Opgaveformulering

- Redegør for talteorien bag RSA kryptering. Forklar nødvendigheden af primtal og hvordan RSA metoden fungerer.
- Gør kort rede for hashing.
- Analyser forskellen på hashing og kryptografi og hvilken rolle de har i forhold til sikkerhed.
- Vurder hvorfor RSA er vigtigt i forbindelse med sikkerhed ved logins.

Studieområdeprojekt 2019

Sikkerhed bag login-formular på en hjemmeside

Jens Tinggaard

Resumé

I løbet af dette projekt, vil talteorien bag RSA-kryptosystemet blive gennemgået, for at kunne lave en vurdering af sikkerheden ved metoden. Derudover, vil metoden hashing blive sammenlignet med RSA, der vil blive udpeget fordele og ulemper ved begge metoder, samt vist hvor man bruger begge metoder på en gang. Slutteligt vil der være en vurdering af nødvendigheden af RSA ved logins.

Det vil vises hvordan kryptosystemer, er nødvendige for den digitale hverdag vi lever i. Det vil også vises hvordan RSA både kan bruges til at kryptere data, men også bruges til online autentificering. Et konkreteksempel af RSA vil blive gennemgået, med forklaring af protokollerne som indgår i processen. Samt en forklaring af hvad man kan gøre for at øge onlinesikkerheden, både set ud fra brugerens synspunkt, men også som en systemadministrator.

Vejledere: Søren Søndergaard Eriksen & Lena Erbs
Fag: Matematik A & Programmering B
Odense Tekniske Gymnasium
20. december 2019

Indholdsfortegnelse

Indledning / Motivation 2

1 Talteori 3
1.1 Division med rest og divisor 3
1.1.1 Modulo 3
1.2 Fælles divisor 4
1.2.1 Euclids algoritme 5
1.3 Primtal 6
1.4 Eulers ϕ-funktion 6
1.5 Sætninger bag RSA 7

2 Hashing 9
2.1 Hashfunktioner 9
2.2 Fordelen ved hashing 9
2.3 Hashtabeller 10

3 Forskel på hashing og kryptografi 11
3.1 Alice & Bob 11
3.2 RSA i praksis 12
3.3 Kodeord 12
3.3.1 Hashcat 12
3.3.2 Rainbow tables 13
3.3.3 Salt 13
3.4 Online autentificering 14
3.4.1 GNU Privacy Guard 14
3.4.2 Certifikater 15
3.5 Delkonklusion 15

4 Hvorfor RSA er vigtig 16
4.1 Matematisk protokol for RSA 16
4.1.1 Bevis for RSA 16
4.2 RSA i aktion 17
4.3 UNI•Login 19
4.4 Hvis vi ikke havde RSA 19

Konklusion 20

Litteratur 21

Appendiks A Beviser 23
A.1 Bevis for sætning 1.11 23
A.2 Bevis for RSA ved $(m, n) \neq 1$ 23

Appendiks B Python kode 24
B.1 euclid.py 24
B.2 prime_factor.py 25
B.3 phi_func.py 26
B.4 eulers_sent.py 27

Indledning / Motivation

Uden nødvendigvis at være klar over det, er vi alle afhængige af kryptografi, det er en helt fundamental ting af vores hverdag.
Kryptografi er grunden til at man kan handle på nettet, uden at få stjålet sine kreditkortoplysninger og kryptografi er grundlaget for kryptovalutaer, som bitcoin.[1]
Hashing er også en stor del af den online-sikkerhed, som vi alle er så afhængige af. Hashing bruges til at ændre i kodeord, sådan at det ikke er så ligetil for en hacker, at bryde ind på en konto.
Derudover medvirker både kryptografi og hashing, til lave online autentificering og digitale signaturer, som bruges til at validere ægteheden af personer og/eller dokumenter. Det er måske ikke noget man tænker så meget over, men det virker umiddelbart langt nemmere at lave dokumentforfalskning online, da man blot kan ændre et navn eller lignende. Heldigvis sætter kryptografi også en stopper for dette.
Det skal ses hvordan man bruger hhv. RSA og hashing i forbindelse med den online sikkerhed. Samt hvilken matematik der ligger bag. Slutteligt ses det også hvorfor RSA er så vigtigt i hverdagen.

RSA

RSA er en metode, brugt til at kryptere data, døbt efter sine tre stiftere: *Ron Rivers*, *Adi Shamir* og *Len Adleman*, tilbage i 1977.[xol] Metoden er baseret på antagelsen om, at det er svært at primtalsfaktorisere et stort tal. Talene anvendt ligger typisk i intervallet 2^{1024} til 2^{4096}. Hvilket svarer til tal mellem $\approx 1.798 \cdot 10^{308}$ og $\approx 1.044 \cdot 10^{1233}$.
Det er altså nogle meget store tal, man har med at gøre, når man skal kryptere data med RSA

[1] https://bitcoin.org/

1 Talteori

Talteorien beskæftiger sig med alle de hele tal $\mathbb{Z}$. Vi skal i løbet af dette afsnit kigge på den talteori, som ligger til grunde for at RSA virker i praksis.
Det er ikke alle definitioner og sætninger der bliver bevist, da det ville tage for lang tid. Alle beviser kan dog findes i *Kryptografi – fra viden til videnskab*. [LN97]
Først vil nogle basale begreber i talteorien blive gennemgået, hvorefter vi skal kigge på de sætninger, der ligger til grunde for RSA.

1.1 Division med rest og divisor

I stedet for at regne brøker som rationelle mængder $\mathbb{Q}$ eller decimaltal, skal vi beskæftige os med hvordan man regner med rester. Det vil altså sige at vi udelukkende vil beskæftige os med heltal $\mathbb{Z}$.
Et tal $a \neq 0$ kan enten gå op i b eller ikke gå op i b. Vi definerer at a går op i b, hvis der ingen rest er, efter division. Eller sagt på en anden måde:

Definition 1

Givet tallet $a \neq 0$ og tallet $b \geq a$, vil a gå op i b, hvis der findes et heltal q, som opfylder $b = q \cdot a$.
Det skrives $a \mid b$, som betyder at a går op i b.
a betenes som *divisoren* eller en *faktor*, mens q kaldes for *kvotienten*.[LN97, s. 70]

Er der i stedet en rest efter division, vil man kunne skrive det som $a \nmid b$, som betyder at a ikke går op i b.

Sætning 1.1

Givet tallet $a > 0$ og tallet b, findes der specifikke tal q og r, som opfylder $b = q \cdot a + r$, hvor $0 \leq r < q$.

1.1.1 Modulo

Når man laver division med heltal, vil man ofte kigge på resten efter division. Derfor er der naturligvis lavet en matematisk operator til at udregne rest efter division – den kaldes for *modulo*. Skriver man f.eks. $a \pmod{n}$, udregner man resten efter division, som angivet i sætning 1.1.

Definition 2

For $m \in \mathbb{Z}$ og $n \in \mathbb{N}$ gælder:

$$n \mid m \iff m \pmod{n} = 0$$

$$r = n - qm = m \pmod{n}$$

Derpå, fyldes nogle vigtige sætninger om regning med modulo.

Sætning 1.2

For $a, b \in \mathbb{Z}, n, t \in \mathbb{N}$

$$(a \cdot b) \pmod n = (a \pmod n) \cdot b \pmod n) \pmod n \tag{1.1}$$
$$a^t \pmod n = (a \pmod n)^t \pmod n \tag{1.2}$$

Bevis. Ifølge sætning 1.1, kan a og b omskrives, hvorefter udtrykket reduceres.

$$\begin{aligned}(a \cdot b) \pmod n &= ((q_1 n + r_1) \cdot (q_2 n + r_2)) \pmod n \\ &= (r_1 r_2 + (q_1 q_2 n + q_1 r_2 + q_2 r_1) n) \pmod n \\ &= r_1 r_2 \pmod n \\ &= (a \pmod n) \cdot b \pmod n) \pmod n\end{aligned}$$

Det lange udtryk i parentesen, ved 2. lighedstegn kunne fjernes, da n, var ganget ind på hvert led. (1.1) er nu bevist, (1.2) følger, ved gentagen anvendelse af (1.1). □

1.2 Fælles divisor

En fælles divisor bruges til at sige noget om sammenhængen mellem to tal.

Definition 3

For tallene $a, b, d \in \mathbb{Z}$, betegnes d som en *fælles divisor*, hvis $d \mid a \wedge d \mid b$.

a og b vil have et endeligt antal fælles divisorer, da et tal højere end enten a eller b naturligvis ikke er en fælles divisor (medmindre a og b begge er 0). Hvorfor der altså også vil være en største fælles divisor.

