PROSJEKT I KRYPTOLOGI
IMT4051

Av:
Ole Kasper Olsen
Fredrik Skarderud
Torkjel Søndrol
Ole Martin Dahl

Forord

Vi har i denne oppgaven sett på kryptografiske hashfunksjoner. Vi starter rapporten med å se på hva en kryptografisk hashfunksjon er og hva som kjennetegner de ulike typene hashfunksjonene. Deretter ser vi på ulike bruksområder, og hvordan funksjonene kan angripes før vi tar en titt på noen av de mest brukte hashfunksjonene i dag.

I siste del av oppgaven vår har vi implementert vår egen hashfunksjon basert på samme prinsipper som moderne hashfunksjoner som MD4/5 og SHA-1 for å lære mer om hvordan moderne hashfunksjoner er bygget opp. Vi har også sammenlignet vår hashfunksjon mot noen av dagens hashfunksjoner for å se litt på ytelse og skalering.

1 Introduksjon

Hashfunksjoner spiller en viktig rolle innen moderne kryptologi. Hashfunksjoner har til hensikt å komprimere en inputsetning av vilkårlig lengde til en setning med fast lengde, og på denne måten gjøre inputen uleselig. Den resulterende strengen blir da hash-koden eller hashverdien til inputsetningen. For å si det mer presist, mapper en hashfunksjon h bitstrenger av vilkårlig lengde ned til en streng av fastsatt lengde på n bit. For en mengde D og et utvalg R med $h : D \rightarrow R$ og $|D| > |R|$ (hvor |D| og |R| er antall elementer i mengdene), sier man at funksjonen er mange-til-en, noe som betyr at det finnes *kollisjoner*, dvs. at flere ulike inputsetninger gir samme hash-kode. I en god hashfunksjon vil det imidlertid være svært lite sannsynlig å finne slike kollisjoner.

Tanken bak kryptografiske hashfunksjoner er at en hashverdi kan være en kompakt representasjon av en inputstreng. Dette kan den kun være hvis det er usannsynlig at man kan bruke den til å identifisere inputstrengen.

Hashfunksjoner brukes til dataintegritet sammen med digitale signaturer, hvor en melding først blir hashet, og hash-koden så blir signert i stedet for den opprinnelige meldingen. En egen gruppe med hashfunksjoner kalt MAC-funksjoner gir mulighet for meldingsautentisering gjennom lignende teknikker.

En typisk bruk av hashfunksjoner er som følger: Hashverdien til en bestemt melding x blir beregnet i tidspunktet T_1. Dataintegriteten til denne hashverdien blir beskyttet på et eller annet vis, for eksempel i en database, og x blir slettet. På et senere tidspunkt T_2 blir det gjort en test for å se om meldingen er blitt endret, dvs. man ser om meldingen x' er den samme som den originale meldingen. Hashverdien til x' blir derfor regnet ut, og sammenlignet med hashverdien til x for å se om meldingene er den samme. Problemet med å ta vare på integriteten til en potensielt lang melding er dermed redusert til å ta vare på en liten hashverdi av fast størrelse. Siden man er garantert at det finnes kollisjoner ved mange-til-en mappinger, kan den unike assosiasjonen mellom input og hashverdi i beste fall være beregningsmessig liten. En hashverdi bør i praksis være unikt identifiserbar med en eneste input, og kollisjoner bør være beregningsmessig umulige å finne.

2 Kort om hashfunksjoner

Når man omtaler hashfunksjoner er kollisjonsfrihet viktig. Det mest grunnleggende kravet til en god hash funksjon *h* er at det skal være ugjennomførbart for en angriper å regne ut en kollisjon, dvs. at det ikke skal være mulig å finne to forskjellige inputtekster *x* og *x'* slik at *h(x)=h(x').* Terminologien "ugjennomførbart" er relativ og er vanskelig å definere. Noe som er praktisk ugjennomførbart i dag vil kanskje være mulig å løse om et år. Dette kommer av at datakraft og muligheten for brute-force knekking øker for hvert år som går jfr. det som kalles "Moores lov", nemlig det at ytelsen dobles hver 17. måned [Moore65].

Det er enkelte egenskaper som gode hashfunksjoner ofte har [Menezes96]:

Compression	*h* gjør en input *x* av hvilken som helst lengde om til en output *h(x)* med en fast bestemt lengde
Ease of computation	Gitt *h* og en input *x* skal det være lett beregne *h(x)*
Preimage resistance	Det er praktisk umulig å finne en input som hasher til en forhåndsdefinert output, dvs. at ved en gitt hashverdi *y*, er det praktisk ugjennomførbart å finne en melding *x* slik at *h(x) = y*
second-preimage resistance	Gitt en forhåndsdefinert input er det praktisk umulig å finne enda en input som har samme hashverdi, dvs. gitt en *x* og *h(x)* er det praktisk ugjennomførbart å finne en melding *x′* slik at *h(x′) = h(x)*
Collison resistance	Det er praktisk ugjennomførbart å finne to unike inputs som hasher til den samme output

Tabell 1 - Viktige egenskaper

For å kunne si at en hash funksjon er enveis eller "one way hash function" (OWHF) må egenskapene c*ompression, ease of computation, preimage resistance og second-preimage resistance* være oppfylt. For å kunne si at en hash funksjon er kollisjonsfri eller at den er en "collision resistant hash function" (CRHF), må egenskapene c*ompression, ease of computation, second-preimage resistance og collison resistance* være oppfylt. I praksis er også egenskapen *preimage resistance* oppfylt i kollisjonsfrie hashfunksjoner.

En annen egenskap som skal sies å være viktig for både kollisjonsfrie og enveis hashfunksjoner er som definert av I. B. Damgård [Damgård88]: Beskrivelsen av hash funksjonen *h* må være allment tilgjengelig og det må ikke være nødvendig med noen hemmelig informasjon for å bruke den. Denne egenskapen bør i grunn gjelde for alle typer hashfunksjoner.

Et eksempel på hvor viktig det er at hashfunksjoner er enveis og kollisjonsfri er ved signering av dokumenter. Hvis det er gitt en hash av et dokument må det være ugjennomførbart å kunne generere en ny tekst som gir den samme hashen. Dette er essensielt hvis en hash av et dokument skal brukes for ivareta integritet til dokumentet.

2.1 Hashfunksjoner med og uten nøkkel

Det finnes i hovedsak to forskjellige typer hashfunksjoner. Såkalte MAC-funksjoner (Message Authentication Code) og MDC-funksjoner (Message Digest Code). Den store forskjellen på disse to er at MAC-funksjonene baserer seg rundt en hemmelig nøkkel for å generere det hashede resultatet av en input, mens MDC funksjoner ikke bruker noen hemmelig nøkkel for å generere det hashede resultatet.

MAC- eller Hashed-MAC-funksjoner (HMAC) ble til som et resultat av kravene til IPSec, en utviklingsgruppe i IETF[1], men har nå blitt en standard. En MAC-kode er en blokk med fast lengde av data generert av meldingen pluss en hemmelig nøkkel. Denne blokken vil fungere som en kryptografisk sjekksum for en melding. MAC-funksjoner har mange bruksområder pga sin evne til å generere unike sjekksummer. Eksempler kan være:

- Kringkastingsmeldinger
- Verifikasjon av programvare
- For å generere sikre sjekksummer av passord

For at en hashfunksjon skal være en MAC-funksjon, må den som andre hashfunksjoner ha egenskapene *ease of computation* og *compression*. I tillegg skal ikke en angriper på bakgrunn av mange sett klartekstverdier og hashverdier kunne beregne en ny hashverdi. Dette vil si at gitt parene $(x_i, h_x(x_i))$, skal det være ugjennomførbart å finne $(x, h_k(x))$ hvor x er forskjellig fra x_i.

Dette betyr at en MAC skal være både enveis og kollisjonsresistent for noen som ikke kjenner nøkkelen k, men overnevnte definisjon gir ikke noe svar på om MAC-funksjonen skal være enveis og kollisjonsresistent for noen som kjenner nøkkelen, k.

[1] The Internet Engineering Task Force, http://www.ietf.org

3 Anvendelser

3.1 Digitale signaturer

Signering av dokumenter gjøres for å autentisere dokumentene. Digitale signaturer kan plasseres på dokumenter og andre data som et bevis på at du garanterer innholdet i dokumentene. Man får dermed en garanti mot at uvedkommende har endret innholdet i dokumentet etter at det ble signert av forfatteren. Her benyttes det som regel en hash-algoritme som produserer en hashverdi av meldingen. Deretter blir hashverdien signert med en offentlig nøkkelalgoritme som for eksempel RSA. Man får med dette en signatur som sendes sammen med meldingen. En som skal lese denne meldingen må først dekryptere signaturen med den offentlige nøkkelen for å få hashverdien. Deretter må man ut fra den mottatte meldingen beregne en hashverdi, og sammenligne denne med den dekrypterte signaturen. Er de like, er meldingen autentisk, er de ulike har noen endret på meldingen, og den må forkastes.

Fordelene med å signere en hashverdi i stede for å signere hele meldingen er mange:

- Man øker hastigheten betraktelig.
- Man kan redusere størrelsen på signaturen.
- Det går tregere å signere og verifisere enn å hashe.
- Det er lettere å manipulere signerte blokker som ikke er hashet.
- Man kan ødelegge den algebraiske strukturen i meldingsmengden. RSA og ElGamal kan ha algebraiske strukturer som angriper kan bruke til sin fordel.
- Man unngår problemet med reblokking. Reblokking oppstår når kryptering og autentisering blir ivaretatt av RSA. Hvis sender først krypterer meldingen med offentlig nøkkel til mottager, kan resultat bli større en senders modulus. Unngåes i RSA ved at rekkefølgen på kryptering og signering må være avhengig av størrelsen på modulusene.
- En som ønsker å endre signaturen vil ikke kunne bruke signeringsprotokollen for å finne klarteksten som tilsvarer den krypterte meldingen.

