var den stora tidsskillnaden mellan AES-baserade PRNG och de övriga två var förväntad eftersom den dels hanterar större datablock, 128 bitar, och som nyss nämnt också innehåller fler operationer än de övriga.

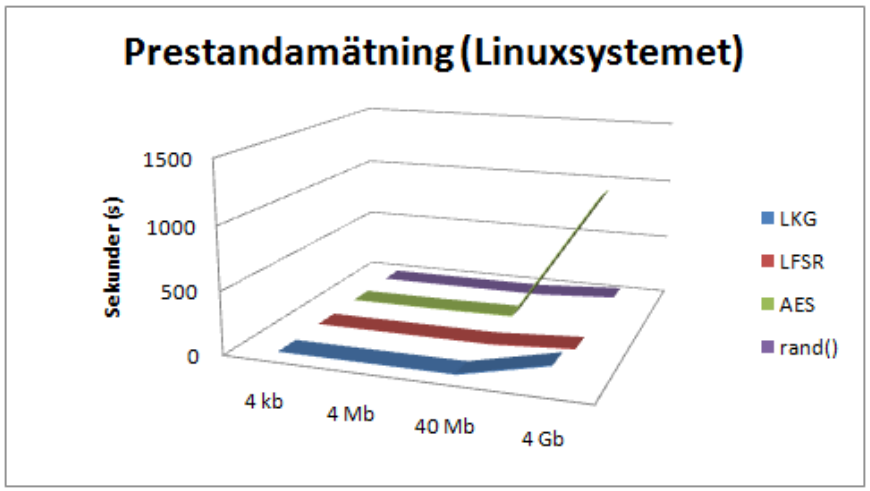
För att eliminera andra orsaker som kan påverka resultatet kontrollerades mängden RAM-minne som vardera algoritm upptog, vilket visade sig vara nästintill identiska. Spannet låg mellan ca. 900-950 Kilobyte för samtliga. Vidare visade det sig att Linuxsystemet endast nyttjade en kärna av fyra möjliga under körning av algoritmerna. Oavsett körtid eller huruvida algoritmen sparade till fil så valdes en kärna att sköta processen. Det skiftade till synes slumpmässigt vilken kärna som operativssystemet valde att nyttja. Windows XP däremot valde att fördela arbetet jämnt över alla fyra kärnor, vid varje körning, oavsett storlek eller körtid. Detta kan kanske förklara vissa av de skillnader som uppstod under mätningen. Det skall noteras att ingen av algoritmerna är optimerade för parallellkörning och trådning, fast man i vissa fall skulle kunna göra det. Dessutom användes inte release vid kompliering av programkoden.

Med mätdatat som underlag kan man konstatera att den AES-baserade algoritmen, oavsett system/hårdvara den körs på, är den som uppenbarligen tar längst tid. I fallet där 10^9 heltal genererades (3 906 250 kB utdata) tog det runt timmen att generera utdata, vilket är en avsevärt stor skillnad till de andra algoritmernas sämsta körtider för samma generering. Givietvis är den AES-baserade algoritmen också avsevärt mer komplicerad än de andra två, dels genom att den innehåller så många steg, men också för att den utför en del matrisoperationer (kopiering) och andra inre loopar. Linux på skoldatorns hårdvara var dock det system som hanterade algoritmen bäst; nästan 4 gånger snabbare än Windows Vista på den bärbara datorn.

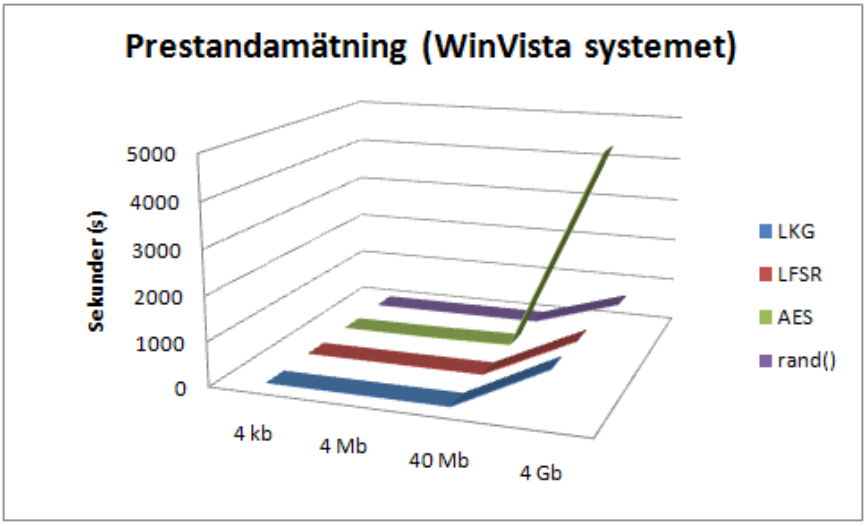
För att åskådliggöra mätdatat och analysen lättare presenteras här en rad diagram över ovan angivna mätdata. Fokus ligger på jämförelsen mellan skoldatorerna med olika operativsystem då jag anser den jämförelsen är mer intressant.



Figur 7.6. Algoritmanalys gällande prestanda på Windows XP (skoldator)

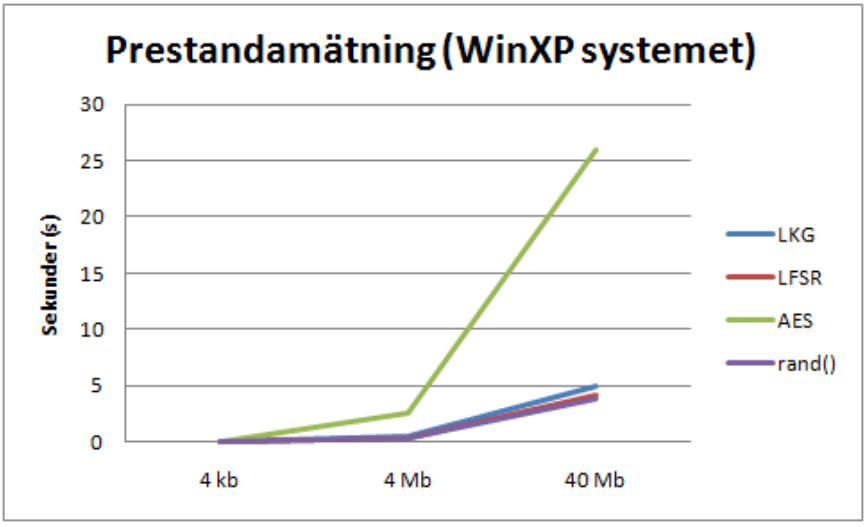


Figur 7.7. Algoritmanalys gällande prestanda på Linux (skoldator)

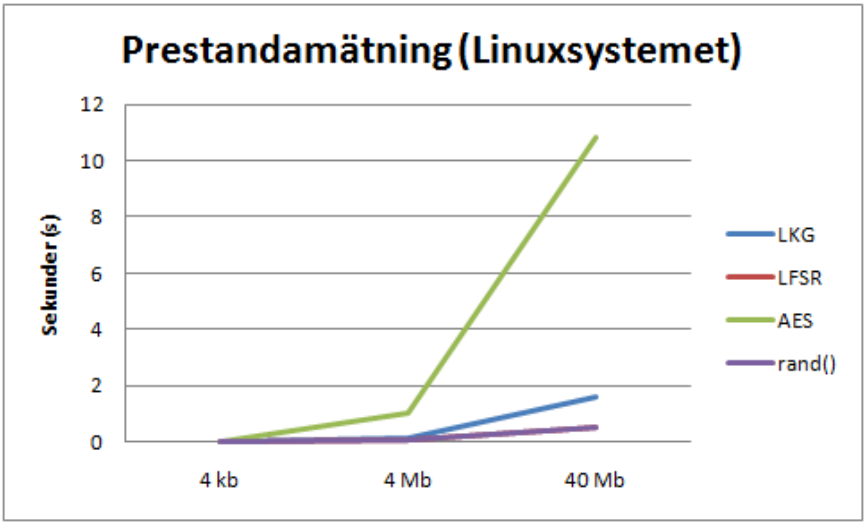


Figur 7.8. Algoritmanalys gällande prestanda på Windows Vista (bärbar dator)

Som diagrammen antyder så är det ingen större *relativ* skillnad i tid systemen emellan, tidsavvikelserna ser likadana ut för alla. Den drastiska ökningen i tid över 39 063 kB till 3 906 250 kB utdata växer på samma sätt. Eftersom den sista mätningen ger en så stor tidsskillnad slätas de övriga mätningarna ut i bottnen på diagrammen, därför följer en mer detaljerad version av diagrammen som exkluderar sista mätningen.



