Hashfunksjoner

for bruk i Digitale Signaturer

Hovedfagsoppgave

JAN ANDERS SOLVIK

Institutt for Informatikk
Det matematisk-naturvitenskaplige fakultet
Universitetet i Bergen

20. Oktober 1995

Forord

Jeg vil få takke min veileder Professor Øyvind Ytrehus for at han gav meg ideen til denne oppgaven. Han er også en medvirkende årsak til at den har blitt gjennomført. Jeg vil også få takke mine medstudenter og andre ved Institutt for Informatikk både av faglige og sosiale grunner. En takk går også til familie og venner som har gitt meg moralsk støtte hele veien.

Innholdsfortegnelse

Innledning 1

1 Kryptering og Digitale signaturer 3

1.1 Klassiske kryptosystemer 3
1.1.1 CAESAR siffer 3
1.1.2 DES 4
1.2 Offentlig nøkkel kryptosystemer 4
1.2.1 RSA 5
1.3 Digitale signaturer 5
1.3.1 Kryptering versus digital signering 6
1.3.2 En modell for digital signering 6
1.4 Protokoller 7
1.4.1 Symmetrisk nøkkel signerings protokoll 7
1.4.2 Offentlig nøkkel signerings protokoll 9
1.5 Digital Signature Standard 9
1.5.1 Bakgrunn for DSS 10
1.5.2 Oversikt over DSS/DSA 10
1.5.3 Hastighet på DSS versus RSA 11
1.5.4 Sikkerhet 11
1.6 PGP 11

2 Hash-funksjoner 13

2.1 Hvorfor benytte hashfunksjoner 13
2.2 Definisjoner 14
2.2.1 One Way Hash Function - OWHF 14
2.2.2 Collision Resistant Hash Function - CRHF 14
2.2.3 Message Autentication Code - MAC 14
2.3 Annen bruk av hash funksjoner 15
2.3.1 Ikke kryptografiske hashfunksjoner 15
2.3.2 Passord 15
2.3.3 Virus/Beskyttelse av software 15
2.3.4 Kryptering 16
2.4 Grunnleggende elementer i hash funksjoner 16
2.4.1 Bits og bytes 16
2.4.2 Bitvise logiske operasjoner: 17
2.4.3 Konkatenering 18
2.4.4 Padding 18
2.4.5 Splitting 18
2.4.6 Iterasjon 19
2.4.7 Trunkering 19
2.5 Krav til iterasjonsfunksjonen 19

2.5.1 Avhengighet 19
2.5.2 Lineære faktorer 19
2.5.3 Sykliske funksjoner 20

2.6 Blokksiffer baserte hashfunksjoner 20

2.6.1 Rabins skjema 20
2.6.2 Blokksiffer lenke (CBC) skjema 21
2.6.3 Blokksiffer lenke (CBC) med sjekksum skjema 21
2.6.4 Kombinert Klar/kryptotekst lenke skjema 22
2.6.5 Nøkkel lenke skjema 22
2.6.6 Winternitz nøkkel lenke skjema 23
2.6.7 Quisquater og Girault 2n bits skjema 24
2.6.8 Merkles skjema 25
2.6.9 N-hash algoritmen 26
2.6.10 MDC2 og MDC4 27

2.7 Ikke Blokksiffer baserte hashfunksjoner 28

2.7.1 RSA blokksiffer lenke (CBC) skjema 29
2.7.2 Kvadrerings skjema 29
2.7.3 Claw-Free baserte skjema 31
2.7.4 Ryggsekk baserte skjema 31
2.7.5 Celluar Automata skjema 32
2.7.6 Matrise baserte skjema 33
2.7.7 FFT hashing 33

2.8 Angrep på hashfunksjoner 35

2.8.1 Angrep som er uavhengig av algoritmen 36
2.8.2 Angrep som er avhengig av lenkingen/chaining 36
2.8.3 Angrep avhengig av signatur skjemaet 38
2.8.4 Angrep basert på blokk sifferet 38
2.8.5 Høynivå angrep 38

2.9 Enkle hash funksjoner 39

2.9.1 XOR-hash 39
2.9.2 Hash-16 40

2.10 Praktisk utprøving 40

2.10.1 Angrep 40
2.10.2 Statistiske vurderinger 42

3 Dedikerte Hash-funksjoner 45

3.1 Snefru 45

3.1.1 Oversikt 45
3.1.2 Sikkerhet 46

3.2 MD4 47

3.2.1 Oversikt 47
3.2.2 Sikkerhet 48

3.3 MD5 49

3.3.1 Oversikt 49
3.3.2 Sikkerhet 50

3.4 SHA 50
3.4.1 Oversikt 50
3.4.2 Sikkerhet 52
3.5 RIPEMD 52
3.5.1 Oversikt 53
3.5.2 Sikkerhet 54
3.6 HAVAL 54
3.6.1 Oversikt 54
3.6.2 Sikkerhet 56
3.7 Hastighet 56
3.7.1 Kompilering av hashfunksjoner 57
3.8 Implementering på 64 bits arkitektur 58

4 Statistiske vurderinger av SHA og MD5 59

4.1 Pseudorandomtall og hashfunksjoner 59
4.2 Vurdering av maskinens pseudorandomfunksjon 60
4.3 Praktisk utføring 63
4.4 Resultater SHA 68
4.4.1 MessageIndexPart 0, Index 0, 8 bits 68
4.4.2 MessageIndexPart 2, Index 0, 16bits 69
4.4.3 MessageIndexPart 3, Index 1, 4 bits 70
4.5 Resultater MD5 70
4.5.1 MessageIndexPart 0, Index 0, 8 bits 71
4.5.2 MessageIndexPart 2, Index 0, 16bits 71
4.5.3 MessageIndexPart 3, Index 1, 4 bits 72
4.6 Konklusjon 72

Referanser 73

Appendiks 79

A DES 79
B RSA 81
B.1 Sikkerhet/hastighet RSA 82
C Praktisk bruk av PGP 84
C.1 Kommandoer i PGP 2.6.2i 84
D Kompleksitet 87
D.1 O Notasjonen 87
E Tallteori 87

Programlistinger 89

A XOR-Hash 89
B Hash-16 90
C SHA 101
D MD5 116

Innledning

Motivasjon

I dagens samfunn foregår mye av den daglige kommunikasjonen via større og mindre nettverk. Det er et økende behov for både kryptering og autentisering. Da det etterhvert er store mengder data som overføres er det essensiellt at effektiviteten er på topp. Hashfunksjoner er et viktig bidrag for å øke hastigheten og forbedre sikkerheten ved bruk av digitale signaturer. Da det ikke er ønskelig å innføre ekstra svakheter for å øke hastigheten, er det viktig at hashfunksjonene også holder mål rent sikkerhetsmessig.

Målsetting

Denne hovedfagsoppgaven har som mål å gi en innføring i hvordan hashfunksjoner fungerer og deres bruksområde. Den vil vurdere hastighet og sikkerhet på flere av de dedikerte kryptografiske hashfunksjonene som er i bruk i dag.

Form

Oppgaven består av 4 kapittel. Kapittel 1 gir en generell innføring i klassisk og offentlig nøkkel kryptografi, samt en kort innføring i digitale signaturer og enkle protokoller. Kapittel 2 er en innføring i hashfunksjoner, generelle definisjoner og hvordan de er bygget opp. Kapittel 3 vurderer og gir en generell innføring i de forskjellige dedikerte kryptografiske hashfunksjoner med hensyn på bl.a. sikkerhet. Kapittel 4 gir statistiske vurderinger av de to mest vanlige kryptografiske hashfunksjonene som er i bruk i dag, SHA og MD5. Jeg gir også en komplett utlisting av de to mest vanlige kryptografiske hashfunksjonene MD5 og SHA, inkludert mine modifikasjoner og tillegg. I Appendikset er er det en mer grundig innføring i DES, RSA og PGP, samt en kort innføring i kompleksitet og tallteori.

Forutsetninger

Mine forutsetninger for å gjennomføre denne oppgaven er en ingeniørutdannelse i elektronikk/telematikk fra Agder Ingeniør og Distriktshøgskole, samt en Cand. Mag. grad i informatikk ved Universitetet i Bergen.

Notasjon

I teksten benyttes generelle notasjoner, kjent fra litteratur angående kryptografi. Jeg har valgt å beholde endel engelske notasjoner, da oversettelse av disse vil kunne skape konflikter med den gjeldende litteratur.

Kapittel 1 Kryptering og Digitale signaturer

Formålet med dette kapittelet er å gi en generell innføring i kryptografi, slik at leseren har mulighet for å forstå begrepene som blir brukt senere i oppgaven. Jeg vil gi eksempler på klassiske kryptosystemer, offentlig nøkkel kryptosystemer og digitale signaturer, dette leder etterhvert mot hovedemnet i denne oppgaven, hash-funksjoner for bruk i Digitale Signaturer.

1.1 Klassiske kryptosystemer

Grunnlaget for kryptografi i klassisk forstand er behovet for å holde noe hemmelig. Vi kan tenke oss en situasjon hvor en vil sende en melding fra Alice til Bob og hvor de ikke vil at noen andre skal få se hva denne meldingen inneholder. Alice har en melding m og bruker en hemmelig nøkkel for å lage en kryptert melding c. Denne meldingen er umulig å lese dersom du ikke har nøkkelen. Alice sender denne meldingen c til Bob som bruker nøkkelen for å dekryptere, det vil si finne tilbake til meldingen m. For å få dette til må begge ha en nøkkel, det vil si informasjon som kun Alice og Bob har og som ingen andre har tilgang til. Vi kan tenke oss at denne informasjonen er et tall eller et ord. Jeg skal nå vise hvordan dette kan realiseres. Som et eksempel bruker jeg et gammelt velkjent kryptosystem, et CAESAR siffer.

1.1.1 CAESAR siffer

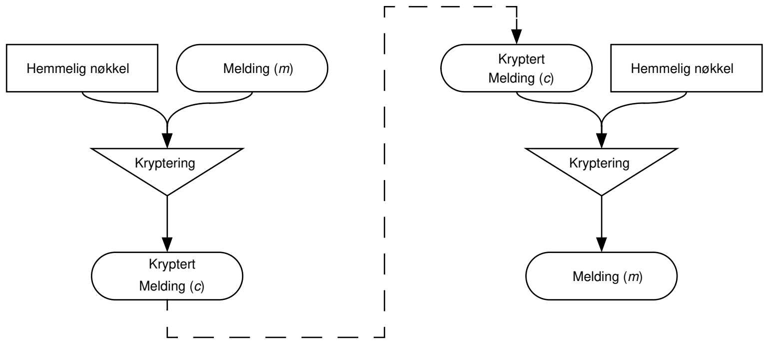
CAESAR siffer er en krypteringsmetode som bytter ut hver enkelt bokstav med en annen i et gitt mønster.
Vi kan tenke oss to alfabeter hvor vi forandrer det nederste, her en forskyvning på 3, k=3, som er det tradisjonelle CAESAR sifferet.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
```

Teksten “DETTE ER EN TEKST” vil da bli kryptert til “GHWWH HU HQ WHVW”. I dette eksempelet vil den hemmelige informasjonen være at det er en tradisjonell CAESAR som brukes. En annen variant av CAESAR vil være at en bytter ut de første bokstavene i alfabetet med et ord og “fyller etter” med de gjenværende bokstavene på følgende måte:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
K R Y P T O A B C D E F G H I J L M N Q S U V W X Z
```

Teksten “DETTE ER EN TEKST” vil da bli kryptert til “PTQQT TM TH QTENQ”. Å benytte denne metoden for å kryptere meldinger er ikke ideel. En har problemet med hvordan en skal overføre nøkkelen (k=3 og at det er et CAESAR siffer) til mottageren. Denne overføringen må skje over sikre kanaler, for eksempel ved hjelp av en kurer eller over et sikkert samband. Nøkkelen må også sendes på forhånd, da en ellers like så godt kunne sendt meldingen over det sikre sambandet. CAESAR sifferet og lignende kryptosystemer er også i seg selv usikre, da en kan benytte frekvenstelling for å knekke kryptosystemet. Frekvenstelling benytter seg av det faktum at i et språk er det bokstaver, eller kombinasjoner av bokstaver som opptrer oftere enn andre. Prinsippet bak symmetriske/klassiske kryptosystemer er vist i Figur 1.1.



Figur 1.1 *Klassisk kryptosystem (symmetrisk nøkkel)*

1.1.2 DES

Det mest kjente klassiske kryptosystemet i dag er DES [FIPS46]. DES ble utviklet tidlig på 70 tallet i USA. Algoritmen var ment å være en standard krypteringsalgoritme, som i seg selv var ganske oppsiktsvekkende, da krypterings-algoritmer tidligere skulle holdes hemmelig for enhver pris. Mange av metodene som benyttes i DES finner en igjen i hashfunksjonene. I DES benytter bl.a. S-bokser (Snefru), Høyre/Venstre deling (RIPE-MD), ekspandering (SHA) og rotering. For angrep på DES som er relevant for hashskjemaer henvises til kapittel 2.8.4. Henviser ellers til Appendiks A.

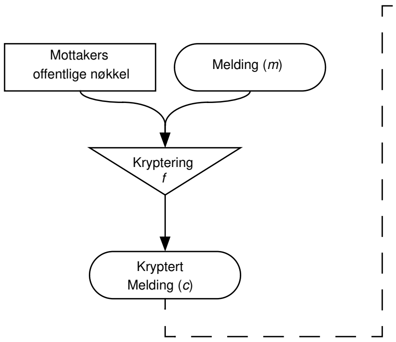
1.2 Offentlig nøkkel kryptosystemer

I 1977/78 presenterte Diffie og Hellman en revolusjonerende metode innenfor kryptografien [DiH76]. Denne metoden går ut på at en kan la krypterings metoden være kjent, d.v.s. at en har en *offentlig nøkkel*, det finnes også en *privat nøkkel* som er hemmelig. Ideen bak offentlig nøkkel kryptografi er nært knyttet til begrepet enveis funksjoner. En enveis funksjon er en funksjon hvor det er lett å finne $f(x)$ hvis en har x, men vanskelig å finne $f^{-1}(y)$ hvis en har y. Som et eksempel kan en nevne en telefonkatalog, det er lett å finne et telefonnummer hvis en kjenner navnet, men det er vanskelig å finne navnet hvis telefon nummeret er kjent.

$$x \quad \underset{\textit{vanskelig } f^{-1}(y)}{\overset{\textit{lett } f(x)}{\rightleftarrows}} \quad y$$

Figur 1.2 *Enveis funksjon*

Offentlig nøkkel kryptosystemer løser mange av problemene som oppstår ved bruk av klassisk kryptografi. En slipper problemene med nøkkelutveksling og en kan opprette en sentral nøkkel katalog.



Figur 1.3 *Offentlig nøkkel kryptosystem (asymmetrisk kryptosystem)*

1.2.1 RSA

RSA ble oppfunnet av Ron Rivest, Adi Shamir og Leonard Adleman [RSA78]. Den er pr. i dag det mest populære offentlig nøkkel kryptosystemet, det benyttes blant annet i PGP, jfr kapittel 1.7 og WWW verktøyet Netscape. RSA kan benyttes både til offentlig nøkkel kryptering og til digital signering av meldinger. RSA baserer seg på at det er vanskelig å faktorisere store tall. Den offentlige og den private nøkkelen er begge funksjoner av par av store (100 til 200 siffer) primtall. Vanskeligheten med å finne klarteksten utifra den krypterte teksten skal tilsvare vanskeligheten ved å faktorisere produktet av disse primtallene. RSA kan også benyttes for å signere meldinger. Prinsippet for denne signeringen blir vist i Kapittel 1.5.2. For nærmere informasjon om RSA henvises til Appendiks B.

1.3 Digitale signaturer

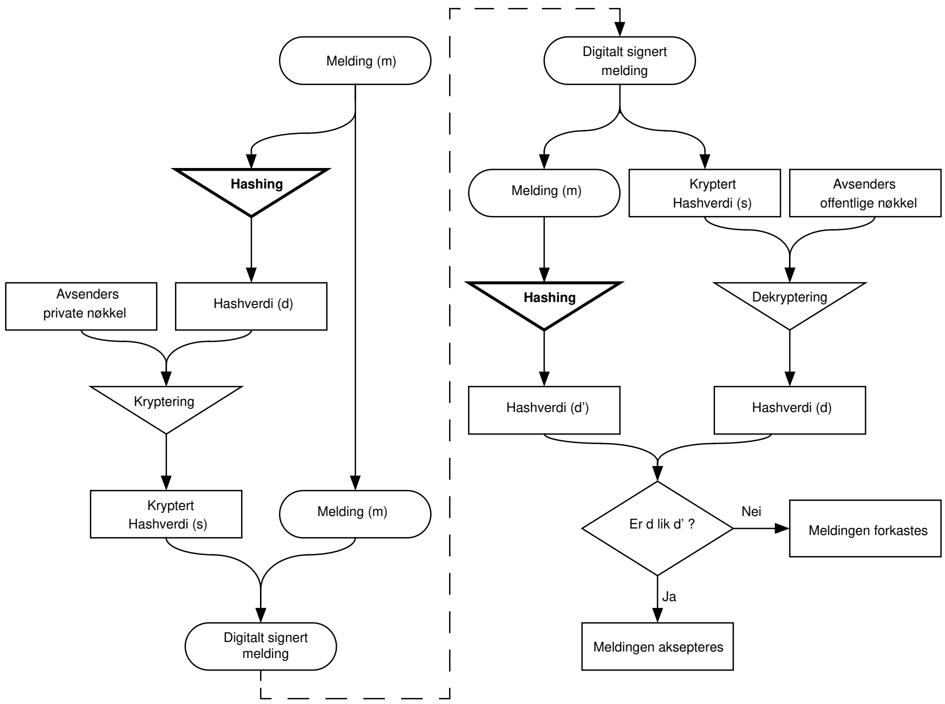
Grunnlaget for begrepet digitale signaturer er signaturbegrepet som vi benytter i dagligtalen. Vi snakker om at vi signerer et dokument, det vil si at hvis Alice har signert en melding kan alle verifisere at meldingen ble signert av henne. F.eks. vil vi når vi signerer en kontrakt godta det som står i den og underskriften vil ved en senere tvist være bindende for den som har signert. Problemet synes å være tilnærmet uløselig hvis en tenker seg den elektroniske varianten av et dokument. Når en signerer et papir vil blekket feste seg til papiret og det vil være vanskelig å fjerne/forfalske signaturen uten å skade papiret. Å gjøre det samme elektronisk, vil ikke være like innlysende. Det er ikke mulig bare å legge til et navn på elektronisk form (Konkatenere melding og signatur, j.f.r. 2.4.3), da hvem som helst vil være i stand til å bytte den ut med et annet navn. En mulighet vil være å benytte klassisk kryptografi, hvor kun sender og mottaker er i stand til å signere en melding korrekt. Problemet er da at både sender og mottaker vi kunne signere meldingen og signeringen vil derfor ikke være unik. En måte å unngå dette problemet er å benytte en tredjepart som begge stoler på. Framgangsmåten for dette vil bli omtalt i kapittel 1.5.1. I praksis foregår ofte signering ved at en “pakker” inn meldingen og signerer slik at det ikke er mulig å skille signaturen fra meldingen.

1.3.1 Kryptering versus digital signering

Det er enkelt å blande sammen begrepene signering og kryptering av elektroniske dokumenter. Kryptering brukes for å gjøre klartekst uforståelig for uvedkommende. Kryptering er kun nødvendig dersom dokumentets innhold skal holdes hemmelig. Signeringen er en rutine for å sikre dokumentenes autentitet og inkluderer ikke automatisk kryptering av selve meldingen. Signeringen må være en del av dokumentet slik at innholdet ikke kan endres etter at meldingen er signert. Det kan også tenkes at det ikke er uvedkommende som prøver å forandre innholdet, men for eksempel et virus eller rett og slett en lesefeil fra en disk. Prinsippene bak signering og autentisering er uansett de samme.

1.3.2 En modell for digital signering

En typisk implementasjon av digitale signaturer involverer en hashalgoritme, som produserer en hashverdi av en melding *m* og en offentlig nøkkelalgoritme for kryptering av meldingsekstraktet. Fordelene med å benytte hashfunksjoner i forbindelse med Digitale Signaturer vil bli omtalt i Kapittel 2.1. Vi benevner senderen som Alice og mottageren som Bob. Se også Figur 1.4. Alice reduserer meldingen til en hashverdi ved hjelp av en hashalgoritme. Dertetter krypterer hun hashverdien med sin private nøkkel. Resultatet er en kryptert hashverdi *s,* denne krypterte hashverdien fungerer da som en signatur til meldingen *m*. Hun sender så meldingen *m* og den krypterte hashverdien *s*, som samlet utgjør den digitalt signerte meldingen, til mottakeren Bob. Bob dekrypterer hashverdien *s* med Espens offentlige nøkkel, og får hashverdien *d*. Deretter beregner han, ut fra den mottatte meldingen *m*, en sammenlignbar hashverdi *d'* og sammenligner dette med hashverdien *d*. Dersom de to er identiske aksepterer han sendingen som autentisk. I motsatt fall, dersom de avviker, forkaster han meldingen.



Figur 1.4 *Digital signering av elektroniske meldinger*

1.4 Protokoller

Det sies at ingen lenke er sterkere en det svakeste ledd. Dette gjelder også i kryptografiens verden, det hjelper ikke å ha verdens sikreste passordsystem hvis brukerne skriver passordet på en lapp ved siden av terminalen. Det hjelper ikke å ha "sikre" signeringsalgoritmer hvis det er mulig for hvem som helst å signere andres meldinger. I dette avsnittet vil jeg ta for meg protokoller, det vil si sekvensielle skritt som to eller flere parter må gjennomgå for å utføre en oppgave. Til daglig gjennomfører vi flere protokoller, vi tenker ikke så mye over dem, da de er så innarbeidet i de daglige gjøremål. Som et eksempel kan du tenke deg situasjonen hvor vi gjør innkjøp: Først henter du varene, så går du til kassen, prisen på varene blir summert, du betaler med penger fra din egen lommebok, får en kvittering og forlater butikken. Dette er en protokoll som er utviklet gjennom mange år med handel. Hvis en tenker over det, vil en se at en til daglig benytter utallige protokoller, for eksempel når en treffer gamle kjente eller skal låne en bok på biblioteket. Det stilles endel generelle krav til protokoller, disse er som følger:

1. Alle de involverte parter må kjenne til den aktuelle protokollen på forhånd
2. Alle partene må være enige om å følge protokollen.
3. Alle deler/skritt i protokollen må være godt definert slik at misforståelser unngås.
4. Protokollen må være komplett, det vil si at protokollen må inneholde spesifiserte skritt for enhver mulig situasjon.

Det er i årenes løp laget flere forskjellige protokoller for forskjellige oppgaver. I denne hovedoppgaven vil det kun være aktuelt å se på kryptografiske protokoller, det vil si protokoller som benytter seg av kryptografi. Det er laget kryptografiske protokoller for å kaste mynt/krone over et samband, hemmelige valg, overføring av hemmeligheter og mange flere. I mange protokoller er det vesentlig at en unngår en tredjepart som begge stoler på. Jeg skal konsentrere meg om kryptografiske protokoller som brukes ved digital signering av meldinger. Da det meste av litteraturen omkring kryptografiske protokoller og kryptografi generellt, er skrevet på engelsk vil jeg benytte endel engelske konvensjonener når jeg beskriver protokollene. For å gjøre stoffet lettere tilgjengelig er det i litteraturen innført personer som representerer de legale og illegale partene som kan forekomme i et kommunikasjonssystem. Vi har de følgende parter:

Alice	Legal deltager i protokollene. Ofte initiativtaker til sendingen.
Bob	Legal deltager i protokollene. Ofte som mottager.
Carol, Dave	Også mulige legale deltagere i protokollen.
Eve	En (illegal) passiv lytter til protokollen.
Mallet	En (illegal) ondskapsfull aktiv angriper.
Trent	En tredjepart som alle stoler på.

I de påfølgende kapittel vil jeg omtale basisprotokollene for digital signering av meldinger som sendes mellom to parter.

1.4.1 Symmetrisk nøkkel signerings protokoll

Denne protokollen baserer seg på et symmetrisk nøkkel (klassisk) kryptosystem. I denne protokollen benyttes en tredjepart (Trent) som begge parter stoler på. Trent deler en hemmelig nøkkel K_A med Alice og en annen hemmelig nøkkel K_B med Bob. Disse nøklene har blitt etablert lenge før signeringen skal foregå og kan brukes flere ganger for å signere flere meldinger.

1. Alice krypterer meldingen til Bob med K_A og sender den til Trent.
2. Trent dekrypterer meldingen med K_A
3. Trent tar den dekrypterte meldingen, en bekreftelse på at han har mottatt meldingen fra Alice og den krypterte meldingen. Han krypterer hele denne pakken med K_B.
4. Trent sender denne pakken til Bob.
5. Bob dekrypterer med K_B. Han kan nå lese både meldingen og Trents sertifisering for at Alice har sendt meldingen

Hvordan vet Trent at meldingen er fra Alice, og ikke fra en forfalsker? Trent ved dette utifra den krypterte meldingen. Siden det er han og Alice som deler denne hemmelige nøkkelen, er det kun Alice som kan kryptere en melding ved hjelp av K_A. Er dette like godt som en signatur på et papir? La oss se nærmere på det:

1. Denne signaturen kan ikke forfalskes. Bare Alice (og Trent, som alle stoler på) kjenner K_A, slik at kun Alice kan ha sendt Trent en melding kryptert med K_A. Hvis noen prøvde å late som de var Alice, ville trent gjennomskue dette i Steg 2 og ville ikke sende meldingen til Bob.

2. Signaturen er autentisk. Trent er en som alle stoler på og Trent vet at meldingen kom fra Alice. Trent sin sertifisering er bevis godt nok.

3. Signaturen kan ikke brukes flere ganger. Hvis Bob prøvde å benytte Trent sin sertifisering og bruke den på en annen melding, ville Alice protestere på dette. Trent ville spørre Bob om å produsere både meldingen og Alices krypterte melding. Trent kan da kryptere meldingen med K_A og se at den ikke var sammenfallende med den krypterte meldingen som Bob ga ham. Bob kan ikke lage en kryptert melding som Trent vil godta, da han ikke kjenner K_A.

4. Det signerte dokumentet kan ikke forandres. Hvis Bob prøver å forandre på dokumentet etter å ha mottatt det, vil Trent kunne bevise juks på samme måte som ovenfor.

5. Signaturen kan ikke benektes. Hvis Alice senere påstår at hun ikke har sent meldingen vil Trents sertifisering bevise det motsatte.

Hvis Bob vil vise Carol et dokument som er signert av Alice, kan han ikke avsløre sin hemmelige nøkkel K_B til henne. Han må gå via Trent igjen:

1. Bob tar meldingen og bekreftelsen på at meldingen kom fra Alice, krypterer dem med K_B og sender dem tilbake til Trent.
2. Trent Dekrypterer disse med K_B.

```
3.      Trent krypterer med den hemmelige nøkkelen med den hem-
        melige nøkkelen han deler med Carol, K_C, og sender den
        til Carol.

4.      Carol dekrypterer med K_C. Hun kan nå lese både meldin-
        gen og Trents sertifisering på at Alice var avsender.
```

Disse protokollene virker, men de er meget tidkrevende for Trent. Han må bruke tiden til å dekryptere og kryptere meldinger mellom alle par av sendere og mottakere som vil sende signerte dokumenter til hverandre. Trent vil etterhvert bli en flaskehals i kommunikasjonssystemet. Et annet problem er hvordan en lager og vedlikeholder noen som Trent. Trent må være ufeilbarlig, hvis han gjør feil i bare 1 promille av signaturene, vil ingen stole på ham. Tidligere signerte meldingen må kanskje kasseres, det er lett å forestille seg at det vil kunne skape kaos.

1.4.2 Offentlig nøkkel signerings protokoll

Denne protokollen baserer seg på et offentlig nøkkel krypto system. Jeg benytter RSA som eksempel, men det er også mulig å basere på andre offentlig nøkkel kryptosystem, disse må være kommutative. Ideen ble først oppfunnet av Diffie og Hellman [DiH76] og videreutviklet av andre senere. Basisprotokollen er som følger:

```
1.      Alice bruker en digital signatur algoritme med hennes
        private nøkkel for å signere meldingen.

2.      Alice sender dokumentet til Bob.

3.      Bob bruker den digitale signatur algoritmen med Alice
        sin offentlige nøkkel for å verifisere signaturen.
```

Denne protokollen er bedre enn den forrige. Signeringen er uavhengig av Trent, Alice og Bob kan klare det selv. Bob trenger ikke engang Trent for å løse konflikter, hvis han ikke kan utføre steg 3, vet han at signaturen ikke er gyldig. Denne protokollen tilfredsstiller også kravene vi stiller til en digital signerings protokoll:

1. Signaturen kan ikke forfalskes, bare Alice kjenner hennes egen private nøkkel.
2. Signaturen er autentisk, når Bob verifiserer meldingen med Alices offentlige nøkkel, vet han at hun har signert den.
3. Signaturen kan ikke benyttes flere ganger, signaturen er en funksjon av dokumentet og kan ikke bli overført til et annet dokument.
4. Signaturen kan ikke forandres, hvis dokumentet forandres kan det ikke lenger bli verifisert med Alice sin offentlige nøkkel.
5. Signaturen kan ikke benektes. Bob trenger ikke Alice sin hjelp for å verifisere signaturen.

1.5 Digital Signature Standard

The National Institute of Standards and Technology (NIST) fremla i 1994

Federal Information Processing Standard (FIPS) 186, Digital Signature Standard (DSS)[FIPS186]. I dette kapittelet vil jeg ta for meg denne standarden nokså grundig da Secure Hash Algorithm (SHA) [FIPS180] som omtales i Kapittel 3.4 er en del av denne. Dette kapittelet vil stort sett omhandle Digital Signature Algorithm som er signerings algoritmen som er definert i DSS. Se ellers Figur 1.4.

1.5.1 Bakgrunn for DSS

For å øke produktiviteten og minske kostnadene ville mange statlige byrå i USA gå over fra papir baserte systemer til elektroniske varianter. Det oppsto da et behov for et sikkert, pålitelig system for å erstatte den håndskrevne signaturen med en digital signatur. Dette førte til at det ble foreslått en standard for digitale signaturer, DSS. Signaturalgoritmen i DSS kalles DSA.

1.5.2 Oversikt over DSS/DSA

DSA bygger på ElGamals signaturskjema [ElG85], men er utvidet slik at en 160 bits melding (hashverdien fra SHA) signeres ved å benytte en 320 bits signatur. Dette gjøres ved å bruke en 512 bits modulus p. DSA spesifiseres videre med følgende parametre:

1. p = en primtalls modulus hvor $2^{L-1} < p < 2^L$ for $512 \leq L \leq 1024$ og hvor L er et multippel av 64.
2. q en primtalls divisor av p - 1, hvor $2^{159} < q < 2^{160}$
3. $g= h^{(p-1)/q}$ mod p hvor h er et heltall hvor $1 < h < p - 1$ slik at $h^{(p-1)/q}$ mod $p > 1$
4. x = et random eller pseudorandom generert heltall hvor $0 < x < q$
5. $y = g^x$ mod p
6. k et random eller pseudorandom generert heltall hvor $0 < k < q$

Heltallene p, q og g kan være offentlige og kan være felles for en gruppe brukere. En brukers private og offentlige nøkkel er x og y respektivt. Normalt er disse brukt kun i en begrenset periode. Parametrene x og k brukes kun for å lage signaturene og må holdes hemmelig. Parameteren k må genereres på nytt for hver enkelt signatur.

1.5.2.1 Signatur generering:

Signaturen av en melding M er paret av tallene r og s som blir regnet ut som følger:

$$r = (g^k \bmod p) \bmod q \text{ og}$$
$$s = (k^{-1}(\mathrm{SHA(M)} + xr)) \bmod q.$$

I de ovenstående ligningene er k^{-1} den multiplikative invers av k mod q. Verdien av SHA(M) er en 160 bits streng som genereres av Secure Hash Algorithm (SHA), jfr. Kapittel 3.4.

1.5.2.2 Signatur verifikasjon:

Før en kan verifisere en signatur må p, q, g og senderens offentlige nøkkel og identitet være gjort tilgjengelig.
La M', r' og s' være de mottatte versjonene av M, r og s. La y være den

offentlige nøkkelen til den som signerer. For å verifisere signaturen sjekker mottageren at $0 < r' < q$ og $0 < s' < q$, hvis en eller begge disse ikke stemmer skal ikke signaturen godkjennes. Hvis disse to stemmer overens regner mottageren ut:

$$w = (s')^{-1} \bmod q$$
$$u1 = ((\mathrm{SHA}(M'))w) \bmod q$$
$$u2 = ((r')w) \bmod q$$
$$v = (((g)^{u1}(y)^{u2}) \bmod p) \bmod q$$

Hvis $v = r'$ er signaturen verifisert og kan mottageren ha rimelig tiltro til at meldingen var sendt av den som har den hemmelige nøkkelen. Hvis v ikke tilsvarer r' kan meldingen ha blitt modifisert, meldingen kan ha blitt signert feil av senderen eller den kan ha blitt signert av en annen. Meldingen må sees på som falsk.

1.5.3 Hastighet på DSS versus RSA

En av de største ankepunktene mot DSS er at den ikke er like effektiv som RSA, hvis denne benyttes til signering. Å lage en signatur er like raskt når en benytter DSS som RSA mens signatur verifisering er 10-40 ganger mindre effektivt når en benytter DSS. For å øke hastigheten er det mulig å regne ut enkelte parametere i DSS på forhånd, da disse er ikke meldingsavhengige.

1.5.4 Sikkerhet

Sikkerheten i et offentlig nøkkel kryptosystem, slik som DSS, er bestemt av flere faktorer. Det er viktig at algoritmen rent matematisk holder mål, at nøkkelhåndteringen foregår på en forsvarlig måte og at systemet er implementert på en sikker måte. Sikkerheten til DSA baserer seg på at det er praktisk umulig å finne den diskrete logaritmen til et stort tall. Det skal derfor være praktisk umulig å generere en falsk signatur hvis en ikke kjenner de private parametrene x og k. I det opprinnelige forslaget til standard var nøkkel størrelsen på 512 bits, dette vil muligens være for lite på lang sikt. Den endelige versjonen av DSS inneholder mulighet for å øke nøkkellengden til 1024 bits. Et annet ankepunkt angående DSS er at NSA (National Security Agency) har gjort en vesentlig del av utviklingsarbeidet og det har derfor blitt spekulert i om det er lagt inn en bakdør slik at NSA har muligheten for å lage falske signaturer, det er ikke funnet noen bevis for denne hypotesen og det er vel heller ikke så veldig sannsynlig, da hvem som helst kan teste algoritmen (DSA).

1.6 PGP

PGP (Pretty Good Privacy) ble laget i 1991 av Philip Zimmermann som en løsning på problemet med å kunne signere post som skal sendes over Internet. PGP er en hybridløsning som benytter seg av både symmetrisk og offentlig nøkkel kryptosystemer. PGP er en generell algoritme som både kan benyttes til kryptering og til signering. En benytter MD5 til hashing, IDEA til kryptering og RSA til nøkkeloverføring. På grunn av USAs strenge regler for eksport av krypterings programmer finnes det pr. i dag to brukte varianter av PGP, 2.6.2 for amerikanske brukere og 2.6.2i for internasjonale brukere. Det kan også nevnes at Philip Zimmermann er tiltalt i USA på grunn av PGP.

For en fyldig dokmentasjon om PGP henvises til Philip Zimmermanns [Zim94] egen dokumentasjon. Henviser ellers til Appendiks C.

Kapittel 2 Hash-funksjoner

2.1 Hvorfor benytte hashfunksjoner

Som det ble nevnt i innledningen er det primært på grunn av hastigheten at en er interessert i å benytte hashfunksjoner i forbindelse med digitale signaturer. Isteden for å signere hele meldingen, signerer en hashverdien av meldingen. Da hashverdien kan fås entydig fra meldingen er dette mulig.
I tillegg til en øket hastighet er det også flere fordeler med å benytte hashfunksjone i forbindelse med digitale signaturer. De viktigste fordelene er som følger [Pre93]:

1 Størrelsen av signaturen kan reduseres fra å ha samme eller større lengde som informasjonen, til en fast blokklengde, uavhengig av lengden på meldingen. Hvis man ikke benytter en hashfunksjon (MDC) er det selvfølgelig mulig å lagre bare den signerte informasjonen og å regne ut informasjonen fra signaturen når det behøves, men denne løsningen blir lite effektiv.

2. Signerings og verifiserings -funksjoner er vesentlig mindre effektive enn symmetriske krypteringsfunksjoner, slik som hashfunksjoner. En parallell er forskjellen mellom klassisk og offentlig nøkkel kryptering, for eksempel er DES omtrent 100 ganger raskere enn RSA.

3. Hvis en ikke benytter en hashfunksjon og informasjonen som skal signeres er lengre en blokk, er det lettere å manipulere disse blokkene. Det enkleste eksempelet er å bytte om på rekkefølgen på disse blokkene.