3.2 Passord

Dette var det første området hvor hashing ble benyttet, hvor passord ble lagret i en passordfil og sammenlignet med det brukeren skrev inn. Passordene lå da krypterte, og brukerne hadde ikke tilgang til passordfilen, men dette var ikke tilstrekkelig rent sikkerhetsmessig. Derfor sikrer man passordene vha. hashfunksjoner, hvor det var hashverdiene til passordene som ble lagret i filen. Når en bruker skrev inn et passord ble også dette passordet hashet og sammenlignet med hash-koden av passordet som ligger lagret. Hvis verdiene var identiske ble inntastingen godkjent. Heller ikke denne metoden var helt sikker, siden en inntrenger kunne prøve flere passord fra en liste, og dårlig valgte passord ville dermed kunne knekkes. Sikkerheten var her basert på at hash-algoritmen som benyttes er enveis – dvs. at man ikke kan beregne passordet dersom man har kjennskap til hashverdien med andre metoder enn brute force.

3.3 Virus og beskyttelse av software

Dersom man har et program som skal beskyttes mot modifisering fra for eksempel et virusangrep eller uønsket endring av programmer, kan utvikleren lagre hashverdien av programmet. Hashverdien kan beregnes både for kjørbar kode og for kildekode, og blir distribuert uavhengig av programmet. Man kan dermed sammenligne denne hashverdien med hashverdien av programmet man har for å se om endringer er gjort.

3.4 Kryptering

Man kan ikke benytte hashfunksjoner direkte til kryptering, siden man ikke skal kunne beregne inputmeldingen ut fra hashverdien. Men deler av hashalgoritmer kan benyttes som en klassisk krypteringsalgoritme, da enkelte eldre hashalgoritmer var basert på DES og lignende systemer.

3.5 Ikke-kryptografiske hashfunksjoner

Det er ikke bare innen kryptografi man bruker hashfunksjoner. Man kan benytte seg av hashfunksjoner for å fordele meldinger eller data over et gitt område i minnet. I motsetning til vanlige kryptografiske hashfunksjoner kan disse ha kollisjoner. Dette resulterer bare i mindre effektiv lagring i minnet. Som navnet tilsier har disse funksjonene ingenting med kryptografi å gjøre. Vi utdyper derfor ikke dette noe videre.

4 Angrep på hashfunksjoner

4.1 Sikkerhetsobjektiver

Sikkerhetsobjektivene man ser på under evaluering av hashfunksjoner kan deles i to; målene for designen og målene for angrep. Disse målene er illustrert i tabellen under:

Hash type	Mål for design	Ideell styrke	Mål for angrep
OWHF	- *Preimage resistance* - *second preimage resistance*	2^n	- Generere *preimage* - Finne 2. input til samme verdi
CRHF	- *Collision resistance*	2^n	- Produsere en kollisjon
MAC	- Skjult nøkkel - Kan ikke beregnes	$2^{n/2}$	- Finne MAC nøkkelen - Produsere ny melding eller nøkkel

Tabell 2 - sikkerhetsobjektiver, hvor n er antall bit i hashverdien

Gitt en spesifikk hashfunksjon, er det ønskelig å kunne bevise en nedre grense for kompleksitet ved forskjellige typer angrep med så få antagelser som mulig. Resultatene av dette er ofte skremmende. Det beste bildet av sikkerheten til en spesifikk hashfunksjon tilsvarer kompleksiteten av det mest effektive angrepet, noe som gir en *øvre* grense med hensyn på sikkerhet. Et angrep med kompleksiteten 2^t er en funksjon som krever omtrent 2^t operasjoner, hvor hver operasjon er en del av prosessen. Dette kan for eksempel være et kall til en kompresjonsalgoritme eller en kryptering av en underliggende tekst. Spesifikke metoder for å angripe hashverdier blir beskrevet under.

4.2 Tilfeldige angrep

Angriperen bruker en falsk melding, finner hashverdien til denne og håper på den tilsvarer hashverdien til den opprinnelige meldingen. Da hashfunksjonen vil ha 2^n mulige utfall, vil sannsynligheten for å lykkes være 2^{-n}, hvor n er antallet bits i hashverdien. Man kan si at angrepet lykkes hvis angriperen klarer å finne enten et *preimage* eller *second preimage*. Hvis angriperen har muligheten til å velge begge meldingene (dvs. at han vil kun prøve å finne en kollisjon), vil fødselsdagsparadokset inntreffe som beskrevet i neste avsnitt.

4.3 Fødselsdagsangrep

Ideen bak denne typen angrep er at i en gruppe på 23 personer, er sannsynligheten for at 2 personer har fødselsdag på samme dag større enn 50%. Da dette er en større sannsynlighet enn hva man kanskje skulle vente kalles det ofte fødselsdagsparadokset. Resultatet forutsetter at fødselsdagene er jevnt fordelt utover året. Hvis en for eksempel antar at de fleste er født om vinteren vil sannsynligheten for å finne kollisjoner være ennå høyere. Det er lav sannsynlighet for at en tilfeldig valgt elev har fødselsdag på samme dag som en annen elev, men i hele klassen (23 elever) er denne sannsynligheten 50% fordi man ser på alle mulige par av elever.

Med denne grunnleggende teorien i bakhodet, kan vi komme fram til at hvis angriperen har mulighet til å velge to meldinger fritt, vil han eller hun med mer enn 50% sannsynlighet ha funnet to meldinger som hasher til samme verdi (dvs. at han eller hun finner x og x' hvor $h(x) = h(x')$) etter bare $2^{n/2}$ forsøk.

Også hashfunksjoner med nøkler er sårbare overfor fødselsdagsangrep. Et eksempel:

Alice "signerer" en melding M ved å generere en n-bits hash som krypteres med en hemmelig nøkkel. M sendes i klartekst mens den krypterte hashen er vedlagt. Fødselsdagsparadokset sier at man med en n-bits hash vil finne en kollisjon med sannsynlighet på 50% etter $2^{n/2}$ input. Angriper genererer $2^{n/2}$ variasjoner av M og beregner hash av hver av disse. Angriper lager en falsk melding og generer $2^{n/2}$ variasjoner av denne og beregner hash av hver av disse. Fra disse hashverdiene vil han med sannsynlighet på større enn 50% finne to meldinger som har lik hash!

4.4 Chaining/lenking-angrep

Chaining angrep retter seg mot bruken av chaining variable i hashfunksjonene. Disse angrepene fokuserer på kompresjonsfunksjonene og ikke selve hashfunksjonen.

For eksempel: CBC-baserte (Chipher-Block-Chaining) hashfunksjoner er sårbare for Meet-In-The-Middle angrep. Møtes i midten angrep er variant av fødselsdagsangrep. Forskjellen er at det er mellomliggende lenkede variable som blir sammenlignet og ikke resultatet av hashen. Angrepet gjør det mulig for angriper å konstruere en melding med en forhåndsspesifisert hashverdi, som ikke er mulig ved fødselsdag angrep. Angriper kan altså legge inn informasjon i meldingen uten at hashen endres.

4.5 Angrep på en MDC med fast bitstørrelse

Et angrep på en MDC med fast bitstørrelse er som regel tilsvarende et tilfeldig angrep som beskrevet i kapittel 4.2. Man er altså gitt en melding x med en hashverdi $h(x)$ med størrelsen n bit. For å finne input som kolliderer med x må man velge en tilfeldig bitstreng x' og sjekke om $h(x') = h(x)$. Går man ut fra at hashverdien er en tilnærmet tilfeldig (dvs. totalt uforutsigbar), er sannsynligheten for treff lik 2^{-n}. Som også nevnt tidligere vil en angriper som kun er ute etter å finne kollisjoner i hashfunksjonen bare trenge å forsøke $2^{n/2}$ ganger grunnet fødselsdagsparadokset.

4.6 Angrep på nøkkelen til en MAC

Man kan prøve å finne nøkkelen vha. uttømmende søk (*exhaustive search*). Dersom man vet om et enkelt par av klartekst og hashverdi, kan en angriper prøve å beregne hashverdien på nytt med alle mulige nøkler. For et nøkkelrom på t bit krever dette 2^t operasjoner, hvor man forventer at det er igjen $1+2^{t-n}$ uprøvde nøkler. Dersom man går ut fra at MAC oppfører seg som en tilfeldig mapping, kan man finne nøkkelen etter bare t/n forsøk. Som hovedregel vil en MAC-nøkkel bli funnet etter mindre enn 2^t operasjoner. Dersom man skal finne en t bits nøkkel vha tilfeldig gjetting og en fast input, vil sannsynligheten for å finne rett nøkkel være ca 2^{-n} dersom $t < n$.

4.7 Angrep på MAC med fast bitstørrelse

Angrep på MAC går ut på at man genererer en input x og den korresponderende MAC verdien uten å ha kjennskap til nøkkelen. For en n-bit MAC algoritme vil, enten gjetting av MAC verdien til teksten, eller gjetting av et preimage for en gitt MAC verdi ha en sannsynlighet for suksess på ca 2^{-n}, som for MDC. Forskjellen her er derimot at de gjettede MAC verdiene ikke kan verifiseres uten kjennskap til enten nøkkelen eller en "black box" som gir MAC verdier for en gitt input. Ideelt sett bør man klare å produsere nye og korrekte tekst/MAC-verdipar med en sannsynlighet som er bedre en maksimum av 2^{-t} eller 2^{-n}.

4.8 Angrep vha forhåndsberegninger, flere mål og lange tekster

4.8.1 Forhåndsberegning av hashverdier

For angrep på *preimage* og *second preimage* kan en angriper som beregner store antall av par med input tekster og hashverdier kunne kutte ned på beregningene og lagringskapasiteten for senere angrepstid. For eksempel, kan en angriper for en 64-bit hashverdi tilfeldig velger 2^{40} input tekster og beregner deres hashverdier. Tekstene og verdiene blir lagret indeksert på hashverdi. Angriperen har dermed beregnet et sett i løpet av tiden $O(2^{40})$, som gjør at han øker sannsynligheten for å finne et preimage fra 2^{-64} til 2^{-24}. Samtidig øker sannsynligheten for å finne *second preimage* til *r* ganger den opprinnelige verdien hvor *r* er antallet input/output-par av en OWHF han har beregnet.