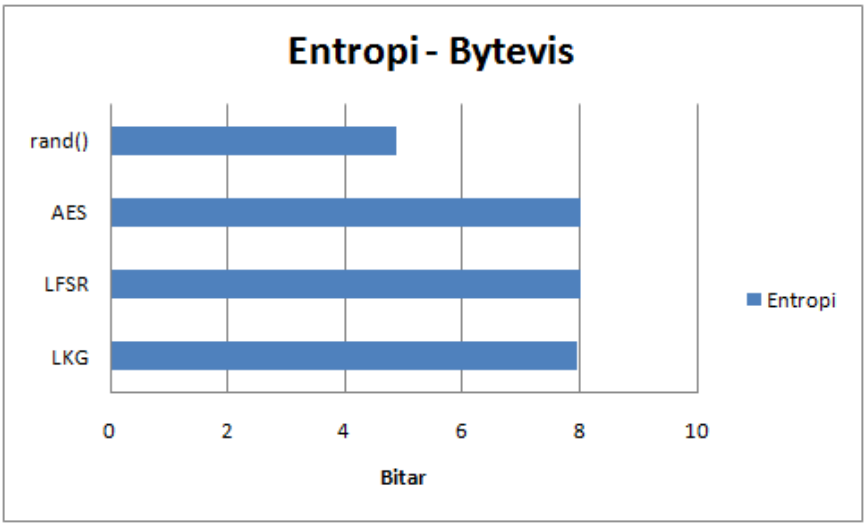
Figur 7.9. Algoritmanalys gällande prestanda på Windows XP (skoldator)



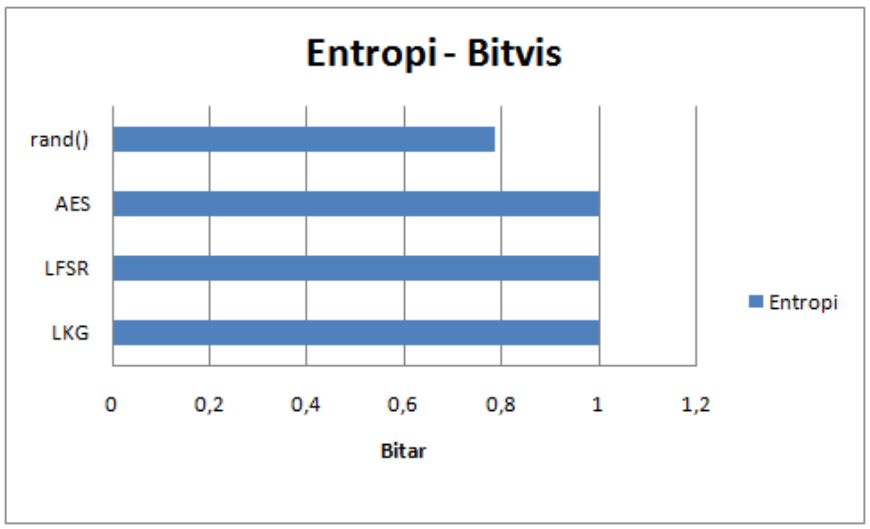
Figur 7.10. Algoritmanalys gällande prestanda på Linux (skoldator)

På det sista diagrammet ser man hur LFSR och rand() i princip följer samma linje, vilket innebär att de är likvärdiga i prestandamätningen upp till ca 40 Mb genererad utdata.

Fortsättningsvis redovisas resultatet av den statistiska mätningen med ENT.

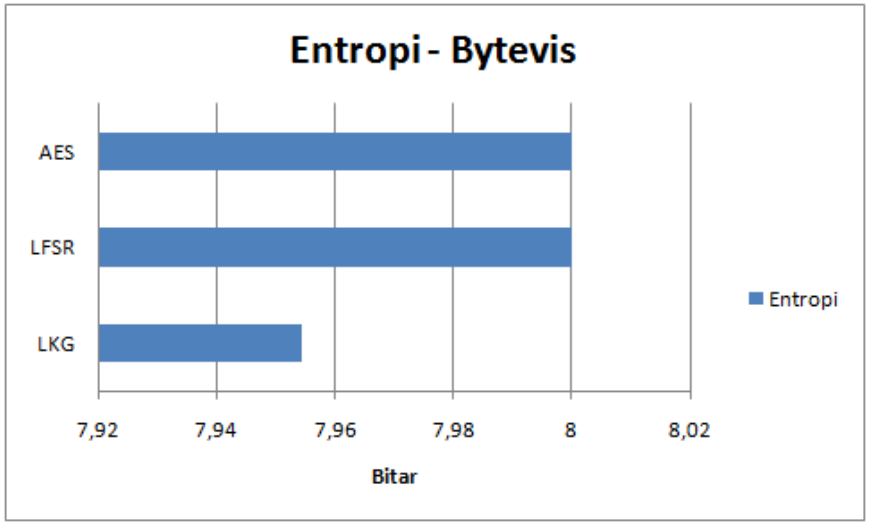


Figur 7.11. Mätning av entropi bytevis



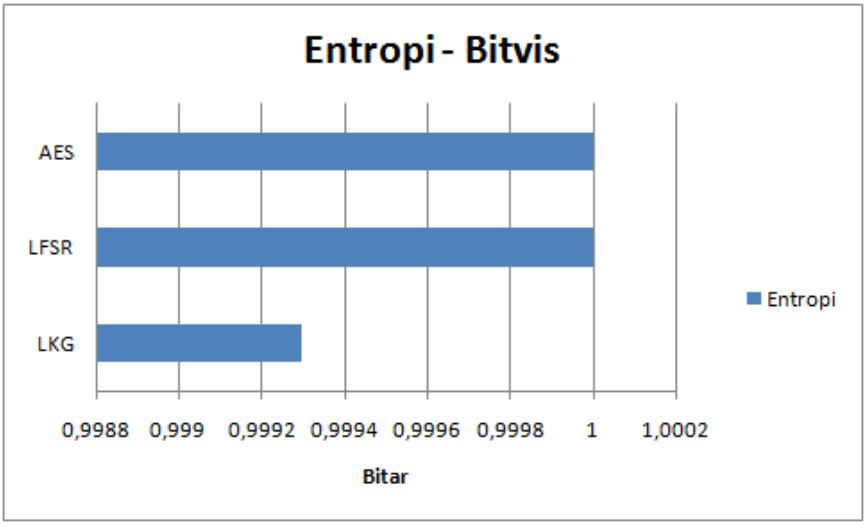
Figur 7.12. Mätning av entropi bitvis

Liksom diagrammen för prestandamätningen så ser man inte alltför tydligt skillnaden i entropi mellan generatorerna exklusive referensgeneratorn rand(). Därför presenteras ytterligare två diagram i mer detalj för LKG, LFSR och AES-baserade PRNG.



