Digitale Signaturer i RSA
2. udgave

Jesper Møller Ken Larsen

Institut for Informationsteknologi

Januar 1997

Forord

Denne rapport er et informatik-fagpakkeprojekt skrevet ved Institut for Datateknik på Danmarks Tekniske Universitet. I rapporten viser vi, hvordan man i et public key kryptosystem kan arbejde med digitale signaturer.

Det forventes, at læseren har kendskab til matematik og datalogi i et omfang svarende til indholdet af informatik-fagpakkens kurser inden for disse områder: dvs. grundlæggende algebra (kursus 0116), sandsynlighedsregning (kursus 0142), funktionsprogrammering (kursus 4305), programmelkonstruktion (kursus 4306 og 4307), programmeringsteori (kursus 4312), og gerne objektorienteret programmering i C++ (kursus 43cc) og videregående algebra (kursus 0125).

2. udgave af rapporten er skrevet i LaTeX, og vi har valgt følgende typografi: vigtige begreber skrives med *kursiv*, kildetekst skrives med `courier`, program-moduler skrives med sans serif, og referencer skrives [x, y], hvor x er nummeret på bogen i referencelisten, og y er sidetallet.

Vi vil gerne sige tak til *Tom Høholdt* (Matematisk Institut), *Jens Thyge Kristensen* og *Anders P. Ravn* (Institut for Datateknik), der har været os behjælpelige under hele projektet.

Indhold

1 Indledning **3**
1.1 Oversigt over rapporten 3

2 Introduktion til kryptografi **5**
2.1 Kryptosystemer 5
2.2 Digitale signaturer 7
2.3 Problemstilling og afgrænsning 8

3 RSA kryptering **10**
3.1 Nøgler i RSA 10
3.2 RSA algoritmen 11
3.3 Digitale signaturer i RSA 12
3.4 Nøglegenerering i RSA 14
3.4.1 Gode nøgler i RSA 14
3.5 Primtalstest 15
3.5.1 Deterministiske test 15
3.5.2 Probabilistiske test 15
3.6 Tilfældige tal 17
3.7 Sikkerheden i RSA kryptosystemet 17
3.8 Angreb på RSA 18
3.9 Nøgleadministration 19

4 Specifikation og konstruktion **21**
4.1 Specifikation 21
4.2 Programstruktur 22
4.3 Konstruktion 22
4.4 Afprøvning 24
4.4.1 Num 24
4.4.2 Prime 25
4.4.3 Keyset 25
4.4.4 RSA 25

5 Brugervejledning **27**
5.1 Interface 27
5.2 Brug af `rsa` programmet 27
5.3 Fejlmeddelelser 31

6 Konklusion **32**
6.1 Diskussion 32
6.2 Konklusion 33
6.3 Perspektivering 33

A RSA algoritmen **37**

Kapitel 1

Indledning

Rapporten er en systematisk gennemgang og implementation af public key kryptosystemet *RSA*, som gør det muligt at arbejde med digitale signaturer. I rapporten har vi valgt en matematisk indfaldsvinkel og forsøgt at koncentrere os om det teoretisk essentielle samtidig med at berøre de fleste væsentlige aspekter. Ved den datatekniske konstruktion har vi som helhed lagt hovedvægten på valg af algoritmer og datatyper, fremfor at tilstræbe en optimering af de enkelte programdele. Motivationen for disse to valg er, at rapporten gerne skulle fremstå som en samlet oversigt over, hvilke komponenter der skal bruges i et RSA kryptosystem.

Motivationen for projektet er den voksende interesse for verificering og beskyttelse af data, der er opstået i kølvandet på informationssamfundets stigende behov for datakommunikation. Hemmeligholdelse, som opnås ved kryptering, er ikke længere det primære formål i et public key kryptosystem; *integritet*, *autenticitet*, og *uafviselighed* er blevet nøgleordene sammenknyttet i begrebet *digitale signaturer*.

For nogle ganske få årtier siden var det kun ved militære og diplomatiske anliggender, at man havde brug for at give meddelelser sådanne elektroniske underskrifter. I dag benyttes digitale signaturer i stadig højere grad, f.eks. som PIN-kode i kreditkort systemer, ved home-banking, ved mange former for telekommunikation (telefoni og satellit kommunikation) og i globale computer- net. Den teknologi, som førhen kun blev brugt af stormagternes regenter og generaler, vil efterhånden blive så udbredt, at også menigmand bliver berørt.

1.1 Oversigt over rapporten

Rapporten er komponeret i seks kapitler. Efter denne indledning gives i kapitel 2 dels en introduktion til kryptografi, dels gennemgås problemstillingen, og til sidst diskuterer vi den afgrænsning, der er foretaget undervejs ved problemløsningen. Hensigten med kapitel 3 er at give en grundig beskrivelse af RSA kryptosystemet, hvor vi har lagt vægten på det teoretiske. I kapitel 4 giver vi en specifikation af et RSA kryptosystem og beskriver overordnet konstruktionen og afprøvningen af det implementerede RSA program, mens kapitel 5 er en decideret brugervejled ning. Den resultatorienterede læser kan allerede nu gå direkte til kapitel 6, som dels er en diskussion af, hvor godt vi har løst problemet, og dels en samlet konklusion og perspektivering for hele rapporten. Appendix A-E indeholder

forskellige definitioner og beviser, og bilag 1-16 er programudskrifter. Bagest i rapporten findes en annoteret referenceliste.

Kapitel 2

Introduktion til kryptografi

Dette kapitel indeholder dels en introduktion til emnet kryptografi, og dels en problemstilling for projektet. Først gives nogle definitioner, som er vigtige for forståelsen, og som bruges igennem resten af rapporten. Så beskrives forskellige kryptosystemer med hovedvægten lagt på public key kryptosystemer, og vi motiverer anvendelsen af digitale signaturer ved transmission af meddelelser. Dernæst diskuteres projektets formål, problemstilling og afgrænsning, og til sidst peger vi på nogle de perspektiver, projektet har.

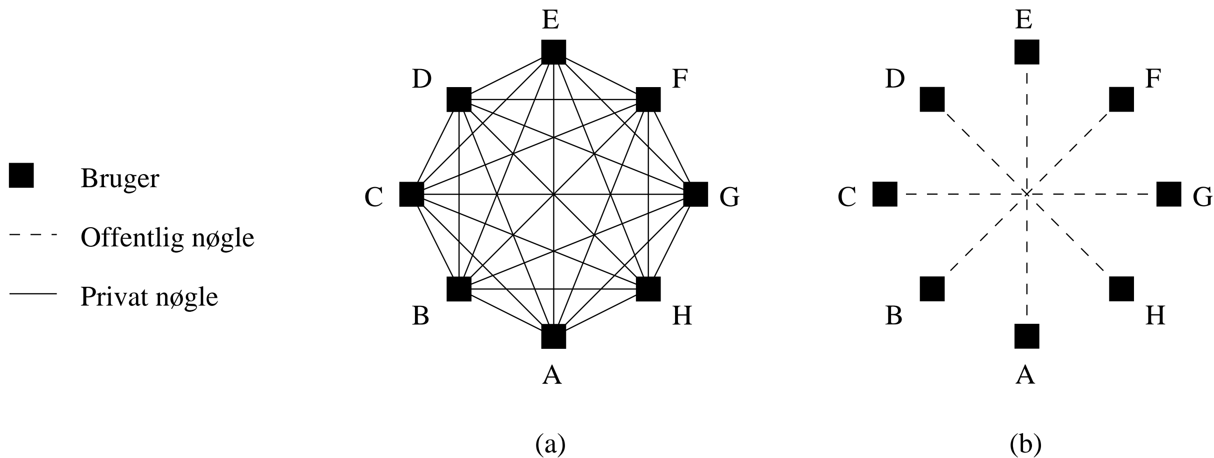
2.1 Kryptosystemer

Ordet krypto er græsk og betyder hemmelig, så *kryptografi* betyder *hemmelig skrift*. Et kryptogram er en hemmelig tekst, dvs. en tekst, der er blevet omformet, så den ikke længere er forståelig. Man kan ved hjælp af en *krypteringsfunktion* kryptografere eller blot *kryptere* en tekst. For at gøre teksten forståelig eller meningsfuld igen skal man ved hjælp af en *dekrypteringsfunktion* (den inverse til krypteringsfunktionen) tilsvarende dekryptografere eller blot *dekryptere* den krypterede tekst.

Ofte bruger man betegnelserne *ciffertekst* for krypteret tekst og *klartekst* for almindelig tekst, som ikke er krypteret. Kaldes cifferteksten C, klarteksten eller meddelelsen M, og krypterings funktionen f, skal der altså gælde: $f(M) = C$, og $f^{-1}(C) = M$.

Hvis to eller flere *brugere* ønsker at sende *meddelelser* (dvs. information af en eller anden art) til hinanden og ikke ønsker, at meddelelserne undervejs kan læses af andre, kan de benytte ovenstående teknik: Som det ses på figur 2.1 krypterer A først klarteksten, så transmitteres cifferteksten via en *kanal* til modtageren B, og ved modtagelsen dekrypteres cifferteksten til den oprindelige meddelelse. Man har således etableret et *kryptosystem*.

Figur 2.1: Transmission af en meddelelse i et kryptosystem med to brugere: A krypterer klarteksten M og sender den fremkomne ciffertekst C via transmissionskanalen. B modtager cifferteksten C og dekrypterer denne, hvorved klarteksten M atter fremkommer.



Figur 2.2: To forskellige kryptosystemer: (a) et *private key kryptosystem*, (b) et *public key kryptosystem*.

Som regel vil formen af krypterings- og dekrypteringsfunktionen være kendt af alle, mens nogle af de indgående konstanter holdes hemmelige. Man bruger metaforen en *nøgle* for disse konstanter, og hvis den samme nøgle bruges både til kryptering og dekryptering, taler man om et *private key kryptosystem*, mens et *public key kryptosystem* har to separate nøgler en til hver operation.

Et private key kryptosystem er desuden karakteriseret ved, at hver bruger har en nøgle for hver af de andre brugere af kryptosystemet. I et public key kryptosystem derimod har hver bruger kun to nøgler, nemlig en offentlig og en privat nøgle.

Eksempel 2.1 Lad krypteringsfunktionen være $f(x) = kx + n$. Så er dekrypteringsfunktionen $f^{-1}(x) = \frac{x-n}{k}$. Her er k og n nøglerne. ∎

Eksempel 2.2 Vi forestiller os et private key baseret *postsystem* med 8 brugere (se figur 2.2). Brugeren A skal have en postkasse sammen med hver af de andre brugere i systemet $(B, C, \ldots, H)$. Disse indbyrdes fælles postkasser, som i (a) er symboliseret ved streger, er brugernes private nøgler. I et public key baseret postsystem derimod er det kun nødvendigt med én postkasse pr. bruger. Disse postkasser, som i (b) er symboliseret ved punkterede streger, er brugernes offentlige nøgler, mens de private nøgler hemmeligholdes af hver enkelt bruger. ∎

Der findes mange eksempler på private key kryptosystemer. De simpleste benytter fotrskellige former for substitution og blev blandt andet brugt af Julius Cæsar. Det nok mest udbredte private key kryptosystem er *DES* (Data Encryption Standard), som blev udviklet af IBM og adopteret af NBS (National Bureau of Standards) i 1977. DES bruges i dag især ved password kontrol, og der er endda lavet VLSI chips, som udfører DES algoritmen på hardware-niveau.

