SCAM (Secure Compression Algorithm for Multimedia)

- Utveckling och Implementation

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH Ingenjörshögskolan vid Campus Helsingborg
Datateknik

Examensarbete:
Joel Lovén
Einar Persson

© Copyright Joel Lovén, Einar Persson

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2009

Sammanfattning

Denna rapport har som uppgift att gå igenom hur den nya krypterings och komprimeringsalgoritmen SCAM har utvecklats. Kryptering och komprimering är två vanliga begrepp inom datateknik och det finns många olika typer av algoritmer inom båda områdena. Rapporten går igenom en möjlig kombination av dessa två begrepp till en enda algoritm. Under projektets gång så har två nya algoritmer utvecklats, en komprimeringssalgoritm och en krypteringsalgoritm.

Komprimeringsalgoritmen LPRL (Lovén Persson Run-Length) är en egenutvecklad algoritm. Inspiration till LPRL har mestadels kommit från komprimeringsalgoritmen Run-Length kodning. LPRL kollar igenom data efter upprepningar och om där finns sådana så tas de bort från datan och representeras istället i en speciell tabell.

Krypteringsalgoritmen LPCR (Lovén Persson Crypto) bygger en del på funktionaliteten hos strömchiffer med den skillnaden att den även har blockchifferegenskaper som är tagna från blockchiffertypen Räknare (CTR). LPCR behöver inte behålla någon synkronisering mellan sändare/mottagare såsom många andra krypteringsalgoritmer måste utan kan skicka data mycket mer fritt mellan en eller flera sändare/mottagare.

Till sist så har LPRL och LPCR blivit integrerade med varandra för att skapa SCAM algoritmen. Utvecklingen av LPRL och LPCR har gjorts med tanke på att de senare skulle integreras med varandra.

Resultatet av projektet är en prototyp av SCAM-algoritmen som är implementerad i en testversion av ett egenutvecklat chatprogram.

Nyckelord: kryptering, komprimering, säkerhet, kommunikation, information, strömning, chiffer.

Abstract

This purpose of this paper is to explain the development of the new encryption and compression algorithm SCAM. Encryption and compression are two common notions in computer science and there exist quite a few different algorithms in both areas. This paper will demonstrate a possible merge of these two types of algorithms into a single one. Two new algorithms have been developed during the course of the project, one encryption algorithm and one compression algorithm.

The compression algorithm LPRL (Lovén Persson Run-Length) is a new algorithm developed during the project. LPRL have been mostly inspired by the compression algorithm Run-Length encoding. LPRL checks the data for repetitions and if some are found they will be removed from the data and represented in a special table instead.

The encryption algorithm LPCR (Lovén Persson Crypto) is partly based on the functionality of stream ciphers. The difference is that it also incorporates ideas from block ciphers and to be more precise it uses features from the Counter (CTR) type of block cipher. LPCR does not require any continuous form of synchronization with the sender/recipient which most other ciphers need. This means that LPCR can send and receive data from one or more sender/recipient much more freely.

The last step in the process was to integrate both LPRL and LPCR with each other in order to create the final SCAM algoritm. Both LPRL and LPCR was developed with the intent that they would be integrated with each other at a later stage.

The result of the project is a prototype of the SCAM algorithm that is implemented in a test version of a chat program. The chat program has been developed during the project also.

Keywords: encryption, compression, security, communication, information, streaming, cipher.

Förord

Denna rapport är skriven som examensarbete och slutmoment i högskoleingenjörsutbildning inom datorteknik vid Lunds Tekniska Högskola-Campus Helsingborg.
Ämnet har valts eftersom båda författare är intresserade av datasäkerhet, algoritmkonstruktion och mjukvaruutveckling.

Vi vill tacka Mats Lilja för handledning under projektet.

Joel Lovén & Einar Persson
2009-05-27 Helsingborg

Innehållsförteckning

1 Inledning 1
1.1 backgrund 1
1.2 Mål 1
1.2.1 Eftersträvade mål 1
1.2.2 Krav 2
1.3 Metod 2
1.4 Presentationsmetod för resultat 2

2 Kompressionsalgoritmen LPRL (Lovén Persson Run-Length) 3
2.1 Undersökning av befintliga algoritmer 3
2.1.1.1 Huffman Coding 3
2.1.1.2 Matematisk kompression 4
2.1.1.3 Run-length kodning 4
2.2 Utveckling av Kompressionsalgoritmen (LPRL) 5
2.2.1 Nackdelar med run-length kodning 5
2.2.2 Effektiviserande av datarepresentationen 7
2.2.2.1 Avskalning av onödiga representationer 7
2.2.2.2 Effektivisering av representation 9
2.2.3 Utdatakonvention 9
2.2.4 Slututveckling av LPRL 12
2.3 Sammanfattad beskrivning av LPRL 13
2.4 Implementation av LPRL 14
2.4.1 LPRL kompression i pseudokod 14
2.4.2 LPRL avkompression i pseudokod 15
2.4.3 Hjälpalgoritmer 16
2.4.3.1 Ljudbehandling 16
2.4.3.2 Specialimplementation av LPRL för Javaljud 17
2.4.3.3 Hjälpalgoritm för bildbehandling 18

3 Krypteringsalgoritmen LPCR (Lovén Persson Crypto) 19
3.1 Bakgrund och idéer 19
3.1.1 Blockchiffer 19
3.1.2 Strömchiffer 21
3.2 Utveckling av LPCR 23
3.2.1 Prototyper 24
3.2.1.1 LPCRP1 (Lovén Persson Crypto Prototype 1) 24
3.2.1.2 LPCRP2 (Lovén Persson Crypto Prototype 2) 25
3.2.1.3 LPCRP3 (Lovén Persson Crypto Prototype 3) 26
3.3 Beskrivning av LPCR 27
3.3.1 Utökningsmetoden 27
3.3.2 Chiffergenerator 28

3.3.3 Säkerhet 29

4 Kompressions & krypteringsalgoritmen SCAM (Secure compressions algorithm for multimedia) 32

4.1 Integration av algoritmerna 32

4.1.1 Utvecklingsversion1 32

4.1.2 Utvecklingsversion2 33

4.1.3 Slutversion av algoritmen 34

4.1.4 Tidskomplexitet 36

4.2 Implementation 37

4.2.1 krypterings/kompressions-algoritm i pseudokod 37

4.2.2 avkrypterings/avkompressions-algoritm i pseudokod 39

5 Proof of Concept 40

5.1 Chat programmet Libercom 40

5.2 LPRL vissningsprogram 42

5.3 Testresultat 43

5.3.1 LPCR 43

5.3.2 LPRL 44

5.3.2.1 LPRL och LPIS bildtest 45

5.3.3 SCAM 45

Slutsatser 46

6 Referenser 48

7 Ordlista 50

1 Inledning

1.1 backgrund

Idag finns det många anledningar att skydda sin information. Hoten består bland annat av kriminella hackers, bedragare och myndigheter som överträder sina befogenheter. Därför finns det ett stort intresse för krypteringsteknologi och om denna teknik dessutom kan utvecklas för att ge fördelar i form av snabbare överföring och mindre lagringsutrymme är detta bara positivt.
Detta projekt strävar därför efter att skapa en algoritm som kan åstadkomma just detta, nämligen att komprimera och kryptera i en och samma algoritm.

Idén till projektet uppkom vid ett tillfälle då författarna till rapporten spelade onlinespel och uppmärksammade att det behövdes en form av kommunikationsverktyg som var svår för motståndare att avlyssna.
Vid närmare diskussion framkom även ett behov av att kunna komprimera data för att kunna sända större mängder, därför blev det en naturlig utveckling att integrera de två egenskaperna till en gemensam algoritm som senare kunde implementeras i ett kommunikationsverktyg.

1.2 Mål

Målet som eftersträvats under projektet har bestått av att empiriskt undersöka om det är möjligt att effektivt kombinera komprimering och kryptering i en gemensam algoritm.
Om en sådan algoritm är möjlig är tillkommer även ett andra mål nämligen att utveckla den resulterande algoritmen till ett stadie där den åstadkommer samma eller bättre tidskomplexitet som för nuvarande befintliga lösningar.
Ett tredje mål har varit att producera ett kommunikationsverktyg för att illustrera funktionaliteten av sådan algoritmen.

1.2.1 Eftersträvade mål

Mål som önskas uppnås är att:

- Skapa en kompressions/krypterings-algoritm som har en bättre tidskomplexitet än alternativet att utföra algoritmerna sekventiellt.
- Åstadkomma en kompressionsalgoritm för integration i slutresultatet som har ett värstafall som inte resulterar i att datamängden blir större än originaldatan.
- Åstadkomma en krypteringsalgoritm för integration i slutresultatet som är relativt säker.
- Skapa ett kommunikationsverktyg som har verkliga användningsområden utöver att fungera som proof of concept*

1.2.2 Krav

Dessa krav måste uppnås för att projektet skall räkna som lyckat:

- Skapa en kompressionsalgoritm som vid normalfallet* åstadkommer förminskning i datamängd.
- Skapa en kompressionsalgoritm som motsvarar LossLess* definitionen.
- Skapa en krypteringsalgoritm som gömmer klartexten till en sådan grad att klartexten ej går att utläsa om en angripare enbart har tillgång till krypterad data.
- Skapa en integrerad kompressions- och krypterings-algoritm som har samma tidskomplexitet som motsvarande algoritmer för kompression och kryptering hade haft om de utfördes sekventiellt.
- Skapa ett kommunikationsverktyg som implementerar den sammanslagna krypterings och kompressions-algoritmen.

1.3 Metod

Projektet utfördes under relativt fria premisser vad gäller tillvägagångssätt men den grundmetodik som användes var enligt följande:

1. Informationsinsamling och undersökning av befintliga algoritmer och metoder.
2. Analysera insamlad information efter idéer och uppslag för egna algoritmer
3. Använda kreativitet och utveckla egna idéer.
4. Implementera idéer från steg 3.
5. Testning och utvärdering av implementation från steg 4
6. Vidareutveckla och förbättra idéer och koncept från steg 3 och 4
7. Upprepa från steg 3 tills önskat resultat nås.

1.4 Presentationsmetod för resultat

Resultatet av projektet kommer att presenteras som tre distinkta delar där först kompressionen och krypteringen gås igenom i detalj följt av ett kapitel där integrationen mellan de två delarna gås igenom.
Detalj beskrivningarna av kompressions- och krypterings-algoritmerna beskriver först var eventuell inspiration och idéer hämtats från befintliga algoritmerna följt av en beskrivning om hur de utvecklades och detaljer om dess funktionalitet.
Kapitlet där integrationen mellan algoritmerna beskrivs bygger till stor del på information från kapitel 2 och 3 så här förekommer en del förenklingar och en del kunskap tas för given av läsaren då denna förmedlats i de två föregående kapitel.
Ett kapitel beskrivande Proof och concept programmen som skappades under rapporten beskrivningen är av en avskalad sort då källkoden är det verkliga resultatet av denna del av projektet.

2 Kompressionsalgoritmen LPRL (Lovén Persson Run-Length)

Detta kapitel ämnar klargöra kompressionsalgoritmens grundläggande struktur och design.
Algoritmen som redovisas i detta kapitel beskrivs utifrån sin grundidé och kommer i den mån det är möjligt att beskrivas fristående från den kryptografiska algoritm den vid ett senare stadium ämnas integreras med.

2.1 Undersökning av befintliga algoritmer

Grundidéerna som eftersträvades vid utvecklandet av algoritmen var att dess värstafall* skulle resultera i minimal expansion av datamängd* samt att dess tidskomplexitet skulle vara relativt låg.
De krav som ställdes på algoritmen var att den skulle vara en så kallad lossless* kompression, detta innebär att ingen data får förkastas eller gå förlorad vid kompression som är fallet med så kallade lossy* algoritmer. Utöver detta skall snabbhet prioriteras över kompressionseffektivitet i de fall dessa egenskaper står i konflikt med varandra.
Algoritmen som projektet resulterade i är en egenutvecklad design som hämtat inspiration och uppslag från befintliga kompressionsalgoritmer.
Nedan följer beskrivningar av de algoritmer som undersökts samt en redogörelse för vad som använts i utvecklingen av projektets algoritm.

2.1.1.1 Huffman Coding

”Huffman coding” på svenska kallat Huffmankodning fungerar genom en kombination av principen om binärträd och om adaptiv teckenlängd beroende på frekvensen i indata. Data beskrivs genom att ange en sökväg i ett binärt träd som pekar på en representation av den ursprungliga datan. Detta nyttjas för att komprimera data genom att binärkombinationer som är frekventa i indatan placeras högt upp i binärträdet vilket resulterande i korta sökvägar.
En av de stora fördelarna med Huffmankodning är att dess flexibilitet när det gäller olika typer av indata. Algoritmens kompressionseffektivitet påverkas i relativt låg grad av vilken typ av information datan representerar. Detta eftersom algoritmen drar nytta av alla upprepningar av binära mönster för att åstadkomma kompression, detta medan till exempel Run-lengthkodning (se 2.1.1.3) ger bäst resultat då förekomsten av lokala upprepningar är hög.

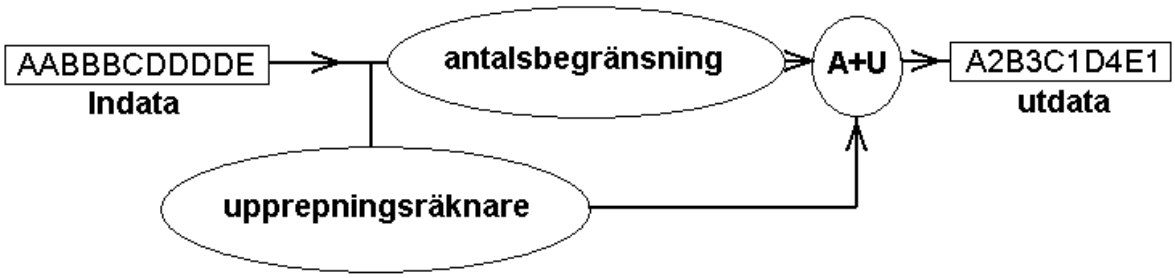
En nackdel med Huffmankodning är att dess tidskomplexitet är relativt hög i förhållande till exempel Run-lengthkodning.
För detta projekts ändamål har den dessutom den nackdelen att den inte passar bra för integration med kryptografiska funktioner . Detta på grund av att den skapar förutsägbara mönster i samband med komprimering, detta kan eventuellt användas för att kryptoanalysera utdata vilket kan medföra säkerhetsbrister i algoritmen som helhet.

En annan nackdel med Huffmankodning är att vid integration med en krypteringsalgoritm skulle algoritmerna uppträda sekventiellt i stället för parallellt som varit ett av målen med projektet. Anledningen till detta är att Huffmankodning bygger på att hela dokumentet* som skall komprimeras genomgår en frekvensanalys innan det mappas in i binärträdet. Dessutom behövs ett intakt binärträd under mappning av tecknen eftersom de som läggs in sist kodas med enligt sökvägar enligt de tidigare inlagda tecknen. Detta leder till att praktiskt taget hela komprimeringsalgoritmen behöver slutföras innan någon del av det kan krypteras.
Även avkodning av ett Huffmankodat träd kräver ett komplett binärträd för att datan skall vara tillgänglig, detta medför att total dekryptering krävs innan datan kan göras tillgänglig för avkomprimering.