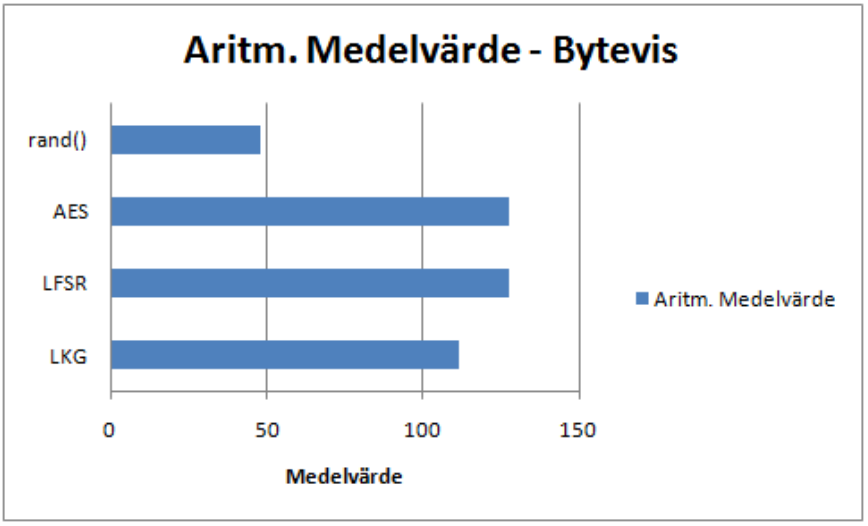
Figur 7.13. Mätning av entropi bytevis

Som förväntat har den AES-baserade PRNG högst entropi, medans LKG (och rand()) hade lägst. Rand() hade dessutom signifikant lägre entropi vilket tyder på dess dåliga statistiska slumpegenskaper i utdatat.

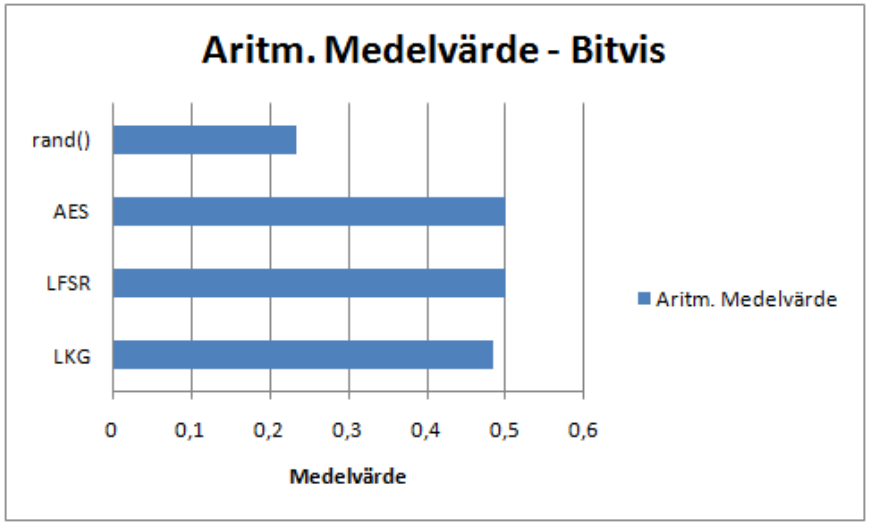


Figur 7.14. Mätning av entropi bitvis

Under χ^2-testningen, så misslyckades alla generatorer utom den AES-baserade PRNG. Både bytevis och bitvis. Värden nära 0.01% eller 0.99% anses som att testet misslyckats (ej slumpegenskaper) och LKG, LFSR samt rand() hade 0.01 % medans AES-baserade PRNG hade 74.55 % och 20.61 %. Därmed presenteras inga diagram över χ^2-testets mätdata.

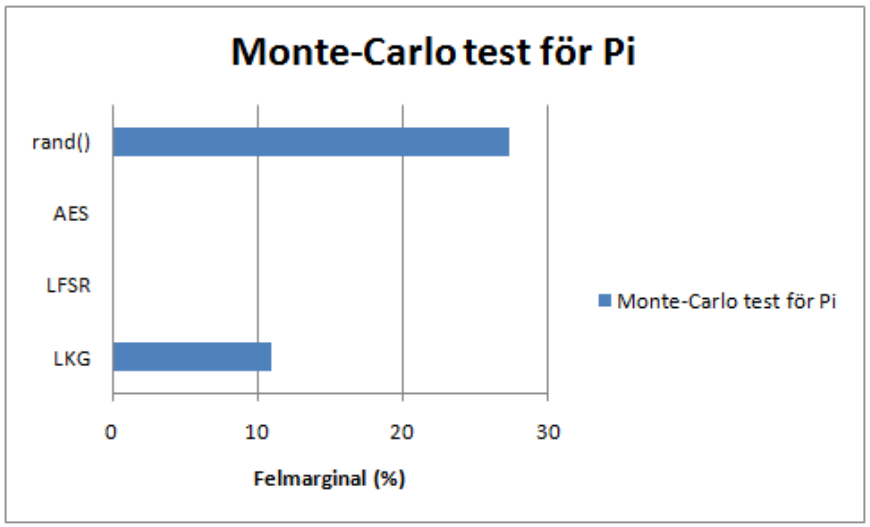


Figur 7.15. Mätdata av aritmetiska medelvärdet-testet bytevis



Figur 7.16. Mätdata av aritmetiska medelvärdet-testet bitvis

Gällande det aritmetiska medelvärdestestet skall värdet vara så nära 127,5 som möjligt för att ha slumpmässiga egenskaper (bytevis) eller så nära 0,5 som möjligt (bitvis). AES-baserade PRNG och LFSR var de algoritmer som var närmast i båda fallen (dessutom ganska nära varandra) medans rand() var i särklass sämst.

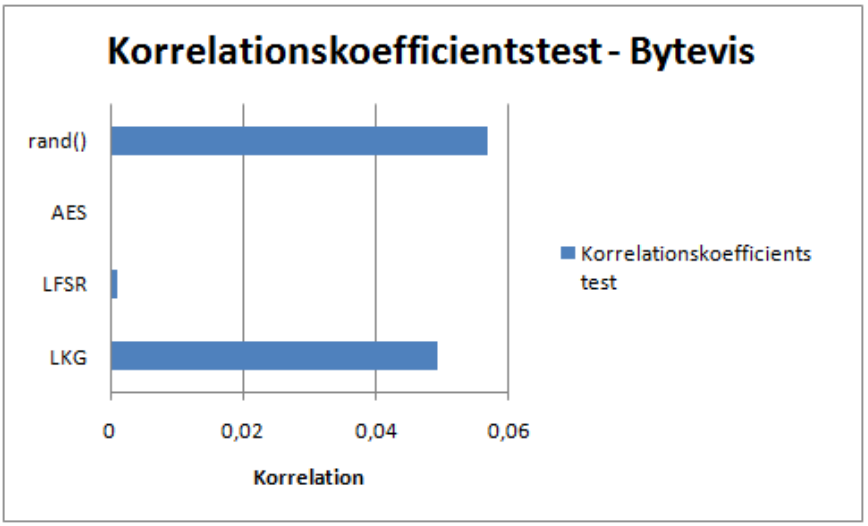


Figur 7.17. Mätdata av Monte-Carlo testningen för pi, både för byte- och bitvis mätning.

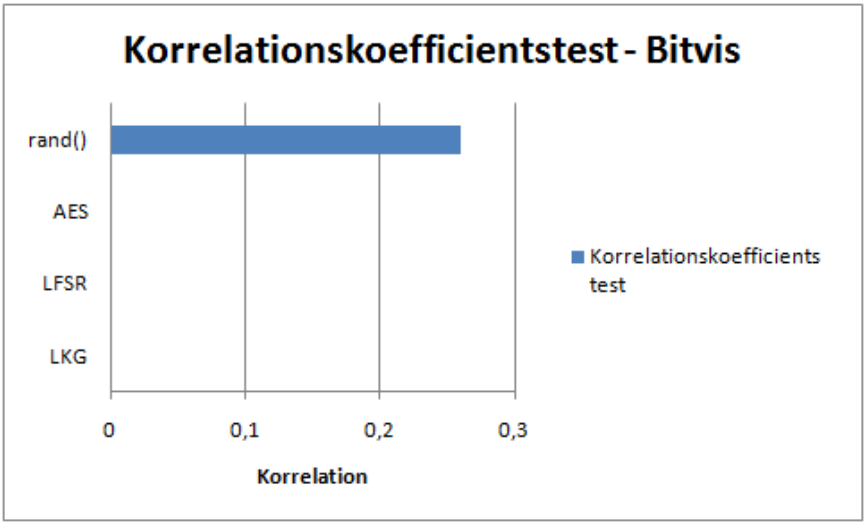
Nästa mätning som presenteras är Monte-Carlo testningen för Pi. Testvärdet som fås av ENT är en procentangivelse av felmarginalen till hur nära π framräknas.