Som det fremgår af eksempel 2.2, indeholder et private key system med n brugere langt flere nøgler end et tilsvarende public key system. I et private key system er antallet

Integritet:	Indholdet af meddelelsen må under transmissionen fra afsender til modtager ikke være ændret af tekniske uheld eller af ondsindede personer.
Autenticitet:	Modtager har sikkerhed for, at meddelelsen virkelig kommer fra afsender.
Uafviselighed:	Afsender kan på et senere tidspunkt ikke afvise at have sendt meddelelsen til modtager.

Figur 2.3: Sikkerhedskriterier ved datatransmission.

af nøgler

$$k_{\text{private}} = \binom{n}{2} = \frac{n(n-1)}{2}, \tag{2.1}$$

nemlig $n - 1$ for hver bruger, hvoraf halvdelen er ens. I et public key kryptosystem er antallet af nøgler

$$k_{\text{public}} = 2n, \tag{2.2}$$

fordi hver bruger har to nøgler.

I et private key kryptosystem med n brugere skal der altså laves n nøgler, hver gang en ny bruger tilmeldes. Problemerne med dels at generere disse mange nøgler og dels for brugerne at holde styr på dem er nogle af de faktorer, som har tilskyndet udviklingen af public key kryptosystemer som f.eks. El-Gamal og *RSA*, hvor sidstnævnte blev udviklet af R. Rivest, A. Shamir og L. Adleman i 1978. I disse systemer skal der kun laves to nøgler, når en ny bruger tilmeldes, og hver bruger har også kun to nøgler. Til gengæld opstår der så et nyt problem, nemlig nøgleadministration: Brugernes offentlige nøgler skal være tilgængelige i en database (dvs. en slags telefonbog), som f.eks. kan administreres af en central instans. Dette problem diskuteres kort i afsnit 3.9.

Selvom der i begge typer af kryptosystemer er visse problemer med at administrere og generere nøglerne, er det alligevel public key kryptosystemerne, der har opnået den største udbredelse. Årsagen hertil er dels, at public key kryptosystemer er nemmere at udvide med flere brugere, og dels at man let kan tilføje teknikker til at lave *digitale signaturer* i meddelelserne. I næste afsnit beskriver vi digitale signaturer, og i afsnit 3.3 viser vi, hvordan man bruger dem i RSA.

2.2 Digitale signaturer

I det foregående afsnit har vi koncentreret os om kryptering af meddelelser i kryptosystemer. Denne hemmeligholdelse er selvfølgelig vigtig i mange sammenhænge, men oftest er man mere interesseret i, at en række andre kriterier vedrørende datasikkerhed og verifikation er opfyldt:

Det første kriterium vedrører datasikkerhed, mens de sidste to kriterier bruges ved verificering. Følgende eksempel forklarer betydningerne.

Eksempel 2.3 Vi betragter igen postsystemet fra eksempel 2.2. Hvis man forsyner sine meddelelser med et unikt *segl*, så er de tre sikkerhedskriterier opfyldt. For det første giver det modtageren sikkerhed for, at indholdet af en meddelelse ikke er blevet læst eller ændret af andre undervejs (for så ville seglet være brudt); modtageren har desuden sikkerhed for, at meddelelsen kommer fra afsender (for ingen andre har et segl med samme motiv); og endelig kan afsender ikke på et senere tidspunkt afvise at have sendt meddelelsen (for den bærer hans segl). ■

Da man er holdt op med at bruge segl, og da en stadig større del af kommunikationen i vor tids informationssamfund foregår via datanet, har man brug for at kunne lave en *digital signatur* (dvs. en elektronisk underskrift), som kan leve op til de tre sikkerhedskriterier i figur 2.3. Følgende simple eksempel illustrerer vigtigheden af digitale signaturer.

Eksempel 2.4 Vi har to firmaer A og B, der geografisk er placeret langt fra hinanden, men kan kommunikere via et stort, offentligt datanet. Nu ønsker A via datanettet at afgive en ordre til B, som derefter også via datanettet skal bekræfte ordren. Der opstår følgende problemstilling:

- Hvordan kan B være sikker på, at ordren kommer fra A?
- Hvordan kan B sikres imod, at A benægter at have sendt ordren?
- Hvordan kan A være sikker på, at bekræftelsen kommer fra B?
- Hvordan kan A sikres imod, at B benægter at have sendt bekræftelsen?
- Hvordan kan A og B sikres imod, at ordren og bekræftelsen ændres under transmissionen?

Ovenstående problemstilling kan løses ved, at A påfører ordren sin digitale signatur, og at B påfører bekræftelsen sin digitale signatur. Dette kan sammenlignes med, at A og B underskriver en fælles kontrakt. Nu er det kun A, der har kunnet sende den pågældende ordre, så B er fuldt ud sikret. Tilsvarende er A også sikret, idet han har B's digitale signatur på bekræftelsen. ■

Kravet om integritet er meget svært at opfylde i praksis, idet det ville kræve, at transmissionskanalerne var overvågede og beskyttede. Derfor benytter man ofte fejlkorrigerende koder (se f.eks. [7, 115] til at identificere og eventuelt rette tekniske fejl, mens man benytter digitale signaturer til at afgøre, om integritetskriteriet er opfyldt eller ej. Det leder naturligt frem til følgende problemstilling.

2.3 Problemstilling og afgrænsning

Formålet med dette projekt er *at forstå og formidle den matematiske teori bag RSA kryptosystemer*, og problemstillingen er *at implementere en prototype af et RSA kryptosystem*, hvor man kan lave digitale signaturer. Rapporten, som således gerne skulle munde ud i *en samlet oversigt over, hvordan man laver et RSA kryptosystem i praksis*, kan eventuelt ses som et indledende forarbejde til et større, kommercielt RSA kryptosystem.

Afgrænsningen er foretaget på to niveauer: Dels er det teoretiske stof blevet afgrænset, og dels er implementeringen af RSA kryptosystemet blevet afgrænset. Førstnævnte bliver beskrevet i det følgende, mens afgrænsningen af programmet præsenteres i afsnit 4. Valget af public key kryptosystemet er afgrænset til kun at omhandle et RSA system, og vi vil specielt betragte digitale signaturer i dette system. Vi vil ikke gå i detaljer med angreb og brydning af RSA kryptosystemer, ligesom vi heller ikke vil koncentrere os om administration af nøgler. Vi vil desuden undlade at anvende checksums- eller hashfunktioner ved digitale signaturer i RSA systemet, og ved generering af tilfældige tal vil vi benytte en metode, som bruger en såkaldt blandet multiplikativ generator. Som værktøjer til at realisere løsningen af de programmeringsmæssige opgaver har vi valgt SML og C++. Med SML vil vi først lave en køreklar prototype, som ikke er optimeret på nogen måde. Denne prototype bliver så senere implementeret i C++, hvor vi opprioriterer valg af datatyper og algoritmer.

Motivationen for valget af RSA er dels, at det er det mest udbredte public key kryptosysem i praksis, dels at det endnu ikke er lykkes for nogen at komme med en effektiv måde at bryde det på, og dels at den matematiske teori bag et RSA kryptosystem er enkel og meget smuk. Når man bruger RSA-algoritmen, har man desuden den store fordel, at det er forholdsvist enkelt at lave digitale signaturer.

Som situationen tegner sig nu i vor tids informationssamfund, vil behovet for at kunne anvende digitale signaturer være stadigt stigende. Dette er en konsekvens af den øgede datakommunikation via offentlige datanet, hvor områder som *home-shopping* og *home-banking* vinder større og større udbredelse. Eksempler herpå er Frankrigs Minitel, Danmarks Diatel, Lån & Spar Banks PC-Bank og det verdensomspændende Internet. Digitale signaturer skal her garantere, at de tre kriterier i figur 2.3 vedrørende sikkerhed og verifikation er opfyldt, således at de økonomiske forhold ved handel og banktransaktioner er sikre og juridisk bindende.

I det følgende kapitel beskrives dels det teoretiske grundlag for RSA-kryptering, og dels gennemgås nogle praktiske forhold i RSA kryptosystemer.

Kapitel 3

RSA kryptering

I dette kapitel gennemgås to hovedemner. I første del beskriver vi den matematiske teori bag RSA kryptering: Vi gennemgår grundigt, hvordan RSA algoritmen virker, og vi viser, hvordan man kan lave digitale signaturer i et RSA kryptosystem. Herefter beskriver vi, hvordan man laver et nøglesæt til en bruger i et RSA kryptosystem. Herved kommer vi ind på generering af tilfældige tal, og vi beskriver forskellige metoder til at undersøge, om et tal er et primtal.

I kapitlets anden del analyserer vi nogle af sikkerhedsaspekterne i RSA kryptosystemer: Vi berører emnet faktorisering af heltal, vi undersøger forskellige angreb på RSA kryptosystemer, og til sidst diskuterer vi nogle forhold vedrørende administration af nøgler i et RSA kryptosystem.

3.1 Nøgler i RSA

Som beskrevet i indledningen er RSA et public key kryptosystem, hvor hver bruger har et nøglesæt, der består af en offentlig og en privat (hemmelig) nøgle. Et nøglesæt i et RSA kryptosystem består af i alt 3 positive heltal, og der er tradition for at kalde disse 3 heltal e, d og n. Den private nøgle udgøres af (d, n) og bruges til at dekryptere (eng. decrypt) de meddelelser, man modtager. Den offentlige nøgle, som udgøres af (e, n), bruges af andre brugere, der vil kryptere (eng. encrypt) meddelelser til en.

Vi vil med figur 3.1 angive, hvordan e, d og n bestemmes teoretisk [15, 77], mens vi i afsnit 3.4 indgående beskriver, hvordan man finder disse tal.

1. Vælg to primtal p og q.
2. Beregn $n = pq$.
3. Bestem e, således at: $\gcd(e, \phi(n)) = 1$.
4. Bestem d, således at: $ed \bmod \phi(n) = 1$.

$\phi(n)$ er Eulers totient funktion (se definitionen A.1).

Efter at have beregnet e, d og n offentliggøres e og n, mens d, p og q hemmeligholdes. Man har således fået genereret krypteringsnøglen (e, n) og dekrypteringsnøglen (d, n).

3.2 RSA algoritmen

Som vi beskrev i kapitel 2, er der til ethvert kryptosystem tilknyttet en krypteringsfunktion, som angiver, hvordan en klartekst skal krypteres, og en dekrypteringsfunktion, som angiver, hvordan man udfra en ciffertekst kan beregne den tilhørende klartekst. Vi vil nu præsentere RSA algoritmen, som beskriver, hvordan man krypterer og dekrypterer i et RSA kryptosystem [15, 78].

