**UNIVERSITETET I OSLO**
**Institutt for informatikk**

# Implementasjon av elliptisk kurve kryptering

Andreas Dobloug

Hovedoppgave

**Våren 2002**

# Innhold

# 1

# Forord

Kjeller, våren 2002.

Først av alt vil jeg takke min veilleder for svært nyttig tilbakemelding. Uten hans tålmodighet ville jeg aldri klart å fullføre denne oppgaven. Jeg vil også takke Universitetsstudiene på Kjeller for kontorplassen jeg har disponert samt det gode studiemiljøet de har klart å skape. Sist, men ikke mist, vil jeg takke mine foreldre for all støtte de har gitt meg.

Denne oppgaven er i stor grad et resultat av selvstendig arbeid. Ettersom jeg har bakgrunn fra informatikk, og av den grunn har hatt fokus på helt andre ting enn en matematikkstudent i de tidligere studiene, har en del av matematikken voldt mye hodebry. Jeg har derfor brukt svært lang tid på selve implementasjonsbiten i oppgaven. I implementasjonen har jeg hatt fokus på hvilke algoritmer som vil oppnå stor effektivitet samt på å lage abstraksjonslag for å skjule den underliggende matematikken. Jeg vil påstå at dette er oppnådd til en stor grad.

Andreas Dobloug

# 2

# Innledning

Kryptologi er læren om kryptografi og kryptoanalyse. Kryptografi er metoder for å kryptere (skjule) en melding for å gjøre den uleselig for andre enn de med kjennskap til en nøkkel. Kryptoanalyse er metoder for å analysere en kryptert melding for å få ut den skjulte meldingen eller finne nøkkelen.
Kryptografi oppstod med behovet for sikker kommunikasjon over en usikker kommunikasjonskanal. Kryptografisystemene har vært under en stor utvikling det siste århundret. Fra å være noe som man forbant med magi og som i mange år stod stille med tanke på utvikling[Kha67] – har det vokst frem en hel vitenskap.

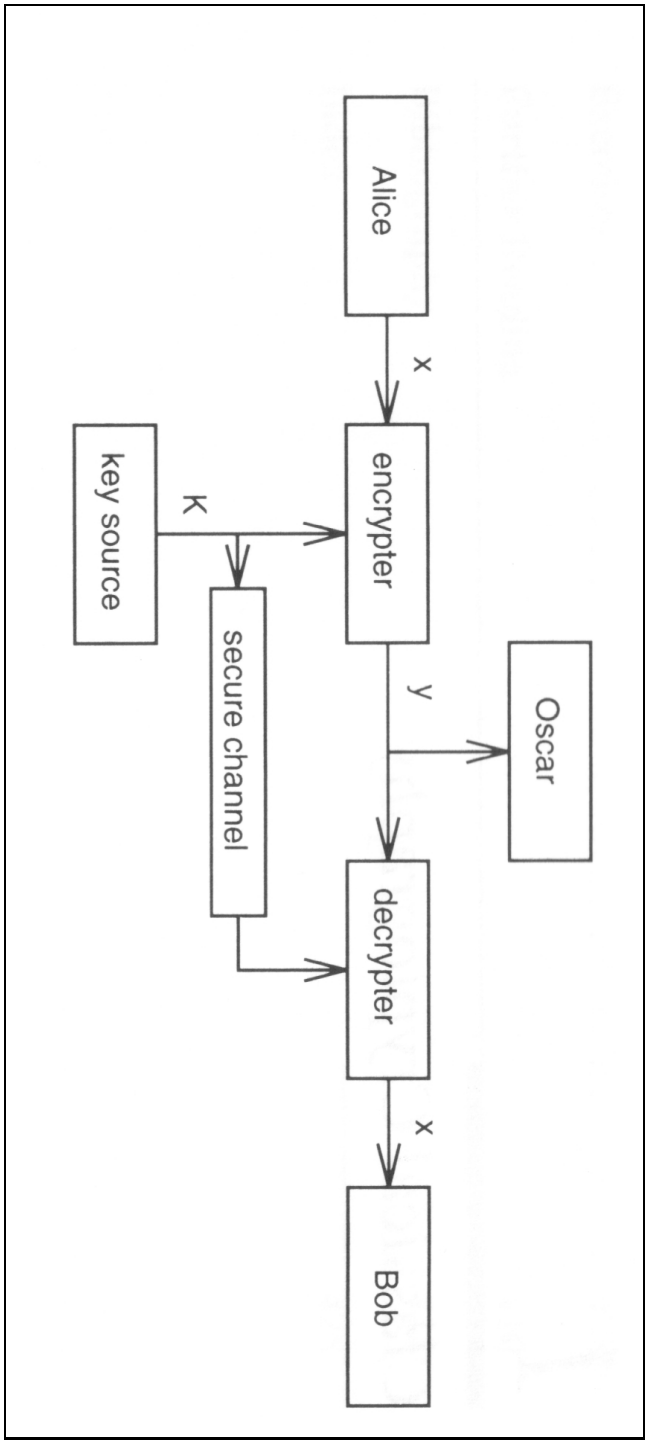
Det er nødvendig å bruke systemer som har god nok sikkerhet. Sikkerheten defineres ut i fra hvor lang tid det tar å dekryptere en melding uten nøkkelen. Kravet til sikkerhet er at det ikke må være mulig å dekryptere en melding så lenge den har relevans og kan misbrukes[1]