4.8.2 Parallelle mål for enveis hashfunksjoner

I et vanlig angrep prøver angriperen å finne et *second preimage* for et gitt mål, hvor målet er verdien beregnet fra et første *preimage*. Dersom det er *r* mål og angriper prøver å finne et *second preimage* til et av disse, har sannsynligheten for suksess økt til *r* ganger den opprinnelige sannsynligheten. En ting som da er underforstått er at når man bruker hashfunksjoner i forbindelse med nøkler, som for eksempel digitale signaturer, vil gjentagende bruk av den samme nøkkelen svekke sikkerheten. Dersom man har *r* signerte meldinger har sannsynligheten for kollisjon økt *r* ganger, og kolliderende meldinger vil få samme signatur, som en angriper ikke kunne beregnet på egenhånd.

4.8.3 Angrep på *second preimage* med lange meldinger

La h være en iterert n-bit hashfunksjon med kompresjonsfunksjonen f. La x være en melding bestående av t blokker. Da kan *second preimage* bli funnet med $(2n/s)+k$ operasjoner av f, og vil kreve minne tilsvarende $n(s+\lg(s))$ bits. Tanken er da å benytte fødselsdagsangrep på den mellomliggende hashverdien, og dermed få et utkast $s = t$. Man kan så beregne $h(x)$ og lagre (H_i, i) for hver av de t mellomliggende hashverdiene H_i som korresponderer med de t inputblokkene x_i i en tabell så de senere kan indekseres på verdi. Deretter beregner man $h(z)$ for tilfeldig valgte z, og sjekker om de kolliderer med noen av verdiene i tabellen. Dette vil kreve omtrent $2^n/s$ forskjellige z verdier, som følge av fødselsdagsparadokset. Man kan nå finne indeksen j for den verdien som gir kollisjon, og inputen $zx_{j+1}x_{j+2}\ldots x_t$ vil kollidere med x.

Dette viser at for "lange" meldinger er et *second preimage* vanligvis lettere å finne enn et *preimage*, noe som øker med lengden av x. Dersom $t \geq 2^{n/2}$ blir antall beregninger minket ved å velge $s = 2^{n/2}$, noe som vil føre til at *second preimage* koster omtrent $2^{n/2}$ kjøringer av f.

5 Dagens hashfunksjoner

5.1 Introduksjon

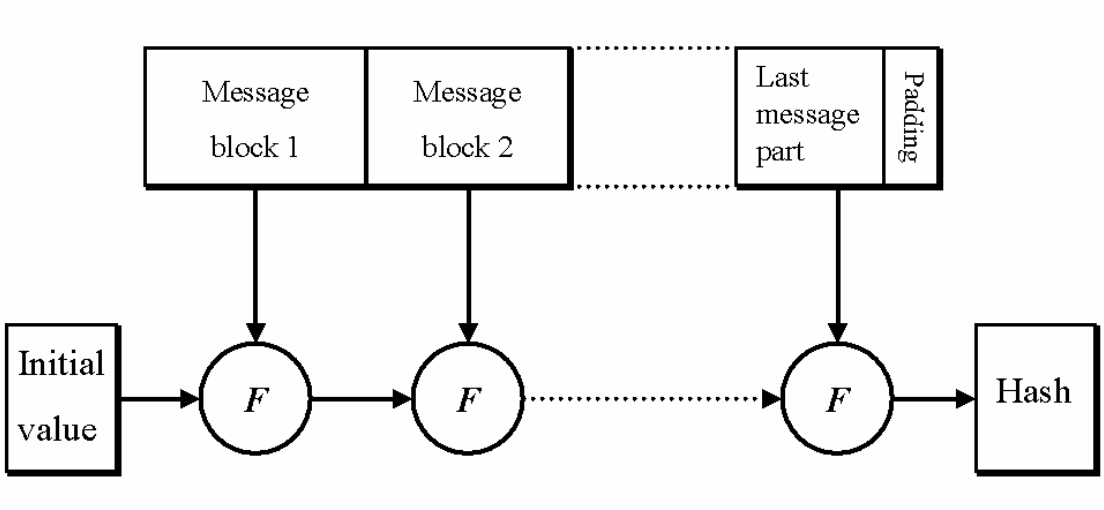
MD4 ble utviklet av Ronald Rivest i RSA Security, og gjort tilgjengelig i 1990 [Rivest90]. Dette var startskuddet for en helt ny gren av hashfunksjoner, som bygde på de samme prinsippene som MD4.

5.2 MD4 og MD5

MD4-algoritmen tar som argument en melding av hvilken som helst lengde, og genererer en hash på 128 bits fra argumentet. Som med de fleste andre hashfunksjoner som skal være av praktisk nytte, er det fastslått med forholdsvis stor sannsynlighet at det ikke vil være gjennomførbart med tanke på prosesseringskraft i datamaskiner å generere en melding som har samme hash (*second preimage resitance* og *collision resistance*), eller å generere den opprinnelige meldingen ut i fra den ferdige hashverdien (*preimage resistance*). MD4 var i første rekke ment brukt i digitale signaturer. Et hovedpoeng under utviklingen av MD4 var at den skulle være meget rask og effektiv.

Effektivitet kan ofte ha uønskede konsekvenser når man snakker om kryptologi, og det ble etter hvert oppdaget at MD4 ikke var sikker nok. Man ble redd for at noen kunne utføre et vellykket kryptoanalyseangrep på funksjonen, noe som også senere skjedde. Rivest kom da med MD5 i 1991 [Rivest92], som er basert på MD4, men går en runde i tillegg til MD4s tre runder, i tillegg til enkelte andre sikkerhetsøkende grep.

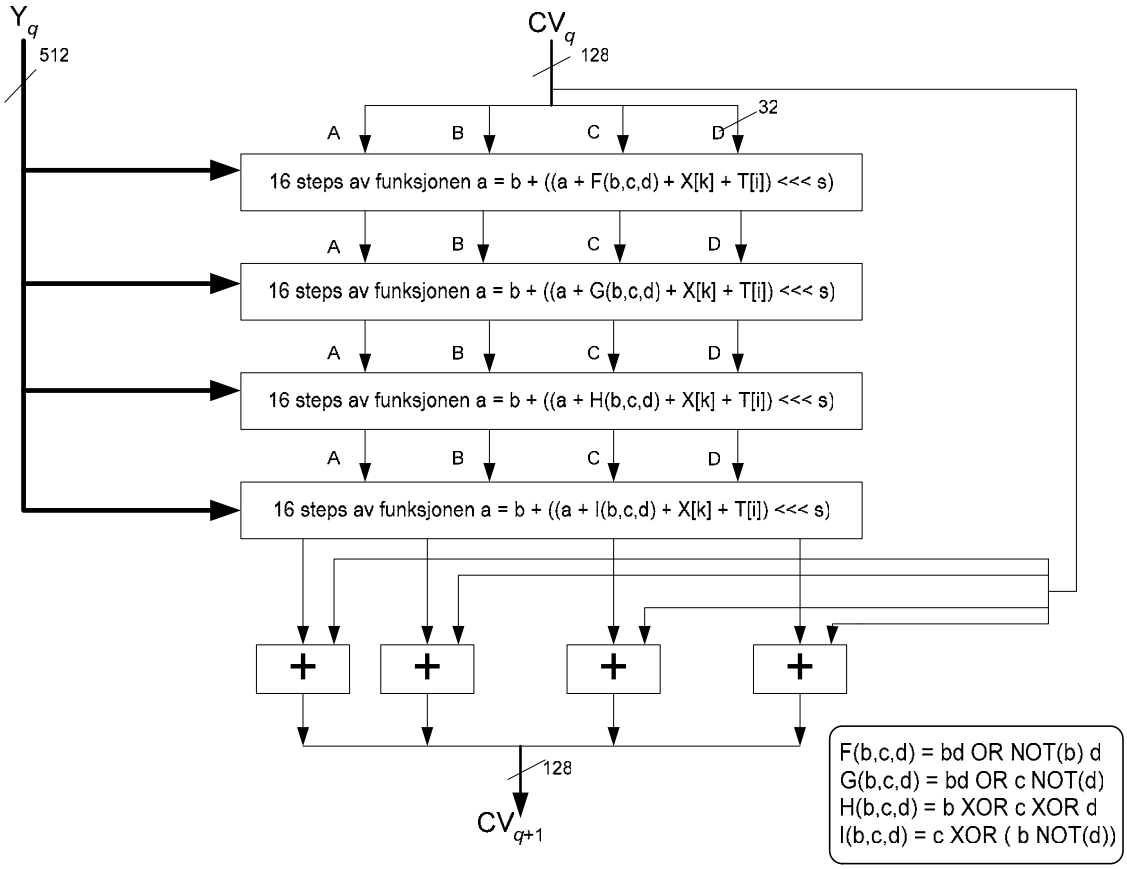
En melding som skal hashes med MD4 eller MD5 blir først utvidet slik at den er 64 bits fra at 512 kan dele antall bits (dvs meldingen blir utvidet slik at den har lengde kongruent med 448 modulo 512). Disse 64 bits benyttes til å legge til en binærrepresentasjon av meldingens lengde. Hver 512 bits blokk av meldingen blir så prosessert i en Damgård/Merkle iterativ struktur, hvor hver blokk igjen går tre (MD4) eller fire (MD5) algoritmiske runder.



Figur 1 - Damgård/Merkle iterativ struktur for hashfunksjoner

Figuren viser hvordan Darmgård og Merkles iterative struktur deler opp en melding i blokker som hver blir komprimert. Outputverdien fra komprimeringsverdien blir tatt med som input-data til neste komprimeringsfunksjon. Den siste blokken i meldingen blir utvidet slik at den fyller nøyaktig ut blokklengden.

Det som er spesielt med MD4 og etterfølgende hashfunksjoner i forhold til tidligere funksjoner er de algoritmiske rundene som utføres på hver enkelt blokk av meldingen. MD4 kjører blokken gjennom en rekke logiske bit-operasjoner med resultatet fra foregående blokk, som AND, OR, XOR og NOT, dessuten bitvis venstreskifting av verdier.