RSA algoritmen: Lad M være en klartekst (meddelelse), som er repræsenteret ved et tal. Da siger RSA algoritmen, at hvis cifferteksten C beregnes som:

$$C = M^e \bmod n \tag{3.1}$$

så vil klarteksen kunne beregnes som:

$$M = C^d \bmod n \tag{3.2}$$

RSA algoritmen kan også formuleres som:

Sætning 1 (RSA algoritmen) *Lad e, d og n være heltal, som opfylder kravene i figur 3.1, og lad endvidere $M \in [0; n-1]$. Da gælder:*

$$(M^e \bmod n)^d \bmod n = M \tag{3.3}$$

BEVIS. Vi beviser kun sætningen, når $\gcd(M, n) = 1$. I appendix A er sætningen bevist generelt. Idet

$$ed \bmod \phi(n) = 1 \qquad \leftrightarrow \qquad ed = \phi(n)t + 1 \tag{3.4}$$

hvor t er et heltal, fås

$$\begin{aligned}
(M^e \bmod n)^d \bmod n &= M^e d \bmod n \\
&= M^{\phi(n)t+1} \bmod n \\
&= MM^{\phi(n)t} \bmod n \\
&= (M \bmod n)(M^{\phi(n)t} \bmod n) \bmod n \\
&= (M \bmod n)(M^{\phi(n)} \bmod n)^t \bmod n \\
&= (M \bmod n)1^t \bmod n \qquad (*) \\
&= M \bmod n \\
&= M
\end{aligned} \tag{3.5}$$

Here har vi i (*) brugt sætning A.3. □

Eksempel 3.1 Vi forestiller os et RSA kryptosystem med to brugere A og B, som har hver sit nøglesæt. A's offentlige nøgle er (n_A, e_A), mens den private nøgle er (n_A, d_A). Tilsvarende har B nøglesættet (n_B, e_B) og (n_B, d_B). A ønsker nu at sende meddelelsen $M \in [0; n-1]$ til B. Først beregner A cifferteksten C:

$$M^{e_B} \bmod n_B = C \tag{3.6}$$

Så sendes cifferteksten C til B, som for at dekryptere cifferteksten beregner:

$$C^{d_B} \bmod n_B = M \tag{3.7}$$

Som det ses, kan B nu læse meddelelsen fra A. ■

Som det ses af sætning 3.1, er det et krav, at $M \in [0; n-1]$. I praksis sikrer man dette ved, at man først deler sin meddelelse op i blokke af M_i'er, hvor der gælder $\forall i. M_i \in [0; n-1]$, og herefter krypterer M_i'erne. RSA kryptosystemet kaldes derfor et blok-krypterings system [10, 47].

3.3 Digitale signaturer i RSA

Som det blev fremhævet afsnit 2.2, er hemmeligholdelse ikke altid det væsentlige ved data- kommunikation. Ofte er man mere interesseret i, at visse *datasikkerhedskriterier* (integritet, autenticitet og uafviselighed) er opfyldt. Dette kan gøres ved at forsyne sine meddelelser med en digital signatur, og i et RSA kryptosystem har man flere muligheder for at gøre dette, uden at skulle udvide nøglesættet. Vi vil nu med to formelle eksempler vise, hvordan man kan lave en digital signatur i et RSA kryptosystem [15, 79].

Eksempel 3.2 Som i eksempel 3.1 vil brugeren A sende meddelelsen $M \in [0; n-1]$ til brugeren B, men denne gang således at B kan verificere, at meddelelsen kun kan være sendt af A, mens at hemmeligholdelsen af M er underordnet. A's offentlige og private nøgle er (e_A, n_A) henholdsvis (d_A, n_A), mens B's nøglesæt er (e_B, n_B) og (d_B, n_B).

Først beregner A:

$$M^{d_A} \bmod n_A = M_1 \tag{3.8}$$

Derefter sender A parret (M, M_1) til B, som sammenligner:

$$M_1^{e_A} \bmod n_A = M \tag{3.9}$$

Som det ses er det kun A, der har kunnet beregne M_1, så udsagnet er sandt. ■

Følgende eksempel illustrerer, hvordan man desuden sikrer, at meddelelsen holdes hemmelig.

Eksempel 3.3 Vi har igen brugerne A og B, med samme nøglesæt som i eksempel 3.1 og 3.2. A vil sende meddelelsen M til B på samme måde som i eksempel 3.2, men nu skal meddelelsen også hemmeligholdes. Derfor beregner A først:

$$M^{d_A} \bmod n_A = M_1 \tag{3.10}$$

og dernæst:

$$M_1^{e_B} \bmod n_B = C_1 \land M^{e_B} \bmod n_B = C \tag{3.11}$$

A sender nu parret (C_1, C) til B, som efter at have modtaget (C_1, C) beregner:

$$C_1^{d_B} \bmod n_B = M_1 \land C^{d_B} \bmod n_B = M \tag{3.12}$$

For at få verificeret, at beskeden kommer fra A, sammenligner B:

$$M_1^{e_A} \bmod n_A = M \tag{3.13}$$

Som det ses, er det som i eksempel 3.2 kun A, der kan have beregnet M_1, så udsagnet er sandt. ■

I eksempel 3.3 vil den opmærksomme læser have bemærket, at hvis $n_A > n_B$, vil man kunne risikere, at $M_1 \notin [0; n_B - 1]$, og derved vil man ikke kunne dekryptere C_1. Dette problem kan løses ved, at man (når $n_A > n_B$) istedet bruger følgende 4 trin:

$$M^{e_B} \bmod n_B = C \tag{3.14}$$

$$C^{d_A} \bmod n_A = C_1 \tag{3.15}$$

$$C^{d_B} \bmod n_B = M \wedge C_1^{e_A} \bmod n_A = M_1 \tag{3.16}$$

$$M_1^{d_B} \bmod n_B = M \tag{3.17}$$

I forlængelse af eksempel 3.2 og 3.3 er det er meget vigtigt at bemærke, at B med sikkerhed ved, at M kun kan komme fra A. Det skyldes, at A er den eneste, der kender d_A og derfor er den eneste, som kan beregne M^{d_A}. Man kan let bevise, at RSA signatur algoritmen er korrekt ved at anvende sætning 3.1 to gange.

Af eksemplerne ses desuden, at man i begge tilfælde sender både M og M_1 (krypteret eller ikke krypteret). I praksis er dette heller ikke nødvendigt, da man kun vil få en forståelig klartekst ud, hvis M_1 "dekrypteres"med (e_A, n_A). Dermed kan man på entydig vis se, at meddelelsen kommer fra A.

Denne afsnit afsluttes med et lille taleksempel, som viser, hvordan man manuelt krypterer en klartekst og dekrypterer en ciffertekst.

Eksempel 3.4 Som i eksempel 3.1 har vi et RSA kryptosystem med to brugere A og B. A vil gerne sende meddelelsen $M = 12$ til B, der ud fra primtallene $p_B = 5$ og $q_B = 7$ har beregnet nøglesættet $(e_B = 11, n_B = 35)$ og $(d_B = 11, n_B = 35)$. Først beregner A cifferteksten $C = M^{e_B} \bmod n_B$:

$$\begin{aligned} C &= 12^{11} \bmod 35 \\ &= 12 \times 12^{10} \bmod 35 \\ &= 12 \times (12^2)^5 \bmod 35 \\ &= 12 \times (144 \bmod 35)^5 \bmod 35 \\ &= 12 \times 4^5 \bmod 35 \\ &= (48 \bmod 35)16^2 \bmod 35 \\ &= 13 \times (256 \bmod 35) \bmod 35 \\ &= 13 \times 11 \bmod 35 \\ &= 3 \end{aligned} \tag{3.18}$$

Så sendes cifferteksten $C = 3$ til B, som dekrypterer den modtagne tekst ved at beregne $M = C^{d_B} \bmod n_B$:

$$M = 3^{11} \bmod 35 = 12 \tag{3.19}$$

Som det ses, er den dekrypterede meddelelse lig med den oprindelige. ■

3.4 Nøglegenerering i RSA

Vi vil nu se nærmere på, hvordan man bestemmer de krypterings- og dekrypteringsnøgler, som blev defineret i afsnit 3.1. Selvom en bruger faktisk godt kan sende ciffertekster til de øvrige brugere uden selv at have et nøglesæt (som A i eksempel 3.4), er han afskåret fra at benytte digitale signaturer og modtage krypterede meddelelser. Hver bruger skal derfor have lavet et nøglesæt, som opfylder kravene fra figur 3.1.

Det viser sig, at det største problem ved bestemmelse af nøglerne, er at finde de to primtal p og q. Som det fremgår af afsnit 3.7, gælder det nemlig om at finde to *store* primtal af størrelsesorden 100 cifre. Generering af primtal er et omfattende emne, som har interesseret matematikere i flere tusind år. Problemet er, at der endnu ikke er fundet en lukket funktionsforskrift for, hvordan man bestemmer primtal. I determinismens navn ønsker man derfor i det mindste at kunne afgøre, om et givet tal er et primtal eller ej. I afsnit 3.5 gennemgår vi forskellige metoder, som kan afgøre dette.

Tilbage står så at bestemme e og d ud fra de to sidste krav i figur 3.1. Tallet d kan bestemmes ved hjælp af Euklids udvidede algoritme (se appendix B), mens det viser sig, at den letteste måde at bestemme e på, er ved at prøve sig frem med indsættelse.

3.4.1 Gode nøgler i RSA

I afsnit 3.7 viser vi, at det gælder om at vælge $pogq$ således, at det er svært at faktorisere produktet $n = pq$. I [5, 106] refereres til R.Rivest, A.Shamir og L.Adleman, som har foreslået, at der skal tages følgende hensyn til primtallene p og q fra figur 3.1:

1. Antallet af cifre i $p \neq$ antallet af cifre i q (bit-mæssigt).
2. Såvel $p - 1$ som $q - 1$ skal have store primfaktorer.
3. $\gcd(p - 1, q - 1)$ skal være lille.

Betingelsen (1) sikrer, at n ikke kan faktoriseres ved almindelig rodtagning. Betingelsen (2) forhindrer en hurtig faktorisering baseret på den såkaldte Pollards $(p-1)$-metode [13, 172]. For at finde et primtal p, hvorom det gælder, at $p-1$ har en stor primfaktor, kan der gås frem på følgende måde: Først findes et primtal r ved hjælp af en de primtalstest, der er beskrevet i næste afsnit. Herefter beregnes $p = 2ir + 1$, hvor $i = 1, 2, 3, \ldots$ indtil p er et primtal, dvs. passerer en primtalstest. Det bemærkes, at man kan opnå en endnu større sikkerhed, hvis man desuden vælger r, så også $r - 1$ har en stor primfaktor. Et primtal p, hvor både $p - 1$ og $r - 1$ har store primfaktorer, kaldes et *stærkt primtal* [2, 53].

Yderligere krav til heltallene e og d fra figur 3.1 er anført nedenfor:

1. $e \in [\max(p, q) + 1; \phi(n) - 1]$.
2. $d > n^?$

Betingelse (1) forhindrer angreb baseret på lav eksponent (se afsnit 3.8), mens betingelse (2) er nødvendig, idet det er bevist, at når $d < n^?$, da kan d let blive bestemt, og så kan n faktoriseres [14, 519].