Definition 4: Største fælles divisor

Den største fælles divisor for tallene a og b, betegnes som (a, b).

Eksempel 1.3

Lad $a = 8$ og $b = 12$.

Det ses at divisorerne i 8 er:	$\pm 1, \pm 2, \pm 4, \pm 8$.
Og det ses at divisorerne i 12 er:	$\pm 1, \pm 2, \pm 3, \pm 4, \pm 6, \pm 12$.
Til fælles har de altså divisorerne:	$\pm 1, \pm 2, \pm 4$.

Dermed bliver $(8, 12) = 4$.

Definition 5: Indbyrdes primiske tal

Hvis $(a, b) = 1$, siges a og b at være indbyrdes primiske.

Som vist, kan det være bøvlet at udregne fælles divisorer for to tal, derfor kommer nu en sætning, til at mindske vanskeligheden lidt.

Sætning 1.4

$(a, b) = (a, b \pmod{a})$

Sætning 1.4 bevises ikke, i stedet vil Euclids algoritme (som bygger på sætningen) blive vist.

1.2.1 Euclids algoritme

Euclids algoritme er essentielt gentagen anvendelse af sætning 1.4. Algoritmen bruges til at udregne største fælles divisor for $a, b \in \mathbb{N}$.

Definition 6: Euclids algoritme

Lad $a, b \in \mathbb{N}$, hvor $a \geq b$. Hvis $a \mid b$, så er $(a, b) = b$. Hvis $a \nmid b$, bruges følgende algoritme:

$$\begin{aligned}
a &= bq_0 + r_0 && 0 < r_0 < b \\
b &= r_0q_1 + r_1 && 0 \leq r_1 < r_0 \\
r_0 &= r_1q_2 + r_2 && 0 \leq r_2 < r_1 \\
r_1 &= r_2q_3 + r_3 && 0 \leq r_3 < r_2 \\
&\vdots
\end{aligned}$$

Denne proces slutter, idet der findes en rest på 0. Det vil ske efter et endeligt antal gennemløb t.

$$\begin{aligned}
r_{t-2} &= r_{t-1}q_t + r_t && 0 \leq r_t < r_{t-1} \\
r_{t-1} &= r_tq_{t+1} + 0
\end{aligned}$$

Den sidste rest, som ikke er 0, er dermed den største fælles divisor.[Hun97, s. 11]

Eksempel 1.5 (Anvendelse af Euclids algoritme)

Lad $a = 1048$ og $b = 672$. Ved brug af Euclids algoritme, findes da største fælles divisor.

$$\begin{aligned}
(1048, 672) &= (376, 672) && \text{idet} && 1048 = 1 \cdot 672 + 376 \\
&= (376, 296) && - && 672 = 1 \cdot 376 + 296 \\
&= (80, 296) && - && 376 = 1 \cdot 296 + 80 \\
&= (80, 56) && - && 296 = 3 \cdot 80 + 56 \\
&= (24, 56) && - && 80 = 1 \cdot 56 + 24 \\
&= (24, 8) && - && 56 = 2 \cdot 24 + 8 \\
&= (0, 8) && - && 24 = 3 \cdot 8 + 0 \\
&= 8
\end{aligned}$$

Appendiks B.1, viser en implementering af Euclids algoritme i Python og kan køres således (kræver at Python er installeret[2]):

[2] `https://www.python.org/downloads/`

```
$ python euclid.py a b
```

1.3 Primtal

Primtal viser sig at være interessante, i beskæftigelsen med talteori og divisorer.

Definition 7: Primtal

Primtal er heltal, større end 1, som kun går op i sig selv og 1.

Der findes algoritmer, som kan vurdere med rimelig stor sikkerhed, om et gevent tal er et primtal.[Ves07, s. 21] Derudover er der lavet lange lister over kendte primtal.[3]

Sætning 1.6

Denne sætning er også kendt som 'Algebraens fundamentalsætning"
Alle positive heltal, kan skrives som et produkt af primtal. Denne faktorisering er unik, hvis man ser bort fra rækkefølgen af primtallene.

Bevis. Dette bevis bygger naturligvis på definitionen af primtal. Antag at n ikke er et primtal, det kan da skrives som et produkt af to tal $n = n_1 + n_2$. Er n_1 et primtal, forbliver det uberørt fremover, ellers faktoriseres det endnu en gang. Denne process fortsætter, frem til alle produkter er primtal. □

Appendiks B.2, kan udregne primtalsfaktorerne for et gevent tal n:

```
$ python prime_factor.py n
```

Det ses da, at der findes uendeligt mange primtal, som følge af sætning 1.6. Samt at man for at tjekke om et tal n er et primtal, ikke behøver at tjekke for divisoere større end $\sqrt{n}$.

1.4 Eulers ϕ-funktion

Euler har fremskrevet en funktion, der ud fra et givet tal n, kan bestemme mængden af tal, som er indbyrdes primiske med n. Først betragtes mængden $\mathbb{Z}_n$. Som er angivet ud fra $n \in \mathbb{N}$.

$$\mathbb{Z}_n = \{0, 1, 2, 3, \dots, n-1\}$$

Dette sæt repræsenterer nu alle de mulige rester efter division med n, altså $\pmod{n}$. Der laves nu et nyt sæt bestående udelukkende af tallene, som er indbyrdes primiske med n.

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n | a \neq 0 \land (a, n) = 1\}$$

Definition 8

Eulers ϕ-funktion, siges nu at være antallet af elementer i sættet $\mathbb{Z}_n^*$.

[3] https://primes.utm.edu/lists/small/millions/

Eksempel 1.7

$$n = 14 \qquad \mathbb{Z}_{14}^* = \{1, 3, 5, 9, 11, 13\} \qquad \phi(14) = 6$$
$$n = 9 \qquad \mathbb{Z}_{9}^* = \{1, 2, 4, 5, 7, 8\} \qquad \phi(9) = 6$$

Det er bevist at Eulers ϕ-funktion kan defineres ud fra alle primfaktorerne i n. Det er ikke et bevis som vil blive gennemgået her.

Sætning 1.8

Givet $p_1, p_2 \dots p_r$, som er alle primtalsfaktorerne i n.

$$\phi(n) = n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_r}\right)$$
$$\Updownarrow$$
$$\phi(n) = (p_1 - 1) \cdot (p_2 - 1) \cdot \ldots \cdot (p_r - 1)$$

Det viser sig, at hvis n er et primtal, eller et produkt heraf, er udregningen nem.

Eksempel 1.9

Lad p og q være primtal. Det haves da at.

$$\phi(p) = p - 1 \tag{1.3}$$
$$\phi(pq) = (p - 1)(q - 1) \tag{1.4}$$

Dette vil vise sig nyttigt ved senere brug.

Appendiks B.3, er en implementering af Eulers ϕ-funktion:

```
$ python phi_func.py n
```

1.5 Sætninger bag RSA

Vi skal nu kigge på de sætninger, som bliver direkte anvendt i RSA metoden. Mange af disse sætninger, bygger på sætninger, som ikke er bevist og måske endda ikke nævnt, da de alle bygger på hinanden. Der henvises dog til appendiks A, hvor udvalgte sætninger bevises. Alternativt står beviserne naturligvis i *Kryptografi – fra viden til videnskab*. [LN97]

Sætning 1.10

For $a \in \mathbb{Z}$ og $n, t \in \mathbb{N}$ hvor $(a, n) = 1$, gælder det at:

$$a^t \pmod{n} = a^{t \pmod{\phi(n)}} \pmod{n}$$

Bevis. t angives på formen $t = q \cdot \phi(n) + r$, hvorved $r = t \pmod{\phi(n)}$. Det bevises nu at $a^t \pmod{n} = a^r \pmod{n}$.

$$a^t \pmod{n} = a^{q \cdot \phi(n) + r} \pmod{n}$$

Først deles summen af potensen op i to dele.

$$a^t \pmod{n} = (a^{\phi(n)})^q \cdot a^r \pmod{n}$$

Ifølge sætning 1.2, kan et produkt splittes op som følger.

$$a^t \pmod{n} = \Big[\big((a^{\phi(n)} \pmod{n})^q \pmod{n}\big) \cdot \big(a^r \pmod{n}\big)\Big] \pmod{n}$$

Og da $a^{\phi(n)} \pmod{n} = 1$ (fordi $(a, n) = 1$) [LN97, s. 90], fås

$$a^t \pmod{n} = \Big[\big(1^q \pmod{n}\big) \cdot \big(a^r \pmod{n}\big)\Big] \pmod{n}$$

Da $1^q \pmod{n} = 1$, kan dette blot udelades, hvorved det ses at.

$$a^t \pmod{n} = a^r \pmod{n}$$

□

En anden vigtig sætning for RSA, er sætningen om inverse elementer. Den lyder som følger:

Sætning 1.11

For $a \in \mathbb{Z}_n$, for hvor det gælder at $(a, n) = 1$, vil a have et entydigt inverst element $\pmod{n}$.[LN97, s. 93]
Eller sagt på en anden måde:
Der vil være en entydig løsning til ligningen $ax \pmod{n} = 1$, for $x \in \mathbb{Z}_n$.

