Haaga-Helia
ammattikorkeakoulu Oy

Kellon toistuvaan lukuun perustuva satunnaisbittigeneraattori -ohjelma

Jari Kuivaniemi

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2016

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

22.5.2016

Tekijä(t)
Jari Kuivaniemi

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko	**Sivu- ja liitesivumäärä**
Kellon toistuvaan lukuun perustuva satunnaislukugeneraattori	27 + 29

Opinnäytetyön otsikko englanniksi
Random generator based on repeated reading of clock

Opinnäyte esittelee kellon vaihteluihin perustuvan satunnaislukugeneraattorin, Ressun. Siinä kerrotaan mitä nämä vaihtelut ovat. Lisäksi käydään läpi ressun toiminnot, jaetaan se palasiin ja esitellään niiden toimintaa. Lopuksi tutkitaan hiukan ongelmia, ja esitellään tilastollisia testejä satunnaisuudelle.

Ressu koostuu kahdesta toistorakenteesta, joista ensimmäisessä (~kellon luku ja bittien järjestely) puskuriin luetaan täyteen kellon merkkejä, ja vaihdetaan puskurin ylin bitti alas ja siirretään alimpia bittejä 1 bitti ylöspäin. Toisessa toistorakenteessa (~puskurin merkkien sekoitus) vaihdetaan jokainen puskurin merkki toisen puskurin merkin kanssa. Näitä kahta toistorakennetta toistetaan b kertaa. Ensimmäinen toistorakenne takaa että kellon alin bitti koskee kaikkia puskurin bittejä ja toinen kierros takaa että vierekkäin luetut bitit ovat eri paikoissa puskurissa ja että jokainen kellobitti vaikuttaa ~jokaiseen puskurin bittiin.

Työn aikana listasin mahdollisia ongelmia: kello ainoa satunnaisuudenlähde, muuteltu kellofunktio, käsitelty kellojono muodostaa hyökkääjän haluaman puskurin, hyökkääjä arvaa kellojonon, hyökkääjä arvaa puskurin, laitteistoerot, laitteiston kuormitus, kello ei ole riittävän tarkka, kello on liian säännöllinen, kellon sarjat ovat liian pitkiä, bittigeneraattori on tunnistettavissa, bittigeneraattori ei tuota uniikkia tietoa.

Ongelmat, joissa kellosarjaa on muuteltu ei katsota olennaiseksi, koska jos hyökkääjä voi muokata kellosarjaa, hän voi tehdä mitä tahansa muutakin, esimerkiksi lähettää generaattorin tuottaman puskurin itselleen.

Satunnaislukugeneraattorin arvaamiseen generaattori ei anna mitään vihjeitä (jos b on riittävän suuri), joten en pidä sitäkään olennaisena.

Ongelma jossa bittigeneraattori ei tuota uniikkia tietoa, ei johdu generaattorin rakenteesta, vaan ajatuksesta että jos generaattori tuottaa samanlaisia puskureita, kellosta luettavat bitit eivät riitä koko puskurin sisällöksi. Puskuriin saadaan lisää bittejä kasvattamalla kutsun b muuttujan arvoa.

Työ esittelee tilastollisia testejä, joilla voidaan testata lukujonon satunnaisuutta, testeistä mukana ovat monobit, poker 2, poker 4, runs ja runs8.

Raportin liitteenä on työn aikana tehdyt ohjelmat kellon vaihtelujen havainnollistamiseen, tilastojen laskemiseen,varsinainen satunnaislukugeneraattori, ja kutsuvat rutiinit, jos haluat sen heti käyttöön (tässä b on toivoakseni ylimitoitettu niin ettei pitäisi tulla ongelmia). Raportissa suositellaan kuitenkin että tätä ei käytetä ainoana generaattorina.

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

22.5.2016

Asiasanat
Satunnaisbittigeneneraattori, variaatiot kellon toistuvassa luvussa, avaimeton satunnaisbittigeneraattori

Haaga-Helia
University of Applied Sciences Ltd.

Abstract

22 May 2016

Author(s)
Jari Kuivaniemi

Degree programme
Business Information Technology

Report/thesis title	**Number of pages and appendix pages**
Random generator based on repeated reading of clock	27 + 29

This thesis presents random number bit stream generator based on variances reading the clock, Ressu. The thesis shows what these variances are, shows how Ressu works, divides it into pieces, and analyses the parts. The last part of the thesis lists the problems discovered during writing and shows some statistical tests for randomness.

Ressu contains two repeat structures. The first (~reading clock and moving bits) structure fills the buffer with clock bytes and moves the highest bit to lowest and the lowest bits one step up. The second structure (~shuffling buffer characters) swaps every character in buffer with random character in buffer. These repeat structures are repeated 8*b times. The first repeat structure guarantees that the lowest bit in the clock touches every bit of the buffer. The second structure guarantees that the bits read next to each other are in different parts of the buffer, and that every bit of the clock affects ~every bit of the buffer.

While working on the thesis I listed some possible problems in the generator: clock is the only source for bits, corrupted clock function, corrupt clock gives buffer wanted by attacker, attacker guesses clock stream, attacker guesses the buffer, hardware differences, hardware load, clock not accurate enough, clock is too regular, clock series are too long, attacker is able to identify generator, or generator does not provide unique buffers.

Problems where clock is corrupted are not considered relevant, as if an attacker can change clock stream he can send buffer containing collected random bits to himself.

The generator does not give any hints to guessing clock stream or clock series (if b is large enough), this is not considered relevant either.

The problem that random bit generator does not give unique buffers is not about the structure of the generator, but it is about the thought that if the generator returns duplicate buffers, we don't have enough randomness in clock stream. We get more random clock bits to buffer by increasing b.

The thesis gives statistical tests that can be used to test randomness of bit stream. Included are monobit-, poker 2-, poker 4-, runs- and runs8 -tests.

The programs written during the work are attached to the thesis. There are programs to show these variances, programs for statistical tests, and for finding b values that return unique buffers. The generator itself is in attachment 2. Also, there is a program to use the program right now, it has hopefully conservative enough buffer size and b.

Keywords
random bit generator, variances reading clock repeatedly, keyless random stream generator

Sisällys

1 Johdanto 1
2 Satunnaislukugeneraattorit 2
2.1 Satunnaisuuden lähteet 2
2.2 Satunnaisuuden käyttö 6
3 Ressu-satunnaislukugeneraattori 7
3.1 Vaihtelut kelloa luettaessa 7
3.2 Ohjelman toiminta 10
3.3 Kellomerkit lähellä puskurin kokoa 11
3.4 Puskurin pituus sama kuin kellojonon pituus 12
3.5 Ohjelman palaset 14
3.6 Osatoiminnot 16
3.7 Tuloksen satunnaisuuden mittaaminen 17
3.8 Generaattorin mahdolliset ongelmat 20
3.9 Vielä hiukan b:stä 22
4 Pohdinta 25
Lähteet 26
Liitteet 29
Liite 1: Makefile 30
Liite 2: ressu.c – Ressu generaattori 32
Liite 3: rn1.c – Kello 8 bittisenä merkkijonona 33
Liite 4: rn2.c – Kello 8 bittisinä sarjoina 34
Liite 5: rn3.c – Kello 4 bittisenä merkkijonona 35
Liite 6: rn4.c – Kello 4 bittisinä sarjoina 36
Liite 7: rn5.c – Tiivistetty kellosarjojen tulostus 37
Liite 8: rn22.c – Eripituisten merkkiketjujen lukumäärä 38
Liite 9: rn23.c – Ensimmäisen uniikkeja puskureita tuottavan b:n haku 39
Liite 10: rn30.c – Monobit testi 43
Liite 11: rn31.c – Poker-2 testi 45
Liite 12: rn32.c – Poker-4 testi 47
Liite 13: rn33.c – Runs testi 49
Liite 14: rn34.c – Runs-8 testi 52
Liite 15: rn40.c – Kaikki satunnaisuus testit 54
Liite 16: rn99.c – Mallitoteutus 56

1 Johdanto

Opinnäytetyöni kohteena on satunnaislukugeneraattori, Ressu, joka perustuu vaihteluihin luettaessa kelloa toistuvasti. Opinnäytteen tavoitteena on selittää tai havainnollistaa näitä vaihteluita, näyttää tarkemmin generaattorin toimintaa ja tutkia generaattorin ongelmia.

Ressu on syntynyt osana Terttu-ohjelmistoani, sen istuntoavaimen muodostusta varten. Terttu on kaupan järjestelmän runko, jossa asiakas saa päättää mitä kenttiä lomakkeilla on ja terttuun on ohjelmoitu, kuinka tietovirrat toimivat (Kuivaniemi 2016). Ohjelman viimeistä versiota voi ajaa tästä www.moijari.com:5002.

Opinnäytetyön tavoitteet on kuvata vaihtelut, joihin Ressu perustuu, Kuvata ressun rakennetta ja selittää sen toimintaa ja toisaalta listata sen ongelmia.

Johdannon jälkeen on kappale 2 joka kertoo yleistä tietoa satunnaisgeneraattoreista. Kappale 3 kertoo Ressusta, katsellaan kellon vaihteluita, selitetään ohjelman suoritusta, katsellaan sen osia ja listataan sen ongelmia. Liitteissä on generaattorin lähdekoodi, työn aikana luodut lähdekoodit vaihtelujen havainnollistamiseen, kellosarjojen tutkimiseen ja b:n arvon etsimiseen.

2 Satunnaislukugeneraattorit

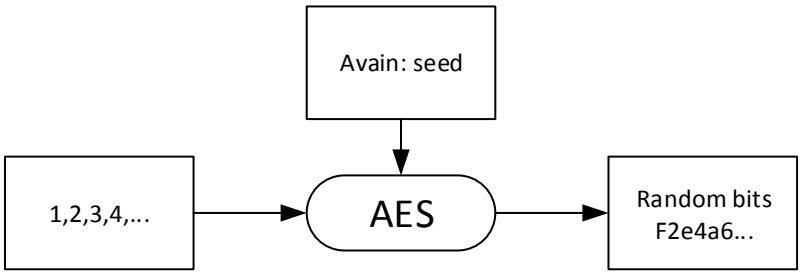
2.1 Satunnaisuuden lähteet

Perinteisiä satunnaislukujen arvontatapoja ovat nopanheitto, kolikon heitto ja sekoitetusta korttipakasta kortin vetäminen. On kuitenkin selvää, että nämä mekaaniset arvontatavat eivät ole mahdollisia, jos satunnaislukuja tarvitaan paljon.

Satunnaislukuja tarvitaan ainakin (Knuth 1998, 1) simulaatioissa ympäristön luomisessa, osajoukon valintaan tilastoinnissa, numeeriseen analyysiin, ohjelmoinnissa, päätöksen teossa, viihteessä ja salakirjoituksessa.

Näille eri tarpeille on keksitty aikojen kuluessa erilaisia menetelmiä luoda satunnaislukuja. Yksinkertaisimmat luovat satunnaislukuja jonkun siemeneen perustuvan kaavan avulla. Näitä ovat Linear Congruential generaattorit esimerkiksi lehmer generaattori (Payne 1969), Linear Feedback Shift rekisterit (Berkeley University 1). Näissä tarvitaan yleensä siemenarvo, sen voi päätellä niiden jaksoa tarkkailemalla.

Toinen tapa luoda satunnaislukuja on käyttää lohkosalakirjoitusta esimerkiksi Data Encryption Standard DES (NIST 1999), Advanced encryption standard AES (ennen standardiksi valintaa nimi oli Rijndael) (Daemen & Rijmen 1998). Siinä avaimena käytetään kehitettyä siementä, ja salakirjoitettavana blokkina käytetään lukusarjan lukuja kasvatten aina lukua yhdellä (esimerkiksi 1,2,3,4,5,6,7,8,9,10) (Kuva 1). Tuloksena saadaan sarja satunnaislukuja. Tämän tyylisissä generaattoreissa siemen pitää tietysti generoida jollain muulla tavalla.

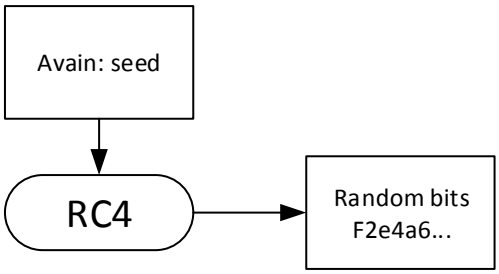


Kuva 1: Satunnaisbittejä lohkosalakirjoitusjärjestelmällä

DES ei ehkä nykyään ole toimiva valinta, sen lyhyen lohkon ja salasanan lyhyyden vuoksi. Lohkon lyhyyttä voidaan käyttää hyväksi hyökkäyksessä ja esimerkiksi luoda tiedosto kaikilla salasanoilla tehdyistä ykköslohkoista (katso edellinen lukusarja). Sitten kun

ensimmäinen satunnaisbittisarja saadaan järjestelmältä, voidaan tiedostosta katsoa millä salasanalla se on luotu ja päätellä salasanan perusteella muut lohkot.

Satunnaisuutta voidaan saada stream cipher funktioista kuten RC4 (Kuva 2), joka alunperin oli salainen, ja algoritmin nähdäkseen piti kirjoittaa julkaisukieltosopimus(Rivest RC4). Algoritmi vuosi kuitenkin ja se oli aikanaan osana muunmuassa WEP salausta. Nyttemmin siitä on löydetty haavoittuvuuksia. RC4 on myös avaimeen perustuva bittigeneraattori.



Kuva 2: Satunnaisbittejä stream cipher funktiosta

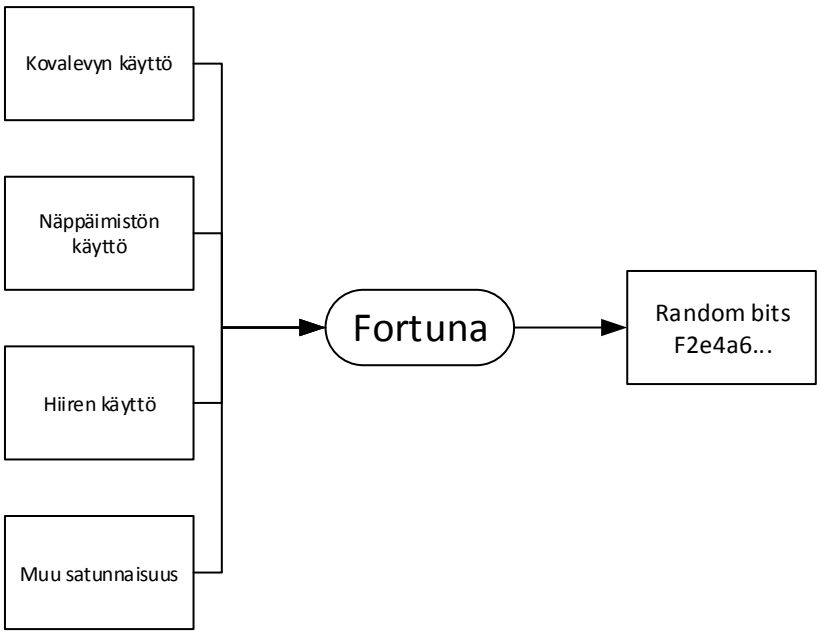
Samaan tapaan voidaan saada satunnaislukuja yksisuuntaisesta hash funktiosta MD5 (Rivest 1992), SHS (SHA-1-512) (Nist 2012).Tällä kertaa siemen ja lukusarjan tämänhetkinen numero syötetään tiivistefunktiolle ja tuloksena saadaan tiiviste, joka koostuu satunnaisbiteistä (Kuva 3). Numeron kasvatuksen jälkeen suoritetaan hash funktio uudestaan ja saadaan uudet 512-1024 merkkiä.