4. Den algebraiske strukturen i meldingsmengden kan ødelegges. I tilfellet med RSA har meldingsmengden en multiplikativ struktur, med andre ord, signaturen til produktet av to meldinger tilsvaret produktet av signaturene. En tilsvarende sammenheng finnes også i ElGamal.

5. Problemet med reblokking kan unngås. Dette problemet oppstår når både kryptering og autentisering blir ivaretatt av RSA. Hvis senderen først krypterer meldingen med den offentlige nøkkelen til mottageren (for kryptering), kan resultatet bli større en senderens modulus. Før han kan benytte sin hemmelige nøkkel (for signering), må han ta modulusen av resultatet av krypteringen. Det er egentlig ikke vanskelig å unngå dette, da en kan la rekkefølgen på operasjonene (kryptering/signering) være avhengig av størrelsen på modulusene. Det er dog en fordel å unngå dette problemet, for eksempel hvis en skal standardisere fremgangsmåtene.

6. Dersom en benytter en hashfunksjon vil ikke signeringsprotokollen være brukbar for en opponent som prøver å finne klarteksten som tilsvarer den krypterte meldingen. Dette problemet vil bare oppstå dersom en benytter den samme nøkkel for både kryptering og autentisering.

2.2 Definisjoner

En hash funksjon er en funksjon som komprimerer et input av vilkårlig lengde til et resultat av fast lengde. Hvis hashfunksjonen tilfredsstiller endel krav i tillegg, kan de brukes som et verktøy for å autentisere informasjon. Hash funksjonen kan da brukes som en del av et skjema for digitale signaturer. En skiller mellom hash funksjoner hvor en hemmelig nøkkel inngår, denne kalles for Message Autentication Code (MAC), og hash funksjoner uten hemmelig nøkkel, denne kalles for Manipulation Detection Code (MDC). En MDC kan igjen deles inn i to, nemlig One Way Hash Function (OWHF) og Collision Resistant Hash Funktion (CRHF). Jeg vil nå gi den formelle definisjonen på OWHF, CRHF og MAC.

2.2.1 One Way Hash Function - OWHF

4 egenskaper som en hash funksjon h må tilfredstille [Mer79, Rab78]

1. Beskrivelsen av h må være allment tilgjengelig og det må ikke være nødvendig med noen hemmelig informasjon for å bruke den.
2. Argumentet X kan være av uavhengig lengde og resultatet $h(X)$ har en fast lengde av n bits (hvor $n \geq 64$).
3. Gitt h og X, å regne ut $h(X)$ må være enkelt i praksis.
4. Hash funksjonen må være en-veis, dvs at ved en gitt Y, som er enn hash er det praktisk umulig å finne en melding X slik at $h(X) = Y$ og gitt en X og $h(X)$ er det "vanskelig" å finne en melding $X' \neq X$ slik at $h(X') = h(X)$.

2.2.2 Collision Resistant Hash Function - CRHF

5 egenskaper som en CRHF må tilfredstille [Dam88]

1. Beskrivelsen av h må være allment tilgjengelig og det må ikke være nødvendig med noen hemmelig informasjon for å bruke den.
2. Argumentet X kan være av uavhengig lengde og resultatet h(X) har en fast lengde av n bits (hvor $n \geq 128$)
3. Gitt h og X, å regne ut h(X) må være enkelt i praksis.
4. Hash funksjonen må være en-veis, dvs at ved en gitt Y, som er enn hash er det praktisk umulig å finne en melding X slik at $h(X) = Y$ og gitt en X og $h(X)$ er det praktisk umulig å finne en melding $X' \neq X$ slik at $h(X') = h(X)$.
5. Hash funksjonen må være kollisjons resistent, dvs. at det er "vanskelig" å finne to distinkte meldinger som hasher til det samme resultat.

2.2.3 Message Autentication Code - MAC

4 egenskaper som en MAC må tilfredstille [Pre93].

1. Beskrivelsen av h må være allment tilgjengelig og den eneste hemmelige informasjonen ligger i nøkkelen, K.
2. Argumentet X kan være av uavhengig lengde og resultatet $h(K, X)$ har en fast lengde av n bits (hvor $n \geq 32...64$)
3. Gitt h, X og K, å regne ut $h(K, X)$ må være enkelt i praksis.
4. Gitt h og X, må det være praktisk umulig å bestemme $h(K, X)$ med en sannsynlighet for suksess mye større enn $1/2^n$. Til og med når et stort antall par $\{X_i, h(K, X_i)\}$ en kjent, hvor X_i har blitt valgt av opponenten, er det praktisk umulig å bestemme nøkkelen K eller å regne ut $h(K, X')$ for hvilken som helst $X' \neq X_i$.

Dette betyr at en MAC skal være både en-veis og kollisjons resistent for noen som ikke kjenner nøkkelen, K. Denne definisjonen gir ikke noe svar på om MACen skal være en-veis og kollisjons resistent for noen som kjenner nøkkelen, K.

2.3 Annen bruk av hash funksjoner

Det er tidligere omtalt hvorfor hashfunksjoner bør brukes i forbindelse med digitale signaturer,jfr.Kapittel 2.1. Det er primært denne bruken det vil bli fokusert på i denne hovedoppgaven. Jeg vil dog i dette kapittelet nevne andre bruksområder hvor en kan benytte hash-funksjoner. Jeg vil komme nærmere inn på bruken av hash funksjoner i forbindelse med passord, beskyttelse av software, virus og kryptering.

2.3.1 Ikke kryptografiske hashfunksjoner

Det mest kjente bruksområdet for hashfunksjoner i dag er vel bruken av ikke kryptografiske hashfunksjoner. Disse hashfunksjonene brukes for å fordele meldinger i et gitt område av minnet. Hashfunksjonen må være konstruert slik at meldingene fordeler seg jevnt i minneområdet, dette for å få færrest mulige kollisjoner. I motsetning til kryptografiske hashfunksjoner kan disse ha kollisjoner. Det har ingen betydning bortsett fra at lagringen i minne vil være vesentlig mindre effektivt, da en enten må finne en ny minnelokasjon eller legge dem i en liste/tabell som starter i den aktuelle tabellposisjonen

2.3.2 Passord

Dette bruksområdet var vel det første hvor hash funksjoner ble benyttet i praksis. Det var tidlig i datamaskinenes barndom et problem at passordene som brukerne skrev inn ble sammenlignet med en passordfil, denne passordfilen ble lagret som en ren tekst eller v.h.a. en enkel krypterings algoritme. Denne passordfilen var selvfølgelig skjult for allmen tilgang, men generelle sikkerhetsproblemer gjorde at denne ble tilgjengelig. Dette kunne da føre til at en inntrenger fikk tilgang til alle brukernes konto, uten at dette kunne oppspores. Ved å bruke en hash funksjon vil bruken av passord bli sikrere, da en ikke trenger å lagre passordet i klartekst, men lagrer hashverdien av passordene. Når en bruker skriver inn passordet, vil passordet bli hashet og sammenlignet med passordene på filen, hvis den tilsvarende hasverdien blir funnet, blir passordet godtatt. Denne varianten er heller ikke sikker, en kan tenke seg at en inntrenger prøver seg frem med flere passord fra en ordliste, hasher og sammenligner. Hvis en bruker da har valgt et enkelt passord vil dette passordet kunne bli tilkjennegjort. En annen mulighet er hvis en inntrenger klarer å bytte ut hashfunksjonen og passordfilen, men dette vil være arbeidskrevende. Sikkerheten i denne bruken av en hashfunksjon baserer seg på at hashfunksjonen er en OWHF, da en inntrenger ellers ville kunne finne passordet utfra passordfilen ved å reversere hashfunksjonen.

2.3.3 Virus/Beskyttelse av software

Vi kan tenke oss en situasjon hvor vi har en forfatter av et software program og en bruker av det samme programmet. Dette programmet skal beskyttes mot modifisering, f.eks. et virusangrep eller en konkurrent som vil ødelegge forfatterens renomme. I dette tilfellet er det klart at det ikke er behov for hemmeligholdelse. En kan tenke seg at forfatteren bruker en av de kjente hashfunksjonene for å lage en hashverdi for det aktuelle programmet, hashfunksjonen kan brukes både på en kjørbar kode og på en evt. kildekode. Denne haskoden blir så distribuert uavhengig av programmet, slik at en kan sammenligne denne verdien med en hashverdi som blir regnet ut lokalt. F.eks. kan man trykke hashverdien i manualen eller gjøre den tilgjengelig via Internett. Det er en forutsetning at en ikke sender med en hashfunksjon sammen med programmet, da en evt. modifikator også vil kunne modifisere hashfunksjonen. Det er også mulig å sende hashverdien via et offentlig nøkkel kryptosystem, jfr. Kap. 1.3.

2.3.4 Kryptering

Da det ikke skal være mulig å reversere resultatet fra en hashfunksjon er det ikke mulig å benytte en hashfunksjon direkte for kryptering. Det er mulig å benytte deler av hashfunksjonen som en klassisk krypteringsalgoritme. I praksis vil en kunne bruke iterasjonsfunksjonen som en DES lignende algoritme. Henviser ellers til Kapittel 2.6 hvor sammenhengen mellom blokksiffer og noen hashfunksjoner er beskrevet.

2.4 Grunnleggende elementer i hash funksjoner

De fleste software hash funksjoner er bygget opp ved hjelp av standard elementer. Disse elementene blir kombinert med konstanter og tabeller for å lage den konkrete hash funksjonen. For å øke forståelsen for hvordan en hash funksjon er bygget opp, vil jeg forklare de forskjellige elementene som brukes.

2.4.1 Bits og bytes

Et ord tilsvarer en 32-bits streng som kan representeres som en sekvens av 8 hexadesimale siffer. For å konvertere et ord til 8 hexadesimale siffer blir hver 4 bits streng konvertert til sin hexadesimale ekvivalent som beskrevet ovenfor.
Eksempel:

```
1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.
```

For å vise at tallet er et hexadesimalt tall bruker vi ofte prefikset 0x, dvs:

```
A103FE23 = 0xA103FE23
```

Denne måten å benevne hexadesimale tall vil bli benyttet når det er fare for forveksling.
Lengden n på en streng X benevnes ofte |X|.
Eksempel:

```
X = 1010 0001 0000 0011 1111 1110 0010 0011
n = |X| = 32
```

Et heltall mellom 0 og 2^{32}-1 kan representeres som et ord. Den minst signifikante 4 bits av heltallet representeres av det hexadesimale sifferet lengst til høyre. F.eks. Heltallet 291 = 256 + 32 + 2 +1 = $2^8 + 2^5 + 2^1 + 2^0$ kan representeres som det hexadesimale ordet 0x00000123.

En blokk = 512 bits streng, som kan representeres av en sekvens av 16 ord.

Det sirkulære venstre skift S(*n, X*) , hvor *X* er et ord og *n* er et heltall $0 \leq n < 32$ er definert ved:

```
S(n,X) = (X « n) OR (X » 32 - n)
```

Som er ekvivalent med en sirkulær skift (rotering) av *X, n* posisjoner til venstre. Eksempel, *n* = 6:

```
X = 1001 0010 0010 1001 0010 1010 0010 0011
S(6,X) = 1000 1010 0100 1010 1000 1000 1110 0100
```

2.4.2 Bitvise logiske operasjoner:

X AND Y = bitvis logisk “og” av *X* og *Y*, her benevnes dette *XY*.
Denne benevnes også ofte $\wedge$.
Eksempel:

```
          01101100101110011101001001111011
AND       01100101110000010110100110110111
          --------------------------------
=         01100100100000010100000001111111
```

X OR *Y* = bitvis logisk “eller” av *X* og *Y*.
Denne benevnes også ofte $\vee$.
Eksempel:

```
          01101100101110011101001001111011
OR        01100101110000010110100110110111
          --------------------------------
=         01101101111110011111101111111111
```

X XOR *Y* = bitvis logisk “eksklusiv eller” av *X* og *Y*.
Denne benevnes også ofte $\oplus$.
Eksempel:

```
          01101100101110011101001001111011
XOR       01100101110000010110100110110111
          --------------------------------
=         00001001011110001011101111001100
```

NOT *X* = bitvis logisk “komplement” av *X*.
Denne benevnes også ofte $\neg$.

Eksempel:

```
X =        01101100101110011101001001111011
NOT X =    10010011010001100010110110000100
```

Denne benevnes også ofte med en strek over ordet/strengen på følgende måte:

NOT $X = \overline{X}$

Vi vil benytte denne benevningen når det er passende.

2.4.3 Konkatenering

Å konkatenere bit strenger vil si å legge dem etter hverandre, tegnet || benyttes for å symbolisere dette. Dersom både *X* og *Y* har lengden *n* vil den konkatenerte strengen *X* || *Y* ha lengden 2*n*.

```
X =        0110110010111001
Y =        1101001001111011
X || Y =   01101100101110011101001001111011
```

2.4.4 Padding

Inputstrengen blir fylt med bits for å sikre at strengen er et multippel av *m*, slik at strengen kan deles i like deler. Dette sikrer at en melding hasher til den samme verdien hver gang, da den ikke er avhengig av "tilfeldige" bits som blir liggende i minnet. I hashfunksjoner kan en benytte forskjellige metoder for padding, de følgende metodene for padding har økende sikkerhet.

1. Den enkleste metoden for å padde en melding er å fylle opp med 0ere, denne metoden er litt tvilsom da en ikke vet hvor mange 0ere som er lagt til da meldingen i seg selv kan inneholde 0ere på slutten.

2. Padding av informasjonen med først en 1er og deretter med 0ere. Hvis den siste blokken er komplett legger en til en ekstra blokk med en 1er og resten 0ere.

3. Informasjonen blir paddet med *z* 0ere bortsett fra de siste *r* bitene, disse inneholder en *r* bits representasjon av *z*. Hvis det ikke gjenstår *r* bits *i* den siste blokken, må en legge til en eller flere blokker.

4. Informasjonen blir paddet med *z* 0ere bortsett fra de siste *r* bitene, disse inneholder lengden av informasjonen i bits. Hvis det ikke gjenstår *r* bits *i* den siste blokken, må en legge til en eller flere blokker. Hvis denne regelen for padding benyttes kan ikke en melding fås fra en annen ved å slette de første blokkene.

2.4.5 Splitting

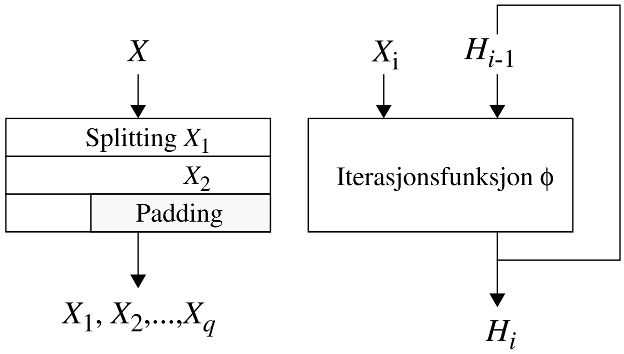
Meldingen *X* deles opp i *m*-bits blokker, X_1, X_2,...,X_q.

2.4.6 Iterasjon

$H_i = \phi(X_i, H_{i\text{-}1})$, hvor ϕ er en iterasjonsfunksjon og H_1 er en start verdi.

2.4.7 Trunkering

Hash verdien H er en gitt del av H_q.



Figur 2.1 *Padding, splitting og iterasjon*

2.5 Krav til iterasjonsfunksjonen

De fleste hashfunksjoner består av en eller flere iterasjonsfunksjoner ϕ som i seg selv bør være bygget opp for å tilfredstille endel krav. De viktigste av disse kravene er som følger:

2.5.1 Avhengighet

Dette sier litt om hvordan utbitene forandres avhengig av hvor mange innbit som blir forandret.

2.5.1.1 Komplett

Det vil si at alle utverdi bits er avhengig av alle innverdi bits.

2.5.1.2 Skred effekt

Hvis en forandring i en innbit fører til en forandring i gjennomsnittlig havparten av utbitene.

2.5.1.3 Strengt skred kriteriet

En funksjon tilfredsstiller dette kriteriet dersom hver ut bit forandres med en sannsynlighet på 1/2 dersom en tilfeldig inn bit forandres.

2.5.2 Lineære faktorer

En funksjon sies å ha en lineær faktor hvis modulo 2 summen av et sett av inn og ut bits ikke forandrer seg når en spesifikk inn bit forandres. Slike lineære faktorer indikerer et sikkerhetsproblem i en kryptografisk funksjon. For eksempel er det publisert et angrep på DES som er basert på lineære faktorer [ChE85]

2.5.3 Sykliske funksjoner

For enhver funksjon f som forandrer n bits til n andre bits og ethvert n-bit tall x vil sekvensen $x, f(x), f^2(x) = f(f(x))$... etterhvert bli periodisk. Denne perioden bør selvfølgelig være lengst mulig.

2.6 Blokksiffer baserte hashfunksjoner

De fleste hashfunksjoner som er i bruk i dag er basert på blokksiffer, det er flere grunner for dette. I begynnelsen var det greit å kunne basere seg på en teknikk som var velkjent og utprøvd, det var derfor mange hashfunksjoner som baserte seg på DES eller varianter av denne. Det største problemet men å benytte DES er at en hashfunksjon basert direkte på DES vil ha en størrelse på hashverdien på 64 bits. En hashverdi på bare 64 bit vil ikke være motstendsdyktig mot et fødselsdagsangrep, det er derfor etterhvert kommet varianter som bruker en dobbel variant av DES for å lage en hashfunksjon, dog vil alle de kjente svakheten ved DES fremdeles gjelde. Generellt kan hashfunksjoner basert på blokksiffer beskrives på følgende måte.

1. Benytte en initialverdi *IV*
2. Dele inn meldingen i blokker
3. Iterere en algoritme $E(K, M)$, hvor M er en melding og K er en nøkkel (evt. *IV*).
4. Gjenta punkt 3

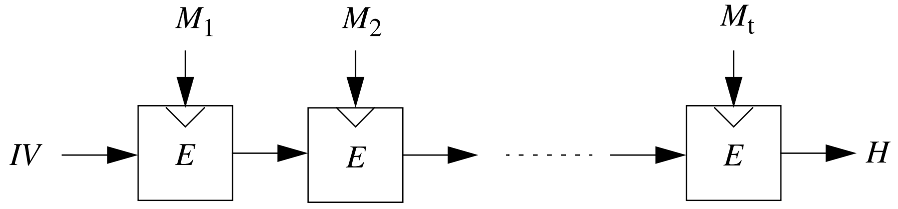
En vil også se at de dedikerte hashfunksjonen bygger på de samme grunnprinsippene. Jeg vil nå gi en oversikt over de viktigste variantene av hashfunksjoner basert på blokksiffer, de bør legges til at dette i prinsippet er enkle oversikter og ikke beskrivelser av konkrete implementasjoner. For en omtale av konkrete implementasjoner av hashfunksjoner henvises til kapittelet om dedikerte hashfunksjoner. De fleste av figurene er hentet fra [PiS93]

2.6.1 Rabins skjema

Dette skjemaet deler først inn meldingen i blokker, og benytter en krypteringsalgoritme E som for eksempel kan være DES [Rab78].

$$H_0 = IV$$
$$H_i = E(M_i, H_{i-1}), \; i = 1, 2, \ldots, t$$
$$H(M) = H_t$$

Som tidligere nevnt vil ikke dette skjemaet være motstandsdyktig mot et eventuelt fødselsdagsangrep hvis en benytter DES.



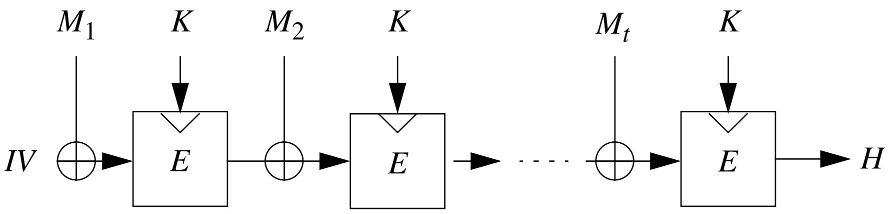
Figur 2.2 *Rabins skjema*

2.6.2 Blokksiffer lenke (CBC) skjema

I dette skjemaet er hashverdien resulatet av den siste blokken som resulterer fra krypteringsalgoritmen i blokksiffer lenke modus til meldingen, mens nøkkelen og den initielle verdien er offentlige. Skjemaet kan beskrives som følger:

$$
\begin{aligned}
H_0 &= IV \\
H_i &= E(K, M_i \oplus H_{i-1}),\ i = 1, 2, \ldots, t \\
H(M) &= H_t
\end{aligned}
$$

En variasjon av dette skjemaet benytter siffer tilbakekobling (CFB).



Figur 2.3 *Blokksiffer lenke skjema*

2.6.3 Blokksiffer lenke (CBC) med sjekksum skjema

En variasjon av det forrige skjemaet er å legge til ekstra informasjon i form av en eksklusiv OR av klarteksten, hvor den initielle vektoren er antatt være null. Den ekstra informasjonen gir en sjekksum som legges til klarteksten [Akl83]. En bruker så en blokk krypterings algoritme i blokksiffer lenke (CBC), siffer tilbakekobling (CFB) eller utgang tilbakekobling (OFB) modus på hele bitstrengen. I dette skjemaet kan nøkkelen være enten privat eller offentlig. Hvis nøkkelen er hemmelig bestemmes sikkerheten på modusen til krypteringsalgoritmen. Det er dog funnet angrep på alle tre modusene som baserer seg på manipulering av blokker. Tabellen viser hvilke manipuleringsmetoder som er mulig uten at det kan detekteres.

Modus	CBC	CFB	OFB
Innsetting	✓	✓	
Permutering	✓		✓
Substitusjon			✓

Figur 2.4 *Manipulering på blokker i CBC, CFB og OFB modus*

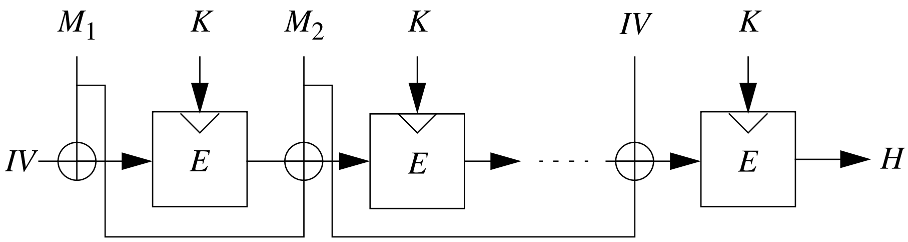
Hvis nøkkelen ikke er hemmelig er skjemaet mottagelig for såkalt *møtes i midten* angrep, jfr Kapittel 2.8.2.1. For å forbedre skjemaet ovenfor ble det framlagt et versjon [MeM82] hvor en legger klartekstblokken til sjekksummen i Galois felt med 2^m elementer for en eller annen m.

2.6.4 Kombinert Klar/kryptotekst lenke skjema

Hvis vi legger til et blokk siffer algoritme for både kryptering av meldingen og genereringen av hashverdien, må forskjellige nøkler benyttes for hver operasjon, hvis ikke vil skjemaet være mottagelig for flere typer manipulasjon. Det er dog foreslått skjema [MeM82] som behøver en hemmelig nøkkel for både krypering og autentisering. Skjemaet er beskrevet som følger:

$$
\begin{aligned}
M &= M_t \ldots M_1 \\
M_{t+1} &= IV \\
H_i &= E(K, M_i \oplus M_{i-1} \oplus H_{i-1}) \quad i = 1, 2, \ldots, t \\
H(M) &= H_{t+1}
\end{aligned}
$$

I det ovenstående skjemaet er M_0 og H_0 satt til å være null. Mens $H(M)$ er hashverdien av meldingen bestemmer H_i siffertekst blokken. Det er verd å merke seg at denne algoritmen er mottagelig for fødselsdagsangrep.



Figur 2.5 *Kombinert Klar/kryptotekst lenke skjema*

2.6.5 Nøkkel lenke skjema

Dette skjemaet ble foreslått av Davies i 1983 [Dav93] og Denning i 1984 [Den84] og er en forbedring av Rabins skjema. Skjemaet kan beskrives som følger:

$$
\begin{aligned}
H_0 &= IV \\
H_i &= E(M_i \oplus H_{i-1}, H_{i-1}), \quad i = 1, 2, \ldots, t \\
H(M) &= H_t
\end{aligned}
$$

Selv om skjemaet er en forbedring av Rabins skjema er det fortsatt mottagelig for *møtes i midten* angrep, j.f.r. Kapittel 2.8.2.1. Flere modifikasjoner er blitt foreslått for å forbedre skjemaet ytterligere. Den første som ble foreslått av Davies og Price i 1980 [DaP80] er å gjenta skjemaet to ganger på meldingen, og den andre modifikasjonen er å kjøre algoritmen med to forskjellige initialverdier (*IV*). Coppersmith [Cop85] har dog vist at møtes i midten angrep kan fremdeles knekke disse forbedrete versjonene av skjemaene. Den tredje forbedringen er å først kryptere meldingen i CBC eller CFB modus før en benytter hash skjemaet. Det fjerde forsøket på forbedring er å legge til en sjekksum av alle meldingsblokken før en benytter hash skjemaet.
Hvis DES benyttes som blokksiffer vil hver av de ovenstående skjemaer være mottagelige for angrep som baserer seg på nøkler med svakheter. Quisquater og Delescaille [QuD89] arbeidet på en kollisjons søke algoritme som resulterte i et angrep på den fjerde modifikasjonen.

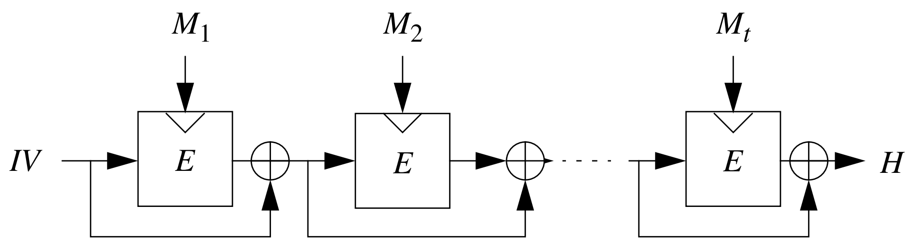
2.6.6 Winternitz nøkkel lenke skjema

Som det er tidligere nevnt er nøkkel lenke skjemaet og de modifiserte versjonene kan angripes med et møtes i midten angrep. Winternitz [Win83] har foreslått et skjema for konstruksjon av en enveis funksjon fra ethvert blokksiffer. I alle gode blokksiffer, gitt en inn og utverdi, skal det være vanskelig å finne ut den brukte nøkkelen, mens å regne ut innverdien hvis en har nøkkelen og utverdien skal være enkelt. I Winternitz konstruksjon kan vi lage en enveisfunksjon fra ethvert godt blokksiffer slik at gitt utverdien og nøkkelen er det vanskelig å gjette verdien på innverdien. Konstruksjonen er definert som:

$$E^*(K, M) = E(K, M) \oplus M$$

Basert på denne konstruksjonen fremla Donald Davies en hashalgoritme som er som følger:

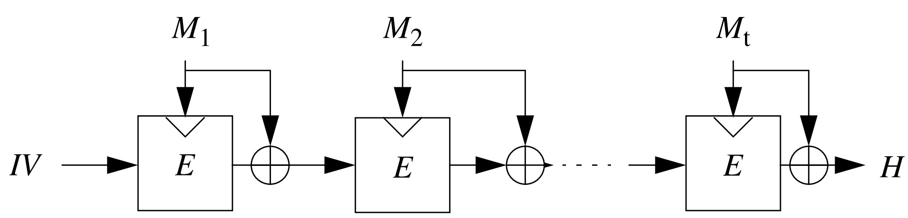
$$\begin{aligned} H_0 &= IV \\ H_i &= E(M_i, H_{i-1}) \oplus H_{i-1} \quad i = 1, 2, \ldots, t \\ H(M) &= H_t \end{aligned}$$



Figur 2.6 *Davies skjema*

Et lignende skjema ble fremmet av Matyas, Meyer og Oseas og er beskrevet som følger:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(H_{i-1}, M_i) \oplus M_i \quad i = 1, 2, \ldots, t \\ H(M) &= H_t \end{aligned}$$



Figur 2.7 *Meyer, Matyas og Oseas skjema*

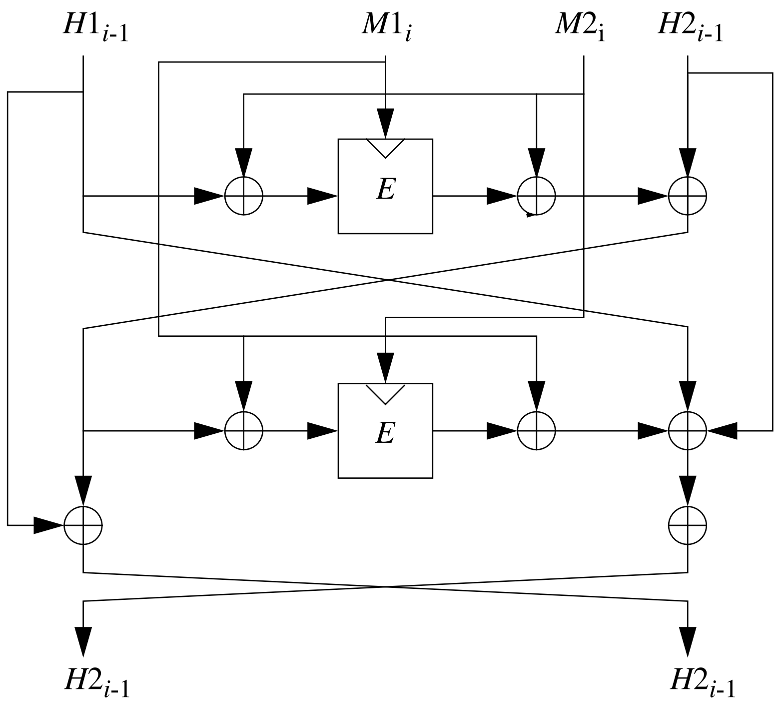
Begge disse skjemaene er ment å baseres på DES, og er derfor mottagelige for angrep basert på svake nøkler og nøkkel kollisjonsangrep, mens møtes i midten angrep er vanskelig på grunn av den brukte funksjonen er enveis.

2.6.7 Quisquater og Girault 2n bits skjema

Det er mulig å angripe alle hashskjemaene som benytter 64 bits hashverdier med fødselsdagsangrepet, siden en bare trenger 2^{32} meldinger og deres tilsvarende hashverdier for å finne kollisjoner. De fleste av dagens krypteringsskjema, slik som DES, FEAL og LOKI er 64 bits skjema, det har blitt foreslått mange skjema som er basert på 64 bits blokksiffer, men som lager en 128 bits hashverdi. En av de enkleste løsninger er å benytte 64 bits skjemaet for to forskjellige verdier av en parameter, slik som nøklenes initialverdier.
Et slik forsøk ble framlagt av Quisquater og Giraul når de foreslo en 128 bits hashverdi som baserte seg på DES. Beskrivelsen av algoritmen er gitt:

$$
\begin{aligned}
H1_0 &= \mathrm{IV}_1 \\
H2_0 &= \mathrm{IV}_2 \\
T1_i &= E(M1_i,\ H1_{i-1} \oplus M2_i) \oplus M2_i \\
T2_i &= E(M2_i,\ H2_{i-1} \oplus M1_i \oplus T1_i) \oplus M1_i \\
H1_i &= H1_{i-1} \oplus H2_{i-1} \oplus T2_i \\
H2_i &= H1_{i-1} \oplus H2_{i-1} \oplus T1_i \\
H(M) &= H1_t \parallel H2_t
\end{aligned}
$$

Med DES som det underliggende blokksifferet vil det for noen svake nøkler være mulig å finne kollisjoner for meldinger i dette skjemaet. Miyaguchi, Hota og Iwata [MHI90] har vist hvordan en kan finne kollisjoner ved hjelp av komplementering og svake nøkler i DES. Det er også rapportert av Coppersmith [PGV92] har knekket dette skjemaet for ethvert blokksiffer på grunn av lineæriteter.

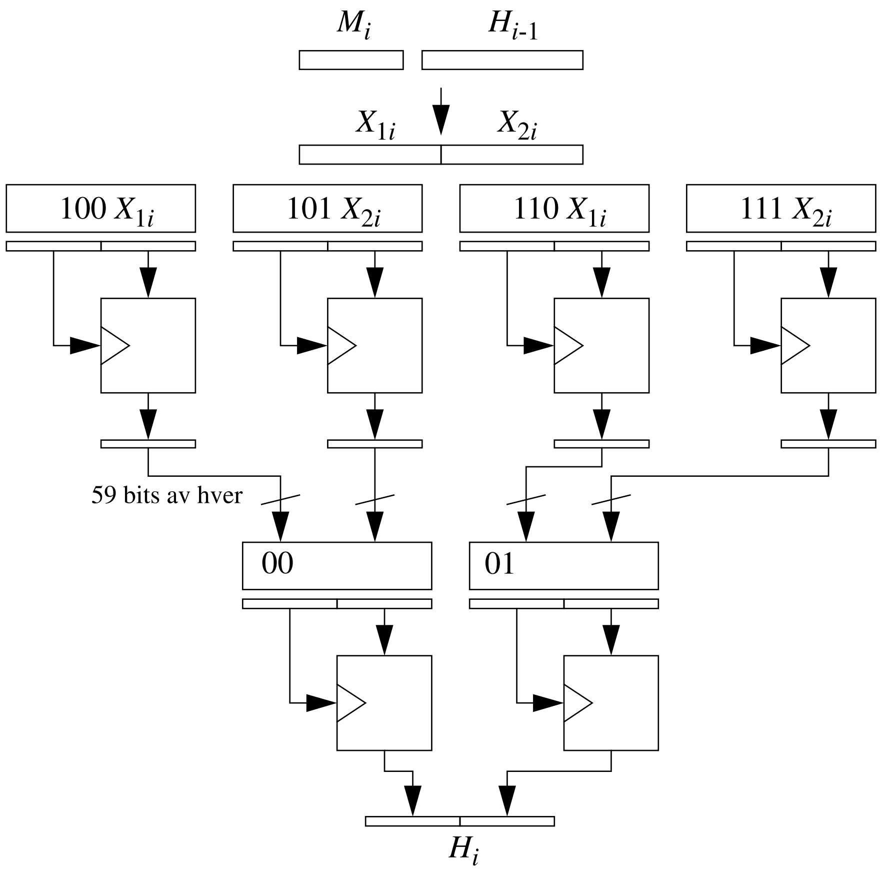


Figur 2.8 *Quisquater & Giraults skjema*

2.6.8 Merkles skjema

Basert på Winternitz konstruksjon fremla Merkle flere skjema [Mer79] og [Mer89]. Disse skjemaene, som er basert på DES, resulterer i hashverdier på 128 bits. Konstruksjonen av disse skjemaene følger en generell metode for å konstruere hash algoritmer. Merkle kalte denne metoden for meta metoden, som er den samme som serie metode prinsippet, beskrevet av Damgård [Dam89]. Merkles skjema tar fordel av Winternitz konstruksjon. I noen forslag, som er raske og komplekse, blir meldingen først delt inn i blokker på 106 bits. Konkateneringen av hver av 106 bits blokkene M_i av data med 128 bits blokkene $H_{i\text{-}1}$ og resultatet av den forrige hashingen, gir en 234 bits blokk. Vi benevner denne konkateneringen som $X_i = M_i \| H_{i\text{-}1}$. Hver blokk X_i blir igjen delt opp i to deler X_{i1} og X_{i2}, det vil si 117 bits lengde. Beskrivelsen av metoden er som følger:

$$
\begin{aligned}
H_0 &= IV \\
X_i &= H_{i-1} \| M_i \\
& H_i = E^*(00 \| \texttt{første 59 bits av } \{E^*(100 \| X_{1i})\} \| \\
& \texttt{første 59 bits av } \{E^*(101 \| X_{2i})\} \| \\
& E^*(01 \| \texttt{første 59 bits av } \{E^*(110 \| X_{1i})\} \| \\
& \texttt{første 59 bits av } \{E^*(111 \| X_{2i})\} \\
H(M) &= H_i
\end{aligned}
$$

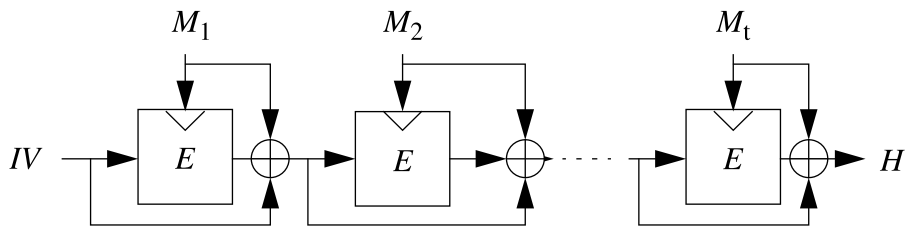


Figur 2.9 *Merkles skjema*

I dette skjemaet er E* definert som Winternitz konstruksjon og strengene 00, 01, 100, 101, 110 og 111 har blitt inkludert for å motvirke angrep basert på svake nøkler.