Sætning 1.11 har et langt bevis, som bygger på tidligere ikke-dokumenterede sætninger, hvorfor sætningen ikke bevises her. Det kan dog stadig findes i appendiks A.1.
Python koden i appendiks B.4, kan udregne inverse elementer:

```
$ python eulers_sent.py a n
```

2 Hashing

Hashing er en metode, brugt til at omdanne data til en tekststreng, med en fikseret længde. Den nye tekststreng vil være *pseudorandom*[4] Hashing er ikke "hemmelig", forståset ved, at alle kan hashe data og få det samme output (ved samme input naturligvis). Outputtet efter en hashfunktion, omtales som et *hash* (eller *digest* på engelsk).
Derudover er hashing en envejsfunktion, hvilket betyder, at man altså ikke gå baglæns, hvis man kun har et digest af noget data.[xol]

2.1 Hashfunktioner

Der findes en del forskellige hashfunktioner, mange af dem er også implementeret i Python biblioteket[5] "hashlib"[6]. De forskellige hashfunktioner varierer i længden af output og dermed også hastighed. Den efterhånden gamle hashfunktion `MD5`, giver et output på 128-bit, mens nyere funktioner som `SHA256` og `SHA384` giver henholdsvis 256 og 384-bit, som navnene antyder. Det skal senere ses, hvorfor en hashfunktion med et langt output er bedre.
Ønsker man at hashe *abc* i `SHA256` i Python, gøres følgende.

```
>>> import hashlib
>>> hashlib.sha256('abc'.encode()).hexdigest()
'ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad'
```

2.2 Fordelen ved hashing

Det at hashing kun går en vej, virker måske lidt nytteløst, da man ikke længere er i stand til at finde ud af hvad man har hashet. Det man gør, er at man bruger hashing til at gemme kodeord i en database, sådan at hvis nogle kodeord bliver lækket, er de stadig beskyttede, da de er hashede.[Møl17a]
Måden man anvender dette på, er at hashe et kodeord inden det bliver puttet ind i databasen. Når så en bruger forsøger at logge ind, hasher man igen det indtastede kodeord og hvis hashet stemmer over ens, med det liggende i databasen, kan man formode at brugeren har indtastet det rigtige kodeord.

Eksempel 2.1 (Hashing ved brug af hashfunktionen `MD5`)

Det ses at, på trods af lignende input, er hashet helt forskelligt.

```
MD5(abc) = 900150983cd24fb0d6963f7d28e17f72
MD5(Abc) = 35593b7ce5020eae3ca68fd5b6f3e031
MD5(abd) = 4911e516e5aa21d327512e0c8b197616
```

For at forstå hvorfor en langsom hashfunktion er god, er det vigtigt at forstå essensen i en hashfunktion. Det at den kun kan bruges en vej, gør nemlig at hvis man gerne vil finde ud af den originale tekst til et hash, er man nødt til at prøve sig frem, i computerterminologi kaldes dette for *brute-force*[7].

[4] Fra engelsk: Fremstå tilfældigt, men i virkeligheden være forudsætlig.
[5] I computerterminologi er et bibliotek, noget kode, som andre har skrevet som kan implementeres i brugerens egen kode.
[6] `https://docs.python.org/3/library/hashlib.html`
[7] Når man prøver alle kobinationer af en fremgangsmetode, for at finde frem til det rigtige svar.

Eksempel 2.2 (Kompleksiteten ved brute-force)

Antag at man har et hash, hashet i `MD5`, hvor man gerne vil finde den oprindelige tekst. Man ved at teksten består af 8 tegn – som kan være både store og små bogstaver samt tal. I det danske alfabet er der 29 bogstaver og disse kan både fremstå stor og små $2 \cdot 29 = 58$, derudover er der 10 forskellige tal at tage hensyn til $58 + 10 = 68$. Da alle tegn har mulighed for at stå på alle 8 pladser, er der nu $68^8 \approx 4.572 \cdot 10^{14}$ unikke kombinationer. En moderne computer kan udregne omkring $4 \cdot 10^9$ hashes i sekundet, i `MD5`. Mens en lille server kan yde omkring 10 gange så meget $4 \cdot 10^{10}$.[Pou16] Det udregnes hvor mange sekunder det vil tage at hashe alle kombinationer af 8 bogstaver og tal, for henholdsvis en computer og en lille server.
Det er dog vigtigt at understrege, at man i dag aldrig bør bruge `MD5`, da metoden er gammel og har vist sige at være for svag til nudagens hardware.[8]

$$\frac{4.572 \cdot 10^{14}}{4 \cdot 10^9} = 114300 \qquad \frac{4.572 \cdot 10^{14}}{4 \cdot 10^{10}} = 11430$$

Det omskrives til timer.

$$\frac{114300}{60 \cdot 60} = 31.75 \qquad \frac{11430}{60 \cdot 60} = 3.175$$

Det kan altså konkluderes at det er klogt at vælge et kodeord på mere end 8 tegn. Samt at en god hashfunktion tager lang tid, hvorved det forstås at funktioner som `SHA256` og `SHA512` er at foretrække over `MD5`.
Derudover har en hashfunktion med et større output, mindre kollisionssandsynlighed[9] hvilket også er ønsket, for ikke at kunne danne ens hashes, ud fra forskellige inputs.

Det skal senere ses, hvordan man kan brute-force endnu længere hashes, uden at det tager helt så lang tid. Samt hvad man gør for at bekæmpe dette.

2.3 Hashtabeller

En anden ting, som hashfunktioner kan bruges til, er at generere hashtabeller. Dette vil blot blive kort gennemgået, da det ikke er den del af hashfunktioner, som har betydning for onlinesikkerhed.
En hashtabel kan bruges idet man har en masse (hashbart) data, som skal sættes op i en tabel. Når man så skal finde noget af dette data tilbage i tabellen efter at have indsat det, vil man normalt skulle gennemgå alle cellerne i tabellen, for at finde ud af hvor, netop hvor det data man søger er. Det er her hashtabeller kommer ind i billedet.
Meget forsimplet, fungerer en hashtabel ved, at man – inden man insætter dataet i tabellen – hasher dataet og i stedet for at indsætte det i den næste ledige celle, indsætter man det i en celle, svarende til det hash som dataet genererede. Når man så vil tilgå noget data i tabel, hasher man blot dataet igen, for så at lave et opslag i tabellen i den celle tilsvarende til hashet. På den måde sparer man en masse tid, ved ikke at skulle lave et gennemløb af alle celler i tabellen.[Ibs02]

[8] `https://www.kb.cert.org/vuls/id/836068/`
[9] Når 2 forskellige inputs, giver samme hash-digest.

3 Forskel på hashing og kryptografi

Nu da de grundlæggende træk i hashing er blevet vist, skal vi kigge på, hvordan et public-key kryptografisystem, som RSA virker.
Pointen med et public-key kryptosystem er, at et parti har både en offentlig og en hemmelig nøgle. Som navnene antyder, er den offentlige nøgle tilgængelige for alle, mens den private, ikke deles med nogen. De to nøgler er hinandens modsætninger, så at sige. Krypterer man noget med den offentlige nøgle, kan det kun dekrypteres med den tilsvarende private nøgle og vice versa. Det vil altså sige, at hvis parti A, vil sende en krypteret besked til parti B. Tager A blot og krypterer beskeden med Bs offentlige nøgle, hvorefter det kun er Bs hemmelige nøgle, som kan dekryptere beskeden.[Erl04, s. 22]
Det ses ved nogle konkreteksempeler.

3.1 Alice & Bob

Til beskrivelse af en fremgangsmetode, bruger man ofte termen "protokol", som er en arbejdsplan for et handlingsforløb. Hvis der bliver afviget fra protokollen, fortsætter man ikke. Til fremvisning af protokoller i kryptografien, bruger man ofte *Alice* og *Bob* som pladsholdere for aktørere. Det skal ses hvordan RSA anvendes i praksis, samt problemer ved anvendelse af metoden.

