Studieretningsprojekt – HTX Roskilde 2017

HTX → Roskilde

Emne/område: Kryptografi
Opgaveformulering: Redegør for forskellige principper for kryptering og kom især ind på den asymmetriske metode. Redegør for RSA-kryptering og i den forbindelse primtalsfaktorisering, modulus, Fermats lille sætning og Eulers sætning. Forklar principperne bag hashing, kom ind på anvendelsesmulighederne og giv eksempler på nogle hash-funktioner. Lav et program som anvender en SHA-algoritme til hashing og vis konsekvensen af små ændringer i input. Vurdér sikkerheden ved anvendelsen af hashing. Diskutér anvendelsesmulighederne for henholdsvis asymmetrisk kryptering og hashing nu og i fremtiden. Projektet skal have et engelsk resumé på 15-20 linjer og forventes at have et omfang på ca. 12-15 normalsider á 2.500 anslag incl. mellemrum og ekskl. forside, indholdsfortegnelse og bilag.

0 Abstract

This paper examines security through encryption and hashing within the science of cryptology. It illustrates how data can be considered secure in different scenarios, and how different approaches are taken depending on the specific need.

Initially, the study explains different principles behind the topic of encryption, and how they have evolved over time. It focuses on asymmetric encryption, also known as Public Key Encryption. As a part of this, the RSA algorithm is shown. Furthermore, the mathematical theories behind it, such as prime factorization, modular arithmetic and multiple theorems, are analyzed.

The paper goes on to explain the principles behind hashing functions, discussing where and why they are used instead of encryption algorithms. This is visualized with a thorough explanation and custom implementation of the SHA1 algorithm.

Finally, it is discussed how the seemingly endless growth in computing power affects the security of these algorithms and what measures are to be taken to counteract this growth.

It is concluded that hashing functions and encryption algorithms both have their place in today's world and that each can be utilized for specific needs. Furthermore, it is concluded that both the RSA and SHA algorithms have *built in* ways to counteract the growth of computing power. However, it comes at a cost of efficiency.

Indholdsfortegnelse

1 Indledning

I nutidens informationssamfund er hele verden forbundet. Kommunikation kan ske når som helst og hvor som helst. Hvis vi ikke tog visse forholdsregler, ville kommunikation mellem to parter dog være tilgængelig for enhver, der havde lyst til at lytte med. Det ville oven i købet være muligt for en tredjepart at *ændre* hvad der blev kommunikeret, som det senere vil blive vist.

Nødvendigheden af *pålidelig*, *sikker* og *privat* kommunikation har skabt det videnskabelige felt, man kalder *kryptologi*, læren om hemmeligholdelse af information.[1] Det er ikke noget nyt felt. Især behovet for privat kommunikation har altid været der. Om det så var soldater i krig, der ville sende hemmelige beskeder tilbage til basen eller om det blot var to elskende, der ikke ville opdages af deres forældre.

Udviklingen af kommunikation over store distancer har derudover åbnet op for den anden udfordring - pålidelig information. Her menes at man kan stole på det, der står i en besked. Igen bruges eksemplet om en soldat, der har sendt en besked tilbage til sin base. I beskeden står at fjenden er i overtal, og at krigen derfor ikke kan vindes. Men kan man stole på at det er en allieret soldat, der har skrevet det?

Der er flere forskellige metoder til at *sikre* sin data - hver af dem har sine fordele og ulemper, og valget af metode kommer også til at afhænge af hvad *sikker* betyder i den specifikke sammenhæng.

2 Kryptering

Fra ordnet.dk fås det at ordet *kryptere* har følgende betydning: *"omsætte oplysninger eller data til en hemmelig kode, især for at forhindre uvedkommende i at få adgang til dem"*.[2]

Det kan siges at kryptering går ud på at skjule informationer fra uvedkommende. Et meget jordnært eksempel er at man som dansker kan snakke dansk i udlandet, hvis man ikke vil have at de lokale skal kunne forstå hvad man siger. Dette overholder, til en hvis grad, også ordets betydning, idet vi omsætter til en *kode* (i dette tilfælde det danske sprog). Hermed kan uvedkommende ikke følge med i samtalen. Det eneste man kan sætte spørgsmålstegn ved her er hvor *hemmelig* en kode det er - dansk snakkes trods alt af over 5 millioner mennesker.[3]

Fælles for alle krypteringsalgoritmer er at de omsætter en såkaldt "plain text" (klartekst, den oprindelige tekst) til en "ciphertext" (ciffertekst, den krypterede tekst). Man kan opskrive en krypteringsproces på følgende måde.[4]

[1] Kryptologi, Wikipedia. Fundet 18/12/2017 på https://da.wikipedia.org/wiki/Kryptologi
[2] kryptere, Den Danske Ordbog. Fundet 14/12/2017 på http://ordnet.dk/ddo/ordbog?query=krypteret
[3] Arndt, Hans. Danish, AU Research. Fundet 14/12/2017 på http://www.hum.au.dk/ling/research/Danish.htm
[4] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 37

$$c = e_k(m)$$

Hvor:

- **c** er cifferteksten
- **e** er krypteringsfunktionen
- **k** er krypteringsnøglen
- **m** er klarteksten

Dekrypteringsprocessen kan således skrives på følgende måde:

$$m = d_k(c)$$

Hvor **d** er dekrypteringsmetoden (den inverse krypteringsfunktion).

2.1 Cæsaralgoritmen

En af de tidligste udbredte krypteringsalgoritmer var Cæsaralgoritmen. Som navnet antyder, blev den blandt andre brugt af Julius Cæsar.[5] Algoritmen går ud på at ethvert bogstav i klarteksten substitueres med et andet k pladser længere henne i alfabetet. k fungerer således som krypteringsnøgle og kan, når man krypterer beskeder skrevet på dansk, antage 28 forskellige værdier.

For at gøre det nemmere at oversætte fra klar- til ciffertekst brugte man ofte tabeller. Ved at kigge på cæsaralgoritmen med nøglen $k = 3$ kan man opstille følgende krypteringstabel:

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

M-rækken viser klartekst-bogstavet mens C-rækken viser hvad det bliver krypteret til.

Ved at knytte en talværdi til hvert bogstav (A er 0, B er 1, C er 2, … Å er 28) kan man beskrive cæsaralgoritmens krypteringsfunktion matematisk. For hvert bogstav i klarteksten gælder det at:

$$c = (m + k)\ mod\ 29$$

Modulus-operatoren sørger for at c aldrig falder udenfor alfabetets rammer (0-28). Hvert klartekst-bogstav har derfor ét tilsvarende ciffer-bogstav. Regning med modulo vil blive beskrevet nærmere i afsnittet *Modulus*.

Denne tilgang har dog flere svagheder. Den første og mest håndgribelige svaghed er at der kun kan benyttes 28 forskellige nøgler. Dette gør at algoritmen realistisk kan brydes ved blot at forsøge sig med alle de forskellige nøgler - især fordi det vil være tydeligt når man har fået fat i den rigtige, idet teksten pludselig vil bestå af rigtige ord.

Det viser sig dog at Cæsaralgoritmen blot er et særtilfælde af det, man kalder monoalfabetiske substitutionsalgoritmer. Disse algoritmer lader nøglen antage langt flere forskellige kombinationer. Algoritmen vil dog stadig være sårbar overfor et matematisk angreb, kaldet frekvensanalyse. Her analyserer man hvor ofte hvert bogstav fremgår af cifferteksten og sammenligner med en normal tekst

[5] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 39

på det givne sprog. Hermed kan man aflede hvad hvert bogstav er substitueret til. Nærmere beskrivelse af monoalfabetiske substitutionsalgoritmer og deres svagheder kan findes i bilag 1, *Monoalfabetisk substitution*.

2.2 Polyalfabetisk substitution

For at løse sårbarheden overfor frekvensanalyse, opfandt man det, der hedder polyalfabetiske substitutionsalgoritmer. Den går ud på at nøglen består af flere forskellige alfabeter som dem fra den monoalfabetiske substitution. En nøgletabel kunne altså se ud som herunder.

```
M: 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 Æ Ø Å

C1: E D V R H O U B J Æ L Z I N S Å X G A P Y M T Ø W K F Q L
C2: X E B O U T Z M Æ G D P Å H K V Ø A Y N R W Q J L S F I P
C3: N V I U X O T P H B Ø G E Q Å A M J K R S D Æ W Y F Z L B
```

Ud fra denne tabel ville man så kryptere skiftevis med de 3 alfabeter. For eksempel ville man kryptere ordet ”PROJEKTET” på følgende måde:

```
 M: P R O J E K T E T

C1: Å G S Æ H L P H P
C2: V A K G U D N U N
C3: A J Å B X Ø R X R

 C: Å A Å Æ U Ø P U R
```

Denne specifikke kryptering viser flere af fordelene ved en polyalfabetisk substitutionsalgoritme:

- P og O bliver krypteret til det samme bogstav (Å)
- T, som findes 2 gange i klarteksten, bliver krypteret til to forskellige ting (P og R)

Med denne metode bliver det pludselig umiddelbart umuligt at lave frekvensanalyse på. Hvis man kender antallet af krypteringsalfabeter, er det dog pludselig ikke så svært idet man blot kan tage (i dette tilfælde) hvert tredje bogstav og se det som 3 separate tekster. Specifikke metoder til angreb på algoritmerne er dog ikke hovedemnet af denne projektopgave, så de vil ikke blive beskrevet nærmere.