2.6.9 N-hash algoritmen

N-hash er en hashalgoritme som produserer en 128 bits hasverdi [MIO89]. Algoritmen, som ble foreslått av designere av FEAL. Både størrelsen på initialverdien og meldingsblokken er på 128 bits. Fundamentet i N-Hash er en krypteringsfunksjon $R(H_{i-1}, X_i)$. Denne funksjonen er reversibel med hensyn på X_i, det vil si at en kan finne X_i hvis H_{i-1} er kjent, utifra $R(H_{i-1}, X_i)$. Krypteringen består av en addisjon til X_i av en konstant og H_{i-1} hvor den øverst og nedserste delen er bytte om, deretter 8 runder med kalkuleringer. Innverdien til hver runde er lik $H_{i-1} \oplus V_i$, hvor V_i avhenger av hvilken runde som kjøres og hvor $\oplus$ er addisjon modulo 2.



Figur 2.10 *Oversikt over N-Hash*

Algoritmen kan nå beskrives som følger:

$$f = R(H_{i-1}, X_i) \oplus H_{i-1} \oplus V_i$$

En runde består er ekvivalent med to runder av et Feistel siffer [Fei73]. Ikke lineæriteten kommer fra addisjonen modulo 256 etterfulgt av en rotering over to posisjoner.

S-boksene i N-hash har flere svakheter, f.eks. en innverdi XOR 0x80 vil bestandig føre til en utverdi XOR av 0x02.
I [MOI90] ble det stilt spørsmål om f er kollisjons sikker.
I 1989 viste B. den Boer [Pre93] hvordan en kunne konstruere tre pseudo meldinger som hasher til en gitt verdi. Han fant konstanter α og β slik at

$$R(H_{i-1} \oplus \alpha, X_i \oplus \beta) = R(H_{i-1}, X_i) \oplus \alpha \oplus \beta$$

som fører til at H_i vil forbli uforandret. En mulig løsning på dette problemet er å velge en α lik $a \| c \| b \| c \| a \| c \| b \| c$ og en β lik $b \| c \| a \| c \| b \| c \| a \| c$ hvor a = 0x8280, b = 0x8080 og c = 0x0000. Etter den initielle adderingenav Hi-1, hvor øvre og nedre del er byttet om, er innverdien XOR av den første runden lik $\alpha \oplus \beta = d \| d \| d \| d$, hvor d = 0x2000. sammen med en innverdi XOR av α i nøkkelen, fører dette til en utverdi XOR av $\alpha \oplus \beta$ etter den første runden. Det er klart at påfølgende runder innverdier og utverdier XOR vil forbli konstante, som betyr at dette angrepet er uavhengig av antall runder.

Eli Biham og Adi Shamir [BiS91] utnyttet andre svakheter i S boksene slik at en fikk en gjentagne karateristikk av tre runder med en sannsynlighet på 2^{-16}. Dette angrepet kan beskrives som følger: La ψ = 0x80608000 og la ϕ = 0x80E08000, en får da følgende mønster av inn og ut verdi XOR:

$$(\psi,\psi,0,0) \Rightarrow (0,0,\phi,\phi) \Rightarrow (\psi,\psi,\phi,\phi) \Rightarrow (\psi,\psi,0,0)$$

Her vil den første og den andre ovegangen ha en sannsynlighet på 1/256 mens den tredje ovegangen vil ha en sannsynlighet på 1. Dette betyr at for N-hash med 3 runder vil en kunne finne en annen melding som hasher til den samme hashverdien med en sannsynlighet lik 2^{-16r} og en kollisjon kan produseres $2^{8+16(r-1)}$, dette betyr at det er raskere en fødselsdagsangrep for varianter av N-hash med opp til 12 runder. F.eks. vil følgende meldinger hashe til samme verdi:

```
CAECE595 127ABF3C 1ADE09C8 1F9AD8C2
```

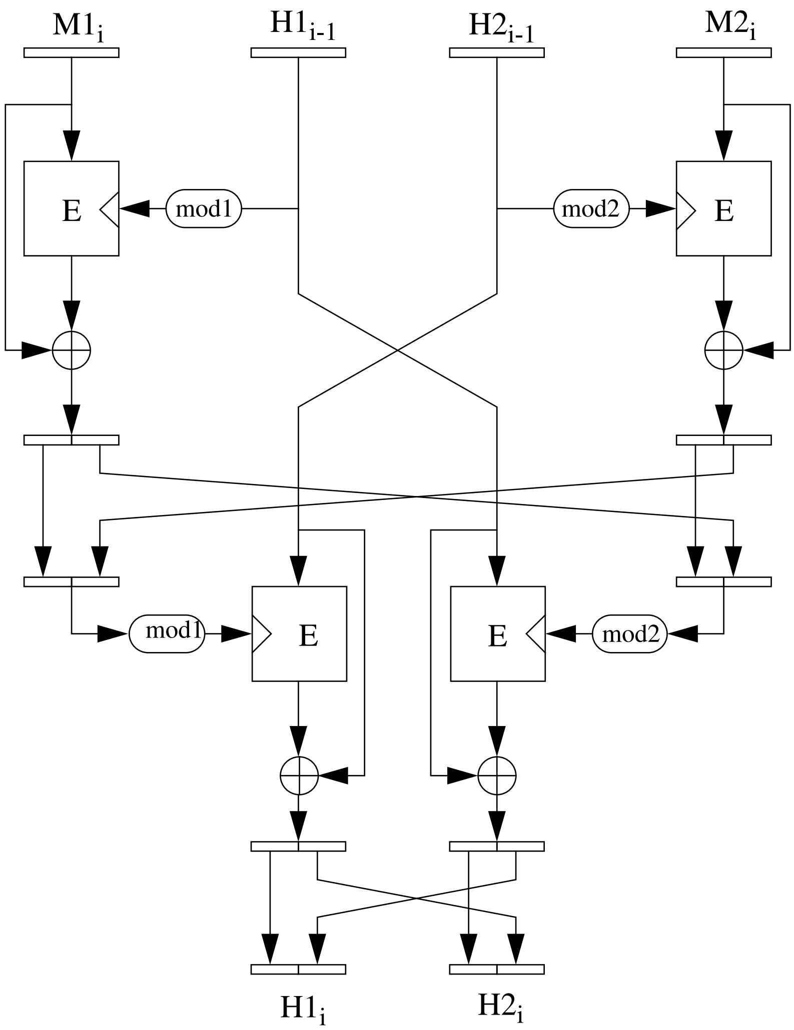
og

```
4A8C6595 921A3F3C 1ADE09C8 1F9AD8C2

Felles hashverdi: 12B931A6 399886B7 640B9289 36C2EF1D
```

Merk at dette angrepet ikke kan benyttes på den vanlige varianten av N-hash da den har 8 runder, det vil derfor være et mål å finne gode karakteristikker for antall runder som ikke kan divideres med 3. Det er etterhvert også publisert flere varianter av dette angrepet også i det japanske forslaget til internasjonal standard som er en variant av N-hash. Selv om det pr. i dag ikke er publisert kollisjoner m.h.p. den originale N-hash, har det avdekket så mange svakheter at N-hash ikke kan regnes som sikker.

2.6.10 MDC2 og MDC4

For å detektere modifikasjoner i sikre overføringer framsatte IBM sine MDC hash skjema. Det er to versjoner av dette hash skjemaet, MDC2 og MDC4. Den første benytter to DES krypteringer pr. (byte innverdi blokk, mens den andre benytter fire DES krypteringer. MDC skjemaene definerer en enveis funksjon som er basert på Winternitz [Win83] teorier. To forskjellige versjoner gir brukeren en mulighet for å finne et kompromiss mellom sikkerhet og hastighet. MDC4 blir beskrevet her og også MDC2 bygger på de samme prinsippene. Først krypteres meldingen ved å benytte den forrige hashverdien som nøkkel, deretter krypteres hash blokken hvor den krypterte meldingen fungerer som nøkkel.



Figur 2.11 *IBMs MDC4 skjema*

I figuren ser en at $M1_i$ og $M2_i$ er 64 bits av lengde og inneholder den venstre og den høyre halvdel av den 128 bits meldingsblokken respektivt. Mod1 funksjonen setter bits 1 og 2 til 1 og 0, mens mod2 funksjonen setter bits 1 og 2 til 0 og 1 respektivt. Disse funksjonen fjerner den symmetriske strukturen i hash skjemaet og begrenser muligheten for angrep basert på svakheten i DES, at $\overline{E}(K,M) = E(\overline{K},\overline{M})$. Det motvirker også angrep basert på svake nøkler i DES. På grunn av at de fire DES krypteringene blir utført på hver 128 bits meldings blokk er skjemaet mindre effektivt enn de tidligere nevnte. Bart Preneel [Pre93] har benyttet differensiellt kryptoanalyse på MDC-2 og MDC-4 fant ut at det finnes angrep som var bedre en generelle fødselsdags-angrep på versjoner med færre runder.

2.7 Ikke Blokksiffer baserte hashfunksjoner

Andre hash skjema som er blitt framsatt baserer seg på algoritmer som ikke er blokk siffer algoritmer, men funksjoner som er kompliserte og vanskelig å invertere. Dette inkluderer enveis funksjoner som fra tallteori, det vil si RSA, kvadrering, ryggsekk baserte problemer, kompliserte software algoritmer eller celluar automata skjema. Jeg vil i dette kapittelet beskrive noen av disse ikke blokksiffer hash skjemaene.

2.7.1 RSA blokksiffer lenke (CBC) skjema

For det første skjemaet bruker en RSA algoritmen som den underliggende enveis funksjonen og utfører blokksiffer lenking (CBC) med den. Beskrivelsen av skjemaet er som følger:

$$H_0 = IV$$
$$H_i = (H_{i-1} \oplus H_i)^e \bmod n,\ i = 1, 2, ..., t$$
$$H(M) = H_t$$

Hvor *n* og *e* er offentlige. Et korrigerende blokk angrep kan angripe skjemaet ved å legge til en blokk for å få en ønsket hashverdi. En modifisert versjon legger til overflødig informasjon for å motvirke korrigerende blokk angrep. For å oppnå en sikker RSA bør *n* være på minst 512 bits lengde, det vil si at denne algoritmen er lite effektiv i praksis.

2.7.2 Kvadrerings skjema

I dette kapittelet viser jeg fem skjema som alle er basert på kvadrering.

2.7.2.1 Davies og Price kvadrerings skjema

For å øke hastigheten på RSA blokksiffer lenke skjemaet framla Davies og Price [DaP84] en applikasjon som brukte kvadrering isteden for å bruke en offentlig eksponent e, det vil si

$$H_0 = \mathrm{IV}$$
$$H_i = (H_{i-1} \oplus H_i)^2 \bmod n,\ i = 1, 2, ..., t$$
$$H(M) = H_t$$

For å unngå korrigerende blokk angrep foreslo de å sette 64 bits av hver meldingsblokk til 0. Girault [Gir87] har vist at det er mulig å finne meldinger som kolliderer for dette skjemaet. For å forbedre dette skjemaet bør den redundante informasjonen som legges til være omkring 128 bits.

2.7.2.2 Giraults skjema

De følgende skjema er basert på de foregående skjema [Gir87]. Jeg viser bare løkkevariasjonen da resten av skjemaene er identiske.

$$H_i = H_{i-1} \oplus (H_i^2 \bmod n),\ i = 1, 2, ..., t$$

Dette skjemaet er sårbart ovenfor angrep basert på permutasjoner av blokker, innsetting av et like antall blokker, innsetting av null blokker eller manipulering på små blokker hvor $H_i^2 < n$.

$$H_i = M_i \oplus (H_{i-1}^2 \bmod n),\ i = 1, 2, ..., t$$

Denne metoden er sårbar ovenfor angrep som baserer seg på blokk korrigering

$$H_i = (H_{i-1} \oplus (M_i^2 \bmod n))^2, \quad i = 1, 2, \ldots, t$$

Det er ikke publisert noen angrep på dette skjemaet, det er dog forventet å være lite effektivt.

2.7.2.3 CCITT X.509

Appendiks D av CCITTs X.509 standard for sikker meldingshåndtering fremsetter et forslag til hashing basert på kvadrering. Det framsatte skjemaet introduserer 256 bits redundans som fordeles over hver 256 bits blokkeved å blande fire bits av meldingen med 1111, slik at det totale antallet bits i hver blokk blir 512. Så blir en eksponeringsalgoritme kjørt i CBC modus med en eksponent lik 2, på den modifiserte meldingen. Dette skjemaet gjør at det mest signifikante biten i hver blokk blir 1. Coppersmith [Cop89] har laget et angrep som finner kollisjoner når en benytter hashskjemaet sammen med RSA.

2.7.2.4 Junemans skjema

Juneman har framsatt flere skjema som er basert på kvadrering [JMM85, Jun86, Jun87]. Hans første forslag er tilnærmet identisk med Davies og Price, med den forskjellen at XOR funksjonen er erstattet med addisjon og at n er primtallet 2^{31}-1. Dette skjemaet resulterer i en 32 bits hashverdi, dette fører til at skjemaet er sårbart ovenfor fødselsdags angrep og i tillegg har det svakheter som kan benyttes i et møtes i midten angrep. For å få en 128 bits hashverdi foreslo Juneman å kjøre igjennom skjemaet fire ganger. Dette skjemaet er også sårbart ovenfor et generellt fødselsdags angrep da alle kjøringene benytter felles modulus. Det tredje forslaget inkluderer fire forskjellige moduser for n, lik de fire største primtallene mindre enn 2^{31}-1. Skjemaet kan beskrives som følger:
Del inn meldingen i blokker på 128 bits lengde. Splitt så hver meldingsblokk M_i til fire blokker $M1_i$ til $M4_i$. En femte blokk konstrueres ved å velge ut noen bits fra $M1_i$ til $M4_i$ på følgende måte:

$$M5_i = (00 \parallel M1_{i31-26} \parallel M1_{i31-24} \parallel M1_{i31-24} \parallel M1_{i31-24})$$

Den andre bit i $M1_i$ til $M4_i$ blir satt til 0.

For j = 1 til 4, er de fire funksjonene $H_{i,j}$ beskrevet som følger:

$$\begin{aligned} H_{i,j} = {} & [(H_{j\bmod 4, i-1} \oplus M1_i) - (H_{(j+1)\bmod 4, i-1} \oplus M2_i) \\ & + (H_{(j+2)\bmod 4, i-1} \oplus M3_i) - (H_{(j+3)\bmod 4, i-1} \oplus M4_i) \\ & + (-1)^{j+1} M5_i] \bmod n_j \end{aligned}$$

Det er rapportert at Coppersmith [PGV92], i løpet av 2^{32} operasjoner, har knekket dette skjemaet med et korrigerende blokk angrep som kombinerer algebraiske manipulasjoner med fødselsdagsangrep.

2.7.2.5 Damgårds kvadrerings skjema

Damgård beskrev, i en artikkel om hvordan å lage kollisjonsfri hashfunksjoner, et skjema basert på kvadrering for å hashe en blokk på *n* bits til en blokk på *m* bits. Beskrivelsen følger:

$$
\begin{aligned}
H_0 &= IV \\
H_i &= extract\ m\ \texttt{bits av}(00111111 \parallel H_{i-1} \parallel M_i)^2 \bmod n \\
H(M) &= H_t
\end{aligned}
$$

I skjemaet over er *extract* en funksjon som plukker ut *m* bits fra resultatet av kvadreringsfunksjonen. For å få en sikkert skjema bør *m* være stor nok for å unngå fødselsdagsangrep. Extract funksjonen bør også velge bits som fører til at det er vanskelig å finne kollisjoner. En mulighet er å velge m uniformt distribuerte bits. Av praktiske årsaker er det bedre å slå dem sammen til bytes. En annen mulighet er å velge ut hver fjerde byte, det er dog meldt at dette skjemaet kan knekkes.

2.7.3 Claw-Free baserte skjema

Damgård viste at det er mulig å konstruere kollisjonsfrie hashfunksjoner basert på eksistensen av claw-free permutasjoner [Dam88]. En claw-free permutasjon er et sett av permutasjoner S = $\{f_0, f_1,..., f_{r-1}\}$, slik at, for hver x i domenet til f_i, er det enkelt å regne ut $f_i(x)$ for alle i = 0,..., r-1, men det er praktisk umulig å lage en claw, det vil si finne en *y* slik at, for en eller annen $i \neq j$, $f_{\mathrm{i}}(\mathrm{x}) = f_{\mathrm{j}}(y)$. Damgård argumenterte for at det følgende skjema ville gi en beviselig kollisjons fri hashfunksjon.

$$
\begin{aligned}
H_0 &= IV \\
H_i &= f_{\mathrm{M}i}(H_{i-1}),\ i = 1,\ 2, ...,\ t \\
H(M) &= H_t
\end{aligned}
$$

Goldwasser, Micail og Rives [GMR88] har også vist at en lignende struktur med r = 0, også vil føre til et sikkert signatur skjema. Damgård har også framsatt ytterligere tre skjema, basert på modulær kvadrering, for claw-free konstruksjoner.

2.7.4 Ryggsekk baserte skjema

De følgende skjema er basert på ryggsekk problemet.

2.7.4.1 Damgårds ryggsekk skjema

Nok et forslag til skjema fra Damgård er basert på ryggsekkproblemet og kan bli beskrevet som følger. Velg tilfeldige tall a_1,..., a_s i intervallet 1,..., N hvor *s* indikerer den maksimale lengden en melding kan ha i antall blokker. Damgård velger s = 256 og N = 2^{120}-1. Da kan den binære meldingen M_1, M_2,..., M_s bli hashet som følger:

$$H(M) = \sum_{i=1}^{s} M_i a_i$$

Dette skjemaet vil gi et hashverdi på 128 bits.

Camion og Patarin [CaP91] har vist at det ovenstående skjemaet ikke er sikkert. De demonstrerte at en deterministisk algoritme med omkring 2^{32} utregninger vil knekke skjemaet, antallet utregninger er praktisk gjennomførbart.

2.7.4.2 Impagliazzo og Naors skjema

Impagliazzo og Naor [ImN89] framla et kryptografisk subsett summerings funksjon som kan anvendes for hash funksjoner. Skjemaet kan beskrives på følgende måte: Velg tilfeldige tall $a_1,..., a_n$ i intervallet 0,..., N hvor n indikerer lengden på meldingen i bits, og $N = 2^l$-1 hvor l < n. Den binære meldingen $M = M_1, M_2,..., M_n$ tilsvarer delmengden S ⊂ {1,..., n} å kan hashes ved:

$$H(M) = \sum_{i \in S} a_i mod\ 2^l$$

Impagliazzo og Naor har ikke nevnt noen konkrete verdier for det ovenstående skjemaet, men har vist at det er teoretisk sikkert.

2.7.5 Celluar Automata skjema

Celluar automata består av en rekke posisjoner eller celler, hvor verdiene er 0 eller 1. Disse verdiene er oppdatert i en sekvens i henhold til en definert regel. Ved å betegne verdiene til en posisjon i ved a_i, gir en enkel regel dennes nye verdisom følger:

$$a_i' = \phi(a_{i-1},\ a_i,\ a_{i+1})$$

hvor ϕ er en boolsk funksjon som spesifiserer regelen [Wol86]. Til tross for at prinsippet er enkelt gir mange celluar automat systemer stor kompleksitet.

2.7.5.1 Damgårds Pseudorandom bitgenerator skjema

Wolframs pseudorandom bit generator består av en endimensjonal celluar automata av n bits. La $x = x_0, x_1,..., x_{n-1}$ være innverdien til bitgeneratoren. Funksjonene $g(x_i)$ definerer verdien til den i-te cellen i den neste omgangen. Funksjonen g er definert som følger:

$$g(x_i) \quad = \quad x_{i-1} \oplus (x_i \vee x_{i+1})$$

Betegn verdien av den i-te cellen i den j-te omgangen ved $g_j(x_i)$ Bite generatoren $b(x)$ starter fra en tilfeldig x og gir som utverdi sekvensen $g_j(x_0)$.
For $d > c$, la $b_{c\text{-}d}(x)$ betegne strengen

$$g_c(x_0),\ g_{c+1}(x_0), ...,\ g_d(x_0)$$

hash funksjonen er definert som følger:

$$\begin{aligned} H_0 &= IV \\ H_i &= b_{c-d}(M_i \| H_{i-1} \| Z) \\ H(M) &= H_t \end{aligned}$$

hvor Z er en tilfeldig verdi som blir lagt til for å gjøre det vanskeligere å finne meldinger som kolliderer. Som et konkret forslag foreslo Damgård $n = 512$, $r = 256$, $c = 257$ og $d = 384$. Den foreslåtte hashfunksjonen vil hashe meldinger av vilkårlig størrelse til 128 bits hashverdier. Daemen, Govaerts og Vandewalle [DGV91] har vist hvordan en kan kryptoanalysere dette skjemaet og finne meldinger som kolliderer.

2.7.5.2 Cellhash skjema

For å lage et skjema som kan implementeres i hardware, framla Daemen, Govaerts og Vandewalle [DGV91] en enveis hashfunksjon som de kalte Cellhash. Cellhash gir en hashverdi på 257 bits og er derfor motstandsdyktig mot fødselsdagsangrep. Det var også et mål med hashfunksjonen at den skulle ha god spredning, tilfredstille skredkrav, se Kapittel 2.5 og være effektiv. Cellhash fungerer som følger: Først blir 0ere lagt til meldingen slik at lengden blir på minst 248 bits og kongruent til 24 (mod 32). Antallet bits som legges til blir representert i en byte som blir lagt til. *IV* er en null streng av lengde 257. Utregningen av H_j fra H_{j-1} er gjort ved hjelp av nøkkelen M_j, den *j*te meldingsblokken og kan ansees som en transformasjon med 5 omganger. Utregningene i hver omgang er gjort samtidig på alle bits i H_{j-1}. La h_0, h_1,...,h_{256} representerer bitene til H_{i-j} og m_0, m_1,...,m_{255} gir bitene til M_j.

Steg 1:	$h_i = h_i \oplus (h_{i+1} \vee h_{i+2})$	$0 \leq i < 257$
Steg 2:	$h_0 = h_0$	
Steg 3:	$h_i = h_{i-3} \oplus h_i \oplus h_{i+3})$	$0 \leq i < 257$
Steg 4:	$hi = h_i \oplus m_{i-1}$	$0 \leq i < 257$
Steg 5:	$h_i = h_{10i}$	

Det er ikke publisert noen angrep på denne skjemaet.

2.7.6 Matrise baserte skjema

Matrise algebra er et annet område som har blitt brukt for å lage hash algoritmer. En slik algoritme er kalt Random Matrix Hashing Algorithm [BaH90]. Algoritmen vurderer innverdien til å være en $1 \times m$ rad vektor av bits og utverdien til å være en $n \times 1$ kolonne vektor. Algoritmen består i å velge en fast m x n random binær matrise og multipliserer denne med input vektoren. Matrisen blir holdt hemmelig, som en nøkkel. Matrisen kan også velges på en slik måte at den kan inverteres dersom ønskelig. En kan forandre størrelsen på algoritmen slik at en får den ønskede lengden både på inn- og utverdien.

Et annet skjema, utviklet av Harari [Har84] bruker som nøkkel en tilfeldig $t \times t$ matrise, hvor *t* er antallet av *m*-bits blokker av klarteksten M. Hashverdien blir regnet ut som følger:

$$H(M) = M^T A M = \sum_{i<j} a_{ij} x_i x_j$$

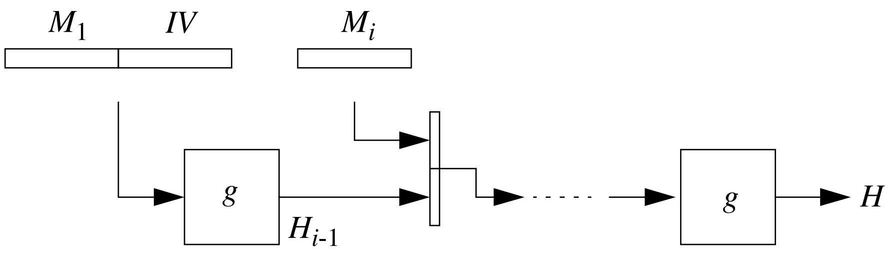
Skjemaet er usikkert hvis inntrenger har muligheten for å velge meldingene som skal hashes, et valgt meldings angrep.

2.7.7 FFT hashing

FFT-hash ble laget av C. Schnorr i forbindelse med Crypto-91 [Sch91].

Algoritmen er basert på en bijektiv transformasjon som bygges oppfra en diskre Fourier transformasjon og polynomiske rekursjoner. Sikkerheten ligger i det faktum at polynomiske tranformasjoner av høy grad over en endelig kropp genererer tilnærmede tilfeldige tall.

$$
\begin{aligned}
H_0 &= IV = \texttt{0x0123456789ABCDEFFEDCBA987654321} \\
H_i &= g(M_i \| H_{i-1}) \\
H(M) &= H_t
\end{aligned}
$$



Figur 2.12 *FFT-hash*

Innverdien til funksjonen består av 16 x 16 bits Ord som vil betegnes $X[0]$, $X[1],...,X[15]$. Den følgende transformasjonen benyttes:

1. $(X[0], X[2],...X[14]) = FT_8(X[0], X[2],...X[14])$.
2. FOR $i = 0,1,...15$ DO
   $X[i]+X[i-1]X[i-2]+X[X[i-3]] + 2^i \bmod p,\ p = 2^{16}+1$
3. $(X[0], X[2],...X[14]) = FT_8(X[0], X[2],...X[14])$.
4. FOR $i = 0,1,...15$ DO
   $X[i]+X[i-1]X[i-2]+X[X[i-3]] + 2^i \bmod p,\ p = 2^{16}+1$

Alle operasjonene er modulo p, bortsett fra eksponentene som er tatt modulo 16. Operasjonene FT_8 er fourier transformasjonene med den primitive rot 2^4 av orden 8:

$$FT_8 : Z_p^8 \to Z_p^8 : (a_0,...,a_7)\ (b_0,...,b_7),\ \text{hvor}\ b_i = \sum_{j=0}^{7} 2^{4ij} a_j$$

til slutt blir ordene redusert modulo p. Fra denne transformasjonen får en kompresjonsfunksjon ved å velge ord $X[8],..., X[15]$ som utverdi. Hash funksjonen er nå definert ved å spesifisere at de første 8 inn ordene blir brukt for variablene og ved å lage en padding prosedyre og en initial verdi. En bør merke seg at bijective transformasjoner kan inverteres enkelt, det vil si at det er enkelt å finne en pseudo innverdi når utverdien er gitt. Det kan bli vist at for tilfeldige innverdier, vil utverdiene ha en uniform fordeling. En svakhet med dette skjemaet er at for å lagre en enkelt variabel modulo p er det nødvendig med to 16 bits ord. Den serielle karakteren på rekursjonen ekskluderer i stor grad effektive hardware implementasjoner. Fouriertransformasjonene FT_8 påvirker kun de variablene med partallsindex(2, 4, 6,...), videre hvis et subsett av 8 innverdier eller utverdier er kjent, kan de gjenværenede innverdiene og utverdiene regnes ut. Disse svakhetene ledet til to uavhengige angrep av J. Daemen, A. Bosselaers, R. Govaerts og J. Vandewalle [DAG91] og av T. Baritaud, H. Gilbert og M. Girault [BGG92]. Begge angrepene behøver bare 223 delvise evalueringer av hashfunksjonen og leder til flere kollisjoner, det vil si mange meldinger som hasher til den samme verdien. Det første angrepet behøver 3 meldingsblokker og varierer det andre ordet av den første meldingsblokken. Det andre angrepet bruker 2 meldingsblokker og varierer det siste ordet av den første meldingsblokken. Disse angrepene førte til at C. Schnorr lanserte en forbedring av FFT-Hash på Eurocrypt 92 [Sch92] denne ble kalt FFT-Hash II, den inneholder kun mindre forskjeller fra den opprinnelige FFT-Hash I:

1. $(X[0], X[2],...X[14]) = FT_8(X[0], X[2],...X[14])$.
2. FOR i = 0,1,...15 DO
   $X[i]=X[i]+X^*[i-1]X^*[i-2]+X[i-3]+2^i \bmod p,\ p = 2^{16}+1$
3. $(X[0], X[2],...X[14]) = FT_8(X[0], X[2],...X[14])$.
4. FOR i = 0,1,...15 DO
   $X[i]=X[i]+X^*[i-1]X^*[i-2]+X[i-3]+2^i \bmod p,\ p = 2^{16}+1$

Her indikerer * at variabelen vil bli erstattet med 1 dersom den er 0.
Det er funnet flere svakheter ved dette skjemaet: Hvis $X[i\text{-}1]$ og $X[i+1]$ er begge lik 0 vil en forandring i $X[i]$ ikke forplante seg til de andre variablene. Tre uker etter annonserte S. Vaudenay at han hadde funnet kollisjoner for FFT-Hash II, Angrepet ble presentert på Crypto 92 [Vau92]. En annen variant av C. Schnorr foreslår å ødelegge reversibiliteten ved å legge til innverdien til transformasjonen til utverdien. Det virker som om hastigheten på dette skjemaet er for dårlig for praktisk bruk.

2.8 Angrep på hashfunksjoner

I dette kapittelet vil jeg ta for meg måter å angrip hashfunksjoner, det er mange forskjellige måter å angripe hashfunksjoner. Jeg har delt disse inn i fem forskjellige grupper:

- Angrep som er uavhengig av algoritmen
- Angrep som er avhengig av lenkingen/chaining
- Angrep basert på blokk sifferet
- Angrep avhengig av signatur skjemaet
- Høynivå angrep

Før en kan diskutere de forskjellige angrepsmetodene vil jeg diskutere hvilken informasjon en eventuell angriper vil ha tilgang til og håper å få tilgang til.

preimage	Her prøver angriperen å finne en melding som passer til en gitt hashverdi.
pseudo preimage	Her prøver angriperen å finne en melding for en gitt hashverdi, hvor $IV' \neq IV$
kollisjon	Her prøver angriperen å finne to meldinger som hasher til samme hashverdi.

Disse begrepene vil blir ofte benyttet i litteraturen.

2.8.1 Angrep som er uavhengig av algoritmen

Denne typen angrep er kun avhengig av størrelsen på hashverdien n og størrelsen på nøkkelrommet k. Det er antatt at hashverdien er uniformt distribuert og at den har en karakterestikk som tilsvarer en random generator.

2.8.1.1 Tilfeldig angrep

Dette angrepet baserer seg på at en eventuell forfalsker bruker en falsk melding, finner hashverdien til denne og håper at den tilsvarer hashverdien til den opprinnelige meldingen. Sannsynligheten for at dette skal lykkes er $1/2^n$, hvor n er antallet bits i hashverdien.

2.8.1.2 Fødselsdags angrep

Ideen bak fødselsdagsangrep er at det i en gruppe på 23 personer, vil sannsynligheten for at to av dem har fødselsdag på samme dag være større enn 50%. Da dette tallet er mindre enn man kanskje skulle vente, kalles dette ofte for fødselsdagsparadokset. Dette resultatet forutsetter at fødselsdagene er jevnt fordelt utover året, hvis en for eksempel antar at de fleste er født på sommeren vil sannsynligheten for å finne kollisjoner bli enda høyere.
For å overføre dette til hashfunksjoner kan en tenke seg at en genererer r_1 variasjoner av en falsk melding og r_2 variasjoner av en ekte melding. Dette er lett selv om r_1 og r_2 er store da en kan trenger å ha $\log_2(r_1)$ respektivt $\log_2(r_2)$ posisjoner hvor en har en eller to alternativer eller synonymer. Hvis $r_1 = r_2 = r = \sqrt{n}$ vil sannsynligheten for å finne et par være 63%.
Å finne det konkrete paret vil ikke kreve r^2 operasjoner, etter å ha sortert, som krever $O(r \log r)$ operasjoner, er det enkelt å sammenligne.

2.8.1.3 Ekstensivt nøkkel søk

Et ekstensivt søk etter nøkler kan kun anvendes på en MAC. Det er en kjent klartekst angrep hvor angriperen vet et antall klartekst/MAC par for en gitt nøkkel. Han vil regne ut en MAC for hver mulig nøkkel for å kunne eliminere de nøklene som er feil. Det kan vises at antallet av klartekst-MAC par som trengs for å regne ut en nøkkel unikt er litt større enn k/n, hvor k er nøkkelrommet og n er lengden på hashverdien.

2.8.2 Angrep som er avhengig av lenkingen/chaining

Denne typen angrep er, som overskriften viser, avhengig av hvordan en utfører lenkingen.

2.8.2.1 Møtes i midten angrep

Dette angrepet er en variant av fødselsdagsangrepet, men istedenfor hashver dier blir mellomliggenede lenkede variabler sammenlignet. Angrepet gjør det mulig for en inntrenger å konstruere en melding med en forhåndsspesifisert hashverdi, hvilket ikke er mulig ved et ordinært fødselsdagsangrep.
En genererer også her r_1 variasjoner av den første delen av en falsk melding og r_2 variasjoner av siste delen av meldingen. Ved å starte på initialverdien og gå bakover fra hashverdien, er sannsynligheten for å finne en kollisjon gitt. Den eneste restriksjonen som er gitt på hvor de møtes er at de ikke kan møtes i den første eller siste verdien i lenkingen.

2.8.2.2 Korrigerende blokk angrep

I dette angrepet blir det lagt til en blokk til den falske meldingen slik at hashverdien blir korrigert og en får en ny hashverdi. Dette angrepet brukes ofte på den siste blokken i meldingen og kalles derfor ofte korrigerende siste blokk angrep, selv om angrepet også kan anvendes på endre blokker i meldingen. Hashfunksjoner som er basert på modulær aritmetikk er spesielt mottagelig for denne typen angrep. Denne typen angrep kan også benyttes for å produsere kollisjoner. En starter med to tilfeldige meldinger X og X' og legger til en eller flere korrigerende blokker Y og Y' slik at $X \| Y$ og $X' \| Y'$ har samme hashverdi. En kan redusere muligheten for at denne type angrep skal lykkes ved å legge til overflødighet til meldingen, men det fører til en mindre effektiv hashalgoritme.

2.8.2.3 Fast utgangspunkt angrep

Tanken bak dette angrepet er å se etter en $H_{i\text{-}1}$ og X_i slik at $f(X_i, H_{i\text{-}1}) = H_{i\text{-}1}$. *H*vis lenke variabelen er lik $H_{i\text{-}1}$, er det mulig å legge til et antall blokker som er lik $H_{i\text{-}1}$ uten å modifisere hashkoden. Å produsere en kollisjon eller et preimage med denne typen angrep er bare mulig hvis lenke variabelen kan gjøres lik $H_{i\text{-}1}$, dette er tilfellet dersom *IV* kan velges lik en spesifikk verdi eller hvis et stort antall faste punkt kan konstrueres, det vil si hvis en kan finne en X_i for hver $H_{i\text{-}1}$. Dette angrepet kan enkelt unngås, det er bare å legge til en teller for hver datablokk inn i skjemaet.

2.8.2.4 Nøkkel kollisjoner

Denne typen angrep kan bare anvendes på blokk siffer skjema. Hvis lenkings modusen ikke er konstruert på en skikkelig måte kan angrep basert på nøkkelkollisjoner anvendes. *E*n nøkkelkollisjon er par av nøkler K_1, K_2 slik at $E(K_1, P) = E(K_2, P)$ for en klartekst *P*. Nøkkelkollisjoner kan konstrueres med ordinære fødselsdagsangrep, men Quisquater [QuD] har vist at det er enklere å benytte sykliske algoritmer hvor en kan produsere kollisjoner både i løpet av kortere tid og med minimalt minneforbruk.

2.8.2.5 Differensielt angrep

Differensiell kryptoanalyse ble oppdaget av matematikerne Biham og Shamir i 1990 [BiS91]. Differensiell kryptoanalyse ser på par av klartekster og kryptotekster. Mere spesifikt ser en på par av kryptotekster hvor klartekstene har spesielle forskjeller. To klartekster kan velges tilfeldig, hvis de har spesielle forskjeller, da kryptoanalytikeren ikke trenger å vite verdiene. Spesielle forskjeller har en høy sannsynlighet for å dukke opp i de resulterende kryptotekstene parene. Disse kalles for karakterestikker. En kan lage modeller som gir større sannsynlighet for å lykkes enn for et tilfeldig angrep.
Spesifikke krav som stilles til iterasjonsfunksjoner på dette og generellt grunnlag omtales i Kapittel 2.5.

2.8.2.6 Analytiske svakheter

For noen skjemaer er det mulig å foreta manipulasjoner som innsetting, sletting, permutering og substituering av blokker. Et stort antall angrep har blitt framsatt som baserer seg på en blokkering av spredningen på innverdiene. Dette betyr at forandringer ikke vil ha noen effekt eller kan kanselleres enkelt på et annet tidspunkt.

2.8.3 Angrep avhengig av signatur skjemaet

I noen tilfeller er det slik at selv om hashfunksjonen er en CRHF er det mulig å knekke signatur skjemaet. Denne typen angrep er en farlig sammenheng mellom hash skjemaet og signatur skjemaet. I de kjente eksemplene på slike sammenhenger har begge skjemaene multiplikative strukturer. Både D. Coppersmith [Cop89] og B. den Boer har fremsatt forslag til slike angrep. Det er vist at sikkerheten til en digitalt signaturskjema, som ikke kan angripes med et valgt meldings angrep, ikke vil bli mere usikkert dersom det kombineres med en CRHF.