Figur 2 - Komprimeringsfunksjonen til MD5

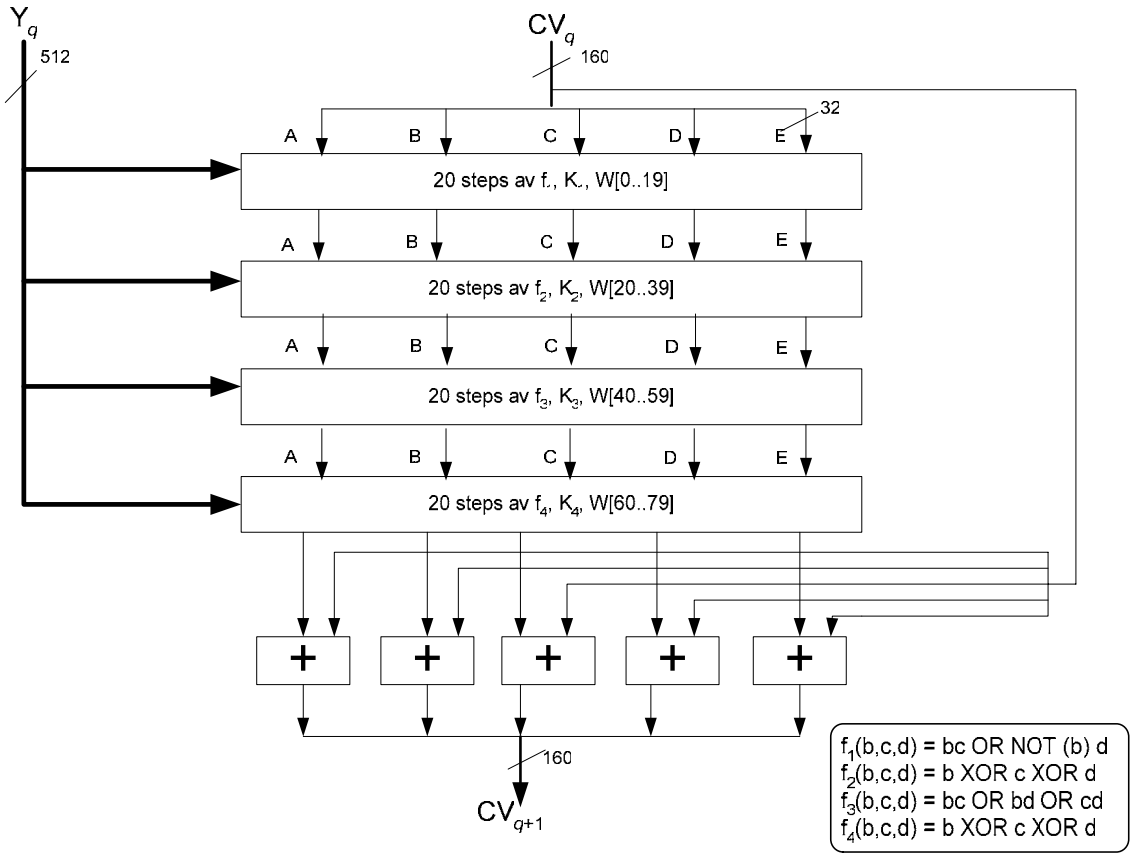
Figur 2 - Komprimeringsfunksjonen til MD5 [Rivest92, Stallings03] viser hvordan komprimeringsfunksjonen virker i detalj, altså funksjonen *F* i *Figur 1 - Damgård/Merkle iterativ struktur for hashfunksjoner*. Her blir en 512 bits blokk av input-strengen (Y_q) komprimert til en 128 bits output (CV_{q+1}). Denne 128 bits verdien tas med som initialverdi for å komprimere neste 512 bits blokk av input-strengen. Ved $q = 0$ er CV gitt en spesifisert initialverdi. MD5 består som tidligere nevnt av 4 runder. Hver runde er representert på figuren som et rektangel. Hver runde består av 16 operasjoner av typen som beskrevet i rektanglet. Variablene *a*, *b*, *c* og *d* endrer innhold slik at *A*, *B*, *C* og *D* alle har vært *a* (dvs. vært på venstresiden av likhetstegnet i formelen som er vist i rektanglene). Variablene *k*, *i* og *s* har varierende verdi i de forskjellige stegene. Alle addisjonsoperasjoner foregår i modulo 2^{32}.

Sikkerhetseksperter advarer nå mot å bruke MD5 i nye applikasjoner, da Hans Dobbertin har funnet kollisjoner i komprimeringsalgoritmen til MD5 som setter MD5s styrke som kollisjonsfri hashfunksjon i sterk tvil [Dobbertin96].

5.3 SHA-1 og RIPEMD-160

SHA-1 står for US Secure Hash Algorithm 1, og er bygget på Ronald Rivests MD5 og er laget for bruk av USAs myndigheter. Algoritmen har lenge vært tilgjengelig og ble i 2001 gjort publisert via RFC-3174 [Eastlake01], basert på informasjon fra FIPS 180-1 (Federal Information Processing Standard, 1993-1994). SHA-1 er altså heller ingen flunkende ny hash-funksjon. Til forskjell fra MD4 og MD5 som genererer en 128 bits output, genererer SHA-1 en 160 bits output. I 1991 da MD5 ble utviklet av Rivest, var 128 bits nok til at man ikke på en praktisk måte kunne finne en melding ut ifra hashen, dvs. at hashfunksjonen var enveis, praktisk sett. Etter hvert som vi har gått inn i det nye millenniet, og maskiners hastighet dobles hver 17. måned er ikke 128 bits lenger nok for å sikre enveis hashing.

SHA-1 er som MD4 og MD5 basert på en Damgård/Merkles iterativ struktur, og en rekke logiske bit-operasjoner og skiftoperasjoner.



Figur 3 – Komprimeringsfunksjonen til SHA-1

Som man ser av *Figur 3 – Komprimeringsfunksjonen til SHA-1* [Eastlake01, Stallings03] er SHA-1s komprimeringsfunksjon forholdsvis lik MD5s. Ved første iterasjon består *CV* av fem 32 bits forhåndsdefinerte variabler. Hver iterasjon komprimerer en 512 bits blokk av inputstrengen (Y_q) komprimert til en 160 bits output (CV_{q+1}), og slik som i MD5 blir denne outputen tatt med som initialverdi ved komprimering av neste 512 bits blokk av inputstrengen. Variablene *a*, *b*, *c*, *d* og *e* endrer innhold på en slik måte at verdien i *A*, *B*, *C*, *D* og *E* forflyttes en plass mot høyre etter hver operasjon. Hovedløkken har fire runder, hver med 20 operasjoner. Hver operasjon utfører en ikkelineær funksjon (f_n) på tre av *A*, *B*, *C*, *D* og *E*. K_n er en forhåndsdefinert konstant i [Eastlake01] som adderes modulo 2^{32} til *e* i hvert steg sammen med en verdi (*W*) fra en funksjon som forandrer meldingen fra seksten 32 bits ord (Y_q) til åtti 32 bits ord ved å bruke en bestemt algoritme.

RIPEMD-160 [Dobbertin96b] ligner også en del på MD5. Den består av fem algoritmiske runder, som hver inneholder både permuteringer, bit-operasjoner og skiftoperasjoner, men til forskjell fra MD5 kjøres det to 16-stegs operasjoner parallelt i hver av de fem rundene. RIPEMD-160 ble utviklet av blant andre Hans Dobbertin som fant kollisjoner i MD5s komprimeringsalgoritme. Som SHA-1 er hashverdien til RIPEMD-160 160 bits, som navnet tilsier.

6 SHABEIST

6.1 Introduksjon

Som en del av prosjektet har vi laget en hashfunksjon for å anskueliggjøre problematikken man støter på ved utvikling av en kryptografisk hashfunksjon. Under utvikling av hashfunksjonen har ikke hensikten vært å utvikle en erstatter for dagens gode hashfunksjoner. Dette har ikke vi nok erfaring eller lærdom til. Hensikten med å utvikle vår egen kryptografiske hashfunksjon er å få bedre innsikt i hvordan moderne hashfunksjoner basert på Damgård og Merkles iterative struktur[2] virker.

Vi har valgt å utvikle hashfunksjonen i Java, åpenbart ikke på grunn av Javas ytelse som i de fleste tilfeller er noe begredelig, men fordi det er et objektorientert språk som gir veldig god oversikt og er lett å jobbe med. Vi har ikke som mål å utvikle en revolusjonerende ny type hashfunksjon, og støtter oss derfor mye på Damgård og Merkles iterative struktur og noe på SHA-1 [Eastlake01].

Gjennom dette arbeidet har vi fått en mer grunnleggende forståelse for hvordan hashfunksjoner virker og hvordan man får oppfylt krav om *preimage* og *second preimage resistance*, dessuten komprimering.

6.2 Hva er SHABEIST?

Vi har kalt hashfunksjonen vi har utviklet for SHABEIST da den er løst basert på informasjon funnet i RFC-3174 [Eastlake01] og var et beist å få til å fungere slik vi ville. SHABEIST tar en streng av vilkårlig lengde, og komprimerer denne til en 96 bits streng. Inputstrengen blir delt opp i blokker på 128 bits. Hver blokk blir deretter prosessert (komprimert) for seg, i tråd med Damgård og Merkles iterative struktur. Inputstrengen vil alltid bli utvidet slik at den er delbar med blokkstørrelsen[3]. Resultatet av kompresjonen blir alltid tatt med til neste iterasjon.

Under vises koden som går igjennom blokkene og kjører komprimeringsfunksjonen på hver blokk (`hashTransform`). Dette gjøres før inputstrengen er utvidet til å være delbar på blokkstørrelsen. Derfor vil det her kjøres gjennom én mindre komprimeringsiterasjon enn det som er nødvendig.

```
    while (length >= blockSize / 8) {
      for (int i = 0; i < blockSize / 8; i++) {
        info.data[i] = clearText[offset + i];
      }
      hashTransform(info);
      offset += blockSize / 8;
      length -= blockSize / 8;
    }
```

De resterende bytene i inputstrengen blir deretter lest inn i `data` for senere å bli utvidet med følgende kode.