Selvom en polyalfabetiske krypteringsalgoritme løser det mest åbenlyse problem ved en monoalfabetisk krypteringsalgoritme, og faktisk er ret svær at knække medmindre man har en tilstrækkelig lang tekst, lider den stadig under for en helt grundlæggende svaghed: Man skal dele koden med modparten.

Dette er et kæmpe problem er flere årsager. For det første er det svært at sikre at den første kontakt (når nøglen skal udveksles) er helt privat. I et stort netværk af mennesker, hvor alle skal være i stand til at kommunikere privat med alle, vil der desuden skulle genereres en nøgle for hvert par. Hvis 100 mennesker skulle kommunikere, ville der således skulle genereres:

$$\sum_{k=1}^{99} k = 4950 \; nøgler$$

For hver ny bruger skal der derudover genereres $n-1$ nye nøgler hvor n er det nye antal brugere. Ikke nok med at nøglerne skal genereres, de skal også udleveres til alle de eksisterende brugere - uden at blive opsnappet af uvedkommende. Dette viste sig at være vældig besværligt, så derfor fandt man et alternativ.

2.3 Asymmetrisk kryptering

At noget er symmetrisk vil sige at det kan spejles om en akse. Indenfor kryptografi kaldes en krypteringsalgoritme symmetrisk, hvis der bruges samme nøgle til at kryptere og dekryptere. Dette har været tilfældet i alle de hidtil gennemgåede krypteringsalgoritmer. Den eneste forskel på krypterings- og dekrypteringsprocessen har været at man fortolkede nøglen forskelligt alt efter om der skulle krypteres eller dekrypteres.

Asymmetrisk kryptering går derimod ud på at man har et *nøglepar* i stedet for en enkel nøgle. I dette nøglepar findes en offentlig nøgle (public key) og en privat nøgle (private key). Den offentlige nøgle må, som navnet indikerer, gerne offentliggøres. Den private nøgle skal dog holdes hemmelig. Idéen er dermed at dem, der vil sende dig en besked bruger din offentlige nøgle til at kryptere den. Hermed er det kun *dig*, der ved hjælp af din private nøgle, kan dekryptere beskeden. Den offentlige nøgle kan altså udelukkende bruges til at kryptere, ikke dekryptere.

Dette koncept løser problemet med at udveksle nøgler. Det lyder dog meget mærkeligt, at det skulle være muligt. Derfor blev det heller ikke opdaget før 1976.[6]

Teorien bag asymmetrisk kryptering (såkaldt *Public Key Cryptography*[7]) er at de to nøgler er matematisk forbundne. Efter at konceptet var blevet fremsat i 1976 gik der omtrent et år før en reel algoritme blev opfundet[8] - nemlig RSA, som senere vil blive beskrevet og analyseret.

Men hvordan kan to nøgler være matematisk forbundne? Det hele går ud på at vi skal finde en såkaldt ”trapdoor, one-way function”[9] - en matematisk operation, der er nem at lave den ene vej (kryptering), men svær at lave den anden vej (dekryptering) uden at have fået et *hint* - den private nøgle.

Her bruges begreberne *nemt* og *svært* ret vagt, men det skal forstås som at krypteringsoperationen skal kunne klares af en computer indenfor et rimeligt tidsrum mens dekrypteringsoperationen skal være for intens en operation til at kunne gennemføres indenfor realistisk tid medmindre man kender den hemmelige nøgle. ”Realistisk tid” er stadig et meget vidt begreb. En forklaring af hvad det vil sige, findes i et senere afsnit.

Det viser sig at man i løbet af mange år har studeret disse envejsfunktioner. Den mest anvendte envejsfunktion er heltalsfaktorisering (eller primtalsfaktorisering), nemlig at faktorisere et tal til primtal. Givet to faktorer, er det super nemt at finde produktet. At gøre det den anden vej, nemlig at finde faktorerne givet produktet, er derimod meget svært - og for store tal såkaldt ”computationally infeasible”.[10]

[6] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 167
[7] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 165
[8] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 167
[9] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 168
[10] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers

2.3.1 Brug

Som nævnt tidligere, løser asymmetrisk kryptering hele problemet med distribution af nøgler. Dermed kan det, som en opgradering i forhold til symmetrisk kryptering, bruges til at sende hemmelige beskeder mellem brugere.

Det kan dog også bruges til andre ting. For eksempel er det et meget brugbart system til at lave digitale signature - online underskrifter. Dette fungerer fordi en besked, der er krypteret med den hemmelige nøgle kan dekrypteres med den offentlige. Dette ville blive vist under afsnittet *Implementation af RSA*.

At man kan underskrive noget online har mange fordele - mest af alt at man pludselig kan være sikker på hvem man egentlig kommunikerer med.

2.3.2 RSA

Som tidligere nævnt blev RSA fremvist blot et år efter konceptet asymmetrisk kryptering blev opdaget. Det blev opfundet af R.L. **R**ivest, A. **S**hamir og L. **A**dleman[11] - deraf navnet. Selvom RSA i dag er over 20 år gammelt, er det stadig den mest udbredte algoritme til asymmetrisk kryptering. Den bygger på 3 grundlæggende koncepter, der tager udgangspunkt i følgende kongruens:[12]

$$c \equiv m^e \ (mod\ n)$$

1) At løse kongruensen med hensyn til c, givet m og e er en relativ simpel operation
2) At løse kongruensen med hensyn til m, givet c og e er derimod det man kalder *computationally infeasible* - ikke realistisk at kunne løse
3) Hvis man kender primtalsfaktoriseringen af n, er punkt 2 pludselig en relativ simpel operation

Hvert af disse punkter vil løbende blive forklaret og analyseret. Først er det dog vigtigt at få noget grundlæggende talteori på plads. Talteori er den del af matematikken, der primært beskæftiger sig med de naturlige tal - altså de positive heltal.[13]

2.3.2.1 Primtalsfaktorisering

Et primtal er et tal større end 1, der kun går op i 1 og tallet selv. Disse tal anses af mange som de naturlige tals byggesten.[14] Talteoriens hovedsætning siger nemlig følgende:

> *"Ethvert naturligt tal n, skarpt større end 1, kan på én og kun én måde faktoriseres"*[15]

Denne sætning kan deles op i to dele: en eksistensdel (at en primtalsfaktorisering findes for ethvert naturligt tal) og en entydighedsdel (at en primtalsfaktorisering af et naturligt tal kun kan foretages på én måde). Beviset for eksistensdelen bliver gennemgået og kommenteret i bilag 2, *Bevis for eksistensdelen af talteoriens hovedsætning.*

Ved at kigge på RSA's 3. grundlæggende koncept ser vi at et tals primtalsfaktorisering fungerer som den nøgle, der skal til, for at dekrypteringsprocessen går nemt. Her udnytter man, at det er en meget

[11] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 147
[12] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 147
[13] Hansen, J. P. (2013) Primtalsmysterier: Matematiske mysterier, 2013, s. 60
[14] Hansen, J. P. (2013) Primtalsmysterier: Matematiske mysterier, 2013, s. 58
[15] Hansen, J. P. (2013) Primtalsmysterier: Matematiske mysterier, 2013, s. 60

langtrukken proces at faktorisere store tal til deres primiske faktorer.[16] Det er derfor ikke realistisk for en angriber selv at "generere" en nøgle ved at faktorisere.

2.3.2.2 Modulus

Tidligere blev det fremlagt at RSA tager udgangspunkt i følgende kongruens:

$$c \equiv m^e \ (mod\ n)$$

Dette kaldes en kongruens, og ikke en ligning, grundet det lidt specielle "lighedstegn" samt modulusoperatoren til højre. Det betyder blot at c's rest efter division med n er ens med m^e's rest efter division med n. Tallene c og m^e behøver derfor ikke at være *ens* (lighed, ligning) - de skal blot være kongruente med hensyn til n. De skal altså ende i samme såkaldte *restklasse*.

Modulus kan også bruges i ligninger. I sådanne tilfælde skriver man det blot $a\ mod\ b = c$. Modulo-operatoren betyder her ikke det samme som i kongruenser. I ligninger gælder modulo kun på den side, den står skrevet, ligesom addition og multiplikation. Forskellen illustreres herunder.

$$12\ mod\ 5 \neq 17$$

$$12 \equiv 17\ (mod\ 5)$$

Ligningen passer ikke, da $10\ mod\ 5 = 2\ \neq 17$. Kongruensen er derimod sand, da:

$$12\ mod\ 5 = 2 = 17\ mod\ 5$$

Ved at definere en funktion ***t***, der blot afkorter et tal til sin heltalsdel (for eksempel $t(2{,}5) = 2$), kan man opstille følgende ligning for regning med modulus:

$$a\ mod\ n = a - t\left(\frac{a}{n}\right) \cdot n$$

Et mere håndgribeligt eksempel end $c \equiv m^e\ (mod\ n)$ kunne være:

$$5 \equiv 1\ \ (mod\ 2)$$

Her løser vi hver side for sig selv:

$$5 - t\left(\frac{5}{2}\right) \cdot 2 = 1$$

$$1 - t\left(\frac{1}{2}\right) \cdot 2 = 1$$

Når modulus bruges i en kongruens skrives det, som allerede vist, til sidst. Nogle gange skrives det oven i købet helt til højre som herunder:

$$c \equiv m^e \qquad\qquad (mod\ n)$$

Det betyder det samme, uanset om det står helt til højre eller lige efter ligningen.