Kuva 3: Satunnaisbittejä tiivistefunktiosta

Tiivisteitä ja lohkosalakirjoitusta käyttävät myös Distilling randomness (Scneier, 426) Yarrow (Schneier 1999, 426) ja Fortuna (Schneier 2010). Nämä satunnaislukugeneraattorit summaavat satunnaisuutta useammasta lähteestä, ja antavat sitä kutsujalle. Kaikissa näissä satunnaislukusarja luodaan siten että lähteeseen kerätystä satunnaisuudesta luodaan siemen, ja edellämainutulla numerosarjamenetelmällä luodaan varsinainen satunnaislukusarja.

Distilling randomness perustuu yhteen satunnaisuuslähteeseen, johon kerätään satunnaisuutta ajamalla churnrand(char *randevent,int randlen) rutiinia kaikelle satunnaisuutta sisältävälle. Sitten satunnaisuutta luetaan genrand(char *buf, int buflen)

rutiinilla. Satunnaisuutta summataan churnrand:illa yhteen pooliin, ja sitä luetaan genrandilla samaisesta poolista. Yarrow:ssa on sama idea kahdella lähdepuskurilla, Fortuna:ssa lähteitä on 32 (Kuva 4). Näissä kolmessa yhteisenä tekijänä on se että käyttäjä antaa generaattorille satunnaisuutta funktiokutsulla. Ressussa tälläistä käyttäjän kutsuja tarvitsemaa rutiinia ei ole.



Kuva 4: Satunnaisbittejä Fortuna generaattorilla

Lisäksi satunnaisuutta varten on kehitetty erilaisia piirejä, ja nyttemmin koneiden keskusyksiköissä on usein ominaisuutena satunnaislukugeneraattori. Intel:in generaattorin nimi on RDRAND (Intel 2012). Näissä piireissä satunnaisuutta saadaan jostain fyysisestä ilmiöstä, kuten eri nopeuksisista oskillaattoreista tai komponenttien toiminnan vaihteluista.

Kelloon tai oikeastaan ajastimeen ja kasvatettavaan muuttujaan perustuu Bruce Scheierin kirjassaan (Schneier 1996, 424) mainitsema “Truly randon number generator”. Se perustuu hälytyksen asettamiseen ja sen aikana muuttujan kasvattamiseen. Kun hälytys tulee luetaan kasvatetun muuttujan arvo ja käytetään siitä kaksi alinta bittiä satunnaislukuun. Sitten luodaan uusi hälytys ja toistetaan toimintaam kunnes bittejä on kerätty haluttu määrä.

Edellisistä rc4 ja Scheierin Truly random number generator muistuttavat ehkä eniten ressua. Toisaalta Ressua voi sanoa avaimettomaksi satunnaislukugeneraattoriksi, jolloin sen voisi sanoa muistuttavan Fortunaa. Tosin Fortunassakin asiakkaan pitää toimittaa satunnaisuutta järjestelmälle (vertaa avain).

Satunnaislukugeneraattoreissa satunnaisuutta muodostetaan kovalevyn, näppäimistön ja hiiren lukuajasta, luvun kestoajasta ja luvun loppumisajasta (Schneier 1996, 426). Lisäksi generaattorin lähteeksi voidaan antaa myös näppäimistön kirjain ja hiiren koordinaatit. Turvallisuuden lisäämiseksi näppäimistön kirjaimen voi ennen lähettämistä lisätä johonkin toiseen satunnaiseen lähteeseen. Tietenkin yhtenä lähteenä voi olla vielä kovoon perustuva generaattori, joita alkaa olla monessa uudessa keskusyksikössä. Aiemmin en ole nähnyt lähteeksi ehdotetun funktion alkuaikaa, suoritusaikaa tai loppuaikaa. Alkuaika ja loppuaika korreloivat tietysti keskenään mutta se ei aiheuta ongelmia.

Satunnaisuutta voi kerätä eri sovelluksilla kamerasta. Lava lampuista (Wired 2003) satunnaisuutensa sai lavarand.sgi.com, joka on avattu uudestaan osoitteessa lavarand.org (Silicon Graphics 1997). Uusi versio tosin saa satunnaisuutensa kameran kuvakohinasta, ja kameran linssinsuoja voi olla paikallaan, lavalampuilla sillä ei ole enää mitään tekemistä.

Kamerasta voi myös kerätä satunnaisbittejä ottamalla kaksi kuvaa, ja käyttämällä eri värien erotuksen alimmat bitit satunnaisuuteen. Tässä tulee mukaan myös kameran kuvakohina, mutta siitähän ei ole haittaa, pelkästään hyötyä.

Toisaalta jos palvelimesi levyllä on tiedostoja, esimerkiksi kirjoja, musiikkia tai elokuvia, myös niistä voi valita satunnaisen lohkon ja käyttää generaattoripinossa, tällöin hyökkääjällä pitäisi olla käytössään myös nuo tiedostot.

Ohjelmassa voi myös olla osio, jossa avainten muodostuksessa käyttäjä kirjoittaa itse näppäimistöltä satunnaisia merkkejä/lauseita. Näin toimii PGP. (Zimmermann 1991)

Jos aiemmin tarvitsit satunnaisuutta, voit valita satunnaisen sivun RAND Corporation:in 1955 julkaisemasta kirjasta, A Million Random Digits with 100,000 Normal Deviates*, ja lukea satunnaisluku jostakin kohdasta aukeamalta.* Kirjasta on tehty uusi painos 2001.

Nyttemmin satunnaisluvut ja lottonumerot saa suositusta www.random.org palvelusta. Se tuntee myös eri alueiden rahapelien säännöt ja muun muassa Suomen Lotto-arvontaan löytää numerot.

Satunnaisuuden keräämisen yksi ongelmakohta on laitteet, joissa ei ole näppäimistöä, hiirtä ja kovalevyä, eikä käytettävissä ole tallennetta aiemmin kerätystä satunnaisuudesta. Tähän ongelmaan tämä generaattori voisi olla ratkaisu. Kun satunnaislukulähdettä

käytetään generaattorin avaimen uudelleen muodostukseen, täytetään lähde ennen käyttöä tämän generaattorin muodostamilla biteillä. Jos esimerkiksi altaan koko on 128 bittiä ja muista lähteistä on saatu 64 bittiä, lisätään vielä altaaseen 64 bittiä tästä generaattorista ennenkuin lähdettä käytetään avaimen muodostukseen.

2.2 Satunnaisuuden käyttö

Salakirjoituksessa satunnaislukuja käytetään lohkosalakirjoituksen avaimena. Menetelmästä riippuen avaimen pituus voi olla jotain 56 bitistä (DES) ylöspäin. Pidemmät avaimet vaikeuttavat avaimen arvaamista (brute force), ja itseasiassa DES:ssä avaimen pituus on aivan liian lyhyt. Sen ikää on kuitenkin lisätty lisäämällä DES:n salakirjoitusvaihe 3 kertaiseksi, jolloin käytetään kolmella salakirjoituskerralla eri avainta, tai kahteen kertaan samaa avainta ja kolmatta eri avainta. Tätä uusittua DES:iä kutsutaan Triple DES:iksi (NIST 2012).

Toinen salakirjoitusmenetelmä RSA (Rivest 1977), julkisavaimeen pohjautuva järjestelmä, käyttää satunnaisbittejä avaimen luonnissa ja itse viestien salaamisessa. Avaimet luodessa generaattorilta saadaan muuttujat P ja Q, joiden pohjalta saadaan D, E ja N(N=P*Q). D ja N muodostavat julkisen avaimen ja E ja N salaisen avaimen. Lisäksi satunnaislukuja tarvitaan kun salakirjoitetaan viesti. Viestin salakirjoitetuksessa käytetään satunnaislukugeneraattorin luomaa avainta ja lohkosalakirjoitusta. Lohkosalakirjoitusjärjestelmän avain salakirjoitetaan sitten käyttämällä julkisavainta D&N (tai salaista avainta E&N). Tämä siksi, että RSA:n toteutus perustuu potenssilaskuun (salakirjoitettu=viesti^D%N ja viesti=salakirjoitettu^E%N) ja on melko hidas suurille bittimäärille.

Myös stream cipher rutiinit käyttävät salaukseen avainta. RC4:ssä avaimen pituus voi olla 256 merkkiä, jollei sitä ole teknisesti rajoitettu.

Salakirjoitukseen liittymättöminä asioina satunnaisbittejä käytetään ainakin salt:ina käyttäjän salasanan salakirjoituksessa. Tällä pyritään ettei hyökkääjä voi tehdä kaikista erilaisista kryptatuista salasanoista tiedostoa, josta sitten haetaan salakirjoitetulla salasanalla selväkielinen salasana.

3 Ressu-satunnaislukugeneraattori

Tämä satunnaislukugeneraattori perustuu vaihteluihin kelloa luettaessa. Se on koodina lyhyt, nelisenkymmentä riviä(Liite 2). Kirjoitin sen marraskuussa 2013 ja sitä käytetään Tertun istuntoavaimia luotaessa. Terttu on ideoimani kaupan järjestelmä, jota olen tehnyt opintojen ohessa (Kuivaniemi 2016). Tertun idea on että asiakas tekee tarvittavat sovellukset määrittelemällä sovellusten otsake ja rivi -kentät ja tietovirrat. Terttu yrittää luoda automaation sovellusten välillä kenttien nimien samanlaisuuden perusteella, varsinaista sovelluskohtaista koodia ei kirjoiteta.

Oma kiinnostukseni aiheesta tulee tarpeesta saada tuo istuntoavaimen muodostus terttuun. Olen aiemmin ihmetellyt jonkin vuoden salakirjoitusta, mutta varsinainen mielenkiintoni on kohdistunut kaupan järjestelmiin.

Sen verran ressu on muuttunut että muuttuja f oli ennen e ja se nollattiin ensimmäisessä versiossa noiden kahden kiepukan välissä. Nyt sen nimi on f ja se nollataan vain kerran funktion alussa. Tämä varmistaa sen että kaikki kellomerkit vaikuttavat palautettavaan puskuriin. (kaikki kellomerkit lisätään muuttujaan f joka määrittelee minkä kellomerkin kanssa puskurin n:s merkki vaihdetaan). Toinen muutos ressussa on että aiemmassa versiossa kaikki gettimeofday() funktiokutsun bitit oli yhdistetty näihin puskuriin lisättäviin bitteihin (Kuivaniemi 2014) nyt käytetään vain alimmat 8 bittiä. Tämä uusi rakenne on selkeämpi, ja sen ei pitäisi vaikuttaa tulokseen, meillä on edelleen samat alimmat bitit, joissa on eniten liikettä.

Ressu on siinä mielessä ongelmallinen, että kutsu sisältää muuttujan b, joka määrittelee kuinka monta kierrosta tehdään yhtä puskurillista varten, ja alusta, missä generaattoria ajetaan vaikuttaa b:n arvoon. Helppo tapa olisi tietysti valita suuri b (esimerkiksi 1024), se takaisi että satunnaisuutta tulee riittävästi puskurin tayttämiseksi, mutta hitaassa koneessa satunnaispuskurin generoiminen veisi suhteettoman paljon aikaa (näyttää siltä että nopeammissa koneissa tarvitaan suurempia b:n arvoja). Toisaalta näyttää siltä että pitkillä puskureilla esimerkiksi 4096 merkkiä, riittää 1 kierros koko puskurin täyttämiseen. Toivottavasti löydän b:n arvon kaavan projektin puitteissa.

3.1 Vaihtelut kelloa luettaessa

Mitä nämä vaihtelut sitten ovat? Yksinkertaisimmassa mahdollisessa kellosarjassa kellon luku palauttaa aina seuraavan kello merkin, esimerkiksi 1,2,3,4,5,6,7,8,9,10, … Toisessa yksinkertaisessa sarjassa kello palauttaa aina kaksi kertaa saman kellomerkin,

esimerkiksi 1,1,2,2,3,3,4,4,5,5,6,6 … Sarjassa voi olla esimerkiksi ensimmäistä merkkiä seitsemän kappaletta ja sen jälkeen kahdeksan merkkiä toista merkkiä ja taas seitsemän merkkiä kolmatta merkkiä. Pointti on että jos sarja toistuu aina samanlaisena, sen päälle ei voi rakentaa satunnaislukugeneraattoria, tällöin on helppo huomata millainen kellon tuottama sarja on, ja arvata vain luku, josta sarja alkaa, montako merkkiä aloitusnumeroa on jäljellä ja millainen seuraava jono on.

Onneksi näitä vaihteluita kuitenkin on. Seuraavassa esimerkkisarjana hidas kone, jossa peräkkäiset jonot ovat 0-3 merkkiä pitkiä: (voit ajaa tätä ohjelmaa liitteellä rn1.c)

```
cb cc cc cd ce cf cf d0 d1 d2 d3 d3 d4 d5 d6 d6
d7 d8 d9 da da db dc dd dd de df e0 e1 e1 e2 e3
e4 e4 e5 e6 e7 e8 e8 e9 ea eb eb ec ed ee ef ef
f0 f1 f2 f2 f3 f4 f5 f6 f6 f7 f8 f9 f9 fa fb fc
fc fd fe ff 00 00 01 02 03 03 04 05 06 07 07 08
09 0a 0a 0b 0c 0d 0e 0e 0f 10 11 12 12 13 14 15
```

Kuva 5: liitteen rn1.c:n tuloste

Kuvassa 5 huomataan, että luvut ovat yhden ja kahden luvun sarjoissa ja että sarja ei ole säännöllinen. Eli jos hyökkääjä tekisi arvausohjelmaa kellon jonolle, hän käytännössä joutuisi arvaamaan kuinka monta kappaletta kutakin eri merkkiä jonossa on.