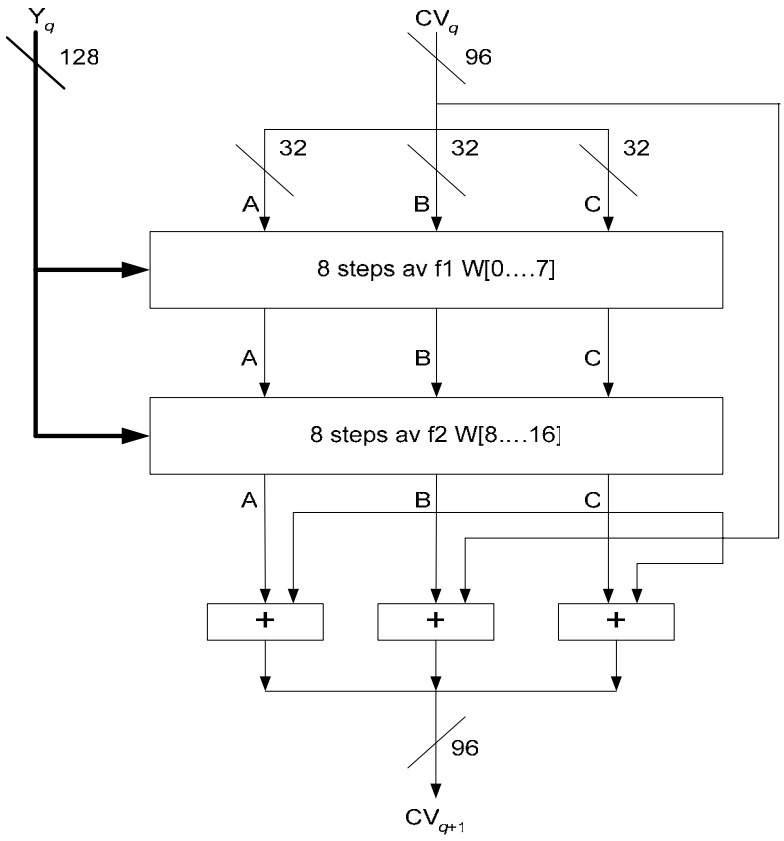
[2] se Figur 1 - Damgård/Merkle iterativ struktur for hashfunksjoner
[3] dvs. padding

```
    info.data[count++] = (byte) 0x80;

    for (int i = 0; i < (blockSize / 8) - count; i++) {
      info.data[count + i] = 0;
    }
```

Jfr. RFC-3174 (SHA-1) utvides blokken først med 0x80 som er bokstaven ”1”, deretter nok ”0” til å fylle blokken. Helhetlig sett er nå meldingens lengde kongruent med 0 i modulo 128. Legg merke til at siden vi **alltid** legger til 0x80, så vil alltid meldingen bli utvidet, selv om lengden til å begynne med var kongruent med 0 i modulo 128. Etter at denne siste resten av meldingen er korrekt utvidet, kjøres komprimeringsfunksjonen på denne blokken.

Under vises en figur av SHABEIST's komprimeringsfunksjon.



Figur 4 - Komprimeringsfunksjonen til SHABEIST

I hver iterasjon blir blokken (Y_q) kjørt gjennom to lineære funksjoner, og resultatet av disse funksjonene blir addert modulo 2^{32} til komprimeringsverdiene fra forrige iterasjon.

Verdien av komprimeringsfunksjonene ligger alltid lagret i tre variabler i et objekt kalt `info`. Til å begynne med er variablene initialisert slik:

```
  info.digest[0] = 0x67452301;
  info.digest[1] = 0xefcdab89;
  info.digest[2] = 0x98badcfe;
```

Disse initialiseringsverdiene er hentet fra RFC-3174 (SHA-1) [Eastlake01]. I de påfølgende iterasjonene blir variablene i `info.digest` lagret unna i `A`, `B` og `C`. Alle komprimeringsfunksjonene jobber da på disse midlertidige variablene. Til slutt i hver komprimeringsfunksjon blir verdiene i `A`, `B` og `C` addert modulo 2^{32} til variablene i `info.digest`. Dette er markert med "+" i figuren.

Videre følger komprimeringsfunksjonene. Notasjonen som blir brukt i Java-koden vises i tabellen under.

|	Bitvis OR
~	Bitvis NOT (invers)
^	Bitvis XOR
&	Bitvis AND
<<	Bitvis skift mot venstre
>>>	Bitvis skift mot høyre uten fortegn (unsigned)

Tabell 3 - Syntaksforklaringer

Komprimeringsfunksjonene `f1` og `f2` som det vises til i *Figur 4 - Komprimeringsfunksjonen til SHABEIST* blir vist under.

```
private void f1(int i, int[] W) {
  int temp = S(5, A) + (B | (~C)) + W[i] + K1;
  C = S(7, B);
  B = A;
  A = temp;
}
```

`f1` er første transformeringsfunksjon i komprimeringsfunksjonen, og kjøres på de 64 første bits av blokken, dvs. `W[0]` til og med `W[7]`.

```
private void f2(int i, int[] W) {
  int temp = S(11, B) + (A ^ C) + W[i] + K2;
  C = S(10, A);
  B = A;
  A = temp;
}
```

`f2` er andre og siste transformeringsfunksjon i komprimeringsfunksjonen, og kjøres da på de resterende 64 bits i blokken, dvs. `W[8]` til og med `W[15]`. Funksjonen `S` som benyttes både i `f1` og `f2` vises under. Denne funksjonen er en sirkulær skiftfunksjon tatt fra RFC-3174. Grunnen til den litt vanskelige notasjonen er at vanlig bitvis venstreskift bare fyller på med 0'er på høyresiden etter hvert som den skifter mot venstre. Derfor må den resulterende verdien fra venstreskiftingen OR-es med verdiene som blir 'dyttet ut' for å få til et sirkulært venstreskift. Se kapittel 6.3.2.3 for en gjennomkjøring av denne funksjonen som bedre beskriver virkemåten.

```
private int S(int n, int x) {
  return (x << n) | (x >>> (32 - n));
}
```

Som nevnt over blir de midlertidige variablene addert til variablene inneholdt i `info.digest`. Dette gjøres her:

```
info.digest[0] += A;
info.digest[1] += B;
info.digest[2] += C;
```

Når alle iterasjonene er gjennomløpt vil hashverdien være verdien som består av `info.digest[0]`, `info.digest[1]` og `info.digest[2]`. Altså får man resultatet av hashfunksjonen ved hjelp av følgende kode:

```
private String hashPrint(HashInfo info) {
  String digest0, digest1, digest2;
  digest0 = Integer.toHexString(info.digest[0]);
  digest1 = Integer.toHexString(info.digest[1]);
  digest2 = Integer.toHexString(info.digest[2]);

  while(digest0.length() < 8) digest0 = "0" + digest0;
  while(digest1.length() < 8) digest1 = "0" + digest1;
  while(digest2.length() < 8) digest2 = "0" + digest2;

  return digest0 + digest1 + digest2;
}
```

Det koden over gjør, er å gjøre om de tre 32 bits integer-verdiene til en heksadesimal streng, som senere konkateneres. Hvis det er nødvendig, legges det til "0"-er for å ivareta hashverdiens korrekte 96 bits lengde.

For komplett, kompilerbar kode, vennligst se vedlegg A.

6.3 Eksempel på en gjennomkjøring i SHABEIST

Her følger en del av en gjennomkjøring gjennom SHABEIST. Eksempelet viser én gjennomkjøring gjennom f1 og én gjennomkjøring gjennom f2, før det til slutt viser resultatet. I tillegg vises det også en gjennomkjøring av den sirkulære skiftfunksjonen, `S`.

6.3.1 Startdata

Fra forrige iterasjon (CV_{q-1}) antar vi at vi har følgende verdier:

```
info.digest[0]: 10110100101111101011010110010100
info.digest[1]: 10100100010100100100001010110011
info.digest[2]: 00000101011111100001110110001111
```

Vi antar videre at vi har en klartekstblokk lagret i arrayet `info.data` som skal igjennom transformasjonen. Denne blokken ser slik ut:

```
 info.data[0]: 11001101010101101010100011110001
 info.data[1]: 00111001111110000011110101011110
 info.data[2]: 11010110010100110010100000100100
    ...                    ...
info.data[15]: 10101000101110100000101011110100
```

Altså, vi har 16 integer-verdier, hver på 32 bits. Så starter vi funksjonen `hashTransform` som komprimerer blokken.

6.3.2 Komprimeringsfunksjonen

Komprimeringen begynner med å kopiere ut de hashverdien som ble resultatet fra forrige iterasjon inn i tre midlertidige variable `A`, `B` og `C`. Dessuten kopieres også blokken inn i et annet array, `W`.

```
A ← info.digest[0] (10110100101111110101101011010100)
B ← info.digest[1] (10100100010100100100001010110011)
C ← info.digest[2] (00000101011111100001110110001111)

W ← info.data
```

Etter at dette har skjedd kjører funksjonen `f1` 8 ganger.

```
f1(0, W) f1(1, W) f1(2, W) ... f1(7, W)
```

Deretter kjøres funksjonen `f2` 8 ganger:

```
f2(8, W) f2(9, W) f2(10, W) ... f2(15, W)
```

6.3.2.1 Ikkelineære transformasjonsfunksjoner

Her kommer et talleksempel fra funksjonen `f1(2, W)`:

Først adderes følgende inn i den midlertidige 32 bits integeren `temp`:

- `S(5,A)`: Dette er skiftfunksjonen (se kapittel 6.3.2.3 for nærmere forklaring). Den skifter `A` 5 plasser til venstre
  `= 10010111110101101011001010010110`
- `B | ~C`: Dette er en bitvis OR operasjon mellom `B` og inversen til `C`

  ```
  B           = 10100100010100100100001010110011
  NOT C       = 11111010100000011110001001110000
  B OR NOT C  = 11111110110100111110001011110011
  ```

- `W[2]`: Dette er en verdi fra blokken. Vi leser fra plass nummer 2 i arrayet da dette ble gitt som parameter til funksjonen.
  `= 11010110010100110010100000100100`
- `K1`: Dette er en konstant (0x5a827999) som vi har tatt fra RFC-3174 (SHA-1)
  `= 01011010100000100111100110011001`

Innholdet i `temp` blir altså summen av overstående liste:

```
   S(5,A)  = 10010111110101101011001010010110
 + B | ~C  = 11111110110100111110001011110011
 + W[2]    = 11010110010100110010100000100100
 + K1      = 01011010100000100111100110011001
 = temp    = 11000111100000000011011101000110 (mod 2^32)
```

Deretter utføres følgende:

```
C ← S(7,B) (00101001001000010101100111010010)
B ← A      (10110100101111101011010110010100)
A ← temp   (11000111100000000011011101000110)
```

`f1` kjøres altså 8 ganger, med forskjellige aktive elementer i `W`.