Det er vanlig å stille krav til at det skal ta flere år å dekryptere en melding uten kjennskap til nøkkelen. Man forventer m.a.o. at en melding kan fanges opp uansett hvilke kanaler man benytter for å sende den.

Sikkerheten i moderne krypteringssystemer baserer seg på s.k. enveis funksjoner (E.g.: "one-way trap door functions" (se 3.3)). Enveisfunksjoner med fallem er funksjoner som er relativt enkle å beregne. Reversering, derimot, er vanskelig å beregne i polynomiell tid uten en nøkkel (se 3.1). Frem til Diffie og Hellmann[DH76] i 1976[2] foreslo bruk av et asym-

[1]Under 2. Verdenskrig ble mye av kommunikasjonen til aksemaktene dekryptert av de allierte. Dette skyldtes både slepphendt bruk av nøkler og bruk av et system som ikke gav god nok sikkerhet, se [Kha67]

[2]Asymmetrisk kryptering ble oppfunnet allerede i 1970 av James Ellis og senere uavhengig av dette av Clifford Cooks. Diffie og Hellmann har ofte fått æren av denne oppdagelsen ettersom de produserte den første artikkelen.




Figur 2.1: Sending av melding over usikker kanal krever kryptering

metrisk nøkkelpar brukte man utelukkende symmetriske krypteringssystemer (dvs. nøklene for å kryptere og dekryptere var identiske). Dette innebar store problemer i forbindelse med distribusjon av nøklene – som måtte sendes med f.eks. kurér eller en annen sikker (ikke avlyttbar) kanal. Asymmetriske algoritmer tok sikte på å løse dette problemet ved at en av nøklene var offentlig tilgjengelig (mer om dette i neste kapittel). Allerede i 1977 kom det første asymmetriske krypteringssystemet: RSA [RSA77]. Sikkerheten i asymmetriske chifre ligger i at det ikke finnes en polynomiell metode for å avlede hverken meldingen eller den hemmelige nøkkelen. Mer om dette i neste kapittel.

Asymmetriske chifre åpner for en rekke sikkerhetstjenester[3]; de viktigste er:

**Ikke-fornekting:** Avsender kan ikke benekte å ha sendt en melding i ettertid.

**Integritet:** Man kan kontrollere om meldingen har blitt endret.

**Konfidensialitet:** Meldingen kan ikke bli lest av andre enn de som innehar de korrekte nøklene.

Dette åpner for en rekke nye anvendelsesområder for kryptografi. De viktigste for øyeblikket er å sikre transaksjoner i forbindelse med internettbanker og handel på internett (se 3.5).

I disse dager holder flere og flere banker og kredittkort-selskap på å innføre nye bankkort med innebygget funksjonalitet for signering av meldinger. På disse kortene ønsker man samtidig å legge inn "rollesertifikater". Disse "sertifikatene" inneholder nøkler man kan utstede til ansatte som bevis på at de har fullmakt til å foreta innkjøp. Det er også planlagt å legge inn andre ting som f.eks. telekort og månedskort til offentlig transport[Hov99]. Sosial- og helsedepartementet har planlagt å lage elektroniske pasientkort[Ukj99].