3.5 Primtalstest

En primtalstest er en algoritme, som givet en *kandidat* (dvs. et tal, man tror er et primtal), fastslår om kandidaten er et primtal eller et sammensat tal med en vis sandsynlighed for at have taget fejl. Afgørelsen fås ved enten at påtrykke kandidaten en deterministisk eller en probabilistisk test. Hvis kandidaten passerer en deterministisk test, er den med garanti et primtal, men sådanne test har meget lange svartider. De probabilistiske test derimod har relativt korte svartider, men til gengæld er der en vis risiko for, at de fejlagigt accepterer et sammensat tal som et primtal; det der i statistikken kaldes fejl af type 2. Som vi skal se, kan sandsynligheden for sådanne fejl negligeres, når blot der gennemføres flere uafhængige test.

Vi vil i det følgende kort gennemgå en deterministisk test, hvorefter vi vil koncentrere os om probabilistiske test, da de i dag er de mest udbredte i forbindelse med RSA-kryptering [14, 30]. Appendix D indeholder en udførlig beskrivelse af de beskrevne primtalstest.

3.5.1 Deterministiske test

En meget velkendt deterministisk test er *Erastosthenes si* [13, 3], der finder alle pimtallene under et givet tal ved at si alle de sammensatte tal fra: Givet et interval $[m, n]$, da findes først alle primtallene $p_i \leq \sqrt{n}$, og herefter identificeres og slettes alle de tal, der er multipla af blot et p_i.

Eksempel 3.5 Vi vil finde alle primtallene under 100. Først findes primtallene under 10, som er $\{2, 3, 5, 7\}$. Herefter betragtes talsekvensen af tallene fra 1 til 100, hvor vi først sletter multipla af 2 (dvs. de lige tal), derefter multipla af 3 og 5. Herved fremkommer følgende sekvens:

$$1, 7, 11, 13, \ldots, 89, 91, 97 \tag{3.20}$$

Til sidst slettes multipla af 7, og resultatet er

$$1, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97 \tag{3.21}$$

der sammen med {2,3,5,7} netop er primtallene under 100 (bortset fra 1). ■

Erastosthenes si kan ikke bruges, hvis der skal findes primtal af størrelsesordnen 10^{200}, da det vil kræve, at man først finder primtallene under 10^{100} og derefter dividerer kandidaterne i intervallet $[10^{100}; 10^{200}]$ med hvert af disse mange primtal. Andre deterministiske test er beskrevet i [1] og [3] og de bruger i størrelsesordnen $T(n) = (\ln n)^{c \ln(\ln(\ln n))}$ karakteristiske operationer, hvor n er antallet af cifre i tallet og $c > 1$ er en konstant. De tager derfor uforholdsmæssig lang tid at udføre og bruges følgelig ikke i praksis til at finde store primtal [14, 268]. Derfor må man benytte sig af andre metoder: de probabilistiske test.

3.5.2 Probabilistiske test

De probabilistiske test har deres ret, idet Tchebycheff har vist, at andelen af primtal under et givet tal m, approksimativt er $\pi(m) = (\ln m)^{-1}$ [14, 29]. Sættes $m = 10^{100}$ haves, at

ca. 1 ud af 230 tal er primtal, og betragtes kun de ulige tal (alle lige tal på nær 2 er jo sammensatte, og derfor ikke kandidater til primtal), er det 1 ud af 115 tal. Der er altså en god chance for at et tilfældigt valgt ulige tal er et primtal. Udgangspunktet for de probabilistiske primtalstest er negationen af Fermats lille sætning:

Sætning 2 (Fermats lille sætning negeret) *Hvis der eksisterer et heltal a, hvor* $0 < a < c$*, således at*

$$a^{c-1} \bmod c \neq 1 \tag{3.22}$$

da er c ikke et primtal.

Bevis. Se sætning A.2 i appendix A. □

Fermats test bruger sætning 3.2 til at afgøre om et tal er sammensat eller et primtal. Hvis c er en kandidat til at være et primtal, og $a^{c-1} \bmod c$ giver et resultat, der er forskelligt fra 1 for blot et $a \in [1; c-1]$, da er c ikke er et primtal. I modsat fald siges c at være et *pseudo primtal.*

Det kan vises [14, 30], at hvis tallet c ikke er et primtal, da kan det kun være pseudoprimtal for (meget) mindre end halvdelen af tallene $a_i \in [1; c-1]$. Det vil sige, at sandsynligheden for, at c passerer 1 test uden at være et primtal, er (meget) mindre end $\frac{1}{2}$. Gennemføres 100 uafhængige test, bliver sandsynligheden for fejlagtigt at acceptere et sammensat tal som et primtal:

$$P_{Fe} \leq \left(\frac{1}{2}\right)^{100} = 7.8 \times 10^{-31} \tag{3.23}$$

Uheldigvis findes der sammensatte heltal, de såkaldte *Carmichael tal*, der uden at være primtal passerer ovenstående test for alle $a \in [1; c-1]$. Heldigvis er disse Carmichael tal meget sjældne (der er kun 2163 Carmichael tal mindre end 25×10^9 [7, 113], og mange af dem bliver afsløret ved i stedet for negationen af Fermats lille sætning at bruge negationen af Eulers kriterie:

Sætning 3 (Eulers kriterie negeret) *Hvis der eksisterer et heltal a, hvor* $0 < a < c$*, således at*

$$a^{\frac{c-1}{2}} \bmod c \neq \pm 1 \tag{3.24}$$

da er c ikke et primtal.

Bevis. Se [13, 280]. □

Lehmann-Peraltas test benytter samme princip som Fermats test, men bruger istedet sætning 3.3, som et bedre kriterie for hvornår kandidaten er sammensat. Solovay-Strassen har videreudviklet Lehmann-Peraltas test, idet man i *Solovay-Strassens test* desuden sammenligner resultatet af kongruensrelationen med Jacobi-symbolet (se appendix C). Begge test benytter sig af *vidner*, der er heltal i intervallet $[1; c-1]$, for at afgøre om et heltal c er sammensat, eller et primtal. Angiver ingen af vidnerne, at c er sammensat, da antages det, at c er et primtal. Sandsynligheden for at et vidne er løgnagtigt, (dvs. angiver, at et

sammensat tal er et primtal), er i begge tilfælde højst ??; dvs. at sandsynligheden for at begå fejl ved 1 test er [14, 270]:

$$P_{LP} \leq ?? \tag{3.25}$$

$$P_{SS} \leq ?? \tag{3.26}$$

Rabins test er principielt opbygget som de 3 øvrige, men har en mere raffineret vidnedefinition, der sikrer, at højst ?? af tallene i intervallet $[1; c-1]$ er løgnagtige vidner, når blot $c > 4$ [12]. Dermed bliver sandsynligheden for at begå fejl ved 1 test:

$$P_{Ra} \leq ?? = (??)^2 \tag{3.27}$$

og følgelig behøver Rabins test kun halvt så mange vidner, som Lehmann-Peraltas test og Solovay-Strassens test for at udtale sig med samme pålidelighed.

I praksis er Solovay-Strassens test og Rabins test de mest brugte, men de er også de mest krævende beregningsmæssigt. Derfor vil det være oplagt først at undersøge, om en kandidat har små primfaktorer ved at afprøve primtallene under 1000, før man kører en af de to test.

3.6 Tilfældige tal

Af ovenstående gennemgang følger det, at generering af tilfældige tal spiller en væsentlig rolle ved RSA-kryptering. De tal, som er kandidater til at være primtal, skal på en eller anden måde genereres tilfældigt. Da det er svært at danne virkeligt tilfældige tal, som f.eks. tidsafstanden mellem klik fra en Geiger-Müller-tæller tæt ved en radioaktiv kilde, har vi valgt at koncentrere os om en meget udbredt metode, nemlig den *Blandede Multiplikative Generator* (BMG, eng: The Mixed Congruential Method). Denne metode benytter en differensligning af følgende simple form:

$$x_n + 1 = f(x_n) \tag{3.28}$$

hvor *frøet* (eng: seed) x_0, er givet. BMG danner ud fra funktionen f en sekvens af pseudo tilfældige tal x_i, $i = 1, 2, 3, \ldots$, således at $0 < x_i < m$ for et givet m. Sekvensen er pseudo tilfældig og ikke "rigtig tilfældig", fordi den kan forudsiges ud fra kendskab til $f(x_n)$ og x_0. Se også appendix E.

I anden del af dette kapitel vil vi i de to følgende afsnit belyse nogle forhold vedrørende sikkerhed i RSA systemet, og i afsnit 3.9 skitsere hvilke problemer der er forbundet med nøgleadministration.

3.7 Sikkerheden i RSA kryptosystemet

Sikkerheden i et RSA kryptosystem bygger først og fremmest på, at der ikke findes algoritmer, som i overskuelig tid kan faktorisere store heltal (på f.eks. 200-300 cifre). Godt nok er det endnu ikke bevist, at et RSA kryptosystem er lige så sikkert, som det er svært at faktorisere n, men man har ikke kendskab til angreb, hvor et RSA kryptosystem med gode nøgler (se afsnit 3.4.1), er blevet brudt hurtigere, end man kan faktorisere n.

Antal cifre i n	$T(n)$	Beregningstid
25	4.3×10^6	0.43 millisek.
50	1.4×10^{10}	1.4 sek.
100	2.3×10^{15}	2.7 dage
129	7.3×10^{17}	839 dage
200	1.2×10^{23}	3.8×10^5 år

Figur 3.1: Beregningstider for faktorisering af store heltal.

Det må derfor konkluderes, at så længe det i praksis ikke muligt i overskuelig tid at faktorisere et n på 200-300 cifre i dets primfaktorer, da er det heller ikke muligt at bryde RSA systemet. Hvis man ikke kan faktorisere n, kan man ikke beregne $\phi(n)$, som skal bruges til at bestemme d (se dog næste afsnit, hvor nogle specielle situationer behandles).

Som vi gennemgik i afsnit 3.4.1, er valget af primtallene p og q af stor betydning, og for at opnå en sikker kryptering foreslog R.Riverst, A.Shamir og L.Adleman i sin tid at vælge p og q i størrelsesordenen 100 cifre. De hurtigste generelt brugbare faktoriseringsalgoritmer, inklusive kvadratisk si algoritmen [14, 303], benytter alle i størrelsesordnen

$$T(n) = e^{\sqrt{\ln n \ln(\ln n)}} \tag{3.29}$$

karakteristiske operationer, hvor n er antallet af cifre i tallet, der skal faktoriseres. Figur 3.4 viser beregningstiden for forskellige værdier af n, idet det er forudsat, at en computer kan udføre i størrelsesordenen 10^{10} karakteristiske operationer pr. sekund [14, 207].

Med mindre der kommer et revolutionerende gennembrud indenfor generelt brugbare faktoriseringsalgoritmer, må RSA kryptosystemet anses for at være ubrydeligt, når p og q vælges i størrelsesordenen 100 cifre, idet n så bliver på omkring 200 cifre.

Hvis beregningstiden f.eks. blev reduceret med en faktor 10^6 som følge af hardwaremæssige forbedringer, ville det være muligt tilsvarende hurtigere at bestemme store primtal i nøglegenerering, og man ville derfor blot vælge primtal af en sådan størrelse, at beregningstiden for faktorisering igen ville blive uforholdsmæssig stor.