Her følger et talleksempel fra funksjonen `f2(15, W)`. Legg merke til at før vi kommer hit i algoritmen vil `A`, `B` og `C` ha forandret seg, da `f1` og `f2` opererer direkte på disse variablene. Vi forutsetter da at `A`, `B` og `C` nå er som følger:

```
A: 11111101010001001010100000110001
B: 01001111110101101010100011000001
C: 11001101000101000000101011110000
```

Deretter adderes følgende inn i den midlertidige 32 bits integeren `temp`:

- `S(11,B)`: Dette er skiftfunksjonen. Den skifter `B` 11 plasser til venstre.
  `= 10110101010001100000101001111110`
- `A ^ C`: Dette er en bitvis XOR-operasjon mellom `A` og `C`.

```
A       = 11111101010001001010100000110001
C       = 11001101000101000000101011110000
A XOR C = 00110000010100001010001011000001
```

- `W[15]`: Dette er en verdi fra blokken. Vi leser fra plass nummer 15 i arrayet da dette ble gitt som parameter til funksjonen.
  `= 10101000101110100000101011110100`
- `K2`: Dette er en konstant (0x6ed9eba1) som vi har tatt fra RFC-3174 (SHA-1)
  `= 01101110110110011110101110100001`

Dermed blir `temp` summen av overstående:

```
 S(11,B)  = 10010010000101011001110100100010
+ A ^ C   = 10110001110000001010100000011011
+ W[15]   = 10101000101110100000101011110100
+ K2      = 01101110110110011110101110100001
= temp    = 01011011011010100011101111010010 (mod 2^32)
```

Deretter utføres følgende:

```
C ← S(10,A)(00010010101000001100011111110101)
B ← A      (11111101010001001010100000110001)
A ← temp   (01011011011010100011101111010010)
```

Dette var altså aller siste kjøring av `f2` for denne blokken.

6.3.2.2 Avslutning av iterasjonen

Etter at den siste blokken er igjennom transformasjonsfunksjonene, gjenstår kun å legge de nye hashverdiene, til de hashverdiene som ble funnet i forrige iterasjon.

```
info.digest[0] ← info.digest[0] + A
info.digest[1] ← info.digest[1] + B
info.digest[2] ← info.digest[2] + C
```

Deretter følger neste iterasjon i Damgård/Merkle-strukturen med neste blokk.

6.3.2.3 Forklaring av den sirkulære skiftfunksjonen

Vi viser her et talleksempel på det som skjer når den sirkulære skiftfunksjonen, `S`, blir kjørt. Eksemplet er funksjonen `S(5, B)` som er nevnt først i kapittel 6.3.2.1.

Vi har gitt `B` fra et tidligere eksempel:

```
B: 10100100010100100100001010110011
```

Funksjonen utfører først et bitvis venstreskift med 5 plasser:

```
i = B << 5 = 10001010010010000101011001100000
```

Vi ser at det som skjedde var at de 5 første bits forsvant ut til venstre, og det ble fylt på med tallet 0 bakerst. Etter at detter er gjort utføres et høyreskift slik at de 5 mest signifikante bitverdiene i `B` havner bakerst i tallet:

```
j = B >>> (32-5) = 00000000000000000000000000010100
```

Hvis vi nå benytter oss av en bitvis OR-operasjon på de to nye verdiene vi har fått etter høyre- og venstreskiftingene, vil vi se at vi får et sirkulært venstreskift (dvs. de fem første verdiene i `B` er nå de fem siste verdiene i den nye verdien som vi returnerer):

```
i | j = 10001010010010000101011001110100
```

6.4 Test av SHABEIST

For å utføre tester på SHABEIST har vi utviklet en applikasjon som lar en bruker kjøre flere forskjellige tester og angrep på SHABEIST. Noen av disse testene og angrepene er beskrevet under.

6.4.1 Permutasjonsangrep

Dette angrepet permuterer vilkårlige tekster av vilkårlig lengde, og ser om de har samme hashverdi som verdien til en gitt tekst. Dvs. at vi tester om hashfunksjonen er *preimage* og *second preimage resistance*. I våre tester har vi ikke funnet noen kollisjoner. Vi har forsøkt følgende strenger som hver har blitt permutert:

'abc', dvs. 6 (3!) mulige permutasjoner
'abcdefghi', dvs. 362 880 (9!) mulige permutasjoner
'abcdefghijkl', dvs. 479 001 600 (12!) mulige permutasjoner

Vi fant ingen kollisjoner da vi utførte testene over. Vi har ikke hatt mulighet til å teste lengre strenger, da slik prosesseringskraft ikke er tilgjengelig for oss. Vi erkjenner at dette ikke er godt

nok bevis for å si at SHABEIST er kollisjonsfri eller er en enveis hashfunksjon (OWHF), men det er åpenbart at SHABEIST har en viss *preimage* og *second preimage resistance*.

6.4.2 Meldinger av varierende lengde

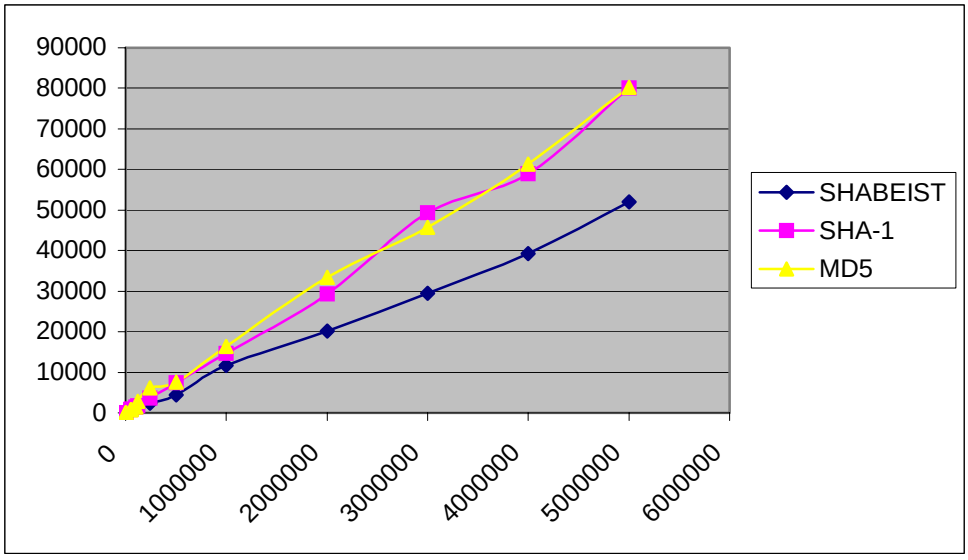
For å teste våre funksjon og hvordan tidsforbruket økte med størrelsen på inputen, kjørte vi en rekke tester på våres hashfunksjon med strenger på forskjellig størrelse. For å få noe sammenligningsgrunnlag mot andre kjente hashfunksjoner, kjørte vi samme test på en SHA-1 og MD5 implementasjon i Java. Tabellen under viser resultatene vi fikk når vi kjørte testen på en 1.2GHz AMD Duron maskin.

Test på maskin 1 (1.2GHz AMD Duron CPU):
Tiden for gjennomkjøring er oppgitt i millisekund, og hver input blir hashet 100 ganger for å få mer nøyaktig resultat.

Inputlengde	SHABEIST	SHA-1	MD5
100	10	10	10
1000	10	20	20
5000	51	70	70
50000	471	731	711
60000	530	882	801
100000	891	1462	1482
120000	1222	1753	2824
240000	2333	3575	6169
500000	4447	7350	7591
1000000	11656	14681	16424
2000000	20109	29392	33389
3000000	29503	49373	45766
4000000	39237	59005	61348
5000000	52015	80125	80295

Tabell 4 - Test av input på AMD Duron 1.2GHz

Basert på resultatene vi fikk i *Tabell 4 - Test av input på AMD Duron 1.2GHz*, laget vi en graf over resultatet. På grafen viser x-aksen inputlengde, og y-aksen viser tiden det tok å beregne hashverdien hundre ganger målt i millisekunder.



Figur 5 – Graf for test av input på AMD Duron 1.2GHz

Ut fra grafene kan lese ut følgende:

- Det ser ut til at alle tre hashfunksjonene er lineære med hensyn på tid. Tidsforbruket øker med andre ord lineært med størrelsen på strengen.
- SHABEIST er raskere til å generere hashverdien av inputstrengen enn MD5 og SHA-1. Dette kommer sannsynligvis hovedsaklig av at SHABEIST kun kjører to runder med operasjoner, mens MD5 og SHA-1 kjører fire runder med operasjoner. Hastigheten kan selvfølgelig variere fra implementasjon til implementasjon av SHA-1 og MD5.
- MD5 og SHA-1 funksjonene ser ut til å være ca. like raske til å generere hashverdien av en input. Dette kan være verdt å merke seg siden MD5 genererer en 128 bits hashverdi, mens SHA-1 genererer en 160 bits hashverdi, og benytter seg da av en ekstra intern variabel og flere funksjonskall til transformasjonsfunksjoner.