Ju lägre procent, dessto bättre. AES-baserade PRNG och LFSR hade nästan ingen felmarginal alls, 0.02 % respektive 0.01 %. Detta var det enda testet där LFSR hade bättre resultat än AES-baserade PRNG.

Samma felmarginaler erhölls vid både bytevis mätning och bitvis mätning vilket presenteras i diagrammet i figur 7.17.



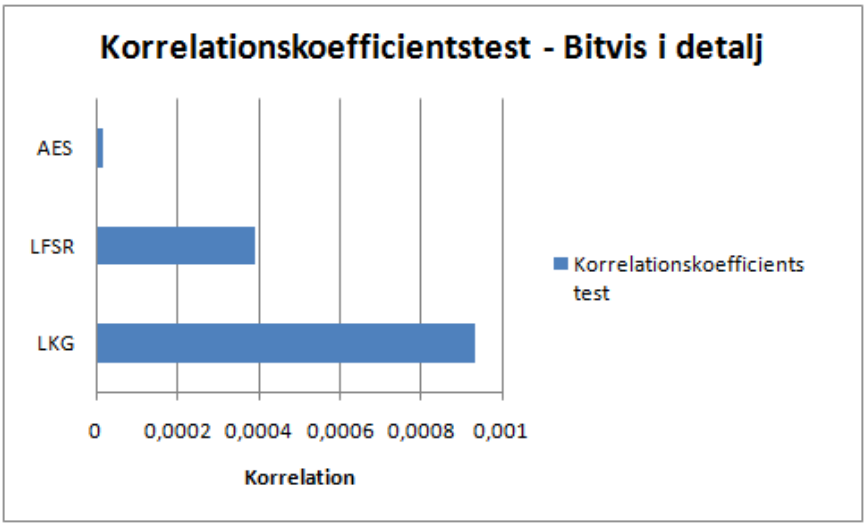
Figur 7.18. Mätdata av korrelationskoeffecientstestet, bytevis



Figur 7.19. Mätdata av korrelationskoeffecientstestet, bitvis

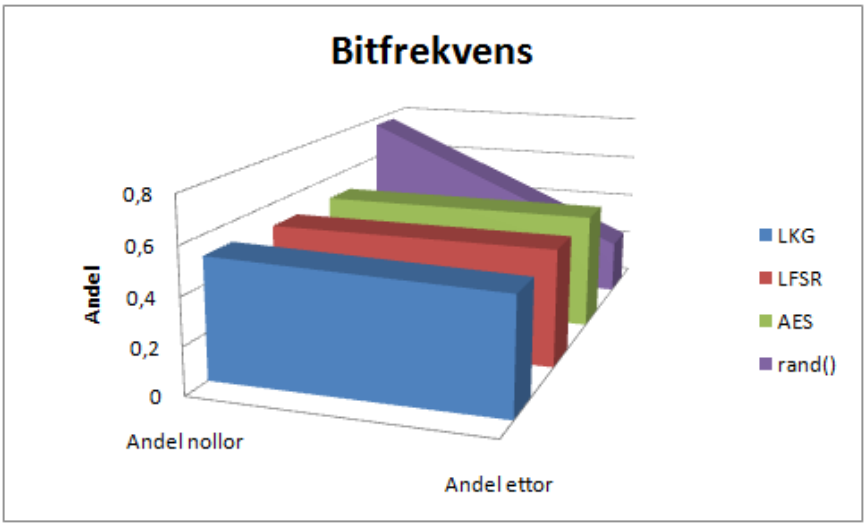
I den sista statistiska mätningen skall värdet ligga så nära noll som möjligt för att anses ha goda slumpmässiga egenskaper. Vissa mätvärden angavs som negativa tal av ENT, och har därför har dessas absolutbelopp använts i diagramet. Detta gör ingen skillnad eftersom det inte spelar nån roll om värdet ligger till höger eller vänster om tallinjen, eftersom beloppet är lika stort oavsett.

Då rand() hade avsevärt sämre värde på den bitvisa testningen syns inte riktigt de övriga generatorernas relativa korrelationer. I figur 7.20 finns ett mer detaljerat diagram över LKG, LFSR och AES-baserade PRNG för den bitvisa mätningen.



Figur 7.20. Mätdata av korrelationskoeffecientstestet, bitvis med rand() exkluderat

Slutligen gjordes en sammanställning av databitsfrekvenserna.



Figur 7.21. Frekvens av databitar

Funktionen rand() hade störst avvikelse mellan frekvensen av nollor och ettor, med nollor som mest förekommande bit. Övriga algoritmer har också fler nollor än ettor, men inte med samma avvikelse som rand(). Statistiskt sett borde ett värde som påvisar slumpmässighet vara jämnt fördelat mellan bitarna, dvs. 0.5 för varje bit (likafördelad fördelning). Den algoritm som ligger närmast detta värde är AES-baserade PRNG med 0.500035 % nollor och 0.499965 % ettor vilket ger en differens på ungefär 0,003%. Referensgeneratorn rand() har i kontrast till AES-baserade PRNG ungefär 53% differens i andelarna förekommande databitar.

Del IV

Slutsats

Diskussion

Av analysen som gjordes i föregående avsnitt kan man konstatera att olika typer av algoritmer är olika snabba gällande generering av utdata och besitter olika pseudoslumpmässiga egenskaper. Jag tror vidare att den statistiska testningen inte är tillfredsställande nog, och att fler test bör göras för att få en bättre bild av de slumpmässiga egenskaperna hos algoritmerna. Användningen av NISTs testbatteri[11] bör också användas tillsammans med ENT för att på så sätt kunna få fler testmetoder.

Olyckligtvis upptäcktes det i slutet av processen att både rand() och LKG har gett otillräcklig utdata; algoritmerna har inte genererat heltal i intervallet $2^{32} - 1$, vilket innebär att stora delar av de mest signifikanta bitarna för ett heltal blir nollor (tex. rand() har intervallet $2^{15} - 1$ och då är det 17 mest signifikanta bitar som är 0 och LKG ger unsigned heltal men med en modulo $m = 2^{31}$) . Detta påverkar den statistiska mätningen och gör den missvisande eftersom stora delar av de genererade bitarna är nollor, vilket också ses i frekvensmätningen (tex. rand() hade runt 76% bitar som var nollor).

Med den statistiska testning som nu ändå gjorts konstateras att den förbättring av slumpmässighet som den långsammaste algoritmen, AES-baserade PRNG, har i förhållande till de övriga måste man vara noga med att veta vad man behöver från sin pseudoslumpgenerator. Är det viktigare med att snabbt generera mycket slumpdata, då kommer man att få bristande slumpegenskaper. Vill man hellre ha goda slumpegenskaper i sin genererade utdata får man räkna med att prestandan blir sämre. Men som testerna visade så var ändå alla generatorer snabba i den mening att man kan räkna genereringen i sekunder, fram tills man genererat utdata i storleksordningen tiotals megabyte. Har man inte behov av större mängder än 10-20 Mb bör AES-baserade PRNG användas.

AES-baserade PRNG gjorde bättre ifrån sig på alla statistiska test utom test 5, korrelationskoefficienttestet, där LFSR var 0,01 % bättre. Dessutom misslyckades de andra generatorerna med χ^2-testet och hade signifikant sämre resultat på de övriga testen, med undantag för LFSR som ändå inte låg alltför långt ifrån AES-baserade PRNG.