Eksempel 3.1 (Generel protokol for kryptografi)

Bob er en server, som Alice gerne vil kontakte. Al kommunikation mellem de to parter, foregår i offentligt rum, så alle og enhver kan læse beskederne sendt mellem dem. For at holde sin besked til Bob hemmelig, tager Alice Bobs offentlige nøgle P_B (Public Bob), og krypterer sin besked med den. Efter dette er gjort, er det kun Bob som kan dekryptere beskeden. Da det kun er ham, som har den hemmelige nøgle S_B (Secret Bob). Nu kan beskeden altså godt sendes over det offentlige rum, uden at Alice skal frygte for at beskeden bliver opsnappet.
Vil Bob gerne sende et svar tilbage til Alice foregår det naturligvis på samme måde, han krypterer sin besked med P_A hvorefter det kun er Alice, som kan dekryptere den med S_A.
Det opskrives som en protokol:

(1) Alice krypterer sin besked med P_B.

(2) Alice sender den krypterede besked ud i det offentlige rum.

(3) Bob opfanger beskeden og dekrypterer den med S_B.

Der er dog visse problemer ved denne form for samtale, da en tredjepart i det offentlige rum *Eve* (eavesdropper[10]), kunne udgive sig for at være Bob. Sådan at når Alice tror hun krypterer sin besked med P_B, krypterer hun den faktisk med P_E. Når beskeden så bliver sendt ud i det offentlige rum, vil Eve kunne læse den, fordi kun hun har den hemmelige nøgle S_E. Er Eve smart, tager hun så – efter at have aflæst beskeden – og krypterer den igen med P_B og sender den så videre, sådan at Bob læser den. På denne måde kan Eve aflytte enhver besked sendt mellem Alice og Bob, uden at hverken Alice eller Bob får mistanke. Denne slags angreb kaldes for *man in the middle*. [Pou17]

[10]Fra engelsk: En person, der smuglytter.

3.2 RSA i praksis

Skal man kryptere en længere besked, et billede, eller måske en hel hjemmeside, bliver det hurtigt en møjsommelig affære, da det at kryptere en lang besked, kræver mange ressourcer. Derfor følger man ikke protokollen fra eksempel 3.1 i praksis, men bruger i stedet en symmetrisk krypterings algoritme (som f.eks. `AES` Advanced Encryption Standard), som er kendetegnet ved, at den bruger samme nøgle til kryptering og dekryptering og samtidig ikke er på så mange bit (128 bit), som en asymmetrisk krypteringsnøgle (som RSA). [xol]
Det vil sige at man blot et nødt til at kryptere den symmetriske krypteringsnøgle med den asymmetriske krypteringsnøgle (RSA nøglen). [LN97, s. 112]
Det opskrives som en ny protokol:

(1) Alice udvælger en tilfældig `AES`-nøgle og krypterer så sin besked med den, udledende c.

(2) Alice krypterer den symmetriske krypteringsnøgle med P_B.

(3) Alice sender nu c afsted til Bob, sammen med den krypterede `AES`-nøgle.

(4) Bob dekrypterer `AES`-nøglen, med S_B og dekrypterer derefter c, med den nu fundne `AES`-nøgle.

3.3 Kodeord

Hashing er som tidligere fortalt, anvendt til at 'hemmeliggøre' kodeord. Det skal ses hvordan man kan forsøge at finde frem til den originale besked, på trods af hashets envejs-egenskab. Hvorved det vises, hvorfor det er vigtigt at bruge et godt kodeord, samt – som udvikler – at bruge en god hashfunktion til at gemme disse.
Hjemmesiden `haveibeenpwned.com`, kan fortælle om hashet af et kodeord er fundet i ved et database-læk og om man derfor bør skifte kodeordet ud.

3.3.1 Hashcat

Hashcat[11] er formentlig et af de allermest brugte værktøjer, til at brute-force hashes. Hashcat er et kommandolinjeværktøj, som udelukkende er bygget til at hashe tekststrenge og sammenligne dem med et (eller flere) hashes. Fordelen ved hashcat er, at man som angriber, kan opsætte mere specifikke regler for, hvordan tekststrengen som man hasher, skal se ud. Det står nærmere beskrevet i dokumentationen.[12] Det skal nu ses, hvordan man kan bruge hashcat til at brute-force kodeord, som set i eksempel 2.2.
Ønsker man f.eks. at finde alle kodeord, som består af 6 små bogstaver, efterfulgt af 4 tal, køres kommandoen

```
$ hashcat -a 3 <hashfil> ?l?l?l?l?l?l?d?d?d?d
```

Det ses da, at i modsætning til eksempel 2.2, vil antallet af hashes, som skal beregnes blot være

$$26^6 + 10^4 = 308,925,776 \approx 3.089 \cdot 10^8$$

[11] `https://hashcat.net/hashcat/`
[12] `https://hashcat.net/wiki/doku.php?id=mask_attack`

Da `?l` specificerer et lille bogstav mellem a-z og ikke a-å, er der blot 26 muligheder og ikke 29. Uanset er det et meget mindre tal, end $4.572 \cdot 10^{14}$. Hvorved det er vist, at man gør klogt i at vælge enten et langt kodeord, eller kodeord med både små bogstaver, store bogstaver, tegn og tal i. Eller i bedste fald, begge.[Møl17b]

3.3.2 Rainbow tables

Der er naturligvis også andre metoder end brute-force, til at forsøge at tilbageskabe kodeord ud fra et hash. Et *rainbow table* er en tabel, hvori der ligger nogle forudberegnede hashes, samt de originale kodeord dertil.[Hau17] Dette gør at man, uden at være nødt til at brute-force alle 68^n muligheder, kan finde finde frem til en brugers kodeord.
Et rainbow table indeholder naturligvis ikke alle hashes – det ville være en uendelig opgave at liste dem alle. Det er altså kun de hashes – og dermed også kodeord, som er mest anvendte, der ligger i et rainbow table. Af samme grund, skal man ikke blot bruge koden `password`, da den helt sikkert ligger i et rainbow table. Det betyder ikke at man er i sikkerhed, ved blot at vælge 8 tilfældige tegn, da de formentlig også er i et rainbow table et sted. Der findes efter håndenrainbow tables som er meget store[13], op til flere GB. I realiteten er rainbow tables noget mere komplekst bygget op, men dette forklarer essensen i dem.

3.3.3 Salt

Som modsvar til rainbow tables, bruger man *salt* sammen med lagringen af kodeord på en server. [Møl17a] Måden man anvender salt på er, at efter brugeren har lavet et kodeord, genererer man et salt – en tilfældig tekststreng. Dette salt konkatenerer[14] man blot med det originale kodeord, inden man hasher det. Dermed vil hash-digestet ikke være identisk med et andet digest af det samme kodeord, da de ikke er hashet med samme værdi. I stedet for blot at gemme et hashet kodeord på serveren, gemmer man nu både det hashede kodeord, samt saltet som er blevet brugt til at generere det. Når en bruger så forsøger at logge ind, kigger man på kodeordet, konkatenerer saltet og hasher det så. Stemmer de to hashes over ens, er det det rigtige kodeord man har fået fat i.
Kigger man fra hackerens side, vil man se at rainbow tables ikke længere er brugbare, da man ikke længere blot kan lave et opslag af et kodeord, fordi det er blevet hashet med dette salt efter. Det vil altså sige, at man ikke længere blot kan forsøge at knække alle kodeordene i en database på én gang. I stedet vil man være nødt til, først at lave et gæt på et kodeord, konkatenere saltet og så hashe det.