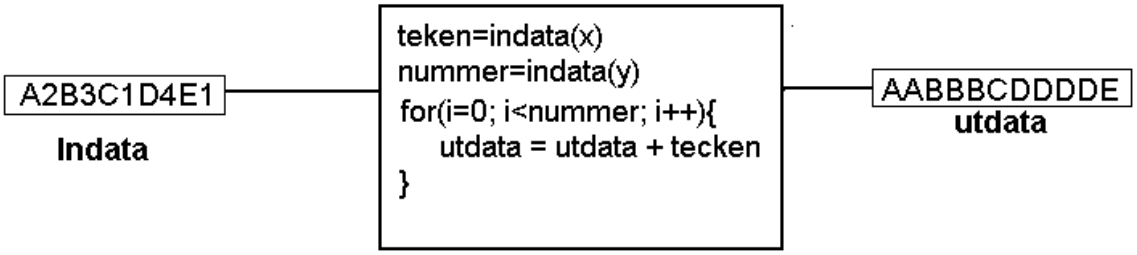
Resultatet från undersökningen av Huffmankodning var att den typen av komprimering inte lämpades för att tjäna projektets mål, detta främst på grund av en relativt låg flexibilitet när det gäller integration med kryptering.
Detta resultat kan med fördel även tillskrivas Shannon-Fano algoritmen då denna bygger på liknande principer dock med en annan metod för att representera och konstruera binärträdet.[1]

2.1.1.2 Matematisk kompression
Det finns ett antal olika typer av matematiska algoritmer för att komprimera data t.ex. Golombkodning, Levenshteinkodning och aritmetisk kodning.
De dessa har gemensamt är de bygger på någon form av matematisk algoritm som resulterar i en komprimerad representation av datan.
Denna typ av algoritm valdes i ett tidigt skedde bort som alternativ för projektet. Detta eftersom inga intressanta idéer rörande möjlig integration mellan denna typ kompressionsalgoritmer och kryptering framkom vid undersökning.[2]

2.1.1.3 Run-length kodning
Run-length kodning är en av de mest elementära formerna av kompression.
Den åstadkommer kompression genom att göra en omskrivning av hur binärkombinationer uppträder i direkt följd, omskrivningen görs genom att låta första binärkombinationen stå kvar medan efterföljande tecken uppger hur många gånger kombinationen skall upprepas vid avkomprimering.
Run-length kodning kan åskådliggöras enligt bilderna *2.1* för komprimering och *2.2* för avkomprimering



2.1. blockrepresentation av Run-length komprimering. A+U lägger en symbol från antalsbegränsningen följt av värdet av upprepningsräknaren i utdata representationen.



2.2. blockrepresentation av Run-length avkomprimering. Indata(x) representerar udda platser i indata medan indata(y) representerar jämna platser.

Run-length kodningens största fördel är dess låga tidskomplexitet som är O(n). En annan stor fördel set i förhållande till projektets mål är att dess utdata är på en form som är lämplig att utföra många av de kända typerna kryptering på.
En av de största nackdelarna med Run-lengths är dess uppträdande vid värsta fallet av indata då denna typ av kodning kan resultera i större utdata än indata. Detta kommer sig av att run-length måste representera även de binära mönster som enbart uppträder en gång med två platser i utdatan. Detta är fallet eftersom indata(x)-indata(y) förhållandet enligt 2.2 måste följas för att avkomprimering skall vara möjlig.
Run-length har till stor del legat till grund för den algoritm som utarbetades under projektet dock med en fundamental omstrukturering och nyutveckling, både i syfte att förbättra funktionalitet vid krypteringsalgoritmens integration och för att ge en ökad komprimeringseffektivitet.[3]

2.2 Utveckling av Kompressionsalgoritmen (LPRL)

För att undvika förvirring anges härmed den egenutvecklade algoritmen som LPRL(Lovén Persson Run-Length).

2.2.1 Nackdelar med run-length kodning

Som beskrivet i kapitel ovan valdes run-lengthkodning som grund för arbetet vid utveckling av LPRL.
Run-length i sin grundform har dock den oönskade egenskapen att den resulterar i dubblering av datamängdens storlek då indatan är av värsta fallet. Ett av

projektets mål var att skapa en algoritm där värstafallet resulterar i minimal expansion av datamängden. Detta medförde att LPRL krävde ett mer effektivt sätt att representera mängdinformationen i utdatan.

Problemet med run-length kodning är att en plats i utdatan krävs för att representera tecknet eller bitmönstret från indatan följt av en plats för att ange hur många iterationer av tecknet som följer i rad.
Detta kan resultera datamängds expansion enligt exemplet nedan där ”Utdata(x)” anger platsen x i utdata representationen.

```
Indata=|A|B|C|D|E|F|

Steg 1: Utdata(0)=Indata(0)= A & Utdata(1)=1
Steg 2: Utdata(2)=Indata(1)= B & Utdata(3)=1
Steg 3: Utdata(4)=Indata(2)= C & Utdata(5)=1
Steg 4: Utdata(6)=Indata(3)= D & Utdata(7)=1
Steg 5: Utdata(8)=Indata(4)= E & Utdata(9)=1
Steg 6: Utdata(10)=Indata(5)= F & Utdata(11)=1

Utdata= |A|1|B|1|C|1|D|1|E|1|F|1|
```

I exemplet ovan åskådliggöras hur varje tecken som inte upprepas resulterar i att uppta två platser i utdatan, detta trots att de endast upptog en plats innan datan behandlades med algoritmen. Av exemplet framgår att värstafallet ger en expansion av indatas längd multiplicerat med två.
Är fallet sådant att en del tecken i indatan är upprepade i direkt följd räknas detta som normalfallet. Detta kan ge resultat av typen illustrerade i följande exempel.

```
Indata=A|B|B|B|C|D|D|D|E|F|F|F|F|

Steg 1: Utdata(0)=Indata(0)= A & Utdata(1)=1
Steg 2: Utdata(2)=Indata(1)= B & Utdata(3)=3
Steg 3: Utdata(4)=Indata(4)= C & Utdata(5)=1
Steg 4: Utdata(6)=Indata(5)= D & Utdata(7)=3
Steg 5: Utdata(8)=Indata(8)= E & Utdata(9)=1
Steg 6: Utdata(10)=Indata(9)= F & Utdata(11)=4

Utdata= |A|1|B|3|C|1|D|3|E|1|F|4|
```

Optimalfallet vid användning av run-length är när all data är en upprepning av första tecknet eller binär mönster, denna typ av fall är dock orealistiska då de i princip aldrig representerar någon verklig information.

Även här ser man att onödiga data sparas vid kompression, nämligen den information som talar om att ingen iteration förekom som i fallet med A, C och E i exemplet ovan.

2.2.2 Effektiviserande av datarepresentationen

Exempelen i 2.2.1 visar tydigt på Run-lengthkodnings nackdel i att lagra onödig information.
Det som krävs för att effektivisera lagringen informationen är att skala bort den datan som inte är nödvändig för att återskapa original informationen.

2.2.2.1 Avskalning av onödiga representationer

Projektets första lösning på detta problem kallat LPRLP1 (**L**ovén **P**ersson **R**un-**L**ength **P**rototyp 1) bestod av att iterationsangivelser inte sparades i de fall de inte tillför något vid avkomprimering, alltså i de fall då tecknet inte upprepades eller endast en teckenupprepning förekom.
Dock innebar detta att avkodning med normal run-length teknik inte längre var möjlig. Detta på grund av att upprepningar ej längre stämmer överrens med original datan. Problemet kan åskådliggöras genom exempelet nedan.

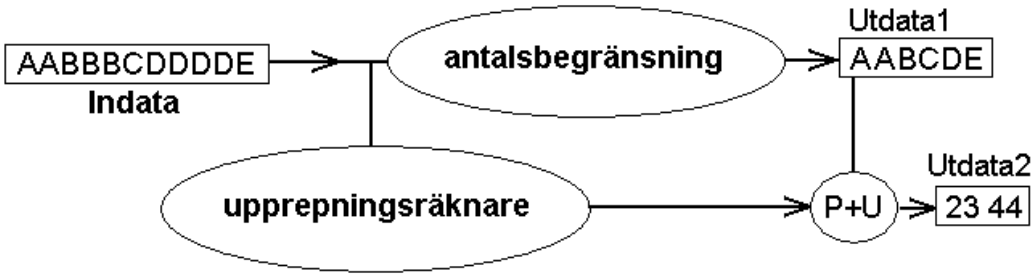
```
Indata=A|B|B|B|C|D|D|D|E|F|F|F|F|

Steg 1: Utdata(0)=Indata(0)= A & Utdata(1)=Null
Steg 2: Utdata(2)=Indata(1)= B & Utdata(3)=3
Steg 3: Utdata(4)=Indata(4)= C & Utdata(5)=Null
Steg 4: Utdata(6)=Indata(5)= D & Utdata(7)=3
Steg 5: Utdata(8)=Indata(8)= E & Utdata(9)=Null
Steg 6: Utdata(10)=Indata(9)= F & Utdata(11)=4

Utdata= |A|B|3|C|D|3|E|F|4|

Avkomprimering utdata:
Indata = utdata
Steg 1: Utdata(0)=Indata(0)=A som upprepas Indata(1) gånger
=B (Felaktig inläggning);

Steg 2: Utdata(1)=Indata(2)=3 som upprepas Indata(3) gånger
=C (Felaktig inläggning);

Steg 3: Utdata(2)=Indata(4)=D som upprepas Indata(5) gånger
=3 (korrekt inläggning);

Position i utdata räknas fram då steg 3 gav tre
iterationer;

Steg 4: Utdata(6)=Indata(6)=E som upprepas Indata(7) gånger
=F (Felaktig inläggning);

Steg 5: Utdata(7)=Indata(8)=4 som upprepas Indata(9) Kan ej
slutföras då Indata(9) saknas;
```

Utdata= |A|3|D|D|D|E|4| Detta stämmer uppenbarligen ej med ursprungliga indatan.

Detta problem löstes genom att avkomprimering sker under premissen att enbart lägga till iterationer i de fall tecknet följs utav en siffra. Därigenom ges tecken sin ursprungliga innebörd och korrekt position i utdatan, tack vare detta kan den ursprungliga indatan avläsas ur den komprimerade representationen.
Detta ålägger dock algoritmen med en oönskad begränsning, nämligen att enbart teckensträngar kan komprimeras med LPRLP1. Dessutom krävs det att tecken ej kan tolkas som siffror.
Dessa krav på indatan reducerar kraftigt algoritmens användbarhet då enbart väldigt specifik indata kan hanteras.

2.2.2.2 Effektivisering av representation

För att komma till rätta med LPRLP1s problem med att den begränsa vilken form indata var utvecklades LPRLP2.
Problemet med LPRLP1 var att det är omöjligt att skilja på de binärmönster som representerar tecken och de som representerar information om antal iterationer om dessa är binärt lika. Ett sätt att göra skillnad på innehåll och iterationsangivelser är att placera dem enligt en konvention som är fallet med ursprungliga run-lengthkodningen. Lösningen som används i LPRLP2 skiljer sig dock mot ursprungs run-lengthkodning genom att iterationsangivelser skrivs i en separat datamängd skild från innehållsrepresentationen. För att separation skall vara möjlig krävs det dock att två värden används för att skapa en iterationsangivelse, ett värde för att ange vilken position och ett värde för att ange antal iterationer, se exempelet ovan.
Detta sätt att skriva gör det möjligt att kombinera run-lengths mångsidighet vad de gäller indata med LPRLP1s sätt att skala bort onödig information.



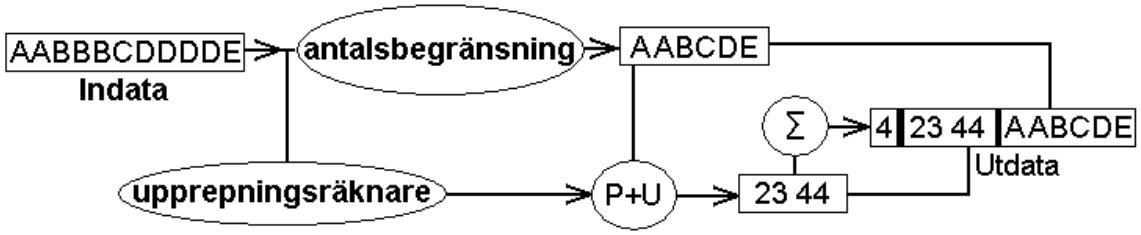
2.3 diagram av LPRLP2, P+U lägger placeringsnummer av senast tillagda tecken i Utdata1 följt av antal iterationer den skall återskapas med i Utdata2.

LPRLP2 kräver att två värden lagras i utdata2 samt ett i utdata1 för att representera en iteration. På grund av detta behövs det tre upprepningar för att en omskrivning skall tjäna något syfte då två upprepningar upptar mer utrymme vid omskrivning än de gjort om det stått i sin grundform.
Nackdelen med LPRLP2 är att utdatan måste lagras i två separata datarepresentationer. Detta är teoretiskt inget problem, men i programspråksmässig implementationer där objekt måste skapas för båda delar medför LPRLP2 en del onödig overhead*. Även vid sändning i nätverk uppkommer en del onödig overhead då två paket måste skickas.

2.2.3 Utdatakonvention

Problemet med dubbel lagring i LPRLP2 kan undvikas genom att lagra båda datarepresentationer som en. Dock återkommer då problemet som fanns med LPRLP1 att en dator ej kan skilja på iterationsangivelser och verklig information.

För att undvika detta problem krävs att någon form av konvention används som definierar vilken del som skall tolkas som iterationsangivelser och vilken som är informationen.
Detta löses i sista testprototypen LPRLP3 genom att ett tal läggs i början för att definiera hur många av de följande tecknen som skall tolkas som iterationsangivelser.



(∑) räknar antalet siffror/tal som läggs till bland iterationsangivelserna, detta tal läggs sedan först i utdatan.

Nu kan avkomprimering ske genom att plocka ut det första talet ur datarepresentationen och där efter dela den återstående datamängden vid indexet angivet av det första talet. Detta kan åskådliggöras enligt följande diagram *2.4*.

```
Indata=4|2344|ABCDE

Steg 1: Läs tal Indata(0) och ta bort platsen från Indata;

Steg 2: Dela återstående Indata vid index av det som lästes
från tidigare Indata(0):
Data1=2344, Data2=AABCDE;

Steg 3: Notera första plats som skall itereras, bestäms av
Data1(0), notera antal iterationer som skall göras på denna
plats, anges i data1(0+1);
```

2.4. Fortsätter på nästa sida

```
Steg 4: Flytta tecken från data2 till slutet av utdata, gör
detta tills index angiven av föregående steg träffas:
Utdata=AA;

Steg 5: Flytta de tecken som skall itereras från Data1 till
utdata och utför iterationerna: B itereras tre gånger,
Utdata = AABBB;

Steg 6: Notera nästa plats som skall itereras, anges av
Data1(2), samt antal iterationer som anges av Data1(2+1);

Steg 7: Flytta tecken från data2 till slutet av utdata, gör
detta tills index angiven av föregående steg träffas:
Utdata=AABBBC;

Steg 8: Flytta de tecken som skall itereras från Data1 till
utdata och utför iterationerna: D itereras fyra gånger,
Utdata = AABBBCDDDD;

Steg 10: Flytta tecken från data2 till slutet av utdata,
gör detta tills index angiven av föregående steg träffas
(finns inget): Utdata=AABBBCDDDE;


Utdata = AABBBCDDDE
```

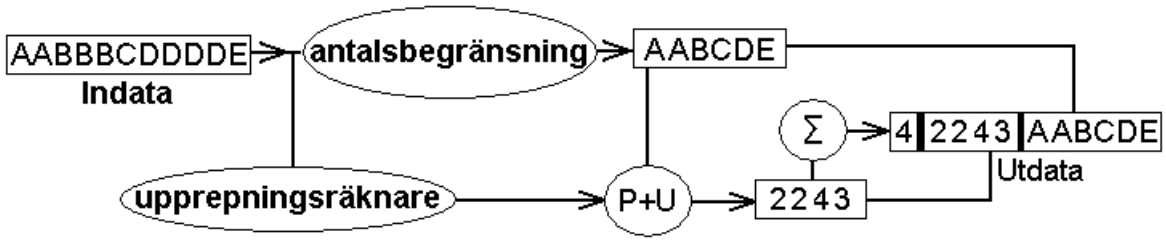
2.4. Beskriver avkomprimering av komprimerad datarepresentation 6132344ABCDE med LPRLP3 algoritmen.

2.2.4 Slututveckling av LPRL

Komprimering med LPRLP3 sker med samma tidskomplexitet som slutversionen LPRL nämligen O(n).
Detta kommer sig av att skapandet av de två utdatadelarna(teckentabellen och iterationsangivelserna) görs med O(n) då endast en övergripande genomgång av indatan görs.
Avkompression sker i LPRLP3 med tidskomplexitet av cirka $T(n)\approx 1.5n$ då ett antal mindre upprepningar sker då tecken skall itereras. Detta skrivs enligt ordoberäkningsreglerna som O(n).
En ändring som gjordes vid övergång från LPRLP3 till LPRL var hur iterationer representeras, från att beskriva antalet upprepningar till att beskriva antalet tecken som skall läggas till efter den som hämtas ur teckentabellen. Till exempel betydde siffran 3 som iterationsangivelse tidigare: AAA men fick i LPRL betydelsen: A+AAA.
Denna ändring gjordes eftersom avkomprimering då kan ske mer linjärt. Data hämtas då alltid först ur representationsmängden (kallad data2 i exempel *2.3*) och läggas i utdatan, för att sedan följas av ett test för att avgöra om den skall itereras. Om iteration skall ske görs detta genom att lägga till samma symbol sist i utdata igen. Tidigare skedde test innan flyttning av data från teckentabell till utdata och olika metoder kördes beroende på om iterationer skall ske eller ej.
Denna ändring gjorde att avkompression i princip sker med $T(n)\approx n = O(n)$ som tidskomplexitet.