Fortsättningsvis anser jag att av de tre testade algoritmerna är AES-baserade PRNG den med bäst egenskaper, vilket också var förväntat. LKG och LFSR har givetvis inte katastrofalt dåliga egenskaper, men det finns andra argument som talar mot ett användade av dem: det finns metoder att utföra kryptanalys på både LKG[27] och LFSR[25] som gör dem sårbara om utdatat används i kryptosam-

manhang. Om man bortser från sådana säkerhetskrav är LFSR annars ett bra val eftersom den är såpass snabb och ändå hade ganska bra värden på vissa av testen.

För att summera resultatet av detta arbete anser jag mig ha påvisat hur viktigt det är med att välja rätt pseudoslumpalgoritm till rätt applikation, och att fall där krav på säkerhet är viktiga, bör pseudoslumptalsalgoritmer ses över extra noga om sådana är involverade. Dessutom har jag kommit fram till att av de tre inspekterade algoritmerna är LFSR den snabbaste och jämförelsevis bästa algoritmen ur prestandamässig synvinkel, även om jag ej skulle rekommendera den till applikationer med starka krav på säkerhet. Bortsett dess säkerhetsbrister så anser jag den enkel att implementera och samtidigt väldigt snabb. Med ökade krav på slumpegenskaper bör den AES-baserade PRNG väljas framför de andra tre testade generatorerna, då ENT påvisat stora skillnader i den statistiska mätningen.

Del V

Referenser och bilaga

Litteraturförteckning

Litteratur och publikationer

[1] Chaitin, Gregory J *Exploring Randomness*. Springer, 2001. p.18.

[2] Cusick, Thomas W. Stanica, Pantelimon *Cryptographic boolean functions and applications* Academic Press. 2009. p.19.

[3] Goldberg, Ian. Wagner, David. *Architectural considerations for cryptanalytic hardware*, p.6-7. Funnen på: `http://www.cs.berkeley.edu/~daw/papers`
Besökt: 09-06-2011

[4] Gutterman, Zvi & Pinkas, Benny *Analysis of the Linux Random Number Generator* 2006.
Funnen på: `http://eprint.iacr.org/2006/086`
Besökt: 09-06-2011

[5] Jun, Benjamin. Kocher, Paul *The INTEL(R) Random Number Generator*, 1999.
Funnen på: `http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf`
Besökt: 09-06-2011

[6] Knuth, Donald E. *The art of computer programing Vol.2* 2nd ed. Addison-Wesley Educational Publishers Inc. 1981. p.4.

[7] Knuth, Donald E. *The art of computer programing Vol.2* 2nd ed. Addison-Wesley Educational Publishers Inc. 1981. p.5.

[8] Knuth, Donald E. *The art of computer programing Vol.2* 2nd ed. Addison-Wesley Educational Publishers Inc. 1981. p.16.

[9] Knuth, Donald E. *The art of computer programing Vol.2* 2nd ed. Addison-Wesley Educational Publishers Inc. 1981. p.19.

[10] Knuth, Donald E. *The art of computer programing Vol.2* 2nd ed. Addison-Wesley Educational Publishers Inc. 1981. p.41.

[11] NIST, *Publication 800-22 rev ed. 1a*, 2010 Funnen på: `http://csrc.nist.gov/publications/PubsSPs.html`
Besökt: 09-06-2011

[12] NIST *Federal Information Processing Standards Publication 197*, 2001
Funnen på: `csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
Besökt: 09-06-2011

[13] Perepelitsa, Dennis V. *Johnson Noise and Shot Noise*, 2006.
Funnen på: `http://web.mit.edu/dvp/www/Work/8.13/`
Besökt: 09-06-2011

[14] Peterson, Ivars *The Jungles of Randomness: A Mathematical Safari*, John Wiley & Sons, 1997. p.170.

[15] S. K. Park and K. W. Mille.*Random Number Generators: Good Ones Are Hard To Find* 1988
Funnen på: `http://portal.acm.org/citation.cfm?id=63042`
Besökt: 09-06-2011

[16] Schlundt, Conrad *MAXIM Application note 4400*, p.1. Funnen på: `http://pdfserv.maxim-ic.com/en/an/AN440.pdf`
Besökt: 09-06-2011

[17] Schneider, Thomas D *Information Theory Primer With an Appendix on Logarithms*, 2010.
Funnen på: `http://www-lmmb.ncifcrf.gov/~toms/paper/primer/`
Besökt: 09-06-2011

[18] Schneier, Bruce *Applied cryptography*, John Wiley & Sons, 2nd ed. 1995. p.76.

[19] Schneier, Bruce *Applied cryptography*, John Wiley & Sons, 2nd ed. 1995. p.508-509.

[20] Stinson, Douglas R. *Cryptography - Theory and practice*, Chapman & HALL/CRC, 3rd ed. 2005. p.323.

[21] Stinson, Douglas R. *Cryptography - Theory and practice*, Chapman & HALL/CRC, 3rd ed. 2005. p.325.

[22] Teukolsky, Saul. Vetterling, William. Flannery, Brian *Numerical recipes in C*,Cambridge University Press, 1992. p.274.

[23] Teukolsky, Saul. Vetterling, William. Flannery, Brian *Numerical recipes in C*,Cambridge University Press, 1992. p.276.

[24] Von Neumann, John *Various techniques used in connection with random digits*, Applied Mathematics Series, 1951. no 12,36-28.

[25] Zenner, Erik *Cryptanalysis of LFSR-based pseudorandom generators*, 2004
Funnen på: http://th.informatik.uni-mannheim.de/pub/zenner04b.pdf
Besökt: 09-06-2011

Internetsidor

[26] Cplusplus.com http://www.cplusplus.com/reference/clibrary/cstdlib/rand/
Besökt: 09-06-2011

[27] Cryptanalysis of linear congruence generators http://www.xs4all.nl/~itsme/projects/crypto/lcg.html
Besökt: 09-06-2011

[28] ENT http://www.fourmilab.ch/random/
Besökt: 09-06-2011

[29] IDQ, http://www.idquantique.com
Besökt: 09-06-2011

[30] LETech, http://letech-rng.jp
Besökt: 09-06-2011

[31] Linear Feedback Shift Registers http://homepage.mac.com/afj/lfsr.html
Besökt: 09-06-2011

[32] New Wave Instruments http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm
Besökt: 09-06-2011

[33] RFC 3686 - Using Advanced Encryption Standard (AES) Counter Mode http://www.faqs.org/rfcs/rfc3686.html
Besökt: 08-06-2011

[34] Whittle, Robin http://www.firstpr.com.au
Besökt: 09-06-2011

[35] Wikipedia om Slump, http://sv.wikipedia.org/wiki/Slump
Besökt: 08-06-2011

[36] Wikipedia on Information theory, http://en.wikipedia.org/wiki/Information_entropy
Besökt: 08-06-2011

[37] Wikipedia on Noise (electronics), http://en.wikipedia.org/wiki/Noise(electronics)
Besökt: 08-06-2011

[38] Wikipedia on Zener diodes http://en.wikipedia.org/wiki/Zener_diode.
Besökt: 08-06-2011

[39] Wikipedia on Linear congruential generators, http://en.wikipedia.org/wiki/Linear_congruential_generator
Besökt: 08-06-2011

[40] Wikipedia on Park-Miller RNG, http://en.wikipedia.org/wiki/Park-Miller_RNG
Besökt: 08-06-2011

[41] Wikipedia on Linear feedback shift register, http://en.wikipedia.org/wiki/Linear_feedback_shift_register
Besökt: 08-06-2011

[42] Wikipedia on Modulo operation, http://en.wikipedia.org/wiki/Modulo_operation
Besökt: 08-06-2011

[43] Wolfram Alpha on Euclid's Theorem, http://mathworld.wolfram.com/EuclidsTheorems.html
Besökt: 09-06-2011

[44] Wolfram Alpha on Relatively prime, http://mathworld.wolfram.com/RelativelyPrime.html
Besökt: 09-06-2011

[45] Park-Miller-Carta Pseudo-Random Number Generator http://www.firstpr.com.au/dsp/rand31/
Besökt: 09-06-2011