Det bemærkes, at man i midten af 1994 faktoriserede et n på 129 cifre, som opfylder kravene fra afsnit 3.4.1; en udfordring fremsat af R.Riverst, A.Shamir og L.Adleman. Det tog ca. $\frac{3}{4}$ år og blev foretaget i et netværk af computere [4]. Ved sammenligning med ovenstående figur 3.4 ses, at der er en faktor 3 til forskel.

3.8 Angreb på RSA

RSA kryptosystemet har været udsat for mange angreb gennem tiden, men som allerede nævnt er det indtil dato ikke lykkedes for nogen *kryptoanalytiker* at bryde et RSA kryptosystem, hvis nøglerne opfylder kravene i figur 3.2 og figur 3.3, og hvis n er på omkring 200-300 cifre. Det vil imidlertid ligge undenfor denne rapports rammer at gennemgå alle typer af angreb, så vi nøjes med at skitsere tre typer [14, 181].

Vi vil i det følgende gennemgå 3 eksempler på den første type angreb, der har umiddelbar relevans i forhold til RSA algoritmen og illustrerer nødvendigheden af krav (1) i figur 3.3. Da den offentlige krypteringsnøgle e kan vælges i intervallet $[1; \phi(n) - 1]$, kunne

Cifferetekst: Det forsøges alene ud fra kendskab til cifferteksten at finde klarteksten eller dekrypteringsnøglen.

Kendt-klartekst: Det forsøges ud fra kendskab til en sekvens af klartekster og deres respektive cifferetekster at bestemme dekrypteringsnøglen.

Valgt-klartekst: Som kendt-klartekst, men her kan kryptoanalytikeren selv vælge, hvilke klartekster han vil bruge.

Figur 3.2: Angreb på kryptosystemer.

det være fristende at vælge e så lille som muligt for på den måde at opnå en hurtig krypteringsproces. Betragt for eksempel $e = 3$ i de følgende 2 eksempler:

Det første og mest iøjnefaldende problem er, at brugeren skal sikre sig, at den krypterede meddelelse $M^e > n$. I modsat fald vil krypteringen være meget usikker: hvis $M^e < n$, fås

$$C = M^e \bmod n \qquad \leftrightarrow \qquad C = M^e \tag{3.30}$$

Klarteksten kan altså bestemmes ud fra kendskab til cifferteksten ved almindelig rodtagning.

Et andet problem opstår, hvis brugeren A ønsker at sende den samme meddelelse ud til $x \geq e$ forskellige brugere $B_1, B_2, \ldots, B_x$, alle med $n_i > M$, hvor $i = 1, 2, \ldots, x$. Antag, at en kryptoanalytiker har opfanget e af de x cifferetekster. Han kan da ved hjælp af den Kinesiske Restsætning finde klarteksten, hvis $n_1, n_2, \ldots, n_x$, er indbydes primiske [14, 209].

Endelig kan der opstå det paradoksale, at der for visse sæt af nøgler gælder, at gentagen kryptering af en allerede krypteret tekst vil genskabe originalteksten. Mere formelt kan dette udtrykkes: Givet $C_0 = M^e \bmod n$ og det offentlige nøglesæt (e, n); da kan M findes ved successivt at beregne $C_i = C_i - 1^e \bmod n$, indtil der fremkommer en meningsfyldt meddelelse. Det er klart, at et sådant angreb kun er aktuelt, hvis en meningsfyldt tekst fremkommer efter et mindre antal (f.eks. en million) gentagne krypteringer. R.Rivest har dog vist, at hvis primtallene p og q er valgt i overensstemmelse med figur 3.2, da vil sandsynligheden for succes ved et sådant angreb være meget lille; stort set lig nul [5, 107].

3.9 Nøgleadministration

Indtil nu har vi forelagt, at RSA systemet var sikkert, blot nøglerne var store nok, og brugerne sørgede for at holde deres dekrypteringsnøgler hemmelige, men ret faktisk har nøgleadministra-

tionen også stor betydning. Nøgleadministration er et generelt problem i public key kryptosystemer og et stort og bredt emne, hvorfor vi ikke vil gå i dybden med det her, men blot give et illustrativt eksempel, samt skitsere en af de mange løsningsmetoder.

Eksempel 3.6 Vi har et kryptosystem med to brugere A og B og en kryptoanalytiker K, som gerne vil vide, hvad A og B sender til hinanden. A og B har begge lagt deres offentlige nøgler (e_A, n_A) og (e_B, n_B) i kryptosystemets database. K danner nu to sæt nøgler

(e_{KA}, d_{KA}, n_{KA}) og (e_{KB}, d_{KB}, n_{KB}) og bytter A's offentlige nøgle ud med (e_{KA}, n_{KA}) og B's offentlige nøgle ud med (e_{KB}, n_{KB}).

B ønsker nu at sende en hemmelig meddelelse til A, og tager derfor e_{KA}, som han tror er A's offentlige krypteringsnøgle fra databasen. B krypterer meddelelsen og sender den til A, men K opsnapper meddelelsen og dekrypterer den ved hjælp af sin hemmelige dekrypteringsnøgle d_{KA}.

Efter at have læst meddelelsen - og måske ændret den - krypterer K meddelelsen med A's krypteringsnøgle e_A og sender den til A. Selv hvis B havde påført meddelelsen en digital signatur med sin hemmelige dekrypteringsnøgle, ville K stadigvæk have kunnet ændre og læse meddelelsen. ■

En situation som ovenfor beskrevet er selvfølgelig uacceptabel. Kryptosystemet har mistet sin værdi, da ingen af kravene fra figur 2.3 længere er opfyldt. En måde at undgå dette på er ved at have en betroet *nøgleadministrator* S, som *er* loyal overfor brugerne af systemet; dvs. han hverken foretager eller medvirker til at ændre i de nøgler han opbevarer. Denne nøgleadministrator har en krypteringsnøgle e_S, som er placeret i den offentlige database. Ligeledes har han en dekrypteringsnøgle d_S, som han holder (meget) hemmelig.

Når en person B ønsker at blive registreret som bruger af systemet, sender han en kopi af sin offentlige krypteringsnøgle e_B til S, som genererer et *certifikat* C_B på følgende måde:

$$C_B = d_S(ID_B, e_B, T) \tag{3.31}$$

Her betyder $d_S(x, y, z)$, at S krypterer meddelelsen (x, y, z) med sin hemmelige nøgle (d_S, n_s). S skriver således under på, at oplysningerne i C_B er korrekte. ID_B er en identifikation af B (f.eks. et navn), e_B er B's krypteringsnøgle, og T er et tidsstempel, der kan bruges til at afgøre, om en nøgle er up-to-date, det vil sige i brug. Inden certifikatet tages i brug, sender S det til B, som verificerer, at indholdet er korrekt ved at dekryptere det med e_S. Er indholdet, som det skal være, giver B nøgleadministratoren besked derom, ellers genererer B et nyt sæt nøgler og proceduren gentages. Den tilsvarende procedure gennemføres for bruger A.

Hvis bruger A ligesom i eksempel 3.6 ønsker at sende en meddelelse til B, retter A kontakt til S og udbeder sig B's certifikat, som han dekrypterer for at få fat i en kopi af B's offentlige nøgle. Herefter kan A med med sindsro kryptere meddelelsen med e_B og sende den til B.

Ovenstående procedure har et svagt led, nemlig nøglen d_S. Dette problem kan dog løses ved hjælp af et såkaldt træ autenticitets skema, der gør brug af en en-vejs hashfunktion. Det vil her føre for vidt, at gennemgå proceduren, og der henvises i stedet til [5, 170].

Kapitel 4

Specifikation og konstruktion

Dette kapitel består af fire dele: Først gennemgås den specifikation, der ligger til grund for det RSA-program, vi har lavet. Derefter vil vi skitsere strukturen af programmet, idet vi kun vil beskrive den endelige udgave i C++ og ikke fordybe os i SML-prototypen, da denne kun blev brugt til at få en ide om, hvordan programmet skulle struktureres. Dernæst beskriver vi konstruktionsfasen og gennemgår de enkelte program-moduler, og til sidst resumerer vi resultaterne af den afprøvning, vi har foretaget af programmet.

4.1 Specifikation

Da opgaveformuleringen ikke stillede nogle krav til programmet, udover at det skulle være muligt at benytte digitale signaturer, startede vi med at udarbejde en målsætning og afgrænsning for programmet:

Programmet skal implementere et simpelt RSA kryptosystem. Det skal for flere brugere være muligt at kryptere små meddelelser til hinanden. Med programmet skal det desuden være muligt at benytte en eller flere former for digitale signaturer på disse meddelelser. Hver bruger skal desuden kunne generere og gemme sit eget nøglesæt. Der skal derimod ikke implementeres en central, offentlig database, og vi vil antage, at den computer, programmet afvikles på, er fuldstændig sikret imod angreb. De nøgler, som programmet skal arbejde med, skal være af en vis størrelse, så der skal derfor laves en datatype, der gør det muligt at arbejde med store heltal. Ved implementationen af datatypen skal der lægges vægt på valget af algoritmer, så beregningerne kan gennemføres i overkommelig tid. Denne datatype skal derefter blandt andet bruges til at implementere forskellige primtalstest, der kan bruges til at finde store primtal. Disse primtalstest skal ligeledes vælges, så beregningerne foretages i overkommelig tid.

Ovenstående målsætning bruges som specifikation for programmet. Med hensyn til heltalsdatatypen, har vi besluttet at foretage en implementation, som tillader, at man arbejder med heltal, der er betydeligt større end de indbyggede heltalstyper i programmeringssprogene SML og C++. Med overkommelig tid mener vi, at svartider i størrelsesordnen minutter ved primtalstest er acceptable.

Figur 4.1: Skematisk oversigt over program-moduler.

0	1	2	3	4	5	6	7	8	9	10
			x	x	x	x	x	x	x	x

Figur 4.2: Repræsentation af store heltal som et array.

4.2 Programstruktur

Figur 4.1 viser, hvordan programmet er struktureret. Udfra denne opbygning har vi først lavet en prototype i SML, som vi har brugt som grundlag for den endelige version i C++. SML prototypen har taget ca. 30-40 mandetimer at færdiggøre, mens det har taget ca. 500-600 mandetimer at udvikle udgaven i C++. I det følgende vil vi beskrive de overordnede funktioner af de otte moduler. Vi henviser den interesserede læser til specifikationerne på bilag 1-8 og implementationerne på bilag 9-16.

4.3 Konstruktion

Som figur 4.1 viser, er programmet opbygget af otte moduler. I `rsa` samles alle komponenterne til det færdige RSA kryptosystem. Vi vil i det følgende kort gennemgå hovedtrækkene i de enkelte moduler. Modulet `num` behandles noget grundigere end de øvrige, dels fordi `num` implementerer heltals-datatypen, og dels fordi dette modul har været det mest tidskrævende at designe.