Ohjelmalla rn2.c jonojen väliin on lisätty rivin vaihto, joka helpottaa peräkkäisten jonojen vertailua. Ohjelman ajaminen jää kuitenkin lukijalle, koska tällä dokumentin kapeammalla rivillä listan selaaminen on hankalaa.

Seuraavassa kuvassa 6 on listattu vain 4 alilnta bittiä luvuista (liite rn3.c). Eli tuloste on samanlainen muuten rn1 tulosteen kanssa.

```
0c 0d 0e 0f 0f 00 01 02 02 03 04 05 06 06 07 08
09 0a 0a 0b 0c 0d 0d 0e 0f 00 01 01 02 03 04 04
05 06 07 07 08 09 0a 0b 0b 0c 0d 0e 0f 0f 00 01
02 02 03 04 05 05 06 07 08 09 09 0a 0b 0c 0d 0d
0e 0f 00 00 01 02 03 04 04 05 06 07 07 08 09 0a
0a 0b 0c 0d 0e 0e 0f 00 01 02 02 03 04 05 05 06
07 08 09 09 0a 0b 0c 0c 0d 0e 0f 00 00 01 02 03
03 04 05 06 07 07 08 09 0a 0a 0b 0c 0d 0e 0e 0f
```

Kuva 6: liitteen rn3.c tuloste

Kuvan 7 merkit käyttäytyvät samoin kun kuvan 3.1b lista, merkki kasvaa koko ajan yhdellä ja jokaista merkkiä on ~1 tai 2 kappaletta. Tulosteessa on edelleen neljä alinta bittiä merkeistä, ja peräkkäisten jonojen välillä on rivinvaihto, näin rivin pienin luku on ensimmäisenä rivillä ja suurin viimeisenä rivillä. Näin variaatiot on helpompi havaita, jos variaatioita ei olisi, kaikki rivit näyttäisivät samalta.

```
00 01 02 02 03 04 05 06 06 07 08 09 09 0a 0b 0c 0d 0d 0e 0f
00 00 01 02 03 04 04 05 06 07 07 08 09 0a 0b 0b 0c 0d 0e 0e 0f
00 01 02 02 03 04 05 05 06 07 08 09 09 0a 0b 0c 0c 0d 0e 0f
00 00 01 02 03 03 04 05 06 07 07 08 09 0a 0a 0b 0c 0d 0e 0e 0f
00 01 01 02 03 04 05 05 06 07 08 08 09 0a 0b 0c 0c 0d 0e 0f 0f
00 01 02 03 03 04 05 06 06 07 08 09 09 0a 0b 0c 0d 0d 0e 0f
00 01 01 02 03 04 04 05 06 07 08 08 09 0a 0b 0b 0c 0d 0e 0f 0f
00 01 02 02 03 04 05 06 06 07 08 09 09 0a 0b 0c 0d 0d 0e 0f
00 01 01 02 03 04 04 05 06 07 08 08 09 0a 0b 0b 0c 0d 0e 0f 0f
```

Kuva 7: liitteen rn4.c tuloste

Seuraavaksi on vuorossa nopeammille koneille tarkoitettu ohjelma rn5.c kuvassa 8. Sen tulosteessa yhtä samaa merkkiä sisältävää merkkijonoa merkataan x(y) tyylisesti, jossa x on samojen merkkien määrä ja y on merkin sisältö: Tässä mallissa perusjono on ~seitsemän nerkin pituinen ja muut pituudet ovat näitä variaatioita, joista olemme kiinnostuneita.

```
5(00) 4(01) 7(02) 7(03) 7(04) 7(05) 7(06) 7(07) 8(08) 6(09) 7(0a)
7(0b) 7(0c) 7(0d) 7(0e) 7(0f) 5(10) 7(11) 7(12) 7(13) 7(14) 7(15)
7(16) 7(17) 7(18) 7(19) 7(1a) 7(1b) 7(1c) 7(1d) 7(1e) 8(1f) 7(20)
7(21) 7(22) 7(23) 7(24) 7(25) 7(26) 7(27) 7(28) 7(29) 7(2a) 7(2b)
7(2c) 7(2d) 7(2e) 7(2f) 8(30) 7(31) 7(32) 7(33) 7(34) 7(35) 7(36)
7(37) 7(38) 7(39) 7(3a) 7(3b) 7(3c) 7(3d) 7(3e) 7(3f) 7(40) 7(41)
7(42) 7(43) 7(44) 8(45) 7(46) 7(47) 7(48) 7(49) 7(4a) 7(4b) 7(4c)
7(4d) 7(4e) 7(4f) 7(50) 7(51) 7(52) 7(53) 8(54) 7(55) 7(56) 7(57)
7(58) 7(59) 7(5a) 7(5b)
```

Kuva 8: liitteen rn4.c tuloste

Selvää on, että satunnaisuutta datassa on, mutta onko sitä riittävästi? Ongelmana tässä on tietysti päätellä, kuinka paljon satunnaisuutta tälläinen data sisältää. Ensimmäinen

ajatukseni oli että yhdessä merkin arvon vaihdoksessa olisi 1 bitti satunnaisuuttam ja toinen on että satunnaisuutta on riittävästi kun kasvatetaan b:tä kunnes ressu alkaa tuottamaan uniikkeja puskureita.

3.2 Ohjelman toiminta

Seuraavissa genbytes:in tulostamissa riveissä on tulostettuna puskurin sisältö. Ykkösellä alkavat rivit ovat kellobittien luvun jälkeen, ja kakkosella alkavat rivit ovat puskurin sekoituksen jälkeen. Ensimmäinen rivi on sen jälkeen kun nollilla täytetty puskuri on täytetty kellon merkeillä.

```
1:d3d7d9d9dadbdcdddddedfe0e1e1e2e3e4e5e6e6e7e8e9eaeaebecedeeeeeff0
```

Kuva 9: Puskuri ensimmäisen kellonlukukierroksen jälkeen

Huomaamme että ensimmäisen kellon luvun jälkeen puskurin sisältö on vielä ihan hyvin tunnistettavissa, rivi alkaa rivin pienimmällä numerolla (d3) ja kasvaa kohti puskurin loppua siten että viimeinen merkki on f0. Huomaa että vaikka puhunkin kellon alimman bitin koskettavan kaikkia puskurin bittejä, myös ainakin toiseksi alimmassa kellon biteissä on vaihtelua ja sekin koskettaa kaikkia kellon bittejä.

Seuraava rivi on tulostettu ensimmäisen sekoitusluupin suorituksen jälkeen. Sekoitusluuppi käy puskurin merkki merkiltä läpi, ja vaihtaa merkin satunnaisen puskurin merkin kanssa.

```
2:e6e2dceae3daefdde4edd9e8eef0dfecd7e1e5d9dbdddeeeebe7ead3e6e9e1e0
```

Kuva 10: Puskuri ensimmäisen merkkienvaihtelun jälkeen

Puskurillinen on vieläkin tunnistettavissa siinä mielessä että vaikkakin merkit on sekoitettu puskurin pienin merkki on d3 ja suurin merkki f0.

Jatkossa vielä vuorottelevat kellon luku ja sekoitus puskurit generaattorin yhdestä kierroksesta.

```
1:3738472ac7b4ddb8cadfb6d7dae6b7d0a5c8c0bfbab5b2cdc7dec7b4d9c6d6d7
2:d0c82ac7dfc63847d9c7daa5bfb8b6cdb2b5d737c7d6b4decae6ddc0b7d7bab4
1:a89c5a80b09d619ca09ca15e69667a837c71b47593b077a28aed9aa34c8c514c
2:b49ca35aa8a0937c71697a9c809ab08ab09c4c5e51614c77a28ceda19d836675
1:643657a5435234ecf7c4e321192c7b0e7d2585a2bde2b8cf673aff671e21ebc2
```

```
2:ff5257f7347de2eb19c23acfec7b642125674336a2b82c21bde31e85670ec4a5
1:f4a9a0e078ebd7c421916189cee1d05b50d59d70586f475d5be61e28ed38ac6d
2:e6d7a9a0ac91ede1d089e061eb58785bd55d5bf4479d70281e50216dce6fc438
1:c2bc47544f34c3dab809dadecaadeea98b9b97cbad1fc5751a876af3b4f4a25c
2:dac31adeeeb85c543409bcdab44fc2adc5adf36a8b1f47ca97a287f4759ba9cb
1:b89725afcf62acbd7e056ead70849f409746f9cb081eafb70d662bcccc1174bf
2:af402bbd7084cf051e97b8ad66af6e9fcc9746cc08cb74b7acf9620dbf7e1125
1:4e93426ef61f881225356b40d042c221860fadbb32b4cc4a7fd5e33256d60861
2:bb882532d04008f6d5424ee321860f3593b4ad7f4a12616b6e32c21fd6cc5642
```
Kuva 11: Tässä kellorivit ja sekoitusrivit kokonaisesta ressu kierroksesta

Viimeinen rivi on yhden genbytes (b:n arvolla 1) kierroksen tuloste. Muuttamalla kutsun muuttujan b arvoa, voimme suorittaa kierrosta useampia kertoja.

Huomaa että ajokerran aivan viimeiset merkit eivät tarkkaan ottaen vaikuta koko puskuriin. Jos haluaisi, että kaikki merkit vaikuttavat puskuriin, voisi toisen kiepukan ajaa lopuksi vielä kerran, eli sitä ajettaisiin viimeisen kellokierroksen jälkeen vielä yksi kerta. En nähnyt sitä tarpeelliseksi.

3.3 Kellomerkit lähellä puskurin kokoa

Huomasin yhden kellomerkkien sekoitusrutiinin heikkouden, jos kellomerkit ovat lähellä puskurin kokoa, kellobitit sekoittuvat tavallista vähemmän. Tilanne korjautuu kuitenkin seuraavalla kierroksella kun merkit saavat lisää kellobittejä. Seuraavassa kellobitit ovat välillä 0x1e(30) ja 0x22(34) ja puskurin pituus on 32. (listassa 1: merkityt rivit ovat kellon luvun jälkeen, ja 2: rivit ovat sekoituksen jälkeen)

```
1:1e1e1e1e1e1f1f1f1f1f1f1f2020202020202121212121212122222222222222
2:2222222121211e1e221f1f1f1f1f1f2020201f202020212121211e221e221e22
1:2b2b2b3232324c4c344e4f4f4f4f4f3132324c3232323031313l4f374f374830
2:4f484e374f4f4f324f4c4c2b324f31324c322b323130374f3130322b32323134
1:e1101cee1e1e1ee41f1919d7e51fe0e61ae6d4e6e0e3ed1de1e3e7d5e0e0e6ec
2:e11e1de0e4e61919e6e0ece61ed5d4e6ede3e0e31a1fd71ce0ee1e1fe5e110e7
1:4cacaa51595da2a35c50485cad3a3b5f49555355a7ad3cab524eafaa5f57b45b
2:5555adac485f3aa7aa514c535f3c524ea259a35b493b575cafabb4ad5c5d50aa
```
Kuva 12: kellobittien jonot välillä 30-34, kun puskurin pituus on 32

Ensimmäinen rivi sisältää puskurin heti kellonluvun jälkeen, pienin merkki on 1e ja suurin 22. Toisella (2:) rivillä merkit eivät kuitenkaan ole sekoittuneet lähes ollenkaan. Seuraavalla kierroksella ne kuitenkin sekoittuvat. Sama ilmiö tapahtuu myös puskurin pituuden kerrannaisilla (tässä tapauksessa 64, 96, 128 jne..).

Seuraavassa merkit ovat 64:n ympärillä (32 merkkinen puskuri):

```
1:3e3e3e3e3f3f3f3f3f4040404040414141414142424242424343434343444444
2:4140434342424242423f404040403f3f3f3f3e41433e433e41443e4441444341
1:0705000002020203f907070707f6f6f6f6f40a0ff50ff50b01f60208020c08
2:f6070207f6f60a0f080200000f07f4f50202030b08f9010702f502f6050cf607
```

Kuva 12: merkkien määrä samaa merkkiä sisältävässä merkkijonossa ~64

Ja 128:n ympärillä: (edelleen 32 merkkinen puskuri):

```
1:7f7f7f7f8080808080808181818181828282828282838383838384848484
2:8081818381818280828082 7f828482847f83807f838480837f83818183848280
1:d2d0d7d3d7d7d1d4d0d4d02bd0dfd3df28d1d729d0ded6df26dfdbdbdfd0dcd8
2:d7d4dcd1d0d4dfd0d129dbdf26d7d3d8d62bd0d3dfd7d0d0d7df28dfdbd0ded2
```

Kuva 13: merkkien määrä samaa merkkiä sisältävässä merkkijonossa ~128

Tästä voi huomata kääntäen, jos 32 merkkisessä tulospuskurissa on merkki 32 kahdeksannella kierroksella, se on ollus samalla paikalla myös edellisellä seitsemännellä kierroksella. Siitä ei kuitenkaan ole haittaa.