2.8.4 Angrep basert på blokk sifferet

Da mange av hash skjemaene baserer seg på blokksiffer, slik som DES, LOKI og IDEA, er det viktig at også de underliggende blokksifferene er sikre. En bør derfor være klar over svakheter som finnes i blokksifferene.

2.8.4.1 Komplement angrep

Et av de første kjente svakhetene med DES er symmetrien ved komplementering, at $\overline{\mathrm{DES}}(K,M) = \mathrm{DES}(\overline{K},\overline{M})$. Denne svakheten kan halvere antallet nødvendige nøkkelsøk og det er også mulig å konstruere trivielle kollisjoner. For eldre versjoner av LOKI er det funnet flere relaterte egenskaper, som både kan angripe forskjellige hash modus og gjøre nøkkelsøk enklere med en faktor på 256. I de siste versjonen er det bare komplement angrep som fungerer.

2.8.4.2 Svake nøkler

En annen kjent svakhet ved DES er de fire svake nøklene, for disse nøklene er kryptoteksten lik klarteksten. Dette betyr at DES(K, DES(K, P)) = P, for alle P. Det finnes også 6 såkalte semi-svake nøkler hvor DES(K_2, DES(K_1, P)) = P, for alle P. Sammenlignet med DES, hadde LOKI flere svake nøkler, men den siste versjonen av LOKI har det samme antallet svake og semi-svake nøklene. Det er også vist at PES og IDEA har samme klartekst og kryptotekst dersom nøkkelen er 0.

2.8.4.3 Fast punkt

Faste punkter i et blokk siffer er en tekst som blir kryptert til seg selv. Da en sikkert blokksiffer er en tilfeldig permutasjon [Pre93], er det sannsynlig at det finnes faste punkt. For hver nøkkel er det en sannsynlighet på $1\text{-}e^{-1}$ for at det er minst et fast punkt. Det vil dog være vanskelig å finne disse. I enkelte tilfeller er det enkelt å lage disse faste punktene. For eksempel kan en utnytte de svake nøklene.

2.8.5 Høynivå angrep

Selv om en skulle kunne sikre seg helt mot de ovenstående angrep, må en sikre seg mot at en gammel melding blir sendt på nytt som en melding eller at flere gamle meldinger blir satt sammen til en ny. For å få dette til må en legge til noe til meldingen som gjør at en legger merke til dersom meldingen er sendt tidligere. Vi har følgende varianter:

2.8.5.1 Tidsstempel (timestamp)

Her legger en til tid og dato da meldingen ble sendt. Hvis klokken er tilstrekkelig nøyaktig, vil den være unik. De største problemene ved å bruke denne metoden er å ha synkroniserte klokker, dette kan løses ved å ha en felles referanseklokke som da må være lett tilgjengelig for både sender og mottaker.

2.8.5.2 Serie nummer

Et unikt tall legges til hver melding. Hvis hver bruker benytter forskjellige tall sekvenser for hver bruker han kommuniserer med, vil tallen være påfølgende og det kan detekteres hvis en melding forsvinner. Dersom hver bruker har en tallsekvens for alle han kommuniserer med, må en sjekke at tallene utgjør en økende sekvens. Dette er bare mulig dersom hver bruker lagrer det høyeste sekvensnummeret for hver kommunikasjon. I denne varianten lar det seg ikke gjøre å finne ut om meldinger har forsvunnet. Et serienummer er mindre kostbart enn et tidsstempel, men tidspunktet meldingen ble sendt kan ikke sjekkes.

2.8.5.3 Tilfeldige tall

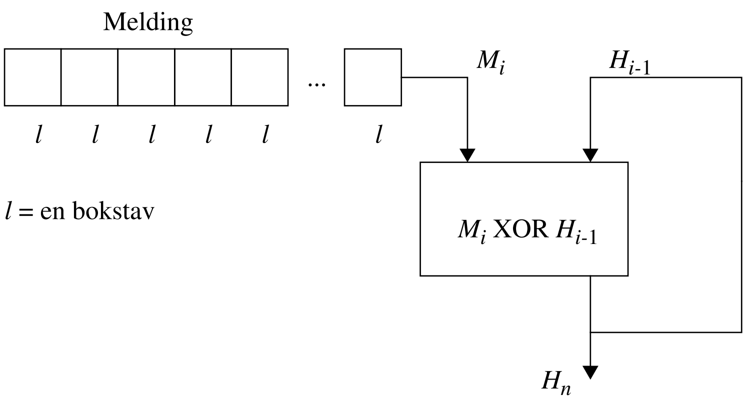
Et tilstrekkelig stort tilfeldig tall blir lagt til meldingen. Et random tall er ikke brukbart dersom alle tidligere sendte tilfeldige tall må lagres for å detektere en resending av en melding.

2.9 Enkle hash funksjoner

For å gjøre det enklere å forstå hvordan en hashfunksjon fungerer har jeg laget to eksempler på hvordan en hash funksjon kan se ut. Jeg har prøvd å gjøre disse så enkle som mulig. Jeg har brukt de tidligere omtalte standard elementene for å lette overgangen til de dedikerte hash funksjonene.

2.9.1 XOR-hash

Denne hash funksjonen baserer seg på et enkelt prinsipp:
En tar den første bokstaven i en streng av vilkårlig lengde som XORes med den neste bokstaven, resultatet XORes så med den neste bokstaven osv. Resultatet vil bli en bokstav. Jeg har også laget en egen hashfunksjon hash16, som er en forenklet versjon av SHA, denne blir beskrevet i neste kapittel.



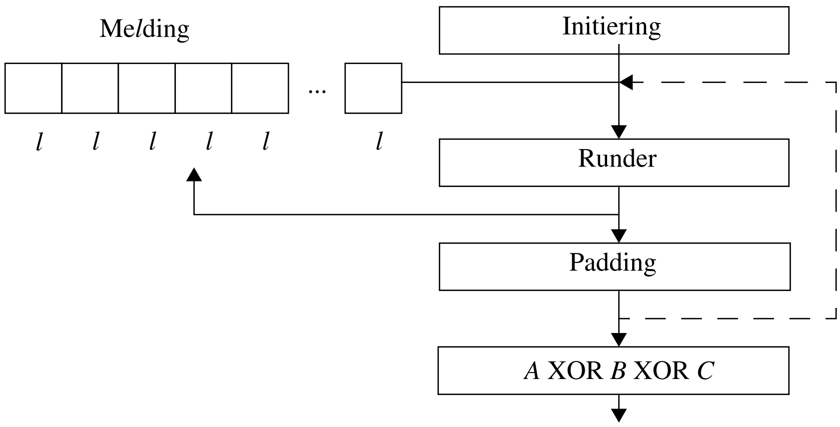
Figur 2.13 *XOR-hash*

2.9.2 Hash-16

Denne hash funksjonen er basert på mange av de samme prinsippene som de mere kjente hash funksjonene, den bruker padding, usymmetriske OR og XOR funksjoner, start variabler og konstanter. Denne hash funksjonen produserer et 16 bits hash. Denne hash funksjonen er i prinsippet en kort versjon av SHA.

- Initialiserer så tre variabler A_0 = `0x0123`, B_0 = `0x89AB`, C_0 = `0xFEDC`.
- Hver runde i hovediterasjonen har 2 runder a 8 operasjoner.
- Gjennomfører så ikkelineære operasjoner på to av A_i, B_i og C_i.
  $F(X, Y) = X$ OR (NOT Y)
  $G(X, Y) = X$ XOR Y
- Resultatet av de ikkelineære funksjonene blir rotert, addert og lagt til A_i, B_i eller C_i.
- Meldingen paddet slik at den blir et multiplum av 64 bit.
- Tilslutt vil resultatet erstatte A_{i+1}, B_{i+1} eller C_{i+1}.
- Det endelige svaret, dvs. hash variabelen, vil være A_i XOR B_i XOR C_i som trunkeres til 16 bits.

Da en ordinær hash funksjon lager hashverdier som forhåpentligvis er for store til å gjøre eksprimenter med, vil jeg bruke denne hash funksjonen til å gjøre eksprimenter på. Dette inkluderer både statistiske forsøk og konkrete fødselsdagsangrep.



Figur 2.14 *Hash-16*

2.10 Praktisk utprøving

For å prøve ut hash16 funksjonen har jeg laget noen C-algoritmer som tester den ut. Disse C-rutinene består bl.a. av en random generator som genererer tilfeldige tekster på 16 bokstaver og generatorer for å lage tekststrenger som benyttes når en skal foreta angrep på hashfunksjonen.

2.10.1 Angrep

Tekstgeneratorene for angrepene er som følger:

```
char *hash16String1(char string[], LONG i)
{
LONG tall, tallmod;
```

```
char tmpstring[5];

     strcpy(string, “JEG HAR TILGODE “);
     sprintf(tmpstring, “%d”, i);
     strcat(string, tmpstring);
     strcat(string, “ KRONER”);

return (string);
}

char *hash16String2(char string[],LONG i)
{
LONG tall,tallmod;
char tmpstring[5];
     strcpy(string, “JEG SKYLDER “);
     sprintf(tmpstring, “%d”, i);
     strcat(string, tmpstring);
     strcat(string, “ KRONER”);

return (string);
}
```

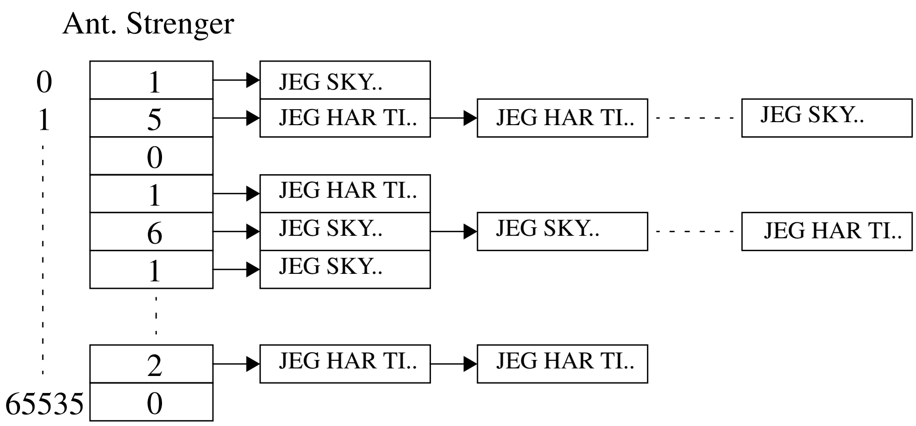
Disse algoritmene genererer teststrenger på følgende form:

```
JEG HAR TILGODE xxxxx KRONER
```

og

```
JEG SKYLDER xxxxx KRONER
```

hvor xxxxx varierer fra 1 til 10000.
Disse teststrengene blir så hashet og lagt inn i en tabell. Tabellen er bygget opp slik at hver tabellposisjon er starten på en lenket liste hvor en legger inn teststrengene og hvor hashverdien bestemmer hvor i listen en skal legge inn strengene. En legger også inn antalle teststrenger som hasher til samme hashverdi.



Figur 2.15 *Testtabell Hash16*

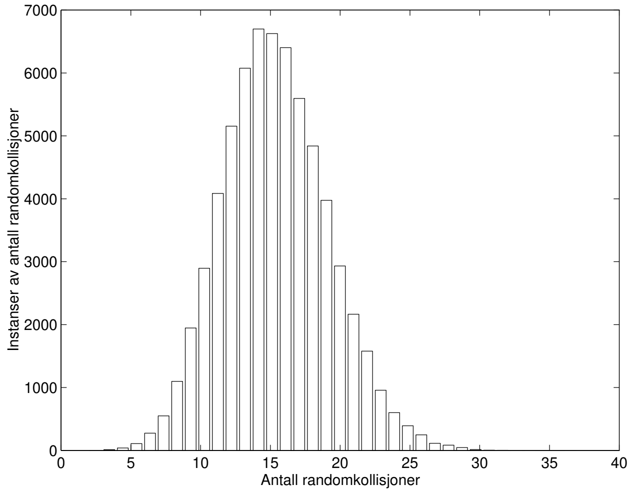
Etter å ha hashet et gitt antall strenger vil det være endel lenkede lister som inneholder som inneholder flere enn et element, disse kan da sjekkes for å finne ut hvilke strenger som har hashet til samme hashverdi. Jeg har laget en algoritme som sjekker listen og skriver ut et passende utvalg av hashverdier med tilhørende teststrenger. For en komplett listing av de forskjellige algoritmene henvises til programlistingen. En vil kunne få f.eks. følgende resultat:

```
Hashverdi: 174a
qsize = 4
Textstring:
JEG HAR TILGODE 4185 KRONER
Textstring:
JEG HAR TILGODE 7873 KRONER
Textstring:
JEG SKYLDER 7625 KRONER
Textstring:
JEG SKYLDER 8494 KRONER
```

Alle disse teststrengene hasher til samme hashverdi, 174a.
I praksis kan en tenke seg at Mallet signerer på meldingen "JEG SKYLDER 8494 KRONER",som har en hashverdi 174a. Signaturen lagres sammen med meldingen. Senere kan Mallet påstå at han ikke signerte på denne meldingen, men på meldingen "JEG HAR TILGODE 7873 KRONER", som har den samme hashverdien. Det bør legges til at dette angrepet ikke er et fødselsdagsangrep i ordets rette forstand. Dette angrepet er egentlig et generellt angrep, men med elementer fra et fødselsdagsangrep. Hvis dette angrepet skulle ha vært et ekte fødselsdagsangrep ville en ha stoppet algoritmen når en hadde funnet en kollisjon. For bedre å ha utnyttet prinsippet burde en også ha benyttet et mindre tabell (256 elementer), og deretter brukt en ikke kryptografisk hashtabell, jfr. Kapittel 2.3.1, til å fordele teststrengene utifra hvert element. Det vil uansett være urealistisk å benytte en hashfunksjon som lager en hashverdi på kun 16 bits, da det vil være enkelt å gjøre et generelt angrep, det vil si å finne en melding som hasher til en på forhånd gitt hashverdi. Ved en hashverdi på 40 bits vil en kunne finne en kollisjon, med en sannsynlighet på over 50%, ved kun en million hashinger. For å gi et lite inntrykk av hvor store verdier det kan bli snakk om bare for å lagre en eventuell tabell, kan jeg nevne at bare for å lage en tilsvarende tabell som figur 2.15 for en 128 bits hashverdi vil det kreve et minne på 340 x 10^{30} MegaByte, hvilket ikke akkurat er praktisk overkommelig. For et fødselsdagsangrep på en 128 bits hashverdi vil det kunne være snakk om mindre tabeller, men praktisk gjennomførbart vil det fremdeles ikke være.

2.10.2 Statistiske vurderinger

I dette kapittelet vil jeg vurdere hvordan hashverdiene fordeler seg i det gitte intervallet, dette for å sjekke om hashfunksjonen lager en tilnærmet random fordeling av hashverdiene. For en nærmere gjennomgang av bakgrunnen for dette henviser jeg til kapittel 4. Jeg vil heller ikke her vise de aktuelle algoritmene som blir benyttet i de statistiske vurderingen, men også her henvise til kapittel 4, hvor de tilnærmet samme rutinene blir benyttet for å teste de statistiske egenskapene til SHA og MD5. Jeg viser her histogrammet for fordelingen av verdiene i intervallet.



Figur 2.16 *Histogram, fordeling av hashverdier på Hash16*

Som en ser av figuren er fordelingen mellom de forskjellige hashverdiene tilnærmet binomisk fordelt.

Kapittel 3 Dedikerte Hash-funksjoner

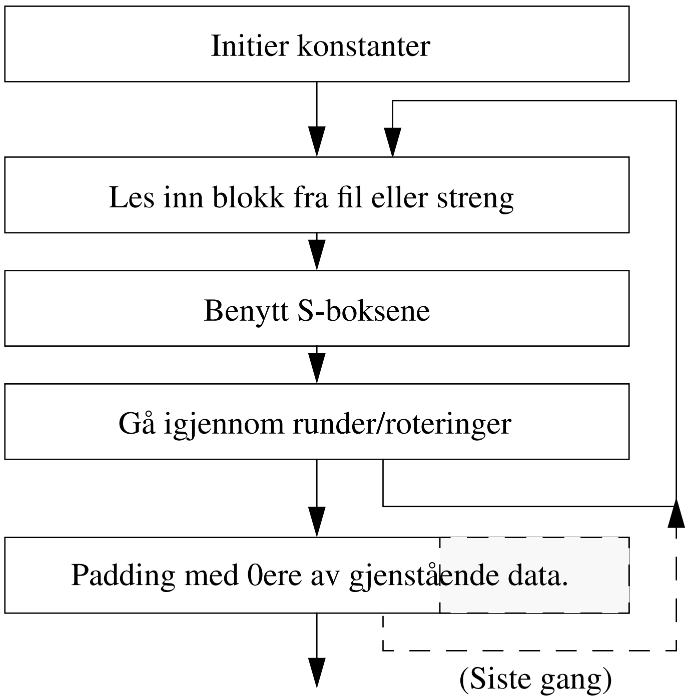
I dette kapittelet vil jeg omtale de mest aktuelle hashfunksjonene som benyttes til digitale signaturer. Jeg vil omtale prinsippene bak dem, sikkerheten og hvordan de fungerer i praktisk bruk. For en nærmere forklaring på hvilke elementer en hashfunksjon består av henviser jeg til kapittel 2.4.

3.1 Snefru

Snefru ble laget av Ralph Merkle [Mer90]. Den lager en 128-bits eller 256-bits hashverdi fra en melding av tilfeldig lengde. Opprinnelig hadde Snefru 2 runder, men ble utvidet til 4 og senere til 8 runder.

3.1.1 Oversikt

Meldingen blir først delt inn i deler a 512 - *m* bits, hvor m er lengden på hashverdien. Hvis den resulterende hasverdien er på 128 bits, vil hver del være på 384-bits. Hvis den resulterende hashverdien er på 256-bits vil delene være på 256 bits lengde.



Figur 3.1 *Oversikt over Snefru*

Kjernen av algoritmen er en funksjon H, som hasher en 512 bits verdi til en m-bits verdi. H ble laget på en slik måte at det skulle være enkelt å regne ut hashverdien, men praktisk umulig å regne ut en melding som genererer en spesifikk hashverdi, dette svarer til krav 3 og 4 i kapittel 2.2.1, One Way Hash Function. De første m bits fra Hs utverdi er hashverdien, resten blir oversett. Den neste blokken blir lagt til hashverdien fra den forrige blokken og hashet på nytt. Den Første blokken blir lagt til en streng av 0-er. Etter den siste blokken, som blir paddet hvis den ikke er et like antall blokker lang, blir de første *m* bits lagt til en binær representasjon av lengden på meldingen og hashet en siste gang. Funksjonen *H* er basert på en annen funksjon *E*, som er en reversibel blokk siffer funksjon som opererer på 512 bits blokker. *H* er de siste *m* bits av resultatet *E* XORet med de første *m* bits av innverdien til *E*.

3.1.2 Sikkerhet

Sikkerheten til Snefru baserer seg på funksjonen *E*, som tilfeldiggjør data ved i flere omganger. Hver omgang består av 64 runder. I hver runde blir forskjellige deler av dataene brukt som innverdi til en substituerings-boks, også kalt en *S*-boks. *S*-boksene fører til at innverdien blir permutert etter et gitt mønster. Snefru har 16 *S*-bokser, som bestå av 256 32 bits ord. Snefru inneholder også endel rotasjoner. Ved bruke differensiell kryptoanalyse viste Biham og Shamir [BiS92] at Snefru med 2 omganger ikke var sikker. De viste at de følgende to meldingene hashet til den samme verdien:

```
3FE15E26 23B7C030 C7089999 90EFC48F A04D87EE 16493392
00046085 00003415 00000000 00000000 00000000 00000000

3FE15E26 23B7C030 C7089999 90EFC48F A04D87EE DF4AF7AE
096C7885 C19EF029 00000000 00000000 00000000 00000000

Felles hashverdi: C8FF5E2C 8F9CF7C7 F08DDAA7 E4F9B44E
```

Også de følgende fire(4) meldingene hasher til samme verdi:

```
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000

00000000 F1301600 13DFC53E 4CC3B093 37461661 CCD8B94D
24D9D35F 71471FDE 00000000 00000000 00000000 00000000

00000000 1D197F00 2ABD3F6F CF33F3D1 8674966A 816E5D51
ACD9A905 53C1D180 00000000 00000000 00000000 00000000

00000000 E98C8300 1E777A47 B5271F34 A04974BB 44CC8B62
BE4B0EFC 18131756 00000000 00000000 00000000 00000000

Felles hashverdi: 2E88E244 E9D4A208 B2D02FBB 72D0EEE6
```

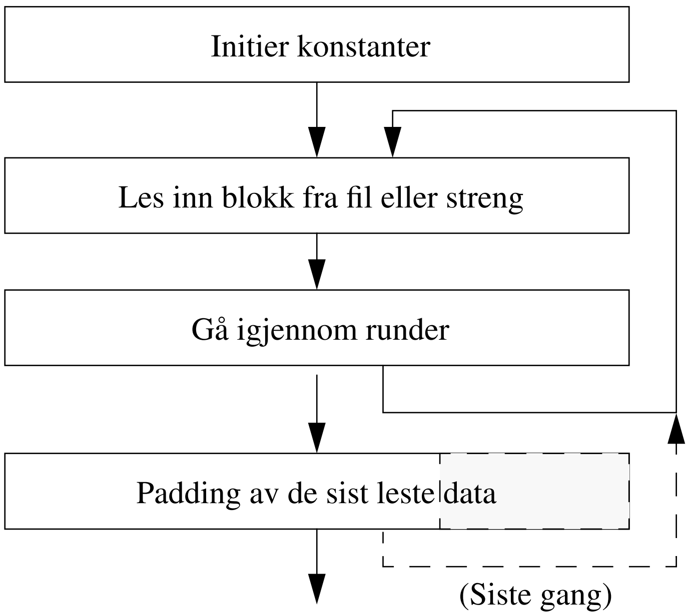
På 128-bits Snefru vil dette angrepet virke bedre enn et tilfeldig angrep for 4 omganger eller mindre. Et fødselsdagsangrep på Snefru tar 2^{64} operasjoner, differensiell kryptoanalyse kan finne par av meldinger som hasher til den samme verdien på $2^{28.5}$ operasjoner for Snefru på 3 omganger og $2^{44.5}$ operasjoner for Snefru på 4 omganger. Å finne en melding som hasher til en gitt verdi ved et tilfeldig angrep trenger 2^{128} operasjoner, differensiell kryptoanalyse bruker 2^{56} operasjoner for Snefru på 3 omganger og 2^{88} for Snefru på 4 omganger. Ralph Merkle anbefaler at en bruker Snefru med minst 8 omganger. Dette gjør den imidlertid vesentlig mindre effektiv enn f.eks. MD5 eller SHA som omtales senere.

3.2 MD4

MD4 ble laget av Ron Rivest[Riv91], den ble annonsert på Eurocrypt 90.

3.2.1 Oversikt

MD4 er en hash-funksjon som opererer på 32 bits ord. Itereringsfunksjonen tar som innverdi en 4 ords initialverdi og en 16 ords del av meldingen. Hash-funksjonen består av 3 runder a 16 operasjoner. I hver operasjon blir en av initialverdiene modifisert som følger: et meldingsord og en ikkelineær funksjon av de tre andre initialverdiene blir lagt til og rotert over et variabelt antall posisjoner. Hver meldings ord blir benyttet nøyaktig en gang i hver runde, men i forskjellige rekkfølger. Etter 3 runder blir de modifiserte initialverdiene lagt til de opprinnelige initialverdiene for at funksjonen skal være irreversibel.



Figur 3.2 *Oversikt over MD4*

Initialverdiene er som følger:

$$h_0 = \texttt{0x67452301}$$
$$h_1 = \texttt{0xEFCDAB89}$$
$$h_2 = \texttt{90x8BADCFE}$$
$$h_3 = \texttt{0x10325476}$$

Konstanter

```
K(j) = 0  (0 ≤ j ≤ 15)
K(j) = 0x5A827999(16 ≤ j ≤ 31)
K(j) = 0x6ED9EBA1(32 ≤ j ≤ 47)
```

De ikkelineære funksjonene

```
f(X,Y,Z) = XY OR (NOT X)Z (0 ≤ j ≤ 15)
```

```
f(X,Y,Z) = XY OR XZ OR YZ (16 ≤ j ≤ 31)
f(X,Y,Z) = X XOR Y XOR Z (32 ≤ j ≤ 47)
```

Utvelgelse av meldingsord

```
r(j) = j           (0 ≤ j ≤ 15)
r(j) = (4*(j-16)) modulo 15 (16 ≤ j < 31)
r(31..47) = 15,0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15
```

Antall posisjoner som roteres

```
n(0..15) = 3,7,11,19,3,7,11,19,3,7,11,19,3,7,11,19
n(16..31) = 3,5,9,13,3,5,9,13,3,5,9,13,3,5,9,13
n(32..47) = 3,9,11,15,3,9,11,15,3,9,11,15,3,9,11,15
```

3.2.2 Sikkerhet

R.Merkle viste i et upublisert resultat at hvis en ser bort ifra den tredje runden kan man enkelt finne to meldinger som er forskjellig i tre bits som hasher til den samme verdien. Angrepet virker for 99.99% av alle initialverdier, inkludert verdiene som er valgt av Rivest. B. den Boer og A. Bosselaers [BoB92] viste hvordan en kunne produsere kollisjoner dersom den første runden ble utelatt. Ideen er å bruke forskjellige meldingsord kun i de midterste 8 operasjoner av både den andre og den tredje runden som begge er avhengige av meldingsblokkene 1, 2, 5, 6, 9, 10, 13 og 14. Metoden utnytter også det faktum at rotasjonene bare skjer over odde posisjoner. Begge angrepen lar seg utføre i løpet av moen få millisekunder på en vanlig PC. Det er fremdeles usikkert om dette kan brukes for å lage en kollisjon for alle tre rundene. Eli Biham [Bih92] diskuterte et differensielt angrep på MD4 som virker for 2 runder, også dette angrepet baserer seg på rekkefølgen på meldingsordene. I forbindelse med Eurocrypt 95 ble det presentert et angrep på MD4. Hans Dobbertin [Dob95b] benyttet metoden som ble brukt for å finne kollisjoner på RIPEMD, jfr kapittel 3.5.2.
Meldingen M består av sekvensen av de følgende 32-bits hexadesimaleord:

```
13985E12 748A810B 4D1DF15A 181D1516 2D6E09AC 4B6DBDB9
6464B0C8 FBA1C097 ABE17BE0 ED1ED4B3 4120ABF5 20771029
20771027 FDFFFBFF FFFFBFFB 6774BED2
```

Meldingen M’ består av den følgende sekvensen 32-bits hexadesimaleord:

```
13985E12 748A810B 4D1DF15A 181D1516 2D6E09AC 4B6DBDB9
6464B0C8 FBA1C097 ABE17BE0 ED1ED4B3 4120ABF5 20771029
20771028 FDFFFBFF FFFFBFFB 6774BED2
```

Meldingene M og M’ har en felles hashverdi:

```
711AD51B BBAB5E22 618B1C76 17C15892
```

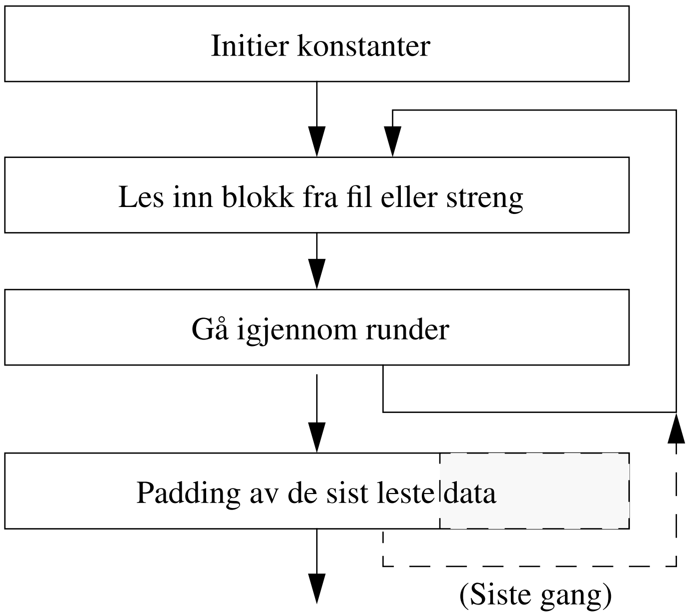
Dette resultatet, samt flere tilsvarende kan en finne i løpet av få minutter på en ordinær PC.

3.3 MD5

Ron Rivest [Riv92] lanserte en utvidelse av MD4 på bakgrunn av angrepene på MD4, denne versjonen inneholder endel forandringer som er ment å styrke sikkerheten.

3.3.1 Oversikt

Da denne hashfunksjonen er en utvidelse av MD4 inneholder den mange av de samme elementene. Den har fire runder, mot MD4s 3 runder og en multiplekser funksjon i den første og andre runde. Rekkefølgen på meldingsordene er forandret og 16 forskjellige roteringer har kommet til. Hver omgang har også sin unike konstant og kjernen i algoritmen er forandret.



Figur 3.3 *Oversikt over MD5*

Initialverdier:

$$h_0 = \texttt{0x67452301}$$
$$h_1 = \texttt{0xEFCDAB89}$$
$$h_2 = \texttt{0x98BADCFE}$$
$$h_3 = \texttt{0x10325476}$$

Konstanter:

$$K(\mathrm{j}) = \text{første 32 bits av } \sin(\mathrm{j}+1) \quad (0 \le \mathrm{j} \le 63)$$

De ikkelineære funksjoner:

$$f(X,Y,Z) = XY \text{ OR } ((\text{NOT } X)Z) \quad (0 \le \mathrm{j} \le 15)$$
$$f(X,Y,Z) = XZ \text{ OR } (Y(\text{NOT } Z)) \quad (16 \le \mathrm{j} \le 31)$$
$$f(X,Y,Z) = XY \text{ OR } XZ \text{ OR } YZ \quad (32 \le \mathrm{j} \le 47)$$
$$f(X,Y,Z) = X \text{ XOR } Y \text{ XOR } Z \quad (48 \le \mathrm{j} \le 63)$$

Utvelgelse av meldings ord

```
r(j) = j        (0 ≤ j ≤ 15)
r(j) = (1+5*(j-16) modulo 16(16 ≤ j ≤ 31)
r(j) = (5+3*(j-32) modulo 16(32 ≤ j ≤ 47)
r(j) = 7*(j-48) modulo 16(48 ≤ j ≤ 63)
```

Antall posisjoner som roteres

```
n(0..15) = 7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22
n(16..31) = 5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20
n(32..47) = 4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23
n(48..63) = 6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21
```

3.3.2 Sikkerhet

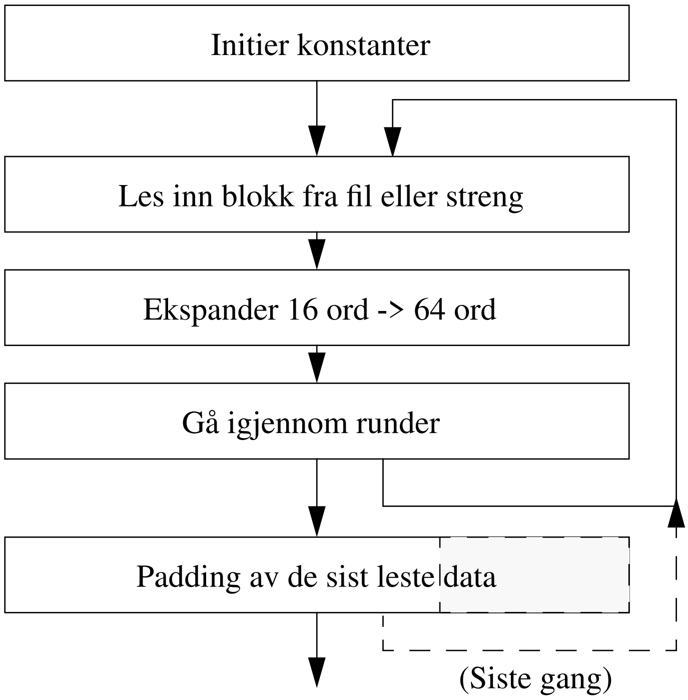
B. den Boer viste at det finnes en omtrentlig relasjon mellom fire tilfeldige påfølgende konstanter, dette ble sammen med A. Bosselaers videreutviklet. til å omfatte et angrep [BoB93] som produserer pseudo kollisjoner, det vil si at de kan konstruere to variabler og en enkelt meldingsblokk som gir den samme hasverdien. Dette betyr ikke at hele MD5 er knekket, men at en av forutsetningene for en kollisjons fri hashfunksjon, at den interne kompresjonsfunksjonen er kollisjonsfri, er brutt. T. Berson [Ber92] diskuterte hvordan en kunne benytte differensielle teknikker for å knekke en enkelt runde av MD5, men dette angrepet er ikke effektivt for bruk på alle fire rundene.

3.4 SHA

National Institute of Standards and Technology laget denne hashfunksjonene sammen med the National Security Agency for at den skulle brukes i Digital Signature Algorithm (DSA). DSA er algoritmen i Digital Signature Standard (DSS), jfr. kapittel 1.6, som er et forsøk på å lage en standard for signering av meldinger. Den ble vedtatt som en standard 11 mai, 1993. men ble revidert 11 juli, 1994 på grunn av en liten feil som gjorde den litt mindre sikker.

3.4.1 Oversikt

Denne standarden spesifiserer en hash algoritme, Secure Hash Algorithm (SHA) som lager et hash av en melding $< 2^{64}$ bits input. SHA produserer en 160 bits hashverdi. Denne hashverdien brukes i Digital Signature Algorithm (DSA), jfr. kapittel 1.6. DSA genererer eller verifiserer en signatur for meldingen. SHA ble laget for å ha de følgende egenskaper: Det er praktisk umulig å finne en melding som tilsvarer et hashverdi eller å finne to forskjellige meldinger som produserer det samme meldingsutsnittet. Den var, med andre ord, ment å inneha egenskapene til en Collision Resistant Hash Function (CRHF), jfr. kapittel 2.2.2.



Figur 3.4 *Oversikt over SHA*

SHA er bygget opp som følger:
Fem 32-bits variabler initialiseres som følger:

A = 0x67452301
B = 0xEFCDAB89
C = 0x98BADCFE
D = 0x10325476
E = 0xC3D2E1F0

Hovedløkken på programmet tar for seg meldinger 512 bits i gangen og fortsetter til den har vært igjennom alle blokkene a 512 bits (etter padding).
Først blir de fem variablene kopiert inn i 5 nye variabler, A som AA, B som BB, C som CC, D som DD, E som EE.
Hovedløkken har 4 runder a 20 operasjoner, hver operasjon utfører en ikkelineær operasjon på tre av A, B, C og D.
SHAs ikkelineære funksjoner er som følger:

$f(X,Y,Z)$ = XY OR (NOT X)Z for de første 20 operasjoner.
$f(X,Y,Z)$ = X XOR Y XOR Z for de neste 20 operasjoner.
$f(X,Y,Z)$ = XY OR XZ OR YZ for de neste 20 operasjoner.
$f(X,Y,Z)$ = X XOR Y XOR Z for de siste 20 operasjoner.

Det er også 4 konstanter som brukes i algoritmene:

K_1 = 0x5A827999 for de første 20 operasjonen.
K_2 = 0x6ED9EBA1 for de neste 20 operasjonen.

```
K3 = 0x8F1BBCDC for de neste 20 operasjonen.
K4 = 0xCA62C1D1 for de siste 20 operasjonen.
```

Meldingen forandres fra seksten 32-bits ord (M_0 til M_{15}) til åtti 32 bits ord (W_0 til W_{79}) ved å bruke den følgende algoritmen:

```
Wt = Mt, for t = 0 til 15
Wt = Wt-3 XOR Wt-8 XOR Wt-14 XOR Wt-16, for t = 16 til 79.
```

Dette var den opprinnelige versjonen fra 1993, den ble som tidligere nevnt modifisert i 1994, den ble da som følger:

```
Wt = Mt, for t = 0 til 15
Wt = S(1, Wt-3 XOR Wt-8 XOR Wt-14 XOR Wt-16) for t = 16 til 79.
```

Hvis *t* er operasjonsnummeret (fra 1 til 80), representerer M_j den *j*te subblokken av meldingen (fra 0 til 15) og S(*n*, X) representerer et venstreskift *n* bits, ser de 80 operasjonene ut som følger:

```
TEMP = S(5, A) + f(B,C,D) + E + Wt + Kt
E = D
D = C
C = S(30, B)
B = A
A = TEMP
```

Tilslutt legges *A*, *B*, *C*, *D* og *E* til *AA*, *BB*, *CC*, *DD* og *EE* og algoritmen fortsetter med den neste datablokken.
Siste del av meldingen blir paddet slik at den blir et multiplum av 512 bit, paddingen foregår ved at en først legger til en enkelt bit deretter så mange 0er som det er nødvendig for å fylle opp 512-64 bits. Til slutt legges en 64 bits representasjon av lengden på den opprinnelige meldingen til.
Hashverdien vil tilslutt være $A \| B \| C \| D \| E$, hvor $\|$ representerer konkatenering. Jeg har valgt SHA sammen med MD5 som de algoritmene jeg vil vurdere nærmere, fordi SHA er vedtatt som en standard og fordi MD5 pr. i dag er en de facto industristandard. Jeg vil i Kapittel 4 ta for meg statistiske vurderinger med hensyn på SHA og MD5.