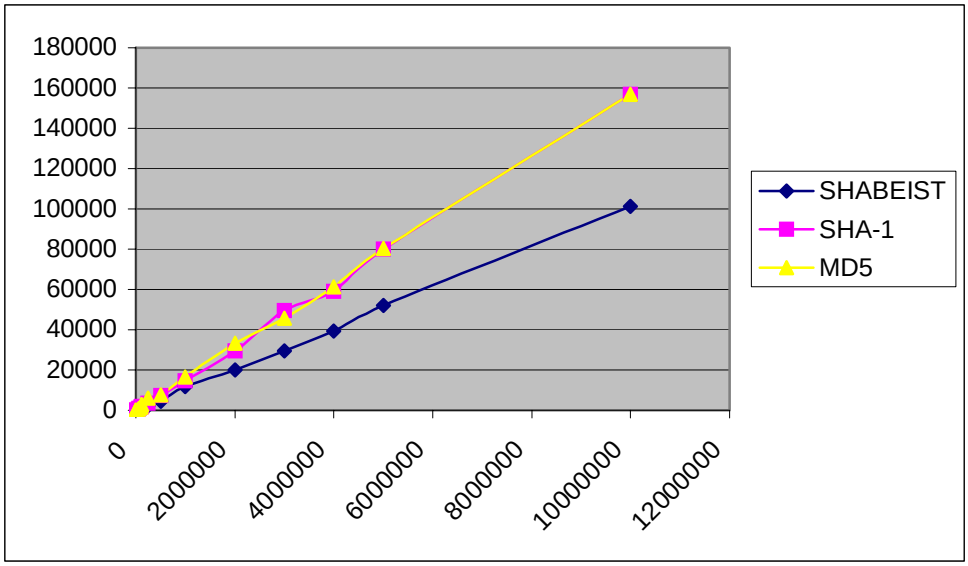
For å videre analysere forskjellen på tidsbruk mellom funksjonene, fant vi stigningstallet til hver av grafene og genererte et matematisk beregnet resultat på en stor inputstreng.

	Inputlengde	SHABEIST	SHA-1	MD5
	100	10	10	10
	1000	10	20	20
	....	....	....	....
	....	....	....	....
	5000000	52015	80125	80295
Stigningstall:		0,010139	0,015672	0,015679

Tabell 5 – Test på maskin 1 med beregnet stigningstall

Hvis vi forutsetter at hashfunksjonenes tidsbruk er en lineær funksjon, kan vi på bakgrunn av de stigningstallene vi nå har funnet kalkulere tidsbruken for å hashe en streng på lengde 10 000 000 bokstaver 100 ganger. Denne vil være ca 101s for SHABEIST, 157s for SHA-1 og 157s for MD5.

Basert på de nye resultatene laget vi en ny graf som også inneholdt de matematiske beregnede verdiene.



Figur 6 – Forsøk med beregnet stort resultat

For å beregne hvor mye raskere SHABEIST er i forhold til MD5 og SHA-1 gjør vi følgende beregninger:

Først finner vi gjennomsnittstiden for MD5 og SHA-1, som er ((156 716 + 156 792) / 2) = 156 754. Deretter finner vi at vår funksjon er 1 - (101 388/156 754) ≈ 35% raskere enn gjennomsnittet av MD5 og SHA-1.

Dette resultatet forteller oss at funksjonen vår er i snitt ca. 35% raskere enn de implementasjonene av SHA-1 og MD5 vi testet mot. Utover dette sier resultatet oss ikke noe om forskjellen i sikkerhet med hensyn på antall kollisjoner. Men vi kan antakelig anta at vår funksjon inneholder flere kollisjoner enn de andre, siden den hashede verdien av resultatet her er 96 bits, og at funksjonen bare kjører to runder med operasjoner mot MD5 og SHA-1 sine fire runder med operasjoner.

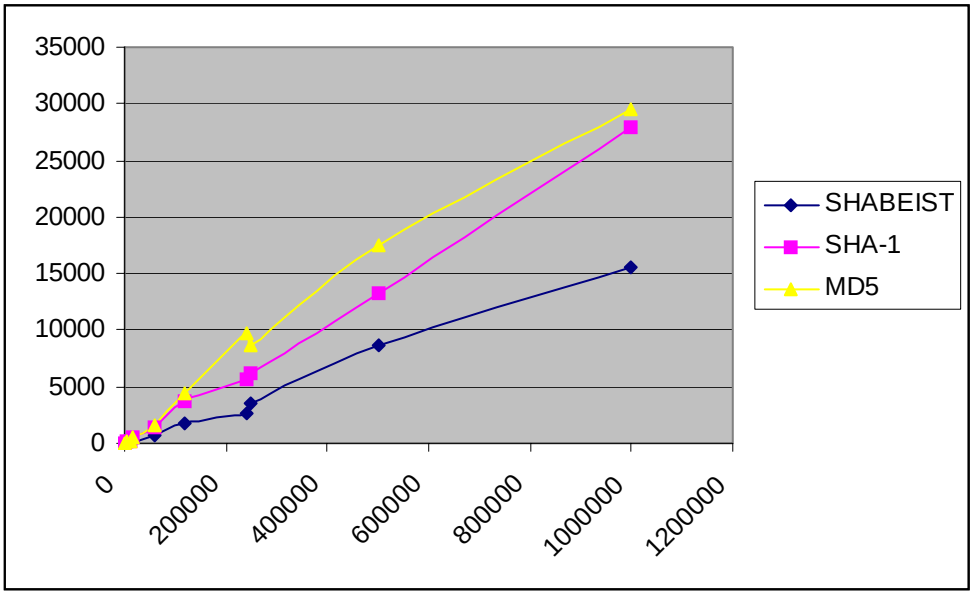
Vi testet også hvordan resultatene av de samme testene med forskjellig inputstreng ga utslag på forskjellige datamaskiner. Resultatene er listet opp under. Resultatene er ikke kommentert ut over hva leseren selv ønsker å trekke ut av tabellene og grafene, siden det ser ut til at vi kan trekke samme slutninger ut fra alle testene.

Test på maskin 2 (1.6GHz Intel Pentium 4 CPU):

Tiden for gjennomkjøring er oppgitt i millisekund, og hver input blir hashet 100 ganger for å få mer nøyaktig resultat.

Inputlengde	SHABEIST	SHA-1	MD5
1000	16	31	47
5000	94	140	188
10000	125	250	265
15000	234	531	469
60000	750	1422	1516
120000	1750	3750	4407
240000	2687	5657	9657
250000	3500	6234	8750
500000	8719	13281	17515
1000000	15469	27905	29531

Tabell 6 - Test av input på Intel Pentium 4 1.6GHz

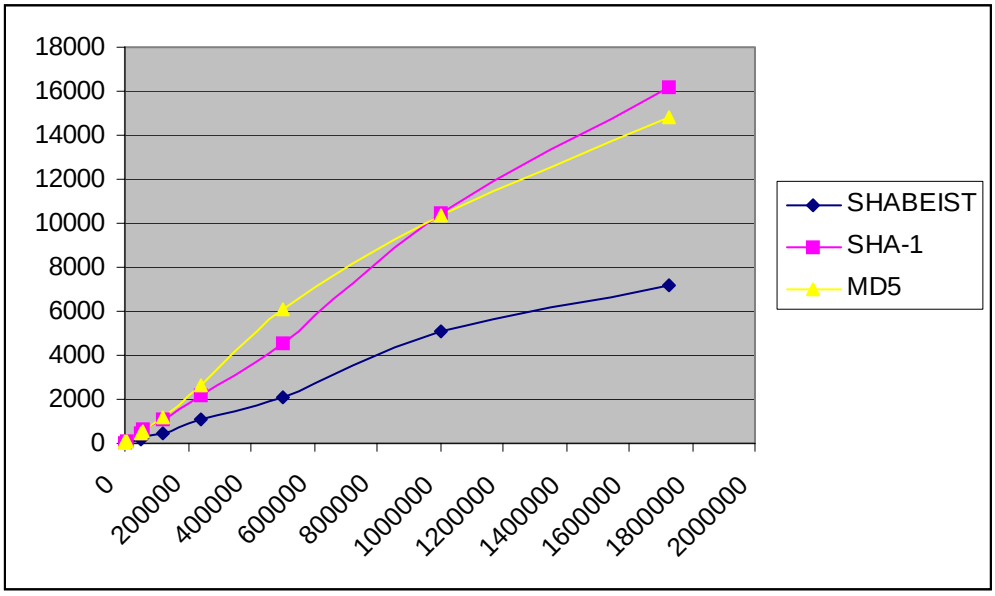


Figur 7- Graf for test av input på Intel Pentium 4 1.6GHz

Test på maskin 3 (2.8GHz Intel Pentium 4 CPU):
Tiden for gjennomkjøring er oppgitt i millisekund, og hver input blir hashet 100 ganger for å få mer nøyaktig resultat.

Inputlengde	SHABEIST	SHA-1	MD5
100	15	16	16
1000	15	32	47
5000	15	47	63
50000	218	438	453
60000	250	609	531
120000	500	1079	1203
240000	1078	2141	2640
500000	2109	4516	6047
1000000	5062	10469	10390
1728000	7172	16172	14860

Tabell 7 - Test av input på Intel Pentium 4 2.8GHz



Figur 8- Graf for test av input på Intel Pentium 4 2.8GHz

6.4.3 Permutasjoner av pseudorandomgenererte meldinger

Da vi ønsket å se den relative hyppigheten av kollisjoner i SHABEIST benyttet vi oss av en permutasjonsalgoritme for å generere alle mulige permutasjoner av en streng av gitt lengde som ble generert ved hjelp av en pseudorandom-algoritme. Deretter testet vi alle permutasjonene mot hverandre for å finne kollisjoner i de 16 minst signifikante bits av hashverdien. Dette er det samme som å kjøre en bitvis AND-operasjon på hashverdien og den heksadesimale verdien `FFFF`. Vi valgte å kun teste mot 16 bits da det ville være urealistisk å tro at vi ville finne kollisjoner med dagens maskinvare hvis vi testet mot hele hashverdien. Da vi testet permutasjoner av forskjellige strenger med lengde på 50 bokstaver, måtte vi i de 21 testene vi utførte gå så mange runder som vist under.

122647	5669	29459	211010	128194	100406	20259
114649	27572	49325	5966	19943	363865	161188
8057	759699	18600	177600	46469	17319	113077

Max = 759 699 – Min = 5 669 – Gjennomsnitt = 125 048,65

Vi ser at hvis man vil forsøke å finne lik hashverdi ved hjelp av to tilfeldig valgte strenger av lik lengde, så er det 50% sannsynlig at man må gjøre minst 125 000 forsøk bare for å finne kollisjoner i de 16 minst signifikante bits. Merk at dette angrepet skiller seg fra fødselsdagsangrep ved at man her ikke prøver å lage par av to sett med $2^{n/2}$ strenger.