Tämä ongelma on helposti kierrettävissä valitsemalla puskurin koko, joka on suurempi kuin 255 (maksimi merkin arvo)

3.4 Puskurin pituus sama kuin kellojonon pituus

Tässä toinen erikoistapaus, jossa puskurin pituus on sama kun kellojonon pituus (tässä 32). Tässä ensimmäinen kellomerkki on 1, joita on 32 kappaletta, seuraava 2, 32 kappaletta jne. Huomaa kellonlukurutiinin shiftin vaikutus. Ensimmäisellä kerralla täytetään alin bittirivi ykkösellä, toisella kierroksella 0:lla jne. Tässä koko merkkirivi on samaa merkkiä, joten merkkien sekoitus ei muuta mitään, se vaihtaa vain samojen merkkien paikkoja. Jos koko rivin merkit ovat samoja mitään ei näytä tapahduvan.

```
1:0101010101010101010101010101010101010101010101010101010101010101
```

```
2:0101010101010101010101010101010101010101010101010101010101010101
1:0000000000000000000000000000000000000000000000000000000000000000
2:0000000000000000000000000000000000000000000000000000000000000000
1:0303030303030303030303030303030303030303030303030303030303030303
2:0303030303030303030303030303030303030303030303030303030303030303
1:0202020202020202020202020202020202020202020202020202020202020202
2:0202020202020202020202020202020202020202020202020202020202020202
1:0101010101010101010101010101010101010101010101010101010101010101
2:0101010101010101010101010101010101010101010101010101010101010101
1:0404040404040404040404040404040404040404040404040404040404040404
2:0404040404040404040404040404040404040404040404040404040404040404
1:0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f
2:0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f
1:1616161616161616161616161616161616161616161616161616161616161616
2:1616161616161616161616161616161616161616161616161616161616161616
```

Kuva 14: kellojonon pituus sama kuin puskurin pituus

Jos haluat kokeilla itse, kellorutiiniina on käytetty seuraavaa rutiinia:

```
int byte=0;
int count=9999;

unsigned char clockbyte()
{
 unsigned long usec,sec;

 if(count>=32) {
    count=0;
    byte++;
 }
 count++;
 return(byte);
}
```

Kuva 15

Tässä sama puskurikuva tapauksesta, jossa kellojonon pituus on 31 (yhtä pienempi kuin puskurin pituus):

```
1:0101010101010101010101010101010101010101010101010101010101010102
2:0101010101010101010101010101010101010101010101010101010101010201
1:0000000000000000000000000000000000000000000000000000000000000701
2:0000000000000007010000000000000000000000000000000000000000000000
1:030303030303030d0103030303030303030303030303030303030303040404
2:0303030303030303030303040303030303030303030d0103040303030303030403
1:020202020202020202020202020c020202020202020202021e06020c020203030d03
2:02020302020c020202020602020202020202020d0c02021e020202020303020202020202
1:01010301011d0101010109010101010101f1d0101390101010103000202020202
2:0139010109011d1d0101010103020102030201020 11f0101010101010100010101
1:0474040414043c3c040404040002040200020402043804040404050505070505
2:040402053c7414040002040404050402040 43c040404380504040500020504070 7
1:0f0f030d7fef2f0f07030f0f0f0d0f030f0f7f0f0f0f770d0f0002080c020006
2:002f7f0f0c0f020f03060f0d0fef0f7f0f0f00020f030f0d0f070d0f0f770308
1:0856f61610160c160e04161216d716f61616080c160e16121707131717e70f19
2:0c16040c161656f60f08121610160e16171616f617d708070e1713e716161912
```

Kuva 16: kellojonon pituus yhtä pienempi kuin puskurin pituus

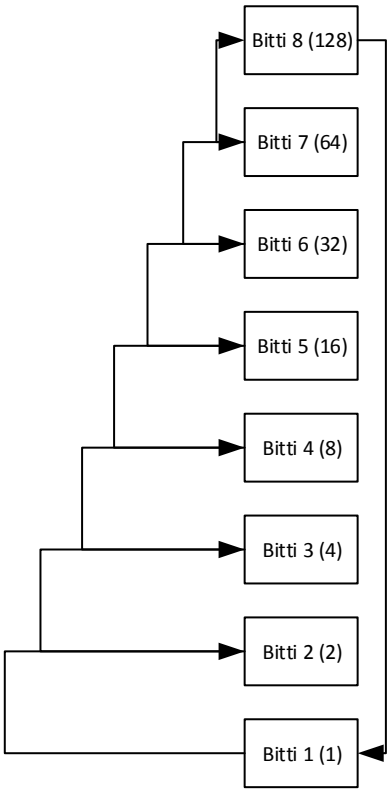
Tässä viimeinen rivi näyttää jotenkin satunnaiselta, mutta taitaa siinä kuitenkin olla liikaa ykkösiä.

3.5 Ohjelman palaset

Kellon palauttava palanen palauttaa 8 alinta bittiä kellosta. Näistä biteistä alin vaihtuu nopeimmin ykkösen ja nollan välillä ja ylin hitaimmin. Merkki saa yleensä arvot 0-255 ja sitten sarja alkaa alusta. Yhtä merkkiä voi olla nolla tai useampia kappaleita.

Genbytes() rutiinilla on kolme parametriä, ensimmäinen kertoo palautettavan (ja työalueen) pituuden ja toinen osoitteen. Kolmas kokonaisluku kertoo, kuinka monta kierrosta rutiini suorittaa. Genbytes() rutiini sisältää kaksi kierrosrakennetta:

Genbytes():in ensimmäinen kierrosrakenne kerää satunnaisbitit kellolta ja vaihtaa puskurin bittien järjestystä siten että kunkin puskurin merkin ylin bitti siirtyy alas ja alimmat bitit siirtyvät askelen ylös (Kuva 17).



Kuva 17: Bittien siirtely ensimmäisessä kierrosrakenteessa

Tämä bittien siirtely takaa sen, että kellon alin bitti koskee kaikkiin puskurin merkkien bitteihin. Toisaalta se että nämä toiminnot tehdään kaikille puskurin merkeille järjestyksessä takaa sen että kaikki puskurin merkit saavat saman kosketuksen kellobiteiltä.

Jos tässä vaihessa tarkastelee alimman bitin bittejä huomataan että ne ovat erillisinä nolla ja ykkös sarjoina, esimerkiksi (0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1) mutta seuraavassa vaiheessa tehtävä puskurimerkkien sekoitus järjestää ne uudelleen satunnaiseksi.

```
for(d=0;d<size;d++) {
   byte=clockbyte();
   e=buffer[d];
   e=((e&0x80)>>7) | ((e&0x7f)<<1);
   e=e^byte;
   buffer[d]=e;
}
```

Genbytes():in toinen toistorakenne vaihtaa puskurissa olevia merkkejä keskenään siten, että jokainen puskurin merkki on vuorollaan ensimmäisenä vaihdettavana merkkinä

(d:nä). Toinen vaihdettava merkki(f) saadaan siten, että ensimmäisen merkin puskurin sisältö lisätään (mod size) kenttään f. Puskurin merkkien d:s ja f:s paikka vaihdetaan.

```
f=0;
...
for(d=0;d<size;d++) { /* see rc4 */
   f=(f+buffer[d])%size;
   e=buffer[d];
   buffer[d]=buffer[f];
   buffer[f]=e;
}
```

Tämä rutiini on mielestäni se asia, joka erottaa tämän muista aiemmista kelloon pohjautusvista generaattoreista. Se huolehtii kellobittien nolla- ja ykkösbittien muodostamista jonoista sekoittaen ne uudelleen kellobittien perusteella.

D luuppi takaa sen, että kaikki merkit vaihdetaan ja muuttuja f takaa sen että merkki vaihdetaan satunnaisen merkin kanssa. Tässä f kehittyy koko rutiinin ajon ajan. jolloin se myös varmistaa että jokainen puskurin (kellon) merkki vaikuttaa lopputulokseen. (katso rc4)

Vielä kerran ohjelman muuttujat

- c jolla lasketaan rutiinin kokonaisia kierroksia
- d jolla käydään koko puskuri läpi merkki merkiltä
- e jota käytetään väliaikaisena puskurina yhdelle merkille
- f jolla lasketaan indeksiä toiseen vaihdettavaan merkkiin
- byte sisältää kellolta palautetun merkin.

3.6 Ohjelman osatoiminnot

Tässä taitavat olla kaikkien yksin toimivien satunnaisgeneraattorien palaset. Uutta tässä on mielestäni tuo puskurimerkkien sekoitus puskurimerkkien perusteella.

Tuloksen palauttavan puskurin osoitteen ja pituuden määrittely: Puskurin pituus ja osoite määritellään funktiokutsussa. Näiden lisäksi kutsussa määritellään myös b, joka kertoo kuinka monta kertaa funktio suoritetaan.

Kierrostoiminnot: tämä generaattori perustuu kahdeen kierrosrakenteeseen, toinen tallettaa kellon palauttamia arvoja puskuriin ja toinen sekoittaa puskurin. Sekoitusrutiini varmistaa sen, että vierekkäiset kellon palauttamat bitit ovat kaukana toisinaan, jolloin lähellä olevilla biteillä ei ole korrelaatioongelmaa. Vierekkäisten bittien korrelaatiohan helpottaa puskurin arvaamista.

Kellomerkit kerätään tiukassa toistorakenteessa (1. kiepukka), ja toistorakenne samalla vaihtaa merkin bittejä keskenään varmistaen että kellon alin bitti koskettaa puskurin kaikkia bittejä.

3.7 Tuloksen satunnaisuuden mittaaminen

Tuloksen satunnaisuutta voidaan testata erilaisin tilastollisin testein (esim. fips 140-2). Ne ovat testejä, joilla voidaan arvioida lukusarjan satunnaisuutta. Tämä ensimmäinen runs testi (rn30.c) laskee ykkösten ja nollien lukumäärän puskurissa. Niiden lukumäärän tulisi olla suurinpiirtein samat. Jos kolikkoa heitetään 20 kertaa, kruunoja ja klaavoja tulee molempia noin 10.

```
2:f983529e1dee22331e043547f63e1acea5bc9a3e25f4b26001a5b01fa65b9bf7
monobit ones: 125 zeroes 131, total: 256
```

Kuva 18: Monobit test

Kakkosella merkitty rivi sisältää puskurin, josta tilastollinen testi on tehty ja tekstillä monobit alkava rivi kertoo testin tuloksen. Tässä ykkösiä on ollut 125 ja nollia 131. Puskurin koko on 32 merkkiä eli 256 bittiä. Puskuri on tuotettu genbytes funktiolla. Jos puskurin pituus olisi näissä testeissä suurempi näkisimme ehkä vielä tasaisempia tuloksia.

Tämän tyylisiin testeihin voidaan rakentaa hylkäysehto, mutta minä en sellaista ole rakentanut, koska satunnaisluvuissa kaikki on mahdollista. Esimerkiksi tässä testissä se puskuri jossa on kaikki bitit ykkösiä ja ei yhtään nollaa tulee varmasti, kysymys on vain siitä kuinka kauan generaattoria ajetaan. Tässä runs testissä hylkäysehto voisi olla esimerkiksi jos jompikumpi luvuista on <40% puskurin koosta.

Toinen testi poker2 (rn31.c) jakaa kaikki merkit neljään kaksi bittiseen lukuun (0,1,2,3) ja laskee näiden lukumäärän:

```
2:970562a47a83434e5cb84812446480912473 2bcc3dc9006dd931a825b68891f0
poker2:  data:  0:43 1:33 2:31 3:21
```

```
poker2: total: 128, lowest: 21, highest: 43
```
Kuva 19: Poker-2 test

Tässä kahden bitin yhdistelmiä, jotka sisältävät luvun 0 (00) on 43 kappaletta, luvun 1 (01) sisältäviä on 33 kappaletta, luvun 2 (10) sisältäviä on 31 kappaletta ja luvun 3 (11) sisältäviä on 21 kpl. Hylkäysehto voisi olla esimerkiksi jos joku luvuista on pienempi kuin 19%. Edelleen mikään ei sano että samalla tavalla tuotettu puskuri, jossa on vähemmän kuin 19% nollia olisi sen vähemmän satunnaisempi kuin mikä tahansa muu puskuri. Saattaa toki olla myös generaattorin virhetoiminta, jos näissä testeissä tulee jotain muuta kuin odotettuja arvoja.

Seuraavassa poker4-testissä (rn32.c) merkki jaetaan kahteen neljä bittiseen merkkiin ja lasketaan bittijonojen eri arvojen lukumäärä. (0-15).

```
2:419cf5a4ccc44ae7eb07f55ec6b0d8ddd8a7f4c70a7d79dce72450992018cf22
poker4:  data:  0:5 1:2 2:4 3:0 4:6 5:4 6:1 7:7 8:3 9:4 10:4 11:2
12:8 13:6 14:4 15:4
poker4: total: 64, lowest: 0, highest: 8
```
Kuva 20: Poker-4 test

Tässä puskurin pituus on ilmeisesti liian pieni, ja nollakin mahtuu materiaaliin. Materiaalin arvojen keskikohta on (64/16) on neljä, joka pitää paikkansa.

Seuraava testi on runs-testi (rn33). Testi tilastoi ykkösbittien ja nollabittien sarjojen pituuden.

```
2:5f285943e16be1ff0d7048e92747b774091fa13855e97d578ff6db0058ba909b
runs: stat:    1:32 2:16 3:8 4:4 5:2 6:1 7:0 8:0 9:0 10:0 11:0
12:0
runs: zeroes:  1:35 2:13 3:4 4:8 5:0 6:0 7:0 8:1 9:0 10:0 11:1
runs: ones:    1:36 2:10 3:8 4:2 5:4 6:2 7:0 8:0 9:0 10:0 11:0
12:1
runs: total:  256
```
Kuva 21: Runs test

Stat rivillä on luku, joka kertoo kuinka monta kutakin sarjaa pitäisi olla tilastollisesti. Yhden bitin pituisia sarjoja pitäisi olla 32, kahden bitin pituisia sarjoja pitäisi olla 16 jne. Todelliset

luvut zeroes ja ones riveillä vertaantuvat hyvin tilastollisiin riveihin paitsi yksi 12 bitin sarja ykkösiä, joka tietysti löytyy sarjasta:

```
e1 10000111 ff 11111111 0d 10110000
```

Seuraavalla runs8-testillä voidaan katsella peräkkäisten saman arvon sisältävien ketjujen lukumääriä. Tällä testillä ei ole suurta merkitystä näin pienillä puskureilla, mutta kokeilin tätäkin niin lisään sen materiaaliin.

```
2:189b29c5c51eb92f5e87993b0a23194d6271eb636cf3612bb0c1cbecded5fdbd
runs8: any:  1:30 2:1
runs8: total:  32
```

Kuva 22: Runs-8 test

Materiaalissa kaksi samaa on peräkköin kohdassa c5c5 ja kaikki muut merkit ovat yksittäisiä ilmentymiä.