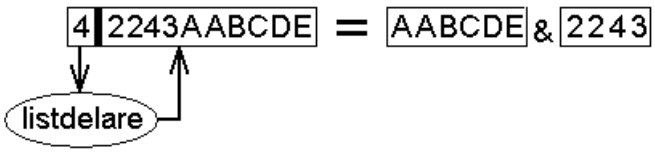
2.3 Sammanfattad beskrivning av LPRL

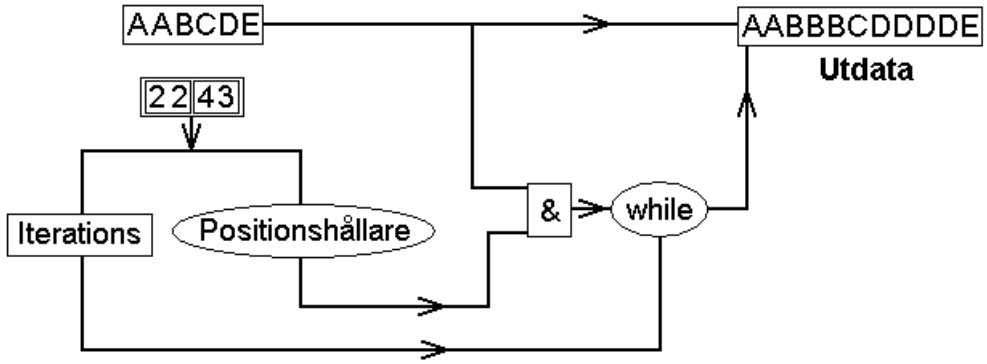
Algoritmen representerar upprepningar i indatan genom att enbart ta med det upprepade tecknet en gång i utdatan och istället ange antalet upprepningar i en egen tabell.
Denna tabell sammanfogas därefter med utdatan varpå ett tal placeras i början på den sammanslagna utdatan. Detta tal anger hur många platser som skall tolkas som tabell för iterationsangivelser, detta gör att iterationsangivelserna och teckeninformationen kan separeras.



Förenklad beskrivning av LPRL.

För att avkomprimera genomgås listan innehållande teckenangivelser igenom och dess innehåll skrivs över till utdatan, när en plats som är markerad för iteration träffas i teckenlistan så läggs tecknet på denna platsen till i utatan så många gånger extra som iterationsangivelsen beskriver.
I exemplet nedan används utdatan från föregående exempel som indata.





Förenklad beskrivning av LPRL avkomprimering. While lägger till symbolen från teckenlistan så många gånger som iterationsangivelsen anger. While metoden aktiveras av OCH grinden styrd av teckenlistan och iterationsangivelserna.

2.4 Implementation av LPRL

2.4.1 LPRL kompression i pseudokod

```
senaste = första tecknet i indata;
första platsen i utdataTecken = första tecknet i indata;
antalSenaste = 0;
i=1;
så länge i<Indatans längd {
        Om senaste = Indata(på platsen i) görs{
                antalSenaste = antalSenaste +1
        }
        Om senaste inte =  Indata(på platsen i) &
        antalSenaste>0 görs{
                Ta reda på nummer representerande platsen
                senaste fick i utdataTecken och placera i
                utdataIterationer;

                placera antalSenaste i utdataIterationer;

                Uppdatera till den nya ”senaste” =
                Indata(på platsen i);
                antalSenaste=0;
                Indata(platsen i) placera sist i
                utdataTecken;
        }

        Om senaste inte = Indata(platsen i) &
        antalSenaste=0 görs{
                placera Indata(platsen i) sist i
                utdataTecken;
                Uppdatera till den nya ”senaste” =
                Indata(på platsen i);
                antalSenaste=0;
        }
        i=i+1;
}
```

Exemplet 4.1 fortsätter på nästa sida

```
Om antalSenaste>0 görs{
                  Ta reda på nummer representerande platsen
                  ”senaste” fick i utdataTecken och placera
                  resulteat i utdataIterationer;

                  placera antalSenaste” resulteat i
                  utdataIterationer;
}


Räkna ut antalet tecken i utdataIterationer, och lägg
värdet på första platsen i Utdata;
Lägg utdataIterationer med start på den andra platsen i
Utdata;
Lägg utdataTecken efter utdataIterationer i Utdata;

Utdata är nu komplett.
```

Exemplet 4.1

2.4.2 LPRL avkompression i pseudokod

```
Ta talet på första platsen av indatan;
Dela återstående indatan vid positionen angiven av talet
uttaget i steget innan,
delarna kallas Iterationslista och Teckenlista;
Ta bort första platsen i iterationslistan och lägg i
IterationsPosition;
Ta bort första platsen i iterationslistan och lägg i
IterationsAntal ;
i=0;
så länge i<Teckenlistans längd {
         Flytta och ta bort det främst placerade teckent
         från Teckenlista och placera det i Utdata;
         Om IterationsPosition = i görs{
                  Lägg till de senast hämtade ur
                  Teckenlistan i Utdata igen, gör detta
                  IterationsAntal gånger;
                  Ta bort första platsen i iterationslistan
                  och lägg i IterationsPosition;
                  Ta bort platsen efter i iterationslistan
                  och lägg i IterationsAntal ;
         }
         i = i +1;
}
Utdata är nu komplett
```

2.4.3 Hjälpalgoritmer

LPRL opererar som tidigare nämnt genom att representera upprepningar på ett effektivare sätt, det medför att indata med en hög grad av upprepning komprimeras effektivt.
Detta medför att data där lossy kvalité är acceptabel med fördel kan behandlas med en algoritm som likställer områden där informationen har ett lägre värde.

2.4.3.1 Ljudbehandling

En enkel hjälpalgoritmsprincip för ljud är att enbart detaljkoda områden med amplitud som ligger inom mänskliga hörselspektrumet. Enligt samma princip räknas enstaka spikar i ett område av annars tyst data som brus och kodas därför om till att vara tyst.
Dessa områden kan alltså skrivas till värdet 0 i utdatan, resulterande i större spann av samma indatavärde när informationen sedan körs med LPRL-algoritmen.

2.4.3.2 Specialimplementation av LPRL för Javaljud

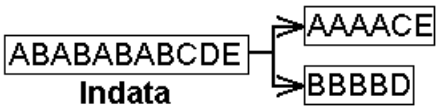
En speciell algoritm utvecklades under projektet för att bättre fungera med det sätt som Java hanterar ljudinspelning. Inspelning i Java lagras i bytearrays* som är datamängder med åtta bitar per position. Vid sexton bitars sampel kodas nyansskillnader i de första åtta bitarna och amplituden i resterande åtta, när sedan LPRL används på bytearrayn ger de i princip ingen kompression då ljudnyans och amplitud sällan är lika.

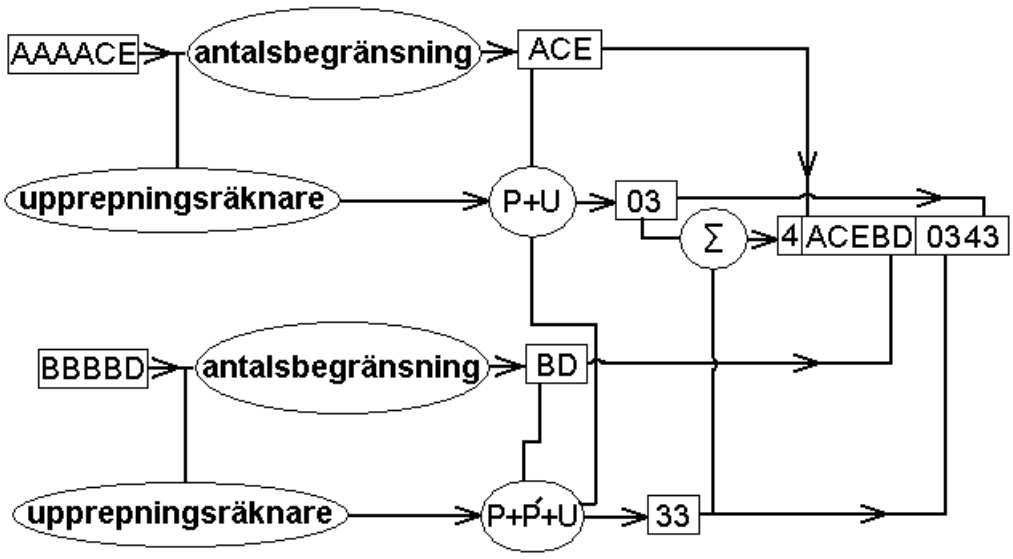
Därför utvecklades en speciell version av LPRL för implementation vid användning av Javaljud.

Indatan behandlas med den normala LPRL-algoritmen med skillnaden att den gås igenom två gånger. Data hämtas från varannan plats ur indatan vid vardera genomgång och andra genomgången startar med en positionsförskjutning.

På så sätt åstadkoms en gemensam iterationsangivelsetabell eftersom de två genomgångarna kodats sekventiellt. Det bildas dessutom enbart en teckentabell eftersom andra genomgången börjar placera tecken efter de sista angivna från den första genomgången.

För att avkomprimera genomgås datan som normal en LPRL-avkompression. Datan ges sedan sin korrekta plats i utdatan genom att dela den genomgångna datan symmetriskt i två delar, varpå varannan plats i utdatan hämtas från vardera av de två delarna.





Dubbel kompression anpassad till Javaljud. P+P'+U lägger placeringsnummer av senast tillagda tecken i utdata. Om algoritmen är på andra genomgången av indatan hanterar P´ de tecken som lades till föregående genomlöpning av indata för att ge ett korrekt värde på iterationsangivelsen.

2.4.3.3 Hjälpalgoritm för bildbehandling

En algoritm utvecklades under projektet för att testa principen om LPRLs effektivitet med förbehandling. Denna algoritmen var inriktad på bilder och gavs namnet LPIS (**L**ovén **P**ersson **I**mage **S**moothing).
Den fungerar enligt principen att likforma områden där informationsvariationen är under en förspecificerad nivå. Informationen behandlas som block tagna ur de tre färgkanalerna som bygger upp en bild.

De tre färgkanalerna som bygger upp en bild behandlas som block där en bestämd mängd data hämtas ut från vardera kanal och analyseras.
Analysen av blocken går till så att ett referensvärde räknas ut för varje kanal, detta värde räknas ut genom att ta snittet från de pixlar av den specifika färgkanalen som ligger i centrum av blocket. Alla pixlar inom blocket gås sedan igenom och jämförs med referensvärdet för sin respektive färgkanal, referensvärdet och pixlen jämförs med en marginal som bestäms av en inparameter till algoritmen. Ligger ett förspecificerat antal pixlar inom marginalgränsen från referensvärdet så likställs blocket, likställning innebär att alla pixlar skrivs om till kanalens referensvärde.
Om för många pixlar av blocket ligger utanför tröskeln delas blocket ned i mindre block. Dessa mindre block kontrolleras sedan var för sig utifrån samma referensvärde som det stora blocket. Detta föra att skapa en återkoppling och enbart utelämna de block som innehåller detaljer från likställning.

Efter behandling med LPIS används LPRL-algoritmen på varje färgkanal för sig, dessa kan sedan returneras var för sig. Skall kanalerna returneras som en gemensam datarepresenation kodas dessa efter varandra enligt ”röd grön blå” följden. Eftersom höjden och bredden på bilden är kända innan LPIS körs är det enkelt att bestämma avgränsningar eftersom avkomprimering av en kanal slutar då bredden multiplicerat med höjden antal tecken avkodats.
Skall bilden kodas enligt ett mönster som skall kunna skickas och avkodas utan ursprungs datan läggs bredd och höjd angivelserna först i datamängden.

3 Krypteringsalgoritmen LPCR (Lovén Persson Crypto)

Tanken var att integrera LPCR i SCAM algoritmen men den ska också fungera på sin egen hand.
Krav på LPCR har varit att den ska vara snabb nog för att kunna strömma data över Internet och självklart också att den ska vara tillräckligt säker för att kunna skicka känslig data med hjälp av den.

För att uppfylla kravet på snabbhet så har vi valt att skapa en symmetrisk krypteringsalgoritm och inte en asymmetrisk krypteringsalgoritm.

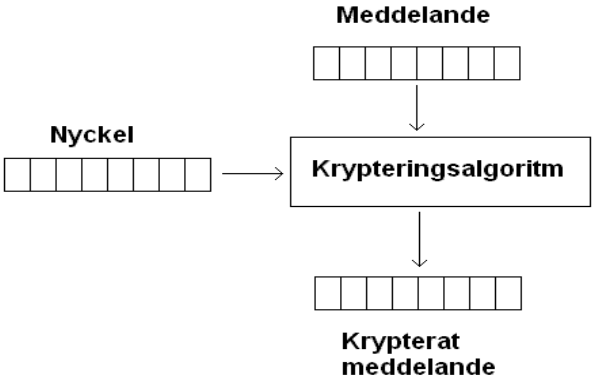
3.1 Bakgrund och idéer

Det finns två olika typer av kryptering när det gäller överföring av information. Det är symmetrisk kryptering och asymmetrisk kryptering.
Symmetrisk kryptering innebär att båda parter har samma nyckel medan asymmetrisk kryptering använder olika nycklar för kryptering och dekryptering. Normalt är symmetrisk kryptering mer tidseffektiv än asymmetriska algoritmer, detta beror oftast på att matematiken i asymmetriska algoritmer är mer beräkningsmässigt komplexa. Ett av målen med LPCR var att den skulle vara snabb nog för att kunna strömma data och det var därför enkelt att välja en symmetrisk krypteringsalgoritm och arbeta vidare utifrån det.

3.1.1 Blockchiffer

Blockchiffer är en kategorisering av chiffer som krypterar en bestämd längd av data. Oftast 64 eller 128 bitar åt gången[6]. För att kunna kryptera meddelanden av godtycklig längd så får man använda olika operationstyper. Normalt fungerar blockchiffer enligt principen att de läser in den okrypterade datan tillsammans med den nyckel som skall användas vid krypteringen. Blockskiffret utför sedan någon form av krypteringsoperation på parametrarna vilket resulterar i krypterad utdata.
Ett problem som uppstår är när indatan inte går att dela med antalet bitar som varje block krypterar. Då får någon form av utfyllnad göras på det sista blocket.

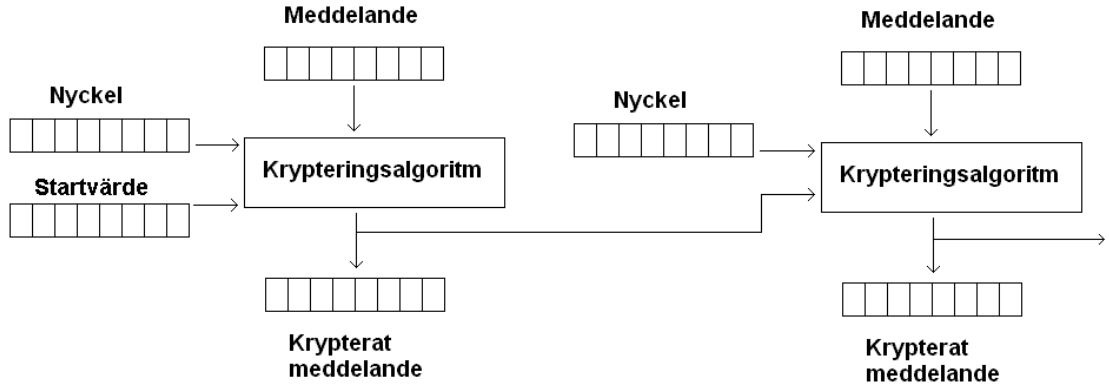
Den enklaste formen av blockchiffer är Electronic Codebook (ECB)[8] vilket figur 1 visar.



Figur 1

Detta är ingen bra kryptering eftersom samma meddelande ger samma utdata och man kan därför analysera fram det okrypterade meddelandet. Upprepningar av indata ger en viss antydan om vad originalinformationen bestod av.
Denna typ av kryptering används i princip aldrig eftersom de är relativt trivial att kryptoanalysera.