3.4.2 Sikkerhet

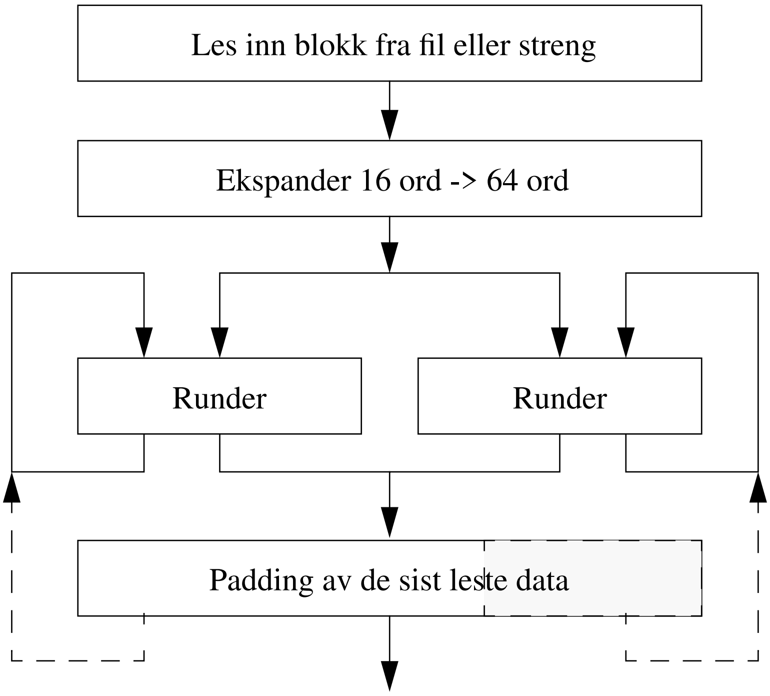
SHA kan på mange måter sammenlignes med MD5, de bygger begge på MD4, hvor en legger til flere runder. SHA kan regnes for å være like sikker som MD5, dog har den en 160 bit hashverdi som gjør den mere motstandsdyktig mot algoritmeuavhengige angrep, jfr. 2.8.1 enn MD5. Det er, såvidt meg bekjent, ikke publisert noen angrep på SHA.

3.5 RIPEMD

RIPEMD ble laget For EUs RACE prosjekt [RIPE93]. Algoritmen er en variasjon av MD4. Den kjører to parallele varianter av MD4 hvor rotasjonene og hvordan deler av meldingen er organisert er forandret.

3.5.1 Oversikt

RIPEMD lager en hashverdi på 128 bits fra en melding av vilkårlig størrelse, den baserer seg på en iterert kompresjons funksjon *compress* som komprimerer en 20 ords innverdi til en 4 ords utverdi. For å få en forståelse av hvordan denne funksjonen fungerer, defineres først basisfunksjonene i denne *compress* funksjonen.



Figur 3.5 *Oversikt over RIPEMD*

Compress bruker tre basisfunksjoner som gjør om tre ord til ett. Disse funksjonene er definert som følger:

$$F(X,\ Y,\ Z) = XY \text{ OR } (\text{NOT } X)Z$$
$$G(X,\ Y,\ Z) = XY \text{ OR } XZ \text{ OR } YZ$$
$$H(X,\ Y,\ Z) = X \text{ XOR } Y \text{ XOR } Z$$

Disse tre funksjonene er brukt i seks andre operasjoner. Disse brukes på en melding som består av 16 ord X_0, X_1,...,X_{15}. For ordene A, B, C og D og heltall $0 \leq k \leq 16$ og $0 \leq n \leq 32$ er disse operasjonen definert som følger:

$$FF() \Rightarrow A = S(n,\ ((A+F(B,C,D)+X_k) \bmod 2^{32}))$$
$$GG() \Rightarrow A = S(n,\ ((A+G(B,C,D)+X_k+0x5A827999) \bmod 2^{32}))$$
$$HH() \Rightarrow A = S(n,\ ((A+H(B,C,D)+X_k+0x6ED9EBA1) \bmod 2^{32}))$$
$$FFF() \Rightarrow A = S(n,\ ((A+F(B,C,D)+X_k+0x50A28BE6) \bmod 2^{32}))$$
$$GGG() \Rightarrow A = S(n,\ ((A+G(B,C,D)+X_k) \bmod 2^{32}))$$
$$HHH() \Rightarrow A = S(n,\ ((A+H(B,C,D)+X_k+0x5C4DD124) \bmod 2^{32}))$$

Først blir meldingen paddet til en passende lengde og representert av en sekvens av ord. Etter å ha initialisert 4 ord, som i første runde er A_0, B_0, C_0 og D_0, en lager så en sekvens ved å legge til 16 meldings ord, de resulterende

20 ordene blir så komprimert til 4 ved å anvende *compress* resultatet blir så gjentatte ganger lagt til 16 meldingsord og komprimert inntil det ikke gjenstår noen meldingsord, dette fører til at resultatet er på 4 ord, dvs 128 bits.De initielle ordene A_0, B_0, C_0 og D_0 er som følger:

$$A_0 = \texttt{0x67452301}$$
$$B_0 = \texttt{0xEFCDAB89}$$
$$C_0 = \texttt{0x98BADCFE}$$
$$D_0 = \texttt{0x10325476}$$

RIPEMD har to parallelle kopier av en MD4 lignende kompresjonsfunksjonene, disse blir kombinert til slutt.

3.5.2 Sikkerhet

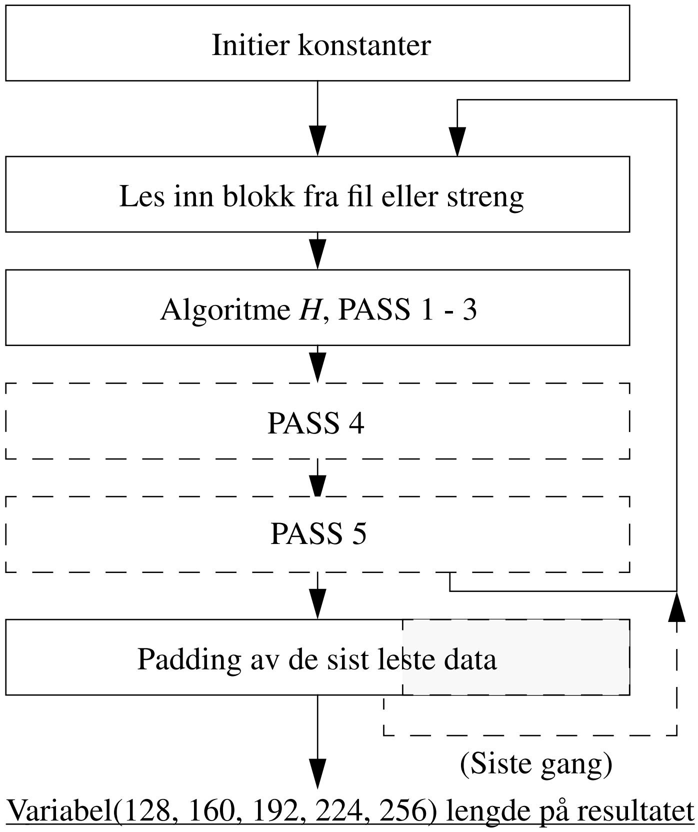
Denne hashfunksjonen sies å være en forbedring av MD4. Grunnlaget for denne forbedringen er at det ble publisert et angrep på de to av rundene i MD4. Designerne av RIPEMD mener at den er sikrere enn MD5 på grunn av disse parallelle funksjonene. RIPEMDs interne funksjonen *compress* er testet utifra kriteriene i kapittel 2.2.2, konstruktørene mener at disse kravene er tilfredsstilt. Det ble fremsatt et angrep på den andre og tredje runden i *compress* funksjonen av Hans Dobbertin [Dob95a]. Angrepet baserer seg på en kombinasjon av differensiell kryptoanalyse, jfr. Kapittel 2.8.2.5, analyse av svakheter, jfr. Kapittel 2.8.2.6 og genetisk programmering. Han benytter seg av to Lemma som blir bevist. Lemma 1 gir simultane kollisjoner på deler av compress. Lemma 2 gir en metode for å finne "bakoverkollisjoner" ved hjelp av en genetisk algoritme fram til posisjonen hvor Lemma 1 gir en løsning. Tilsammen gir Lemma 1 og Lemma 2 en metode for å regne ut kollisjoner på to runder av RIPEMD med omtrent 2^{96} forskjellige initialverdier (*IV*). Dette vil være praktisk mulig å regne ut ved hjelp av genetiske algoritmer og vil ta omtrentlig tre dager på en 486 PC.

3.6 HAVAL

HAVAL ble laget i Australia, University of Wollongong, av Yuliang Zheng, Josef Pieprzyk og Jennifer Seberry [ZPS92]. HAVAL er en hashfunksjon hvor lengden på hashverdien er variabel, den kan produsere hashverdier på lengde 128, 160, 192, 224 eller 256 bits. En kan også variere mellom 3, 4 og 5 omganger for på denne måten øke sikkerheten på bekostning av hastigheten.

3.6.1 Oversikt

HAVAL kan sees på som en utvidelse av MD5, den behandler meldingene i blokker a 1024 bits, som er dobbelt så mye som MD5. Den erstatter MD5 enkle ulineære funksjoner med sterkt ulineære funksjoner med 7 variabler. Hver runde benytter den samme funksjonen, men for hver runde blir det benyttet forskjellige permutasjoner av innverdiene. Den benytter en ny meldingsrekkefølge og forskjellige konstanter. HAVAL benytter også to rotasjoner.



Figur 3.6 *Oversikt over HAVAL*

HAVAL vil først padde meldingen for å sørge for at den er en multippel av 1024 bits, denne paddingen legger til 0ere, antallet bits i meldingen, antallet bits i hashverdien (valgt av bruker), antallet omganger (valgt av bruker) og versjonsnummeret av HAVAL. Hvis en nå tenker seg at meldingen, som er paddet, er $B_{n-1}B_{n-2}...B_0$, hvor hver B_i er en 1024-bits blokk. HAVAL starter med blokk B_0 og et 8-ords konstant (256-bits) konstant $D_0 = D_{0,7}D_{0,6}...D_{0,0}$ som taes fra desimaltalldelen av = 3.141592...(π), og regner ut fra meldingen $B_{n-1}B_{n-2}...B_0$ blokk for blokk, med andre ord så blir meldingen komprimet ved gjentatte ganger å regne ut:

$$D_{i+1} = H(D_i, B_i)$$

hvor i går fra 0 til n-1 og H er basis komprimeringen i HAVAL. H kan beskrives mere nøyaktig, da den er forskjellig for hver runde/omgang:

$$E_0 = D_{in}$$
$$E_1 = H_1(E_0, B)$$
$$E_2 = H_2(E_1, B)$$
$$E_3 = H_3(E_2, B)$$
$$E_4 = H_4(E_3, B), \text{ hvis 4 eller 5 runder}$$
$$E_5 = H_5(E_4, B), \text{ hvis 5 runder}$$
$$D_{ut} = E_3 + E_0, \text{ hvis 3 runder}$$

D_{ut} = E_4 + E_0, hvis 4 runder
D_{ut} = E_5 + E_0, hvis 5 runder

hvor + representerer en ord vis heltalls addisjon modulo 2^{32}.
Til slutt justerer HAVAL 256-bits verdien D_n som er resultatet av utregningen ovenfor og justerer den slik at den har den lengden som er spesifisert av brukeren. Resultatet er da hashverdien av meldingen *M*.

3.6.2 Sikkerhet

I likhet med de andre hashfunksjonene er det heller ikke mulig å bevise at HAVAL er sikker. Pr. i dag er det ikke publisert noen angrep på HAVAL, det eneste angrepet som vel kan benyttes er Bersons angrep på en enkelt runde av MD5, denne kan også benyttes på en enkelt runde på HAVAL, det er dog ikke sannsynlig at dette angrepet kan benyttes på to eller flere runder. Det er også publisert et angrep av Boer og Bosselaers på en enkelt runde av MD5, men den kan ikke benyttes direkte på HAVAL. Det virker derfor som at den beste metoden for å finne kollisjoner ved bruk av HAVAL pr. i dag er å bruke fødselsdagsangrep.

3.7 Hastighet

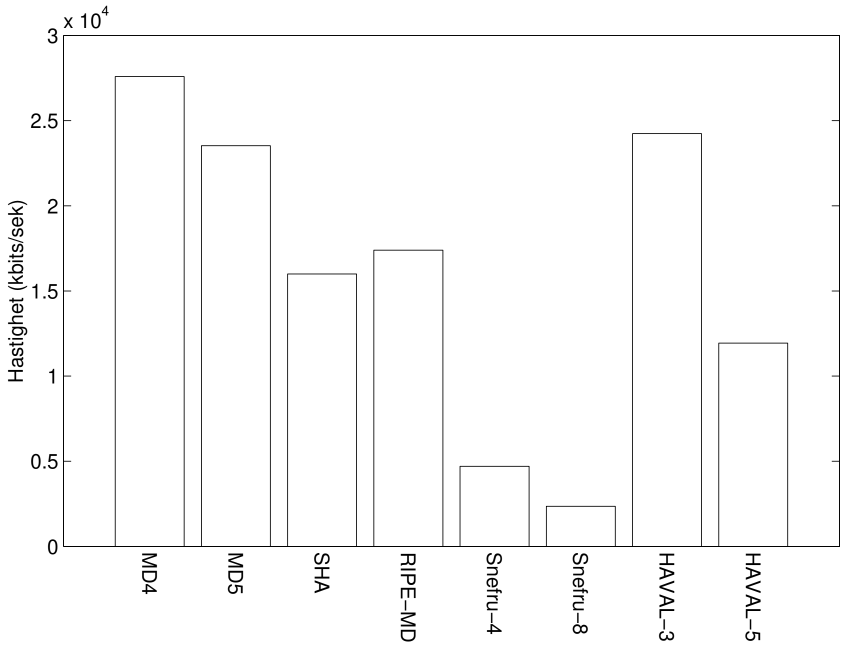
I dette kapittelet vil jeg vurdere hastigheten på de forskjellige dedikerte hashfunksjonene mot hverandre. Hashfunksjonene er implementert i C.

Hashfunksjon	IBM 386/16 (kbit/sek) [Pre93]	Sun SPARCstation 10 (kbit/sek)	DEC Alpha (kbit/sek)
Snefru-4	520	4706	4061
Snefru-8	270	2353	1900
MD4	2669	27590	22220
MD5	1849	23530	18180
SHA	710	16000	15690
RIPEMD	1334	17390	16660
HAVAL (3, 128)		24240	25000
HAVAL (5, 128)		11940	12310

Figur 3.7 *Hastighet på hashfunksjoner, implementert i C.*

Det er på sin plass med noen kommentarer til tallene i tabellen.
For RIPEMD gjelder tallene for versjon 1.0, da denne implementasjonen ligner mest på implementasjonene av de andre hashfunksjonene. RIPEMD har kommet i både versjon 1.2 og 2.0, disse versjonene må kompileres sammen med rsaref bibliotekene og kan vanskelig sammenlignes direkte med implementasjonene av SHA og MD5. Implementasjonene av versjonene 1.2 og 2.0 av RIPEMD er optimalisert for effektivitet, og vil følgelig være bedre.

HAVAL er i følge tabellen mindre effektiv enn MD5, hvis en benytter HAVAL med 5 runder, dette i motsetning til hva forfatterne har oppgitt i sin dokumentasjon. En av årsakene til dette antas å være at HAVAL er optimalisert for “Little endian” maskiner, det vil si maskiner som plasserer det minst signifikante ordet (byte) først i minnet, mens Sun SPARC er en “Big endian”, det vil si at den plasserer det mest signifikante ordet først i minnet. Dette kan også sees i kolonnen for DEC Alpha, som er av typen “Little endian”. Effektiviteten på HAVAL er den samme uansett antall bit i hashverdien, da utregningene er de samme, bare utskriftene er forskjellige.
Verdiene som gjelder for målingene på DEC Alpha bør kun betraktes relativt til de andre målingene på samme maskin og ikke i forhold til Sun SPARCstation. Dette fordi implementasjonene som ble benyttet, ikke tar i betraktning at DEC Alpha er en 64 bits maskin. Det er også en stor ulempe for Alpha maskinene at de opprinnelige definisjonene for hashfunksjonene er definert med hensyn på 32 bits variabler. En DEC Alpha vil måtte maskere bort 32 bits for hver variabel, da alle interne variabler i en DEC alpha er 64 bits. Etterhvert som 64 bits maskiner blir mere og mere vanlig og kravene til hastighet blir større, vil det nok bli nødvendig å definere hashfunksjoner som benytter 64 bits verdier. Jeg valgte konsekvent de samme optimeringsopsjonene og den samme kompilatoren på de forskjellige hastighetstestene for å få mest mulig sammenlignbart resultat. Jeg benyttet GNUs C kompilator, da den godtar både Kernighan/Ritchie og ANSI C, samt en miks av disse. Den relative hastighetsforskjellen mellom de forskjellige hasfunksjonene ville ha forandret seg dersom en hadde benyttet forskjellige kompilatorer/opsjoner for de forskjellige hashfunksjonene, se ellers Kapittel 3.8.



Figur 3.8 *Hastighet på Hash funksjoner (Sun SPARC)*

3.7.1 Kompilering av hashfunksjoner

Av hensyn til hastigheten er det meget viktig å passe på hvilke opsjoner en bruker når en kompilerer på UNIX maskiner. Det var tildels store avvik når en benyttet forskjellige optimeringsopsjoner ved kompilering av de forskjellige hashfunksjonene, dette gjelder spesielt DEC Alpha maskiner. Jeg testet 3 av hashfunksjonene på 2 forskjellige kompilatorer med forskjellige optimerings varianter. Kompilatorene som ble benyttet var GNUs C kompilator *gcc*, og DECs optimerte kompilator for Alpha *cc -migrate*. Jeg vil illustrere forskjellen for HAVAL, MD5 og SHA. De bør nevnes at -O3 og -O4 er maksimal hastighetsoptimering for henholdsvis gcc og cc -migrate.

	gcc -O0	gcc -O3	cc -migrate -O0	cc -migrate -O4
Haval(3,128)	10390	33330	12120	40000
Haval(5,128)	5930	14040	7340	16000
MD5	9091	27586	10256	36364
SHA	4420	8511	3980	32000

Figur 3.9 *Hastighet ved forskjellige kompileringsopsjoner på DEC Alpha, kbits/sek.*

3.8 Implementering på 64 bits arkitektur

Det viser seg at en må ta spesielle forhåndsregler ved kompilering på 64 bits arkitektur. De fleste maskinene som benyttes i dag er 32 bits, dette skaper problemer når en skal kompilere på for eksempel DEC Alpha som har en 64 bits arkitektur. I praksis betyr dette at en *long integer* bruker 64 bits og en vanlig *integer* bruker 32 bits. De fleste implementasjonene av hashfunksjoner, slik som SHA og MD5, ble laget før det var aktuellt med 64 bits arkitekturer og det er derfor ikke tatt hensyn til disse problemene. Heltallene som benyttes i SHA er definert som følger:

```
typedef unsigned long LONG;
```

Dette skaper problemer hvis en prøver å implementere dette direkte på en 64 bits maskin, da standarden er definert med 32 bits operasjoner og resultatet vil bli at det samme dokumentet vil få forskjellig hashverdi avhengig av hvilken maskin (32- eller 64-bits) en hasher dokumentet på. For å unngå dette problemet må en definere heltallene på følgende måte.

```
typedef unsigned int LONG;
```

dette vil gi 32 bits heltall og resultatene vil bli riktige. I lys av dette er det viktig, uansett hvilken plattfor en kompilere programmene på, at en er klar over hvordan heltallene blir representer på de forskjellige plattformene. En bør også benytte test modusen (sha -x, md4/5 -x, haval -c, ripemd -a) for å sjekke om resultatene blir korrekte. Snefru benytter et eget program checkhash.

Kapittel 4 Statistiske vurderinger av SHA og MD5

4.1 Pseudorandomtall og hashfunksjoner

En hashfunksjon bør ha en karakteristikk som ligner på et randomgenerator, årsaken til dette vil bli vurdert etter at jeg har gjennomgått kravene en kan stille til en randomgenerator. Det er i all vesentlighet tre kriterium som må tilfredstilles av en random generator:

1. Resultatet ser random ut. det vil si at en kan foreta statistiske tester på de generete sekvensene, uten finne særegenheter.
2. Sekvensen er uforutsigbar, det vil si at det ikke er mulig å forutse hvilken verdien det neste genererte tallet vil ha. Dette gjelder selv om en har full oversikt over algoritmen eller maskinvaren som genererer sekvensen.
3. Sekvensen kan ikke gjentas, det vil si at den samme sekvensen ikke kan gjentas ved en senere anledning. Hvis en kjører generatoren med det samme initialverdien flere ganger, vil en få forskjellig svar hver gang.

Det er ganske opplagt at en ikke kan lage en pseudorandom generator, ved hjelp av en datamaskin, som tilfredsstiller alle disse kravene. Den viktigste grunnen for dette er at en datamaskin i seg selv er forutsigbar, en datamaskin kan kun produsere forutsigbare sekvenser. Det er derfor det ideelle å lage pseudorandomgeneratorer som i størst mulig grad tilfredsstiller disse kriteriene. For en hashfunksjon er det ikke ønskelig å lage en generator som tilfredsstiller kriterium 3, da en ønsker å få det samme resultatet hver gang en hasher den aktuelle filen og også få den samme hashverdien uansett på hvilken maskin en utfører hashingen på. Det er også tvilsomt om kriterium 2 kan oppfylles, da en nødvendigvis vil kunne forutsi hashverdien dersom filen er hashet tidligere, det er dog vanskelig å forutsi hashverdien dersom en ikke har hashet denne filen før. På grunnlag av disse vurderingene vil jeg konsentrere meg om å finne ut om gitte hashfunksjoner tilfredsstiller kriterium 1. En hashfunksjon bør fordele hashverdiene jevn i intervallet, dersom dette ikke er tilfelle vil det bli enklere å angripe hashfunksjonen. For eksempel vil det være enklere å utføre et fødselsdagsangrep, jfr. 2.8.1.2, hvis de antatte hashverdiene ligger i et mindre intervall enn det som er ønskelig. Det vil også være enklere å utføre et møtes i midten angrep hvis en kun trenger å forholde seg til en hashfunksjon som fordeler hashverdiene i et gitt (begrenset) intervall. Det formelle grunnlaget for dette kapittelet er vurderinger angående pseudorandom generatorer, spesielt boolske funksjoner og Feistel siffer [Fei73]. En grundig innføring i dette ligger utenfor det som denne oppgaven er ment å inneholde, for spesielt interessert henvises til [Pre93]. Formålet med dette kapittelet er å vurdere om de mest brukte hashfunksjonene har en god, tilnærmet randomfordeling av hashverdiene.

De fleste av de dedikerte hashfunksjonene benytter blokksiffer i iterasjonsfunksjonen. Årsaken til det er at disse prinsippene er enkle å implementere både i hardware og software. Design og implementasjon av blokksiffer baseres ofte på de teoretiske arbeidene av Shannon [Sha49]. Han mente at flere runder med spredning og blanding (diffusjon/confusjon) ville føre til en sterkere kryptografisk algoritme. DES og de fleste blokksiffer algoritmer benytter seg av Feistel type permutasjoner [Fei73], en slik permutasjon er en funksjon som er kontrollert av en nøkkel for å foreta den ønskede spredning og blandingen. Luby og Rackoff [LuR88] viste at tre runder av Feistel type permutasjoner fører til et blokksiffer som er sikker mot valgt klartekst angrep (chosen plaintext attack). Selv om bakgrunnen for valgene bak rundene i DES ikke er offentliggjort antas de å ha sin bakgrunn i teorier omkring Feistel liknende blokksiffer.

4.2 Vurdering av maskinens pseudorandomfunksjon

Et viktig element når en skal foreta statistiske vurderinger som baserer seg på en random generator er at randomgeneratoren er best mulig, slik at resultatene en får ikke er en direkte følge av en utilfredstillende randomgenerator. Da det ikke er mulig å finne en ekte randomgenerator som kan kjøres som en algoritme og jeg ikke ville bruke en stor tabell på grunn av minneproblemer, måtte jeg benytte en pseudorandom generator. Jeg vurderte to randomgeneratorer. Den første er den ordinære pseudo-random generatoren som finnes i hvilket som helst programmeringsspråk, i dette tilfellet i C.

```
(I)  PseudoRandomTall = random()
```

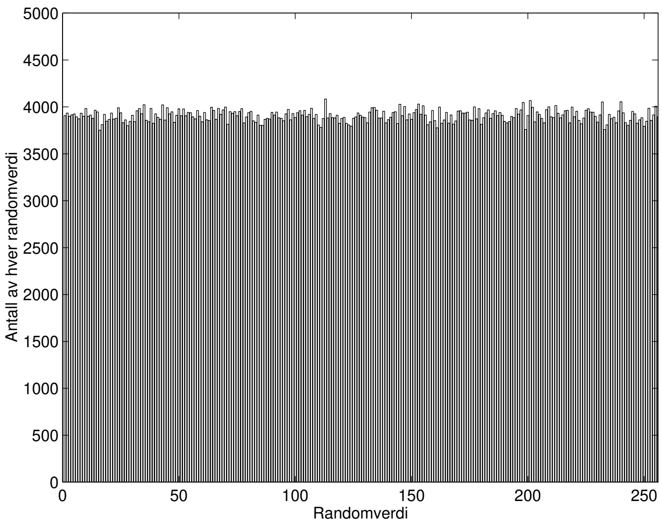
Denne pseudo-random generatoren (I) gir tall i et gitt intervall. Denne randomgeneratoren er ikke regnet for å være særlig bra, men spørsmålet var om den var brukbar for mitt formål. Den andre pseudorandomgeneratoren er en variant av den første, men i dette tilfellet benytter en også maskinenes innebyggete klokke og ikke bare en gitt sekvens slik som den første. Algoritmen er som følger:

```
(II) int uniform (int max)
     {
          struct timeval tp;
          struct timezone tzp;

          gettimeofday (&tp, &tzp);
          srand((375467*tp.tv_sec)+(235783*tp.tv_usec));
          return (rand()% max + 1);
     }
```

som benyttes på følgende måte for å få pseudorandom tall i et gitt intervall:

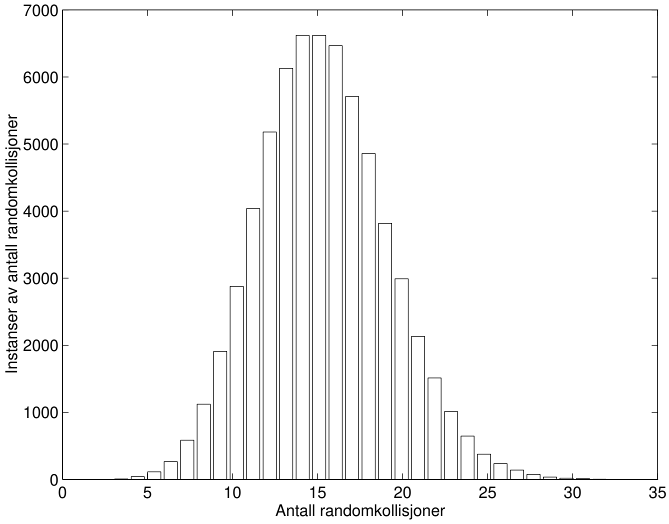
```
     PseudoRandomTall = uniform(intervall)
```

Jeg foretok tester på begge variantene, for å se om de fungerte til mitt bruk. Testingen foregikk ved å generere tall i et gitt intervall, som så ble sortert. De ble talt opp for å se hvordan verdiene fordelte seg i intervallet. Den ideelle randomfunksjonen vil spre verdiene jevnt i intervallet. Dette er dog en sannhet med modifikasjoner, da dette kun vi være tilfellet dersom antall verdier går mot uendelig. Jeg vil presentere endel grafer for å illustrere hva jeg mener. Grafene er laget i MATLAB ved å importere en datafil og lage histogram og bardiagram. De første to figurene er fordelingen av 1000000 genererte pseudorandomtall i intervallet 1 til 256.



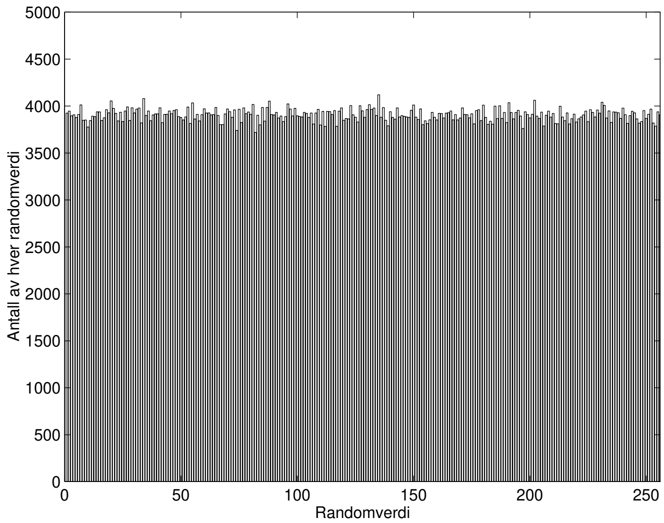
Figur 4.1 *Bargraf av 1000000 randomverdier, generert av (II).*

Verdiene bør fordele seg jevnt i hele intervallet, hvor tyngdepunktet bør ligge omkring 3906 (1000000/256) instanser av hver verdi. Et histogram over denne fordelingen ville ikke inneholde så mye informasjon da så mange (1000000) instanser fordelt på forholdsvis få (256) mulige verdier vil gi store avvik. Jeg har derfor valgt å fordele 1000000 instanser på 65536 forskjellige hashverdier for histogrammet. Histogrammet viser fordelingen av de forskjellige verdiene. Verdiene bør fordele seg jevnt i intervallet og tyngdepunktet bør ligge omkring 15 (1000000/65536) instanser av hver verdi.



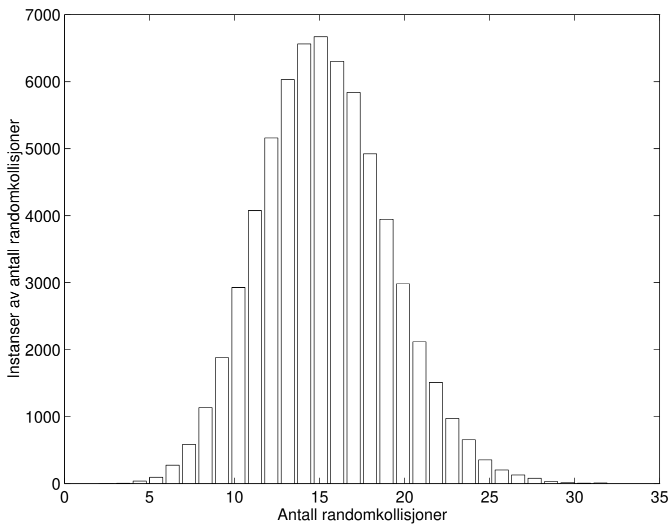
Figur 4.2 *Histogram av 1000000 randomverdier, generert av (II).*

Som en ser av figuren er verdiene tilnærmet binomisk fordelt omkring 15, så tatt i betraktning av at antallet (1000000) ikke er så alt for stort virker dette bra. Jeg kjørte også tilsvarende målinger for (I) og resultatet ble som følger:



Figur 4.3 *Bargraf av 1000000 randomverdier, generert av (I).*

Fordelingen for (I) virker også bra, som en ser av figuren er ikke variasjonen i forhold til (II) særlig stor. Jeg velger også å ta med et histogram av 1000000 randomverdier fordelt på 65536 verdier, generert av (I).

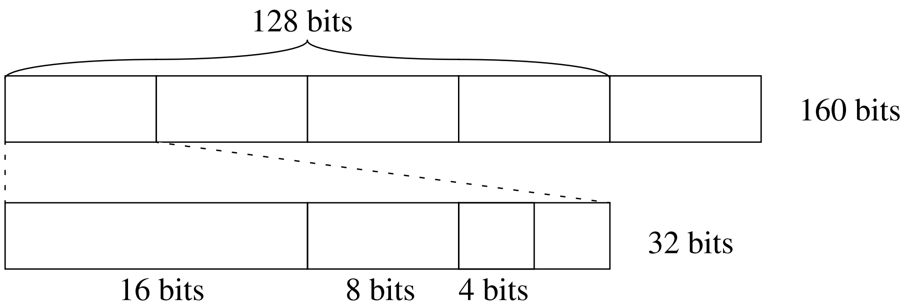


Figur 4.4 *Histogram av 1000000 randomverdier, generert av (I).*

I testprogrammet ble forsøkene kjørt sekvensielt, med flere forsøk for hver programkjøring. Årsaken til dette var å kunne få best mulig sammenligningningsgrunnlag, da (I) ellers ville ha kommet ut med det samme resultatet hver gang (p.g.a. fast startverdi i sekvensen). Utifra forsøkene var det umulig å si om den ene var vesentlig bedre enn den andre og det ble heller ikke dratt noen konklusjoner i så måte. Til mitt bruk valgte jeg å benytte (I), da jeg har muligheten til å kjøre den samme pseudorandom sekvensen flere ganger. Et annet moment er at (I) er vesentlig raskere enn (II) og det vil ha betydning når jeg etterhvert kjører tyngre jobber.

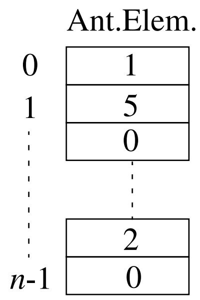
4.3 Praktisk utføring

For å kunne utføre statistiske forsøk på hashfunksjoner må hashverdiene deles opp slik at det er praktisk mulig å utføre forsøk på dem. Da SHA har en 160 bits og MD5 en 128 bits hashverdi, vil det være praktisk umulig å telle opp hvordan verdiene fordeler deg. En eventuell tabell ville måtte ha en størrelse på 2^{160} henholdsvis 2^{128}, en tabell i den størrelsesorden vil ikke kunne implementeres i praksis. I testprogrammene (shsdrvr.c og md5drvr.c) har jeg laget testrutiner som kan håndtere deler av hashverdien. Jeg har begrenset meg til 3 forskjellige størrelser på henholdsvis 4, 8 og 16 bits. Når verdiene er såpass små er det mulig å utføre forsøk på dem. Tabellene har en størrelse på henholdsvis 16, 256 og 65536 elementer.



Figur 4.5 *Deling av hashverdiene*

For det meste ble forsøkene gjort ved å benytte 8 bits, som svarer til en tabellstørrelse på 256 rader. Forsøkene ble utført ved at en benyttet pseudorandomgenererte tekststrenger av en gitt lengde, denne lengden ble gitt utifra hvor lang tid en ville benytte på forsøkene. De fleste forsøkene ble utført med en strenglengde på 196 karakterer. Resultatet, hashverdien, når disse strengene ble hashet av SHA eller MD5 tilsvarer en posisjon i tabellen. For hver gang en tekststreng blir hashet blir det lagt til en for den respektive posisjonen i tabellen.