Tässä vielä yhteenvedot muutamasta tapauksesta, tavallinen genbytes:in luoma puskuri: listat on laatinut ohjelma rn40.c.

```
2:cc5c44990a5e8ea7fb204997bac5f012704546aa90f949c11216f522b4cbb86d
monobit ones: 137 zeroes 119, total: 256
poker2:  data:  0:34 1:35 2:34 3:25
poker2: total: 128, lowest: 25, highest: 35
poker4:  data:  0:5 1:4 2:5 3:0 4:7 5:5 6:3 7:3 8:2 9:7 10:5 11:5
12:6 13:1 14:2 15:4
poker4: total: 64, lowest: 0, highest: 7
runs: stat:    1:32 2:16 3:8 4:4 5:2 6:1 7:0
runs: zeroes:  1:32 2:20 3:9 4:4 5:3 6:0 7:1
runs: ones:    1:40 2:14 3:8 4:4 5:1 6:1
runs: total:  256
runs8: any:  1:32
runs8: total:  32
```

Kuva 23: Testien yhteenveto tavallisesta ressu puskurista

```
2:0000000000000000000000000000000000000000000000000000000000000000
monobit zeroes: 256 ones 0, total: 256
```

```
poker2:  data:  0:128 1:0 2:0 3:0
poker2: total: 128, lowest: 0, highest: 128
poker4:  data:  0:64 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 10:0 11:0
12:0 13:0 14:0 15:0
poker4: total: 64, lowest: 0, highest: 64
runs: stat:    1:32 2:16 3:8 4:4 5:2 6:1 7:0 8:0 9:0 10:0 11:0
12:0 13:0 14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22:0 23:0 24:0
25:0 26:0 27:0 28:0 29:0 30:0 31:0 32:0 33:0 34:0 35:0 36:0 37:0
38:0 39:0 40:0 41:0 42:0 43:0 44:0 45:0 46:0 47:0 48:0 49:0
runs: zeroes:  1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 10:0 11:0 12:0
13:0 14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22:0 23:0 24:0 25:0
26:0 27:0 28:0 29:0 30:0 31:0 32:0 33:0 34:0 35:0 36:0 37:0 38:0
39:0 40:0 41:0 42:0 43:0 44:0 45:0 46:0 47:0 48:0 49:1
runs: ones:
runs: total:  49
runs8: any:  1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 10:0 11:0 12:0
13:0 14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22:0 23:0 24:0 25:0
26:0 27:0 28:0 29:0 30:0 31:0 32:1
runs8: total:  32
```

Kuva 24: Testien yhteenveto nollia sisältävästä puskurista

3.8 Generaattorin mahdolliset ongelmat

Satunnaislukugeneraattoreihin voi kohdistaa erilaisia hyökkäyksiä (Wikipedia 1) . Tässä listattuna tähän generaattoriin sopivia.

Kello ainoa satunnaisuudenlähde, muuteltu kellofunktio, käsitelty kellojono muodostaa hyökkääjän haluaman puskurin, hyökkääjä arvaa kellojonon, hyökkääjä arvaa puskurin, laitteistoerot, laitteiston kuormitus, kello ei ole riittävän tarkka, kello on liian säännöllinen, kellon sarjat ovat liian pitkiä, bittigeneraattori on tunnistettavissa, bittigeneraattori ei tuota uniikkia tietoa.

Kello ainoa satunnaisuudenlähde: Tällä hetkellä suositut satunnaislukugeneraattorit (Ferguson, 2010) keräävät satunnaisuutta useasta lähteestä, jolloin jos hyökkääjä pääsee selville yhdestä lähteestä, generaattorilla on vielä käytettävissään loput lähteet. Kukaan ei tietenkään kiellä käyttämästä tätä generaattoria yhtenä lisälähteenä fortunassa. Toisaalta jos laitteen kello on korruptoitunut, todennäköisesti koko laite on.

Muuteltu kellofunktio: Hyökkääjä voi muokata kellofunktiota, ja tällä tavoin helpottaa omaa hyökkäystään. Muokataan sitä esimerkiksi siten, että kelloa kasvatetaan yhdellä jokaisella kutsulla, kun tämä generaattori kutsuu sitä. Näin hyökkääjän tarvitsee löytää vain generaattorijono ensimmäinen merkki, ja sen jälkeen kasvattaa kellojonosta palautettavaa merkkiä yhdellä. Hänellä on vain 256 kokeiltavaa numerosarjaa. Toisaalta jos hyökkääjä voi muokata kellosarjoja, hän voi aivan yhtä helposti lisätä prosessorin ajettavaksi ohjelman, joka tunnistaa ajettavat ohjelmat ja lähettää kaikki mielenkiintoiset puskurit itselleen.

Käsitelty kellojono muodostaa hyökkääjän haluaman puskurin. Edellisessä kellon muuntelu yrityksessä kellojonosta näkee, että se on säännöllinen, mutta jos hyökkääjä tallentaa oman avaimenmuodostuksessa saamansa kellojonon ja muodostetun puskurin ja palauttaa kellojonon uhrin kellojonona hänen pyytäessään satunnaislukuja generaattorilta. Näin uhrin generaattori muodostaa saman avaimen, jonka hyökkääjä talletti omalla suorituskerrallaan.

Hyökkääjä arvaa kellojonon: Yksinkertaisin arvaustapaus oli jo esiteltynä aiemmin. Jos kellojono muodostuu sarjasta: 1,2,3,4,5,6,7,8, … tarvitaan vain kellojonon aloitusmerkki, ja koko kellojono on hyökkääjän tiedossa. Jos taas kellojono on tyyliä 1,1,1,1,1,2,2,2,2,3,3,3,3,3,4,4,4,4 Arvataan vain ensimmäinen merkki, ja se onko ensimmäinen jono 4:n vai 5:n pituinen, ja se montako merkkiä ensimmäisestä sarjasta on jäljellä, eli tarvitaan 256*2*5 arvausta. Tämä hyökkäystapa on tietenkin hyödytön, jos hyökkääjä ei pääse kokeilemaan arvauksiaan mihinkään. Jos esimerkiksi tuotetaan istuntoavain, ja väärän avaimen saadessaan kohdeohjelma estää liikenteen, arvaus on mahdotonta.

Hyökkääjä arvaa puskurin: Jos satunnaislukugeneraattorin puskuri on hyvin muodostettu, hyökkääjän pitää arvata kaikki puskurin bitit. Jos taas puskurin muodostuksessa on korrelaatioita, niitä voidaan käyttää helpottamaan arvausta. Jos tässä generaattorissa ei olisi tuota merkkien sekoitusta (toinen kiepukka) ykkösen viereiset bitit olisivat todennäköisesti ykkösiä, ja nollan viereiset bitit olisivat todennäköisesti nollia. Tämän voisi ottaa huomioon arvauksessa.

Laitteistoerot: Laitteisto, jolla generaattoria ajetaan vaikuttaa lopputulokseen. Ohjelman kutsussa on muuttuja b, jolla määritellään kuinka monta yksittäistä “kierrosta” tehdään. “Kierros” lukee kelloa puskurinkoko*8 kertaa. Jos b on määritelty oikein, kellosta saadaan riittävä määrä satunnaisbittejä ja ajoaika on mahdollisimman pieni. Jos b on liian alhainen,

se saattaa helpottaa kellojonon arvaamista. Jos b on liian korkea puskuri kyllä sisältää tarpeeksi satunnaisuutta, mutta ajoaika tietenkin venyy.

Laitteiston kuormitus: Mitä enemmän satunnaisesti ajoaikaa vieviä ohjelmia laitteessa on, sen parempi. Yhden prosessin laitteessa kellojonon variaatioissa on ilmeisesti vähemmän satunnaisuutta.

Kello ei ole riittävän tarkka: Generaattori haluaa kellon, jossa alin bitti vaihtuu riittävän monta kertaa yhdessä kelloa puskuriin lukevassa kierroksessa(1 kiepukka). Jos alin bitti on kierroksen aikana aina nolla tai 1, se ei tietenkään sekoituksessa muuksi muutu, ja koko bittirivi jää ykköseksi (tai nollaksi).

Kello on liian säännöllinen: Jos kello palauttaa aina (edellinen kello+1), Tulos näyttää kyllä puskurissa hyvältä, mutta puskurilla on tietenkin vain 256 vaihtoehtoa, yksi jokaiselle kellosarjan aloitusmerkille.

Kellon sarjat ovat liian pitkiä: Jos kellon sarjat ovat pidempiä kuin puskuri yhdessä kierroksessa bitin arvot ovat joko nollia tai ykkösiä johtaa siihen että koko kellorivi on ykkösiä. Tämä voi korjautua osittain sillä että jollain toisella kierroksella kellon alin bitti vaihtuu puskurin sisällä, jolloin saadaan mukaan myös toista bittiä.

Bittigeneraattori on tunnistettavissa (rc4): bittijono on tunnistettavissa ensimmäisen kellonlukukierroksen jälkeen siitä, että puskurin numerot alkavat pienimmästä ja kasvavat jossakin jaksossa. Ensimmäisen sekoituskierroksen jälkeen puskurissa on edelleenkin ainoastaan merkkejä, jotka ovat ensimmäisen kierroksen pienimmän ja ensimmäisen kierroksen suurimman välissä. Toisella kierroksella puskurissa voi olla jono saman arvon sisältäviä merkkejä, joiden arvo on lähellä puskurin pituutta. Viimeisellä kierroksella voi olla merkkejä jotka ovat yhtä suuria kun puskurin pituus, ja ne ovat olleet samalla paikalla puskurissa jo edellisellä kierroksella. Nämä ovat tietenkin näkyviä vain osittaisella ressun ajoilla, ja eivät mielestäni vaikuta ”tavallisessa ajossa”, jossa puskurin koko on riittävä ja b:n arvo sopiva.

Bittigeneraattori ei tuota uniikkia tietoa.

3.9 Vielä hiukan b:stä

Aiemmin olen kertonut, että tämä b muuttuja on olennainen ressun tuottaman satunnaisuuden kannalta. B:hän kertoo suoraan kuinka monta kertaa koko puskuri

täytetään. Teoria on että jos b:n arvoa kasvatetaan riittävästi, ressu alkaa tuottamaan uniikkeja puskureita. Tässä pisteessä kellojono antaa riittävästi satunnaisuutta puskuriin. Olen ajanut ressua muutamam kuukauden neljällä laitteella, yrittäen löytää b:n arvoa, joka toisi koko ajan uniikkipuskureita(rn23.c). Tässä ovat tulokset: raspi 2, ja raspi 3 ovat mallit 2 ja kolme tuosta englantilaisesta yhden kortin linux koneesta, inuc on Intelin 3.18Ghz NUC ja probook on vanha probook 5320m läppärini. Luvut muuttuvat varmasti vielä kun uusia tuplia tulostavia b:n arvoja löytyy.

,

puskurin pituius/laite	raspberry pi 2	raspberry pi 3	inuc	probook
16	28	60	690	181
32	8	27	301	92
64	5	11	153	44
128	1	3	52	23
256	0	2	10	10
512	0	1	19	0
1024	0	0	2	0
2048	0	0	1	0
4096	0	0	0	0

Kuva 25: korkeimmat tuplia palauttavat b:n arvot (22.5.2016)

Taulukko kertoo siis viimeisen arvon, joka tuottaa tuplapuskureita. Taulukosta tällä hetkellä voisi sanoa, että jos käyttää jotain pisimmistä puskureista ja käyttää b:n arvoa, sanotaan varovaisesti 32, pitäisi olla turvallisilla vesillä.

Toinen asia on, että esimerkiksi puskurin pituudella 1024 (8192 bittiä) pitää jo käydä 2^8191(puolet) puskuria läpi, ennenkuin tuplia tilastollisestikaan löytyy, joten pitkien puiskureiden kanssa kannattaa käyttää todella harkintaa.

Kolmas asia b:stä on että ajattelin että satunnaisuus olisi kellomerkkijonon arvon vaihdoksissa, jolloin vaihdoksiahan on suurin piirtein yhtä paljon kaikissa koneissa. Tarkoitan siis sitä että millisekunteja on jokaisessa sekunnissa tuhat, ja nämä arvot tulevat kaikissa koneissa, tekijä joka vaihtelee on määrä paljonko kutakin arvoa palautuu. Tällöin

käytännöllistä voisi olla löytää oikea suoritusaika, jonka ressu pyörii ennen puskurin palautusta.

Yksi potentiaalinen kaavan lähtökohta on samoja merkkejä sisältävien merkkijonojen pituus, jonka näkee tällä rn22.c ohjelmalla.

Toki jos satunnaisuuden “hyvyys” ei ole niin olennaista, ihan kohtuulisia lukuja saa myös b:n arvolla 1.

Kirjoittamisen jälkeen olen löytänyt yhden uuden tavan, jolla voi haarukoida b:n arvoa. Se perustuu siihen että lisätään yksi/useampia bittejä jokaisesta kellomerkkien perussarjaan kuulumattomasta pituudesta. Jos esimerkiksi kellosarja on (4, 4, 5, 4, 4, 5, 4, 5, 5, 4, 4, 5, 1, 4, 5, 5,…), poistetaan sarjasta kaikki neloset ja viitoset ja jäljelle jäävistä jonoista katsotaan saatavan 1 bitti satunnaisuutta kustakin. Edellinen sarja antaisi 1 bitin satunnaisuutta. Toinen samaan asiaan perustuva tapa antaa hieman enemmän laskennallisia satunnaisbittejä ja siinä poistetaan vain perussarja laskennasta, eli edellisestä poistettaisi (4, 4, 5, 4, 4, 5, 4), (5, 4, 4, 5) ja (4, 5). Poistetaan siis kaikki variaatiot ja niiden osat perussarjasta: (4,4,5), (4,5,4) ja (5,4,4). Eli jäljelle jäisi 3 (yksi 5, yksi 1 ja yksi 5) bittiä satunnaisuutta. Näiden testiohjelman luominen jää lukijalle.

4 Pohdinta

Ressua voidaan mielestäni käyttää missä tahansa satunnaislukuja tarvitsevassa sovelluksessa. Kun b on riittävän suuri kellosarjan rakenne ei anna vihjeitä generoitavasta puskurista. Ominta aluetta ovat laitteet, joissa ei ole muuta satunnaisuutta kun kello tarjolla (ei fyysisiä levyjä, ei näyttöä, hiirtä, näppäimistöä, talletuspaikkaa edellä kerätylle satunnaisuudelle) tai ohjelmiston ensimmäisellä käynnistymiskerralla. Ressu sopii myös hyvin yhdeksi lähteeksi pseudo satunnaislukugeneraattoria, kuten fortuna. Se voi myös toimia hyvin osana ohjelman satunnaislukugeneraattoripinoa. Itse en uskalla suositella sitä yksinään ilman toista generaattoria ilman jatkotutkimusta.

Raportissa listatut satunnaisuustestit eivät sinänsä todista merkkijonon satunnaisuutta, ne antavat hyviä tuloksia Ressulle, vaikka kello palauttaisi koko ajan numerosarjaa 1,2,3,4,5,6,7,8,... Se kertoo kai ressun kellon keräily ja sekoitustoimintojen yhdistelmän toimivuudesta.

Ressu on yksinkertainen, se on helppo kirjoittaa muistista. ohjelmassa rn99.c on ohjelman mallitoteutus.

Jatkotutkimuksia voisi olla koettaa löytää heikkouksia, ja kehittää hyökkäystapoja. Ilmeiset ovat arvata kellojono, ja sitä kautta puskurin sisältö. Ensimmäiseksi arvaamista voisi kokeilla b:n arvolla 1. Voisi yrittää löytää korrelaatioita kellojonon ja palautettavan satunnaislukupuskurin väliltä. Yrittää löytää kellojono, joka palauttaa halutun puskurin. Tietysti jos minulta jää b:n kaava laskematta voitte kehittää sen.