Modulet `def` (se bilag 2 og 10) indeholder datatyper og funktioner, som er fælles for alle moduler. Blandt andet er her defineret datatypen `dword_reg` der kan opfattes som et 32-bit cpu-register:

```
reg[low]    reg[high]
00FF_16 000F_16
reg_x       00FF000F_16
```

Datatypen `dword_reg` er en union med to felter: reg, som er et array med to 16-bit heltal, og `reg_x`, der er et 32-bit heltal. I en union placeres felterne i det samme lagerområde, så hvis f.eks. `reg[low]` $= 00FF_{16}$, og `reg[high]` $= 000F_{16}$, så vil `reg_x` $= 00FF000F_{16}$. Motivationen for at bruge denne datatype er dels, at det gør det nemmere i modulet `num` at foretage addition, subtraktion, multiplikation og division og dels, at det senere hen vil gøre det nemmere at omskrive denne del af koden til assemblerkode.

Modulet `random` (se bilag 7 og 15) er et lille modul, som implementerer den tilfældigtals generator, der er beskrevet i kapitel 3.6 og appendix E.

Modulet `num` (se bilag 5 og 13) er en implementation af store heltal, og det er det største af alle otte moduler. Vi har valgt at repræsentere store heltal som et array af 16-bit heltal (words):

Størrelsen af arrayet (her 11) er bestemt af konstanten max, som er erklæret i def. Som figur 4.2 viser, er det mindst betydende word længst til højre på plads nr. max-1, mens det mest betydende word (msw) i eksemplet er på plads nr. 3. Motivationen for at vælge en

statisk datatype (et array) fremfor en dynamisk datatype (f.eks. en enkelt-hægtet liste), er primært, at det er nemmere at implementere og at den er hurtigere at arbejde med, men dette vil blive diskuteret mere uddybende i kapitel 6.

Negative tal repræsenteres ved 2's komplement, som fås ved at invertere alle bit og til sidst lægge 1 til resultatet. Hvis man f.eks. vil negere tallet $x = 0001101100011110_2 = 6942_{10}$, fås $-x = 1110010011100001_2 + 1_2 = 1110010011100010_2 = 58594_{10}$. Bemærk, at repræsentationen er konsistent, idet $x + -x = 6942 + 58594 = 65536 = 0 \bmod 65536$. Den enumererede variabel `sign` $\in \{$`pos`, `neg`$\}$ angiver, om tallet skal opfattes som positivt eller negativt.

Addition foregår ved, at man, så længe der er elementer, adderer to words og menten fra forrige word-addition. Hvis hver af disse word-additioner foretages i et `dword_reg`, vil resultatet stå i `reg[low]`, mens menten vil stå i `reg[high]`. Subtraktion foregår på samme måde, bortset fra, at menten er negativ. Addition og subtraktion har begge kompleksitet $T(n) = n$, hvor n er antal betydende words (dvs. max - msw + 1), idet de begge kun kræver gennemløb af en for-løkke.

Multiplikation foretages på samme måde, som man manuelt multiplicerer to tal og kræver derfor gennemløb af to for-løkker. Det giver kompleksitet $T(n) = n^2$, men man kan dog ved at benytte del-og-hersk princippet gøre dette noget bedre. I [8, 80] vises, hvordan multiplikation af to n-words heltal kan udføres med kompleksitet

$$3^{\log_2 n} < 4^{\log_2 n} = n^2, \tag{4.1}$$

hvis det antages, at multiplikation af to words kan udføres i konstant tid. Selvom dette er en markant bedre kompleksitet, har vi alligevel valgt at implementere den simple metode med kompleksitet $T(n) = n^2$, fordi algoritmen for den er meget enkel sammenlignet med del-og-hersk algoritmen.

Division mellem to heltal er en kompliceret algoritme og derfor direkte implementeret i overensstemmelse med [9]. Denne algoritme har også kompleksitet $T(n) = n^2$. Ved hjælp af de fire grundlæggende regneoperatorer laves en funktion fastexp, som med algoritmen i [11, 46] hurtigt beregner $y = x^k \bmod n$. Funktionen fastexp bruges både når man krypterer og dekrypterer.

```
num fastexp(num x, num k, num n)
{
    num Res(1), Two(2);          // Res = 1, Two = 2
    while (k>=Two)
    {
        if (!k.is_even())
            Res = Res*x%n;
        x = x*x%n;
        k = k>>1;                // k = k/2
    }
    if (!k.is_even())
        Res = Res*x%n;
    return Res;
}
```

Modulet convert (se bilag 1 og 9) bruges til at læse et bestemt antal bytes i en input fil, konvertere denne blok af bytes til et heltal (en num), påtrykke dette heltal funktionen fastexp, og til sidst skrive resultatet i en output fil. Det ligger uden for denne rapports rammer at gå i detajler med dette modul.

Modulet keylist (se bilag 3 og 11) er en simpel implementation af en enkelt-hægtet liste lavet direkte efter opskriften i [8, 40]. Formålet med modulet er, at holde styr på nøglerne i RSA kryptosystemet ved at hægte nøglerne sammen i en telefonbog, hvor man enten kan slå nøgler op, slette nøgler eller indsætte nye nøgler.

Modulet keyset (se bilag 4 og 12) genererer nøglerne i RSA kryptosystemet. Tallene af e, d og n beregnes som beskrevet i figur 3.1: e findes ved indsættelsesmetoden, dvs. der genereres et tilfældigt tal, som checkes for at være primisk med n, og d beregnes med Euklids udvidede algoritme (se appendix B).

Modulet prime (se bilag 6 og 14) er en primtalsgenerator, som er implementeret udfra beskrivelserne i kapitel 3 og appendix C og D. Vi har valgt, at implementere alle de gennemgåede primtalstest, selvom det i praksis kun er Rabins test, der benyttes. På den anden side gør f.eks. Fermats test det muligt hurtigt at generere forholdsvis store primtal på kort tid, hvilket må siges at være en fordel i en testsituation (se figur 4.4). Funktionen til beregning af Jacobi-symbolet er baseret på en rekursiv algoritme fra [5, 106], men er implementeret iterativt.

Modulet rsa (se bilag 8 og 16) er en funktionsadministrator, dvs. en skal, som knytter de syv andre moduler sammen til et RSA kryptosystem. Ved hjælp af en lille parser afgøres, hvillke funktioner brugeren ønsker at udføre, og derefter aktiveres de relevante moduler. Det kommandolinie baserede brugergrænseflade er beskrevet i næste kapitel.

4.4 Afprøvning

I dette afsnit præsenteres kort resultaterne af afprøvningen af udvalgte program-moduler: num, prime, keyset og rsa. Vi vil udelukkende koncentrere os om modulernes effektivitet, da vi løbende under programudviklingen har testet om modulerne virker korrekt. Afprøvningen er foretaget under DOS på en PC med en Pentium 60 Mhz processor, og tidstagningen er foretaget ved hjælp af den indbyggede Timer i Borland C++ version 3.1.

4.4.1 Num

En fuldstændig test af alle funktioner og operatorer i num, ville fylde flere sider, og iøvrigt være uinteressant. Vi vil derimod opsumere de mest interessante resultater: addition og subtraktion udføres begge med lineær tidskompleksitet og multiplikation udføres med kvadratisk tidskompleksitet (jf. afsnit 4.3). Division viste sig i praksis at have en betydeligt bedre tidskompleksitet end forventet. For at give et indtryk af beregningshastigheden i RSA algoritmen har vi i figur 4.3 gengivet et udpluk af kørselstiderne for fastexp. Bemærk, at størrelsen af x ikke har den store betydning, mens en fordobling k omtrent giver dobbelt så lange kørselstider. En fordobling af n giver en kørselstid, der er cirka 3 gange så lang.

$n = 64$	$k = 64$	$k = 128$	$k = 256$
$x = 64$	0,31	0,64	1,25
$x = 128$	0,32	0,64	1,30
$x = 256$	0,34	0,67	1,32

$n = 128$	$k = 64$	$k = 128$	$k = 256$
$x = 64$	0,66	1,38	2,79
$x = 128$	0,67	1,40	2,76
$x = 256$	0,70	1,40	2,79

$n = 256$	$k = 64$	$k = 128$	$k = 256$
$x = 64$	2,20	4,67	9,14
$x = 128$	2,23	4,69	9,30
$x = 256$	2,28	4,92	9,34

Figur 4.3: Kørselstider for x^k mod n i sekunder. I tabellen betyder $x = 64$, at x er på 64 bit (og tilsvarende for k og n).

4.4.2 Prime

Vi har valgt at teste modulet prime ved at gennemføre 25 tidsmålinger pr. primtalstest, og så betragte middelværdien, som et estimat for den forventede kørselstid, selvom der kan forekomme såvel væsentligt hurtigere eller langsommere kørselstider. Primtalstestene er gennemført med 100 vidner; Rabins test dog kun med 50 vidner. Vi har valgt generere primtal på 16, 32, 64 og 128 bit svarende til cirka 5, 10, 20 og 40 cifre. Nedenfor er har vi gengivet resultaterne for de fire primtalstest, der er beskrevet i appendix D. Bemærk, at kørselstiderne stiger voldsomt, når primtalslængden vokser.

Primtalslængde i bit Rabin 50 vidner Solovay-Strassen 100 vidner Lehmann-Peralta 100 vidner Fermat 100 vidner 16 1,8 2,8 1.5 1.5 32 5,8 7,0 4,4 4.5 64 24 22 20 18 128 249 348 531 304

Figur 4.4. Kørselstider i sekunder for fire primtalstest.

4.4.3 Keyset

Under afprøvning af keyset har det vist sig, at kørselstiderne i praksis ligger meget tæt på den tid, det tager at finde de to primtal p og q. Dette skyldes, det tager væsentligt længere tid, at finde et primtal, end det gør at bestemme e og d (jf. afsnit 4.3).

4.4.4 RSA

I RSA programmet har vi målt, hvor lang tid det tager at kryptere en tekstfil på 400 bytes med forskellige nøglesæt. Disse resultater er gengivet i figur 4.5. Bemærk, at jo større n er, desto langsommere bliver regningerne. Bemærk også, at signering tager cirka dobbelt så lang tid som almindelig kryptering, fordi vi signerer hele meddelelsen istedet for kun at signere en del af den.

Længde af n i bit Kryptering Dekryptering Kryptering med signatur Dekryptering med signatur 32 880 420 276 276 64 600 302 198 197 128 276 124 98 95 256 100 48 32 32

Figur 4.5. Kørselstider i bits pr. sekund for kryptering og dekryptering.

Til sidst skal det bemærkes, at vi under afprøvningen af rsa konstaterede, at programmet i visse tilfælde "gik ned". Vi lokaliserede fejlen til at være fragmentering af hukommelsen, men havde ikke kapacitet til at rette fejlen. Dette problem diskuteres nærmere i kapitel 6.

Kapitel 5

Brugervejledning

I dette kapitlel beskrives det `rsa` program, der er vedlagt på diskette bagest i rapporten, og som skal afvikles på en almindelig PC med DOS. Programmet er en prototype af et RSA kryptosystem, og brugen af det kræver ikke kendskab til hverken programmering eller matematikken bag RSA kryptering. Da programmet kun er en prototype, har vi ikke implementeret en fælles offentlig telefonbog, men hver bruger må vedligeholde sin egen private telefonbog.

I kapitlet gennemgås dels, hvordan man laver et nøglesæt, og dels hvordan man med eller uden signatur krypterer og dekrypterer filer. Ligeledes gennemgår vi de fejlmeddelelser, der kan forekomme under brugen af programmet.