Eksempel 3.2 (Kodeord og salt)

Antag at Alice og Bob opretter hver deres konto på `example.com`. Da de begge er ukendte i viden om sikkerhed ved kodeord, vælger de begge kodeordet `kodeord123`. Sitet de opretter sig på, har heldigvis styr på sikkerheden, da de bruger salt, til at hashe deres kodeord.
Når Alice opretter sig, får hun tildelt et salt af værdien `*j2!#6wdGQ`.
Idet Bob opreter sig, får han tildelt salt-værdien `Cc8h*cZ^Kg`.
Det ses nu hvordan saltet bruges til at generere forskellige hashes, af samme kodeord. Det er dog vigtigt at understrege at `MD5` er en gammel hashfunktion, som aldrig bør

[13] `https://project-rainbowcrack.com/table.htm`
[14] Når man sammenkæder to tekststrenge til én.

benyttes til at lagre kodeord. Den anvendes blot her, da den ikke returnerer så mange bits (længden af hashet). Pointen ses dog den samme ved alle hash metoder.

```
MD5(kodeord123*j2!#6wdGQ)   = 2132e21016df4bae1bde175262cefac3
MD5(kodeord123Cc8h*cZ^Kg)   = 552288e992d32eea0b6dd0a57f6e251d
MD5(kodeord123)             = c1b1e28f2d5c35233298c0d0bbb4886d
```

I databasen gemmes nu værdierne

Navn	**Kodeord**	**Salt**
Alice	2132e21016df4bae1bde175262cefac3	*j2!#6wdGQ
Bob	53caf0a58f60602ae7f3d3fc6fb99832	Cc8h*cZ^Kg

3.4 Online autentificering

Skal man autentificere sig online, er der heldigvis lavet en protokol, som gør at man kan validere gyldigheden af en besked.[Tho09, s. 12] Protokollen anvender både hashing samt RSA til at verificere ægteheden af et dokument eller lignende.
Vil Bob gerne sikre sig at en besked stammer fra Alice, kræver det at Alice følger protokollen: Udover at sende beskeden, bruger hun en hashfunktion på beskeden $H(m)$, signerer det så med sin private nøgle S_A og sender så dette krypterede hash y afsted, sammen med den originale besked. Det siges at Alice nu har sat sin *digitale signatur* på beskeden.
Når Bob så modtager beskeden sammen med det krypterede hash, tager han blot og hasher beskeden selv $H(m)$. Og dekrypterer så det tilsendte hash y med Alices offentlige nøgle P_A. Hvis de to hashes stemmer over ens, kan han verificere at beskeden er fra Alice.
Dette virker, da det kun er Alices offentlige nøgle P_A, som kan bruges til at dekryptere y, bruger Bob i stedet Eves offentlige nøgle P_E, vil han ikke få samme hash, hvorfor beskeden ikke er fra Eve.
Protokollen foregår som følger:

(1) Alice hasher beskeden m med en bestemt hashfunktion $H(m)$.

(2) Alice krypterer hashet med sin private nøgle S_A, som afgiver y.

(3) Alice sender så den originale besked m, samt det krypterede hash y afsted.

(4) Når Bob modtager m og y, tager han og hasher beskeden $H(m)$.

(5) Bob dekrypterer y med P_A.

(6) Bob validerer at de to hashes er ens.

Fordelen ved dette er at man ikke behøver at kryptere hele beskeden m, som kan være meget stor. I stedet krypterer man blot den hashede del af beskeden, som jo altid har en fikseret længde.

3.4.1 GNU Privacy Guard

En meget brugt metode til at generere digitale signaturer, er systemet GNU Privacy Guard (gpg)[15]. gpg bruges til at generere nøgler, kryptering af data samt autentificering.

[15] https://gnupg.org/

Når man genererer en nøgle, vælger man først og fremmest størrelsen (mellem 1024 og 4096 bit). Derudover skal man vælge et navn samt en email at 'binde' nøglen til.
Dette kan gøres i gpg ved kommandoen:

```
$ gpg --full-generate-key
```

Efter nøglen er blevet genereret, ligger den som en binær fil, som altså kun er læsbar af computeren. For at konvertere den til noget vi kan forstå, køres kommandoen:

```
$ gpg --export -a
```

Et eksempel på outputtet af denne kommando, kan ses på figur 1. Slutteligt vises det, at man kan signere en fil i gpg. Dette gøres med flaget `clearsign`.

```
$ gpg --clearsign <fil>
```

gpg er altså en stor del af kryptografi, hvis man selv ønsker at kryptere sine data. Det kan f.eks. være idet man udgiver noget vigtigt offentligt. Så vil læserne gerne validere hvem udgiveren er – det kan gøres i gpg.
På downloadsiden til hashcat[16], kan man – udover at downloade kildekoden til hashcat, downloade en digital signatur af kildekoden, ud fra hvilken, man kan vadidere at dowloadet er ægte.


```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mI0EXfDtTQEEALWpPUJ25y4qV0bnDMRRSfWZftBiIdz2YNH90wz6e/DPXU5hsQEa
Au/so7yH2m4+Im6gCDNWFcmsdBsDbHa8mCc5G/pcLnQd7j9RWZMjqa0RHvh8Q1KU
2Sdd2zQHKfrgt5OBU/bkAe0q/HuO/eUthG4/8fEOzy4CSVC6OPRCCia9ABEBAAG0
P0plbnMgVGluZ2dhYXJkIChFeGFtcGxlIGtleSBmb3IgU09QIGV4YW0pIDx0aW5n
Z2FhcmRAeWFob28uY29tPojUBBMBCgA+FiEEK2olsIcMmok7SJU7woFctsWBC/AF
Al3w7U0CGwMFCQHhM4AFCwkIBwIGFQoJCAsCBBYCAwECHgECF4AACgkQwoFctsWB
C/AOKgP8Drshrf2lL+HqzOY0twPNv2jA9542Z2If6HKrb2ekvq5I0dzWlDUe86v1
j37rKpbEXMEOdaWTabP43iud74B62iApN4kbXi87ALvlv69otdogVJkVTnESWFSA
fo9Mbh2/Wcu4xNE8wtgK3oIEfDRQFAsi94uk2who38GzucjyQEm4jQRd8O1NAQQA
2EXG9FfYXpPtH2fnhsM78fYWIlLAtDVKUvhTFOFiRo5wuHQ7sjV2XY4qDKtgUd5x
qmvib93IZkdeu/4HVt/Jqyp9uu1OYS0m5MNz6YvWjvYbE2Qa9mbtxdWxPJ7qht9t
ZgY/2GbUXVqYGVuNO/in6V5v5XkW6kIsc0G29ZJDf8UAEQEAAYi8BBgBCgAmFiEE
K2olsIcMmok7SJU7woFctsWBC/AFAl3w7U0CGwwFCQHhM4AACgkQwoFctsWBC/DV
vQP+JuiYQww4lIfofYfsZtdEZA3ttBQKGSONC+FumEXkF1OvalVr9rpAR/ddqvXZ
3VetuLeDIncw9nT1hc9LER5+MUVCPTUCJiGl2rDiTSvfL+YhGRS+cwIZsR7i1Qps
w3rU2RVHfyTxOCsVcLgV0hgxbPoSWs/c5ojgGcro7ugbHPc=
=y7be
-----END PGP PUBLIC KEY BLOCK-----
```


Figur 1: 1024-bit nøgle genereret i gpg

3.4.2 Certifikater

For at undgå, hele tiden at skulle signere et dokument (eller hjemmeside), inden det deles online, bruger man certifikater.[doc19a] Et certifikat er udstedt af en annerkendt certifikatudbyder. Certitikatet understreger blot at en bestemt offentlig nøgle, faktisk stammer fra den udbyder der står på den. Det vil altså sige, at når man skal have en hjemmeside op at køre, går man til en certifikatudbyder og får sin offentlige nøgle verificeret. Når en bruger så ønsker at tilgå hjemmesiden, sender hjemmesiden blot sin offentlige nøgle tilbage. Browseren laver så et tjek med de kendte certifikatudbydere, for at verificere ægteheden af hjemmesidens offentlige nøgle. Denne protokol for brug af certifikater, bruges for at modvirke angreb som "man in the middle". Samtidig med at den er langt hurtigere end processen beskrevet i afsnit 3.4

3.5 Delkonklusion

Det forstås altså, at kryptografi og hashing begge er meget vigtige for den digitale sikkerhed i dag. Begge metoder har sine fordele og det blev vist hvordan man kan bruge de begge metoder på én gang, til at lave online autentifikation.

[16] `https://hashcat.net/hashcat/`

4 Hvorfor RSA er vigtig

Først vil det kort blive gennemgået, hvordan RSA virker, på matematisk niveau. Hvorefter der vil være et konkreteksempel.

4.1 Matematisk protokol for RSA

Der følges igen i protokol, til fremstilling af den offentlige og den hemmelige nøgle.[Bra15, s. 1]

(1) Udvælg 2 primtal p og q, de skal gerne være omkring de 100 cifre hver og sæt så $n = p \cdot q$

(2) Udregn $\phi(n)$, som er $(p-1)(q-1)$, da p og q er primtal.

(3) Vælg et tal $e \in Z^*_{\phi(n)}$, altså sådan at $0 < e < \phi(n)$ og $(e, \phi(n)) = 1$

(4) Udregn det inverse element d til $e \pmod{\phi(n)}$, sådan at $e \cdot d \equiv 1 \pmod{\phi(n)}$.