För att undvika upprepningar så nyttjas vanligen någon form av återkoppling vid kryptering. Denna typ av lösning kallas för Cipher-Block Chaining (CBC)[8].



Figur 2

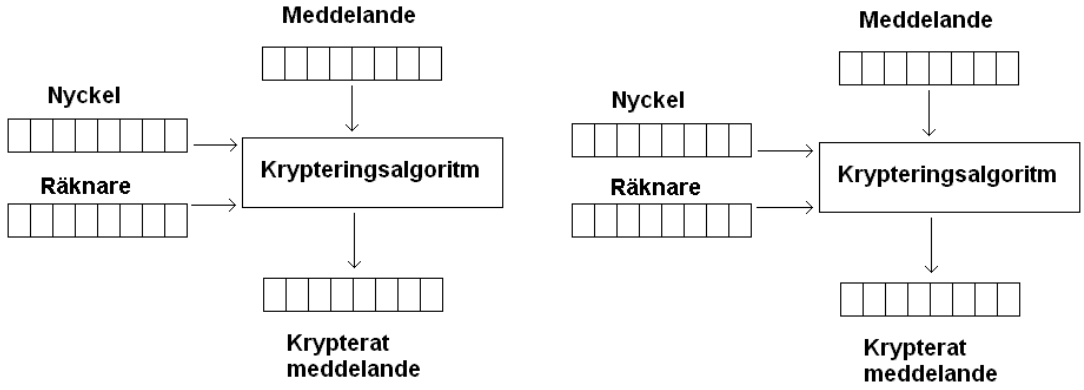
I denna typ av chiffer så beror utdatan av en blockoperation delvis av en parameter baserad på utdata från föregående blockoperation. Algoritmen tar nu utöver meddelandet och nyckeln även in en extra parameter. Vid allra första krypteringen så skickar man in ett specifikt startvärde som extra parameter till algoritmen medan alla efterföljande krypteringar tar in en parameter baserad på det föregående krypterade meddelandet.
Se figur 2.

Denna typen av blockchiffer ger inte ut samma krypterade meddelande vid samma meddelande vilket gör att det blir mycket mer komplext att kryptoanalysera utdatan. Detta gäller även vid kryptoanalys rörande stora

mängder av krypterad data eftersom återkopplingen påverkas av alla tidigare block.
Även attacker ämnade att analysera fram krypteringsnyckeln försvåras avsevärt. Om en attackerare har tillgång till klartexten och kryptotexten så blir analysen inte enklare. Dock så beror säkerheten till stor del på hur säker själva blockalgoritmen är.

En CBC typ blockchiffer var en möjlig kandidat som modell för krypteringsalgoritmen i detta projekt. Dock så kräver den att mottagaren tar emot alla datapaketen och i rätt ordning. Det framstod senare att detta inte var en så bra egenskap då algoritmen skulle användas i ett chatprogram.

En tredje typ av blockchiffer är Counter (CTR)[8]. I stället för att låta nästa block beror på data från föregående block så används en form av räknare som extra parameter till algoritmen.



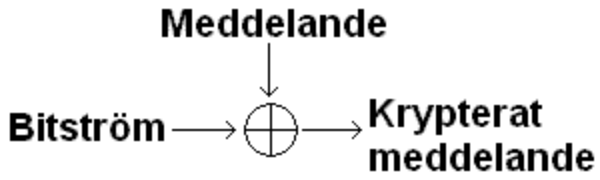
Figur 3

Räknaren behöver inte nödvändigtvis räkna upp tal i sekventiell ordning utan kan till exempel vara ett slumpmässigt tal. Denna typ av kryptering uppfyller kraven att inte behöva ta emot alla paket eller att behöva ta emot dem i rätt ordning. Istället krävs det enbart att mottagaren vet vilket nummer som meddelandet krypterades med.
Denna typen av blockchiffer blir i verkligheten relativt likt ett strömchiffer. Skillnaden är att ett blockchiffer utför sin operation på en bestämd längd av bitar medan ett strömchiffer opererar på det minsta antalet bitar som kan skickas åt gången.
Blockchiffer enligt Counter modellen användes som en av grundidéerna till LPCR.

3.1.2 Strömchiffer

Strömchiffer har som egenskap att de är snabba att kryptera och avkryptera.
De används ofta till att skicka strömmande data där mängden data ej är känd. Därför kallas denna typ av chiffer just för strömchiffer.

Till skillnad från blockchiffer som krypterar ett specifikt antal bitar varje gång så krypterar strömchiffer bara så många bitar som behövs för att representera den minsta informationen som kan skickas. Vanligast är att strömskiffret tar en klartextsymbol som indata åt gången.
En strömchifferalgoritm använder en generator som har krypteringsnyckeln som parameter. Generatorn har som uppgift att skapa en pseudorandom bitström. Bitströmmen ska sedan användas för att kombineras med klartextmeddelandet och därmed kryptera klartextmeddelandet. Vid avkryptering så kombineras bitströmmen med det krypterade meddelandet och får då ut klartextmeddelandet. Den vanligaste typen av kombinationsoperation är en XOR funktion. Som figur 4 visar görs XOR mellan bitströmmen och klartexten och resultatet blir det krypterade meddelandet.



Figur 4

Denna typ av kryptering kräver en viss synkronisering mellan sändare och mottagare då båda parter måste vara på samma position i bitströmmen för att kunna kryptera och avkryptera meddelandet.
En XOR funktion är tidseffektiv, säker och lätt att implementera. Kryptering sker enligt figur 4 och avkryptering sker på samma sätt förutom att i stället för att ta in ett meddelande i klartext och ge ut det krypterade meddelandet så tas krypterad data in istället för klartext, och de som tidigare var krypterad utdata resulterar nu i klartext. (se figur 5) Mottagaren använder sin nyckel som ska vara identisk med sändarens symmetriska nycklar och kan därmed generera en bitström som är densamma som den sändaren använde vid krypteringen.
En XOR operation kan också implementeras i blockchiffer och användas till den delen som skall gömma själva meddelandet, d.v.s. den kan användas som sista steget i algoritmen och resultatet från XOR-operationen är det krypterade meddelandet som är redo att skickas.

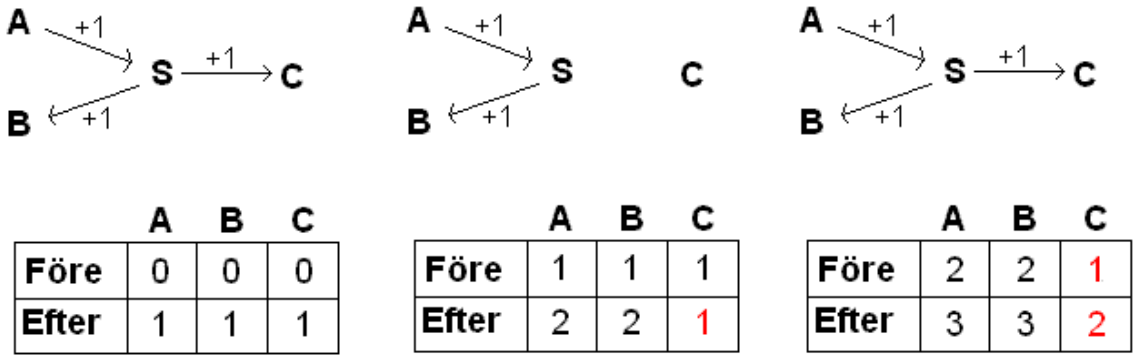
Figur 5

Eftersom ett av kraven är att inte behöva någon synkronisering mellan sändare och mottagare, så har strömchiffer inte kunnat användas rakt av. Dock har XOR-

operationen visat sig vara enkel och snabb nog för att uppfylla kraven på LPCR-algoritmen. Därmed så blev XOR-operationen utvald som en möjlig del i LPCR.

3.2 Utveckling av LPCR

De första idéerna till LPCR var att använda XOR metoden eftersom den är snabb vilket är ett av kraven på den slutgiltiga algoritmen. En typ av strömchiffer var uppe som förslag i början.
Vid närmare undersökning så blev det uppenbart ganska snabbt att det blir problem med synkroniseringen när algoritmen ska användas till att kryptera konferenssamtal i en chatklient och klienterna även kan skicka privata meddelanden samtidigt.



	A	B	C
Före	0	0	0
Efter	1	1	1

	A	B	C
Före	1	1	1
Efter	2	2	1

	A	B	C
Före	2	2	1
Efter	3	3	2

Figur 6. Synkroniseringproblemet med konferenschat

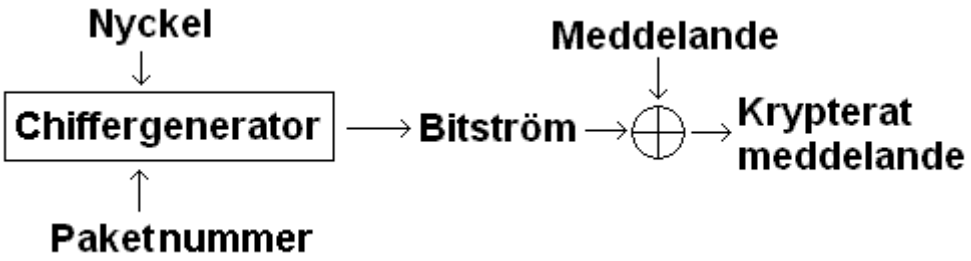
A, B och C har öppnat en chatkonferens med varandra och alla har samma krypteringsnyckel och alla använder ett strömchiffer för att kryptera alla meddelanden som skickas.
A räknar upp sin räknare innan han krypterar meddelandet med det värdet hans räknare står på. B och C räknar upp sina räknare innan de avkrypterar sina mottagna meddelanden.
Så länge som A skickar meddelande som når fram till både B och C så är synkroniseringen inget problem. I de fallet har alla samma uppräkningssiffra. Varje gång någon skickar ett meddelande eller tar emot ett meddelande så räknar klienten upp sin räknare med 1. Problem uppstår när A skickar ett privat meddelande till B samtidigt som konferensen pågår och att de skickar det privata meddelandet med samma nyckel och algoritm. Vad som händer då är att alla andra klienter tappar synkroniseringen med de klienterna som skickade privat. Som figur 5 visar så hamnar de andra klienterna efter i sin uppräkning. När nu A skickar ett vanligt meddelande till alla klienter så kan inte C längre dekyptera meddelandet. C försöker avkryptera med nummer 2 medan meddelandet har krypterats med nummer 3. C kan heller inte skicka nya meddelanden till varken A eller B eftersom A och B kommer att försöka avkryptera med nummer 3 när meddelandet har krypterats med nummer 2.

Ett annat problem med synkroniseringen vid konferenser är om två klienter råkar skicka var sitt meddelande exakt samtidigt. De båda meddelanden blir då krypterade med samma paketnummer i synkroniseringen medan mottagarna kommer att försöka avkryptera de båda meddelanden med olika paketnummer. Anledningen är eftersom mottagarna kommer att räkna upp sina räknare ett steg per mottaget meddelande när i verkligheten båda meddelanden har blivit krypterade med samma paketnummer. Det kan också uppstå problem med ordningen som meddelanden kommer fram om man använder till exempel Internet för att skicka meddelanden. Beroende på avståndet mellan de olika klienterna så kan det hända att även om A skickar ett meddelande före B, så kan det inträffa att C tar emot meddelandet från B före meddelandet från A. C kommer då att försöka avkryptera meddelandet från B med fel nummer och kan inte avkryptera meddelandet.

Lösningen på detta problem var att skapa en form av kryptering som inte är beroende av synkronisering. I alla fall inte på den nivån som strömchiffer behöver.
Lösningen på problemet är inspirerad av räknande (CTR)[8] blockchiffer, d.v.s. att den har ett specifikt nummer som används vid kryptering och avkryptering. (se kapitel 2.1.2) Resultatet är en algoritm som är en mix mellan ett strömchiffer och ett räknande blockchiffer. Den använder en XOR operation mellan meddelandet och en bitström som genereras av en matematisk algoritm. Det stora problemet som skulle lösas var att skapa en algoritm som skall generera bitströmmen till XOR-operationen.

3.2.1 Prototyper
Ett krav på generatorn är att den ska ge ut en bitström som är åtminstone ett specifikt antal bitar. Det är också krav på bitströmmen att den ska se ut att vara en mängd slumpvisa tal. Det finns också krav på säkerhet men det tas upp i kapitel 2.3.3.

3.2.1.1 LPCRP1 (Lovén Persson Crypto Prototype 1)

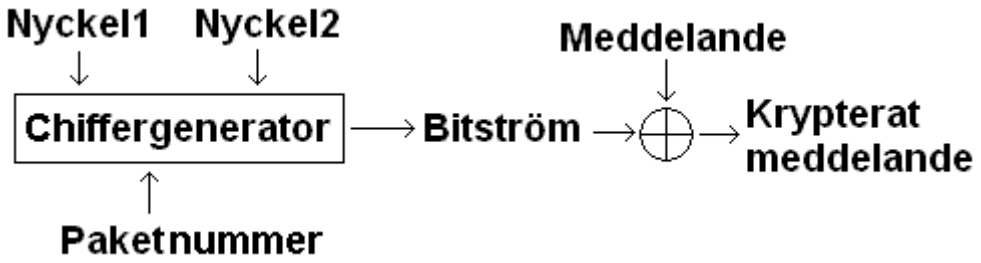


Figur 7

Paketnummer är tänkt att skickas tillsammans med meddelandet och den ska visa hur meddelandet ska avkrypteras. Paketnummer fungerar som en automatisk

synkronisering genom att mottagaren bara skickar in paketnummer till sin algoritm och därmed kan avkryptera utan att behöva bry sig om någon form av synkronisering.
Denna version fungerade inte alls på grund av att matematiken i generatorn inte kunde fungera på ett säkert sätt och ge den bitström som behövdes. För att uppfylla säkerhetskravet så skall en *x^y mod z* funktion användas. (se kapitel 3.3.3) Problemet är att generatorn bara har två parametrar, vilket figur 7 visar, medan funktionen behöver tre parametrar. Därmed kunde inte denna version testas och blev aldrig implementerad.

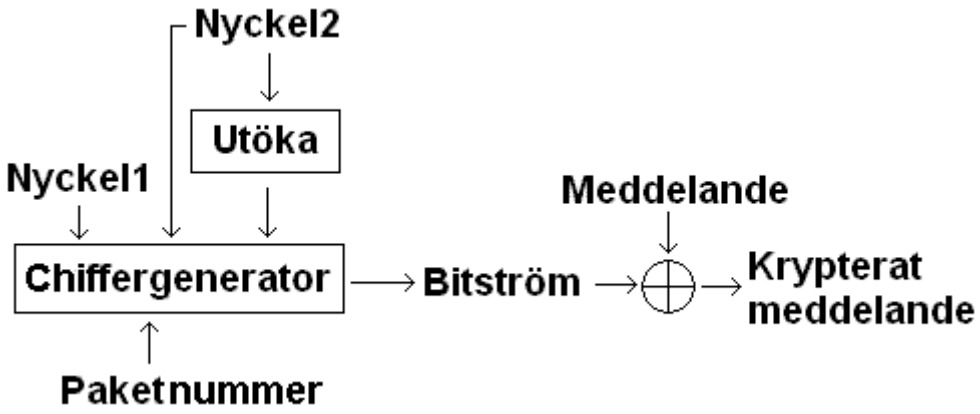
3.2.1.2 LPCRP2 (Lovén Persson Crypto Prototype 2)



Figur 8

Denna version fixade problemet med att matematiken behövde mer inparametrar genom att använda två stycken nycklar istället för en enda nyckel. Det går förstås att säga att det bara finns en nyckel men att den behöver delas i två delar innan den ska användas. De första testerna visade att denna version fungerade bra och gav ut en bitström som såg ut som en slumpvis bitström. Tyvärr så visade ytterligare tester att LPCRP2 inte alls gav den bitström som de första testerna visade och att de testerna inte gjordes på att korrekt sätt.
I de första testerna så kontrollerades inte bitströmmen på bitnivå utan bara på decimalnivå. Dock så var de flesta bitarna nollor vilket såg ut som ett slumpvis tal tolkat som decimaltal när den i själva fallet bestod mestadels av nollor.
Anledningen att strömmen till stor del bestod av nollor är att bitströmmen som ska genereras är mycket större än parametrarna som används . Matematiken som används gör att bitströmmen inte kommer att ha mer data eller bitar än vad inparametrarna har. Med *x^y mod z* funktionen så är matematiken sådan att resultatet aldrig kan bli större än det värdet som *z* har.
Vad det innebär är att bitströmmen aldrig kan bli större än nycklarna som används vid uträkningen.
Prototypen fungerade inte enligt kraven och blev ej godkänd.