5.1 Interface

`rsa` programmet er kommandolinie baseret og syntaksen er

```
rsa <command> <userid> <file>
```

`command` angiver hvilken funktion eller kommando, der ønskes udført. `userid` er et heltal repræsenterende en bruger i telefonbogen, og `file` er et filnavn.

`rsa` programmet består af i alt 3 filer:

- `rsa.exe`, som er hovedprogrammet;
- `key.dat`, som indeholder ens offentlige og private nøgle; og
- `phone.dat`, som er en telefonbog over de andre brugere i systemet.

Til at starte med er det kun `rsa.exe`, der nødvendig, da programmet selv danner de to andre filer, hvis de ikke findes. I det følgende gennemgås detaljeret brugen af hver enkelt funktion i programmet.

5.2 Brug af `rsa` programmet

I dette afsnit gennemgår vi `rsa` programmets syntaks i detaljer og viser med eksempler, hvordan man bruger programmet.

Syntax: `rsa c <filnavn>`

Beskrivelse: Denne kommando har to funktioner alt efter, hvornår den bliver kaldt.

1. Første gang denne kommando kaldes, genereres (eng: create) et komplet nøglesæt. Dette nøglesæt gemmes i filen `key.dat`, og dernæst laves en kopi af den offentlige nøgle, som gemmes i filen specificeret ved filnavn.
2. Hvis filen `key.dat` eksisterer, laves en kopi af den offentlige nøgle, som gemmes i filen specificeret ved filnavn. Det er filen filnavn, der skal distribueres til de øvrige brugere i systemet, hvorefter den ikke længere er nødvendig og kan slettes.

Ønsker man at generere en ny offentlig og hemmelig nøgle, skal man først slette filen `key.dat`, og derefter bruger kommandoen `rsa c <filnavn>`. Hvis man derimod kun har brug for at distribuere en offentlig nøgle, skal filen `key.dat` ikke slettes, inden man skriver `rsa c <filnavn>`.

Eksempel: A er ny i systemet og ønsker at generere et sæt nøgler og skriver derfor:

```
rsa c a.dat
```

Herved er dannes der to filer: `key.dat`, som indeholder den offentlige og hemmelige nøgle; og `a.dat` som kun indeholder den offentligenøgle, der skal distribueres.

B informeres om, at A er blevet tilmeldt systemet og vil derfor sende ham en kopi af sin offentlige nøgle og skriver derfor:

```
rsa c b.dat
```

Herved dannes der kun filen `b.dat`, som indeholder en kopi af B's offentlige nøgle. Denne fil distribueres til A.

Syntax: `rsa i <filnavn>`

Beskrivelse: Indsætter (eng: insert) en bruger, hvis offentlige nøgle er i filen filnavn, i telefonbogen. Dette er nødvendigt, hvis man vil sende krypterede meddelelser til denne brugere.

Eksempel: A ønsker at kommunikere med B og C og har modtaget filerne `b.dat` og `c.dat`. A indsætter B og C i telefonbogen ved at skrive kommandoerne:

```
rsa i b.dat
rsa i c.dat
```

Syntax: `rsa p`

Beskrivelse: Viser indholdet af telefonbogen (eng: phonebook) på skærmen.

Eksempel: A ønsker at se hvem, der er registreret i hans telefonbog og skriver derfor:

```
rsa p
```

Herved fremkommer følgende udskrift på skærmen:

```
userid  name
  1      B
  2      C
```

`userid` bruges til at identificere en bruger. Dette benyttes dels til at vedligeholde telefonbogen, og dels som vi senere skal se ved kryptering og dekryptering af filer.

Syntax: `rsa r <userid>`

Beskrivelse: Fjerner (eng: remove) en bruger fra telefonbogen. `userid` er det nummer, brugeren har i telefonbogen, hvilket kan udskrives med

```
rsa p
```

Eksempel: A ønsker at fjerne B fra sin telefonbog, og skriver derfor:

```
rsa r 1
```

Den resulterende telefonbog, der er udskrevet med `rsa p` er:

```
userid  name
  2      C
```

Syntax: `rsa e <userid> <filnavn>`

Beskrivelse: Krypterer (eng: encrypt) en fil med navnet `filnavn`, så kun brugeren med nummeret `userid` i telefonbogen kan læse den. Den krypterede fil kaldes `filnavn.rsa`, og det er denne fil, der skal sendes til brugeren med nummeret `userid`.

Eksempel: A ønsker at kryptere en fil ved navn `message.txt`, så kun C (med `userid` = 2) kan læse dokumentet, og skriver derfor:

```
rsa e 2 message.txt
```

Herefter sender A filen `message.rsa` til C.

Syntax: `rsa d <rsa-fil>`

Beskrivelse: Dekrypterer (end: decrypt) en krypteret fil.

Eksempel: C har modtaget en filen `message.rsa` fra A, og dekrypterer den ved at skrive:

```
rsa d message.rsa
```

Nu kan C læse meddelelsen fra A i filen `message.txt`.

Syntax: `rsa es <userid> <filnavn>`

Beskrivelse: Krypterer og signerer (eng: encrypt with signature) en fil med navnet filnavn, så kun brugeren med nummeret `userid` i telefonbogen kan læse den. Herved dannes en fil med navnet `filnavn.rsa`, og det er denne fil, der skal sendes til brugeren med nummeret `userid`.

Eksempel: A ønsker at kryptere og signere en fil ved navn `message.txt` så kun C (med `userid` = 2) kan læse meddelelsen og skriver derfor:

```
rsa es 2 message.txt
```

Herefter sender A filen `message.rsa` til C.

Syntax: `rsa ds <userid> <rsa-fil>`

Beskrivelse: Dekrypterer (eng: decrypt) en krypteret og signeret fil (eng: signature) fra brugeren med nummeret `userid` i telefonbogen.

Eksempel: C har modtaget en krypteret og signeret fil fra A og dekrypterer den ved at skrive:

```
rsa ds 3 message.rsa
```

i det tilfælde hvor A har `userid = 3` i C's telefonbog.

Syntax: `rsa h`

Beskrivelse: Udskriver en hjælpeskærm.

Eksempel: Hvis man ikke kan huske `rsa` programmets syntaks, kan man skrive:

```
rsa h
```

5.3 Fejlmeddelelser

I dette afsnit gennemgås de fejlmeddelelser, der kan forekomme under brug af `rsa` programmet, og vi angiver de typiske årsager til fejlene.

Fejl: `No phonebook created yet`

Beskrivelse: Filen med navnet `phone.dat` findes ikke i det directory, hvor man står.

Fejl: `'x' is not a command`

Beskrivelse: Syntaksen for brug af programmet er ikke overholdt. Denne fejl opstår typisk, hvis man glemmer at anføre `userid` og/eller `filnavn`, når man krypterer eller dekrypterer en fil.

Fejl: `Invalid file`

Beskrivelse: Fil eksisterer ikke, eller også er den ikke en `rsa`-fil. Denne fejl opstår typisk, hvis man forsøger at dekryptere en fil, der ikke har extensionen `.rsa`.

Fejl: `No user with Id x`

Beskrivelse: Det `userid`, man har angivet, eksisterer ikke. Denne fejl opstår typisk under kryptering og/eller dekryptering og ved sletning fra telefonbogen.

Fejl: `Cannot find file x`

Beskrivelse: Filen ved navn `x` eksisterer ikke. Denne fejl opstår som regel, hvis `key.dat` ikke eksisterer og man forsøger at kryptere eller dekryptere med signatur

---

Fejl: `Block is Zero`

Beskrivelse: Intern programfejl. Der er sket en fejl under nøglegenereringen. Hvis problemet ikke kan løses, så kontakt JKC Software.

Fejl: `Division by Zero`

Beskrivelse: Intern programfejl. Kontakt JKC Software.

JKC Sofware er synonymt med rapportens tre forfattere.

Kapitel 6

Konklusion

Dette kapitel er komponeret i tre dele. I første del diskuterer vi, hvor godt vi har løst problemstillingen fra kapitel 2. Vi påpeger, hvilke teoretiske emner man bør koncentrere sig om, hvis man vil arbejde videre med RSA kryptering. Desuden analyserer vi, hvilke dele af prototypen vi kunne have implementeret bedre, og vi angiver, hvilke funktioner og datatyper man bør opprioritere under konstruktionen af et kommercielt program. Kapitlets anden del er en samlet konklusion for hele projektet. I sidste del perspektiverer vi rapportens vigtigste resultater og fremhæver de teknologiske muligheder, der er ved anvendelsen af digitale signaturer.

6.1 Diskussion

I dette afsnit vil vi først påpege, hvilke teoretiske aspekter vi har undladt at koncentrere os om, men som man ikke bør nedprioritere ved udvikling af et RSA kryptosystem, som skal bruges i praksis. Dernæst vil vi diskutere, hvilke dele af implementationen, vi kunne have lavet bedre.

Man bør sætte sig grundigt ind i de forskellige typer af angreb på RSA kryptosystemet, således at man kan sikre sig bedst muligt imod dem. Ved nøglegenereringen bør man være omhyggelig ved implementationen og vælge tallene i overensstemmelse med kravene i figur 3.2 og 3.3. Vi har valgt at se bort fra dette, da det ikke har betydning for funktionaliteten af programmet. En bedre tilfædigtals generator vil også kunne øge systemets sikkerhed, da primtallene således også vil blive dannet "mere tilfældigt".

Man kan med fordel anvende hashfunktioner, når man implementerer digitale signaturer, da de vil forøge hastigheden af systemet uden at sætte sikkerheden over styr. I stedet for, som vi har valgt at signere hele meddelelsen, kan man nøjes med at signere en hashværdi, dvs. en værdi der beregnes udfra indholdet af meddelelsen. Det er imidlertid meget svært at finde hashfunktioner, som overholder sikkerhedskravene, da hver eneste informationsbit i meddelelsen skal vægtes, således at selv en lille ændring af meddelelsen vil ændre hashværdien. Det skal ligeledes være umuligt at lave to forskellige ændringer, som ophæver hinanden. Da teorien om disse hashfunktioner tilmed er kompliceret, valgte vi ikke at komme nærmere ind på dette store område.

Selv om programmet kun er en prototype, vil vi alligevel påpege, hvilke algoritmer og datastrukturer, vi kunne have implementeret bedre. Vi var fra starten klar over, at

datatypen num, som er vores repræsentation af store heltal, er meget pladskrævende: man allokerer 2max bytes, ligegyldigt hvor stort heltallet er. Dette problem kan løses ved løbende at fastsætte længden af det array, som bruges til at gemme værdien af heltallet, men det var vi ikke rutinerede nok til implementere. Det var især repræsentationen af negative tal (2's komplement), som ikke ville fungere sammen med en variabel længde af arrayet.

Vi holdt derfor fast ved den version af num, der ses på bilag 5 og 13, idet vi var overbeviste om, at vi ikke ville løbe ind i pladsproblemer. Vi blev derfor meget overraskede, over nums "formidable evne"til at fragmentere hukommelsen. Dette resulterede i en del problemer under afprøvningen, og man kan nogle gange komme ud for at programmet "går ned", specielt hvis man arbejder på meget store filer. Hastigheden af num og dermed hele programmet vil kunne forbedres betydeligt, hvis centrale dele af num konstrueres i assembler. En anden løsning er at bruge tredjeparts-produkter som f.eks. matematikmodulet Miracle.