Figur 4.6 *Oppbygging av testtabell*

I figuren ovenfor betyr dette at det for eksempel er fem(5) meldinger som har hashet til hashverdien 1. Det må legges til at dette ikke er en kollisjon i ordets rette forstand, da det kun er i en enkelt del av hashverdien at en tekststreng hasher til den samme verdien. I likhet med pseudorandom verdier er det også et ønske at hashverdiene fordeler jeg jevnt i et gitt intervall, i dette tilfellet 16, 256 eller 65536 forskjellige verdier. Resultatene av disse forsøkene vil bli vist i neste kapittel. Jeg vil først vise hvordan jeg har laget testalgoritmene som jeg benytter for å gjøre de statistiske forsøkene. Jeg valgte å bygge videre på implementasjoner som tidligere er lagt ut på Internett, slik at en ikke trenger å lage hele algoritmene fra bunnen av. Jeg valgte å bygge videre på RSA Data Security sin implementasjon av MD5 og Peter C. Gutmanns implementasjon av SHA. Begge implementasjonene har omtrent den samme strukturelle oppbygning og dette gjør det lettere å implementere testalgoritmene. Den strukturelle oppbygging for dem begge er som følger:

```
Init      /* Initiering av konstanter */
Update    /* Den iterative lesing, hashing */
Final     /* Padding og avsluttende hashing */
Print     /* Utskrift av hashverdien */
```

Jeg antar at Peter C. Gutmann har benyttet RSAs implementasjon av MD4 som et grunnlag for sin implementasjon av SHA, da både navnevalg og struktur har mye til felles. Det er da også at faktum at både MD5 og SHA bygger på MD4. Begge implementasjonene benytter samme struktur for sine .h og .c filer.

```
shs.h/md5.h /* Definisjoner, globale variabler */
shs.c/md5.c /* Hovedalgoritmer, Init, Update osv. */
shsdrvr.c/md5drvr.c /* Testrutiner, preprosessering */
```

I mine forsøk har jeg stort sett forandret på shsdrvr.c og md5drvr.c filene da det er en fordel å beholde mest mulig av strukturen internt i programmene intakt. Jeg skal nå presentere nærmere algoritmene jeg benytter i mine forsøk. Jeg presenterer kun algoritmene som benyttes i SHS, da algoritmene for MD5 er i prinsippet identiske. Til tross for at denne oppgaven er skrevet på norsk, har jeg valgt å la programmene være på engelsk, både av hensyn til at programmene opprinnelig er på engelsk og at programmenes universelle bruksområder kan ivaretas. Dette er definisjonen av tabellen som benyttes i forsøkene.

```
LONG Table[TableSize];
```

TableSize blir definert i .h filen for de respektive programmene, da størrelsen er avhengig av hvor stor del av hashverdien en skal teste på. Det er derfor best å definere både størrelsen, antall bit, en skal teste på samtidig som en definerer tabellstørrelsen. I tillegg til tabellen må en også ha en algoritme som genererer en tilfeldig streng av bokstaver. Bakgrunnen for denne og valg av pseudorandomgenerator finner en i det foregående kapittelet.

```
char *RandomString (int stringlength, char string[])
{
LONG tall, tallmod, i;
char bokstaver[26] =
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't','u', 'v',
'w', 'x', 'y', 'z'};

       for (i = 0; i < stringlength; i++)
       {
```

```
            tall = random();
            tallmod = (tall%26);
            string[i] = bokstaver[tallmod];
        }
return (string);
}
```

Her benyttes random()-funksjonen for å lage de pseudorandom tekststrengene som senere blir benyttet til hashing, jfr Kapittel 4.2, algoritme (I). En må også ha muligheten for å plukke ut den aktuelle delen av hashverdien, dette gjøres i den følgende algoritmen.

```
LONG shsPartOfMessageDigest(SHS_INFO *shsInfo,
  LONG MessagDigestPartIndex, LONG Index, LONG TableSize)
{
LONG temp, Part, maxindex;
  {
  temp = (Info->digest[MessageDigestPartIndex]);
  if (TableSize == 65536) /* Del opp i deler a 16 bits*/
      {
      if (Index == 1) temp = (temp >> 16);
      }
  if (TableSize == 256) /* Del opp i deler a 8 bits*/
      {
      if (Index == 0) temp = (temp >> 24);
      if (Index == 1) temp = (temp >> 16);
      if (Index == 2) temp = (temp >> 8);
      }
  if (TableSize == 16) /* Del opp i deler a 4 bits */
      {
      if (Index == 0) temp = (temp >> 28);
      if (Index == 1) temp = (temp >> 24);
      if (Index == 2) temp = (temp >> 20);
      if (Index == 3) temp = (temp >> 16);
      if (Index == 4) temp = (temp >> 12);
      if (Index == 5) temp = (temp >> 8);
      if (Index == 6) temp = (temp >> 4);
      }
  Part = (temp & (TableSize - 1)); /* Aktuell del */
  }
return (Part);
}
```

Denne algoritmen plukker ut den aktuelle delen av hashverdien.
MessageDigestPartIndex forteller hvilke av de 5(4) 32 bits delene av SHA (MD5) som en vil teste på. Index er hvilken del av MessageDigest en vil bruke, denne er bestemt av hvor mange bits en plukker ut og hvor i MessageDigest den er plassert.

```
Table[PartOfMessageDigest
        (Info, messageindex, index, TableSize)]++;
```

Denne programlinjen øker tabellverdien i den aktuelle tabellposisjonen som tilsvarer hashverdien. For å få skrevet ut statistikken må dette skrives ut på skjermen.

```
void shsPrintStatistics (const LONG InnTabell[], LONG table-
size, LONG MDIndex, LONG Index)
{
LONG i,j,tempmax,tabindex;
LONG collisions[1000];

   tempmax = 0;
   for (j = 0; j < 1000; j++) /* Reset statistics table */
        collisions[j] = 0;

   printf("\nPartOfMessageDigest: %d Index: %d\n", MDIndex,
        Index);
   for (i = 0; i < tablesize; i++)/* Search through the whole
                                        table */
   {
        for (j = 0; j < 1000; j++)/* Maximum 1000 collisions */
        {
        if (InnTabell[i] == j)
             (collisions[j])++;
        }
   }
        printf("There are %ld empty hashvalues\n" ,
                collisions[0]);
        printf("There are %ld single hashvalues\n",
                collisions[1]);
        for (j = 2; j < 1000; j++)/* Maximum 1000 collisions */
        {
        if ((collisions[j]) > 0)
             printf("There are %ld hashvalues with %ld
                collision(s)\n", collisions[j],j);
        }
}
```

Denne algoritmen skriver ut antallet kollisjoner og samt en oversikt over fordelingen for kontrollformål. For å kunne lage bargrafer og histogrammer, må en kunne legge inn grafene på fil, slik at MATLAB kan lese inn matriseverdiene fra denne filen.

```
void shsPrintStatisticsToFile(const LONG InnTabell[], LONG
tablesize, LONG MDIndex, LONG Index)
{
LONG i,j,collisions[1000];
```

```
char  file_name[15];
FILE *outfileptr;

   sprintf(file_name, "TestFile_shs_%d_%d%d.txt",HashPartSize,
            MDIndex, Index);
   outfileptr = fopen(file_name, "w");
/* fprintf(outfileptr, "PartOfMessageDigest: %d Index:
                                       %d\n",MDIndex, Index);*/
   for(i = 0; i < tablesize; i++)
        /* Search through the whole table */
   {
/* fprintf(outfileptr, "%x", i); If you want the hashvalue
                                                on the file */
   fprintf(outfileptr, "%d\n", InnTabell[i]);
        /* Utskrift til fil, for matlab */
   }
fclose(outfileptr);
}
```

Denne algoritmen skriver ut resultatet til fil på et format som MATLAB godtar, det er også mulig å gjøre enkelter små forandringer slik at både indeksene og hashverdiene med tilhørende antall kollisjoner blir skrevet ut, en kan da ikke lese verdiene inn i MATLAB.

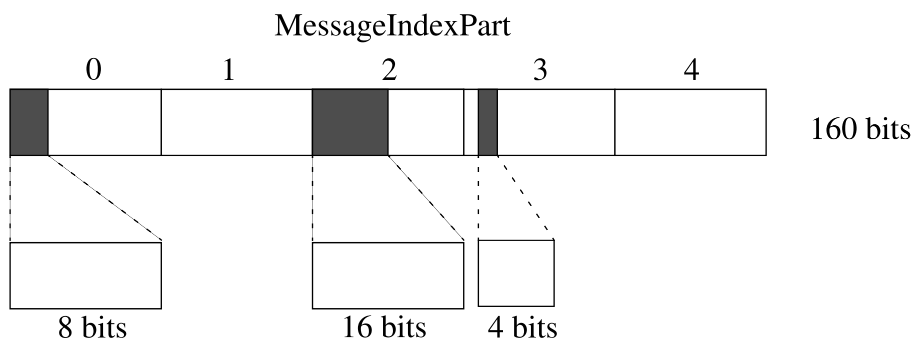
```
local void Random(LONG TableSize)
{
INFO Info;
LONG i, j, messageindex, maxindex, index,
            NumberOfRandomNumber, StringLength;
LONG MessageDigestPart;
char RandomString[StringLength];
QUEUE* Table[TableSize];

   printf("Enter # RandomNumber> ");
   scanf("%d", &NumberOfRandomNumber);
   for (messageindex = 0; messageindex < 5; messageindex++){
        for (index = 0; index < maxindex; index++){
            make_table(Table, TableSize);
            for (i = 0; i < NumberOfRandomNumber; i++)
                 {
                 RandomString(StringSize, RandomString);
                 Init (Info);
                 Update (&Info, (unsigned char *)
                 RandomString, StringSize);
                 Final (&Info);

                 Table(RandomString,PartOfMessageDigest
                           (&Info, (&shsInfo, messageindex,
                                index, TableSize), Table);
```

```
                    PrintStatistics(Table, TableSize,
                         messageindex, index);
                    PrintStatisticsToFile(Table,TableSize,
                         messageindex,index);
                    printf("\n");
                    free_table(Table, TableSize);
                    }
               }
          }
}
```

Denne algoritmen er hovedalgoritmen som blir kalt opp direkte fra *main* og som igjen kaller opp de tidligere nevnte algoritmene.

4.4 Resultater SHA

Det viser seg i praksis at alle de forskjellige delene av hashverdien har tilnærmet lik intern fordeling. Da det ikke er videre oversiktlig å liste opp statistiske tester for alle de forskjellige delene av hashverdien har jeg valgt å plukke ut enkelte av dem. Disse vil jeg presentere i de statistiske forsøkene. For SHA har jeg plukket ut tre forskjellige deler av hashverdien på 4, 8 og 16 bits. Jeg viser ellers til figur 4.7.

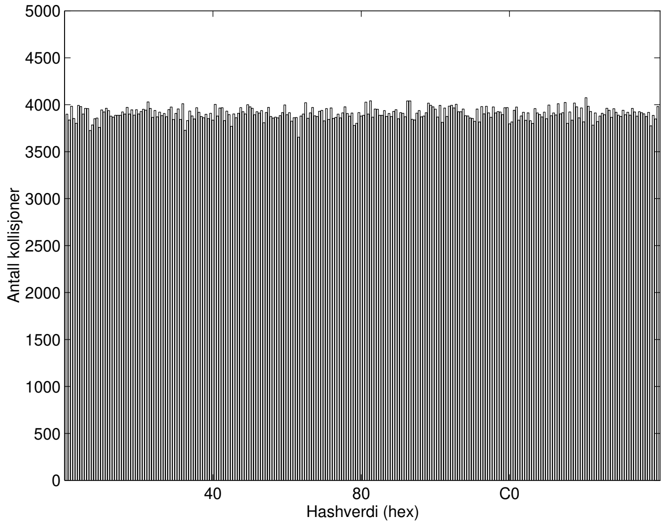


Figur 4.7 *Deler av SHA*

Dette tilsvarer MessageIndexPart 0 med index 0 for 8 bits varianten, MessageIndexPart 2 med index 0 for 16 bits varianten og MessageIndexPart 3 med index 1 for 4 bits varianten. Av hensyn til mengden av utdata, velger jeg å kjøre 1000000 pseudorandomstrenger på hver av forsøkene.

4.4.1 MessageIndexPart 0, Index 0, 8 bits

Grafisk framstilles dette på samme måte som med pseudorandom tallene.



Figur 4.8 *Kollisjoner, 1000000 pseudorandom strenger, 256 hashverdier.*

Det virker som om fordelingen er bra, den bør være jevnt fordelt og med tyngdepunktet omkring 3906 (1000000/256) kollisjoner. Som en stikkprøve har jeg også skrevet ut en hashverdi med tilhørende randomtekst. For å få til dette har jeg benyttet *queue* algoritmene fra hash16 programmet.

```
Hashverdi:11
Textstring:
   cryjdugcmkkkixje
Textstring:
   olwlykrkfhrbmkfh
Textstring:
   cjveqycqotcpxkvq
```

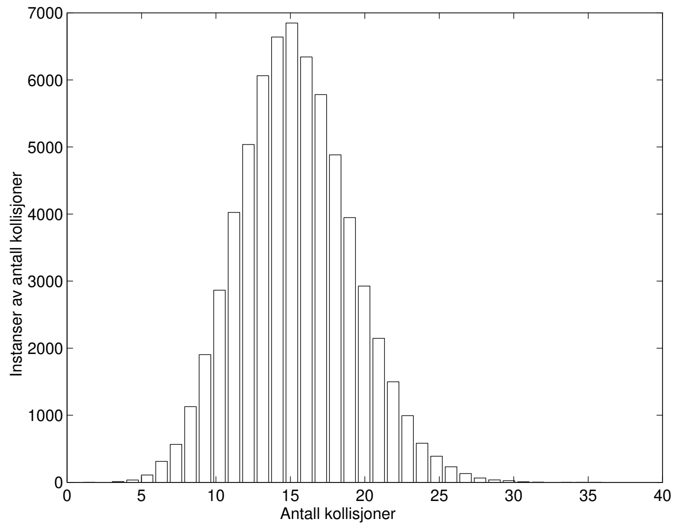
Ved å kjøre disse separat i shs, som strenger -s, får vi som følger:

```
lomvi:~/hovedfag/shs/alpha> shs -scryjdugcmkkkixje
1134351e 5877aef6 d972f54f 97b6bf9c 83be4b6e

lomvi:~/hovedfag/shs/alpha> shs -solwlykrkfhrbmkfh
1165c9f6 da942970 51921c82 e35225fa ca32df3b

lomvi:~/hovedfag/shs/alpha> shs -scjveqycqotcpxkvq
11a4a823 2169335b 734d561c 8b7954be 46bd8237
```

Vi ser da at disse verdiene stemmer overens og at de to mest signifikante hexadesimale hashverdiene for alle 3 er lik 0x11.

4.4.2 MessageIndexPart 2, Index 0, 16bits

Grafisk framstilles dette på samme måte som med pseudorandom tallene, som et histogram over fordelingene på de forskjellige verdiene.

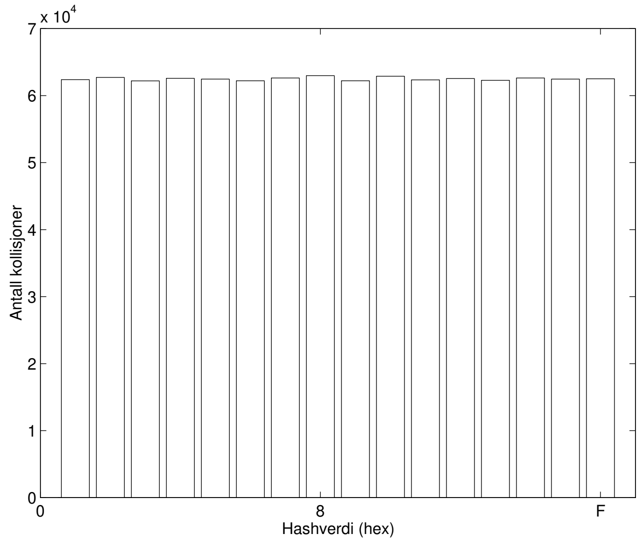


Figur 4.9 *Kollisjoner, 1000000 pseudorandom strenger, 65536 hashverdier.*

Også i denne figuren er fordelingen tilnærmet binomisk.

4.4.3 MessageIndexPart 3, Index 1, 4 bits

Dette blir vist ved hjelp av et stolpediagram.

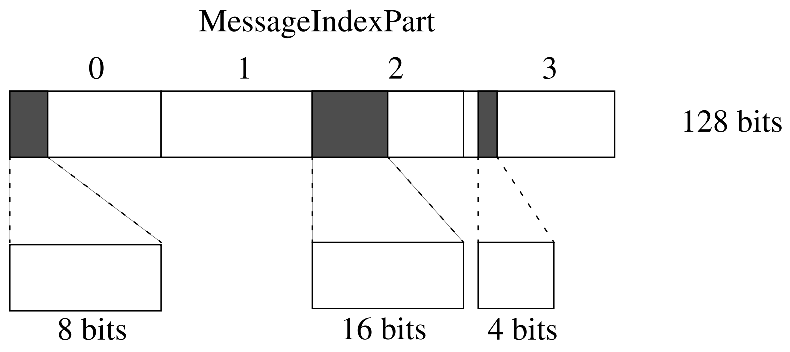


Figur 4.10 *Kollisjoner, 1000000 pseudorandom strenger, 16 hashverdier.*

En ser at verdiene fordeler seg jevnt i intervallet med en tilnærmet jevn fordeling mellom de forskjellige verdiene.

4.5 Resultater MD5

Da det ikke er videre oversiktlig å liste opp statistiske tester for alle de forskjellige delene av hashverdien og de i praksis viser seg å være relativt like har jeg valgt å plukke ut enkelte av delene som jeg presenterer de statistiske forsøkene. For MD5 har jeg plukket ut tre forskjellige deler av hashverdien på 4, 8 og 16 bit. Jeg viser ellers til Figur 4.11.

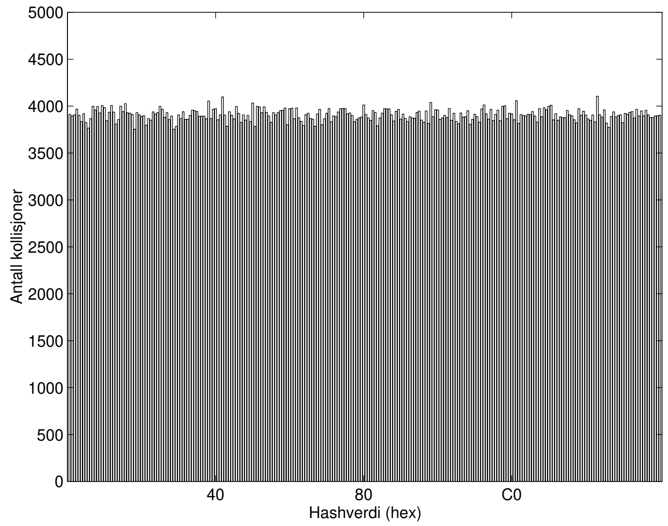


Figur 4.11 *Deler av MD5*

Dette tilsvarer MessageIndexPart 0 med index 0 for 8 bits varianten, MessageIndexPart 2 med index 0 for 16 bits varianten og MessageIndexPart 3 med index 1 for 4 bits varianten.

4.5.1 MessageIndexPart 0, Index 0, 8 bits

Grafisk framstilles dette på samme måte som med pseudorandom tallene.

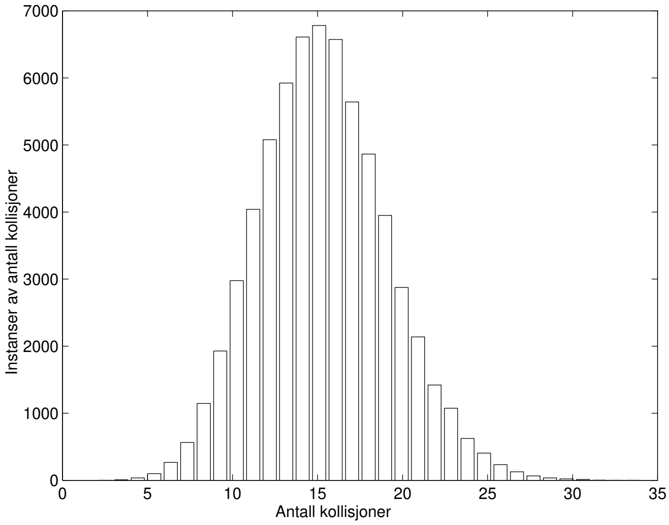


Figur 4.12 *Kollisjoner, 1000000 pseudorandom strenger, 256 forskjellige hashverdier.*

Det virker som om fordelingen er bra, den bør være jevnt fordelt og med tyngdepunktet omkring 3906 (1000000/256) kollisjoner.

4.5.2 MessageIndexPart 2, Index 0, 16bits

Grafisk framstilles fordelingern ved hjelp av et histogram.

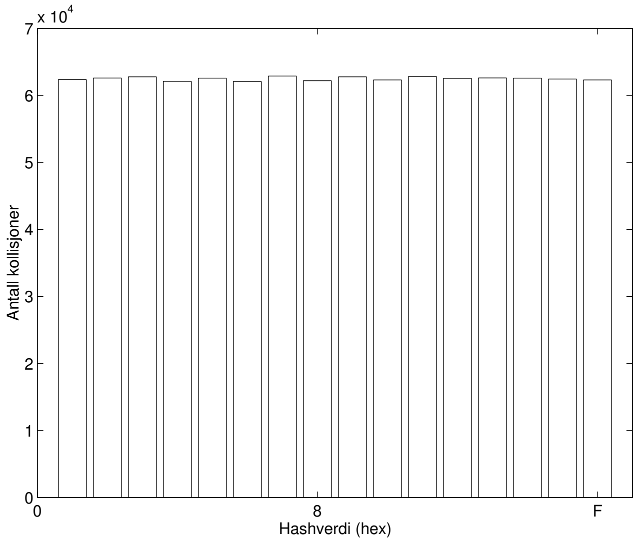


Figur 4.13 *1000000 pseudorandom strenger, 65536 forskjellige hashverdier.*

Som en ser av histogrammet er det som forventet en tilnærmet binomisk fordeling.

4.5.3 MessageIndexPart 3, Index 1, 4 bits

Grafisk framstilles ved hjelp av et bardiagram på samme måte som med pseudorandom tallene.



Figur 4.14 *Kollisjoner, 1000000 pseudorandom strenger, 16 hashverdier.*

Som en ser av figuren er er fordelingen rimelig god, hashverdiene fordeler seg jevn i intervallet.

4.6 Konklusjon

Det kan nok innvendes at disse statistiske forsøkene er for lite omfattende for å dra noen entydige konklusjoner. Det virker likevel som om hashfunksjonene fordeler verdiene jevnt, i en tilnærmet random fordeling.

Referanser

[Akl83] S. G. Akl. *On the Security of Compressed Encoding*, CRYPTO '83. Plenum Publishing Corporation 1983

[BaH90] B. Banieqbal, S. Hilditch, *The Random Matrix Hashing Algorithm*. Technical Report DCS Univ. of Manchester. 1990.

[Ber92] Tom Berson, *Differential Cryptoanalysis Mod 2^{32} with applications to MD5*, EUROCRYPT '92, Springer Verlag 1992.

[BFS91] Th. Beth, M. Frisch, G.J. Simmons (Editorer), *Public-Key Cryptography: State of the art and Future Directions*. E.I.S.S. Workshop, Juli 1991.

[BGG92] T. Beritaud, H. Gilbert, M. Girault, *FFT hashing is not collision-free*, Advances in Cryptology, EUROCRYPT '92, 1992.

[Bih92] Eli Biham. *On the Applicability of Differential Cryptoanalysis to Hash Functions*, lecture at Workshop on Cryptographic Hash Functions, Mars 1992.

[BiS91] Eli Biham, Adi Shamir. *Differential Cryptoanalysis of DES-like Cryptosystems*. EUROCRYPT '91. 1991

[BiS92] Eli Biham, Adi Shamir. *Differential Cryptoanalysis of Snefru, Khafre, REDOC II, LOKI and Lucifer.* Advances in Cryptology, CRYPTO '91, Springer Verlag, 1991.

[BoB92] B. den Boer, A. Bosselaers. *An Attack on the Last Two Rounds of MD4*, CRYPTO '91, Springer Verlag 1992.

[BoB93] B. den Boer, A. Bosselaers. *Collitions for the Compression Function of MD5*, EUROCRYPT '93, Springer Verlag 1993

[CaP91] P. Camion, J. Patarin, *The Knapsack hash Function proposed at Crypto '89 can be broken*. Abstracts of EUROCRYPT '91, 1991.

[Chi88] Lindsay Childs. *A Concrete Introduction to Higher Algebra*, 4 utgave 1988.

[Cop85] D. Coppersmith. *Another Birthday attack*. CRYPTO 85, Lecture Notes. Springer-Verlag 1985.

[Cop89] D. Coppersmith. *Analysis of ISO/CCITT, X.509. Annex D*, 1989.

[Dob95a] Hans Dobbertin, *Collitions for the last two rounds of the RIPEMD compress function*, rump session EUROCRYPT '95, ikke publisert.

[Dob95b] Hans Dobbertin, *MD4 is not collisionfree*, rump session EUROCRYPT '95, 1995, ikke publisert.

[DBG91] J. Daemen, A. Bosselaers,R. Govaerts, J.Vandewalle, *Collitions for Schnorr's FFT-hash*, ASIACRYPT '91, 1991.

[DGV91] J. Daemen, R. Govaerts, J. Vanderwalle. *A Framework for the Design of One-way Function Based on Celluar Automaton*, Abstracts of ASIACRYPT '91, 1991.

[Dam88] I. B. Damgård. *Collision free hash functions and public key signature schemes*. EUROCRYPT '87, Springer-Verlag 1988.

[Dam89] I. B. Damgård. *A Design Principle for Hash Funktions*. CRYPTO 89, Lecture Notes Springer-Verlag 1989.

[DaP80] D. W. Davies, W. L. Price. *Digital Signaturees based on Public Key Cryptosystems*. Proceedings of the Fifth International Conference on Computer Communications, 1980

[DaP84] D. W. Davies, W. L. Price. *Digital Signature - an update*. Proceedings of the Seventh International Conference on Computer Communications, 1984

[Dav83] D. W. Davies. *Applying the RSA Digital Signature to Electronic Mail*, 1983.

[Den84] D. E. Denning. *Digital Signatures with RSA and Other Public-Key Cryptosystems*. Com. of the ACM, 1984.

[DiH76] W. Diffie, M. E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory, No. 6, 1976.

[ElG85] T. ElGamal. *A public key cryptosystem and a signature scheme based on discrete logarithms*. IEEE Transactions on Information Theory, 1985.

[Fei73] H. Feistel, *Cryptography and computer privacy.* Scientific American, Vol. 228, No. 5, 1973.

[FIPS46] National Bureau of Standards, U.S.Department of Commerce, NBS PUB 46 *Data Encryption Standard*, Januar 1977.

[FIPS180] National Institute of Standards and Technology, Department of Commerce. FIPS PUB 180, *Secure Hash Standard*, 31 Mai 1994.

[FIPS186] National Institute of Standards and Technology, Department of Commerce. FIPS PUB 186, *Digital Signature Standard*, 19 Mai 1994.

[Gir87] M. Girault. *Hash-Functions Using Modulo-N Operations*. In Advances in Cryptology - EUROCRYPT '87. Lecture Notes in Computer Science. Springer-Verlag, 1987.

[GMR88] S. Goldwasser, S. Micali, R. L. Rivest. *A Digital Signature Scheme Secure against Adaptive Chosen-Message Attacks*. SIAM Journal on Computing, 1988.

[Har84] S. Harari, *Nonlinear Non Commutative Functions for Data Integrity*, EUROCRYPT 84, Lecture Notes in Computer Science, Springer-Verlag, 1984.

[HKF93] Jeri R. Hanly, Elliot B. Koffman, Frank L. Friedman, *Problem Solving and Program Design in C*, Addison Wesley 1993.

[Håk93] Espen Håkonsen, *Digitale Signaturer*, Seminaroppgave for Revisorstudiet, Bedriftsøkonomisk Institutt 1993

[ImN89] R. Impagliazzo, M. Naor, *Efficient cryptographic schemes provably as secure as subset sum*, 30th IEEE Symposium on Foundations of Computer Science, 1989.

[JMM85] R. R. Jueneman, S. M. Matyas og C. H. Meyer. *Message Authentication*. IEEE Communication Magazine, 1985.

[Jue86] R. R. Jueneman. *A High Speed Manipulation Detection Code*. CRYPTO '86. Lecture Notes in Computer Science. Springer-Verlag, 1986.

[Jue87] R. R. Jueneman. *Electronic Document Authentication* IEEE Network Magazine, 1987.

[LuR88] M. Luby, C. Racoff, *How to Construct Pseudorandom Permutations from Pseudorandom Functions*, SIAM Journal on Computing, 1988.

[Mer79] R. C. Merkle. *Secrecy, Autentication, and Public Key Systems*. UMI Research Press, 1979.

[Mer89] R. C. Merkle. *One Way Hash Functions and DES*. CRYPTO '89, Lecture Notes in Computer Science. Springer-Verlag, 1989.

[Mer90] R. C. Merkle. *A fast software one-way hash function*, Journal of Cryptology, Vol. 3, No. 1, 1990.

[MeM82] C. H. Meyer, S. M. Matyas. *Cryptography: a New Dimension in Data Security.* Wiley & Sons, 1982.

[MIO89] S. Miyaguchi, M. Iwata og K. Otha. *New 128 bits Hash Function*. Proceedings of International Joint workshop on Computer and Communication, 1989.

[MOI90] S. Miyaguchi, K. Otha og M. Iwata. *Confirmations that Some Hash Functions Are Not Collition Free*. EUROCRYPT '90, Lecture Notes in Computer Science, Springer-Verlag, 1990

[Mjø93] Stig Frode Mjølsnes, *Digitale signaturer*, rapport SINTEF Delab.

[PiS93] Josef Pieprzyk, Babak Sadeghiyan, *Design of Hashing Algorithms,* Lecture Notes in Computer Science, Springer-Verlag 1993.

[PGV92] B. Preneel, R. Govaerts, J. Vanderwalle. *Cryptographically Secure Hash Functions: an Overview*, 1992.

[Pre93] Bart Preneel, "*Analysis and Design of Cryptographic Hash Functions*" Ph.D. thesis Katholieke Universiteit Leuven, Jan 1993.

[QuD90] J. -J. Quisquater, J. -P. Delescaille. *How easy is collition search? Application to DES*, Advances in Cryptology, EUROCRYPT '89, Springer-Verlag, 1990.

[Rab78] M. O. Rabin, *Digitalized Signatures*. Foundations of Secure Computation, Academic Press, 1978.

[Rab79] M. O. Rabin, *Digital Signatures and Public-Key Functions as Intractabe as Factorization*, MIT Laboratory of Computer Sciience, Technical Report, MIT/LCS/TR-212, Januar 1979.

[Riv91] R.L.Rivest, The MD4 message digest algorithm, CRYPTO '90, Springer Verlag 1991.

[Riv92] R.L.Rivest, The MD5 message digest algorithm, rump session CRYPTO '91, RFC 1321, IAB, IPTF, April 1992.

[RIPE93] B. den Boer, J.P.Boly m.fl. *RIPE integrity primitives Part I og II*. RIPE report 1993.

[RSA78] R. Rivest, A. Shamir og L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, Februar 1978.

[Sal90] Arto Salomaa, *Public-Key Cryptography*. EATCS Monographs on teoretical Computer Science, Springer Verlag 1990.

[Sch91] C. P. Schnorr, *FFT-Hashing, An Efficient Cryptographic Hash Function*, CRYPTO '91, 1991.

[Sch92] C. P. Schnorr. FFT-Hash II. *Efficient Cryptographic Hashing*. Abstracts of EUROCRYPT '92, 1992.

[Sch93] Bruce Schneier, *Applied Cryptography, Protocols, Algorithms and Source Code in C*. Wiley 1993.

[SeP89] J. Seberry og J. Pieprzyk. Cryptography, *An Introduction to Computer Security*, Prentice Hall, 1989.

[Sha49] C. E. Shannon, *Communication Theory of Secrecy Systems*, The University of Illinois Press, Urbana, 1949.

[Vau92] S. Vaudenay, *FFT-hash II is not yet collition free*, Advances i Cryptology, CRYPTO '92. 1992.

[Wil80] H. C. Williams, *A Modification of the RSA Public-Key Encryption Procedure*, IEEE Transactions on Information Theory, November 1980.

[Win83] R. S. Winternitz. *Producing a One-Way Hash Function from DES*. Advances in Cryptology - CRYPTO '83, Plenum Publishing Corporation, 1983.

[Wol86] S. Wolfram, *Teory and Applications of Cellular Automata*. World Scientific, 1986.

[Zim94] Philip Zimmerman, *PGP User's Guide*, volume I og II. Public Domain 1994.

[ZPS92] Y. Zheng, J. Pieprzyk, J. Seberry, *HAVAL One-Way Hashing Algorithm with Variable Length of Output*. AUSCRYPT '92, Springer-Verlag 1993.

Appendiks

A DES

DES og hashfunksjoner har flere fellestrekk, de har begge som formål å for andre på klarteksten slik at resultatet (kryptoteksten/hashverdien) ser ut som et tilfeldig tall, dette for å gjøre det vanskelig å knekke denne. For DES blir resultatet litt større enn utgangspunktet da også nøkkelen skal være en del av kryptoteksten. Pseudokoden for DES er som følger:

1. Gitt en klartekst M, en bitstreng M_0 blir konstruert ved å permutere bits i M i henhold til en initiell permuntasjon IP. Vi skriver $M_0 = IP(M) = L_0R_0$, hvor L_0 er de første 32 bits i M_0 og R_0 er de siste 32 bits i M_0.

2. 16 iterasjoner av en gitt funksjon f, som defineres senere. Vi regner ut L_iR_i, $1 \leq i \leq 16$ i henhold til følgende regel:

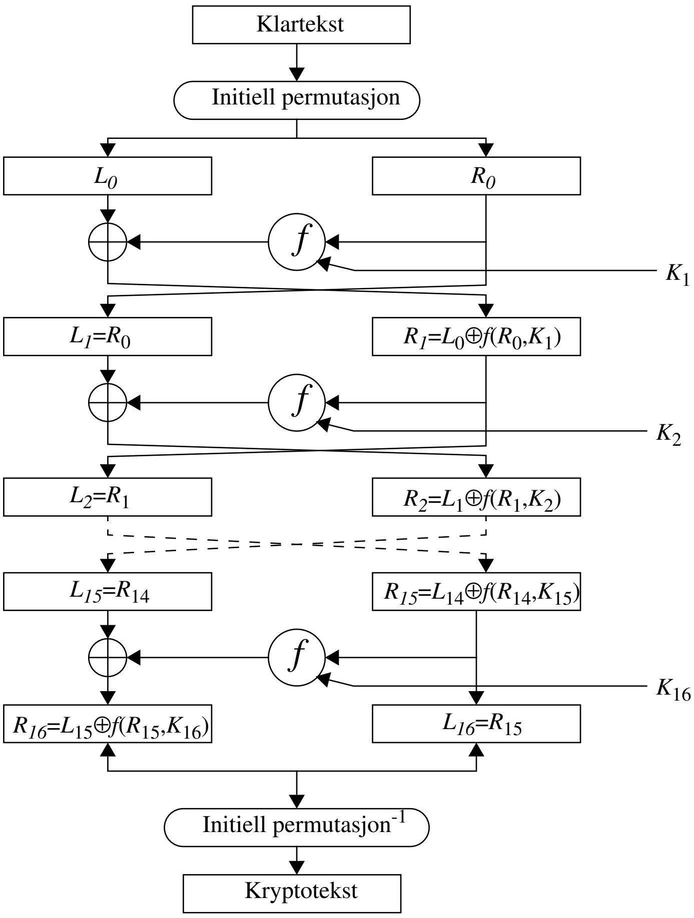
   $L_i = R_{i-1}$
   $R_i = L_i + f(R_{i-1}, K_i)$

   K_1, K_2,...,K_{16} er hver bitstrenger av lengde 48 bits som blir regnet ut som en funksjon av nøkkelen K.

3. Legg til den inverse permutasjonen IP^{-1} til bitstrengen $R_{16}L_{16}$ for å få kryptoteksten C. Det vil si at $C = IP^{-1}(R_{16}L_{16})$

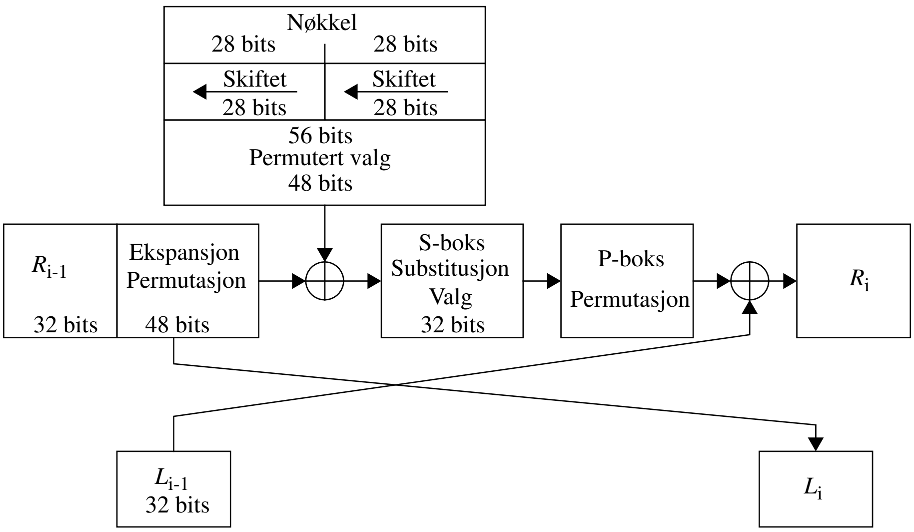
f: Ekspansjon
Bitaddisjon
Deling i blokker
Transformering v.h.a. S-bokser

For å dekryptere gjøres det samme, bare i motsatt rekkefølge. DES og hashfunksjoner har flere fellestrekk, de har begge som formål å forandre på klarteksten slik at resultatet (kryptoteksten/hashverdien) ser ut som et tilfeldig tall, dette for å gjøre det vanskelig å knekke denne. For DES blir resultatet litt større enn utgangspunktet da også nøkkelen skal være en del av kryptoteksten. For å ytterligere klargjøre virkemåten til DES, presenteres Figur A.1 og Figur A.2.



Figur A.1 *DES [Sch93]*

For å klargjøre dette ytteligere viser jeg også en figur som viser en runde av DES.