Når man opererer med modulus, kan man indskrænke talmængden for mulige resultater til følgende:

[16] Hansen, J. P. (2013) Primtalsmysterier: Matematiske mysterier, 2013, s. 62

$$\mathbb{Z}/N\mathbb{Z} = \{0, \dots, N-1\}$$ [17]

Denne mængde indeholder nemlig alle de mulige restværdier man kan få ved division med N.

2.3.2.3 Fermats lille sætning

Fermats lille sætning er navngivet efter den franske matematiker Pierre de Fermat.[18] Den lyder som følger:

$$a^p \equiv a \ (mod\ p)$$

Hvor p er et primtal mens a er et vilkårligt heltal. Denne sætning fortæller at så længe p er et primtal, vil p gå op i $a^p - a$. Dette kan man udlede fra sætningen da modulus-operatoren fortæller at $\frac{a^p}{p}$ giver resten $a\ (mod\ p)$. Når der kigges på specifikke tal, er der to muligheder. $a\ mod\ p = 0$, altså at p går op i a og $a\ mod\ p > 0$, altså at p *ikke* går op i a. Gennemgang af disse to eksempler kan findes i bilag 3, *Eksemplificering af Fermats lille sætning*.

Ser man nærmere på tilfældet hvor $a\ mod\ p > 0$, ses det at det er muligt at dividere med a på begge sider.[19]

Hermed fremkommer følgende sætning:

$$\frac{a^p}{a} \equiv \frac{a}{a}\ (mod\ p)$$

$$a^{p-1} \equiv 1\ (mod\ p)\ \Updownarrow$$

Dette var sætningen Fermat originalt fremlagde og fremadrettet er det derfor den, der vil blive refereret til med "Fermats lille sætning".

Fermats lille sætning blev brugt af skaberne af RSA, som bevis på at deres metode altid virkede. Dette er dog nemmere at gøre med Eulers sætning - det vil blive gennemgået under afsnittet *Implementation af RSA*.

2.3.2.4 Eulers sætning

Ved at kigge tilbage på RSAs 3. grundvilkår ses det at dekrypteringsprocessen (at løse kongruensen med hensyn til m) er nem, hvis man kender primtalsfaktoriseringen af n. Her kommer Eulers sætning, som ofte er grundlaget for beviser for RSA, ind i billedet. Den siger følgende:[20]

"Lad a og n være indbyrdes primiske heltal, så gælder det at:

$$a^{\varphi(n)} \equiv 1\ (mod\ n)"$$

Her er $\varphi(n)$ Eulers phi-funktion (også kaldet Eulers totientfunktion), som tæller antallet af heltal mellem 1 og n, som er indbyrdes primiske med n.[21] Det noteres at Eulers phi-funktion til et primtal p nødvendigvis må give $p - 1$ idet et primtal er indbyrdes primisk med alle tal mindre end det selv.

[17] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 3
[18] Fermat's Little Theorem, Wikipedia. Fundet 16/12/2017 på https://en.wikipedia.org/wiki/Fermat%27s_little_theorem
[19] Fermat's Little Theorem, Wikipedia. Fundet 16/12/2017 på https://en.wikipedia.org/wiki/Fermat%27s_little_theorem
[20] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 147
[21] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 147

Derudover ser vi fra phi-funktionens definition at der også er følgende regel:[22]

$$n = p_1 \cdot p_2 \Rightarrow \varphi(n) = (p_1 - 1)(p_2 - 1)$$

Hvis vi kender et tal n, samt dets faktorisering til p_1 og p_2 kan vi altså uden problemer finde funktionsværdien for Eulers phi-funktion til n. Denne funktionsværdi er grundlaget for den private nøgle i RSA, og hvis man kender primtalsfaktoriseringen af n, må den derfor være nem at beregne.

2.3.2.5 Implementation af RSA

Det er allerede blevet vist at RSA grundlæggende handler om følgende kongruens:

$$c \equiv m^e \ (mod\ n)$$

At løse denne kongruens med hensyn til m kaldes RSA-problemet.[23] Dette afsnit vil vise hvordan RSA-algoritmen er implementeret.[24]

Først vælges 2 store primtal p og q, hvorefter produktet $n = p \cdot q$ findes. Her huskes tilbage til Eulers phi-funktion og det ses dermed at:

$$\varphi(n) = (p - 1)(q - 1)$$

Herefter vælges e, som ses i RSA-problemet. Denne bruges til at kryptere klarteksten. e vælges *tilfældigt*, men med de to krav at $1 < e < \varphi(n)$ og at e og $\varphi(n)$ er indbyrdes primiske.

Herefter findes d, som er dekrypteringskonstanten. Denne skal udregnes, og der er følgende krav:

$$1 < d < \ \varphi(n)$$

$$e \cdot d \equiv 1 \left(mod\ \varphi(n)\right)$$

d skal altså, ligesom e, være mellem 1 og $\varphi(N)$. Derudover skal $e \cdot d - 1$ gå op i $\varphi(n)$. Til at finde denne kan man bruge den Euklids udvidede algoritme.[25]

Her findes også svaret på hvorfor RSA-problemet er nemt at løse, hvis man kender n's faktorisering. Man kan hermed blot selv udregne $\varphi(n)$, som nu er simpel at finde. e har man allerede givet. Dermed er det nu lige så nemt for en uvedkommende som for ejeren af nøglen at beregne d og dermed kunne dekryptere alle ejerens beskeder.

Nu har man i realiteten genereret et fungerende nøglepar til RSA-algoritmen. Følgende information er fundet:

[22] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers s. 148
[23] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s 172
[24] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 148-151
[25] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 149

Offentligt (public key)	Privat (private key)
• N • e	• d • p • q • $\varphi(N)$

Det noteres at den private del kan reduceres til udelukkende at bestå af d. Det kan dog sænke mængden af arbejde nødvendigt for at dekryptere en besked med en faktor 4 at huske p og q.[26]

Efter at have udgivet N og e, kan alle sende private beskeder til brugeren, der genererede dem. De omskriver blot beskeden til et tal og løser RSA-kongruensen med hensyn til c - en simpel operation. Til dette bruges blot den offentlige nøgle, som består af n og e.

For at dekryptere beskeden skal man kende d. Hermed kan man blot løse følgende kongruens med hensyn til m - klarteksten:

$$m \equiv c^d \ (mod\ n)$$

Dette er man sikker på altid passer. I bilag 4, *Bevis af* $m \equiv c^d \ (mod\ n)$ er det bevist at RSA-algoritmen altid vil give samme output, som der blev givet af input, efter en kryptering-dekryptering-process.

Det er dog vigtigt at beskeden er mindre end n. Ellers vil man miste information idet:

$$m > n \Rightarrow m\ mod\ n\ \neq m$$

2.4 Konklusion på kryptering

Det kan konkluderes at krypteringsalgoritmer er blevet langt mere avancerede gennem tiden. Fra Cæsaralgoritmen, der blot gik ud på at forskyde hvert bogstav, til RSA, der anvender avancerede matematiske koncepter, og sætninger, til at sikre at koden ikke kan brydes. Derudover anvendes disse koncepter til at sikre at algoritmen altid vil give det rigtige svar, hvis man dekrypterer med den rigtige nøgle.

Derudover kan det konkluderes at man fik løst mange af de problemer, der var ved symmetriske krypteringsmetoder, men at problemet med nøgleuddeling var uløseligt medmindre man helt skiftede retning - heraf asymmetrisk kryptering. Denne metode sikrer at hele værdien bag krypteringen ikke går tabt, bare fordi en uvedkommende fik fat i en brugers nøgler under uddelingsprocessen.

Det kan derudover konkluderes, at asymmetrisk kryptering kan bruges som en digital underskrift, idet den matematiske måde, hvorpå nøglerne hænger sammen, gælder begge veje. En besked, der er krypteret med den private del af nøglen, kan dekrypteres ved hjælp af den offentlige del af nøglen.

[26] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 151

3 Hashing

En hashingfunktion er en funktion, der tager et input af vilkårlig længde[27] og producerer et output af en bestemt længde. Det er en envejsfunktion, da man mister data under processen. Det er altså ikke muligt at få sit input tilbage ved hjælp af hashing-outputtet. At man *reducerer* dataen under hashingfunktionen forårsager også at to forskellige beskeder kan resultere i samme hash.

Hashingfunktioner bruges i bund og grund til at skabe et "fingeraftryk" for en bestemt besked. Hvis man sender en besked samt dets hash kan modtageren selv køre hashingfunktionen på beskeden og tjekke at de to er ens. Hermed har man bekræftet at beskeden ikke er blevet ændret.

Et simpelt eksempel på en sådan hashingfunktion ville være et tals reducerede tværsum.[28] Her bliver ethvert tal, uanset længde, forkortet ned til ét decimal.