[3]Bortsett fra ikke-fornekting kan de andre operasjonene implementeres ved hjelp av symmetriske teknikker. De andre tjenestene tas med for ordens skyld

Det er dog ikke uproblematisk å legge alt for mye inn på et slikt smartkort: Systemet man har valgt å bruke i Norge (Statens Datasentral har allerede et under utprøving.) har f.eks. kun plass til ca. 16kB. (Andre kortsystemer har tilsvarende plassproblemer). Det er m.a.o. ønskelig å ha flest mulig nøkler på et smartkort uten at dette går på bekostning av sikkerheten. Ettersom man er interessert i relativ stor sikkerhet for alle transaksjoner har nøkkelstørrelsen vokst til e.g. 1024bit for implementasjonen som brukes av Statens Datasentral[Hov99]. Det har derfor blitt nødvendig å finne nye systemer som har relativt liten nøkkelstørrelse samtidig som sikkerheten ivaretas. Et av disse systemene er basert på "elliptiske kurver", og vil være fokus for denne oppgaven. "Elliptiske kurver" vil bli behandlet i kapittel 4, og i de påfølgende kapitlene vil jeg ta for meg hvordan krav til sikkerhet og hastighet påvirker algoritmedesignen. Jeg vil også se på hvorvidt et objektorientert språk egner seg for å implementere elliptisk kurve kryptering. Til slutt vil jeg diskutere hvilke algoritmer som lar seg konvertere til maskinvare og i lys av dette diskutere hvilken gevinst dette gir. Først vil jeg dog gi en kort innføring i kryptografi.

# 3

# Kryptografisk bakgrunnsmateriale

Jeg vil i dette kapitlet først gi en kort innføring i kompleksitetsteori. Deretter skal vi se på forskjellige kryptografiske funksjoner og definisjoner brukt i denne oppgaven.

## 3.1 Kompleksitetsteori

Først et par definisjoner:

**Algoritme:** En detaljert oppskrift for hvordan man skal løse et problem.

$\mathcal{O} - notasjon$**:** Viser hvor mange operasjoner en algoritme må gjennomføre og kompleksiteten av operasjonene avhengig av datamengden som skal behandles. Algoritmene deles deretter inn i kompleksitetsklasser. $\mathcal{O}$-notasjon (også kalt "ordenen" til algoritmen) er definert slik at *alle* konstanter ignoreres. Dersom en operasjon har $n + k$ eller $k * n$ operasjoner, hvor $k$ er en konstant, vil $\mathcal{O}(n + k) = \mathcal{O}(kn) = \mathcal{O}(n)$. Tilsvarende vil $\mathcal{O}(k) = \mathcal{O}(1)$ dersom $k$ er konstant. Poenget med dette er å inndele problemer i forskjellige kompleksitetsklasser. $\mathcal{O}$-notasjon er definert som følger: La $f$ og $g$ være funksjoner. Vi skriver $f(n) = \mathcal{O}(g(n))$ dersom det eksisterer $c, n_0 \in N$ slik at $\forall n \geq n_0 : 0 \leq f(n) \leq c * g(n)$

**Kompleksitetsteori** er teorier for hvorfor enkelte problemer er vanskeligere å løse på en datamaskin enn andre[Pap95]. Kompleksitetsteorien kategoriserer problemer i kompleksitetsklasser avhengig av hvor vanskelig de er å løse.

### 3.1.1 Kompleksitet og kryptografi

Definisjonen av kompleksitetsklasser henger nøye sammen med definisjonen av en Turingmaskin. Interesserte bes lese[GJ79]. Kompleksitetsklassene som er omtalt i denne oppgaven er

**Logaritmisk-tid algoritme ($L$):** Kjøretiden til algoritmen vokser logaritmisk i forhold til input. Algoritmer av denne typen har orden $\mathcal{O}(\log n)$

**Polynomiell-tid algoritme ($FP$):** Antall beregninger i forhold til input er polynomiell. Dette vil si kjøretiden til algoritmen har orden på formen $\mathcal{O}(n^k)$.

**Eksponensiell-tid algoritme ($EXP$):** Antall beregninger i forhold til input er eksponensiell. Alle funksjoner som vokser raskere enn funksjoner i $L$ og $FP$ havner i denne kategorien.