Figur A.2 *DES, en runde [Sch93]*

B RSA

For å generere de to nøklene, velger en to store primtall p og q, en regner ut produktet:

$$n = p \times q$$

så velges tilfeldig krypteringsnøkkelen e slik at e og $(p - 1) \times (q - 1)$ er relativt primiske. En benytter så Euclids algoritme for å regne ut dekrypteringsnøkkelen d slik at:

$$e \times d = 1 \pmod{(p - 1) \times (q - 1)}$$

Tallene e og n er den offentlige nøkkelen, mens d er den private. For å kryptere en melding, må en først dele opp meldingen slik at den er representert ved tall og i passende blokker slik at meldingen har en unik representasjon (mod n). Det vil si at hvis p og q er tall i størrelsesorden 100 siffer, vil n ha en størrelsesorden like under 200 siffer, og hver meldingsblokk m_j vil ha en størrelse på under 200 siffer. Den krypterte meldingen c vil bli dannet av blokker c_j på omtrent samme lengde. Krypteringen foregår som følger:

$$c_i = m_i^e \pmod{n}$$

for å dekryptere en melding bruker en hver enkelt blokk c_i og regner ut:

$$m_i = c_i^d \pmod{n}$$

siden

$$c_i^d = (m_i^e)^d = m_i^{ed} = m_i^{k(p-1)(q-1)+1} = m_i \times m_i^{k(p-1)(q-1)} = m_i$$

alt bergenet modulo n vil meldingen bli entydig dekryptert. Jeg vil illustrer det med et lite eksempel. Jeg velger $p = 47$ og $q = 71$, da er

$$n = p \times q = 3337$$

Krypteringsnøkkelen e har ingen faktorer til felles med:

$$(p - 1) \times (q - 1) = 46 \times 70 = 3220$$

Velger e tilfeldig til å være 79. I så tilfelle:

$$d = 79^{-1} \pmod{3220} = 1019$$

Dette tallet ble beregnet ved å benytte Euclids algoritme. Publiser e og n, behold d hemmelig. For å kryptere meldingen:

$$m = 6882326879666683$$

deler en først opp meldingen i små blokker, i dette tilfellet i blokker på 3 siffer. Vi får seks blokker m_i som ser ut som følger:

$$m_1 = 688$$
$$m_2 = 232$$
$$m_3 = 687$$

m_4 = 966
m_5 = 668
m_6 = 3

Den første blokken krypteres til:

$688^{79} \pmod{3337} = 1570 = c_1$

Ved å benytte den samme framgangsmåten på de andre blokken genereres en kryptert melding:

c = 1570 2756 2714 2276 2423 158

Dekryptering av meldingen gjøres ved å benytte dekrypteringsnøkkelen, tallet 1019, slik at:

$1570^{1019} \pmod{3337} = 688 = m_1$

Resten av den krypterte meldingen kan dekrypteres på samme måte.

B.1 Sikkerhet/hastighet RSA

Jeg vil ikke komme nærmere inn på sikkerheten og hastigheten på RSA, men bare nevne at kryptering med RSA er ca. 100-1000 ganger mindre effektivt enn for eksempel DES. RSA-129, det vil si RSA med en lengde (*n*) på 129 siffer er pr. i dag faktorisert. RSA-129 ble faktorisert ved hjelp av "quadratic sieve" faktorerings metode. Faktoreringen ble gjort mulig ved hjelp av over 200 frivillige fra mere enn 20 land. En stor del av arbeidet ble utført ved Institutt for Informatikk i Bergen på MasPar MP-1. De konkrete resultatene er som følger:

```
   RSA-129 =
11438162575788886766923577997614661201021829672124236256256 1\
84293570693524573389783059712356395870505898907514759929002 6\
879543541
          =
34905295108476509491478496199038981334177646384933878439908 2\
0577 * 32769132993266709549961988190834461413177642967992 94\
2539798288533
```

Den krypterte meldingen var som følger:

```
96869613754622061477140922254355882905759991124574319874695 1\
20930816298225145708356931476622883989628013391990551829945 1\
57815154
```

Dette tallet kom fra en RSA kryptering av en hemmelig melding ved å bruke den offentlige nøkkelen 9007. Når en dekrypterer med den private nøkkelen

```
10669861436857802444286877132892015478070990663393786280122 6\
22449663106312591177447087334016859746230655396854451327710 9\
053606095
```

som blir

```
200805001301070903002315180419000118050019172105011309190800\
151919090618010705
```

Ved å bruke 01=A, 02=B, ...,26=Z og 00 et mellomrom får en den dekrypterte meldingen

```
THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE
```

I dag er det vanlig å benytte RSA varianter som har en n på over 200 siffer det finnes også varianter som benytter 308 siffer (1024 bits), tall i denne størrelsesorden er det ikke trolig at en vil klare å faktorisere i den nærmeste framtid. Det finnes også varianter av RSA som beviselig er like vanskelig å knekke som det er å faktorisere n [Rab79, Wil90].

C Praktisk bruk av PGP

Det forutsettes her at leseren kjenner til begrepen offentlig nøkkel kryptering, digitale signaturer og kan bruke mailverktøy. Det forutsettes videre at PGP, versjon 2.6.2i er innstallert og tilgjengelig for deg som bruker. Pr. i dag vil PGP være installert ved Institutt for Informatikk, Universitetet i Bergen og ved flere Universitet og høyskoler i Norge.

C.1 Kommandoer i PGP 2.6.2i

For å kryptere en klartekstfil med mottagerens offentlige nøkkel:

```
pgp -e tekstfil brukerid
```

For å signere en klartekstfil med din hemmelige nøkkel:

```
pgp -s tekstfil [-u din_brukerid]
```

For å signere en klartekstfil med din hemmelige nøkkel, og deretter kryptere med mottagerens offentlige nøkkel:

```
pgp -es tekstfil brukerid [-u din_brukerid]
```

For å kryptere en klartekstfil ved hjelp av konvensjonell kryptering:

```
pgp -c tekstfil
```

For å dekryptere en kryptert fil, eller kontrollere signaturen på en signert fil:

```
pgp chiffertekstfil [-o klartekstfil]
```

For å kryptere en melding til flere mottagere:

```
pgp -e tekstfil brukerid1 brukerid2 brukerid3
```

C.1.1 Kommandoer for nøkkelhåndtering

For å opprette ditt eget unike offentlige/hemmelige nøkkelpar:

```
pgp -kg
```

For å legge en offentlig eller hemmelig nøkkel til din offentlige eller hemmelige nøkkelring:

```
pgp -ka nøkkelfil [nøkkelring]
```

For å hente ut (kopiere) en nøkkel fra din offentlige eller hemmelige nøkkelring:

```
        pgp -kx brukerid nøkkelfil [nøkkelring]
eller:  pgp -kxa brukerid nøkkelfil [nøkkelring]
```

For å vise innholdet i din offentlige nøkkelring:

```
pgp -kv[v] [brukerid] [nøkkelring]
```

For å vise “fingeravtrykket” til en offentlig nøkkel, for lettere å verifisere den over telefon med eieren:

```
pgp -kvc [brukerid] [nøkkelring]
```

For å vise innholdet og kontrollere signaturene i din offentlige nøkkelring:

```
pgp -kc [brukerid] [nøkkelring]
```

For å endre bruker-ID eller passordfrase for din hemmelige nøkkel:

```
pgp -ke brukerid [nøkkelring]
```

For å endre tillitsparametrene til en offentlig nøkkel:

```
pgp -ke brukerid [nøkkelring]
```

For å fjerne en nøkkel eller en bruker-ID fra din offentlige nøkkelring:

```
pgp -kr brukerid [nøkkelring]
```

For å signere (attestere) en annens offentlige nøkkel i din offentlige nøkkelring:

```
pgp -ks brukerid [-u din_brukerid] [nøkkelring]
```

For å fjerne signaturer fra en bruker-ID i en nøkkelring:

```
pgp -krs brukerid [nøkkelring]
```

For å inndra din egen nøkkel og utstede en tilbaketrekningssertifikat:

```
pgp -kd din_brukerid
```

For å deaktivere eller reaktivere en offentlig nøkkel i din offentlige nøkkelring:

```
pgp -kd brukerid
```

C.1.2 Spesielle kommandoer

For å dekryptere en melding og bevare signaturen intakt:

```
pgp -d chiffertekstfil
```

For å lage en signatur som er adskilt fra dokumentet:

```
pgp -sb tekstfil [-u din_brukerid]
```

For å fjerne signaturen fra en signert melding:

```
pgp -b chiffertekstfil
```

C.1.3 Kombinerte opsjoner

For å beskytte en chiffertekstfil med et 7-bits ASCII transportpanser, legg til -a opsjonen ved kryptering, signering, eller kopiering av en nøkkel:

```
       pgp -sea tekstfil brukerid
eller: pgp -kxa brukerid nøkkelfil [nøkkelring]
```

For å slette og overskive den opprinnelige klartekstfilen, legg til -w (wipe) opsjonen ved kryptering eller signering av en melding:

```
pgp -sew klartekstfil brukerid
```

For å angi at en klartekstfil inneholder ASCII-text (i motsetning til binære data) og skal konverteres til mottagerens lokale tegnsett, bruk -t (text) opsjonen i tillegg til de andre parametrene:

```
pgp -seat klartekstfil brukerid
```

For å vise den dekrypterte klarteksten på skjermen din (tilsvarende Unix-kommandoen “more”) uten å lagre den til en fil, bruk -m (more) opsjonen ved dekryptering:

```
pgp -m chiffertekstfil
```

For å angi at mottagerens dekrypterte klartekst KUN skal vises på skjermen og ikke lagres til disk, angi -m (more) opsjonen ved kryptering:

```
pgp -steam klartekstfil brukerid
```

For å lagre klartekstfilen med dens originale filnavn, angi -p opsjonen ved dekryptering:

```
pgp -p chiffertekstfil
```

For å omdirigere input og output slik som under Unix, ved å lese fra standard inn og skrive til standard ut, bruk -f opsjonen:

```
pgp -feast brukerid <innfil >utfil
```

D Kompleksitet

Kompleksitetsteori er viktig når en skal vurdere kryptografiske algoritmer. Sikkerheten i et kryptosystem baserer seg ofte på kompleksitetsteoretiske vurderinger og begrepet “praktisk umulig” som blir benyttet i denne oppgaven er egentlig et begrep som baserer seg på kompleksitetsteori.
Noe av det som er viktig når vi vurderer hastigheten på algoritmer er at vi som regel ser bort fra konstanter og konsentrerer oss om hvordan algoritmen oppfører seg når størrelsen på innverdiene går mot uendelig. for eksempe hvis innverdien er en tabell av størrelse n og algoritmen består av $100n$ iterasjoner så ingnorer en konstanten 100 og sier at kjøretiden er omtrent n. Hvis antallet iterasjoner er $2n^2 + 50$ ignorerer en konstantene 2 og 50 og sier at kjøretiden er omtrent n^2. Siden n^2 er større enn n regner en at den siste algoritmen er mindre effektiv, selv om n for eksempel er 4 og den første algoritmen behøver 400 iterasjoner, mens den andre bare behøver 82 iterasjoner. Denne tilnærmingen er dog realistisk dersom n er stor nok. I praksis viser det seg at denne måten å vurdere en algoritme er tilstrekkelig for det store flertall av vurderingene.
I kryptografien er det spesiellt viktig å skille mellom algotritmer som bruker lineær tid, det vil si at store input kan behandles effektivt og algoritmer som bruker eksponensiell tid, det vil si at store input er praktisk umulig å behandle i rimelig tid. Det kan også legges til at både tidskompleksitet som vi har omtalt her og minnekompleksiteten, som beregner forbruk av minne mens programmet kjører, benytter seg av O notasjonen som blir definert i neste delkapittel.

D.1 *O* Notasjonen

Vi har allerede nevnt at vi vil ignorere konstanter nå vi skal vurdere algoritmer, og vi skal nå innføre en notasjon som blir benyttet for dette formål. Vi sier at en funksjon $g(n)$ er $O(f(n))$ for en annen funksjon $f(n)$, hvis det eksisterer konstanter c og N, slik at for alle $n \geq N$, har vi $g(n) \leq cf(n)$. Med andre ord for store nok n, er funksjonen $g(n)$ inne mere enn en konstant ganger funksjonen $f(n)$. Funksjonen $g(n)$ kan til og med være mindre enn $cf(n)$, O notasjonen setter kun en øvre grense. For eksempel, $5n^2 + 15 = O(n^2)$ siden $5n^2 + 15\ 6n^2$ for $n \geq 4$ på samme tid er $5n^2 + 15 = O(n^3)$, siden $5n^2 + 15 \leq n^3$ for alle $n \geq 6$.

E Tallteori

Jeg gir her en kort innføring i noen få tallteoretiske begreper som blir brukt i denne oppgaven. For en grundigere innføring henvises til [Chi88].
Et heltall a deler et annet heltall b, symbolsk $a \mid b$, dersom $b = da$ holder for et eller annet heltall d. Da er a kalt en divisor eller faktor av b. La a være et heltall større enn 1. Da er a et primtall dersom de eneste positive divisorene er 1 og a, ellers er tallet sammensatt. Ethvert heltall kan representeres som et produkt av primtall. Det essensielle faktum, utifra et kryptologisk synspunk, er at det ikke finnes noen effektiv måte å faktorisere store tall i primtallsfaktorer.

Programlistinger

A XOR-Hash

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
int      i;
FILE     *innfil;
int      ch, temp = 0;    /* Bokstavene i innfil og
                                        hashverdien */

  if (argc!=2)
       {
       printf("Du glemte å skrive inn filnavnet\n");
       exit(1);
       }

  if((innfil = fopen(argv[1], "r")) == NULL)
       {
       printf("Kan ikke åpne '%s' for hashing\n",argv[1]);
       exit(1);
       }

  printf("Teksten på filen:\n");
  for (ch = getc(innfil); ch != EOF; ch = getc(innfil))
       {
       temp = temp ^ ch;
       printf("%c", ch);
       }
/* Hashverdien blir skrevet ut som en bokstav */
  printf("\n"); /* Linjeskift */
  printf("Hashverdi: %c",temp);
  printf("\n"); /* Linjeskift */
  fclose(innfil); /* Lukker filen */
}
```

B Hash-16

Programlistingen av hash16 fra kapittel 2.10. Hash16drvr.c er hovedprogrammet som bruker de andre modulene. Queue.h og queue.c er implementasjonene av køen som brukes til angrepet på hash16. For rutinene som kjører statistiske tester på hashfunksjonene viser jeg til programlistingene av MD5 og SHA.

B.1 Hash16drvr.c

```
/*
  En forenklet versjon av SHSdrvr.c.
  Liten hashverdi (16 bits), kun for å illustrere.
  ANSI C - August 1995, Jan Anders Solvik.
*/
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "hash16.h"
#include "queue.c"

/*
  Skriver hashverdien fra bufferet i hash16info som
  4 hexadsimale siffer.
*/
void hash16Print (hash16_INFO *hash16Info)
{
  printf ("%04x\n", hash16Info->digest [0]);
}

/*
  Regner ut hashverdien for strengen inString. Skriver ut
  hashverdien, et mellomrom, strengen og linjeskift.
*/
void hash16String (char *inString)
{
  hash16_INFO hash16Info;
  unsigned int len = strlen (inString);
  hash16Init (&hash16Info);
  hash16Update (&hash16Info,
       (unsigned char *) inString, len);
  hash16Final (&hash16Info);
  hash16Print (&hash16Info);
  printf (" \"%s\"\n", inString);
}
```

```
/*
   Regner ut hashverdien for en spesifisert fil. Skriver ut
   hashverdien, et mellomrom, filnavnet og linjeskift.
*/
void hash16Fil (char *filename)
{
FILE *inFile = fopen (filename, "rb");
hash16_INFO hash16Info;
int bytes;
unsigned char data [1024];

   if (inFile == NULL) {
        printf ("%s kan ikke åpnes.\n", filename);
        return;
        }
   hash16Init (&hash16Info);
   while ((bytes = fread (data, 1, 1024, inFile)) != 0)
             hash16Update (&hash16Info, data, bytes);
   hash16Final (&hash16Info);
   hash16Print (&hash16Info);
   printf (" %s\n", filename);
   fclose (inFile);
}

/*
   Skriver hashverdien fra stdin til stdout
*/
void hash16Filter (void)
{
hash16_INFO hash16Info;
int bytes;
unsigned char data [hash16_BLOKKSTORRELSE];

   hash16Init (&hash16Info);
   while ((bytes=fread(data, 1,
        hash16_BLOKKSTORRELSE, stdin))!= 0)
             hash16Update (&hash16Info, data, bytes);
   hash16Final (&hash16Info);
   hash16Print (&hash16Info);
   printf ("\n");
}

/*
   Lag tabellen før kjøring
*/
QUEUE *make_table(QUEUE *InnTabell[], LONG TabellStorrelse)
{
LONG i;

   for (i =0; i < TabellStorrelse; i++)
        InnTabell[i] = qcreate();
   return(*InnTabell);
}
```

```
/*
  Frigi tabellen etter kjøring
*/
void free_table(QUEUE *InnTabell[], LONG TabellStorrelse)
{
LONG i;
QUEUE* localqueue;

  for (i = 0; i < TabellStorrelse; i++)
        {
        localqueue = InnTabell[i];
        while (localqueue->size > 0 )
             free_node(localqueue);
        free(localqueue);
        }
}

/*
  Spesifikasjon av tabell for kollisjoner/angrep
*/
void hash16Table(char message[], LONG hashvalue,
        QUEUE* table[])
{
LONG Size;
ELM* e; /* ELM er et element i en struktur av char[size] */
LONG i, j;

  Size = strlen(message);
  if ((e = (ELM*) malloc (sizeof(ELM)))==NULL)
        {
        printf("Feil i minneallokasjonen - exit");
        exit(1);
        }
  for (i = 0;  i < Size; i++)
        {
        e->textstring[i] = message[i];
        }
  if (!qpush(table[hashvalue], e))
        printf("Kan ikke legge inn i tabell");
}
```

```
/*
  Lager tekst strenger for kollisjons testing.
  "JEG HAR TILGODE XXXXX KRONER"
*/
char *hash16String1(char string[],LONG i)
{
LONG tall,tallmod;
char tmpstring[5];

  strcpy(string, "JEG HAR TILGODE ");
  sprintf(tmpstring, "%d", i);
  strcat(string, tmpstring);
  strcat(string, " KRONER");

return (string);
}

/*
  Lager tekst strenger for kollisjons testing.
  "JEG SKYLDER XXXXX KRONER"
*/
char *hash16String2(char string[],LONG i)
{
LONG tall,tallmod;
char tmpstring[5];

  strcpy(string, "JEG SKYLDER ");
  sprintf(tmpstring, "%d", i);
  strcat(string, tmpstring);
  strcat(string, " KRONER");

return (string);
}

/*
  Lager random tall
*/
LONG hash16RandomNumber(LONG range)
{
LONG tempnumber, Number;

  tempnumber = random();
  Number = (tempnumber%range);

return (Number);
}
```

```
/*
   Lager en 16 bits hashverdi fra en 32 bits
*/
LONG hash16GetDigest(hash16_INFO *hash16Info)
{
LONG Part, temp;

   temp = hash16Info->digest[0];
   Part = (temp & 65535);

return (Part);
}

/*
   Utfører et angrep på hash16
   Hasher 20000 setninger, finner kollisjoner
*/
void hash16Angrep(void)
{
hash16_INFO hash16Info;
LONG i, j;
char Setning[StrengLengde];/* StrengLengde fra queue.h */
QUEUE* Table[TabellStorrelse]; /* TabellStorrelse fra
hash16.h */
LONG temphash;

   make_table(Table, TabellStorrelse);
   for (i = 0; i < 10000; i++)
   {
        hash16String1(Setning, i);
        hash16Init (&hash16Info);
        hash16Update (&hash16Info,
             (unsigned char *) Setning, StrengLengde);
        hash16Final (&hash16Info);
        hash16Table(Setning,
                  hash16GetDigest(&hash16Info, Table);
   }

   for (i = 0; i < 10000; i++)
   {
        hash16String2(Setning, i);
        hash16Init (&hash16Info);
        hash16Update (&hash16Info,
             (unsigned char *) Setning, StrengLengde);
        hash16Final (&hash16Info);
        hash16Table(Setning,
                  hash16GetDigest(&hash16Info, Table);
   }

   hash16PrintStatistics(Table, TabellStorrelse);

   free_table(Table, TabellStorrelse);
}
```

```
void main (int argc, char **argv)
{
int i;

/* For hver kommando linje:
   filename      - Skriver hashverdi og navn på fil
   -sstring      - Skriver hashverdi og innhold i strengen
   -a            - Utfører et angrep på hash16
   (no args)     - Skriver hashverdien fra stdin til stdout
*/

   if (argc == 1)
         hash16Filter ();
   else
         for (i = 1; i < argc; i++)
               if (argv [i] [0] == '-' && argv [i] [1] == 's')
                     hash16String (argv [i] + 2);
               else if (strcmp (argv [i], "-a") == 0)
                     hash16Angrep ();
               else
                     hash16Fil (argv [i]);
}
```

B.2 Hash16.c

```
/*
   Hoveddelen av hashingen, iterasjonsfunksjonen
*/
#include <string.h>
#include "hash16.h"

/* Hash16 funksjonen */

#define f1(x,y)    ( x | ( ~y ))/* Rundene 0-19 */
#define f2(x,y)    ( x ^ y )/* Rundene 20-39 */

/* Hash16 konstantene */

#define K1  0x5A82L /* Rundene 0-19 */
#define K2  0x6ED9L /* Rundene 20-39 */

/* hash16 initielle verdier */

#define h0init0x6745L
#define h1init0xEFCDL
#define h2init0x2301L

/* 32-bit rotering */

#define S(n,X)((X << n) | (X >> (32 - n)))
```

```
/* De to hash16 sub-rundene */

#define subRound1(count)    \
  { \
       temp = S (5, A) + f1 (B, C) + W [count] + K1; \
       C = S (7, B); \
       B = A; \
       A = temp; \
  }

#define subRound2(count)    \
  { \
       temp = S (11, B) + f2 (A, C) + W [count] + K2; \
       C = S (10, A); \
       B = A; \
       A = temp; \
  }

/* Buffere av 2 32-bit words */

LONG h0, h1, h2;
LONG A, B, C;

/*
  Initialisere hash16 verdiene
*/
void hash16Init (hash16_INFO *hash16Info)
{
  /* Initialisere h variablene */
  hash16Info->digest [0] = (LONG)h0init;
  hash16Info->digest [1] = (LONG)h1init;
  hash16Info->digest [2] = (LONG)h2init;

  /* Initialisere bit tellingen*/
  hash16Info->countLo = hash16Info->countHi = (LONG)0;
}

/*
  Transform med runder
*/
void hash16Transform (hash16_INFO *hash16Info)
{
LONG W [80], temp;
int i;

/* Steg A. Kopier data inn i det lokale bufferet */
  for (i = 0; i < 16; i++)
       W [i] = hash16Info->data [i];

/* Steg B.Setter opp første buffer */
  A = hash16Info->digest [0];
  B = hash16Info->digest [1];
  C = hash16Info->digest [2];
```

```
/* Step C.To forskjellige runder med forskjellige utg.pkt. */
   subRound1(0); subRound1(1); subRound1(2); subRound1(3);
   subRound1(4); subRound1(5); subRound1(6); subRound1(7);
   subRound1(8); subRound1(9); subRound(10); subRound2 (11);
   subRound2(12); subRound2(13); subRound2(14); subRound2(15);
   subRound2(16); subRound2(17); subRound2(18); subRound2(19);

/* Step D.Bygg opp hashverdien */
   hash16Info->digest [0] += A;
   hash16Info->digest [1] += B;
   hash16Info->digest [2] += C;
}

/*
   Oppdaterer hashverdien mens en går igjennom filen/strengen
*/
void hash16Update (hash16_INFO *hash16Info, BYTE *buffer,
                   int count)
{
/* Oppdater bit-telling */
   if ((hash16Info->countLo + ((LONG) count << 3))
         < hash16Info->countLo)
   hash16Info->countHi++;/* Mente fra lav til høy biteller */
   hash16Info->countLo += ((LONG) count << 3);
   hash16Info->countHi += ((LONG) count >> 29);

/* Kjør data i hash16_BLOKKSTORRELSE blokker */
   while (count >= hash16_BLOKKSTORRELSE) {
         memcpy (hash16Info->data, buffer,
                 hash16_BLOKKSTORRELSE);
         hash16Transform (hash16Info);
         buffer += hash16_BLOKKSTORRELSE;
         count -= hash16_BLOKKSTORRELSE;
   }
   memcpy (hash16Info->data, buffer, count); /* Resten */
}

/*
   Siste kjøring, med padding
*/
void hash16Final (hash16_INFO *hash16Info)
{
int count;
LONG   lowBitcount = hash16Info->countLo,
       highBitcount = hash16Info->countHi;
LONG temphash;

   /* Regn ut antallet bytes mod 64 */
   count = (int) ((hash16Info->countLo >> 3) & 0x3F);

   /* Sett de n første char i paddingen til 0x08 */
   ((BYTE *) hash16Info->data) [count++] = 0x80;

   /* Padding av blokkene til64 bytes */
   memset ((BYTE *) hash16Info->data + count, 0, 64 - count);
   hash16Transform (hash16Info);
```

```
  /* XORing av A, B og C */
  temphash = (hash16Info->digest[0] ^ hash16Info->digest[1]
              ^ hash16Info->digest[2]);
  hash16Info->digest[0] = (temphash & 65535);
  hash16Info->digest[1] = 0;
  hash16Info->digest[2] = 0;
}
```

B.3 Hash16.h

```
typedef unsigned char BYTE;
typedef unsigned int LONG; /* 64 bits  ALPHA*/
/* typedef unsigned long LONG; 32 bits processors*/

/* Hash16 blokk størrelse og meldings størrelse , i bytes */

#define hash16_BLOKKSTORRELSE64
#define TabellStorrelse 65536

/* Strukturen for midlertidig lagring av hashverdien */

typedef struct {
  LONG digest [1];/* Message digest */
  LONG countLo, countHi;/* 64-bit bit count */
  LONG data [16];/* hash16 data buffer */
} hash16_INFO;
```

B.4 Queue.c

```
#include <stdlib.h>
#include <stdio.h>
#include "queue.h"

/*
  Operasjoner på en kø av strenger
*/
unsigned long qpush ( QUEUE* q, ELM* e )
{
QNODE* qn;

  qn = (QNODE*) malloc ( sizeof (QNODE) );
  if ( qn == NULL ) return ( 0 );
  qn->e = e;
  qn->next = NULL;
  if ( q->size == 0 ) {
      q->tail = qn;
      q->head = qn;
      }
  else {
      q->tail->next = qn;
      q->tail = qn;
  q->size++;
return ( 1 );
}
```

```
ELM* qpop ( QUEUE* q )
{
QNODE *tempnode;

   ELM* e = q->head->e;
   tempnode = q->head;
   q->head   = q->head->next;
   free(tempnode);
   q->size--;
   if ( q->size == 0 ) q->tail = NULL;
   return ( e );
}

void free_node(QUEUE *q)
{
QNODE *tempnode;
ELM *tempe;

   ELM* e = q->head->e;
   tempe = e;
   tempnode = q->head;
   q->head   = q->head->next;
   free(tempe);
   free(tempnode);
   q->size--;
   if ( q->size == 0 ) q->tail = NULL;
}


unsigned long qempty ( QUEUE* q )
{
   if ( q->size == 0 ) return ( 1 );
   else return ( 0 );
}

QUEUE* qcreate ()
{
   QUEUE* q;
   q = (QUEUE*) malloc ( sizeof (QUEUE) );
   q->size = 0;
   q->head = NULL;
   q->tail = NULL;
   return ( q );
}

static void elm_print ( QNODE* qn )
{
   unsigned long i;
   printf ( "Tekststreng: \n");
   for (i = 0; i < StrengLengde; i++)
        printf ( "%c", qn->e->tekststreng[i] );
   printf ( "\n");
}
```

```
unsigned long ant_element(QUEUE* q)
{
  return(q->size);
}

void qprint ( QUEUE* q )
{
QNODE* qn;

  printf ( "qsize = %d\n", q->size );
  for ( qn = q->head; qn != NULL; qn = qn->next )
      elm_print ( qn );
  printf ( "----------------------------------\n" );
}
```

B.5 Queue.h

```
/*
  Definisjonene for queue.c
*/

#define StrengLengde 32   /* Maks lengde på tekststrengene */

struct elm {              /* Innholdet i hvert element*/
  char tekststreng[StrengLengde];
  };

typedef struct elm ELM;

struct queuenode {        /* Definisjonen av hvert element */
  ELM* e;
  struct queuenode* next;
  };

typedef struct queuenode QNODE;

struct queuetype {        /* Def. av køen med pekere */
  unsigned long  size;
  QNODE* head;
  QNODE* tail;
  };

typedef struct queuetype QUEUE;

unsigned long qpush   (QUEUE* q, ELM *e);
ELM* qpop ( QUEUE* q );
unsigned long qempty (QUEUE* q);
QUEUE* qcreate( );
void qprint(QUEUE* q);
```

C SHA

Kildekoden for sha, inkludert mine tillegg for statistisk testing.

C.1 shsdrvr.c

```
/*
  NIST proposed Secure Hash Standard.
  Written 2 September 1992, Peter C. Gutmann.
  This implementation placed in the public domain.

  Added several testroutines spring 1995, Jan Anders Solvik.
  Adapted to ANSI C - August 1995, Jan Anders Solvik.
*/
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <math.h>
#include <time.h>
#include "shs.h"

/* Prototypes of the local functions */
local void shsPrint OF((SHS_INFO *shsInfo));
local void shsTimeTrial OF((void));
local void shsString OF((char *inString));
local void shsFile OF((char *filename));
local void shsFilter OF((void));
local void shsTestSuite OF((void));
local void shsRandom OF((void));
      void main OF((int argc, char **argv));

/*
  Prints message digest buffer in shsInfo as 40 hexadecimal
  digits. Order is from low-order byte to high-order byte of
  digest. Each byte is printed with high-order hexadecimal
  digit first.
*/
local void shsPrint (SHS_INFO *shsInfo)
{
  int i;

  for (i = 0; i < 5; i++)
       {
       printf ("%08x", shsInfo->digest [i]);
       printf (" ");
       }
}

/* size of test block */
#define TEST_BLOCK_SIZE 1000

/* number of blocks to process */
#define TEST_BLOCKS 100000
```

```
/* number of test bytes = TEST_BLOCK_SIZE * TEST_BLOCKS */
local long TEST_BYTES =
       (long) TEST_BLOCK_SIZE * (long) TEST_BLOCKS;

/*
  A time trial routine, to measure the speed of SHA.
  Measures wall time required to digest
  TEST_BLOCKS * TEST_BLOCK_SIZE characters.
*/
local void shsTimeTrial (void)
{
SHS_INFO shsInfo;
time_t endTime, startTime;
local unsigned char data [TEST_BLOCK_SIZE];
unsigned int i;

  /* initialize test data */
  for (i = 0; i < TEST_BLOCK_SIZE; i++)
       data [i] = (unsigned char) (i & 0xFF);

  /* start timer */
  printf ("SHA time trial. Processing %ld characters...\n",
           TEST_BYTES);
  time (&startTime);

  /* digest data in TEST_BLOCK_SIZE byte blocks */
  shsInit (&shsInfo);
  for (i = TEST_BLOCKS; i > 0; i--)
       shsUpdate (&shsInfo, data, TEST_BLOCK_SIZE);
  shsFinal (&shsInfo);

  /* stop timer, get time difference */
  time (&endTime);
  shsPrint (&shsInfo);
  printf (" is digest of test input.\nSeconds to process
           test input: %ld\n", long) (endTime - startTime));
  printf ("Characters processed per second: %ld\n",
           TEST_BYTES / (endTime - startTime));
}

/*
  Computes the message digest for string inString. Prints out
  message digest, a space, the string (in quotes) and a
  carriage return.
*/
local void shsString (char *inString)
{
SHS_INFO shsInfo;

  unsigned int len = strlen (inString);
  shsInit (&shsInfo);
  shsUpdate (&shsInfo, (unsigned char *) inString, len);
  shsFinal (&shsInfo);
  shsPrint (&shsInfo);
  printf (" \"%s\"\n", inString);
}
```

```
/*
  Computes the message digest for a specified file.
  Prints out message digest, a space, the file name,
  and a carriage return.
*/
local void shsFile (char *filename)
{
FILE *inFile = fopen (filename, "rb");
SHS_INFO shsInfo;
int bytes;
local unsigned char data [1024];

if (inFile == NULL) {
  printf ("%s can't be opened.\n", filename);
  return;
  }

  shsInit (&shsInfo);
  while ((bytes = fread (data, 1, 1024, inFile)) != 0)
           shsUpdate (&shsInfo, data, bytes);
  shsFinal (&shsInfo);
  shsPrint (&shsInfo);
  printf (" %s\n", filename);
  fclose (inFile);
}

/*
  Writes the message digest of the data from stdin
  onto stdout, followed by a carriage return.
*/

local void shsFilter (void)
{
SHS_INFO shsInfo;
int bytes;
local unsigned char data [SHS_BLOCKSIZE];

  shsInit (&shsInfo);
  while ((bytes = fread (data, 1, SHS_BLOCKSIZE, stdin))!= 0)
       shsUpdate (&shsInfo, data, bytes);
  shsFinal (&shsInfo);
  shsPrint (&shsInfo);
  printf ("\n");
}
```

```
/*
  Runs a standard suite of test data.
*/
local void shsTestSuite (void)
{
  printf ("SHA test suite results:\n");
  shsString ("");
  shsString ("a");
  shsString ("abc");
  shsString ("message digest");
  shsString ("abcdefghijklmnopqrstuvwxyz");
  shsString ("ABCDEFGHIJKLMNOPQRSTUVWXYZ
                abcdefghijklmnopqrstuvwxyz0123456789");
  shsString ("12345678901234567890123456789012345678901\
                12345678901234567890123456789012345678901");

  /* Contents of file foo are "abc" */
  shsFile ("foo");

}
/*
  Prints statistical information on the screen
*/
void shsPrintStatistics(const LONG InnTabell[], LONG table-
size, LONG MDIndex, LONG Index)
{
LONG i,j,tempmax,tabindex;
LONG collisions[1000];

  tempmax = 0;
  for (j = 0; j < 1000; j++)/* Reset statistics table */
       collisions[j] = 0;

  printf("\nPartOfMessageDigest: %d Index: %d\n"
           MDIndex, Index);
  for (i = 0; i < tablesize; i++)/* Search the whole table */
  {
       for (j = 0; j < 1000; j++)/* 0 or 1 collision */
       {
       if (InnTabell[i] == j)
            (collisions[j])++;
       }
  }
  printf("There are %ld empty hashvalues\n" ,collisions[0]);
  printf("There are %ld single hashvalues\n",collisions[1]);
  for (j = 2; j < 1000; j++)/* Maximum 1000 collisions */
       {
       if ((collisions[j]) > 0)
            printf("There are %ld hashvalues with %ld
                      collision(s)\n", collisions[j],j);
       }
}
```

```
/*
  Prints statistical information to file,
  To be used in MATLAB
*/
void shsPrintStatisticsToFile(const LONG InnTabell[], LONG
tablesize, LONG MDIndex, LONG Index)
{
LONG i,j,collisions[1000];
char  file_name[15];
FILE *outfileptr;

  sprintf(file_name, "TestFile_shs_%d_%d%d.txt",HashPartSize,
            MDIndex, Index);

  outfileptr = fopen(file_name, "w");
  for (i = 0; i < tablesize; i++)/* Search the whole table */
       {
       fprintf(outfileptr, "%d\n", InnTabell[i]);
       }
fclose(outfileptr);
}
/*
  A better randomfunction
*/
int uniform ( LONG max )
{
struct timeval tp;
struct timezone tzp;

     gettimeofday ( &tp, &tzp );
     srand((375467*tp.tv_sec) + (235783*tp.tv_usec));
     return ( random() %  max );
}

/*
  Makes random textstrings
*/
char *shsRandomString(int stringlength, char string[])
{
LONG tall,tallmod,i;
char bokstaver[26] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
   'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
   'u', 'v', 'w', 'x','y', 'z'};

  for (i = 0; i < stringlength; i++)
       {
       tall = rand();
       tallmod = (tall%26);
/*     tallmod = uniform(26) A better randomgenerator (II)*/
       string[i] = bokstaver[tallmod];
       }
return (string);
}
```

```
/*
   Makes random numbers
*/
LONG shsRandomNumber(LONG range)
{
LONG tempnumber, Number;

   tempnumber = random(); /* random generator (I) */
   Number = (tempnumber%range);

/* Number = uniform(range); random generator:(II) */

return (Number);
}

/*
   1/10 of Message Digest, 16 bits
   MessageDigestPartIndex 0 - 4
   Index to part of MessageDigestPart
*/
LONG shsPartOfMessageDigest(SHS_INFO *shsInfo,
   LONG MessageDigestPartIndex, LONG Index, LONG tablesize)
{
LONG temp, Part;

   temp = (shsInfo->digest[MessageDigestPartIndex]);
   /* Split into 1/2 of MessageIndexPart 16 bits*/
   if (tablesize == 65536)
        {
        if (Index == 1) temp = (temp >> 16);
             }
   /* Split into 1/4 of MessageIndexPart 8 bits*/
   if (tablesize == 256)
        {
        if (Index == 0) temp = (temp >> 24);
        if (Index == 1) temp = (temp >> 16);
        if (Index == 2) temp = (temp >> 8);
        }
   /* Split into 1/8 of MessageIndexPart, 4 bits */
   if (tablesize == 16)
        {
        if (Index == 0) temp = (temp >> 28);
        if (Index == 1) temp = (temp >> 24);
        if (Index == 2) temp = (temp >> 20);
        if (Index == 3) temp = (temp >> 16);
        if (Index == 4) temp = (temp >> 12);
        if (Index == 5) temp = (temp >> 8);
        if (Index == 6) temp = (temp >> 4);
        }
   Part = (temp & (tablesize - 1)); /* Actual part */
   }
return (Part);
}
```

```
/*
  Main procedure for the statistical testing
*/
local void shsRandom(void)
{
SHS_INFO shsInfo;
LONG i, j, messageindex, maxindex, index,
       NumberOfRandomNumber;
LONG MessageDigestPart;
char RandomString[StringLength];
LONG Table[TableSize];

  printf("Enter # RandomNumber> ");
  scanf("%d", &NumberOfRandomNumber);

/* Converting from TableSize to maxindex */
maxindex = 32 / HashPartSize;
  for (messageindex = 0; messageindex < 5; messageindex++)
  {
  for (index = 0; index < maxindex; index++)
       {
       for (j = 0; j < TableSize; j++) /* Reset table */
            Table[j] = 0;

       for (i = 0; i < NumberOfRandomNumber; i++)
            {
            shsRandomString(StringLength, RandomString);
            shsInit (&shsInfo);
            shsUpdate (&shsInfo,
                 (unsigned char *) RandomString,StringLength);
            shsFinal (&shsInfo);

            Table[shsPartOfMessageDigest(&shsInfo,
                 messageindex, index, TableSize)]++;

            }

       shsPrintStatistics(Table, TableSize,
            messageindex, index);

       shsPrintStatisticsToFile(Table, TableSize,
            messageindex, index);
       printf("\n");
       }
  }
}
```

```
void main (int argc, char **argv)
{
int i;

/*
  For each command line argument in turn:
  filename - prints message digest and name of file
  -sstring - prints message digest and contents of string
  -t       - prints time trial statistics for 10M characters
  -x       - execute a standard suite of test data
  -r       - prints statistics of part of the message digest
                of a random string
  (no args) - writes messages digest of stdin onto stdout
*/

  if (argc == 1)
       shsFilter ();
  else
       for (i = 1; i < argc; i++)
            if (argv [i] [0] == '-' && argv [i] [1] == 's')
                 shsString (argv [i] + 2);
            else if (strcmp (argv [i], "-t") == 0)
                 shsTimeTrial ();
            else if (strcmp (argv [i], "-x") == 0)
                 shsTestSuite ();
            else if (strcmp (argv [i], "-r") == 0)
                 shsRandom ();
            else
                 shsFile (argv [i]);
}
```

C.2 shs.c

```
/*
  NIST Secure Hash Standard.