Kellosarjan laadun tarkistus voisi olla kanssa kehityskohde. Voisi tarkistaa että tässä raportissa olevia ongelmia esiinny tuloksessa. Voisi myös tarkistaa että kellorutiini palauttaa oikean muotoista tietoa. Toisaalta voisi tarkistaa onko kellojonossa tarpeeksi satunnaisuutta, Esimerkiksi kellojonolla 1,2,3,4,5,6,7 ohjelma varoittaa riittävän satunnaisuuden puutteesta.

Mutta toisaalta jos tämä rutiini on vain yksi useammasta satunnaislukugeneraattorista, se voi vain lisätä turvallisuutta.

Lähteet

Berkeley University 1, Linear Feedback Shift Registers(LFSR:s), Luettavissa: http://inst.eecs.berkeley.edu/~cs150/sp03/handouts/15/LectureA/lec27-6up, Luettu: 21.4.2016

Daemen, J. & Rijmen, V., Advanced Encryption Standard (AES), Luettavissa: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf#page=1, Luettu: 21.4.2016

Diffie W. & Hellman M., 1976, New directions in cryptography, Luettavissa: https://www-ee.stanford.edu/~hellman/publications/24.pdf, Luettu: 12.5.2016

Ferguson N., Schneier, B. & Kohno, T., 2010, Cryptography Engineering, Design Principles and Practical Applications, Luettavissa: https://www.schneier.com/cryptography/paperfiles/fortuna.pdf, Luettu: 21.4.2016

FIPS180-4, 2012, Secure Hash Standard, 2012, Luettavissa: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, Luettu: 21.4.2016

Intel, Intel Digital Random Number Generator: Software Implementation Guide, 2012, Luettavissa: https://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf, Luettu: 21.4.2016

Kelsey, J., Schneier, B., Ferguson, N., 1999, Notes on the Design and Analysis of the Yarrow Cryptographic Preudorandom Number Generator, Luettavissa: https://www.schneier.com/cryptography/paperfiles/paper-yarrow.pdf, Luettu: 21.4.2016

Knuth, 1998, The Art of Computer Programming, Volume 2: Seminumerical algorithms, Third Edition, Addison Wesley

Kuivaniemi, J., 2016, Terttu järjestelmä, Luettavissa: http://moijari.com

Kuivaniemi, J., 2014, Terttu jatkuu, Luettavissa http://moijari.com/?p=62

National Institute of Standards and Technology, 2000, AES and Triple DES, Luettavissa: http://csrc.nist.gov/groups/ST/toolkit/block_ciphers.html, Luettu: 21.4.2016

National Institute of Standards and Technology , 2012, Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher, Luettavissa: http://csrc.nist.gov/publications/nistpubs/800-67-Rev1/SP-800-67-Rev1.pdf, Luettu: 22.5.2016

National Institute of Standards and Technology, 1999, DES, Luettavissa: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, Luettu: 9.5.2016

Payne W., Rabung J., & Bogyo, T., 1969, Coding the Lehmer Pseudo Random Number generator, Luettavissa: http://www.firstpr.com.au/dsp/rand31/p85-payne.pdf, Luettu: 21.4.2016

Rivest, R., 1992, The MD5 Message-Digest Algorithm, Luettavissa: http://tools.ietf.org/html/rfc1321?ref=driverlayer.com, Luettu: 5.4.2016

Rivest, R., RC4, Luettavissa: https://en.wikipedia.org/wiki/RC4, Luettu: 21.4.2016, Julkaisematon

Rivest, R., Shamir, A., Adleman, L., 1977, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Luettavissa: https://people.csail.mit.edu/rivest/Rsapaper.pdf, Luettu: 5.4.2016

Schneier, B., 1996. Applied Cryptography. Second Edition. John Wiley & Sons, Inc.

Silicon Graphics, 1997, Lavarand, Luettavissa: https://en.wikipedia.org/wiki/Lavarand, Luettu: 9.5.2016

Wikipedia 1, Random number attack, Luettavissa: https://en.wikipedia.org/wiki/Random_number_generator_attack, Luettu: 22.5.2016

Wired, 2003, Totally Random, Luettavissa: http://www.wired.com/2003/08/random/, Luettu: 9.5.2016

Zimmermann, P., 1991, Pretty Good Privacy, Luettavissa: https://en.wikipedia.org/wiki/Pretty_Good_Privacy, Luettu: 16.5.2016

Liitteet

Makefile	
ressu.c	Ressu generaattori
rn1.c	Kello 8 bittisenä merkkijono
rn2.c	Kello 8 bittisinä sarjoina
rn3.c	Kello 4 bittisenä merkkijonona
rn4.c	Kello 4 bittisinä sarjoina
rn23.c	Ensimmäisen uniikkeja puskureita palauttavan b:n haku
rn30.c	Monobit test
rn31.c	Poker-2 test
rn32.c	Poker-4 test
rn33.c	Runs test
rn34.c	Runs-8 test

Liite 1: Makefile

```
rn1objects = rn1.o
rn2objects = rn2.o
rn3objects = rn3.o
rn4objects = rn4.o
rn5objects = rn5.o
rn30objects = rn30.o
rn31objects = rn31.o
rn32objects = rn32.o
rn33objects = rn33.o
rn34objects = rn34.o
rn40objects = rn40.o

all:            rn1 rn2 rn3 rn4 rn5 rn30 rn31 rn32 rn33 rn34 rn40 lista

rn1:            $(rn1objects)
               cc -o rn1 $(rn1objects)

rn2:            $(rn2objects)
               cc -o rn2 $(rn2objects)

rn3:            $(rn3objects)
               cc -o rn3 $(rn3objects)

rn4:            $(rn4objects)
               cc -o rn4 $(rn4objects)

rn30:       $(rn30objects)
               cc -o rn30 $(rn30objects)

rn31:       $(rn31objects)
               cc -o rn31 $(rn31objects)

rn32:       $(rn32objects)
               cc -o rn32 $(rn32objects)

rn33:       $(rn33objects)
               cc -o rn33 $(rn33objects) -lm

rn34:       $(rn34objects)
               cc -o rn34 $(rn34objects)

rn40:       $(rn40objects)
               cc -o rn40 $(rn40objects) -lm

rn5:            $(rn5objects)
               cc -o rn5 $(rn5objects)

clean:
               rm rn1 $(rn1objects)
               rm rn2 $(rn2objects)
               rm rn3 $(rn3objects)
               rm rn4 $(rn4objects)
               rm rn30 $(rn30objects)
               rm rn31 $(rn31objects)
               rm rn32 $(rn32objects)
               rm rn33 $(rn33objects)
               rm rn34 $(rn34objects)
               rm rn40 $(rn40objects)
               rm *~
               rm \#*\#
lista:
               echo Executable version `./jarik2 --procid` >lista.txt
               echo "==========Makefile==========" >>lista.txt
```

```
cat Makefile >>lista.txt
echo "==========random.c==========" >>lista.txt
cat random.c >>lista.txt
echo "==========rn1.c==========" >>lista.txt
cat rn1.c >>lista.txt
echo "==========rn2.c==========" >>lista.txt
cat rn2.c >>lista.txt
echo "==========rn3.c==========" >>lista.txt
cat rn3.c >>lista.txt
echo "==========rn4.c==========" >>lista.txt
cat rn4.c >>lista.txt
echo "==========rn5.c==========" >>lista.txt
cat rn5.c >>lista.txt
echo "==========rn30.c==========" >>lista.txt
cat rn30.c >>lista.txt
echo "==========rn31.c==========" >>lista.txt
cat rn31.c >>lista.txt
echo "==========rn32.c==========" >>lista.txt
cat rn32.c >>lista.txt
echo "==========rn33.c==========" >>lista.txt
cat rn33.c >>lista.txt
echo "==========rn34.c==========" >>lista.txt
cat rn34.c >>lista.txt
echo "==========rn40.c==========" >>lista.txt
cat rn40.c >>lista.txt
echo "==========jarik2.c==========" >>lista.txt
cat jarik2.c >>lista.txt
cp lista.txt listat/listajarik2`date +%Y%m%d%H%M%S`.txt
```

Liite 2: ressu.c – Ressu generaattori

```
/*
* Ressu-satunnaislukugeneraattori versio 1.0
*
* (c)2013-2016 Jari Kuivaniemi, Kaikki oikeudet pidätetään!
*/
unsigned char clockbyte() /* JariK ~2013*/
{
	unsigned char byte;
	unsigned long usec,sec;

	struct timeval tv;

	gettimeofday(&tv,NULL);

	usec=tv.tv_usec;
	sec=tv.tv_sec;

	byte=(usec&0xff);

	return(byte);
}

genbytes(int size, unsigned char *buffer, int b) /* JariK ~2013*/
{
	int c,d,e,f;
	char byte;

	f=0;
	for(c=0;c<8*b;c++) {
		for(d=0;d<size;d++) {
			byte=clockbyte();
			e=buffer[d];
			e=((e&0x80)>>7) | ((e&0x7f)<<1);
			e=e^byte;
			buffer[d]=e;
		}
		for(d=0;d<size;d++) {
			f=(f+buffer[d])%size;
			e=buffer[d];
			buffer[d]=buffer[f];
			buffer[f]=e;
		}
	}
}
```

Liite 3: rn1.c – Kello 8 bittisenä merkkijonona

```
#include <stdio.h>
#include <sys/time.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavanlaisen listan:
*
f6 f6 f7 f8 f9 f9 fa fb fc fc fd fe ff 00 00 01
02 03 03 04 05 06 07 07 08 09 0a 0a 0b 0c 0d 0e
0e 0f 10 11 11 12 13 14 14 15 16 17 18 18 19 1a
1b 1b 1c 1d 1e 1e 1f 20 21 22 22 23 24 25 25 26
27 28 29 29 2a 2b 2c 2c 2d 2e 2f 2f 30 31 32 33
33 34 35 36 36 37 38 39 3a 3a 3b 3c 3d 3d 3e 3f
40 41 41 42 43 44 44 45 46 47 47 48 49 4a 4b 4b
4c 4d 4e 4f 4f 50 51 52 52 53 54 55 55 56 57 58
59 59 5a 5b 5c 5c 5d 5e 5f 60 60 61 62 63 63 64
65 66 66 67 68 69 6a 6a 6b 6c 6d 6d 6e 6f 70 71
*/

#define SIZE 1024

unsigned char buffer[SIZE];

main()
{
	int c;

	for(c=0;c<SIZE;c++) {
		buffer[c]=clockbyte();
	}

	for(c=0;c<SIZE;c++) {
		if(c%16==0 && c>0) {
			fprintf(stdout,"\n");
		}
		fprintf(stdout," %02x",buffer[c]);
	}
	fprintf(stdout,"\n");
	fflush(stdout);
}
```

Liite 4: rn2.c – Kello 8 bittisinä sarjoina

```
#include <stdio.h>
#include <sys/time.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavanlaisen listan:
*
00 01 01 02 03 04 04 05 06 07 08 08 09 0a 0b 0b 0c 0d 0e 0f 0f 10 11 12 12 13 14
15 16 16 17 18 19 19 1a 1b 1c 1c 1d 1e 1f 20 20 21 22 23 23 24 25 26 27 27 28 29
2a 2a 2b 2c 2d 2e 2e 2f 30 31 31 32 33 34 35 35 36 37 38 38 39 3a 3b 3c 3c 3d 3e
3f 3f 40 41 42 43 43 44 45 46 46 47 48 49 4a 4a 4b 4c 4d 4d 4e 4f 50 51 51 52 53
54 54 55 56 57 58 58 59 5a 5b 5b 5c 5d 5e 5e 5f 60 61 62 62 63 64 65 65 66 67 68
69 69 6a 6b 6c 6c 6d 6e 6f 70 70 71 72 73 73 74 75 76 77 77 78 79 7a 7a 7b 7c 7d
7e 7e 7f 80 81 81 82 83 84 85 85 86 87 88 88 89 8a 8b 8b 8c 8d 8e 8f 8f 90 91 92
92 93 94 95 96 96 97 98 99 99 9a 9b 9c 9d 9d 9e 9f a0 a0 a1 a2 a3 a3 a4 a5 a6 a7
a7 a8 a9 aa aa ab ac ad ae ae af b0 b1 b1 b2 b3 b4 b5 b5 b6 b7 b8 b8 b9 ba bb bc
bc bd be bf bf c0 c1 c2 c2 c3 c4 c5 c6 c6 c7 c8 c9 c9 ca cb cc cd cd ce cf d0 d0
d1 d2 d3 d4 d4 d5 d6 d7 d7 d8 d9 da da db dc dd de de df e0 e1 e2 e2 e3 e4 e5 e5
e6 e7 e8 e9 e9 ea eb ec ec ed ee ef f0 f0 f1 f2 f3 f3 f4 f5 f6 f7 f7 f8 f9 fa fa
fb fc fd fd fe ff
00 01 01 02 03 04 04 05 06 07 08 08 09 0a 0b 0c 0c 0d 0e 0f 0f 10 11 12 12 13 14
15 16 16 17 18 19 19 1a 1b 1c 1d 1d 1e 1f 20 20 21 22 23 24 24 25 26 27 27 28 29
2a 2a 2b 2c 2d 2e 2e 2f 30 31 31 32 33 34 35 35 36 37 38 38 39 3a 3b 3c 3c 3d 3e
3f 3f 40 41 42 43 43 44 45 46 46 47 48 49 4a 4a 4b 4c 4d 4d 4e 4f 50 51 51 52 53
54 54 55 56 57 58 58 59 5a 5b 5b 5c 5d 5e 5e 5f 60 61 62
*/

#define SIZE 1024

unsigned char buffer[SIZE];

main()
{
	int c,start;

	for(c=0;c<SIZE;c++) {
		buffer[c]=clockbyte();
	}

	start=0;
	for(c=0;c<SIZE;c++) {
		if(start==1&&buffer[c]<buffer[c-1]) {
			fprintf(stdout,"\n");
		}
		fprintf(stdout," %02x",buffer[c]);
		start=1;
	}
	fprintf(stdout,"\n");
	fflush(stdout);
}
```