3.2.1.3 LPCRP3 (Lovén Persson Crypto Prototype 3)

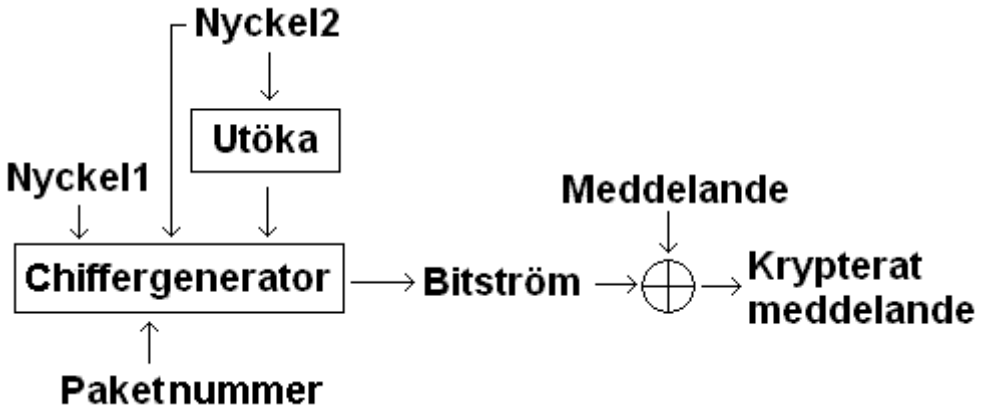


Figur 9

Den tredje prototypen försökte lösa problemet med LPCRP2 bara genom att göra så att en av inparametrarna är minst lika lång som chiffergeneratorns utdatalängd ska vara. Nyckel2 valdes som den parameter som skall expanderas och en ny funktion skapades som tog en nyckel och ändrade om den så den blev tillräckligt stor. Resultatet är en chiffergenerator som har 4 parametrar, *Nyckel1*, *Nyckel2*, *paketnummer* och *Utökad Nyckel.* I i provimplementationen av LPCRP3 så används storleken 1024 byte eller 8192 bit.
Med denna förändring så visade tester att generatorn nu gav ut en tillräckligt lång bitström som såg ut att vara slumpade bitar.
Eftersom denna prototyp uppfyllde alla krav på generatorn så blev den godkänd till LPCR algoritmen.

3.3 Beskrivning av LPCR

Algoritmen är symmetrisk vilket betyder att båda parter behöver ha samma nyckel eller nycklar. LPCR använder sig av en tvådelad nyckel på grund av de anledningar som diskuteras i kapitel 2.2.1. Det finns några olika metoder för att etablera symmetriska nycklar på ett säkert sätt. Till exempel kan man använda asymmetrisk kryptering för att kryptera nycklarna när de skall distribueras. RSA-kryptering* är ett exempel på asymmetriskt kryptering som lämpligen kan användas. En annan algoritm/metod som med fördel kan användas är *Diffie-Hellman key exchange*[11] protokollet för att etablera nycklarna. När båda klienter har samma nycklar så skickas Nyckel 2 in i en speciell metod som utökar nyckelns längd och ger ut en tredje nyckel som kallas *Utökad Nyckel*. (se kapitel 3.3.1)
När klienterna har etablerat sina nycklar så kan sändning av data påbörjas. Sändning av data går till på följande sätt.
Sändarklienten som ska skicka meddelandet väljer slumpvis ett värde som blir Paketnummer. Sändarklienten skickar sedan in sina tre nycklar (*Nyckel1*, *Nyckel2* och *Utökad Nyckel)* och paketnummer till chiffergeneratorn och får ut en bitström. Sändarklienten gör sedan XOR mellan bitströmmen och meddelandet och får ut det krypterade meddelandet. Till sist så lägger sändarklienten till paketnummer i klartext på det krypterade meddelandet och skickar alltihop till mottagaren.
När mottagaren tar emot meddelandet så läser denna vad det är för paketnummer och skickar in denna paketnummer plus de tre nycklarna, d.v.s. *Nyckel1*, *Nyckel2*, *Utökad Nyckel*, mottagarklienten redan har till sin chiffergenerator. Mottagarklienten får då ut samma bitström som avsändaren använde för att kryptera meddelandet med. Mottagarklienten gör då bara XOR mellan det krypterade meddelandet och sin bitström och får då ut meddelandet i klartext.



Figur 10

3.3.1 Utökningsmetoden

Denna metod har som enda uppgift att skapa en ny nyckel som beror på en av de två originalnycklarna och har minst en specifik bitlängd. Den behövs eftersom

matematiken i chiffergeneratorn (se kapitel 2.3.2) behöver en parameter som har åtminstone lika många bitar som bitströmmen den ska ge ut.
Den fungerar enligt följande pseudokod:

```
Utöka(nyckel){
  Spara nyckel som en bytevektor
  Dela bytevektorn i två lika delar, FirstArray och
  LastArray
  Gör om FirstArray och LastArray till nummer igen
  While(nyckel är mindre än önskad bitlängd){
     Bitskifta nyckel vänster 32 steg
     Räkna ut nyckel^FirstArray mod LastArray
     Addera ovan uträkning till nyckel
  }
  returnera nyckel
}
```

Det kan vara lite krångligt att räkna ut det värdet som ska adderas på nyckeln. Det finns olika sätt att lösa det, ett sätt är att dela upp originalnyckeln i olika delar och göra envägsformler med dem. (se kapitel 3.3.2 om envägsformler)

3.3.2 Chiffergenerator

Huvuddelen av LPCR är chiffergeneratorn. Den behöver 4 parametrar och ger ut en bitström som används för att kryptera ett meddelande. De fyra parametrarna är: Nyckel1, Nyckel2, Utökad nyckel och Paketnummer. Om samma parametrar skickas in två gånger så får man ut samma bitström. Algoritmen är konstruerad på ett sådant sätt att utdatan ändras totalt beroende på en värdesändring i någon utav inparametrarna. Detta krävs eftersom vid en normal sändning så används samma nycklar vid flera sändningar och det är bara paketnummer som ändras. Man behöver inte bry sig om synkronisering utan det räcker att mottagaren får veta vad paketnummer är så kan han avkryptera meddelandet.

Generatorn fungerar enligt följande pseudokod:

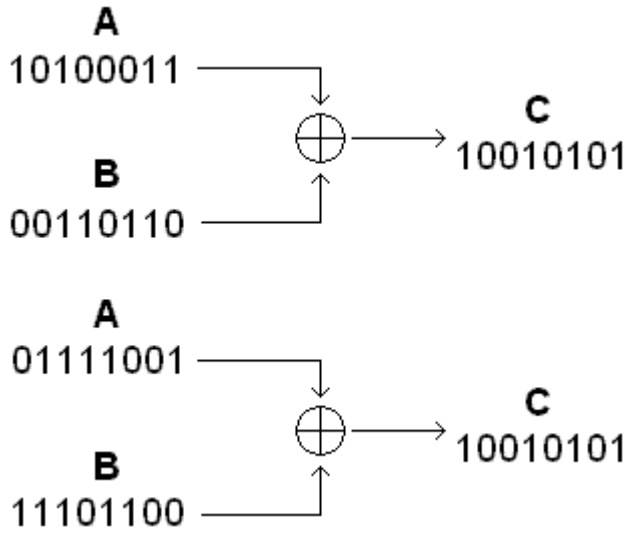
```
generator(nyckel1, nyckel2, utökad nyckel, paketnummer){
        bitström = nyckel1^paketnummer mod nyckel2
        while(bitströmmens bitlängd < antal bitar som
        bitströmmen ska vara){
                multiplicera bitströmmen med utökad
                nyckel
        }
        släng extra bitar så att bitströmmen är önskad
        längd
        returnera bitströmmen
}
```

Chiffergeneratorn behöver ha en viss säkerhet, d.v.s. att det inte ska gå att kunna räkna ut vad nycklarna är även om man vet hur bitströmmen ser ut och om man vet vad paketnummer är.
Den första säkerheten ligger i *a=x^y mod z* uträkningen. Detta är en envägsformel och det finns inget matematiskt sätt att räkna den baklänges även om man vet vad a, x eller z har för värden.[12]
Den andra ligger i det faktum att chiffergeneratorn inte ger tillbaka exakt vad som har räknats ut, den klipper av en del värden i slutet så att bitströmmen alltid har samma storlek. Uträkningen av bitströmmen blir nämligen olika längd beroende på parametrarna men alltid minst så lång som bitströmmen ska vara. Detta gör så att an utomstående ej har möjlighet att analysera fram information om nycklarna eftersom bitströmmen ej är garanterat den som uträkningen egentligen resulterade i.
Slutligen så skickas heller aldrig bitströmmen utan det är alltid bara det krypterade meddelandet och paketnummer som skickas.
Säkerheten utgår från att en möjlig attackerare inte har fysisk tillgång till datorn som krypteringen och avkrypteringen sker på, d.v.s.tillgång till själva hårdvaran eller datorn som skickar eller tar emot meddelanden. En sådan säkerhet får lösas på andra sätt än med LPCR algoritmen.

3.3.3 Säkerhet

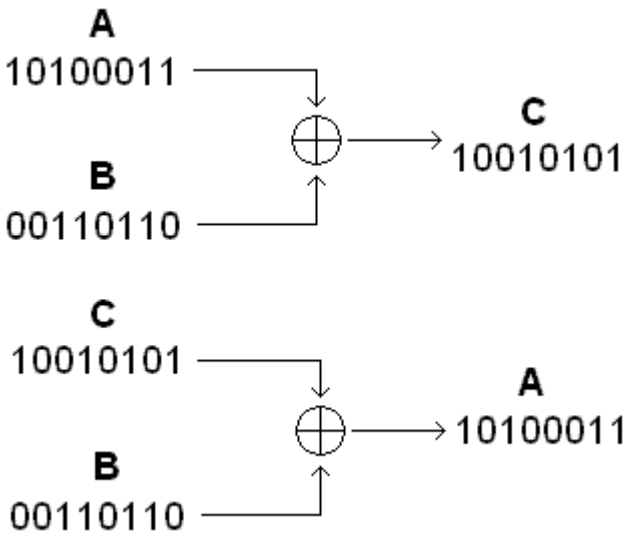
Den allra viktigaste aspekten med en krypteringsalgoritm är att den håller en tillräckligt hög säkerhetsstandard. Säkerheten kan delas in i två kategorier. Den första ska se till att meddelandet ej ska gå att läsa eller avkoda. Den andra ska se till att nycklarna ej ska gå att ta reda på vid en eventuell attack.
För att gömma själva meddelandet så används den logiska operationen XOR.

Som figur 11 visar så går det aldrig att veta vad A och B var om man känner till C. Anledningen är att A och B kan vara nästan vad som helst då det finns många olika kombinationer som ger exakt samma utvärde, d.v.s. C värdet. Figur 11 visar att två helt olika kombinationer som efter XOR operationen blir exakt samma resultat.



Figur 11

XOR-operationen har också den egenskapen att den är reversibel. Som figur 11 visar så innebär det att om A och B ger C efter operationen så är det även sant att C och B ger A efter operationen.[13]



Figur 12

Detta betyder också att om en angripare får tillgång till A eller B värdet så kan han få ut det andra värdet.
Om man säger att A representerar meddelandet i klartext och B representerar bitströmmen så kan en angripare få reda på bitströmmen om han vet vilket klartextmeddelande som hör ihop med ett visst krypterat meddelande. Angripare

kan också på andra hållet få fram klartextmeddelandet om han vet bitströmmen som hör ihop med ett visst krypterat meddelande.
Detta leder till den andra säkerhetsdelen, chiffergeneratorn. Den ska se till att en angripare inte kan komma åt eller räkna ut krypteringsnycklarna även om han vet en viss bitström och paketnummret som hör till den bitströmmen. Har man kommit åt nycklarna så kan man avkryptera alla meddelanden som skickas med just de nycklarna, detta går aldrig att komma ifrån då själva principen av ett krypto är att de måste gå att avkryptera om man har nycklarna.
För att åstadkomma detta så används en så kallad envägsfunktion. Det är en matematisk formel som är enkel att räkna på ena hållet men så gott som omöjlig att räkna på andra hållet.[12]
Vad chiffergeneratorn använder är modulär exponentialräkning. Den gör uträkningen:

temporär bitström = nyckel1^paketnummer mod nyckel2

Utöver detta så kastas även delar av det uträknade värdet bort vilket betyder att en angripare aldrig kan se vad det riktiga uträknade värdet verkligen är.
Uträkningen $a = x^y \bmod z$ bygger på problemet att räkna ut den Diskreta Logaritmen vilket det för tillfället inte finns någon matematisk formel för att göra på ett effektivt sätt.[13] Problemet är att kunna räkna ut i ovan nämnda formel y om man känner till a, x och z.
Dock så är fallet med LPCR så att x och z är de okända nycklarna som en angripare vill få reda på. Han vet heller inte vad a är för något eftersom delar av a slängs innan den skickas ut för att användas vid kryptering. En angripare vet bara y vilket alltid skickas som klartext. Denna konstruktion bör vara tillräckligt säker enligt Diskreta Logaritmens premisser.

För att skydda sig mot en vanlig bruteforce* attack så behöver nycklarna vara tillräckligt stora så att en bruteforce tar för lång tid för att vara ett effektivt sätt att knäcka kryptot. Minimum längd på nycklarna är 64-bit nycklar om krypteringen ska anses som säker, Helst högre för att göra det säkrare. Om man använder kortare nycklar så blir det för lätt att knäcka och bör bara göras i testsyfte eller utbildningssyfte. För att get an aning perspektiv så tar det 10^13 år att gå igenom alla olika möjliga nycklar i en 128-bit nyckel om man har en dator som kan kolla 10^18 olika nycklar per sekund. 10^13 år är längre än åldern på vårt universum vilket är ca 1.3–10^10 (13,000,000,000) år och att kolla 10^18 nycklar per sekund är ungefär 1000 gånger snabbare än den snabbaste datorn i världen. Världens snabbaste dator för tillfället är Roadrunner som är byggd av IBM. Den klarar 1.71 petaflops (top).[16] Eftersom krypteringen använder två nycklar och de båda måste vara minst 64 bitar var så kan man se dem som en enda 128-bit nyckel.

4 Kompressions & krypteringsalgoritmen SCAM (Secure compressions algorithm for multimedia)

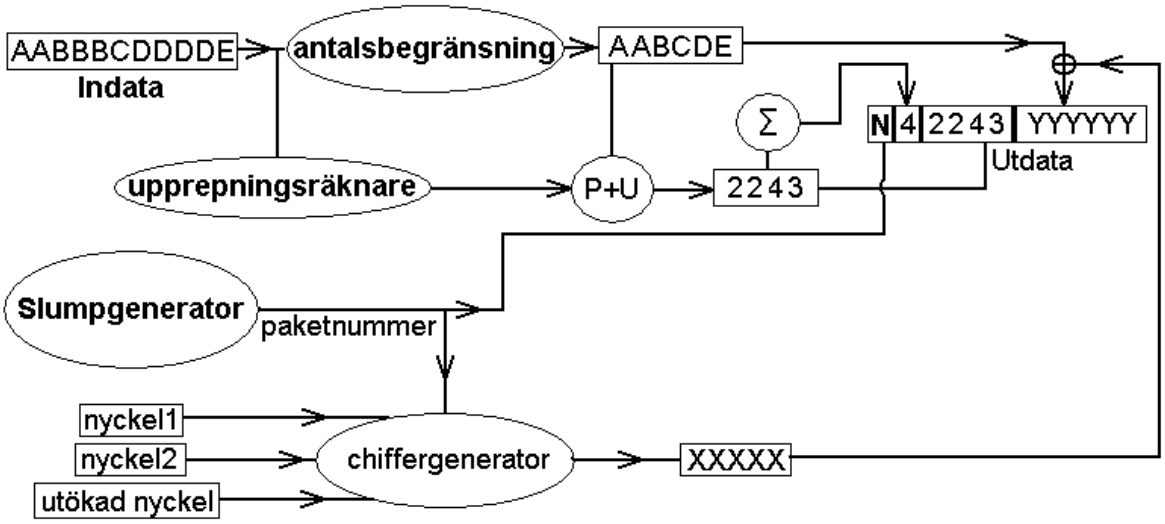
Det tänkta slutresultatet för projektet var att integrera LPRL- och LPCR-algoritmen till en gemensam algoritm med namnet SCAM (**S**ecure **C**ompressions **A**lgorithm for **M**ultimedia).
Denna algoritm var tänkt att kombinera säkerhetsegenskaperna hos LPCR med de datakomprimerande egenskaperna hos LPRL till en gemensam algoritm och därmed få ned tidskomplexiteten jämfört med att köra de båda algoritmerna efter varandra.