Tag for eksempel tallene 1523 og 27. Disse to tal har ikke samme længde. Deres tværsum har dog begge længden 1:

$$1 + 5 + 2 + 3 = 11 \Rightarrow 1 + 1 = 2$$

$$2 + 7 = 9$$

Tværsumsfunktionen er dog ikke en særlig god hashingfunktion. De næste kapitler vil beskrive hvornår man ville bruge en hashingfunktion og hvordan man bestemmer om en bestemt funktion er *god* som kryptografisk hashingfunktion.

3.1 Brug

Ofte er man ikke nødvendigvis interesseret i at en besked forbliver hemmelig - den behøver altså ikke at blive krypteret. Ofte er det vigtigste faktisk at beskeden ikke er blevet *ændret*.

Antag at brugeren A vil sende en besked til brugeren B. A finder B's offentlige nøgle (n_B og e_B), og krypterer ved hjælp af RSA-algoritmen denne. På vej fra A til B bliver beskeden dog opsnappet af en uvedkommende bruger E. E kan ikke læse beskeden, da den jo er krypteret. E har dog en idé om hvad beskeden indeholder. E skriver derfor sin egen besked, krypterer den med B's offentlige nøgle (som jo er offentlig) og sender den afsted i stedet for A's oprindelige besked. Når beskeden kommer frem har B ingen idé om at det ikke er den originale besked, der er blevet modtaget.

I ovenstående situation ville det være smart at anvende en kryptografisk hashingfunktion. På den måde kan A vedhæfte beskedens hash (eller sende det separat). Dette vil lade B tjekke at beskeden ikke er blevet ændret. Her går vi ud fra at A bruger et ekstra system til at sikre at E ikke bare hasher sin nye besked og vedhæfter dette i stedet for A's originale hash. Sådanne systemer findes der flere af, men de vil ikke blive beskrevet nærmere her.[29]

Nogle mere virkelighedsnære eksempler på hvornår man ville bruge hashingfunktioner som besked-verificering er når man modtager emails eller downloader filer fra nettet. Her er det ofte ikke så vigtigt om andre kan læse indholdet (medmindre det er strengt privat indhold, selvfølgelig). Det er dog vigtigt

[27] De fleste algoritmer har en max-længde, men denne er ofte så høj, at den sjældent nås. For specifikke max-længder skal man tjekke algoritmens dokumentation
[28] Tværsum, Wikipedia. Fundet 17/12/2017 på https://da.wikipedia.org/wiki/Tv%C3%A6rsum
[29] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 290

at de ikke er blevet rodet med. Hvis ikke man sikrede sig mod dette, kunne en angriber for eksempel ændre et link i en mail til at føre til deres egen hjemmeside i stedet for den oprindelige.

Den mest udbredte, og den første brug, af hashingfunktioner var dog til beskyttelse af kodeord.[30] Ved at hashe en brugers kode lige så snart de registrerer sig, behøver man ikke at gemme deres egentlige kode. Når de prøver at logge ind, kan man blot sammenligne hashet af inputtet med det gemte hash. Hvis hashingalgoritmen derudover kravene til en god hashingalgoritme, er det pludselig ikke et lige så stort problem hvis en uvedkommende får adgang til brugerdatabasen, idet de ikke har nogle af brugernes kodeord i klartekst.

En meget vigtig problemstilling indenfor hashing er hvornår det overhovedet er relevant at bruge en hashingfunktion. Ulempen er at man mister information, hvis man kun beholder hashet. Det er derfor ikke altid en god idé. Der kan dog være meget at vinde, hvis man ikke nødvendigvis ønsker at gemme originalen - for eksempel ved opbevaring af kodeord.

3.2 Vurderingskriterier

Som nævnt tidligere er den reducerede tværsum et eksempel på en **meget** dårlig hashingfunktion. Men hvordan bestemmer man om en hashingfunktion er god eller dårlig? Helt grundlæggende er der 3 ting, der skal være tilstrækkeligt svære for at en hashingfunktion kvalificeres som god.[31]

1) At finde en besked m, der giver et bestemt hash
2) At finde to beskeder m_1 og m_2, der giver samme hash
3) Givet en besked m, at finde en besked m', der giver samme hash som m

Disse 3 krav kan, hvis man simplificerer lidt, opsummeres til én sætning: *For at en hashingfunktion er god, skal det være svært at forudse hvad der kommer ud af den.*

Det er dog værd at uddybe hver af de tre krav.

Det første krav er, at det skal være svært at finde en besked m, der giver et bestemt hash. Hvis man kigger på forrige eksempel med A, B og E, kan man se hvorfor det er vigtigt. Hvis det var nemt at finde en besked, der gav et bestemt hash, kunne E blot sørge for at den nye besked havde samme hash som den oprindelige besked.

Det andet krav er at det skal være svært at finde to beskeder, der giver samme hash. Går vi ud fra at dette *ikke* er svært, kan vi udnytte svagheden ved at sende to forskellige beskeder til to forskellige modtagere, men vise at de har samme hash. Hermed tror de to modtagere, de har fået samme besked.

Det tredje krav var at det, givet en besked m, skulle være svært at finde en besked m', der giver samme hash. Hvis dette ikke var svært, ville enhver der kendte den oprindelige besked kunne sende en modificeret besked, der gav samme hash. Dette ville skabe de samme problemer som tidligere, hvor modtageren ikke kunne sikre sig at beskeden var ægte.

Som eksempel på en hashingfunktion blev den reducerede tværsum tidligere nævnt. I bilag 5, *Eksempler på tværsum som hashingfunktion,* kan findes en gennemgang af 3 eksempler, der viser at denne *hashingfunktion* ikke opfylder et eneste af de 3 krav.

[30] Thomsen, S. S. (2008) Cryptographic Hash Functions: DTU, 2008, s. 11
[31] Thomsen, S. S. (2008) Cryptographic Hash Functions: DTU, 2008, s. 9

3.3 SHA

Der har været mange forskellige hashingalgoritmer gennem tiden. En af de tidligste var SHA, der står for Secure Hash Algorithm.[32] Den er en del af MD4-familien af hashingalgoritmer.[33] Disse bygger alle sammen på det grundlag, man kalder "the avalanche effect" (på dansk dominoeffekten).[34] Dette koncept går ud på at en lille ændring i input skal give en stor ændring i output. Dette er et godt udgangspunkt, da det gør algoritmen sværere at forudse. Dermed har den nemmere ved at overholde de 3 krav til en god hashingalgoritme.

3.3.1 Bitoperationer

I SHA-algoritmen bruges der rigtig mange forskellige bit-operationer. Disse operationer opererer på enkelte bits, altså 0 eller 1 og sammenligner dem sommetider med hinanden.

For eksempel er der en bit-operation, der hedder **and** (og). Den er udtrykt ved tegnet ∧. Den sammenligner to bit-strukturer og giver et output indeholdende 1-bits der hvor begge inputs har en 1-bit, og 0-bits derudover.

For eksempel:

$$a = 00011101$$

$$b = 10010001$$

Ved at bruge and-operatoren får vi:

$$a \wedge b = 00010001$$

For nemmere at kunne forstå forskellige bit-operationer kigger man ofte på såkaldte *truth tables*, som set herunder.[35]

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

En oversigt over de bit-operationer, der bliver brugt i SHA1-algoritmen, samt truth tables for disse, kan findes i bilag 6, *Oversigt over bitoperationer*.

Nogle bit-operationer bruges dog ikke til at sammenligne. I SHA1-algoritmen bruges også det man kalder *bit shifts.* Det går ud på at man tager en række bits og skubber dem *x* pladser til den ene eller anden side.

Antag for eksempel at vi definerer a ligesom før.

[32] Secure Hash Algorithm, Wikipedia. Fundet 18/12/2017 på https://da.wikipedia.org/wiki/Secure_Hash_Algorithm
[33] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 156
[34] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 156
[35] Truth table, Wikipedia. Fundet 18/12/2017 på https://en.wikipedia.org/wiki/Truth_table

$$a = 00011101$$

Følgende operationer vil give følgende resultater:

$$a \lll 1 = 00111010$$

$$a \ggg 1 = 10001110$$

Bemærk at hver af operationerne har skubbet a's bits 1 plads. Først til venstre, så til højre. De bits der "ryger ud over kanten" kommer ind fra den anden side.

3.3.2 Gennemgang af algoritmen

Først fastsættes fem algoritmekonstanter, fire rundekonstanter og tre reduktionsfunktioner. De er ikke vigtige for gangen i algoritmen, så de kan findes i bilag 7, *Oversigt over SHA1's konstanter*.

Disse 9 konstanter og 3 funktioner er defineret for algoritmen selv. De er altså ens for alle implementeringer af SHA1-algoritmen, og må aldrig ændres. Hvis de bliver ændret, vil alle gamle SHA1-hashes være ugyldige, da det samme input ville give et nyt hash med de nye konstanter.

Derefter tages brugerens input, der straks bliver konverteret til binær. På denne måde kan man køre bit-operationer på det. Derefter kopieres de 5 H-konstanter over i nogle nye variable kaldet A, B, C, D og E. Nu kan hashing-delen af algoritmen begynde.