  Written 2 September 1992, Peter C. Gutmann.
  This implementation placed in the public domain.

  Comments to pgut1@cs.aukuni.ac.nz

  Adapted to ANSI C - 1 August 1995, Jan Anders Solvik.
*/
#include <string.h>
#include "shs.h"

/* The SHS f()-functions */

/* Rounds  0-19 */
#define f1(x,y,z)   ( ( x & y ) | ( ~x & z ) )
/* Rounds 20-39 */
#define f2(x,y,z)   ( x ^ y ^ z )
/* Rounds 40-59 */
#define f3(x,y,z)   ( ( x & y ) | ( x & z ) | ( y & z ) )
/* Rounds 60-79 */
#define f4(x,y,z)   ( x ^ y ^ z )
```

```
/* The SHS Mysterious Constants */

#define K1  0x5A827999L /* Rounds  0-19 */
#define K2  0x6ED9EBA1L /* Rounds 20-39 */
#define K3  0x8F1BBCDCL /* Rounds 40-59 */
#define K4  0xCA62C1D6L /* Rounds 60-79 */

/* SHS initial values */

#define h0init0x67452301L
#define h1init0xEFCDAB89L
#define h2init0x98BADCFEL
#define h3init0x10325476L
#define h4init0xC3D2E1F0L

/* 32-bit rotate - kludged with shifts */

#define S(n,X)((X << n) | (X >> (32 - n)))

/* The initial expanding function */

#define expand(count) W [count] = (1,(W [count - 3]
       ^ W [count - 8] ^ W [count - 14] ^ W [count - 16]))

/* The four SHS sub-rounds */

#define subRound1(count)    \
  { \
       temp = S (5, A) + f1 (B, C, D) + E + W [count] + K1; \
       E = D; \
       D = C; \
       C = S (30, B); \
       B = A; \
       A = temp; \
  }

#define subRound2(count)    \
  { \
       temp = S (5, A) + f2 (B, C, D) + E + W [count] + K2; \
       E = D; \
       D = C; \
       C = S (30, B); \
       B = A; \
       A = temp; \
  }

#define subRound3(count)    \
  { \
       temp = S (5, A) + f3 (B, C, D) + E + W [count] + K3; \
       E = D; \
       D = C; \
       C = S (30, B); \
       B = A; \
       A = temp; \
  }
```

```
#define subRound4(count)    \
  { \
      temp = S (5, A) + f4 (B, C, D) + E + W [count] + K4; \
      E = D; \
      D = C; \
      C = S (30, B); \
      B = A; \
      A = temp; \
  }

/* The two buffers of 5 32-bit words */

LONG h0, h1, h2, h3, h4;
LONG A, B, C, D, E;

local void byteReverse OF((LONG *buffer, int byteCount));
void shsTransform OF((SHS_INFO *shsInfo));

/* Initialize the SHS values */

void shsInit (SHS_INFO *shsInfo)
{
  /* Set the h-vars to their initial values */
  shsInfo->digest [0] = (LONG)h0init;
  shsInfo->digest [1] = (LONG)h1init;
  shsInfo->digest [2] = (LONG)h2init;
  shsInfo->digest [3] = (LONG)h3init;
  shsInfo->digest [4] = (LONG)h4init;

  /* Initialise bit count */
  shsInfo->countLo = shsInfo->countHi = (LONG)0;
}
```

```
/*
  Perform the SHS transformation.  Note that this code, like
  MD5, seems to break some optimizing compilers - it may be
  necessary to split it into sections, eg based on the four
  subrounds
*/
void shsTransform (SHS_INFO *shsInfo)
{
LONG W [80], temp;
int i;

/* Step A.Copy the data buffer into the local work buffer */
  for (i = 0; i < 16; i++)
       W [i] = shsInfo->data [i];
/* Step B.Expand the 16 words into 64 temporary data words */
  expand(16);expand(17);expand(18);expand 19);expand(20);
  expand(21);expand(22);expand(23);expand(24);expand(25);
  expand(26);expand(27);expand(28);expand(29);expand(30);
  expand 31);expand(32);expand(33);expand(34);expand(35);
  expand(36);expand(37);expand(38);expand(39);expand(40);
  expand(41);expand(42);expand(43);expand(44);expand(45);
  expand(46);expand(47);expand(48);expand(49);expand(50);
  expand(51);expand(52);expand(53);expand(54);expand(55);
  expand(56);expand(57);expand(58);expand(59);expand(60);
  expand(61);expand(62);expand(63);expand(64);expand(65);
  expand(66);expand(67);expand(68);expand(69);expand(70);
  expand(71);expand(72);expand(73);expand(74);expand(75);
  expand(76);expand(77);expand(78);expand(79);

/* Step C.Set up first buffer */
  A = shsInfo->digest [0];
  B = shsInfo->digest [1];
  C = shsInfo->digest [2];
  D = shsInfo->digest [3];
  E = shsInfo->digest [4];

/* Step D.Serious mangling, divided into four sub-rounds */
  subRound1(0);subRound1(1);subRound1(2);subRound1(3);
  subRound1(4);subRound1(5);subRound1(6);subRound1(7);
  subRound1(8);subRound1(9);subRound1(10);subRound1(11);
  subRound1(12);subRound1(13);subRound1(14);subRound1(15);
  subRound1(16);subRound1(17);subRound1(18);subRound1(19);

  subRound2(20);subRound2(21);subRound2(22);subRound2(23);
  subRound2(24);subRound2(25);subRound2(26);subRound2(27);
  subRound2(28);subRound2(29);subRound2(30);subRound2(31);
  subRound2(32);subRound2(33);subRound2(34);subRound2(35);
  subRound2(36);subRound2(37);subRound2(38);subRound2(39);

  subRound3(40);subRound3(41);subRound3(42);subRound3(43);
  subRound3(44);subRound3(45);subRound3(46);subRound3(47);
  subRound3(48);subRound3(49);subRound3(50);subRound3(51);
  subRound3(52);subRound3(53);subRound3(54);subRound3(55);
  subRound3(56);subRound3(57);subRound3(58);subRound3 59);

  subRound4(60);subRound4(61);subRound4(62);subRound4(63);
```

```
  subRound4(64);subRound4(65);subRound4(66);subRound4(67);
  subRound4(68);subRound4(69);subRound4(70);subRound4(71);
  subRound4(72);subRound4(73);subRound4(74);subRound4(75);
  subRound4(76);subRound4(77);subRound4(78);subRound4(79);


/* Step E.Build message digest */
  shsInfo->digest [0] += A;
  shsInfo->digest [1] += B;
  shsInfo->digest [2] += C;
  shsInfo->digest [3] += D;
  shsInfo->digest [4] += E;
}

/*
  Find out what the byte order is on this machine.
  Big endian is for machines that place the most significant
  byte first (eg. Sun SPARC). Little endian is for machines
  that place the least significant byte first (eg. VAX).
  We figure out the byte order by stuffing a 2 byte string
  into a short and examining the left byte. ‘@’ = 0x40  and
  ‘P’ = 0x50 If the left byte is the ‘high’ byte, then it is
  ‘big endian’. If the left byte is the ‘low’ byte, then the
  machine is ‘little endian’.

  - Shawn A. Clifford (sac@eng.ufl.edu)

  Several bugs fixed       -- Pat Myrto (pat@rwing.uucp)
*/
local void byteReverse (LONG *buffer, int byteCount)
{
LONG value;
int count;

  if ((*(unsigned short *) (“@P”) >> 8) == ‘@’)
       return;
  byteCount /= sizeof (LONG);
  for (count = 0; count < byteCount; count++) {
       value = (buffer[count] << 16) | (buffer[count] >> 16);
       buffer[count] = ((value & 0xFF00FF00L) >> 8) |
            ((value & 0x00FF00FFL) << 8);
       }
}
```

```
/*
  Update SHS for a block of data.  This code assumes that the
  buffer size is a multiple of SHS_BLOCKSIZE bytes long,
  which makes the code a lot more efficient since it does away
  with the need to handle partial blocks between calls to
  shsUpdate()
*/
void shsUpdate (SHS_INFO *shsInfo, BYTE *buffer, int count)
{
  /* Update bitcount */
  if ((shsInfo->countLo + ((LONG) count << 3))
           < shsInfo->countLo)
  shsInfo->countHi++;/* Carry from low to high bitCount */
  shsInfo->countLo += ((LONG) count << 3);
  shsInfo->countHi += ((LONG) count >> 29);

  /* Process data in SHS_BLOCKSIZE chunks */
  while (count >= SHS_BLOCKSIZE) {
       memcpy (shsInfo->data, buffer, SHS_BLOCKSIZE);
       byteReverse (shsInfo->data, SHS_BLOCKSIZE);
       shsTransform (shsInfo);
       buffer += SHS_BLOCKSIZE;
       count -= SHS_BLOCKSIZE;
       }

/*
  Handle any remaining bytes of data.
  This should only happen once on the final lot of data
*/
  memcpy (shsInfo->data, buffer, count);
}

void shsFinal (SHS_INFO *shsInfo)
{
int count;
LONG   lowBitcount = shsInfo->countLo,
       highBitcount = shsInfo->countHi;

  /* Compute number of bytes mod 64 */
  count = (int) ((shsInfo->countLo >> 3) & 0x3F);

  /* Set the first char of padding to 0x80.This is safe since
       there is always at least one byte free */
  ((BYTE *) shsInfo->data) [count++] = 0x80;

  /* Pad out to 56 mod 64 */
  if (count > 56) {
  /* Two lots of padding:  Pad the first block to 64 bytes */
       memset ((BYTE *) shsInfo->data + count, 0, 64 - count);
       byteReverse (shsInfo->data, SHS_BLOCKSIZE);
       shsTransform (shsInfo);

  /* Now fill the next block with 56 bytes */
       memset (shsInfo->data, 0, 56);
       }
  else
```

```
    /* Pad block to 56 bytes */
        memset ((BYTE *) shsInfo->data + count, 0, 56 - count);
        byteReverse (shsInfo->data, SHS_BLOCKSIZE);

    /* Append length in bits and transform */
    shsInfo->data [14] = highBitcount;
    shsInfo->data [15] = lowBitcount;

    shsTransform (shsInfo);
    byteReverse (shsInfo->data, SHS_DIGESTSIZE);
}
```

C.3 shs.h

```
/*
  NIST proposed Secure Hash Standard.

  Written 2 September 1992, Peter C. Gutmann.
  This implementation placed in the public domain.

  Comments to pgut1@cs.aukuni.ac.nz
*/

/* Useful defines/typedefs */

#ifndef SHS_H
#define SHS_H

typedef unsigned char BYTE;
typedef unsigned int LONG; /* 64 bits  ALPHA*/
/* typedef unsigned long LONG; 32 bits */

/* The SHS block size and message digest sizes, in bytes */

#define SHS_BLOCKSIZE64
#define SHS_DIGESTSIZE20
#define StringLength 300/* Randomstring Length */
#define HashPartSize    8/* 4, 8 or 16 */
#define TableSize256  /* 2^HashPartSize  16, 256 or 65536 */

/* The structure for storing SHS info */

typedef struct {
  LONG digest [5];/* Message digest */
  LONG countLo, countHi;/* 64-bit bit count */
  LONG data [16];/* SHS data buffer */
} SHS_INFO;

/* Turn off prototypes if requested */
#if (defined(NOPROTO) && defined(PROTO))
# undef PROTO
#endif

/* Used to remove arguments in function prototypes for non-
ANSI C */
#ifdef PROTO
# define OF(a) a
#else/* !PROTO */
# define OF(a) ()
#endif/* ?PROTO */

#definelocalstatic

void shsInit OF((SHS_INFO *shsInfo));
void shsUpdate OF((SHS_INFO *shsInfo,
                     BYTE *buffer, int count));
void shsFinal OF((SHS_INFO *shsInfo));

#endif
```

D MD5

Programlistingen av md5, inkludert mine modifikasjoner. Jeg gjør oppmerksom på at md5 er kopibeskyttet av RSA Data Security, Inc.

D.1 md5drvr.c

```
/*
  md5driver.c -- sample test routines
  RSA Data Security, Inc. MD5 Message-Digest Algorithm
  Created: 2/16/90 RLR
  Updated: 1/91 SRD

  Adapted to ANSI C  - August 1995, Jan Anders Solvik

*/

/*
  Copyright (C) 1990, RSA Data Security, Inc. All rights
  reserved RSA Data Security, Inc. makes no representations
  concerning either the merchantability of this software or
  the suitability of this software for any particular
  purpose. It is provided "as is" without express or implied
  warranty of any kind. These notices must be retained in any
  copies of any part of this documentation and/or software.       **
*/

#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include "md5.h"

/* Prints message digest buffer in mdContext as 32 hexadecimal
  digits. Order is from low-order byte to high-order byte of
  digest. Each byte is printed with high-order hexadecimal
  digit first.
*/
static void MDPrint (MD5_CTX *mdContext)
{
int i;
div_t n;

  for (i = 0; i < 16; i++)
       {
       printf ("%02x", mdContext->digest[i]);
       n = div(i+1,4);
       /* Space between every 32 bits */
       if (n.rem == 0) printf (" ");
       }
}
```

```
/* size of test block */
#define TEST_BLOCK_SIZE 1000

/* number of blocks to process */
#define TEST_BLOCKS 100000

/* number of test bytes = TEST_BLOCK_SIZE * TEST_BLOCKS */
static UINT4 TEST_BYTES =
            (UINT4)TEST_BLOCK_SIZE * (UINT4)TEST_BLOCKS;

/*
  A time trial routine, to measure the speed of MD5.
  Measures wall time required to digest
  TEST_BLOCKS * TEST_BLOCK_SIZE characters.
*/
static void MDTimeTrial (void)
{
MD5_CTX mdContext;
time_t endTime, startTime;
unsigned char data[TEST_BLOCK_SIZE];
unsigned int i;

  /* initialize test data */
  for (i = 0; i < TEST_BLOCK_SIZE; i++)
    data[i] = (unsigned char)(i & 0xFF);

  /* start timer */
  printf ("MD5 time trial. Processing %ld characters...\n",
               TEST_BYTES);
  time (&startTime);

  /* digest data in TEST_BLOCK_SIZE byte blocks */
  MD5Init (&mdContext);
  for (i = TEST_BLOCKS; i > 0; i--)
    MD5Update (&mdContext, data, TEST_BLOCK_SIZE);
  MD5Final (&mdContext);

  /* stop timer, get time difference */
  time (&endTime);
  MDPrint (&mdContext);
  printf ("is digest of test input.\n");
  printf ("Seconds to process test input: %ld\n",
            (UINT4)(endTime-startTime));
  printf("Characters processed per second: %ld\n",
            TEST_BYTES/(endTime-startTime));
}
```

```
/*
  Computes the message digest for string inString.Prints out
  message digest, a space, the string (in quotes)
  and a carriage return.
*/
static void MDString (char *inString)
{
MD5_CTX mdContext;
unsigned int len = strlen (inString);

  MD5Init (&mdContext);
  MD5Update (&mdContext, inString, len);
  MD5Final (&mdContext);
  MDPrint (&mdContext);
  printf (" \"%s\"\n", inString);
}

/*
  Computes the message digest for a specified file.Prints out
  message digest, a space, the file name,
  and a carriage return.
*/
static void MDFile (char *filename)
{
FILE *inFile = fopen (filename, "rb");
MD5_CTX mdContext;
int bytes;
unsigned char data[1024];

  if (inFile == NULL) {
   printf ("%s can't be opened.\n", filename);
   return;
  }

  MD5Init (&mdContext);
  while ((bytes = fread (data, 1, 1024, inFile)) != 0)
       MD5Update (&mdContext, data, bytes);
  MD5Final (&mdContext);
  MDPrint (&mdContext);
  printf (" %s\n", filename);
  fclose (inFile);
}
```

```
/*
   Writes the message digest of the data from stdin onto
   stdout,followed by a carriage return.
*/
static void MDFilter (void)
{
MD5_CTX mdContext;
int bytes;
unsigned char data[16];

  MD5Init (&mdContext);
  while ((bytes = fread (data, 1, 16, stdin)) != 0)
    MD5Update (&mdContext, data, bytes);
  MD5Final (&mdContext);
  MDPrint (&mdContext);
  printf ("\n");
}

/*
   Runs a standard suite of test data.
*/
static void MDTestSuite (void)
{
  printf ("MD5 test suite results:\n");
  MDString("");
  MDString("a");
  MDString("abc");
  MDString("message digest");
  MDString("abcdefghijklmnopqrstuvwxyz");
  MDString("ABCDEFGHIJKLMNOPQRSTUVWXYZ
            abcdefghijklmnopqrstuvwxyz0123456789");
  MDString("12345678901234567890123456789012345678901\
            2345678901234567890123456789012345678901234567890");
  /* Contents of file foo are "abc" */
  MDFile ("foo");
}
```

```
void MDPrintStatistics(const UINT4 InnTabell[], UINT4 table-
size, UINT4 MDIndex, UINT4 Index)
{
UINT4 i,j,tempmax,tabindex;
UINT4 collisions[1000];

  tempmax = 0;
  for (j = 0; j < 1000; j++)/* Reset statistics table */
      collisions[j]= 0;
  printf("\nPartOfMessageDigest: %d Index: %d\n",
          MDIndex, Index);
  for (i = 0; i < tablesize; i++)/* Search the whole table */
  {
      for (j = 0; j < 1000; j++)/* Max. 1000 collisions */
      {
      if (InnTabell[i] == j)
           (collisions[j])++;
      }
  }

  printf("There are %ld empty hashvalues\n" ,collisions[0]);
  printf("There are %ld single hashvalues\n",collisions[1]);

  for (j = 2; j < 1000; j++)/* Maximum 1000 collisions */
      {
      if ((collisions[j]) > 0)
          printf("There are %ld hashvalues with %ld
                  collision(s)\n", collisions[j],j);
      }
}

void MDPrintStatisticsToFile(const UINT4 InnTabell[], UINT4
tablesize, UINT4 MDIndex, UINT4 Index)
{
UINT4 i,j,collisions[1000];
char  file_name[15];
FILE *outfileptr;

  sprintf(file_name, "TestFile_md5_%d_%d%d.txt",
          HashPartSize, MDIndex, Index);

  outfileptr = fopen(file_name, "w");
  for (j = 0; j < 1000; j++)/* Reset statistics table */
      collisions[j]= 0;
  for (i = 0; i < tablesize; i++)/* Search the whole table */
  {
  fprintf(outfileptr, "%d\n", InnTabell[i]);
  }

fclose(outfileptr);
}
```

```
/*
  A better random generator
*/
int uniform ( int max )
{
     struct timeval tp;
     struct timezone tzp;

     gettimeofday ( &tp, &tzp );
     srand((375467*tp.tv_sec) + (235783*tp.tv_usec));
     return ( rand() %  max);
}

/*
  Makes random text strings.
*/
char *MDRandomString(int stringlength, char string[])
{
UINT4 tall,tallmod,i;
char bokstaver[26] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
  'i', 'j','k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
  'u', 'v', 'w', 'x', 'y', 'z'};

  for(i = 0; i < stringlength; i++)
       {
       tall = rand();
       tallmod = (tall%26);
/*     tallmod = uniform(26);  A better random generator*/
       string[i] = bokstaver[tallmod];
        }
return (string);
}

/*
  Makes random numbers 0 - range
*/
UINT4 MDRandomNumber(UINT4 range)
{
UINT4 tempnumber, Number;

       tempnumber = rand();
       Number = (tempnumber%range);
/*     Number = uniform(range); A better random generator*/

return (Number);
}
```

```
/*
  To get parts of the massage digest
*/
UINT4 MDPartOfMessageDigest(MD5_CTX *mdContext, UINT4
       MessageDigestPartIndex, UINT4 Index,  UINT4 tablesize)
{
UINT4 temp, Part, maxindex;

  temp = (mdContext->buf[MessageDigestPartIndex]);
  /* Split into 1/2 of MessageIndexPart 16 bits*/
  if (tablesize == 65536)
       {
       if (Index == 1) temp = (temp >> 16);
  /* Split into 1/4 of MessageIndexPart 8 bits*/}
  if (tablesize == 256)
       {
       if (Index == 0) temp = (temp >> 24);
       if (Index == 1) temp = (temp >> 16);
       if (Index == 2) temp = (temp >> 8);
       }
  /* Split into 1/8 of MessageIndexPart, 4 bits */
  if (tablesize == 16)
       {
       if (Index == 0) temp = (temp >> 28);
       if (Index == 1) temp = (temp >> 24);
       if (Index == 2) temp = (temp >> 20);
       if (Index == 3) temp = (temp >> 16);
       if (Index == 4) temp = (temp >> 12);
       if (Index == 5) temp = (temp >> 8);
       if (Index == 6) temp = (temp >> 4);
       }
  Part = (temp & (tablesize - 1)); /* Actual part */
  }
return (Part);
}
```

```
/*
  Main procedure for the statistical testing
*/
void MDRandom(void)
{
MD5_CTX mdContext;
UINT4 i, j, messageindex, maxindex,
       index, NumberOfRandomNumber;
UINT4 MessageDigestPart;
char RandomString[StringLength];
UINT4 Table[TableSize];

  printf("Enter # RandomNumber> ");
  scanf("%d", &NumberOfRandomNumber);

  for (messageindex = 0; messageindex < 4; messageindex++)
  {
       for (index = 0; index < maxindex; index++)
       {
       for (j = 0; j < TableSize; j++) /* Reset table */
            Table[j] = 0;
       for (i = 0; i < NumberOfRandomNumber; i++)
            {
            MDRandomString(StringLength, RandomString);
            MD5Init (&mdContext);
            MD5Update (&mdContext, (unsigned char *)
                 RandomString,StringLength);
            MD5Final (&mdContext);

            Table[MDPartOfMessageDigest(&mdContext,
                      messageindex, index, TableSize)]++;
            }
  MDPrintStatistics(Table, TableSize, messageindex, index);
  MDPrintStatisticsToFile(Table, TableSize,
                           messageindex, index);
       }
  }
}
```

```
int main (int argc, char *argv)
{
int i;

/*
  For each command line argument in turn:
  filename -    prints message digest and name of file
  -sstring -    prints message digest and contents of string
  -t       -    prints time trial statistics for 10M
                characters
  -x       -    execute a standard suite of test data
  -r       -    prints statistics of part of
                the message digest of a random string
  (no args) -   writes messages digest of stdin onto stdout
*/

  if (argc == 1)
      MDFilter ();
  else
      for (i = 1; i < argc; i++)
          if (argv[i][0] == '-' && argv[i][1] == 's')
              MDString (argv[i] + 2);
  else if (strcmp (argv[i], "-t") == 0)
      MDTimeTrial ();
  else if (strcmp (argv[i], "-x") == 0)
      MDTestSuite ();
  else if (strcmp (argv [i], "-r") == 0)
      MDRandom ();
  else MDFile (argv[i]);

  return(0);
}
```

D.2 md5.c

```
/*
  md5.c -- the source code for MD5 routines                          **
  RSA Data Security, Inc. MD5 Message-Digest Algorithm               **
  Created: 2/17/90 RLR                                               **
  Revised: 1/91 SRD,AJ,BSK,JT Reference C ver.,
  7/10 constant corr.

  Adapted to ANSI C  - August 1995, Jan Anders Solvik

*/

/*
  Copyright (C) 1990, RSA Data Security, Inc. All rights
  reserved.License to copy and use this software is granted
  provided that it is identified as the "RSA Data Security,
  Inc. MD5 Message- Digest Algorithm" in all material
  mentioning or referencing this software or this function.
  License is also granted to make and use derivative works
  provided that such works are identified as "derived from
  the RSA Data Security, Inc. MD5 Message-Digest Algorithm"
  in all material mentioning or referencing the derived
  work. RSA Data Security, Inc. makes no representations
  concerning either the merchantability of this software or
  the suitability of this software for any particular
  purpose. It is provided "as is" without express or implied
  warranty of any kind. These notices must be retained in any
  copies of any part of this documentation and/or software.
*/

#include "md5.h"

/* forward declaration */
static void Transform ();

static unsigned char PADDING[64] = {
  0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

/* F, G, H and I are basic MD5 functions */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))
```

```
/*
  FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4
  Rotation is separate from addition to prevent recomputation
*/
#define FF(a, b, c, d, x, s, ac) \
  {(a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }
#define GG(a, b, c, d, x, s, ac) \
  {(a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }
#define HH(a, b, c, d, x, s, ac) \
  {(a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }
#define II(a, b, c, d, x, s, ac) \
  {(a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
   (a) = ROTATE_LEFT ((a), (s)); \
   (a) += (b); \
  }

/*
  The routine MD5Init initializes the message-digest
  context mdContext. All fields are set to zero.
*/
void MD5Init (MD5_CTX *mdContext)
{
  mdContext->i[0] = mdContext->i[1] = (UINT4)0;

/*
  Load magic initialization constants.
*/
  mdContext->buf[0] = (UINT4)0x67452301;
  mdContext->buf[1] = (UINT4)0xefcdab89;
  mdContext->buf[2] = (UINT4)0x98badcfe;
  mdContext->buf[3] = (UINT4)0x10325476;
}
```

```
/*
   The routine MD5Update updates the message-digest context
   to account for the presence of each of the characters
   inBuf[0..inLen-1] in the message whose digest is being
   computed.
*/
void MD5Update (MD5_CTX *mdContext, unsigned char *inBuf,
                unsigned int inLen)
{
UINT4 in[16];
int mdi;
unsigned int i, ii;

  /* compute number of bytes mod 64 */
  mdi = (int)((mdContext->i[0] >> 3) & 0x3F);

  /* update number of bits */
  if ((mdContext->i[0] +
       ((UINT4)inLen << 3)) < mdContext->i[0])
   mdContext->i[1]++;
   mdContext->i[0] += ((UINT4)inLen << 3);
   mdContext->i[1] += ((UINT4)inLen >> 29);

   while (inLen--) {
   /* add new character to buffer, increment mdi */
   mdContext->in[mdi++] = *inBuf++;

   /* transform if necessary */
   if (mdi == 0x40) {
        for (i = 0, ii = 0; i < 16; i++, ii += 4)
             in[i] =   (((UINT4)mdContext->in[ii+3]) << 24) |
                       (((UINT4)mdContext->in[ii+2]) << 16) |
                       (((UINT4)mdContext->in[ii+1]) << 8) |
                       ((UINT4)mdContext->in[ii]);
        Transform (mdContext->buf, in);
        mdi = 0;
        }
   }
}
```

```
/*
   The routine MD5Final terminates the message-digest
   computation and ends with the desired message digest
   in mdContext->digest[0...15].
*/
void MD5Final (MD5_CTX *mdContext)
{
UINT4 in[16];
int mdi;
unsigned int i, ii;
unsigned int padLen;

  /* save number of bits */
  in[14] = mdContext->i[0];
  in[15] = mdContext->i[1];

  /* compute number of bytes mod 64 */
  mdi = (int)((mdContext->i[0] >> 3) & 0x3F);

  /* pad out to 56 mod 64 */
  padLen = (mdi < 56) ? (56 - mdi) : (120 - mdi);
  MD5Update (mdContext, PADDING, padLen);

  /* append length in bits and transform */
  for (i = 0, ii = 0; i < 14; i++, ii += 4)
    in[i] = (((UINT4)mdContext->in[ii+3]) << 24) |
            (((UINT4)mdContext->in[ii+2]) << 16) |
            (((UINT4)mdContext->in[ii+1]) << 8) |
            ((UINT4)mdContext->in[ii]);
  Transform (mdContext->buf, in);

  /* store buffer in digest */
  for (i = 0, ii = 0; i < 4; i++, ii += 4) {
    mdContext->digest[ii] =
        (unsigned char)(mdContext->buf[i] & 0xFF);
    mdContext->digest[ii+1] =
        (unsigned char)((mdContext->buf[i] >> 8) & 0xFF);
    mdContext->digest[ii+2] =
        (unsigned char)((mdContext->buf[i] >> 16) & 0xFF);
    mdContext->digest[ii+3] =
        (unsigned char)((mdContext->buf[i] >> 24) & 0xFF);
  }
}
```

```
/*
   Basic MD5 step. Transforms buf based on in.
*/
static void Transform (UINT4 *buf, UINT4 *in)
{
  UINT4 a = buf[0], b = buf[1], c = buf[2], d = buf[3];

  /* Round 1 */
#define S11 7
#define S12 12
#define S13 17
#define S14 22
  FF ( a, b, c, d, in[ 0], S11, 3614090360); /* 1 */
  FF ( d, a, b, c, in[ 1], S12, 3905402710); /* 2 */
  FF ( c, d, a, b, in[ 2], S13,  606105819); /* 3 */
  FF ( b, c, d, a, in[ 3], S14, 3250441966); /* 4 */
  FF ( a, b, c, d, in[ 4], S11, 4118548399); /* 5 */
  FF ( d, a, b, c, in[ 5], S12, 1200080426); /* 6 */
  FF ( c, d, a, b, in[ 6], S13, 2821735955); /* 7 */
  FF ( b, c, d, a, in[ 7], S14, 4249261313); /* 8 */
  FF ( a, b, c, d, in[ 8], S11, 1770035416); /* 9 */
  FF ( d, a, b, c, in[ 9], S12, 2336552879); /* 10 */
  FF ( c, d, a, b, in[10], S13, 4294925233); /* 11 */
  FF ( b, c, d, a, in[11], S14, 2304563134); /* 12 */
  FF ( a, b, c, d, in[12], S11, 1804603682); /* 13 */
  FF ( d, a, b, c, in[13], S12, 4254626195); /* 14 */
  FF ( c, d, a, b, in[14], S13, 2792965006); /* 15 */
  FF ( b, c, d, a, in[15], S14, 1236535329); /* 16 */

  /* Round 2 */
#define S21 5
#define S22 9
#define S23 14
#define S24 20
  GG ( a, b, c, d, in[ 1], S21, 4129170786); /* 17 */
  GG ( d, a, b, c, in[ 6], S22, 3225465664); /* 18 */
  GG ( c, d, a, b, in[11], S23,  643717713); /* 19 */
  GG ( b, c, d, a, in[ 0], S24, 3921069994); /* 20 */
  GG ( a, b, c, d, in[ 5], S21, 3593408605); /* 21 */
  GG ( d, a, b, c, in[10], S22,   38016083); /* 22 */
  GG ( c, d, a, b, in[15], S23, 3634488961); /* 23 */
  GG ( b, c, d, a, in[ 4], S24, 3889429448); /* 24 */
  GG ( a, b, c, d, in[ 9], S21,  568446438); /* 25 */
  GG ( d, a, b, c, in[14], S22, 3275163606); /* 26 */
  GG ( c, d, a, b, in[ 3], S23, 4107603335); /* 27 */
  GG ( b, c, d, a, in[ 8], S24, 1163531501); /* 28 */
  GG ( a, b, c, d, in[13], S21, 2850285829); /* 29 */
  GG ( d, a, b, c, in[ 2], S22, 4243563512); /* 30 */
  GG ( c, d, a, b, in[ 7], S23, 1735328473); /* 31 */
  GG ( b, c, d, a, in[12], S24, 2368359562); /* 32 */
```

```
  /* Round 3 */
#define S31 4
#define S32 11
#define S33 16
#define S34 23
  HH ( a, b, c, d, in[ 5], S31, 4294588738); /* 33 */
  HH ( d, a, b, c, in[ 8], S32, 2272392833); /* 34 */
  HH ( c, d, a, b, in[11], S33, 1839030562); /* 35 */
  HH ( b, c, d, a, in[14], S34, 4259657740); /* 36 */
  HH ( a, b, c, d, in[ 1], S31, 2763975236); /* 37 */
  HH ( d, a, b, c, in[ 4], S32, 1272893353); /* 38 */
  HH ( c, d, a, b, in[ 7], S33, 4139469664); /* 39 */
  HH ( b, c, d, a, in[10], S34, 3200236656); /* 40 */
  HH ( a, b, c, d, in[13], S31,  681279174); /* 41 */
  HH ( d, a, b, c, in[ 0], S32, 3936430074); /* 42 */
  HH ( c, d, a, b, in[ 3], S33, 3572445317); /* 43 */
  HH ( b, c, d, a, in[ 6], S34,   76029189); /* 44 */
  HH ( a, b, c, d, in[ 9], S31, 3654602809); /* 45 */
  HH ( d, a, b, c, in[12], S32, 3873151461); /* 46 */
  HH ( c, d, a, b, in[15], S33,  530742520); /* 47 */
  HH ( b, c, d, a, in[ 2], S34, 3299628645); /* 48 */

  /* Round 4 */
#define S41 6
#define S42 10
#define S43 15
#define S44 21
  II ( a, b, c, d, in[ 0], S41, 4096336452); /* 49 */
  II ( d, a, b, c, in[ 7], S42, 1126891415); /* 50 */
  II ( c, d, a, b, in[14], S43, 2878612391); /* 51 */
  II ( b, c, d, a, in[ 5], S44, 4237533241); /* 52 */
  II ( a, b, c, d, in[12], S41, 1700485571); /* 53 */
  II ( d, a, b, c, in[ 3], S42, 2399980690); /* 54 */
  II ( c, d, a, b, in[10], S43, 4293915773); /* 55 */
  II ( b, c, d, a, in[ 1], S44, 2240044497); /* 56 */
  II ( a, b, c, d, in[ 8], S41, 1873313359); /* 57 */
  II ( d, a, b, c, in[15], S42, 4264355552); /* 58 */
  II ( c, d, a, b, in[ 6], S43, 2734768916); /* 59 */
  II ( b, c, d, a, in[13], S44, 1309151649); /* 60 */
  II ( a, b, c, d, in[ 4], S41, 4149444226); /* 61 */
  II ( d, a, b, c, in[11], S42, 3174756917); /* 62 */
  II ( c, d, a, b, in[ 2], S43,  718787259); /* 63 */
  II ( b, c, d, a, in[ 9], S44, 3951481745); /* 64 */

  buf[0] += a;
  buf[1] += b;
  buf[2] += c;
  buf[3] += d;
}
```

D.3 md5.h

```
/* typedef unsigned long int UINT4; 32 bits */
typedef unsigned int UINT4; /* 64 bits Alpha */

#define StringLength 300/* Randomstring Length */
#define HashPartSize    8/* 4, 8 or 16 */
#define TableSize256  /* 2^HashPartSize  16, 256 or 65536 */

/* Data structure for MD5 (Message-Digest) computation */
typedef struct {
  UINT4 i[2]; /* number of _bits_ handled mod 2^64 */
  UINT4 buf[4]; /* scratch buffer */
  unsigned char in[64]; /* input buffer */
  unsigned char digest[16]; /* after MD5Final call */
} MD5_CTX;

void MD5Init ();
void MD5Update ();
void MD5Final ();
```