[46] Protego, http://www.protego.se.
Besökt: 09-06-2011

[47] Random.org http://www.random.org/bytes.
Besökt: 08-06-2011

[48] Random.org http://www.random.org/randomness.
Besökt: 08-06-2011

[49] Random.org http://www.random.org/analysis/.
Besökt: 08-06-2011

[50] RobertNZ, http://robertnz.net/true_rng.html.
Besökt: 09-06-2011

[51] Simtec Electronics, http://entropykey.co.uk.
Besökt: 09-06-2011

.1 LKG implementation

```
1
2  #include <iostream>
3  #include <string>
4  #include <cmath>
5  #include <time.h>
6  #include <fstream>
7
8  using namespace std;
9
10 //
11 // rand31pm
12 //
13 // 31 bit Pseudo Random Number Generator based on Park Miller "Integer
14 // Version 1" - but done with double-precision floating point so we are
       not
15 // concerned with the limits of integer operations.  This is not
       intended for
16 // fast operation - but *maybe* it would be faster than the integer
17 // implementation on some CPUs.
18 //
19 // Methods:
20 //
21 //     seedi     Set seed with a 31 bit unsigned integer between 1 and
22 //               0x7FFFFFFE inclusive.  Don't use 0!
23 //
24 //     ranlui    Provides the next pseudorandom number as a long unsigned
25 //               integer (31 bits).
26 //
27 //     ranf      Provides the next pseudorandom number as a float between
28 //               nearly 0 and nearly 1.0.
29
30 class rand31pm {
31                                       // The sole item of state - a 32
                                             bit
32                                       // integer.
33     unsigned int seed31;
34
35 public:
36                                       // Constructor sets seed31 to 1,
37                                       // in case no seedi operation is
38                                       // used.
39     rand31pm() {seed31 = 1;}
40
41                                       // Declare methods.
42
43     void              seedi(long unsigned int);
44     unsigned int ranlui(void);
45     float             ranf(void);
46
47
48 private:
```

```
49                                      // nextrand()
50                                      //
51                                      // Generate next pseudo-random
                                            number.
52
53                                      // Multiplier constant = 16807 =
                                            7^5.
54                                      // Park and Miller in 1993
                                            recommend
55                                      // 48271.
56      #define consta 16807
57                                      // Modulus constant = 2^31 - 1 =
58                                      // 0x7FFFFFFF.  Use .0 to deter
                                            compiler
59                                      // from complaining about a very
                                            large
60                                      // integer constant.
61      #define constm 2147483647.0
62
63      unsigned int nextrand()
64      {
65          double const a = consta;
66          double const m = constm;
67                                      // This is the linear congrentual
68                                      // generator:
69                                      //
70                                      // Multiply the old seed by
                                            constant a
71                                      // and take the modulus of the
                                            result
72                                      // (the remainder of a division) by
73                                      // constant m.
74
75          seed31 = (fmod((seed31 * a), m));
76
77          return (seed31);
78      }
79  };
80
81                                      //
82                                      // Implementations of the methods.
83
84                                      // seedi()
85                                      //
86                                      // Set the seed from a long
                                            unsigned
87                                      // integer.  If zero is used, then
88                                      // the seed will be set to 1.
89
90  void rand31pm::seedi(unsigned int seedin)
91  {
92      if (seedin == 0) seedin = 1;
93      seed31 = seedin;
94  }
```

```
95
96                                      // ranlui()
97                                      //
98                                      // Return next pseudo-random value
                                            as
99                                      // a long unsigned integer.
100
101 unsigned int rand31pm::ranlui(void)
102 {
103     return nextrand();
104 }
105
106                                     // ranf()
107                                     //
108                                     // Return next pseudo-random value
                                            as
109                                     // a floating point value.
110 float rand31pm::ranf(void)
111 {
112     return (nextrand() / constm);
113 }
114
115
116 int main ()
117 {
118                                     // Instantiate two random number
                                            objects.
119     rand31pm ran;
120   unsigned int test1;
121   int seed = 1;
122   ran.seedi(seed);
123
124   clock_t start = clock();
125   double loopcount = 1;
126
127   ofstream outFile;
128   outFile.open("lkgutdata.bin", ios::out | ios::app | ios::binary);
129
130   for(loopcount = 1; loopcount<10000000;loopcount++)
131   {
132     test1 = ran.ranlui();
133     outFile.write(reinterpret_cast<char *>(&test1),sizeof(test1));
134   }
135
136   outFile.close();
137
138   cout<<"Run time: "<<( ( clock() - start ) / (double)CLOCKS_PER_SEC )
        <<" seconds."<<endl;
139   cout<<"Last generated integer: "<<test1<<" , after loopcount "<<(
        unsigned int)loopcount<<endl;
140   cin.get();
141
142     return 0;
143
```

```
144
145 }
```

.2 LFSR implementation

```
1  /*
2  * Hannes Salin, CDATE3. 23-02-2011
3  * Simple LFSR fibonacci-style implementation,
4  * Uses four taps and a "32 bit register".
5  * This tapsequence does not give a maximal period,
6  * but > 1000000.
7  */
8
9  #include <iostream>
10 #include <time.h>
11 #include <fstream>
12
13 using namespace std;
14
15 int main() {
16
17   // 00000000 00000000 00000000 00000001
18   unsigned int lfsr = 0x00000001;
19   bool check = false;
20
21   unsigned int bit1, bit2, bit3, bit4, output;
22   int period = 0;
23
24   clock_t start = clock();
25
26   ofstream utData;
27   utData.open("lfsrutdata.bin", ios::out | ios::app | ios::binary);
28
29   while(check == false) {
30
31     // tap bits = 32,31,25,1
32     // 11000001 00000000 00000000 00000001
33
34     bit1 = lfsr & 0x00000001;
35     bit2 = lfsr & 0x80000000;
36     bit3 = lfsr & 0x40000000;
37     bit4 = lfsr & 0x01000000;
38
39     bit2 = bit2 >> 31;
40     bit3 = bit3 >> 30;
41     bit4 = bit4 >> 24;
42
43     // execute the XOR-operations
44
45     output = (((bit1 ^ bit2) ^ bit3) ^ bit4);
46
```

```
47      // if the output was not a zero,
48      // set the 'bit' = 1.
49
50      if(output != 0) {
51        output = 0x00000001;
52      }
53
54      // do the actual shifting
55
56      lfsr = lfsr >> 1;
57      lfsr =  lfsr | (output << 31);
58      period++;
59
60      if (period == 10000000 ) {
61        check = true;
62      }
63
64      utData.write(reinterpret_cast<char*>(&lfsr),sizeof(lfsr));
65    }
66
67    utData.close();
68
69    cout<<"length: "<<period<<endl;
70    cout<<"Run time: "<<( ( clock() - start ) / (double)CLOCKS_PER_SEC )
          <<" seconds."<<endl;
71    cout<<"Last generated integer: "<<lfsr<<endl;
72
73    return 0;
74  }
```