Brugerens input opdeles i blokke af 512 bits. Hvis beskeden er større end dette, bliver den altså delt over flere blokke. Herefter sættes en 1-bit bagpå. Dette er for at vise at beskeden nu er slut. Derefter bliver der tilføjet 0-taller indtil antallet af bits går op i 512, altså $N \equiv 0 \ (mod\ 512)$. Uanset hvor mange blokke der er, er man nu sikker på at de alle har samme længde, nemlig 512 bits.

Algoritmen, der bliver gennemgået herunder, bliver kørt på hver blok.

Hver blok består af 16 *ord*. Et ord består altså af 32 bit. Før man begynder at operere på disse 16 ord, skaber man dog 64 nye, så der i alt er 80 ord (med indeks fra 0 til 79). Disse *opstillede* ord skabes ud fra følgende algoritme:

For j = 16 til 79, kør

$$X_j = \left(\left(X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}\right) \lll 1\right)$$

Slut

Hvor X er listen af ord, og X_j refererer til det j'te element i denne liste. Herefter kan blokken begynde sin gang gennem de 4 runder.

Pseudokoden for de 4 runder ser således ud:[36]

[36] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 159

Runde 1

For j = 0 til 19, kør

$$t = (A \lll 5) + f(B, C, D) + E + X_j + y_1$$

$$(A, B, C, D, E) = (t, A, B \lll 30, C, D)$$

Slut

For j = 20 til 39, kør

$$t = (A \lll 5) + h(B, C, D) + E + X_j + y_2$$

$$(A, B, C, D, E) = (t, A, B \lll 30, C, D)$$

Slut

For j = 40 til 59, kør

$$t = (A \lll 5) + g(B, C, D) + E + X_j + y_3$$

$$(A, B, C, D, E) = (t, A, B \lll 30, C, D)$$

Slut

For j = 60 til 79, kør

$$t = (A \lll 5) + h(B, C, D) + E + X_j + y_4$$

$$(A, B, C, D, E) = (t, A, B \lll 30, C, D)$$

Slut

Det ses hvordan de 4 runder ligner hinanden ret meget. Forskellen ligger i hvilken reduktionsfunktion (f, g og h), der bliver brugt, samt hvilken rundekonstant (y_1-y_4), der bliver lagt til.

Det er også interessant at se hvordan de variable A, B, C, D, E bliver roteret for hver omgang i løkkerne. A bliver sat til t, B bliver sat til A, og så videre.

Efter denne algoritme er kørt gennem med alle de blokke, der måtte være, lægges A, B, C, D, E til algoritmekonstanterne. Hvis O er vores output, ender det altså med at være:

$$O_1 = H_1 + A$$

$$O_2 = H_2 + B$$

$$O_3 = H_3 + C$$

$$O_4 = H_4 + D$$

$$O_5 = H_5 + E$$

Efter dette er gjort, får man det endelige hash ved at sætte O'erne op efter hinanden og konvertere til hexadecimal (talsystemet med base 16).

Et hash kunne se ud som herunder (SHA1-hashet for "SKOLE"):[37]

a5943f82cbe8038fdf33097f64abec29aaf2a287

3.4 Implementering af en SHA-agtig algoritme

I virkeligheden ville man aldrig implementere sin egen hashingalgoritme. Det er simpelthen for usikkert, i forhold til at bruge en, der er udviklet af hundredevis af eksperter indenfor området over mange årtier.

Jeg har dog her udviklet min egen implementering af SHA-1-algoritmen. Programmer er udviklet i JavaScript og kan køres ved hjælp af programmet NodeJS.[38] Det startes ved at køre følgende kommando i en terminal: [39]

$$node\ hashing.js\ input$$

Dit input vil derefter blive hashet og skrevet til konsollen. Herunder ses et eksempel på hvordan man bruger programmet samt hvad man får tilbage. Bemærk at programmet kun understøtter input på op 480 bit, da der kun bliver gennemgået en enkel blok. Ordet 'SRP' bliver hashet ved hjælp af programmet.

```
lukas@DESKTOP-QQOJQFM MINGW64 /G/Google Drive/Skole/2017-2018 (3.g)/SRP
$ node hashing.js SRP
Dit hash er: 8c37483333eeffea121f7664a996eddc214d0f6b3c
```

Hvis vi i stedet kørte programmet med input 'SRO', får vi et andet output.

```
lukas@DESKTOP-QQOJQFM MINGW64 /G/Google Drive/Skole/2017-2018 (3.g)/SRP
$ node hashing SRO
Dit hash er: eff579c997ad3180bae6f2b0325e6a28e5fef7a2
```

Det ses her at outputtet er markant anderledes. Dette er godt og verificerer implementationen af SHA1's dominoeffekt - det er trods alt grundprincippet bag MD4-familien. Det bemærkes dog også at outputtet fra SRP er 2 karakterer længere end outputtet fra SRO. Dette er ikke hensigtsmæssigt da et af kravene til en hashing-funktion jo er at det skal være samme længde. Det kommer dog af den mærkelige måde hvorpå JavaScript håndterer bit-operationer samt binær til hexadecimal konvertering. Dette vil der senere bliver kommenteret yderligere på.

Hvis man ønsker mere information omkring processen, kan man give et ekstra paremeter '*debug*'. Dette vil sørge for at programmet printer midterstadierne under hashingen til konsollen. Et eksempel på hvordan dette ser ud kan ses i bilag 8, *Program kørt med debug-mode*.

Programmet er en implementering af SHA-1 algoritmen - gennemgangen er derfor den samme og for det meste ret triviel. Den mest interessante del af koden er der hvor de 4 runder bliver kørt.

[37] SHA1 Online. Fundet 19/12/2017 på http://www.sha1-online.com/
[38] https://nodejs.org/
[39] For en fuld gennemgang af hvordan programmet køres, se venligst bilag 9, *Programmets kildekode*

```
for(let j = 0; j < 80; j++) {
	// Vi kigger på hvilken runde vi er nået til
	if(j < 20) {
		t = rol(A, 5) + f(B, C, D) + E + binaryArray[j] + roundConstants[0];
	} else if(j < 40) {
		t = rol(A, 5) + h(B, C, D) + E + binaryArray[j] + roundConstants[1];
	} else if(j < 60) {
		t = rol(A, 5) + g(B, C, D) + E + binaryArray[j] + roundConstants[2];
	} else {
		t = rol(A, 5) + h(B, C, D) + E + binaryArray[j] + roundConstants[3];
	}

	// Vi opdaterer værdierne
	A = t;
	B = A;
	C = rol(B, 30);
	D = C;
	E = D;
}
```

Under gennemgangen af SHA1-algoritmen blev det noteret, at runderne lignede hinanden meget. Det er her blevet udnyttet idet det hele står i én løkke. Den eneste forskel er hvordan t bliver defineret, og selv dette kunne være blevet generaliseret endnu mere, da $rol(A, 5) + E + binaryArray[j]$ altid indgår.

Jeg udnytter her at JavaScript har indbyggede bit-operationer. Når disse bliver brugt, er hvert tal dog defineret som en "32-bit signed integer".[40] Dette foråsager at bit-rotationer kan ændre tallets fortegn - og dermed gøre det negativt. Når dette tal så bliver lagt til et positivt tal, bliver der lavet koks i det. Derfor giver denne selvimplementerede version af SHA1 ikke samme output som en rigtig implementation.

Den fulde kildekode kan findes i bilag 9, *Programmets kildekode*.

3.5 Konklusion på hashing

For hashing kan det konkluderes at en hashingfunktion kun er brugbar hvis dens kvalitet (målt i forhold til de tidligere nævnte krav) overholder en hvis standard - ellers giver den blot en falsk fornemmelse af tryghed, der kan blive udnyttet af en uvedkommende.

Derudover blev der sat fokus på SHA1-algoritmen. Det kan konkluderes at denne lægger stor vægt på dominoeffekten. Dette gør at den flot overholder de 3 opstillede krav. Den fungerer derfor godt som hashingfunktion.

Det blev også pointeret at hashingfunktioner kan bruges til mange ting, vigtigst af alt til at sikre password og at sikre sig at en besked ikke er blevet ændret. Derudover blev det nævnt at det ikke altid er nødvendigt (eller overhovedet en god idé) at bruge en hashingfunktion, da man jo mister information, hvis man ikke opbevarer originalen. Og hvis man gør, kan det blot være spild af plads.

[40] Bitwise Operators, Mozilla Developer Network. Fundet 18/12/2017 på https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators

4 Kryptering og hashing i fremtiden

Der er ingen tvivl om at kryptering (især asymmetrisk) og hashing kommer til at have stor relevans i fremtiden. Digitale signature, der betyder lige så meget som en fysisk underskrift, er et stort spring i retning af en komplet digitalisering. Derudover bliver det kun vigtigere at kunne stole på de informationer man modtager, for eksempel over mail. At kryptering og hashing fortsat kommer til at have mange anvendelsesmuligheder, afhænger dog af at sikkerheden fortsat er i top. Og dette bliver kun sværere.

4.1 Computerens udvikling

Tidligere er det blevet nævnt at både RSA- og SHA1-algoritmen er sikre, idet den computerkræft, der skal til for at bryde dem, ikke er opnået endnu. For RSA passer det stadig. For SHA1 gør det dog ikke - den er blevet brudt.[41] Derfor anbefales det, at man rykker til en af de nye SHA-algoritmer. De nye er en del af den familie, man kalder SHA2, og de er endnu ikke blevet brudt.