Modulet keylist kan forbedres på flere punkter: En lille forbedring kunne være at bruge binært træ eller et B-træ i stedet for den anvendte enkelthægtede liste, og man kunne desuden også med fordel lave flere operationer på datastrukturen. I praksis ville man nok indsætte en tredjeparts relationsdatabase med dertil hørende funktioner til manipulering af data. Klassen prime burde have været implementeret som en nedarvet klasse af num. Dette var vi desværre ikke rutinerede nok til at nå indenfor den tidsgrænse, vi havde afsat til implementationen af programmet.

Sikkerheden i vores program kunne forbedres ved f.eks. at XOR're indholdet af datafilen KEY.DAT med et password, som yderligere ville sikre hemmeligholdelsen af den private nøgle. Man kunne også sørge for slette maskinens hukommelse, når programmet afsluttes, men da vi har antaget, at brugerens computer er sikker, falder dette uden for projektets rammer. Endvidere ville en implementation af stærke primtal også kunne forbedre sikkerheden af programmet.

6.2 Konklusion

Som nævnt i afsnit 2.3 er formålet med dette projekt er at formidle den matematiske teori bag RSA kryptosystemer og lave et program, hvor man kan benytte digitale signaturer. Dette mål synes at være nået, idet vi i kapitel 2 og 3 giver vores bud på, hvordan den teoretisk del kan gennemgås, mens vi i kapitel 4 og 5 beskriver, hvordan vi i praksis har implementeret digitale signaturer i et RSA program.

Som nævnt i problemstillingen skulle dette projekt gerne fremstå som et samlet oversigt over, hvilke komponenter der indgår i et RSA kryptosystem. På baggrund heraf mener vi, at rapporten blandt andet kan bruges som udgangspunkt for et større og eventuelt tværfagligt projekt.

6.3 Perspektivering

Vi vil nu afrunde projektet med en perspektivering, hvor vi blandt andet understreger rapportens værdi for andre faggrupper end de matematiske og datalogiske, og kort belyser

hvorfor digitale signaturer er en nødvendighed i et moderne informationssamfund.

Ved de fleste former for handel og banktransaktioner er det essentielt, at kravene om autenticitet, integritet og uafviselighed er opfyldt. Når for eksempel to virksomheder handler med hinanden, er der som regel indgået en fælles kontrakt, som begge virksomheder har underskrevet. Idet handel via datanet er blevet mere og mere almindelig, kan håndskrevne underskrifter ikke længere benyttes. Løsningen er at bruge en digital signatur, og i kapitel 3 viste vi, at der er matematisk belæg for at betegne denne form for signatur som sikker.

Imidlertid har dette ikke den store værdi, da der endnu ikke er nogen lovgivning på området. Digitale signaturer har derfor ingen retsvirkning og er følgelig ikke juridisk gyldige. Inden for visse dele af bankverdnen har man dog omgået dette problem ved at indgå en kontrakt med kunderne, således at den digitale signatur (PIN-kode, adgangskode eller lignende) er juridisk gyldig.

Det er dog klart, at en lovgivning inden for området vil rejse en masse politiske, ideologiske, juridiske og etiske spørgsmål, men vi mener alligevel, at der snarest muligt bør lovgives på området, så alle kan drage nytte af informationssamfundets fordele.

Litteratur

[1] L. H. Adleman, C. Pomerance og R. S. Rumely. *On distinguishing prime numbers from composite numbers.* Ann. Math, volume 117, 1983.

[2] Tommy Baunwall og Peter Bertelsen. *PC-RSA, RSA-kryptografering i software.* Datalogisk Afdeling, Matematisk Institut, Aarhus Universitet, 1987.

[3] H. Cohen og A. K. Lenstra. *Primality testing and Jacobi sums.* Mathematical Computing, volume 42, number 165, 1984.

[4] Ivan Damgaard. Kryptologigruppen, Århus Universitet. Telefonsamtale, 1995.

[5] Dorothy E. Denning. *Cryptography og data security.* Addison-Wesley Publishing Company, 1983.

[6] Helge Elbrønd Jensen og Tom Høholdt. *Grundlæggende matematik for dataloger.* Matematisk Institut, 1993.

[7] Thomas W. Judson. *Abstract Algebra: Theory and applications.* PWS publishing company, 1993.

[8] Jeffrey H. Kingston. *Algotithms and Datastructures. Design, Correctness, Analysis.* Addison-Wesley Publishing Company, 1991.

[9] Donald Erwin Knuth. *The Art of Computer Programming, Seminumerical Algorithms.* Addison-Wesley Publishing Company, 1981.

[10] Wayne Patterson. *Mathematical Cryptology for Computer Scientists and Mathematicians.* Rowman & Littlefield, 1987.

[11] Lawrence C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[12] Michael O. Rabin. *Probabilistic Algorithm for Testing Primality.* Journal of Number Theory, volume 12, number 1, februar 1980.

[13] Hans Riesel. *Prime Numbers and Computer Methods for Factorization.* Birkhauser Boston, 1985.

[14] Gustavus J. Simmons. *Contemporary Cryptology, The Science of Information Integrity.* Institute of Electrical and Electronics Engineers, 1992.

[15] Henk C. A. van Tilborg. *An Introduction to Cryptology.* Kluwer Academic Publishers, 1988.

[16] Thomas H. Cormen, Charles E. Leiserson og Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1994.

[17] Donald Erwin Knuth. *The Art of Computer Programming, Fundamental Algorithms.* 2. udgave. Addison-Wesley Publishing Company, 1968.

Bilag A

RSA algoritmen

I dette appendix bevises sætning 3.1 (RSA algoritmen). Først defineres Eulers totient funktion, og dernæst bevises fire lemmaer ved hjælp af Fermats lille sætning og Eulers udvidelse af Fermats lille sætning.

Definition 4 (Eulers totient funktion) Eulers totient funktion $\phi(n)$ er defineret ved:

$$\phi(n) = \Big|\{1 \leq i \leq n | gcd(i,n) = 1\}\Big| = n \prod_{p|n} \Big(1 - \frac{1}{p}\Big) \tag{A.1}$$

■

Bemærkning: $\phi(n)$ er altså antallet af positive heltal, der er mindre end n og primiske med n, se [7]. Hvis p og q er to primtal, fås følgende to specialtilfælde:

$$\begin{aligned} \phi(p) &= p - 1 \\ \phi(pq) &= (p-1)(q-1) \\ &= pq - p - q + 1 \end{aligned} \tag{A.2}$$

Bevis. Vi viser det sidste af de to specialtilfælde ved gennemgang af de forskellige led:

pq	:	antallet af positive tal mindre end n.
$-p$	:	antallet af tal i, hvor $\gcd(i,n) = q$, dvs: $q, 2q, 3q, \ldots, pq$.
$-q$	:	antallet af tal i, hvor $\gcd(i,n) = p$, dvs: $p, 2p, 3p, \ldots, qp$.
1	:	pq er trukket fra 2 gange.

□

Sætning 5 (Fermats lille sætning) *Lad p være et primtal og a et heltal, hvor $\gcd(p,a) = 1$. Så gælder:*

$$a^{p-1} \bmod p = 1 \tag{A.3}$$

Bevis. Se [6, 4.31]. □

Sætning 6 (Eulers udvidelse af Fermats lille sætning) *Lad n og a være to heltal, hvor $\gcd(n,a) = 1$, og lad $\phi(n)$ være Eulers totient funktion. Så gælder:*

$$a^{\phi(n)} \bmod n = 1 \tag{A.4}$$

Bemærkning: Hvis n er et primtal, er sætning A.2 og sætning A.3 identiske.
BEVIS. Se [7, 99]. □

Vi viser nu sætning 3.1 ved hjælp af fire lemmaer.

Lemma 7 *Lad p og q være primtal, og lad $n = pq$. Hvis $M \in [0; n-1]$ og $\gcd(M,n) = 1$, så gælder:*

$$M^{\phi(n)+1} \bmod n = M \tag{A.5}$$

BEVIS.

$$\begin{aligned} M^{\phi(n)+1} \bmod n &= MM^{\phi(n)} \bmod n \\ &= (M \bmod n)(M^{\phi(n)} \bmod n) \bmod n \\ &= M \qquad\qquad (A.3) \end{aligned} \tag{A.6}$$

□

Lemma 8 *Lad p og q være primtal, og lad $n = pq$. Så gælder*

$$p^{\phi(n)+1} \bmod n = p \tag{A.7}$$

BEVIS.

$$\begin{aligned} & & p^{q-1} \bmod q &= 1 \qquad\qquad (A.2) & \text{(A.8)} \\ &\Rightarrow & (p^{q-1} \bmod q)^{p-1} \bmod q &= 1 & \text{(A.9)} \\ &\Leftrightarrow & p^{(p-1)(q-1)} \bmod q &= 1 & \text{(A.10)} \\ &\Leftrightarrow & p^{(p-1)(q-1)} &= 1 + tq, \qquad t \in \mathbb{Z} & \text{(A.11)} \\ &\Leftrightarrow & p^{\phi(n)+1} &= p + tn \qquad (A.1) & \text{(A.12)} \\ &\Rightarrow & p^{\phi(n)+1} \bmod n &= (p + tn) \bmod n & \text{(A.13)} \\ & & &= p & \text{(A.14)} \end{aligned}$$

□

Lemma 9 *Lad p og q være primtal, og lad $n = pq$. Hvis i er et positivt heltal, så gælder:*

$$(p^i)^{\phi(n)+1} \bmod n = p^i \bmod n \tag{A.15}$$

BEVIS.

$$\begin{aligned} (p^i)^{\phi(n)+1} \bmod n &= (p^{\phi(n)+1})^i \bmod n \\ &= p^i \bmod n \qquad (A.5) \end{aligned} \tag{A.16}$$

□

Lemma 10 *Lad p og q være primtal, og lad $n = pq$. Hvis i er et positivt heltal, og A er et positivt heltal, hvor $\gcd(A, n) = 1$, så gælder:*

$$(Ap^i)^{\phi(n)+1} \bmod n = Ap^i \bmod n \tag{A.17}$$

BEVIS.

$$\begin{aligned}(Ap^i)^{\phi(n)+1} \bmod n &= A^{\phi(n)+1}(p^i)^{\phi(n)+1} \bmod n \\ &= Ap^i \bmod n \qquad \text{(A.1)og(A.6)}\end{aligned} \tag{A.18}$$

□

Vi kan nu bevise sætning 3.1 for ethvert $M \in [0; n-1]$. Da $n = pq$, hvor p og q er primtal, er $\gcd(M, n)$ lig 1, p eller q. Hvis $\gcd(M, n) = 1$, så bruges lemma A.4; hvis $\gcd(M, n) = p$, er enten $M = p$, $M = p^i$ eller $M = Ap^i$, og så bruges lemma A.5, A.6 eller A.7; tilsvarende hvis $\gcd(M, n) = q$. Derfor fås:

$$\begin{aligned}(M^e \bmod n)^d \bmod n &= M^{\phi(n)t+1} \bmod n \\ &= M\end{aligned} \tag{A.19}$$