Når dette er gjort, er vi klar til at sende hemmelige beskeder.

- Den offentlige nøgle er hhv. n og e.
- Den hemmelige nøgle er d.

Bemærk at p og q udelukkende er med til at generere n og $\phi(n)$ og faktisk er $\phi(n)$ også kun brug til at generere e og d. For at kunne bryde krypteringen, skal man kende d og d kan faktisk genereres ud fra den offentlige nøgle.
RSA bygger altså på antagelsen om, at det tager lang tid at primtalsfaktorisere n.
Den besked m vi ønsker at sende, skal nu opfylde $m < n$. Ellers må den opdeles i mindre bidder.
Beskeden m, som skal sendes, krypteres ved at udregne $c = m^e \pmod{n}$. c er nu den krypterde besked.
For at dekryptere c, udregner man $m = c^d \pmod{n}$

4.1.1 Bevis for RSA

Det skal ses, hvorfor RSA virker, på baggrund af sætninger gennemgået i afsnit 1. Krypteringen opskrives som en lang lighed.[Jan08, s. 83]

$$c^d \pmod{n} = (m^e)^d \pmod{n} = m^{ed} \pmod{n} = m$$

Det sidste lighedstegn bevises.

Bevis. Der er to muligheder: $(m, n) = 1$ og $(m, n) \neq 1$.
Kigger man på den første $(m, n) = 1$, ses det at ifølge sætning 1.10 er $m^{ed} \pmod{n} = m^{ed \pmod{\phi(n)}} \pmod{n}$. Og da $e \cdot d = 1 \pmod{\phi(n)}$, som blev defineret ved step (4) i genereringen af tal. Ses det nu, at $m^1 \pmod{n} = m$ hvorved det er bevist for $(m, n) = 1$.
Beviset for $(m, n) \neq 1$ er noget længere og bygger på en del sætninger, som også tager tid at bevise, derfor bevises det blot hvordan der i forvejen er en meget lille sandsynlighed for at $(m, n) \neq 1$.
Antag at $(m, n) \neq 1$, det må betyde at $p \mid m \vee q \mid m$. Da m er genereret ud af de to

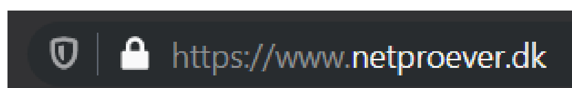
primtal. De tal mellem 1 og n, som gå op i m, kan nu tælles:
p går op i $p, 2p, 3p, \ldots, (q-1) \cdot p$.
q går op i $q, 2q, 3q, \ldots, (p-1) \cdot q$.
Der er derfor i alt $(p-1)+(q-1)+1 = p+q-1$, tal mellem 1 og n som går op i m. Sammenlignes dette med antallet af tal i alt, mellem 1 og n – antaget at p og q begge er på omkring de 100 cifre – haves at.

$$\frac{p+q-1}{p \cdot q} \approx \frac{2 \cdot 10^{100}}{10^{200}} = 2 \cdot 10^{-100}$$

Sandsynligheden for at $(m, n) \neq 1$, er altså minimal.[LN97, s. 106]
Ønskes beviset for at ligheden er sand for $(m, n) \neq 1$, henvises der til Appendiks A.2 $\square$

4.2 RSA i aktion

Vi skal nu tage et kig på hvordan RSA rent faktisk fungerer. Tallene anvendt er naturligvis blot til eksempel – i virkeligheden er de som sagt på omkring 100 cifre hver.
Alice vil gerne logge på `https://www.netproever.dk/`, som Bob (staten) ejer, for at aflevere sin SOP, men på det netværk hun er forbundet til, sidder der nogle lumske typer, som gerne vil have fat i Alices kodeord, sådan at de kan logge ind som hende og aflevere noget bras.
For at forhindre dette benytter Bobs website sig heldigvis af kryptering, hvilket også indikeres af browseren, hvor der står `https` oppe i søgebaren samtidig med at der er en hængelås før URLen – se figur 2. `https` "HyperText Transfer Protocol Secure" er en protokol som lader browseren vide, hvordan den skal handle forespørgsler til websitet. [doc19b] Havde der blot stået `http`, ville forbindelsen ikke være krypteret. Det er altså vigtig at kigge på, inden man logger ind på en hjemmeside, for at undgå identitetstyveri.



Figur 2: Https og hængelås ud for URLen

Det ses nu, hvordan Bob genererer en offentlig og en hemmelig nøgle ud fra protokollen fra afsnit 4.1.

(1) $p = 41$ og $q = 83$, $n = 41 \cdot 83 = 3403$

(2) $\phi(n) = (41-1)(83-1) = 40 \cdot 82 = 3280$

(3) e kan være mange værdier, men sættes i dette tilfælde til $e = 43$

(4) d beregnes til at være 1907, da $1907 \cdot 43 \equiv 1 \pmod{3403}$

Kodeordet, som skal sendes omdannes til tal, ved at sætte $A = 01, B = 02, C = 03, \ldots, Z = 26$, _ sættes til at være 00. Alices kodeord er tilfældigvis "SOP_OM_RSA", som omdannes til tal, vha. nævnte definition. Derudover deles beskeden op, sådan at hver bid, er mindre end n, som er 3403.

S O	P _	O M	_ R	S A
1915	1600	1513	0018	1901

Nu tager Alice så fat i Bobs offentlig nøgle $(43, 3403)$, for at kryptere sit kodeord. For hver bid i hendes kodeord, skal hun udregne $m^{43} \pmod{3403}$. Det ses, at allerede ved første bid, skal hun udregne $1915^{43} \approx 1.35 \cdot 10^{141}$. Hvilket overgår de fleste lommeregneres kapacitet. Heldigvis kan m^{43} omskrives, vha potensregneregler. Hvorefter det er langt nemmere at udregne modulo.

$$\begin{aligned}
m^{43} &= m \cdot m^{42} \\
&= m \cdot (m^{21})^2 \\
&= m \cdot (m \cdot m^{20})^2 \\
&= m \cdot m^2 \cdot (m^{20})^2 \\
&= m \cdot m^2 \cdot ((m^{10})^2)^2 \\
&= m \cdot m^2 \cdot (((m^5)^2)^2)^2 \\
&= m \cdot m^2 \cdot (((m \cdot m^4)^2)^2)^2 \\
&= m \cdot m^2 \cdot ((m^2)^2)^2 \cdot (((m^4)^2)^2)^2 \\
&= m \cdot m^2 \cdot ((m^2)^2)^2 \cdot ((((m^2)^2)^2)^2)^2
\end{aligned}$$

Beregningen udføres nu blot med henhold til sætning 1.2, sådan at der foretages en modulo operation, for hvert potens-led. Det ville fylde alt for meget at vise her, hvorfor det overlades til læseren. Alternativt er der et fint eksempel i. [LN97, s. 104]
Krypteringen udføres, for at få den krypterede besked. Dette kan også gøres i Python med den indbyggede funktion `pow()`[17]

```
>>>pow(base, exp[, mod])
```

$$\begin{aligned}
1915^{43} \pmod{3403} &= 2454 \\
1600^{43} \pmod{3403} &= 944 \\
1513^{43} \pmod{3403} &= 469 \\
18^{43} \pmod{3403} &= 174 \\
1901^{43} \pmod{3403} &= 2637
\end{aligned}$$

Det ses altså, at de krypterede bidder, på ingen måde minder om de originale bidder. Krypteringen er på denne måde lykkedes og Alice kan trygt sende den krypterede besked over netværket. Når beskeden så lander ved Bob, som jo er den eneste, der kender d, vil han kunne dekryptere den, validere at det er Alices' kodeord og derefter logge hende ind. Dekrypteringen foregår som sagt ved at udregne $c^d \pmod{n}$.

$$\begin{aligned}
2454^{1907} \pmod{3403} &= 1915 \\
944^{1907} \pmod{3403} &= 1600 \\
469^{1907} \pmod{3403} &= 1513 \\
174^{1907} \pmod{3403} &= 0018 \\
2637^{1907} \pmod{3403} &= 1901
\end{aligned}$$

Det er hermed vist, ved et eksempel, hvordan RSA virker.