Men hvornår sker det? Computere bliver ved med at blive kraftigere og kraftigere, så selv hvis man ikke finder et logisk smuthul i algoritmerne, kan man på et tidspunkt bare knække dem ved at prøve alle de forskellige kombinationer.

For SHA-algoritmerne handler sikkerheden om at det er for svært at generere et identisk match, som det tidligere er gennemgået. Det man kan gøre for at undgå dette, efterhånden som computere bliver hurtigere og hurtigere, er at lave output-hashets længde længere. Det har man for eksempel gjort fra SHA1 (med en hashlængde på 160 bits) til SHA256 (med en hashlængde på 256 bits) og videre til SHA512 (med en hashlængde på 512 bits). Dette skaber langt flere mulige kombinationer, og hvis man kan sikre at alle disse bliver ramt med ca. lige stor sandsynlighed, kan man dermed gøre det sværere for en angriber at ramme en kollision.

For RSA-algoritmen handler sikkerheden om at det er svært at faktorisere store tal. Hvis det er muligt at faktorisere n fra RSA-algoritmen, vil en angriber være i stand til selv at generere d og dermed dekryptere brugerens beskeder. I 2010 faktoriserede en gruppe forskere fra hele verden et 768-bit tal (232 decimale cifre).[42] Dette *sikkerhedshul* kan løses ved at vælge et større n. Der er dog en ulempe. Ved en fordobling af RSA-nøglelængden vil dekrypteringsprocessen blive 6-7 gange langsommere.[43] Dette er et kæmpe problem, da dekryptering selvfølgelig skal ske så hurtigt som overhovedet muligt.

Når fremtidens computere bliver hurtigere og hurtigere, bliver sikkerheden altså også nødt til at blive stærkere og stærkere - på den ene eller den anden måde.

[41] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 156
[42] Kleinjung, T. & Franke, J. & Lenstra, A. K. & Thomé, E. & Bos, J. W. (2010) Factorization of a 768-bit RSA modulus, 2010
[43] Coffey, Neil. RSA Key Length, Javamex UK. Fundet 18/12/2017 på https://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml

5 Konklusion

Som tidligere observeret er kryptografi en kæmpe del af det moderne informationssamfund - det er ikke til at slippe udenom. Det bliver udnyttet på mange forskellige måder. Både til at sende private beskeder, til at skrive under med en digital underskrift og til at sikre sig at en besked ikke er blevet ændret under transporten. Kravene til hvornår dataen er *sikker*, kan altså variere meget alt efter det specifikke brugstilfælde.

For kryptering kan dataen ses som sikker, hvis en uvedkommende ikke er i stand til at aflæse den. Her er det vigtigt at det ikke er muligt (eller i hvert fald tilstrækkelig svært) for en uvedkommende angriber at gætte eller generere brugerens nøgle således at dataen kan læses.

For RSA-algoritmen gælder dette om at en angriber ikke skal kunne faktorisere n. Det er dog blevet vist at større og større tal er blevet faktoriseret. Dette skaber en sikkerhedsbrist, der kun kan løses ved at vælge større nøgler. Dette forårsager imidlertid at krypterings- og dekrypteringsprocessen bliver væsentligt langsommere.

For hashing kan dataen derimod ses som sikker lige så snart en uvedkommende ikke er i stand til at generere bestemte resultater ved at forudse hvordan algoritmen opfører sig ved et bestemt input. Dermed opfylder hashingalgoritmen alle de 3 krav til en god hashingalgoritme. Efterhånden som computere bliver hurtigere og hurtigere til at teste om et input giver en kollision, bliver det dog sværere og sværere at holde dataen *sikker.* De nye SHA-algoritmer har midlertidigt løst dette problem ved at lave hash-outputtet flere gange længere end originalt. Dette skaber selvfølgelig langt flere mulige outputs, og derfor har computerne sværere ved at følge med.

Om man kan blive ved med at hæve længden af krypteringsnøglen og hashoutputtet eller om man bliver nødt til at finde nye algoritmer, kan kun tiden - eller mere realistisk, nogle meget kloge forskere - vise.

Anslag i alt inkl. mellemrum, ekskl. forside, abstract, indholdsfortegnelse, noter, indholdsfortegnelse, sidefod og -hoved, bilag og denne linje: **37149**

6 Litteraturliste

Forfatter i form af en organisation eller større gruppe

Anonyme bidragere. Kryptologi, Wikipedia. Fundet 18/12/2017 på https://da.wikipedia.org/wiki/Kryptologi

Anonyme bidragere. Fermat's Little Theorem, Wikipedia. Fundet 16/12/2017 på https://en.wikipedia.org/wiki/Fermat%27s_little_theorem

Anonyme bidragere. Secure Hash Algorithm, Wikipedia. Fundet 18/12/2017 på https://da.wikipedia.org/wiki/Secure_Hash_Algorithm

Anonyme bidragere.Truth table, Wikipedia. Fundet 18/12/2017 på https://en.wikipedia.org/wiki/Truth_table

Anonyme bidragere. Tværsum, Wikipedia. Fundet 17/12/2017 på https://da.wikipedia.org/wiki/Tv%C3%A6rsum

Det Danske Sprog- og Litteraturselskab. kryptere, Den Danske Ordbog. Fundet 14/12/2017 på http://ordnet.dk/ddo/ordbog?query=krypteret

Mozilla. Bitwise Operators, Mozilla Developer Network. Fundet 18/12/2017 på https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators

SHA1 Online. Fundet 19/12/2017 på http://www.sha1-online.com/

Forfatter i form af en eller flere privatpersoner

Arndt, Hans. Danish, AU Research. Fundet 14/12/2017 på http://www.hum.au.dk/ling/research/Danish.htm

Coffey, Neil. RSA Key Length, Javamex UK. Fundet 18/12/2017 på https://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml

Hansen, J. P. (2013) Primtalsmysterier: Matematiske mysterier, 2013

Kleinjung, T. & Franke, J. & Lenstra, A. K. & Thomé, E. & Bos, J. W. (2010) Factorization of a 768-bit RSA modulus, 2010

Madeira, Antonio. How does a hashing algorithm work?, *CryptoCompare*. Fundet 09/12/2017 på https://www.cryptocompare.com/coins/guides/how-does-a-hashing-algorithm-work/

Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College

Thomsen, S. S. (2008) Cryptographic Hash Functions: DTU, 2008

Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers

8 Bilag

8.1 Bilag 1 - Monoalfabetisk substitution

Cæsaralgoritmens største udfordring, nemlig det minimale antal mulige nøgler, løste man ved at lade nøglen være hele den nederste del af krypteringstabellen. I stedet for at man altid forskød et bogstav samme antal pladser, erstattede man det nu blot med et andet bogstav. Hvor nøglen til en besked krypteret med Cæsaralgoritmen blot er et tal, er koden til en monoalfabetisk substitutionsalgoritme hele alfabetet.

Man kunne for eksempel sende en besked hvor krypteringskoden er følgende tabel:

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

C-rækken er valgt tilfældigt. Der er altså ikke noget system som det ses i Cæsaralgoritmen. Man kan derimod se at Cæsaralgoritmen er et særtilfælde, en implementering, af en Monoalfabetisk substitutionsalgoritme idet konceptet er det samme, bogstaverne står blot ordnet, så man kan simplificere koden til et enkelt tal.

Da hvert bogstav kun står ét sted i C-rækken, er krypteringen stadig entydig. Denne tilgang løser altså cæsaralgoritmens største svaghed - det meget lille udvalg af mulige nøgler. Til en monoalfabetisk substitutionsalgoritme er der langt flere nøgler. Antallet af nøgler kan beskrives matematisk. Da alle bogstaverne skal fremgå af tabellen, og rækkefølgen er af betydning, må antallet af nøgler svare til følgende.

$$29! \approx 8{,}84 \cdot 10^{30}$$

Der er altså 8000 milliarder milliarder milliarder forskellige nøgler. Det er derfor ikke realistisk at gennemgå dem alle for at finde løsningen på denne måde - heller ikke med en computer.[44]

Monoalfabetiske substitutionsalgoritmer har dog stadig en væsentlig sårbarhed - frekvensanalyse. Ved at analysere hvor ofte hvert bogstav normalt bruges og sammenligne med hvor ofte hvert bogstav fremgår af cifferteksten, kan man, hvis den krypterede tekst er lang nok, se mønstre, der afslører koden.

[44] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 41

8.2 Bilag 2 - Bevis af eksistensdelen af talteoriens hovedsætning

Her vil jeg gennemgå beviset for eksistensdelen.[45]

Beviset går på at at et primtal er sin egen primtalsfaktorisering. Hvis vi antager at n er det mindste tal uden en primtalsfaktorisering, kan dette derfor ikke være et primtal. Da det ikke er et primtal, må det gå op i andre tal end 1 og det selv. Det må derfor kunne skrives som $n = n_1 \cdot n_2$. Da vi antog at n var det mindste tal uden en primtalsfaktorisering, må både n_1 og n_2 have en primtalsfaktorisering, idet begge disse tal er mindre end n. Vi sætter disse to primtalsfaktoriseringer til følgende.

$$n_1 = p_1 \cdot p_2$$

$$n_2 = p_3 \cdot p_4$$

Hvor alle p er primtal. Bemærk at de to primtalsfaktoriseringer lige så godt kunne bestå af flere tal - her går vi blot ud fra at både n_1 og n_2 faktoriseres til 2 primtal for at gøre det kortere.