.3 AES-baserad PRNG implementation

```
1  /*
2  *
3  * Hannes Salin, CDATE3. 03-04-2011
4  * My implementation of a AES-based PRNG; the s_box and rcon
5  * was taken from Wikipedia and the actual understanding for
6  * AES was gained from:
7  * http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
8  *
9  * This is a AES-based PRNG with counter mode (CTR) and is not
10 * a cryptographic algorithm.
11 */
12
13 #include <stdio.h>
14 #include <iostream>
15 #include <time.h>
16 #include <fstream>
17
18 using namespace std;
19
```

```
20 const int mul2[256] = {0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0
       x12,0x14,0x16,0x18,0x1a,0x1c,0x1e,0x20,0x22,0x24,0x26,0x28,0x2a,0
       x2c,0x2e,0x30,0x32,0x34,0x36,0x38,0x3a,0x3c,0x3e,0x40,0x42,0x44,0
       x46,0x48,0x4a,0x4c,0x4e,0x50,0x52,0x54,0x56,0x58,0x5a,0x5c,0x5e,0
       x60,0x62,0x64,0x66,0x68,0x6a,0x6c,0x6e,0x70,0x72,0x74,0x76,0x78,0
       x7a,0x7c,0x7e,0x80,0x82,0x84,0x86,0x88,0x8a,0x8c,0x8e,0x90,0x92,0
       x94,0x96,0x98,0x9a,0x9c,0x9e,0xa0,0xa2,0xa4,0xa6,0xa8,0xaa,0xac,0
       xae,0xb0,0xb2,0xb4,0xb6,0xb8,0xba,0xbc,0xbe,0xc0,0xc2,0xc4,0xc6,0
       xc8,0xca,0xcc,0xce,0xd0,0xd2,0xd4,0xd6,0xd8,0xda,0xdc,0xde,0xe0,0
       xe2,0xe4,0xe6,0xe8,0xea,0xec,0xee,0xf0,0xf2,0xf4,0xf6,0xf8,0xfa,0
       xfc,0xfe,0x1b,0x19,0x1f,0x1d,0x13,0x11,0x17,0x15,0x0b,0x09,0x0f,0
       x0d,0x03,0x01,0x07,0x05,0x3b,0x39,0x3f,0x3d,0x33,0x31,0x37,0x35,0
       x2b,0x29,0x2f,0x2d,0x23,0x21,0x27,0x25,0x5b,0x59,0x5f,0x5d,0x53,0
       x51,0x57,0x55,0x4b,0x49,0x4f,0x4d,0x43,0x41,0x47,0x45,0x7b,0x79,0
       x7f,0x7d,0x73,0x71,0x77,0x75,0x6b,0x69,0x6f,0x6d,0x63,0x61,0x67,0
       x65,0x9b,0x99,0x9f,0x9d,0x93,0x91,0x97,0x95,0x8b,0x89,0x8f,0x8d,0
       x83,0x81,0x87,0x85,0xbb,0xb9,0xbf,0xbd,0xb3,0xb1,0xb7,0xb5,0xab,0
       xa9,0xaf,0xad,0xa3,0xa1,0xa7,0xa5,0xdb,0xd9,0xdf,0xdd,0xd3,0xd1,0
       xd7,0xd5,0xcb,0xc9,0xcf,0xcd,0xc3,0xc1,0xc7,0xc5,0xfb,0xf9,0xff,0
       xfd,0xf3,0xf1,0xf7,0xf5,0xeb,0xe9,0xef,0xed,0xe3,0xe1,0xe7,0xe5};
21
22 const int mul3[256] = {0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0
       x1b,0x1e,0x1d,0x14,0x17,0x12,0x11,0x30,0x33,0x36,0x35,0x3c,0x3f,0
       x3a,0x39,0x28,0x2b,0x2e,0x2d,0x24,0x27,0x22,0x21,0x60,0x63,0x66,0
       x65,0x6c,0x6f,0x6a,0x69,0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,0
       x50,0x53,0x56,0x55,0x5c,0x5f,0x5a,0x59,0x48,0x4b,0x4e,0x4d,0x44,0
       x47,0x42,0x41,0xc0,0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,0xd8,0xdb,0
       xde,0xdd,0xd4,0xd7,0xd2,0xd1,0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0
       xf9,0xe8,0xeb,0xee,0xed,0xe4,0xe7,0xe2,0xe1,0xa0,0xa3,0xa6,0xa5,0
       xac,0xaf,0xaa,0xa9,0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,0x90,0
       x93,0x96,0x95,0x9c,0x9f,0x9a,0x99,0x88,0x8b,0x8e,0x8d,0x84,0x87,0
       x82,0x81,0x9b,0x98,0x9d,0x9e,0x97,0x94,0x91,0x92,0x83,0x80,0x85,0
       x86,0x8f,0x8c,0x89,0x8a,0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,0
       xb3,0xb0,0xb5,0xb6,0xbf,0xbc,0xb9,0xba,0xfb,0xf8,0xfd,0xfe,0xf7,0
       xf4,0xf1,0xf2,0xe3,0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,0xcb,0xc8,0
       xcd,0xce,0xc7,0xc4,0xc1,0xc2,0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0
       xda,0x5b,0x58,0x5d,0x5e,0x57,0x54,0x51,0x52,0x43,0x40,0x45,0x46,0
       x4f,0x4c,0x49,0x4a,0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,0x73,0
       x70,0x75,0x76,0x7f,0x7c,0x79,0x7a,0x3b,0x38,0x3d,0x3e,0x37,0x34,0
       x31,0x32,0x23,0x20,0x25,0x26,0x2f,0x2c,0x29,0x2a,0x0b,0x08,0x0d,0
       x0e,0x07,0x04,0x01,0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a};
23
24 const int s_box[256] =
25  {0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7
       ,0xab,0x76,0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2
       ,0xaf,0x9c,0xa4,0x72,0xc0,0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc
       ,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,0x04,0xc7,0x23,0xc3,0x18
       ,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,0x09,0x83
       ,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f
       ,0x84,0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39
       ,0x4a,0x4c,0x58,0xcf,0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45
       ,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,0x51,0xa3,0x40,0x8f,0x92,0x9d
       ,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,0xcd,0x0c,0x13
       ,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73
```

```
      ,0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde
      ,0x5e,0x0b,0xdb,0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3
      ,0xac,0x62,0x91,0x95,0xe4,0x79,0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e
      ,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,0xba,0x78,0x25,0x2e
      ,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,0x70
      ,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1
      ,0x1d,0x9e,0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87
      ,0xe9,0xce,0x55,0x28,0xdf,0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68
      ,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16};
26
27 const int rcon[255] = {
28     0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36,0x6c,0xd8,0
           xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,0x6a,0xd4,0xb3
           ,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,0x61,0xc2,0
           x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a,0x74,0xe8,0xcb
           ,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36,0x6c,0
           xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,0x6a,0xd4
           ,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,0x61,0
           xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a,0x74,0xe8
           ,0xcb,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36,0
           x6c,0xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,0x6a
           ,0xd4,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,0
           x61,0xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a,0x74
           ,0xe8,0xcb,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0
           x36,0x6c,0xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35
           ,0x6a,0xd4,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0
           xbd,0x61,0xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a
           ,0x74,0xe8,0xcb,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0
           x1b,0x36,0x6c,0xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97
           ,0x35,0x6a,0xd4,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0
           xd3,0xbd,0x61,0xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d
           ,0x3a,0x74,0xe8,0xcb};
29
30 unsigned char key[16] = {'0x8f','0xaf','0x7a','0xf6','0xf8','0xaa'
       ,'0xe7','0x8a','0x1c','0xcf','0x01','0x42','0x9d','0x11','
     0xe2','0x91'};    // input key takes 4 words (=16 bytes)
31 words (=16 bytes)
32
33 unsigned char key_w[176];                          // later expanded key
34 unsigned char cntblock[16] = {'0x00','0x00','0x00','0x00','0x00','0x00'
       ,'0x00','0x00','0x00','0x00','0x00','0x00','0x00','0x00','0
     x00'};
35 unsigned char state[4][4];                          // temporary state-
     matrix which stores intermediate data
36 unsigned char r[4][4];                            // temporary matrix for
       mixcolumn function
37
38 void KeyExpansion();
39 void Cipher();
40 unsigned char ByteSub(unsigned char tw);
41 void AddRoundKey(int r);
42 void ShiftRows();
43 void SubBytes();
44 void MixColumns();
```

```
45 int ConvArrnAdd(int c);
46
47 int main()