4.1 Integration av algoritmerna

För att förstå detaljer om hur algoritmen som beskrivs i detta kapitel fungerar se kapitel 2 & 3 då SCAM bygger på algoritmerna beskrivna i dessa.

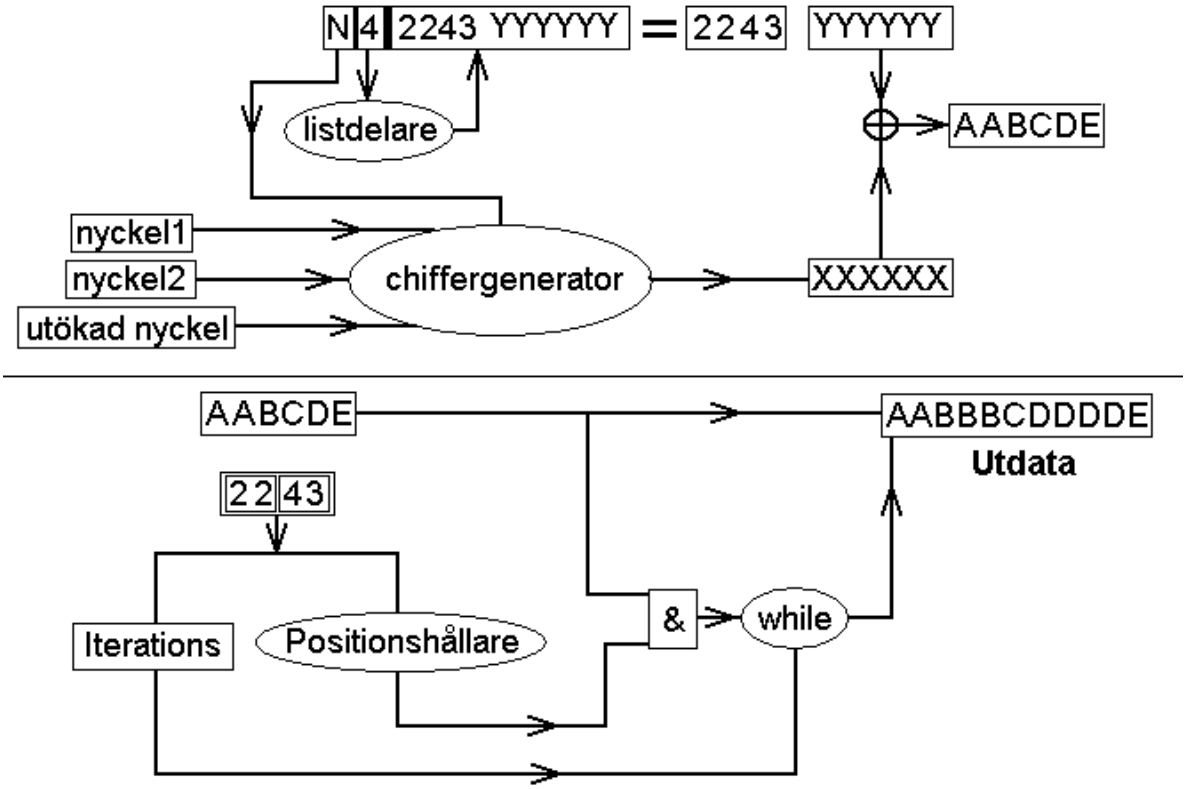
4.1.1 Utvecklingsversion1

Först versionen av SCAM implementerades genom att den krypterande operationen XOR placerades i programflödet av LPRL enligt följande bild.



Kryptering/kompression med SCAMv1. ⊕ symboliserar den logiska operationen XOR. För beskrivning av P+U och ∑ se bilder kap 2.2

Denna implementation krypterar alltså enbart teckentabellen och inte iterationsangivelserna. Anledningen till detta är att det är omöjligt att avgöra vilken plats i chifferströmmen som skall användas för att avkryptera teckentabellen eller iterationsangivelserna om båda krypteras tillsammans. Avkryptering/avkompression sker genom normal LPRL algoritm dock med undantaget att tecknen avkrypteras med LPCR innan de placeras i utdatan.



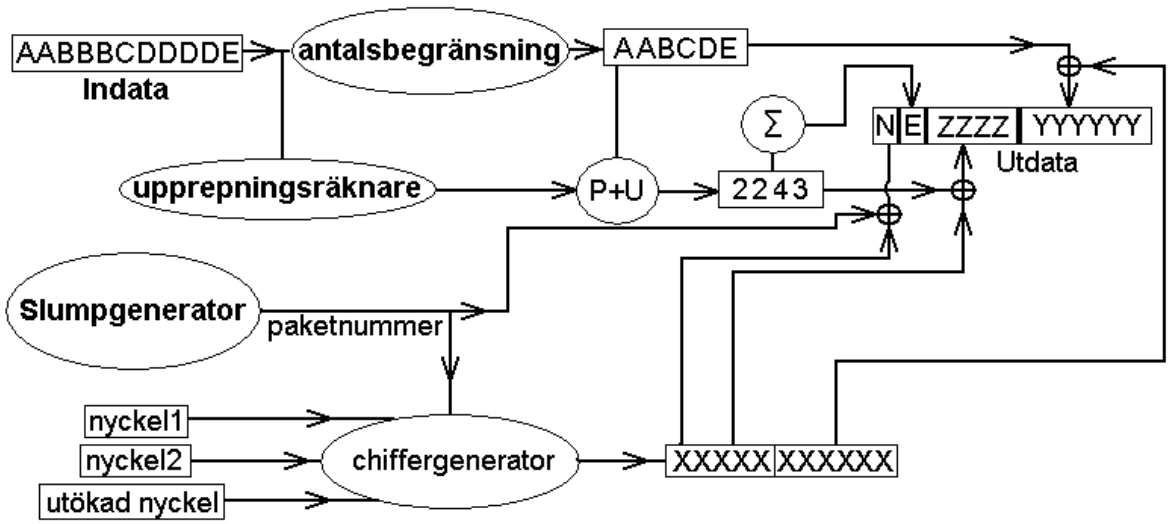
Avkryptering/avkomprimering med SCAMv1. ⊕ symboliserar den logiska operationen XOR. För beskrivning av while se bilder kap 2.2

En nackdel med denna implementationen är att innehållsanalys blir markant enklare då en angripare kan se vilka tecken som förekommer ofta direkt genom att titta på iterationsangivelserna.

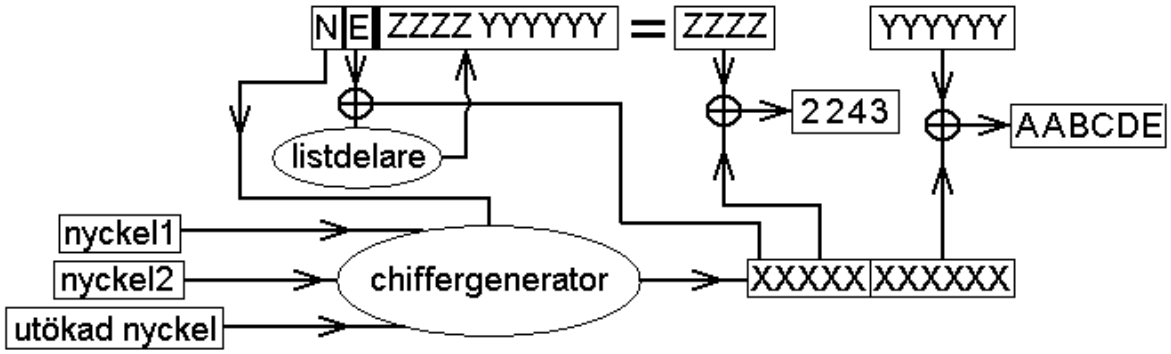
4.1.2 Utvecklingsversion2

För att komma till rätta med säkerhetsbristen i utvecklingsversion1 krävs det att all utdata krypteras med XOR-operationen. För att detta skall vara möjligt krävs att det går att skilja vilka tecken av chiffer som skall användas för att kryptera teckentabellen respektive iterationsangivelserna.
I utvecklingsversion2 görs detta genom att iterationsangivelserna krypteras direkt medan teckentabellen lämnas orörd för stunden. När hela genomgången av indatan är slutförd krypteras även teckentabellen med chiffertecken hämtade från det index som krypteringen av iterationsangivelserna slutade vid.
Även värdet som anger hur många tecken som skall tolkas som iterationsangivelser krypteras, Detta värde krypteras med det första tecknet ur chifferströmmen eftersom det tecknet är det första som måste avkrypteras.

Nackdelen med denna lösning är att den ger onödigt hög tidskomplexitet vid kryptering/kompression då hela teckentabellen måste gås igenom två gånger, först en gång för att skapa den, därefter för att kryptera den. Lika så blir det onödigt hög tidskomplexitet vid avkryptering/avkompression där iterationsangivelserna måste gås igenom två gånger, först för att avkryptera och därefter för att nyttjas som iterationsangivelser.



Kryptering/kompression med SCAMv2. ⊕ symboliserar den logiska operationen XOR. För beskrivning av P+U och ∑ se bilder kap 2.2

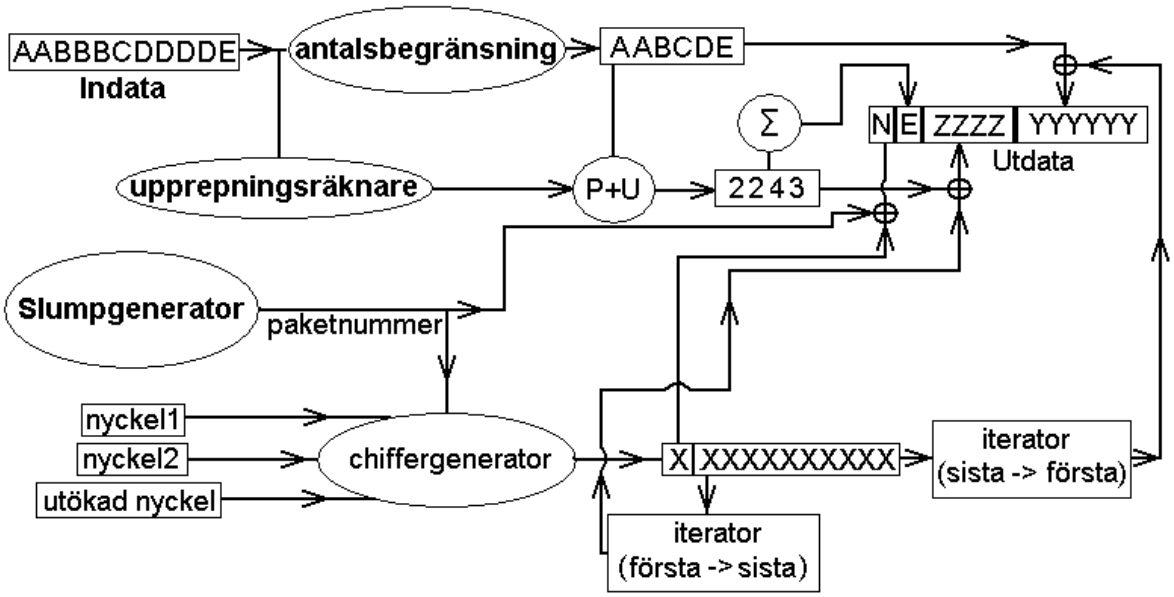


Avkrypteringsdelen av SCAMv2 avkryptering/avkompression. ⊕ symboliserar den logiska operationen XOR.

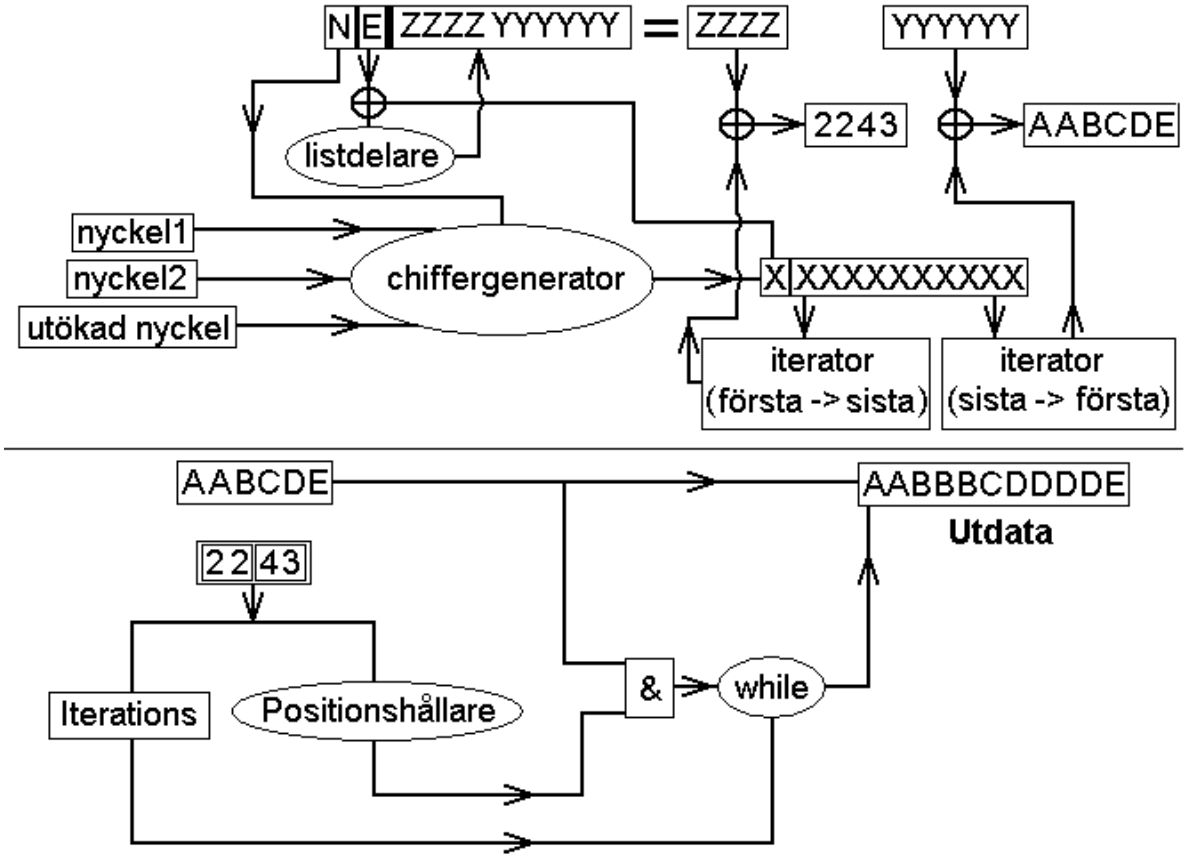
4.1.3 Slutversion av algoritmen

Nackdelen med utvecklings version2 var som nämnt ovan att den hade onödigt hög tidskomplexitet både vad det gäller kryptering/kompression och avkryptering/avkompression.
Detta beror på att en av delarna alltid måste genomlöpas två gånger, antingen för att kryptera den efter kompression eller för att avkryptera innan avkompression. Anledningen till att en av delarna måste behandlas två gånger ligger i att det annars inte går att skilja vilka tecken som skall XOR-opereras med vilka efter att indatan delats upp i teckentabell och iterationsangivelser om inte den ena delen behandlas separat.
I slutversionen av SCAM löstes detta genom att delarna krypteras med tecken från vardera sida av chifferströmmen, det vill säga iterationsangivelserna krypteras med tecken från början till slutet av chifferströmmen medan teckentabellen krypteras med tecken hämtade i ordning från sista till första.
Detta innebär att det inte länge är något problem att avgöra vilka tecken som skall användas till vilken av datarepresentationerna. Tack vare detta kan nu

kryptering av båda delar ske parallellt vilket resulterar i enbart en genomgång av indatan för där både kompressions- och krypteringsoperationer kan utföras. Tidskomplexitet för denna del blir alltså T(n)=n = O(n). Exempel *4.1* och *4.2* visar kryptering/kompression respektive avkryptering/avkompression med SCAM-algoritmen. I *4.2* visas avkryptering och avkompression sekventiellt för att göra det överskådligt för läsaren, dessa operationer händer i verkligheten parallellt vilket syns i pseudokoden i kapitel 4.2.