Ved at tage produktet af disse to primtalsfaktoriseringer, får vi:

$$p_1 \cdot p_2 \cdot p_3 \cdot p_4 = n_1 \cdot n_2 = n$$

Vi kan hermed se at vores antagelse om at n, det antagede mindste tal uden en primtalsfaktorisering, ikke kan faktoriseres er forkert. Da der naturligvis må være et mindste tal uden en primtalsfaktorisering, hvis der skulle findes sådanne tal, må påstanden om at der findes naturlige tal uden primtalsfaktoriseringer altså være forkert.

Beviset for entydighedsdelen vil jeg ikke gennemgå. Det er dog værd at nævne at 1 ikke må ses som et primtal, når man snakker om primtalsfaktorisering - ellers kan talteoriens hovedsætning ikke passe. Dette er fordi 1 er den multiplikative identitet[46] - altså at et tal ganget med 1 blot vil give tallet selv. Hvis dette ansås som et primtal, ville enhver primtalsfaktorisering således kunne udvides til:

$$n = p_1 \cdot p_2 \cdot 1 \cdot 1 \ldots \cdot 1 \cdot 1$$

[45] Hansen, J. P. (2013) Primtalsmysterier: Matematiske mysterier, 2013, s. 60
[46] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s. 4

8.3 Bilag 3 - Eksemplificering af Fermats lille sætning

Der opstilles to eksempler, henholdsvis: $a \ mod \ p = 0$, altså at p går op i a og $a \ mod \ p > 0$, altså at p *ikke* går op i a.

Til første eksempel vælges $a = 14$ og $p = 7$. Hermed er $a \ mod \ p = 0$.

$$14^7 \equiv 14 \ (mod \ 7)$$

$$14^7 \ mod \ 7 = 0$$

$$14 \ mod \ 7 = 0$$

Da $0 = 0$ kan vi konkludere at sætningen passer. At sætningen passer for $a \ mod \ p = 0$ kan hurtigt bevises. Da vores begyndelsesbetingelse er $a \ mod \ p = 0$, altså at p går op i a, kan vi omskrive sætningen:

$$(kp)^p \equiv kp \ (mod \ p)$$

Så længe k er et heltal, hvilket vi ved, det er, er vi sikre på at $kp \ mod \ p = 0$. Hermed ser vi at højresiden i hvert fald er 0.

Ved at analysere $(kp)^p$ ser vi at:

$$(kp)^p = k_1p_1 \cdot k_2p_2 \dots \cdot k_pp_p = k_1 \cdot k_2 \dots k_p \cdot p_1 \cdot p_2 \dots \cdot p_p$$

Da vi kun arbejder med heltal må alle ovenstående værdier være heltal. Dermed kan vi samle alt udover p_p til en konstant.

$$k_{ny} = k_1 \cdot k_2 \dots k_p \cdot p_1 \cdot p_2 \dots p_{p-1}$$

Hermed har vi samme situation som på højresiden, nemlig:

$$(kp)^p = k_{ny}p_p$$

Og vi kan hermed konkludere at $(kp)^p \ mod \ p = 0$.

Herefter vælges $a = 3$ og $p = 7$. Hermed er $a \ mod \ p = 3 > 0$

$$3^7 \equiv 3 \ (mod \ 7)$$

Vi tester ligesom før om denne kongruens er sand.

$$3^7 \ mod \ 7 = 3$$

$$3 \ mod \ 7 = 3$$

Vi kan hermed konkludere at sætningen passer for dette eksempel hvor $a \ mod \ p > 0$.

8.4 Bilag 4 - Bevis af $m \equiv c^d \ (mod\ n)$ [47]

Det påstås at følgende kongruens er korrekt:

$$m \equiv c^d \ (mod\ n)$$

For at bevise dette, huskes hvordan c er defineret, nemlig m^e. Ovenstående kan derfor omskrives til:

$$(m^e)^d = m^{ed} \equiv m \ (mod\ n)$$

Dette bekræfter også den tidligere påstand om at en besked krypteret med d kan dekrypteres med e.

Da vi genererede d, sørgede vi for at $e \cdot d \equiv 1 \left(mod\ \varphi(n)\right)$. Derfor:

$$m^{ed} = m^{1+k\varphi(n)} \equiv m \ (mod\ n)$$

Hvor k er et positivt heltal. Dette giver mening da modulo sørger for at $e \cdot d + k_1\varphi(n) = 1 + k_2\varphi(n)$. k er her forskellen på k_1 og k_2.

Dette kan ved hjælp af potensregnereglerne omskrives til:

$$m \cdot m^{k\varphi(n)} = m \cdot \left(m^{\varphi(n)}\right)^k \equiv m \ (mod\ n)$$

Nu sammenlignes med Eulers sætning, nemlig:

$$a^{\varphi(n)} \equiv 1 \ (mod\ n)$$

Denne sætning gælder når a og n er indbyrdes primiske. Da n er produktet af to primtal, er det meget sandsynligt at det er det - derfor kan denne metode bruges som bevis. Det viser sig dog, at $c^d \equiv m \ (mod\ n)$ også passer selvom de ikke er indbyrdes primiske - det skal blot bevises på en anden måde.[48]

Det ses at sætningen kan anvendes ved $m = a$. Derved fremkommer:

$$m \cdot (1)^k \equiv m \ (mod\ n)$$

Og uanset hvad k er her, vil det give samme resultat. Det kan altså konkluderes at:

$$m \equiv c^d \ (mod\ n)$$

[47] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 153
[48] Tilborg, H. C. (1999) Fundamentals of Cryptology. Eindhoven University of Technology: Kluwer Academic Publishers, s. 153

8.5 Bilag 5 - Eksempler på tværsum som hashingfunktion

Eksempel 1 - Find en besked med et bestemt hash

Vi har givet at den hemmelige besked har hashet (og dermed tværsummen) 4. Vi vil gerne sende en bestemt besked, nemlig 123. Da tværsumsfunktionen er nem at forudse ved vi at dette må give en tværsum på 6. Vi kan derfor regne baglæns og se at vi må skulle tilføje et tal x så $6 + x = 13$. Dette ville nemlig til slut give en tværsum på 4. Vi finder dette tal til at være 7. Vi kan derfor sætte det bagpå. Vi sender altså beskeden 1237. Men vi ville kun sende 123.

For at ordne dette må vi analysere situationen. Vi ved at modtageren læser beskeden på en hvid baggrund. Vi sætter derfor 7-tallet til at være hvidt. Det giver samme hash som hvis det var sort, men modtageren kan ikke se det - og vi har dermed vores besked, nemlig 123.

I virkeligheden udnytter man at alle filer indeholder en masse metadata - information om filen. For eksempel hvornår den er skabt, hvornår den sidst er redigeret, og så videre. Ved at sætte dette til noget bestemt ville man, hvis der blev brugt en dårlig hashingfunktion, kunne genskabe ovenstående situation.

Eksempel 2 - Find to beskeder, der giver samme hash

Vi vil gerne betale to arbejdere. Den ene har krævet at få 150 kroner i løn mens den anden blot har krævet at få lige så meget som den anden - uanset beløbet. Han kræver dog dokumentation på at de får samme løn i form af et tværsumshash. Vi sender hver af dem en check, kun indeholdende beløbet, de skal have udbetalt. Den ene får de 150 kroner som krævet. Vi ser dog at nullet i 150 ikke har nogen indflydelse på tværsummen. Vi sender ham derfor blot en check på 15 kroner.

Begge er tilfredse, men den ene er blevet snydt for 135 kroner.

Eksempel 3 - Find en besked, der giver samme hash som en givet besked

En fjendtlig spejder har sendt besked tilbage til sin hærfører med antallet af soldater i vores hær samt et tværsumshash af dette. Vi fanger beskeden under transport og vil gerne have hærføreren til at tro, der er flere end spejderen har observeret. Beskeden er ikke krypteret. Den er kun verificeret ved hjælp af hashet. Ved hjælp af samme princip som før, nemlig at 0 ikke har nogen indflydelse på tværsummen, tilføjer vi et par nuller.