Liite 5: rn3.c – Kello 4 bittisenä merkkijonona

```
#include <stdio.h>
#include <sys/time.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavanlaisen listan
*
09 09 0a 0b 0c 0d 0d 0e 0f 00 00 01 02 03 04 04
05 06 07 07 08 09 0a 0b 0b 0c 0d 0e 0e 0f 00 01
02 02 03 04 05 05 06 07 08 09 09 0a 0b 0c 0c 0d
0e 0f 00 00 01 02 03 03 04 05 06 07 07 08 09 0a
0a 0b 0c 0d 0e 0e 0f 00 01 01 02 03 04 05 05 06
07 08 08 09 0a 0b 0c 0c 0d 0e 0f 0f 00 01 02 03
03 04 05 06 06 07 08 09 0a 0a 0b 0c 0d 0d 0e 0f
00 01 01 02 03 04 05 05 06 07 08 08 09 0a 0b 0b
0c 0d 0e 0f 0f 00 01 02 02 03 04 05 06 06 07 08
09 0a 0a 0b 0c 0d 0e 0e 0f 00 01 02 02 03 04 05
*/

int size = 1048576;



main()
{
       int c;

       unsigned char buffer[size];

       for(c=0;c<size;c++) {
              buffer[c]=clockbyte()&0x0f;
       }

       for(c=0;c<size;c++) {
              if(c%32==0 && c>0) {
                     fprintf(stdout,"\n");
              }
              fprintf(stdout," %02x",buffer[c]);
       }
       fprintf(stdout,"\n");
       fflush(stdout);
}
```

Liite 6: rn4.c – Kello 4 bittisinä sarjoina

```
#include <stdio.h>
#include <sys/time.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaisen listan
*
00 01 01 02 03 04 05 05 06 07 08 09 09 0a 0b 0c 0c 0d 0e 0f
00 01 02 03 03 04 05 06 06 07 08 09 0a 0a 0b 0c 0d 0d 0e 0f
00 01 01 02 03 04 04 05 06 07 08 09 09 0b 0b 0c 0d 0e 0f 0f
01 01 02 03 04 04 05 06 07 07 08 09 0a 0b 0b 0c 0d 0e
01 04 06 07 09 0b 0d 0f
01 02 04 06 08 0a 0c 0e
00 02 04 05 07 09 09 0a 0b 0c 0d 0e 0f 0f
00 01 02 03 03 04 05 06 06 07 08 09 0a 0a 0b 0c 0d 0e 0f 0f
00 01 02 03 03 04 05 06 07 07 08 09 0a 0a 0b 0c 0d 0e 0e 0f
00 01 01 02 03 04 05 05 06 07 08 09 09 0a 0b 0c 0d 0d 0e 0f
*/

int size = 1048576;



main()
{
       int c,start;
       unsigned char buffer[size];

       for(c=0;c<size;c++) {
              buffer[c]=clockbyte()&0x0f;
       }

       start=0;
       for(c=0;c<size;c++) {
              if(start==1&&buffer[c]<buffer[c-1]) {
                     fprintf(stdout,"\n");
              }
              fprintf(stdout," %02x",buffer[c]);
              start=1;
       }
       fprintf(stdout,"\n");
       fflush(stdout);
}
```

Liite 7: rn5.c – Tiivistetty kellosarjojen tulostus

```
#include <stdio.h>
#include <sys/time.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaisen listan
*
1(00) 2(01) 1(02) 1(03) 1(04) 2(05) 1(06) 1(07) 2(08) 1(09) 1(0a) 1(0b)
2(0c) 1(0d) 1(0e) 2(0f) 1(10) 1(11) 1(12) 2(13) 1(14) 1(15) 1(16) 2(17)
1(18) 1(19) 2(1a) 1(1b) 1(1c) 1(1d) 2(1e) 1(1f) 1(20) 2(21) 1(22) 1(23)
1(24) 2(25) 1(26) 1(27) 2(28) 1(29) 1(2a) 1(2b) 2(2c) 1(2d) 1(2e) 2(2f)
1(30) 1(31) 1(32) 2(33) 1(34) 1(35) 2(36) 1(37) 1(38) 1(39) 2(3a) 1(3b)
1(3c) 1(3d) 2(3e) 1(3f) 1(40) 2(41) 1(42) 1(43) 1(44) 2(45) 1(46) 1(47)
2(48) 1(49) 1(4a) 1(4b) 2(4c) 1(4d) 1(4e) 2(4f) 1(50) 1(51) 1(52) 2(53)
1(54) 1(55) 2(56) 1(57) 1(58) 1(59) 2(5a) 1(5b) 1(5c) 1(5d) 2(5e) 1(5f)
1(60) 2(61) 1(62) 1(63) 1(64) 2(65) 1(66) 1(67) 2(68) 1(69) 1(6a) 1(6b)
2(6c) 1(6d) 1(6e) 2(6f) 1(70) 1(71) 1(72) 2(73) 1(74) 1(75) 2(76) 1(77)
*/

#define SIZE 1048576

unsigned char buffer[SIZE];

main()
{
	int c,first,lastbits,countbits;

	for(c=0;c<SIZE;c++) {
		buffer[c]=clockbyte();
	}

	lastbits=-1;
	first=1;

	for(c=0;c<SIZE;c++) {
		if(lastbits==-1) {
			lastbits=buffer[c];
			countbits=1;
		} else if(lastbits!=buffer[c]) {
			if(!first) {
				fprintf(stdout," ");
			}
			fprintf(stdout,"%d(%02x)",countbits,lastbits);
			first=0;
			if(lastbits>buffer[c]) {
				fprintf(stdout,"\n");
				first=1;
			}
			lastbits=buffer[c];
			countbits=1;
		} else if(lastbits==buffer[c]) {
			countbits++;
		}
	}
	fprintf(stdout,"\n");
	fflush(stdout);
}
```

Liite 8: rn22.c – Eripituisten merkkiketjujen lukumäärä

```
#include <stdio.h>
#include <sys/time.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavanlaisen listan:
*
24(1) 24(2) 16(3) 23(4) 21(5) 22(6) 26(7) 15(8) 22(9) 22(10)
19(11) 23(12) 25(13) 26(14) 28(15) 22(16) 31(17) 24(18)
14(19) 20(20) 19(21) 26(22) 29(23) 20(24) 17(25) 16(26)
19(27) 28(28) 37(29) 21(30) 20(31) 25(32) 27(33) 29(34)
35(35) 31(36) 41(37) 47(38) 77(39) 100(40) 292(41) 473(42)
1203(43) 643(44) 6454(45) 651(46) 30884(47) 3890(48)
*/

main()
{
	int c,d,oldc=-1,nextc,count,counts[1024];

	for(d=0;d<1024;d++)
		counts[d]=0;

	for(d=0;d<1048576;d++) {
		nextc=clockbyte();
		if(oldc==-1) {
			oldc=nextc;
			count=1;
		} else if(nextc!=oldc) {
			counts[count]++;
			oldc=nextc;
			count=1;
		} else if(nextc==oldc) {
			count++;
		}
	}
	for(c=0;c<1024;c++) {
		if(counts[c]!=0)
			fprintf(stdout," %d(%d)",counts[c],c);
	}
	fprintf(stdout,"\n");
	fflush(stdout);
}
```

Liite 9: rn23.c – Ensimmäisen uniikkeja puskureita tuottavan b:n haku

```
struct list {
	struct list *next;
	char *string;
};

struct list *first;

char *filename="test23.txt";
char *procname=NULL;
FILE *fpd;

clearlist(struct list **l)
{
	struct list *l1,*l2;

	l1=*l;

	while(l1!=NULL) {
		free(l1->string);
		l2=l1->next;
		free(l1);
		l1=l2;
	}
	*l=NULL;
}

int addstring(struct list **l, char *string)
{
	struct list *l1,*ln,*llast;

	ln=malloc(sizeof(struct list));
	ln->next=NULL;
	ln->string=malloc(strlen(string)+1);
	strcpy(ln->string,string);
	l1=*l;
	if(l1==NULL) {
		*l=ln;
	} else {
		llast=NULL;
		while(l1!=NULL) {
			llast=l1;
			l1=l1->next;
		}
		if(llast!=NULL)
			llast->next=ln;
		else
			(*l)->next=ln;
	}
}

int findstring(struct list *l,char *string)
{
	int ok;

	ok=1;

	while(l!=NULL) {
		if(!strcmp(l->string,string)) {
			ok=0;
			break;
		}
		l=l->next;
	}
	return(ok);
```

```
}

int testcountmax=32768;

struct list *l;

int dotest(int testsize, int b,int dumpsize,int testcount)
{
	int c,e,alku,loppu,dup;
	char filename[128];
	unsigned char buffer[32768];
	unsigned char string[32768],cdigit[3];
	FILE *fp1;

	dup=0;

	alku=(int)time(NULL);
	for(c=0;c<testcount;c++) {
		memset(buffer,0,testsize);
		genbytes(testsize,buffer,b);
		string[0]='\0';

		if(dumpsize>testsize) {
			for(e=0;e<testsize;e++) {
				sprintf(cdigit,"%02x",buffer[e]);
				strcat(string,cdigit);
			}
		} else {
			for(e=0;e<dumpsize;e++) {
				sprintf(cdigit,"%02x",buffer[e]);
				strcat(string,cdigit);
			}
		}
		if(!findstring(l,string)) {
			dup=1;
			break;
		} else {
			addstring(&l,string);
		}
	}
	loppu=(int)time(NULL);
	fprintf(fpd,"%04d testsize, %03d bytes per bit, %d total tries, %d
testcountmax, %d total seconds, %f seconds per crypt, %d duplicate",
		testsize,b,c,testcountmax,loppu-alku,((double)((double)loppu-
	alku)/testcount),dup);
	if(dup==0) {
	} else {
		if(testcountmax<c)
			testcountmax=c;
	}
	fprintf(fpd,"\n");
	fflush(fpd);

	clearlist(&l);
	return(dup);
}

#define USE100 2
#define USE50 2
#define USE10 2

main(int argc,char *argv[])
{
	int b,c,d,last,alku,loppu;

	procname=argv[0];
```

```
	testcountmax=1024*64;

	if((fpd=fopen(filename,"a"))==NULL) {
		fprintf(stdout,"%s: cannot open file %s\n",procname,filename);
		exit(1);
	}

	alku=(int)time(NULL);

	for(c=16;c<=4096;c+=c) {
		testcountmax=1024*32;
		last=0;

		if(c==16) last=170;
		else if(c==32) last=75;
		else if(c==64) last=44;
		else if(c==128) last=23;
		else if(c==256) last=5;
		else if(c==512) last=4;
		else if(c==1024) last=2;
		else if(c==2048) last=1;
		else if(c==4096) last=0;
#ifdef USE100
		 if(last==0) {
				for(b=last+1;b<1024;b+=100) {
					l=NULL;
					if(dotest(c,b,23,testcountmax*4)==0)
						break;
					last=b;
					clearlist(&l);
				}
#endif
#ifdef USE50
				for(b=last+1;b<1024;b+=50) {
					l=NULL;
					if(dotest(c,b,23,testcountmax*4)==0)
						break;
					last=b;
					clearlist(&l);
				}
#endif
#ifdef USE10
				for(b=last+1;b<1024;b+=10) {
					l=NULL;
					if(dotest(c,b,23,testcountmax*4)==0)
						break;
					last=b;
					clearlist(&l);
				}
#endif
				for(b=last+1;b<1024;b++) {
					l=NULL;
					if(dotest(c,b,23,testcountmax*4)==0)
						break;
					last=b;
					clearlist(&l);
				}
#endif
				for(b=last+1;b<1024;b++) {
					l=NULL;
					if(dotest(c,b,23,testcountmax*4)==0)
						break;
					last=b;
					clearlist(&l);
				}
		}
		for(b=last+1;b<1024;b++) {
```

```
                        l=NULL;
                        for(d=1;d<2;d++) {
                                if(dotest(c,b,23,testcountmax*4)==0)
                                        continue;
                                last=b;
                                break;
                        }
                        clearlist(&l);
                        if(d==2)
                                break;
                }
                fprintf(fpd,"%04d testsize, %03d bytes per bit, **\n",c,last);
        }
          loppu=(int)time(NULL);
          fprintf(stdout,"run took");
          if((loppu-alku)/60*60>0) {
                fprintf(stdout," %d hours",(loppu-alku)/60*60);
        }
          if((loppu-alku)/60*60>0) {
                fprintf(stdout," %d minutes",(loppu-alku)/60);
          }
          if((loppu-alku)%60*60>0) {
                fprintf(stdout," %d seconds",(loppu-alku)%60);
          }
          fprintf(stdout,"(%d seconds)\n",loppu-alku);
}
```

Liite 10: rn30.c – Monobit testi

```
#include <stdio.h>
#include <sys/time.h>
#include <memory.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaiset rivit:
*
2:d6053b72a10ed29c6f2511cb00bfba3c1771558b5a1643258ba4af1c345809e3
monobit ones: 136 zeroes 120, total: 256
*/


#include <stdarg.h>

void message(const char *format, ...)
{
       va_list args;
       char buffer[1024];

       va_start(args,format);
       vfprintf(stdout,format,args);
       fprintf(stdout,"\n");
       fflush(stdout);
       va_end(args);
}

int monobit_zeroes = 0, monobit_ones = 0;

reset_monobit()
{
       monobit_zeroes = 0;
       monobit_ones = 0;
}

test_monobit(int size, unsigned char *buffer)
{
       int c,d,zeroes,ones;

       for(c=0;c<size;c++) {
              for(d=0;d<8;d++) {
                     if((buffer[c]>>d)&1)
                            monobit_ones++;
                     else
                            monobit_zeroes++;
              }
       }
}

results_monobit()
{
       message("monobit zeroes: %d ones: %d, total: %d",
              monobit_zeroes,monobit_ones,monobit_zeroes+monobit_ones);
}

main()
{
       int c;
       unsigned char buffer[32];

       memset(buffer,0,32);
       genbytes(sizeof(buffer),buffer,1024);

       fprintf(stdout,"2:");
```

```
        for(c=0;c<sizeof(buffer);c++) {
                fprintf(stdout,"%02x",buffer[c]);
        }
        fprintf(stdout,"\n");

        reset_monobit();
        test_monobit(sizeof(buffer),buffer);
        results_monobit();
}
```