4.1. Kryptering/kompression med SCAM. • symboliserar den logiska operationen XOR. För beskrivning av P+U och ∑ se bilder kap 2.2



4.2. Kryptering/kompression med SCAM. • symboliserar den logiska operationen XOR. While lägger till symbolen från teckenlistan så många gånger som iterationsangivelsen anger. While metoden aktiveras av OCH grinden styrd av teckenlistan och iterationsangivelserna.

4.1.4 Tidskomplexitet

Normal tidskomplexitet för LPRL är som nämnt i kap o 2.2.4 O(n) (T(n)=n) och LPCR har O(n) (T(n)=n). Detta innebär att tidskomplexiteten om algoritmerna utförs sekventiellt O(n)+O(n) (T(n)=n+n).
Tidskomplexiteten för själva kryptering/kompression operationen i SCAM har som beskrivet i 4.1.3 komplexiteten O(n) (T(n)=n).
Dock krävs det att chiffergeneratorn skapat en chifferström innan krypteringsdelen kan utföras, eftersom chiffergeneratorn som beskrivet i kap 3.3.2 har O(n) blir den totala genomlöpningskomplexiteten O(n+n) = O(2n). Detta medför en förbättring i tidskomplexitet med O(n), alltså motsvarande mängden operationer som krävs för att gå igenom indatans längd en gång.[5]

4.2 Implementation

4.2.1 krypterings/kompressions-algoritm i pseudokod

Exemplet nedan beskriver kryptering/kompressionsdelen av algoritmen, premisserna för denna genomgång är att två krypteringsnycklar finns tillgängliga samt att expanderad nyckel har skapats enligt kap 3.3.1.

```
Slumpa fram paketnummer;
Placera paketnummer först i utdata;
Skapa chifferström enligt metod generator (kap 3.3.2) med
inparametrar: nyckel1, nyckel2, utökad nyckel, paketnummer;
Skapa iterator1 för chifferström med startindex
först+1(läser från början till slut);
Skapa iterator2 för chifferström startindex slut av
strömchiffer(läser från slut till början);

senaste = första tecknet i indata;
första platsen i utdataTecken = ”första tecknet i indata
XOR iterator2”;
antalSenaste = 0;
i=1;
så länge i<Indatans längd {
         Om senaste = Indata(på platsen i) görs{
                  antalSenaste = antalSenaste +1
         }
         Om senaste inte =  Indata(på platsen i) &
         antalSenaste>0 görs{
                  Ta reda på nummer representerande platsen
                  ”senaste” fick i utdataTecken och värdet
                  lägg i dummyvariabel;

                  utför ”iterator1 XOR dummyvariabel” och
                  placera resulteat i utdataIterationer;

                  utför ”iterator1 XOR antalSenaste” och
                  placera resulteat i utdataIterationer;
```

4.3 fortsätter på nästa sida.

```
                    Uppdatera till den nya ”senaste” =
                    Indata(på platsen i);
                    antalSenaste=0;
                    lägg till Indata(platsen i) i
                    dummyvariabel;
                    ”dummyvaraibel XOR iterator2” placera
                    resultat sist i utdataTecken;
          }

          Om senaste inte =  Indata(platsen i) &
antalSenaste=0 görs{
                    lägg till Indata(platsen i) i
                    dummyvariabel;
                    ”dummyvaraibel XOR iterator2” placera
                    resultat sist i utdataTecken;
                    Uppdatera till den nya ”senaste” =
                    Indata(på platsen i);
                    antalSenaste=0;
          }
          i=i+1;
}
Om antalSenaste>0 görs{
                    Ta reda på nummer representerande platsen
                    ”senaste” fick i utdataTecken och värdet
                    lägg i dummyvariabel;

                    utför ”iterator1 XOR dummyvariabel” och
                    placera resulteat i utdataIterationer;

                    utför ”iterator1 XOR antalSenaste” och
                    placera resulteat i utdataIterationer;
}
Räkna ut antalet tecken i utdataIterationer, plasera värdet
i dummyvariabel;
”Dummyvariabel XOR första värdet ur chifferströmmen” och
lägg resultatet på nästförsta platsen i Utdata;
Lägg utdataIterationer med start på plats tre i Utdata;
Lägg utdataTecken efter utdataIterationer i Utdata;

Utdata är nu komplett.
```

4.3

4.2.2 avkrypterings/avkompressions-algoritm i pseudokod

Exemplet nedan beskriver avkryptering/avkompressions delen av algoritmen, premisserna för denna genomgång är att två krypteringsnycklar finns tillgängliga samt att expanderad nyckel har skapats enligt kap 3.3.1.

```
Ta ut första värdet ur indatan och lägg värdet i variabel
paketnummer;
Skapa chifferström enligt metod generator (kap 3.3.2) med
inparametrar: nyckel1, nyckel2, utökad nyckel, paketnummer;
Skapa iterator1 för chifferström med startindex
först+1(läser från vänster till höger);
Skapa iterator2 för chifferström startindex slut av
strömchiffer(läser från höger till vänster);

Ta ut första värdet ur indata (tidigare näst första) och
utför XOR med första värdet i chifferströmmen, lägg
resultat i dummyvariabel;
Dela återstående indatan vid positionen angiven av talet i
dummyvariabel,
De två delarna kallas nu Iterationslista och Teckenlista;
Ta ut första platsen i iterationslistan, utför XOR med
iterator1 och lägg resultat i IterationsPosition;
Ta ut första platsen i iterationslistan, utför XOR med
iterator1 och lägg resultat i IterationsAntal;
i=0;
så länge i<Teckenlistas längd {
ta ut det främst placerade tecknet från Teckenlista och
placera det i dummyvariabel;
Utför ”dummyvariabel XOR iterator2” och lägg resultat först
i utdata;

         Om IterationsPosition = i görs{
                  Lägg till resultatet av ”dummyvariabel
                  XOR iterator2” i Utdata igen, gör detta
                  IterationsAntal gånger;
                  Ta ut första platsen i iterationslistan,
                  utför XOR med iterator1 och lägg resultat
                  i IterationsPosition;
                  Ta ut första platsen i iterationslistan,
                  utför XOR med iterator1 och lägg resultat
                  i IterationsAntal ;
         }
         i = i +1;
}
Utdata är nu komplett.
```

5 Proof of Concept

5.1 Chat programmet Libercom

Under projektet skapades ett program för att testa och demonstrerar SCAMs funktionalitet, detta program kallas Libercom (namn taget från domän programmet publicerats på www.liberit.com).
Programmet är ett text- och röst-chatprogram som implementerar kompression och kryptering med hjälp av SCAM.
Libercom består av en klient och en server, båda är implementerade i Java.
Det normala programflödet är enligt följande:

1. Servern startas, den slumpgenererar två krypteringsnycklar för SCAM, därefter väntar den på att klienter skall ansluta.
2. Klienten startar programmet och skriver in vilken server han vill ansluta mot samt sitt chat-namn och eventuellt lösenord.
3. Klienten skickar n och e nycklar för RSA kryptering till servern.
4. Servern skickar en lösenordsutmaning tillbaka till klienten, utmaningen består av en tidsstämpel som skall hashas* med lösenordet.
5. Klienten hashar eventuellt lösenord med utmaningen från servern och skickar tillbaka till servern. För att undvika birthdayattacker* skickas även lösenordet hashat med RSA e-nyckeln.
6. Klienten skickar önskat chat-namn.
7. Servern kontrollerar att namnet är giltigt, dvs ingen annan användare nyttjar namnet för närvarande samt att namnet inte är för långt eller kort.
8. Servern skickar de två SCAM nycklarna krypterade med RSA.
9. Servern sänder ut signal till alla eventuellt anslutna användare om att ”namn” har anslutit.
10. Klienterna kommunicerar genom att skicka meddelande till servern som servern sedan sänder vidare.
11. Efter en tid avslutar klienten anslutning.
12. Servern sänder då meddelande till eventuella andra klienter att ”namn” avslutat anslutning.

Meddelande kan sändas både som globala och som privata.
Vid privat sändning sker detta genom att klienten placerar ett specialtecken i början av paketet följt av det chat-namn den tänkta mottagaren har. Servern sänder då endast meddelandet till den klient som är registrerad med detta namn.

Libercom är implementerat med grafiskt användargränssnitt och fungerar under de flesta moderna operativsystem, dock under förutsättning att Java finns installerat.

Klientdelen har testats under följande operativsystem med tillhörande Java-versioner:

- Linux Fedora 9 - OpenJDK java 1.6
- Linux Fedora 10 - Sun Java jre1.6
- Linux Debian 3.6 - OpenJDK java 1.6
- Linux Debian 3.6 - Sun Java jre1.6
- Linux Ubuntu 8 - Sun Java jre1.5
- Microsoft Windows Xp – Sun Java jre1.5
- Microsoft Windows Xp – Sun Java jre1.6
- Microsoft 2000 – Sun Java jre1.5
- OpenBSD 4.5 – Sun Java jre1.5

Sämsta testade hårdvara: Intel Pentium II 333Mhz, 256Mb SD-Ram med Linux Debian 3.6.

Serverdelen har testats på följande operativsystem med vederbörande Java-version:

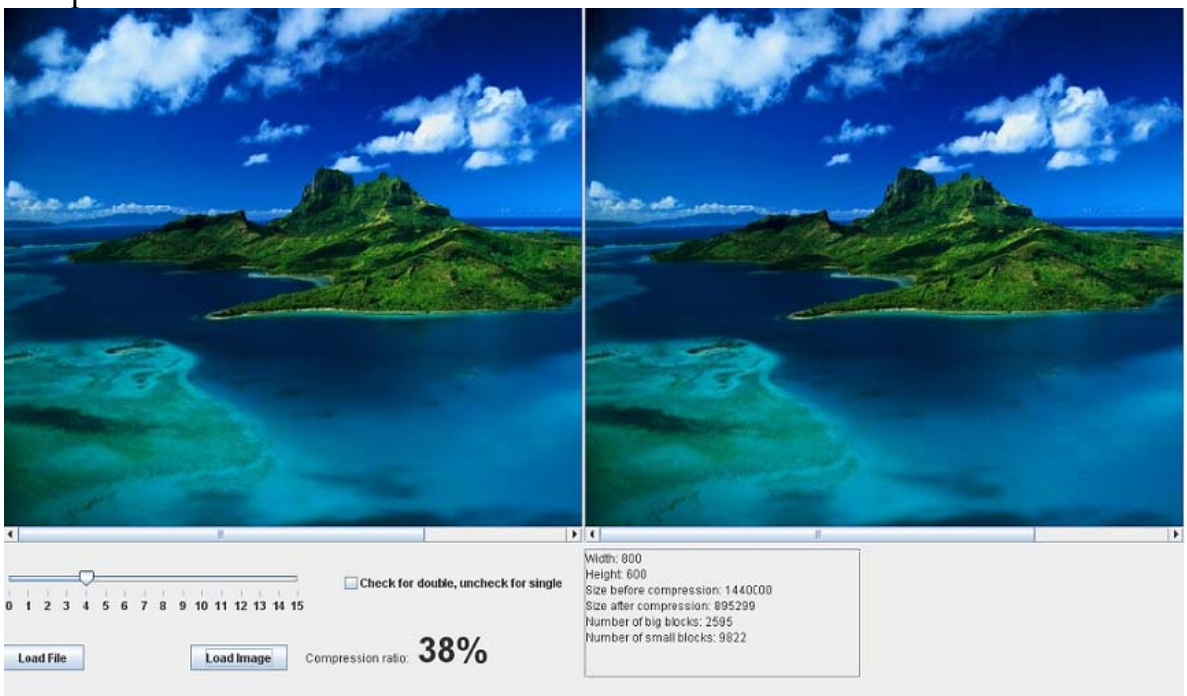
- Linux Fedora 10 - Sun Java jre1.6
- Linux Fedora 10 – OpenJDK java 1.6
- Linux Debian 3.6 - Sun Java jre1.5
- Linux Debian 3.6 - Sun Java jre1.6
- Linux Debian 3.4 – Sun Java jre 1.4
- Microsoft Windows Xp – Sun Java jre1.6
- Microsoft 2000 – Sun Java jre1.5

Sämsta testade hårdvara: Intel Pentium1 266Mhz, 96Mb SD-Ram med Linux Debian 3.6 samt med Linux Debian 1.4.

Klient- och server-programmen som utgör Libercom samt deras respektive källkod finns att ladda ned:
Klient: http://www.liberit.com/files/LibercomClientSRC.rar
Server: http://www.liberit.com/files/LibercomServerSRC.rar

5.2 LPRL vissningsprogram

Även ett program för att visualisera LPRL och LPIS implementerades. Detta Program opererar genom att ladda in en bild och visar den sedan på skärmen. Därefter appliceras LPIS följt av LPRL på bilden, därefter görs en uträkning för att visa vilken kompressionseffektivitet om uppnåtts genom att jämföra original data med komprimerad data.
Programmet kan även utföra LPRL-operationen på vilken typ av fil som helst, dock utförs då ingen LPIS-operation eftersom detta skall ske LossLess. Någon bild för att visuellt representera kompressionen går då heller inte att visa eftersom indatan bara består av binärtal. Denna funktion är enbart implementerad för att räkna ut den procentuella kompressionseffektiviteten.
Här syns ett screenshot av programmet med en exempel bild som testats, bilden till vänster är originalbilden och den högra representerar bilden efter att LPIS och LPRL-kompression och avkompression utförts på den. Procent värdet i som syns i mitten är den procentuella minskningen i filstorlek bilden hade efter LPRL-kompresion använts.



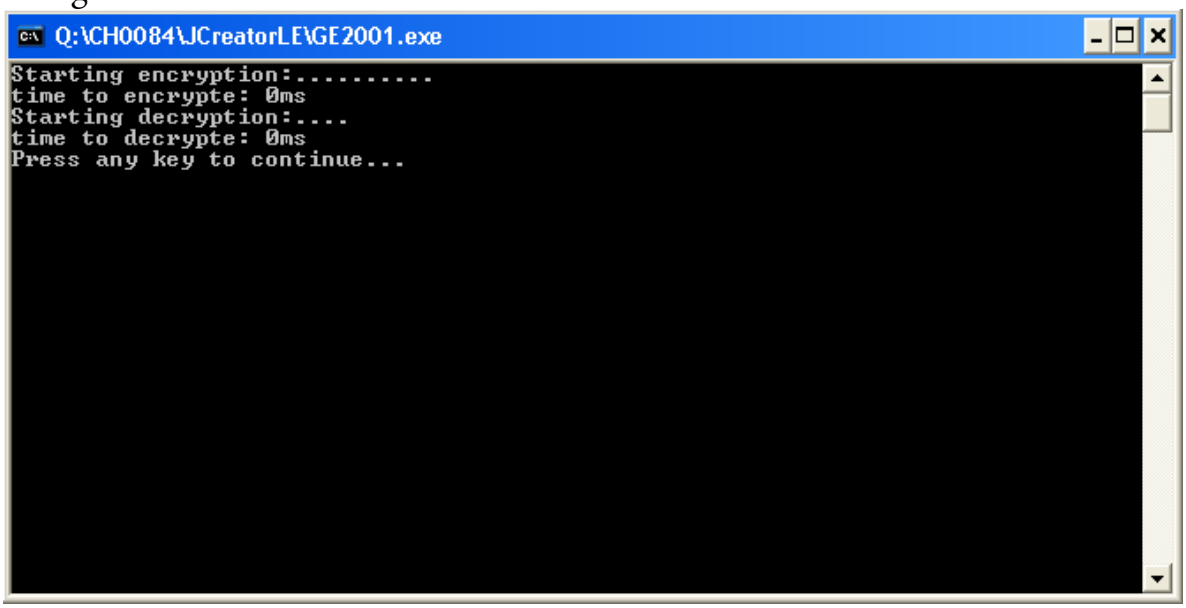
Detta program samt dess källkod finns att ladda ned på:
http://www.liberit.com/files/LPRLtestSRC.rar