48 {
49
50     KeyExpansion();
51     int i = 0;
52     int a=0, b=0, c=0;
53     int utdata;
54     bool done = false;
55
56     clock_t start = clock();
57
58     ofstream utData;
59     utData.open("aesutdata.bin", ios::out | ios::app | ios::binary);
60
61     // 250*1*1*1*(4)     = 1000
62     // 250*250*4*1*(4)     = 1000000
63     // 250*250*40*1*(4)    = 10000000
64     // 250*250*250*16*(4) = 1000000000
65
66     while(!done) {
67
68       for(int i = 0; i<16;i++) {
69         cntblock[12] = cntblock[12] + 1;
70         for(int i = 0; i<250;i++) {
71           cntblock[13] = cntblock[13] + 1;
72           for(int i = 0; i<250;i++) {
73             cntblock[14] = cntblock[14] + 1;
74             for(int j = 0; j<250; j++) {
75               cntblock[15] = cntblock[15] + 1;
76               Cipher();
77
78               for(int i=0;i<4;i++) {
79                 for(int j=0;j<4;j++) {
80                   a++;
81                   utData.write(reinterpret_cast<char*>(&state[j][i]),
                         sizeof(state[j][i]));
82                 }
83               }
84             }
85           }
86         }
87       }
88
89     done = true;
90     }
91
92     utData.close();
93
94     cout<<"Run time: "<<( ( clock() - start ) / (double)CLOCKS_PER_SEC
         ) <<" seconds."<<endl;
95     cout<<endl<<endl<<"Number of generated integers: "<<a/4<<endl;
96     cout<<"Last generated integer: ";
```

```
97      cout<<(unsigned int)state[0][3]<<(unsigned int)state[1][3]<<(
            unsigned int)state[2][3]<<(unsigned int)state[3][3]<<endl;
98
99
100     return 0;
101 }
102
103 /*
104 * This function takes the key and stretches
105 * out the key out to 10 new keys that will
106 * be used in each specific round later on.
107 */
108 void KeyExpansion() {
109
110   unsigned char tempkey[16];              // some temporary space for
        temporary keys
111                             // The first round key is the key itself
112   int i = 0;
113     for(i = 0;i < 4;i++)
114     {
115         key_w[i*4]=key[i*4];
116         key_w[i*4+1]=key[i*4+1];
117         key_w[i*4+2]=key[i*4+2];
118         key_w[i*4+3]=key[i*4+3];
119     }
120
121   for(;i < 44; i++) {
122     for(int j = 0; j < 4; j++) {          // fill temporary word
123       tempkey[j] = key_w[(i-1)*4 + j];
124     }
125     if(i % 4 == 0) {                 // if the current word is a word that
           must rotate and stuff, then..
126
127       unsigned char t = tempkey[0];       // rotate..
128       tempkey[0] = tempkey[1];
129       tempkey[1] = tempkey[2];
130       tempkey[2] = tempkey[3];
131       tempkey[3] = t;
132
133                             //..and substitute
134       tempkey[0] = s_box[tempkey[0]];
135       tempkey[1] = s_box[tempkey[1]];
136       tempkey[2] = s_box[tempkey[2]];
137       tempkey[3] = s_box[tempkey[3]];
138
139       tempkey[0] =  tempkey[0] ^ rcon[i/4]; // XOR first byte with part
              of the rcon matrix
140
141     }
142
143     key_w[i*4+0] = key_w[(i-4)*4+0] ^ tempkey[0]; // Fill key_w with
          all new subkeys!
144         key_w[i*4+1] = key_w[(i-4)*4+1] ^ tempkey[1];
145         key_w[i*4+2] = key_w[(i-4)*4+2] ^ tempkey[2];
```

```
146         key_w[i*4+3] = key_w[(i-4)*4+3] ^ tempkey[3];
147
148   }
149
150 }
151
152 /*
153 * This is the core scheme for the "encryption".
154 */
155 void Cipher() {
156
157   int i,j;                      // put plaintext block into state matrix
158   for(i = 0; i < 4;i++) {
159     for(j = 0; j < 4; j++) {
160       state[j][i] = key[4*i+j];
161     }
162   }
163
164   int r = 0;                                // get roundkey
165   AddRoundKey(r);
166
167   for(r=1; r<10; r++) {                       // get started with "
          encryption"!
168     SubBytes();
169     ShiftRows();
170     MixColumns();
171     AddRoundKey(r);
172   }
173
174   SubBytes();                               // last round!
175   ShiftRows();
176   AddRoundKey(10);
177
178 }
179 /*
180 * This function simply shifts i'th row i steps to the left.
181 * Hardcoded..
182 */
183 void ShiftRows() {
184
185   unsigned trow;
186   trow=state[1][0];
187
188     state[1][0]=state[1][1];
189     state[1][1]=state[1][2];
190     state[1][2]=state[1][3];
191     state[1][3]=trow;
192
193     trow=state[2][0];
194     state[2][0]=state[2][2];
195     state[2][2]=trow;
196
197     trow=state[2][1];
198     state[2][1]=state[2][3];
```

```
199       state[2][3]=trow;
200
201       trow=state[3][0];
202       state[3][0]=state[3][3];
203       state[3][3]=state[3][2];
204       state[3][2]=state[3][1];
205       state[3][1]=trow;
206
207
208  }
209  /*
210  * This function just substitutes the state matrix elements according
211  * to the S-box.
212  */
213  void SubBytes() {
214
215    unsigned char t2;
216
217    for(int i = 0; i < 4;i++) {
218      for(int j = 0; j < 4; j++) {
219        t2 = state[j][i];
220        state[j][i] = s_box[t2];
221      }
222    }
223  }
224  /*
225  * Used the hardcoded multiplication boxes given by Wikipedia,
226  * http://en.wikipedia.org/wiki/Rijndael_mix_columns. T
227  * his function operates on each column of the state matrix
228  * and do some matrix multiplication in the GF(2^8) field.
229  */
230  void MixColumns() {
231    int a,b,c,d,e1,e2,e3,e4;
232
233    for(int i = 0;i<4;i++) {                    // for every column..
234
235      a = mul2[state[0][i]];                    // do the multiplications
           ,
236      b = mul3[state[1][i]];                    // by use the hardcoded
237      c = state[2][i];                        // answers in the mul-boxes.
238      d = state[3][i];
239      e1 = a^b^c^d;
240
241      a = state[0][i];
242      b = mul2[state[1][i]];
243      c = mul3[state[2][i]];
244      d = state[3][i];
245      e2 = a^b^c^d;
246
247      a = state[0][i];
248      b = state[1][i];
249      c = mul2[state[2][i]];
250      d = mul3[state[3][i]];
251      e3 = a^b^c^d;
```

```
252
253     a = mul3[state[0][i]];
254     b = state[1][i];
255     c = state[2][i];
256     d = mul2[state[3][i]];
257     e4 = a^b^c^d;
258
259     r[0][i] = e1;                              // save temporary to r[][]
260     r[1][i] = e2;
261     r[2][i] = e3;
262     r[3][i] = e4;
263   }
264
265     for(int q = 0;q<4;q++)                          // finally copy r -->
          state
266     {
267       for(int p = 0;p<4;p++)
268       {
269         state[p][q]=r[p][q];
270       }
271     }
272
273 }
274
275
276 /*
277 * This function XOR's the state matrix with the key
278 * produced from the keyexpansion function.
279 */
280 void AddRoundKey(int r) {
281   int i,j;
282     for(i = 0;i < 4;i++)
283     {
284         for(j = 0;j < 4;j++)
285         {
286             state[j][i] ^= key_w[r * 4 * 4 + i * 4 + j];
287         }
288     }
289
290 }
```

.4 Referensimplementation med rand()

```
1 /*
2 * Hannes Salin, CDATE3. 07-06-2011
3 * Using rand() to get some pseudorandom
4 * integers to use as reference during
5 * the statistical testings.
6 */
7 #include <iostream>
8 #include <fstream>
```

```
 9 #include <stdlib.h>
10 #include <time.h>
11
12 using namespace std;
13
14 int main()
15 {
16
17   int r,i;
18   srand(1);
19
20   clock_t start = clock();
21
22   ofstream utData;
23   utData.open("randutdata.bin", ios::out | ios::app | ios::binary);
24
25   for(i = 0; i<1000; i++) {
26     r = rand();
27     utData.write(reinterpret_cast<char*>(&r),sizeof(r));
28   }
29
30   utData.close();
31
32   cout<<"length: "<<i<<endl;
33   cout<<"Run time: "<<( ( clock() - start ) / (double)CLOCKS_PER_SEC )
       <<" seconds."<<endl;
34   cout<<"Last generated integer: "<<r<<endl;
35
36   return 0;
37 }
```