[17] https://docs.python.org/3/library/functions.html#pow

4.3 UNI●Login

Vi skal kigge lidt på sikkerheden ved en almindelig login-formular og se hvilke faktorer der spiller ind. I de sidste mange år, har et UNI-login brugernavn, været de første 4 bogstaver i ens navn, efterfulgt af 4 tal (eller evt. bogstaver). Mens koden har bestået af 3 små bogstaver, 2 tal efterfulgt af atter 3 bogstaver. Men det skal langt om længe laves om. [It 19] Set fra onlinesikkerhedens synspunkt, var det heller ikke et øjeblik for tidligt, da det som vist i afsnit 3.3, er meget let at brute-force sig frem til så korte kodeord. Den nye ordning, gør at der bl.a. skal være *2FA*.[18] Dette er naturligvis med til at øge sikkerheden voldsomt, da der er en variabel med inde over loginnet og ikke længere kun et fast brugernavn + kodeord. Der er hovedsageligt 4 faktorer, som spiller ind på sikkerheden af en login-formular.

1. En krypteret forbindelse, til formularen (RSA)
2. Om loginnet er sat op til 2-faktor login
3. Styrken af brugerens kodeord, herunder længde og kompleksitet
4. Om kodeordet bliver hashet, herunder hvilken hashfunktion, samt brug af salt

Udover faktor nummer 3, kan det – som bruger – være svært at gøre noget ved nogle af disse foranstaltninger. Derfor er det vigtigt at tjekke om siden man besøger er krypteret og så vidt muligt at forsøge at vælge et langt unikt kodeord eller bruge en *password manager*.[19]

4.4 Hvis vi ikke havde RSA

Vores verden ville formentlig gå helt i stå, hvis vi en stod i den uheldige situation, at der blev fremvist en metode til at bryde RSA krypteringen.
Antallet af operationer det tager, de bedste kendte algoritmer at faktorisere tallet n er.[LN97, s. 119]

$$e^{\sqrt{\ln(\ln(n))\cdot\ln(n)}}$$

Dog er der fremvist beviser for, at kvantecomputere, er langt bedre til at primtalsfaktorisere, end nudagens computere.[Rei19] Heldigvis formår de samtidig at skabe en ny kryptering, som de ikke selv kan bryde. Det er dog ikke noget, der vil blive taget yderligere op i dette projekt.
RSA er grundlag for al online-sikkerhed i dag, herunder autentificering, anonymisering og som sikkerhedsforanstaltning. Hvis vi ikke havde RSA, ville ingen internettraffik længere være sikker, man ville ikke kunne stole på noget man læser online længere, da man ikke kan verificere udgiveren. Samtidig ville det være dumt at give sig til kende, da alle på den måde, vil få fat i ens oplysninger på det åbne net.
Vi bør altså være glade for at RSA er opfundet, til at holde os sikre på nettet. Ikke kun i forbindelse med logins, men i den generelle færden på internettet.

[18] Engelsk for *Two Factor Authentication*, betyder at man ikke blot skal logge ind med et fast kodeord, men også have noget seperat, som f.eks. nøglekortet til NemID, eller en engangskode tilsendt på SMS.

[19] En password manager kan selv generere og holde styr på dine kodeord. Kodeordene er så beskyttet af et master kodeord, som du selv skal huske, da det er "nøglen" til alle kodeordene. LastPass og dashlane er begge password managers.

Konklusion

Det er i løbet af projektet blevet vurderet hvordan RSA er vigtig i forbindelse med sikkerheden ved logins. Først ved en redegørelse af talteorien bag RSA, dernæst en kort gennemgang af hashingmetoder. Så blev der analyseret på forskellen mellem kryptografi og hashing, for at runde af med en gennemgang af RSA protokollen og et konkreteksempel heraf.
Ud fra dette, kan det konkluderes at RSA er en del af hverdagen, for alle som bruger internettet dagligt. Hvis der en dag bliver fremskrevet et bevis for, hvordan man nemt kan primtalsfaktorisere et stort tal, vil hele vores verden formentlig gå i stå, da stort set alt hvad vi laver i dag, har forbindelse til internettet på den ene eller den anden måde.
Derudover, er det vist at både RSA, såvel som hashing er vigtige metoder, for sikkerheden online. De har hver deres individuelle fordele og ulemper, men de kan også bruges samlet til verificering af dokumenter.
Samtidig er det vist hvorfor det er vigtigt at være opmærksom på sikkerheden, ved de hjemmesider man besøger. Som bruger har man ikke så stor indflydelse på sikkerheden, hvorfor det er godt at holde øje med tegn på sårbarheder, såsom brug af `http` (ikke-krypteret forbindelse) i stedet for `https` (krypteret forbindelse).
Som administrator af en hjemmeside er det vigtigt at sørge for at have sikkerheden i orden, dette gælder både kryptering såvel som hashing.
I den forbindelse, er RSA altså voldsomt vigtig, vores verden ville ikke være den samme i dag, hvis det ikke var for *Rivers*, *Shamir* og *Adleman*.

Litteratur

[Bra15] Brandt, Max. *RSA Kryptosystemet*. Aarhus Universitet. 2015. URL: https://docplayer.dk/1531172-Rsa-kryptosystemet-kryptologi-ved-datalogisk-institut-aarhus-universitet.html.

[doc19a] docs, MDN web. *Digital certificate - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. mozilla.org. Mar. 2019. URL: https://developer.mozilla.org/en-US/docs/Glossary/Digital_certificate.

[doc19b] docs, MDN web. *HTTPS - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. mozilla.org. Mar. 2019. URL: https://developer.mozilla.org/en-US/docs/Glossary/HTTPS.

[Erl04] Erlandsen, Mikkel Kamstrup. *Introduktion til Kryptologi*. Aarhus Universitet. Sep. 2004. URL: http://math.au.dk/fileadmin/Files/matlaererdag/2005/kryptologi.pdf.

[Hau17] Hauge, Mathias. *Rainbow tables - Angreb - Dansk Data sikkerhed*. danskdatasikkerhed. 2017. URL: http://www.danskdatasikkerhed.dk/rainbow-tables/.

[Hun97] Hungerford, Thomas W. *Abstract Algebra, an indtroduction*. Engelsk. second edition. Brooks/Cole, 1997, s. 11–12. ISBN: 0-03-010559-5.

[Ibs02] Ibsen, Stefan. *Hashing og hashtabeller*. Nov. 2002. URL: https://docplayer.dk/9204254-Hashing-og-hashtabeller.html.

[It 19] It og læring, Styrelsen for. *Styrelsen for It og Læring lancerer nyt Unilogin - Styrelsen for It og Lærings vidensbase - Global Site*. stil.dk. Nov. 2019. URL: https://viden.stil.dk/pages/viewpage.action?pageId=70582342.

[Jan08] Jankvist, Uffe Thomas. *RSA og den heri anvendte matematiks historie - et undervisningsforløb til gymnasiet*. Roskolde Universitet. Jan. 2008. URL: http://milne.ruc.dk/imfufatekster/pdf/460.pdf.

[LN97] Landrock, Peter og Nissen, Knud. *Kryptologi - fra viden til videnskab*. Dansk. 1. udg. Abacus, 1997. ISBN: 87-89182-62-6.

[Møl17a] Møllerhøj, Jakob. *MD5, SHA-1 eller Scrypt: Er dine brugeres kodeord (forsvarligt) krypteret? | version2*. ing/version2. Feb. 2017. URL: https://www.version2.dk/artikel/md5-sha-1-salt-scrypt-har-du-styr-paa-dine-brugeres-kodeord-forsvarligt-krypteret-1072899.

[Møl17b] Møllerhøj, Jakob. *Tjek selv styrken af dine kodeord med Hashcat | Version2*. ing/version2. Feb. 2017. URL: https://www.version2.dk/artikel/tjek-selv-styrken-dine-kodeord-med-hashcat-1072974.

[Pou16] Pound, Mike. *Password Cracking - Computerphile*. YouTube. Jul. 2016. URL: https://www.youtube.com/watch?v=7U-RbOKanYs.

[Pou17] Pound, Mike. *Key Exchange Problems - Computerphile*. YouTube. Dec. 2017. URL: https://www.youtube.com/watch?v=vsXMMT2CqqE.

[Rei19] Reich, Henry. *How Quantum Computers Break Encryption | Shor's Algorithm Explained*. YouTube. Apr. 2019. URL: https://www.youtube.com/watch?v=lvTqbM5Dq4Q.

[Tho09] Thomsen, Søren Steffen. „Cryptographic Hash Functions". Engelsk. PhD afh. 2009, s. 12. URL: https://backend.orbit.dtu.dk/ws/portalfiles/portal/5025771/sst_thesis_v1.0.pdf#subsection.1.2.2.

[Ves07] Vestergaard, Erik. *RSA-kryptosystemet*. 2007. URL: https://matematikfysik.dk/mat/noter_tillaeg/RSA.pdf.