5.3 Testresultat

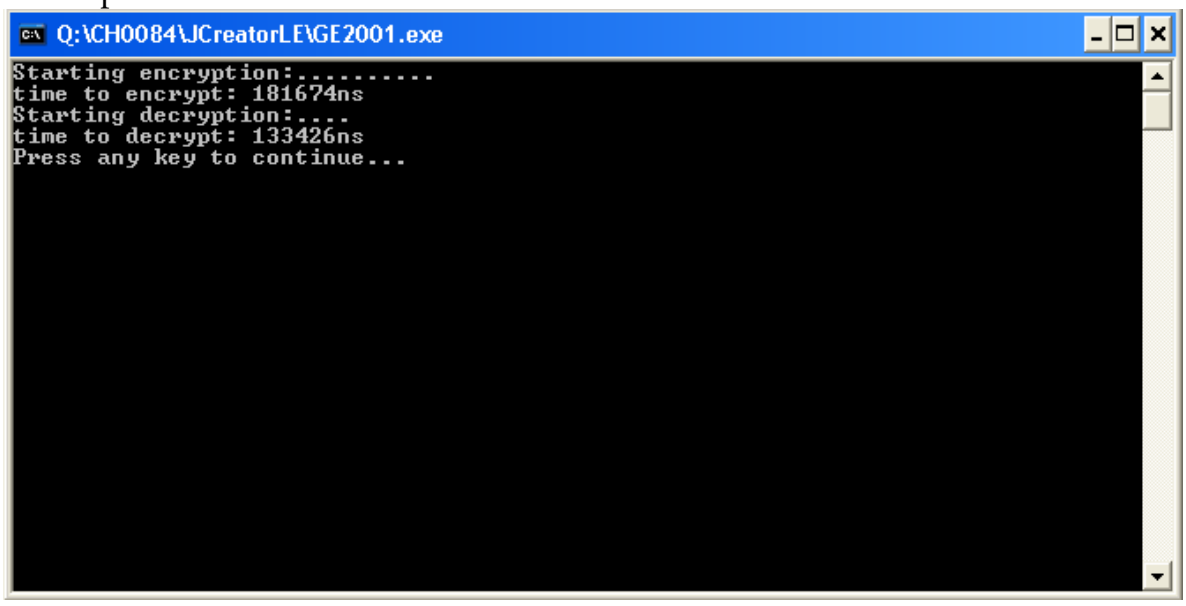
För att testa LPCR och LPRLs tidsåtgång implementerades ett enkelt test, detta test bestod av att algoritmerna skulle utföra sin respektive operation på tusen bytes (åttatusen bitar) och sedan skriva ut antalet nanosekunder det tog att utföra.. Även ett antal test gjordes för att mäta kompressionseffektivitet med LPRL.

5.3.1 LPCR

För att test LPCR implementerades algoritmen fristående i en mindre java applikation. Testning med denna applikation genomfördes på en PC med Windows XP servicepack 3 med en 2.66ghz processor och 2gb RAM. Resultatet skrivs här även ut i millisekunder för att ge en mer överskådlig bild av hastigheten.



Det tog alltså mindre än en millisekund att utföra kompression och avkompression med LPCR.



Av detta framkommer att tiden det tar att kryptera tusen byts är cirka 183473ns och att det sedan tar cirka 139128ns att avkomprimera samma data, tiderna varierar cirka +-10ns beroende på vilka andra processer som körs i bakgrunden.

För att göra testet mindre ensidigt utfördes det även på en dator av en äldre modell, Testutrustningen bestod då av en Pentium1 på 200mhz med 128mb RAM och Linux Debian 3.6 som operativsystem.

```
exjobb@liberit:~/exjobb/src/speed test$ ls
LCryptCompress.class    LPCR.class              StreamGenerator.java
LCryptCompress.java     LPCR.java               test.class
LGenericCompress.class  StreamGenerator.class   test.java
exjobb@liberit:~/exjobb/src/speed test$ java test
Starting encryption:..........
time to encrypt: 5754000ns
Starting decryption:....
time to decrypt: 4427000ns
exjobb@liberit:~/exjobb/src/speed test$
```

Här blir tidsåtgången något tydligare då det tar cirka 5,7ms att kryptera och 4.4ms att avkryptera. Dock bör detta resultat ses som tillfredställande då hårdvaran var av en standard som var modern 1996'.
Källkoden för testet finns att ladda ner på http://www.liberit.com/files/LPRCPrestandaTestSrc.rar.

5.3.2 LPRL

LPRL testades både i sin grundform på tusen bytes indata utan att specificera vilken typ av data som lästes in, samt genom att tolka samma data som en bild. Detta för att ge en uppfattning om både hur mycket effektivare kompressionen blir om den anpassas till indatan, men också för att vissa hur tidsåtgången påverkas av specialbehandling med tillexempel LPIS. Testning med denna applikation genomfördes på en PC med Windows XP servicepack 1 med en 1.66ghz dual processor och 1024mb RAM.
Resultaten av första testningen var enligt följande:

- Kompression av ospecificerad indatatyp: 637132 ns och 8% kompression
- Kompression av bildspecificerad data: 1134144ns 34% kompression
- Avkompression av ospecificerad data komprimerad 8%: 439328ns
- Avkompression av specificerad data komprimerad 34%: 733511ns

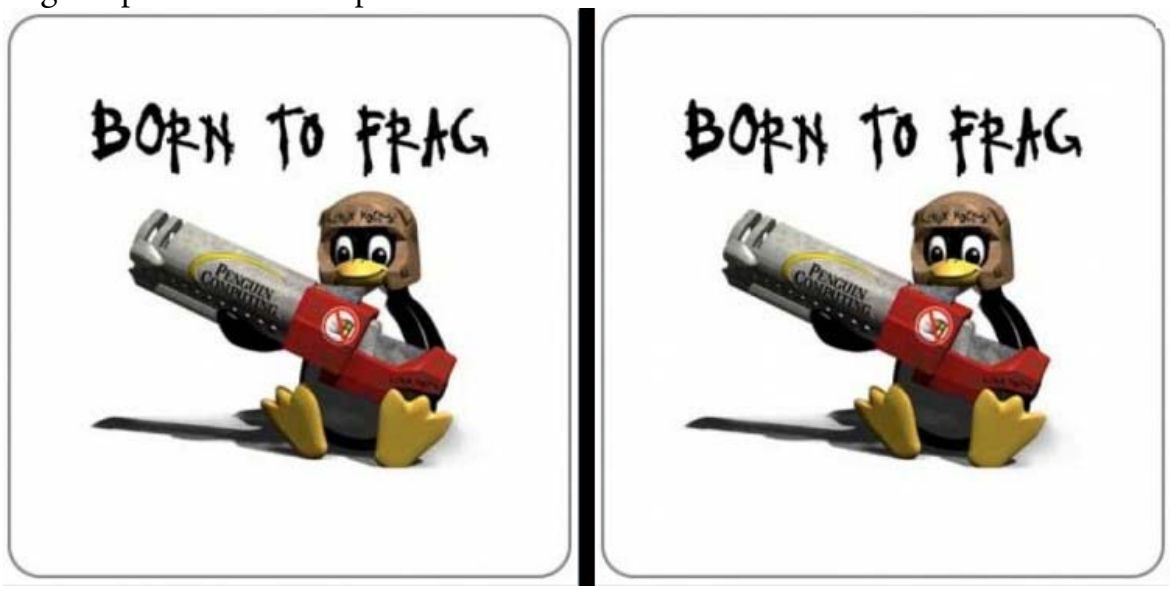
För att verifiera att testvärdena inte påverkades av yttre omständigheter upprepades samma test med följande resultat:

- Kompression av ospecificerad indatatyp: 520141ns och 8% kompression
- Kompression av bildspecificerad data: 1128261ns 34% kompression
- Avkompression av ospecificerad data komprimerad 8%: 439495ns
- Avkompression av specificerad data komprimerad 34%: 744021ns

Den enda direkta slutsats som går av dra utifrån testet är att en bildspecifik kompression blir effektivare, detta beror till stor del på effekten av LPIS. Dock syns det även att LPIS bidrar till en betydlig ökning av tidsåtgången.

5.3.2.1 LPRL och LPIS bildtest

Följande bildexempel illustrerar hur indata påverkar kompressionseffektiviteten, bilderna behandlas först med LPIS. Vänster bild representerar original medan höger representerar komprimerad:



Denna bild komprimerades 64%

Denna bild har komprimerats 34%

Här syns relativt tydligt hur detaljer i bilden påverkar kompressionseffektiviteten, mer detaljer i bilden ger mindre kompression. Samma effekt går att observera oavsett vilken typ av indata det gäller. Ljud med en hög frekvens tolkas även det som detaljerat och det komprimeras därför med relativt låg effektivitet.

5.3.3 SCAM

Då samma typ av test som användes på LPRL användes på SCAM observerades i stort sett samma värden med en variation på ca +- 20ns på samma typ av hårdvara.

Slutsatser

Då detta projekt till största del bedrivits som en empirisk undersökning är slutsatser något svåra att sätta fingret på. Mycket av resultaten består av utveckling och implementation av egna algoritmer samt skapandet av program för att påvisa dess praktiska funktionalitet.
Dock har implementation och testning under projektets gång bevisat att integration mellan en kompressions- och en krypterings-algoritm varit möjlig. Det har även visats att de ur tidskomplexitetsperspektiv varit möjligt att förbättra implementationer av den typ som projektet arbetat med.
De målen som projektgruppen satte som krav för ett lyckat projekt bör kunna anses som uppnådda:

- Skapa en kompressionsalgoritm som vid normalfallet åstadkommer förminskning i datamängd. – resultat ges i kap2 samt kap6.

- Skapa en kompressionsalgoritm som motsvarar LossLess definitionen.- kan verifieras i om i kap2 samt kap6.

- Skapa en krypteringsalgoritm som gömmer klartexten till en sådan grad att klartexen ej går att utläsa om en angripare enbart har tillgång till krypterad data. – kan läsas om i kap 3 specifikt 3.3.3.

- Skapa en integrerad kompressions- och krypterings-algoritm som har samma tidskomplextitet som motsvarande algoritmer för kompression och kryptering hade haft om de utfördes sekventiellt.- kan verifieras i kap4 specifikt 4.1.4.

- Skapa ett kommunikationsverktyg som implementerar den sammanslagna krypterings och kompressions-algoritmen. Kan verifieras i kap5 samt genom att studera källkoden

De eftersträvade målen som sattes upp för projektet kan även de till stor del betraktas som uppfylda:

- Skapa en kompressions/krypterings-algoritm som har en bättre tidskomplexitet än alternativet att utföra algoritmerna sekventiellt – Kan ses som uppnått med tanke på kap 4, specifikt 4.1.4

- Åstadkomma en kompressionsalgoritm för integration i slutresultatet som har ett värstafall som inte resulterar i att datamängden blir större än originaldatan. – delvis uppnåt, vid absolut värstafall av indata blir utdatan ett tecken större än original, detta är dock en markant förbättring jämfört med den kompressionsalgoritm LPRL är baserad på d.v.s run-

lengthkodning, see i kapitel 2, specifikt kap 2.2.2 för anledning och validering.

- Åstadkomma en krypteringsalgoritm för integration i slutresultatet som är relativt säker. – kan anses som uppnådd. Kan dock motbevisas om någon hittar en metod för att knäcka säkerheten* hos LPCR i framtiden, detta är dock en risk som finns i alla kryptografiska funktioner.

- Skapa ett kommunikationsverktyg som har verkliga användningsområden utöver att fungera som proof of concept. – Bör kunna räknas som uppnått då libercom är ett fullt fungerande chatprogram. Testversion samt källkod finns att ladda ner på:
  klient: http://www.liberit.com/files/LibercomClientSRC.rar
  server: http://www.liberit.com/files/LibercomServerSRC.rar

6 Referenser

[1]**Huffman kodning:**
Mark Allan Weiss: *Data structures & problem solving using Java* (third edition)
2005 ISBN: 0321312554
http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf
http://www.cs.duke.edu/csed/poop/huff/info/
http://www.compressconsult.com/huffman/
http://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html

[2]**Matematisk kompression:**
http://hpl.hp.com/techreports/2004/HPL-2004-76.pdf
http://urchin.earth.li/~twic/Golombs_Original_Paper/Golomb1966.djvu
http://www.ics.uci.edu/~dan/pubs/DC-Sec3.html
http://www.cs.toronto.edu/~mackay/itprnn/book.pdf
http://www.ics.uci.edu/~dan/pubs/DataCompression.html

[3]**Run-length kodning:**
http://www.cs.cf.ac.uk/Dave/Multimedia/node205.html
http://michael.dipperstein.com/rle/index.html
http://web.archive.org/web/20020214085725/http://www.rasip.fer.hr/research/compress/algorithms/fund/rl/index.html

[4]**Lossy och lossless-kompression:**
http://computer.howstuffworks.com/file-compression3.htm
http://compression.ru/download/articles/text/witten_1994cj_lossy_text_compression.pdf
http://catcode.com/imgguide/lossy.html
http://www.data-compression.com/lossless.shtml

[5]**Tidskomplexitetsanalys:**
Mark Allan Weiss: *Data structures & problem solving using Java* (third edition)
2005 ISBN: 0321312554
http://books.google.com/books?id=_tnwmvHmVwMC&pg=PA5&vq=%22The+trouble+is%22

[6]**Kryptografi:**
Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone: *Handbook of applied cryptography* 1996 ISBN: 0849385237
Nigel Smart: *Cryptography, An Introduction* 2002 ISBN: 0077099877
http://www.cs.bris.ac.uk/~nigel/Crypto_Book/

[7]**Symmetrisk kryptering:**
http://www.cs.bham.ac.uk/~mdr/teaching/modules04/security/lectures/symmetric-key.html

[8]**Blockchiffer:**
http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[9]**Strömchiffer:**
Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone: *Handbook of applied cryptography* 1996 ISBN: 0849385237

[10]**Asymmetrisk kryptering:**
http://www.cs.cornell.edu/courses/cs513/2007fa/TL04.asymmetric.html

[11]**Diffie-Hellman key exchange:**
http://www.47d.net/~seven/diffie.html

[12]**Matematiska envägsformler:**
http://mathworld.wolfram.com/TrapdoorOne-WayFunction.html

[13]**XOR-operation:**
Lars-Hugo Hemert: *Digitala kretsar* 2001 ISBN: 9144019181

[14]**Modulär potens:**
http://crypto.stanford.edu/pbc/notes/numbertheory/exp.xhtml

[15]**Bruteforce attacker:**
http://www.cl.cam.ac.uk/users/rnc1/brute.html

[16]**IBM Roadrunner:**
http://www.lanl.gov/roadrunner/

7 Ordlista

Normalfall: Den typ av indata som kan ses som det algoritmen normalt behandlar.
Värstafall: Den typ av indata som åstadkommer mest komplexitet för algoritmen att gå igenom.
Dokument: Syftar på indata av en specifik form.
Overhead: Saker som tar plats vid sparning eller sändning av data som egentligen inte är del av innehållet eller informationen, t.ex. nätverkspaket innehåller redundans och felkorrigering och denna kan i vissa fall ses som overhead.
ByteArray: en lagringsform i olika programmeringsspråk, den består av en mängd dataplatser som är av typen ”byte”, alltså åtta bitar stora.
Hash: En typ av matematisk funktion som översätter data till ett tal som har en mindre storlek än originaldatan, normalt är det omöjligt att räkna ut indatan om man bara känner till det hashade värdet.
Java: programmerings språk utvecklat av Sun Microsystems. Se referens http://java.sun.com/index.jsp
Knäcka säkerhet: syftar till att en obehörig person tillgängliggör sig information som är krypterad och ämnar vara skyddad.
Proof of concept: ett program eller algoritm som visar att konceptet och idén är möjlig att genomföra’
Birthday attack: krytografisk attack ämnad att göra en Bruteforce attack snabbare genom att använda statistiska modeller. På så sätt få fram ett nytt hash som lurar en lösenordsmekanisk. se referens:
http://www.efgh.com/math/birthday.htm
Bruteforce attack: kryptografisk attack som försöker att hitta en nyckel som används genom att testa all möjliga nycklar som kan tänkas användas.
Se referens:[14]
Lossy-kompression kompression där data representerande information som inte tillför resultatet något tas bort. se referens [4]
LossLess-kompression: kompression där ingen data eller information tas bort. se refernes [4]
RSA: asymmetrisk krypteringsalgoritm se referens:
http://www.google.com/patents?vid=4405829