**Definisjon:** Klassen $NP$ består av problemer som ikke er løsbare med en polynomisk-tid algoritme. $coNP$ er inversene til disse problemene (dvs. hvor man spør etter det motsatte (e.g. setter *ikke* foran spørsmålet))[1]. Betegnelsen $NP$-komplett betyr at alle problemer i klassen $NP$ er minst like vanskelig som dette problemet.

**Probabilistiske algoritmer:** Det finnes en klasse av probabilistiske algoritmer. Probabilistiske algoritmer kan anvendes for å undersøke et problem som ligger i $NP$. Ved å anvende algoritmen flere ganger kan man øke sannsynligheten på at svaret er korrekt. Det finnes flere typer av disse algoritmene. (Se [GB96, Pap95] for nærmere beskrivelser.)

### 3.1.2 TIME og SPACE-kompleksitet

Minnebruken til en algoritme er ofte like viktig som antall beregninger. Selv om et *problem* skulle vise seg å være løsbart i $\mathcal{O}(n)$, kan det tenkes at

[1] Merk: $NP \cap coNP \neq \oslash$

man bruker **SPACE**$((f^k(n)))$ minne[2]. Problemet kan da, for store nok verdier av $n$ og $k$, være praktisk talt uløselige. Det finnes tilsvarende klasser for tids-kompleksitet.

Følgende definisjoner og korollarer vil være nyttige for denne oppgaven:

- Klassen av problemer som løses i deterministisk tid kalles $TIME(f)$
- Klassen av problemer som løses i deterministisk rom kalles $SPACE(f)$
- Klassen av problemer som løses med en polynomisk grense på minnet kalles $PSPACE$
- $TIME(f(n)) \subseteq SPACE(\frac{f(n)}{logf(n)})$

## 3.2 Symmetrisk og asymmetriske chifre

Betegnelsene "symmetriske" og "asymmetriske" chifre stammer fra hva slags nøkler som brukes. I symmetriske chifre anvendes én nøkkel til kryptering og dekrypteringsoperasjonene. Dette innebærer at sikker kommunikasjon innebærer at man må utveksle nøkler over en sikker kanal før kommunikasjonen skal finne sted. Ulempen med dette er at man trenger et nøkkelpar for hver part som skal kommunisere. Dette innebærer en stor utfordring med administrasjon av nøklene. Det er også viktig at nøkkelen utveksles over en sikker kanal slik at den ikke kan fanges opp. Asymmetriske chifre løser dette problemet ved at en av nøklene er offentlig tilgjengelig. Den kan deles ut til alle som skal sende en melding til en part. Det er ikke mulig å lese meldingen uten å benytte seg av den korresponderende hemmelige nøkkelen.

## 3.3 Enveisfunksjoner

Alle krypteringssystem baserer seg på "enveisfunksjoner". Her er den generelle definisjonen:

[2] $f^k(n) = f(f^{k-1}(n))$

**Enveisfunksjon:** En funksjon $f : X \rightarrow Y$ kalles en enveisfunksjon dersom $f(x)$ er lett å beregne for alle $x \in X$ mens det for de fleste elementene $y \in Im(f)$ er vanskelig å finne $x \in X$ slik at $f(x) = y$.

Et asymmetrisk chiffer benytter seg av en variant av denne funksjonen kalt "trapdoor one-way function".

**"Fallem enveisfunksjon"** : Gitt en enveisfunksjon (som definert ovenfor): $f$. $f$ er en "fallem enveisfunskjon" dersom man gitt $y \in Im(f)$ og ekstrainformasjon (i form av en nøkkel) kan beregne $x$ slik at $f(x) = y$

### 3.3.1 Kompleksitet og enveisfunsjoner

Sammenhengen mellom enveisfunksjoner (se 3.3) og kompleksitet er beskrevet i [Pap95, s283-285]: Kort fortalt:

**Definisjon:** En ikke-deterministisk en-en-tydig Turingmaskin ($UTM$) har følgende egenskap: For input $x$ har den maksimalt *én* aksepterende beregning.

**Definisjon:** Klassen $UP$ er klassen av språkene akseptert av en $UTM$.
Det er innlysende av denne definisjonen at $P \subseteq UP \subseteq NP$ (for bevis se [Pap95]).