[xol] *Algoritmer*. URL: https://www.xolphin.dk/support/Terminologi/Algoritmer.

Appendiks

A Beviser

A.1 Bevis for sætning 1.11

Beviset bygger på yderligere ikke-beviste sætninger, som lades op til læseren at validere

Bevis. Antag at $(a, n) = 1$, der findes nu en *linearkombination* af a og n, således at $as + nt = 1$, hvilket omskrives til $as = 1 - nt$. Ud fra dette, ses det at $as \pmod n = 1$. Ifølge sætning 1.2, kan dette omskrives til $(a \pmod n \cdot s \pmod n) \pmod n = 1$. Det vides at $a < n$, da $a \in \mathbb{Z}_n$, hvorfor $a \pmod n = a$, derfor er $(as \pmod n) \pmod n = 1$, hvilket medfører at $x = s \pmod n$ er løsning til ligningen $ax \pmod n = 1$.
Nu mangler blot entydigheden af beviset. Antag at begge løsningerne $x_1, x_2 \in \mathbb{Z}_n$, er løsninger til ligningen $ax \pmod n = 1$. Det vil altså sige $ax_1 \pmod n = ax_2 \pmod n$, ud fra dette kan det konkluderes at $x_1 \pmod n = x_2 \pmod n$ og da begge løsninger er mindre end n, haves at $x_1 = x_2$. Hvorved entydigheden er vist.
Sætningen bevises også den anden vej: Antag at x er løsning til ligningen $ax \pmod n = 1$, dette medfører at der findes et tal q, så $ax = qn + 1$. Ud fra dette findes en linearkombination af a og n som giver 1: $x \cdot a + (-q) \cdot n = 1$. Ud fra en anden ikke-bevist sætning, medfører dette at $(a, n) = 1$[LN97, s. 93]

□

A.2 Bevis for RSA ved $(m, n) \neq 1$

Beviset bygger på yderligere ikke-beviste sætninger, som lades op til læseren at validere

Bevis. Da n er et produkt af to primtal, vil det ene af dem være en divisor i m, ellers kan n og m ikke have en fælles divisor større end 1. Det antages at p går op i m og at q ikke gør. Dette medfører at $(q, m) = 1$. Det er nu nok at vise at $m^{ed} \pmod p = m \pmod p$ og $m^{ed} \pmod q = m \pmod q$, for da haves at $m^{ed} \pmod n = m \pmod n = m$ ifølge en ikke-bevist sætning. Da $p|m$ fås.

$$m^{ed} \pmod p = (m \pmod p)^{ed} \pmod p = 0^{ed} \pmod p = 0 = m \pmod p$$

Sætning 1.2 er anvendt ved 2. lighedstegn Idet $(q, m) = 1$, kan der benyttes endnu en ikke-bevist sætning. Som siger at $m^{q-1} \pmod p = 1$, da $ed \pmod{\phi(n)} = 1$, findes der et helt tal k, som opfylder $ed = k \cdot \phi(n) + 1$ Omskrevet giver det at

$$m^{ed} = m^{k \cdot \phi(n)+1} = m \cdot m^{k \cdot \phi(n)} = m \cdot m^{k \cdot (p-1)(q-1)} = m \cdot \left(m^{p-1}\right)^{k \cdot (q-1)}$$

Sætning 1.2 benyttes igen, hvorved det fås at:

$$\begin{aligned} m^{ed} &= \left[m \pmod q \cdot \left(m^{q-1} \pmod q\right)^{k \cdot (p-1)} \pmod q\right] \pmod q \\ &= \left[m \pmod q \cdot 1^{k \cdot (p-1)} \pmod q\right] \pmod q \\ &= m \pmod q \end{aligned}$$

Hvorved det er bevist.[Ves07, s. 19] □

B Python kode

B.1 euclid.py

Filen ligger også vedhæftet i denne pdf, åbn den ved at dobbeltklikke på ikonet.

```python
#Implementering af Euclids algoritme i Python
import sys

def euclid(a,b):

    orig_a = a
    orig_b = b

    #Hvis b går op i a
    if a % b == 0:
        q = a//b #Kvotient

        print('{a} = {b} * {q} + 0'.format(a=a, b=b, q=q))
        print('({a},{b}) = {res}'.format(a=a, b=b, res=b))
        return b


    while True:
        q = a//b #Kvotient
        r = a-(b*q) #Rest

        #Print inden skift af variabler
        print('{a} = {b} * {q} + {r}'.format(a=a, b=b, q=q, r=r))

        #Største fællesdivisor fundet
        if r == 0:

            print('({a},{b}) = {res}'.format(a=orig_a, b=orig_b, res=b))
            return b

        #Skubber variabler
        a = b
        b = r




# Kan finde største fælles divisor for tallene a og b
if __name__ == '__main__':
    try:
        a = int(sys.argv[1])
        b = int(sys.argv[2])

    except IndexError:
        print('Angiv venlist 2 heltal')
        print('Brug:')
        print('$ python euclid.py a b')
        sys.exit(1)

    except ValueError:
        print('a og b skal begge være heltal')
```

```
        sys.exit(1)

    if b > a:
        print('b skal være mindre end a')
        sys.exit(1)

    ### til tests; (324,148) = 4
    # a = 324
    # b = 148


    #Hvis alting virker ok, kør algoritmen
    euclid(a,b)
```

B.2 prime_factor.py

Filen ligger også vedhæftet i denne pdf, åbn den ved at dobbeltklikke på ikonet.

```
# Implementering af primtalsfaktorisering i Python
import sys

#Fundet på stackoverflow:
#https://stackoverflow.com/questions/15347174/python-finding-prime-factors#22808285

#Nedenstående algoritme virker på baggrund af primtalenes definition.
#Når et tal som går op i n findes, ved vi at det er et primtal,
#ellers ville nogle mindre tal også gå op.
def prime_factors(n):
    i = 2
    factors = []

    while i * i <= n: #Mens i er mindre end sqrt(n)

        if n % i: #Hvis i ikke går op i n
            i+=1 #Øg i med 1

        else: #Hvis i går op i n
            n //= i #Del n med i
            factors.append(i) #tilføj i som faktor

    if n > 1:
        factors.append(n) #Tilføj det sidste tal som

    return factors




if __name__ == '__main__':
    try:
        n = int(sys.argv[1])

    except IndexError:
        print('Angiv venlist et positivt heltal')
        print('Brug:')
```

```
        print('$ python prime_factor.py n')
        sys.exit(1)

    except ValueError:
        print('n skal være et positivt heltal')
        sys.exit(1)


    primes = prime_factors(n)

    print('Primtalsfaktorerne i {} er:'.format(n))
    print(primes)
```

B.3 phi_func.py

Filen ligger også vedhæftet i denne pdf, åbn den ved at dobbeltklikke på ikonet.

```
# Eulers phi-funktion i Python
import sys
from prime_factor import prime_factors

def phi(n): #Eulers phi-funktion
    #Giver p-1 for alle primfaktorerne
    return produkt([p-1 for p in prime_factors(n)]) #Og ganger dem så sammen


def produkt(liste): #Returnerer produktet af en liste
    n = 1
    for i in liste:
        n *= i
    return n



if __name__ == '__main__':
    try:
        n = int(sys.argv[1])

    except IndexError:
        print('Angiv venlist et positivt heltal')
        print('Brug:')
        print('$ python phi_func.py n')
        sys.exit(1)

    except ValueError:
        print('n skal være et positivt heltal')
        sys.exit(1)


    phi = phi(n)

    print('Antallet af tal som er indbyrdes primiske med {} er:'.format(n))
    print(phi)
```

B.4 eulers_sent.py

Filen ligger også vedhæftet i denne pdf, åbn den ved at dobbeltklikke på ikonet.

```python
# find inverse elementer af a mod n
# fundet her:
#
↪  https://stackoverflow.com/questions/4798654/modular-multiplicative-inverse-function-in-python#97
import sys

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        print('Der findes ikke et inverst element for de givne værdier')
        sys.exit(1)
    else:
        return x % m


# finder det inverse element af a mod n
if __name__ == '__main__':
    try:
        a = int(sys.argv[1])
        n = int(sys.argv[2])

    except IndexError:
        print('Angiv venlist 2 heltal')
        print('Brug:')
        print('$ python eulers_sent.py a n')
        sys.exit(1)

    except ValueError:
        print('a og n skal begge være heltal')
        sys.exit(1)


    #Hvis alting virker ok, kør algoritmen
    inv = modinv(a, n)
    print('Det inverse element af {} mod {} er: {}'.format(a, n, inv))
```