Liite 11: rn31.c – Poker-2 testi

```
#include <stdio.h>
#include <sys/time.h>
#include <string.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaiset rivit:
*
2:a9163e71989ae904a0433cd6f24f6a5f9cfca685edba2a99120d33b6cb368bb1
poker2:  data:  0:29 1:28 2:41 3:30
poker2: total: 128, lowest: 28, highest: 41
*/

#include <stdarg.h>

void message(const char *format, ...)
{
       va_list args;
       char buffer[1024];

       va_start(args,format);
       vfprintf(stdout,format,args);
       fprintf(stdout,"\n");
       fflush(stdout);
       va_end(args);
}

int poker2_i[4];

reset_poker2()
{
       int c;

       for(c=0;c<4;c++) {
              poker2_i[c]=0;
       }
}

test_poker2(int size, unsigned char *buffer)
{
       int c;

       for(c=0;c<size;c++) {
              poker2_i[(buffer[c]>>6)&0x03]++;
              poker2_i[(buffer[c]>>4)&0x03]++;
              poker2_i[(buffer[c]>>2)&0x03]++;
              poker2_i[buffer[c]&0x03]++;
       }
}

results_poker2()
{
       int c,total,lowest,highest;
       char tmp[32],buffer[1024];

       total=0;
       lowest=999999999;
       highest=0;
       *buffer='\0';

       for(c=0;c<4;c++) {
              sprintf(tmp," %d:%d",c,poker2_i[c]);
              strcat(buffer,tmp);
```

```
                total+=poker2_i[c];
                if(lowest>poker2_i[c])
                        lowest=poker2_i[c];
                if(highest<poker2_i[c])
                        highest=poker2_i[c];
        }
        message("poker2:  data: %s",buffer);
        message("poker2: total: %d, lowest: %d, highest:
%d",total,lowest,highest);
}

main()
{
        int c;
        unsigned char buffer[32];

        memset(buffer,0,32);
        genbytes(sizeof(buffer),buffer,1024);

        fprintf(stdout,"2:");
        for(c=0;c<sizeof(buffer);c++) {
                fprintf(stdout,"%02x",buffer[c]);
        }
        fprintf(stdout,"\n");

        reset_poker2();
        test_poker2(sizeof(buffer),buffer);
        results_poker2();
}
```

Liite 12: rn32.c – Poker-4 testi

```
#include <stdio.h>
#include <sys/time.h>
#include <string.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaiset rivit:
*
2:0b8e88f13974e4db387198383568417de8dacef4bec750b84cdb39754b64608e
poker4:  data:  0:3 1:3 2:0 3:5 4:7 5:3 6:3 7:5 8:10 9:3 10:1 11:6 12:3 13:4 14:6
15:2
poker4: total: 64, lowest: 0, highest: 10
*/

#include <stdarg.h>

void message(const char *format, ...)
{
       va_list args;
       char buffer[1024];

       va_start(args,format);
       vfprintf(stdout,format,args);
       fprintf(stdout,"\n");
       fflush(stdout);
       va_end(args);
}

int poker4_i[16];

reset_poker4()
{
       int c;

       for(c=0;c<16;c++) {
              poker4_i[c]=0;
       }
}

test_poker4(int size, unsigned char *buffer)
{
       int c;

       for(c=0;c<size;c++) {
              poker4_i[(buffer[c]>>4)&0x0f]++;
              poker4_i[buffer[c]&0x0f]++;
       }
}

results_poker4()
{
       int c, total, lowest, highest;
       char tmp[32],buffer[1024];

       total=0;
       lowest=999999999;
       highest=0;
       *buffer='\0';

       for(c=0;c<16;c++) {
              sprintf(tmp," %d:%d",c,poker4_i[c]);
              strcat(buffer,tmp);
              total+=poker4_i[c];
```

```
                if(lowest>poker4_i[c])
                        lowest=poker4_i[c];
                if(highest<poker4_i[c])
                        highest=poker4_i[c];
        }
        message("poker4:  data: %s",buffer);
        message("poker4: total: %d, lowest: %d, highest:
%d",total,lowest,highest);
}

main()
{
        int c;
        unsigned char buffer[32];

        memset(buffer,0,32);
        genbytes(sizeof(buffer),buffer,1024);

        fprintf(stdout,"2:");
        for(c=0;c<sizeof(buffer);c++) {
                fprintf(stdout,"%02x",buffer[c]);
        }
        fprintf(stdout,"\n");

        reset_poker4();
        test_poker4(sizeof(buffer),buffer);
        results_poker4();
}
```

Liite 13: rn33.c – Runs testi

```
#include <stdio.h>
#include <sys/time.h>
#include <string.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaiset rivit:
*
2:006521e15a2921029381d37feca61954b8ca0f52b380479d6383f1acefc8053d
runs: stat:    1:32 2:16 3:8 4:4 5:2 6:1 7:0 8:0 9:0
runs: zeroes:  1:27 2:16 3:6 4:4 5:4 6:2 7:1 8:1
runs: ones:    1:32 2:14 3:8 4:3 5:1 6:1 7:0 8:0 9:1
runs: total:  256
*/

#include <stdarg.h>

void message(const char *format, ...)
{
       va_list args;
       char buffer[1024];

       va_start(args,format);
       vfprintf(stdout,format,args);
       fprintf(stdout,"\n");
       fflush(stdout);
       va_end(args);
}

#include <math.h>

#define RUNSMAX 50

int runs_zero[RUNSMAX];
int runs_ones[RUNSMAX];

reset_runs()
{
       int c;

       for(c=0;c<RUNSMAX;c++) {
              runs_zero[c]=0;
              runs_ones[c]=0;
       }
}

test_runs(int size, unsigned char *buffer)
{
       int c,d,firstbit,count,total;

       firstbit=-1;
       count=0;
       for(c=0;c<size;c++) {
              for(d=0;d<8;d++) {
                     if(firstbit==-1) {
                            firstbit=buffer[c]&1;
                        count=1;
                     } else {
                            if(((buffer[c]>>d)&1)==firstbit) {
                                   count++;
                            } else {
                                   if(count>=RUNSMAX)
                                          count=RUNSMAX-1;
```

```
                                        if(count<RUNSMAX) {
                                                if(firstbit==0)
                                                        runs_zero[count]++;
                                                else
                                                        runs_ones[count]++;
                                        }
                                        firstbit=(buffer[c]>>d)&1;
                                        count=1;
                                }
                        }
                }
        }
        if(count>=RUNSMAX)
                count=RUNSMAX-1;
        if(count<RUNSMAX) {
                if(firstbit==0)
                        runs_zero[count]++;
                else
                        runs_ones[count]++;
        }
}

results_runs(int size)
{
        int c,d,total;
        char tmp[32],buffer[4096];

        total=0;

        *buffer='\0';
        for(d=RUNSMAX-1;d>0 && runs_zero[d]==0 && runs_ones[d]==0 ;d--);
        for(c=1;c<=d;c++) {
                sprintf(tmp," %d:%d",c,(int)(pow((double)0.5,c+2)*(size*8)));
                strcat(buffer,tmp);
        }
        message("runs: stat:   %s",buffer);

        *buffer='\0';
        for(d=RUNSMAX-1;d>0 && runs_zero[d]==0;d--);
        for(c=1;c<=d;c++) {
                sprintf(tmp," %d:%d",c,runs_zero[c]);
                strcat(buffer,tmp);
                total+=c*runs_zero[c];
        }
        message("runs: zeroes: %s",buffer);

        *buffer='\0';
        for(d=RUNSMAX-1;d>0 && runs_ones[d]==0;d--);
        for(c=1;c<=d;c++) {
                sprintf(tmp," %d:%d",c,runs_ones[c]);
                strcat(buffer,tmp);
                total+=c*runs_ones[c];
        }
        message("runs: ones:   %s",buffer);
        message("runs: total:  %d",total);
}

main()
{
        int c,d;
        unsigned char buffer[32];

        memset(buffer,0,32);
        genbytes(sizeof(buffer),buffer,1024);

        fprintf(stdout,"2:");
        for(c=0;c<sizeof(buffer);c++) {
```

```
                fprintf(stdout,"%02x",buffer[c]);
        }
        fprintf(stdout,"\n");

        reset_runs();
        test_runs(sizeof(buffer),buffer);
        results_runs(sizeof(buffer));
}
```

Liite 14: rn34.c – Runs-8 testi

```
#include <stdio.h>
#include <sys/time.h>
#include <string.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaiset rivit:
*
runs8: any:  1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0 10:0 11:0 12:0 13:0 14:0 15:0
16:0 17:0 18:0 19:0 20:0 21:0 22:0 23:0 24:0 25:0 26:0 27:0 28:0 29:0 30:0 31:0
32:1
runs8: total:  32
2:e469e36b4526f5f359730d768bb481d1c2ff4cc395e6ab3072a680a73b25bd3a e4 00100111 69
10010110 e3 11000111 6b 11010110 45 10100010 26 01100100 f5 10101111 f3 11001111
59 10011010 73 11001110 0d 10110000 76 01101110 8b 11010001 b4 00101101 81
10000001 d1 10001011 c2 01000011 ff 11111111 4c 00110010 c3 11000011 95 10101001
e6 01100111 ab 11010101 30 00001100 72 01001110 a6 01100101 80 00000001 a7
11100101 3b 11011100 25 10100100 bd 10111101 3a 01011100
runs8: any:  1:32
runs8: total:  32
*/

#include <stdarg.h>

void message(const char *format, ...)
{
       va_list args;
       char buffer[1024];

       va_start(args,format);
       vfprintf(stdout,format,args);
       fprintf(stdout,"\n");
       fflush(stdout);
       va_end(args);
}

#define RUNSMAX 50

int runs8[RUNSMAX];

reset_runs8()
{
       int c;

       for(c=0;c<RUNSMAX;c++) {
              runs8[c]=0;
       }
}

test_runs8(int size, unsigned char *buffer)
{
       int c,d,firstbyte,count,total;

       firstbyte=-1;
       count=0;

       for(c=0;c<size;c++) {
              if(firstbyte==-1) {
                     firstbyte=buffer[c];
                     count=1;
              } else {
                     if(buffer[c]==firstbyte) {
                            count++;
```

```
                        } else {
                                if(count>=RUNSMAX)
                                        count=RUNSMAX-1;
                                if(count<RUNSMAX) {
                                        runs8[count]++;
                                }
                                firstbyte=buffer[c];
                                count=1;
                        }
                }
        }
        if(count>=RUNSMAX)
                count=RUNSMAX-1;
        if(count<RUNSMAX) {
                runs8[count]++;
        }
}

results_runs8()
{
        int c,d,total;
        char tmp[32],buffer[4096];

        total=0;

        *buffer='\0';
        for(d=RUNSMAX-1;d>0 && runs8[d]==0;d--);
        for(c=1;c<=d;c++) {
                sprintf(tmp," %d:%d",c,runs8[c]);
                strcat(buffer,tmp);
                total+=c*runs8[c];
        }
        message("runs8: any: %s",buffer);
        message("runs8: total:  %d",total);
}

main()
{
        int c,d;
        unsigned char buffer[32];

        memset(buffer,0,32);

        genbytes(sizeof(buffer),buffer,1024);

        fprintf(stdout,"2:");
        for(c=0;c<sizeof(buffer);c++) {
                fprintf(stdout,"%02x",buffer[c]);
        }
        for(c=0;c<sizeof(buffer);c++) {
                fprintf(stdout," %02x ",buffer[c]);
                for(d=0;d<8;d++)
                        fprintf(stdout,"%1d",(buffer[c]>>d)&1);
        }
        fprintf(stdout,"\n");

        reset_runs8();
        test_runs8(sizeof(buffer),buffer);
        results_runs8(sizeof(buffer));
}
```

Liite 15: rn40.c – Kaikki satunnaisuus testit

```
#include <stdio.h>
#include <sys/time.h>
#include <memory.h>

#include "ressu.c"

/* Ohjelma tulostaa seuraavankaltaiset rivit:
*
2:7a95c41bc48dbcb148dc7eb2b44281eb262c54dd864de5650bdd1d6815671263
monobit ones: 132 zeroes 124, total: 256
poker2:  data:  0:31 1:41 2:29 3:27
poker2: total: 128, lowest: 27, highest: 41
poker4:  data:  0:1 1:6 2:5 3:1 4:7 5:5 6:6 7:3 8:5 9:1 10:1 11:7 12:5 13:8 14:3
15:0
poker4: total: 64, lowest: 0, highest: 8
runs: stat:    1:32 2:16 3:8 4:4 5:2 6:1
runs: zeroes:  1:40 2:14 3:9 4:5 5:1 6:2
runs: ones:    1:37 2:19 3:9 4:4 5:0 6:1
runs: total:  256
runs8: any:  1:32
runs8: total:  32
*/

#include <stdarg.h>

void message(const char *format, ...)
{
       va_list args;
       char buffer[1024];

       va_start(args,format);
       vfprintf(stdout,format,args);
       fprintf(stdout,"\n");
       fflush(stdout);
       va_end(args);
}

/* Lisää tähän väliin rutiinit ohjelmista rn3*.c */

main()
{
       int c;
       unsigned char buffer[32];

       memset(buffer,0,32);
       genbytes(sizeof(buffer),buffer,1024);

       fprintf(stdout,"2:");
       for(c=0;c<sizeof(buffer);c++) {
              fprintf(stdout,"%02x",buffer[c]);
       }
       fprintf(stdout,"\n");

       reset_monobit();
       test_monobit(sizeof(buffer),buffer);
       results_monobit();

       reset_poker2();
       test_poker2(sizeof(buffer),buffer);
       results_poker2();

       reset_poker4();
       test_poker4(sizeof(buffer),buffer);
```

```
        results_poker4();

        reset_runs();
        test_runs(sizeof(buffer),buffer);
        results_runs(sizeof(buffer));

        reset_runs8();
        test_runs8(sizeof(buffer),buffer);
        results_runs8(sizeof(buffer));
}
```

Liite 16: rn99.c – Mallitoteutus

```
#include <stdio.h>
#include <sys/time.h>
#include <memory.h>

#include "ressu.c"

#define DEBUG 2

#define RESSU_SIZE 1024
#define RESSU_B 64
int ressu_count=9999;
unsigned char ressu_buffer[RESSU_SIZE];

int ressu_reset()
{
	memset(ressu_buffer,0,sizeof(ressu_buffer));
	ressu_count=9999;
}

int ressu_byte()
{
	int c;

	ressu_count++;
	if(ressu_count>=RESSU_SIZE) {
		memset(ressu_buffer,0,sizeof(ressu_buffer));
		genbytes(sizeof(ressu_buffer),ressu_buffer,RESSU_B);
		ressu_count=0;
	}
	return(ressu_buffer[ressu_count]);
}

ressu_xor(int size,unsigned char *buffer)
{
	int c;

	for(c=0;c<size;c++) {
		buffer[c]^=ressu_byte();
	}
}

main()
{
	int c;
	unsigned char buffer[128];

	memset(buffer,0,sizeof(buffer));
	ressu_xor(sizeof(buffer),buffer);
	for(c=0;c<sizeof(buffer);c++) {
		fprintf(stdout,"%02x",buffer[c]);
	}
	fprintf(stdout,"\n");
	ressu_reset(); /* Wipe buffer memory area */
	memset(buffer,0,sizeof(buffer));
}
```