6.4.4 Fødselsdagsangrep

Fødselsdagsparadokset er godt kjent innenfor kryptografi-miljøet. Vi har derfor valgt å forsøke oss på en implementasjon av angrepet, for å se om det var langt mellom liv og lære. Hovedsakelig gjør vi dette for å forstå hvordan man i praktisk kan utføre dette angrepet, og for å bedre skjønne teorien vi har vært igjennom vedrørende dette angrepet.

For å klare å teste dette i praksis ble vi nødt til å trunkere hashverdiene vi benyttet slik at vi arbeidet på 16 bits hashverdier på samme måte som vi gjorde i kapittel 6.4.3. Framgangsmåten er som følger:

1. Vi lagde $2^{16/2} = 256$ tilfeldige strenger av tilfeldig lengde, og beregnet hashverdiene av disse.
2. Vi lagde så $2^{16/2} = 256$ nye tilfeldige strenger av tilfeldig lengde, og beregnet hashverdiene av disse også.
3. Vi prøvde deretter å danne et matchende par fra de to settene med hashede strenger.

Ifølge teorien skal vi med mer enn 50% sannsynlighet kunne finne ett par hashverdier fra disse to settene som er identiske. Våre tester viste at fødselsdagsparadokset stemte. Etter 1000 forsøk fant vi kollisjon i 62.1% av testene vi utførte.

Dersom vi skulle ha kjørt et fødselsdagsangrep på hashverdier som ikke var trunkerte, dvs. på 96 bits hashverdier, ville vi måttet generere $2 \times 2^{96/2} = 562\ 949\ 953\ 421\ 312$ strenger. Dette ville åpenbart tatt for lang tid for en hvilken som helst datamaskin i dag.

7 Konklusjon

I løpet av prosjektperioden har vi nå fordypet oss i kryptografiske hashfunksjoner. Vi synes vi har lært mye og fått en bedre forståelse av hva kryptografiske hashfunksjoner er og hva de blir bruk til. Ved å implementere vår egen hashfunksjon fra bunnen av har vi lært oss på en mer praktisk måte hvordan en hashfunksjon blir laget og fungerer. Dette var vårt første møte med implementasjon av kryptografiske hashfunksjoner, og vi måtte derfor tilegne oss en del kunnskap om emnet før vi kunne gå i gang med koding. Vi fant fort ut at å implementere en god hashfunksjon ikke var så enkelt som vi hadde trodd da vi startet på oppgaven. Det var ikke bare å ta en verdi og kjøre diverse bit-operasjoner på denne, og tro at dette var en god hashfunksjon. Det ble derfor utviklet noen mislykkede hashfunksjoner før den endelige SHABEIST så dagens lys og vi kunne kjøre i gang med tester på denne.

En kombinasjon av litteraturstudier og koding har gitt oss en bred forståelse på hvordan hashfunksjoner blir testet og angrepet, og hvilke bruksområder de har. Vi har sett at teorien bak forskjellige angrep kan være meget vanskelig å implementere i praksis. Vi har sett at angrep på hashfunksjoner ofte krever beregningsmessig og tidsmessig mye mer kapasitet enn vi har hatt tilgang til under prosjektet. Vi kunne nok fått en mer effektiv hashfunksjon og et litt bedre resultat på testingen ved å ha valgt et annet implementasjonsspråk, som for eksempel C, men dette ville vært marginalt og ikke verdt å ta hensyn til. Derfor valgte vi det mer oversiktlige språket Java for implementasjon. På denne måten fikk vi frisket opp programmerings-kunnskapene i Java og objektorientert programmering samtidig som vi lærte om implementasjon av hashfunksjoner.

```
PS: Hashverdien av dette dokumentet (minus selve hashverdien)
                    hashet med SHABEIST er:

                    05d409cd1fabb68710ff8b8a
```

8 Referanser

Damgård88 I. B. Damgård, *Collision free hash functions and public key signature schemes,* EUROCRYPT '87, Springer-Verlag 1988

Dobbertin96a Hans Dobbertin, *The Status of MD5 After a Recent Attack* ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf

Dobbertin96b Hans Dobbertin, Antoon Bosselaers & Bart Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD* http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf

Eastlake01 Donald Eastlake, *RFC-3174: US Secure Hash Algorithm 1 (SHA-1)* http://www.ietf.org/rfc/rfc3174.txt

Menezes96 Alfred J. Menezes, Paul C. van Oorschot & Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press 1996 http://www.cacr.math.uwaterloo.ca/hac/

Moore65 Gordon E. Moore, *Cramming More Components Into Integrated Circuits*, ftp://download.intel.com/research/silicon/moorespaper.pdf

Rivest90 Ronald Rivest, *RFC-1186: The MD4 Message Digest Algorithm* http://www.ietf.org/rfc/rfc1186.txt

Rivest92 Ronald Rivest, *RFC-1321: The MD5 Message Digest Algorithm* http://www.ietf.org/rfc/rfc1321.txt

VEDLEGG A

Kildekode for SHABEIST med tilhørende testverktøy

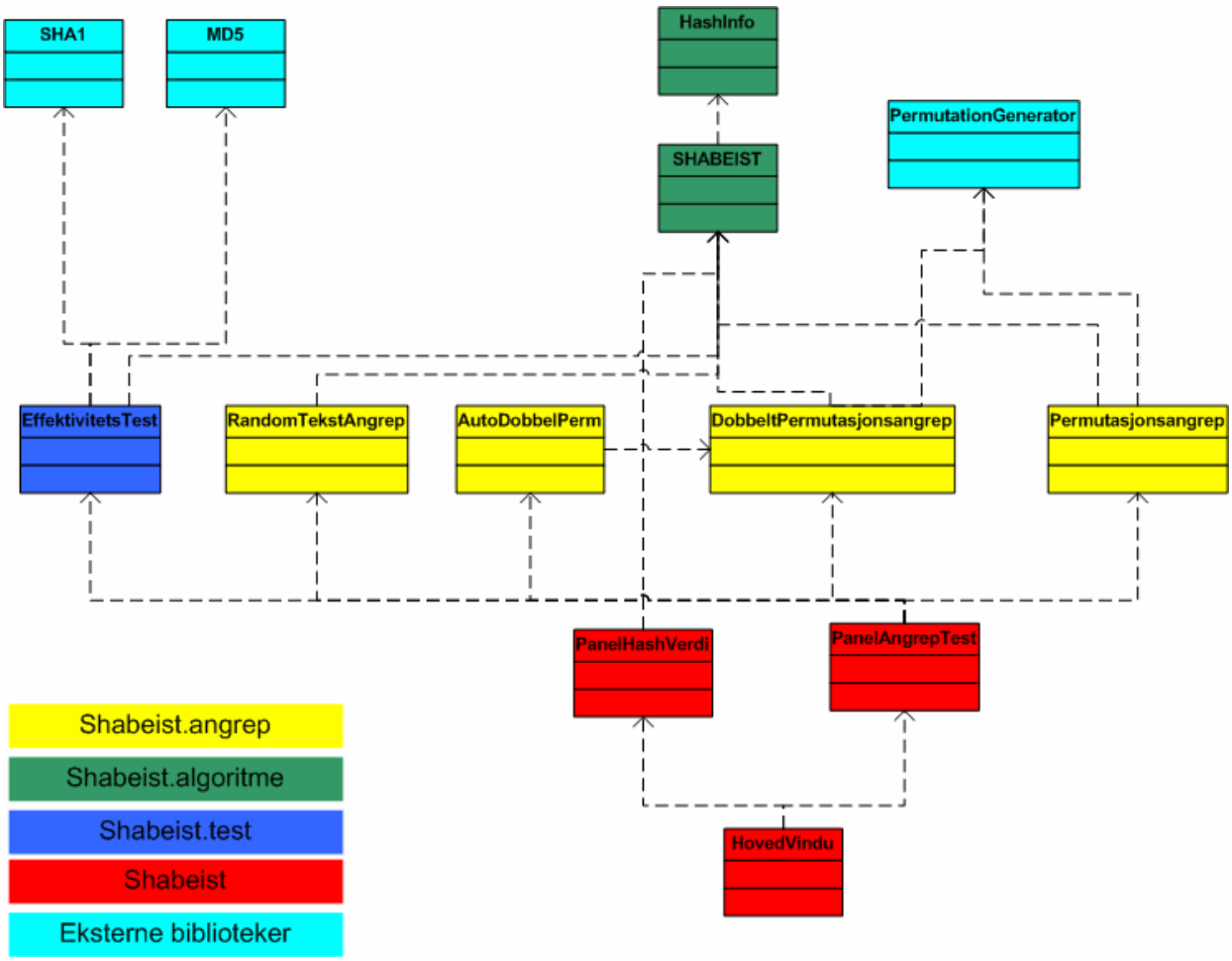
1 Introduksjon

Heretter følger kildekoden til implementasjonen av SHABEIST. Koden er delt opp i pakker, som hver inneholder klasser. Vi benytter oss av noen eksterne kodebiblioteker i koden for angrep og testing:

Kodebibliotek	**Forfatter**
MD5-implementasjon i Java	Santeri Paavolainen, http://www.iki.fi/santtu/programs/md5/
SHA-1-implementasjon i Java	Chuck McManis, http://jabberapplet.jabberstudio.org/
Permutasjonsgenerator	Michael Gilleland, http://www.merriampark.com/perm.htm

2 Oversikt over klassene

Under følger et klassediagram over klassene som inngår i SHABEIST og testverktøyet. Figuren viser de forkjellige klassene vi implementerte. Koblingene mellom klassene viser hvilke klasser som er avhengige av hvilke klasser. A → B betyr at klassen A er avhengig av klassen B, dvs. at det opprettes ett eller flere objekter av klassen B i A.



3 Kompilering av kildekoden

Hvis det er ønskelig å kompilere kildekoden som følger med på CD-ROM, kreves følgende:

- Borland JBuilder (koden er kompilerer på versjon 8 og versjon 9)
- JabberApplet installert som et "Library" i JBuilder med navnet "JabberApplet". Nødvendige kompilerte klasser kan lastes ned fra http://jabberapplet.jabberstudio.org/.