8.6 Bilag 6 - Oversigt over bitoperationer[49]

AND

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

XOR

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

NOT

a	$\neg a$
0	1
1	0

ROL

a	b	$a \lll b$
011	1	110
011	2	101

[49] Truth table, Wikipedia. Fundet 12/18/2017 på https://en.wikipedia.org/wiki/Truth_table

8.7 Bilag 7 - Oversigt over SHA1's konstanter[50]

Algoritmekonstanter:

$$H_1 = 0\text{x}67452301$$

$$H_2 = 0\text{xEFCDAB89}$$

$$H_3 = 0\text{x}98\text{BADCFE}$$

$$H_4 = 0\text{x}10325476$$

$$H_5 = 0\text{xC3D2E1F0}$$

Rundekonstanter:

$$y_1 = 0\text{x}5\text{A}827999$$

$$y_2 = 0\text{x}6\text{ED}9\text{EBA}1$$

$$y_3 = 0\text{x}8\text{F}1\text{BBCDC}$$

$$y_4 = 0\text{xCA}62\text{C}1\text{D}6$$

Reduktionsfunktioner:

$$f(u,v,w) = (u \wedge v) \vee \big((\neg u) \wedge w\big)$$

$$g(u,v,w) = (u \wedge v) \vee (u \wedge w) \vee (v \wedge w)$$

$$h(u,v,w) = u \oplus v \oplus w$$

[50] Smart, Nigel. (2004) Cryptography: An introduction (3rd edition). University of Bristol: McGraw-Hill College, s 157-159

8.8 Bilag 8 - Program kørt med debug-mode

```
lukas@DESKTOP-QQ0JQFM MINGW64 /G/Google Drive/Skole/2017-2018 (3.g)/SRP
$ node hashing.js SRP debug
chars
["S","R","P"]

ascii
[83,82,80]

binary
["1010011","1010010","1010000"]

binary string
"101001110100101010000"

binary string padded
"10100111010010101000010000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000011"

binary array
["10100111010010101000010000000000","00000000000000000000000000000000","0000000000000000000000
0000000000","00000000000000000000000000000000","00000000000000000000000000000000","00000000000
000000000000000000000","00000000000000000000000000000000","00000000000000000000000000000000","
00000000000000000000000000000000","00000000000000000000000000000000","000000000000000000000000
00000000","00000000000000000000000000000000","00000000000000000000000000000000","0000000000000
0000000000000000000","00000000000000000000000000000011"]

binary array decimal
[2806678528,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3]

full binary array
[2806678528,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,-1488288768,0,3,-1488288768,0,3,-1488288768,3,-14882
88765,-1488288768,0,3,-1488288768,0,3,-1488288768,0,3,-1488288765,-1488288768,3,-1488288765,-1
488288768,0,0,0,3,-1488288765,-1488288768,0,0,3,-1488288765,-1488288765,-1488288768,3,-1488288
768,3,-1488288765,-1488288765,-1488288765,-1488288765,-1488288768,3,-1488288765,-1488288768,3,
-1488288768,0,0,0,3,-1488288768,0,0,0,3,-1488288768,3,-1488288765,-1488288768,3,-1488288768,0]


ABCDE
{"A":4914816306,"B":4914816306,"C":-1992521396,"D":-1992521396,"E":-1992521396}

output
[2352433203,871301098,4864829002,2574179778,5587823420]

hex result
["8c374833","33eeffea","121f7664a","996eddc2","14d0f6b3c"]

Dit hash er: 8c37483333eeffea121f7664a996eddc214d0f6b3c
```

8.9 Bilag 9 - Programmets kildekode

For at køre programmet, er følgende nødvendigt:

- *Npm og Node.JS skal installeres (kan findes på https://nodejs.org)*
- *Efter at have navigeret ind i mappen hvor hashing.js ligger, skal følgende kommando køres:*

npm install bitwise-rotation

- *Programmet kan herefter køres med følgende kommando*

node hashing.js <input> [debug]

Hvor <input> er det, der skal hashes og debug er valgfrit (alt efter om du vil se debug-output)

hashing.js

```
const bitwiseRotation = require('bitwise-rotation');
const rol = bitwiseRotation.rolInt32;

// Utility-funktion til at skrive objekter eller arrays til konsollen
function log(label, out) {
	if(debug) {
		console.log(label + '\n' + JSON.stringify(out) + '\n');
	}
}

// Utility-funktion til at lave xor på to strenge
function xor(a, b) {
	return (a != b) ? '1' : '0';
}

// De tre algoritmefunktioner defineret som en del af MD4. Bliver også brugt
i SHA1.
// Fra Introduction To Cryptology, s. 156
function f(u, v, w) { return (u & v) | ((~u) & w); }
function g(u, v, w) { return (u & v) | (u & w) | (v & w); }
function h(u, v, w) { return (u ^ v ^ w); }

// Vi sikrer at brugeren har givet os mindst ét ekstra parameter
// ('node' og 'hashing.js' ligger også i process.argv-arrayet, da det er
kommandoen til at køre programmet)
if(process.argv.length < 3 ) {
	console.log('FEJL: Du mangler et input!');
	return;
}

let debug = false;

// Slå debug til, hvis brugeren har givet et fjerde parameter 'debug'
if(process.argv[3] == 'debug') {
	debug = true;
}

// Vi opsætter nogle konstanter til algoritmen. Dette er for at den har noget
at operere på.
```

```
// Det er dog vigtigt at vi IKKE ændrer disse efter at have fastsat dem - så
ville algoritmen give et andet output for det samme input.
// Disse tal er de samme som for SHA-1 algoritmen.
const algorithmConstants = [
      1732584193, // 0x67452301
      251452088,  // 0xEFCDAB89
      2562383102, // 0x98BADCFE
      271733878,  // 0x10325476
      3285377520  // 0xC3D2E1F0
];

// Derudover definerer vi 4 rundekonstanter
const roundConstants = [
      1518500249, // 0x5A827999
      1859775393, // 0x6ED9EBA1
      2400959708, // 0x8F1BBCDC
      3395469782, // 0xCA62C1D6
];

const input = process.argv[2];

// Inputtets længde, repræsenteret binært
const inputLength = input.length.toString(2);

// Vi deler inputtet op i individuelle karakterer
const chars = input.split('');

log('chars', chars);

// Og laver et array indeholdende ascii-koden til hver karakter
const ascii = [];
for(let i = 0; i < chars.length; i++) {
      ascii.push(chars[i].charCodeAt());
}

log('ascii', ascii);

// Vi konverterer hver ascii-kode til sin binære repræsentation.
const binary = [];
for(let i = 0; i < ascii.length; i++) {
      // Number.toString(n) lader os konvertere et til et hvilket som helst
talsystem - vi vælger her base 2 (binær)
      binary.push(ascii[i].toString(2));
}

log('binary', binary)

// Sammensæt listen af binære værdier til én streng
let binaryString = binary.join('');

log('binary string', binaryString);


// Denne algoritme understøtter kun gennemgang af én blok.
if(binaryString.length > 512 - inputLength.length - 1) {
      console.log('FEJL: For lang besked');
      return;
```

```
}

// Tilføj et 1 for at markere at vores besked er slut
binaryString += '1';

// Sørg for at længden af den binære streng går op i 512 (algoritmens
bloklængde)
// Fyld op med nuller indtil det passer.
while(binaryString.length % (512-inputLength.length) != 0) {
      binaryString += '0';
}

binaryString += inputLength;

log('binary string padded', binaryString);

// Del den binære streng op i dele der har samme længde som vores
algoritmekonstanter.
// Vi udnytter her et regex, se:
https://stackoverflow.com/questions/12686746/split-a-string-at-every-nth-
position-with-javascript
let binaryArray = binaryString.match(/.{1,32}/g);

log('binary array', binaryArray);

// Konverter hvert ord til decimal
binaryArray = binaryArray.map(binary => parseInt(binary, 2));

log('binary array decimal', binaryArray);

// Vi har nu 16 ord. SHA1 bruger dog 80. Disse resterende ord (index 16-79)
bliver nu genereret.
for(let j = 16; j < 80; j++) {
      binaryArray.push(rol(binaryArray[j-3] ^ binaryArray[j-8] ^
binaryArray[j-14] ^ binaryArray[j-16]));
}

log('full binary array', binaryArray);

// Selve hashing-algoritmen foregår egentlig her
// Vi starter med at lave en kopi af algoritmekonstanterne - det ender med at
være vores output
const output = algorithmConstants.slice();

let A = algorithmConstants[0];
let B = algorithmConstants[1];
let C = algorithmConstants[2];
let D = algorithmConstants[3];
let E = algorithmConstants[4];

let t;

// De 80 runder køres samtidig
// Da den eneste forskel mellem runderne er hvordan t bliver sat, kan resten
blot ekstraheres ud i et ydre lag.
for(let j = 0; j < 80; j++) {
      // Vi kigger på hvilken runde vi er nået til
```

```
      if(j < 20) {
            t = rol(A, 5) + f(B, C, D) + E + binaryArray[j] +
roundConstants[0];
      } else if(j < 40) {
            t = rol(A, 5) + h(B, C, D) + E + binaryArray[j] +
roundConstants[1];
      } else if(j < 60) {
            t = rol(A, 5) + g(B, C, D) + E + binaryArray[j] +
roundConstants[2];
      } else {
            t = rol(A, 5) + h(B, C, D) + E + binaryArray[j] +
roundConstants[3];
      }

      // Vi opdaterer værdierne
      A = t;
      B = A;
      C = rol(B, 30);
      D = C;
      E = D;
}

log('ABCDE', {A,B,C,D,E});

// Vi konverterer alle tallene til unsigned integers (så det kun kan blive
positivt)
output[0] = output[0] + (A >>> 0);
output[1] = output[1] + (B >>> 0);
output[2] = output[2] + (C >>> 0);
output[3] = output[3] + (D >>> 0);
output[4] = output[4] + (E >>> 0);

log('output', output)



// Vi konverterer hver byte til et tal
const hexResult = output.map(decimal => decimal.toString(16));

log('hex result', hexResult);

// Og sammensætter tallene til det endelige 12-cifrede hash.
const hash = hexResult.join('');

    console.log('Dit hash er: ' + hash);
```