**Teorem:** $UP = P$ dersom det ikke eksisterer en en-veis funksjon.

Av dette ser vi at dersom vi beviser at det *ikke* eksisterer en-veis funksjoner, så har vi *ikke* bevist at $P = NP$.

Klassen $UP$ inneholder alle enveisfunksjoner. Eksempel: Vi bruker det vanlige scenarioet hvor Alice og Bob ønsker å sende en hemmelig melding til hverandre. De bruker to algoritmer: $E$ og $D$ for hhv. kryptering og dekryptering. Meldingen de ønsker å sende, $x \in \Sigma^*, \Sigma = \{0, 1\}$, nøkkelparet $e, d$. Kryptering er gitt ved $y = E(e, x)$ og dekryptering er gitt ved $x = D(d, y)$. $D$ og $E$ er inverse funksjoner som kun fungerer med et passende nøkkelpar $d, e$. Det skal ikke være mulig å dedusere $x$ fra $y$ uten å kjenne $d$.

M.a.o.: Klassen $UP$ forteller oss at det underliggende problemet til en enveisfunksjon *ikke* skal være løselig av en algoritme som ligger i $FP$. En $UTM$ krever også at det kun skal finnes én algoritme som kan brukes til en beregning. M.a.o. skal det f.eks. ikke gå an å løse faktoriseringsproblemet (definert nedenfor) på andre enn én måte. Dette er vanskelig å bevise da det finnes utallige tilnærmingsmåter til et problem. Eksempelvis finnes det probabilistiske algoritmer som kan løse problemet i polynomisk tid for en del problemer. I praksis betyr dette at man må ta høyde for probabilistiske algoritmer og andre angrepsmetoder når en skal avgjøre hvor sikker et krypteringssystem er. Det underliggende problemet gir ikke nødvendigvis en garanti for sikkerheten. Som eksempel kan nevnes "Pretty Good Privacy" (PGP), hvor det ble anvendt en dårlig primtallsgenerator. Dette innebar at man kunne innskrenke søketreet for angrepsalgoritmene betraktelig. Her var det m.a.o. mulig å angripe systemet uten at man hadde noen effektive algoritmer for det underliggende problemet.

## 3.4 De vanligste asymmetriske chifrene

Siden 1977, da det første asymmetriske chifret ble oppfunnet, har det kommet en rekke nye varianter. Disse systemene baserer seg på problemer hvor man ikke kjenner til en polynomisk tids-algoritme for å knekke krypteringen. Ved konstruksjon av et chiffer går man alltid ut i fra at motstanderne vet hvilket krypteringssystem som blir brukt. Dette kalles "Kerckhoffs" prinsipp. Krypteringssystemet må derfor være sikkert uavhengig av hva en angriper måtte ha av kjennskap til krypteringssystemet. Det er vanlig å skille mellom forskjellig type angrip på et krypteringssystem:

**Chiphertext-only:** Angriperen innehar kun kjennskap til den krypterte teksten.

**Known plaintext:** Angriperen har både kjennskap til den krypterte og ukrypterte teksten.

**Chosen plaintext:** Angriperen har tilgang til krypteringsmaskineriet og kan velge en ukryptert tekst for å konstruere en kryptert variant.

**Chosen ciphertext:** Angriperen har tilgang til dekrypteringsmaskineriet og kan velge en kryptert tekst for å konstruere den korresponderende ukrypterte teksten.

Målet med disse angripene er å avgjøre hvilken nøkkel som blir brukt. "Chosen ciphertext"-angrep er relevant for asymmetriske chifre hvor én av nøklene er kjent. Det må derfor tas høyde for dette angrepet når man konstruerer et asymmetrisk krypteringssystem.

## Faktoriseringsproblemet

**Instans:** $x \in \mathbb{Z}^+$

**Problem:** Finn $m, n \in \mathbb{Z}^+$ slik at $x = mn$

Faktoriserings-problemet brukes i bl.a. det første, og kanskje mest kjente asymmetriske krypteringssystemet:

## RSA kryptering[RSA77]

RSA[3] krypteringssystemet var det første[4] asymmetriske krypteringssystemet som ble tatt i bruk.

La $n = pq$ hvor $p$ og $q$ er prim. La så $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$ og definer

$$\mathcal{K} = (n, p, q, a, b)$$

hvor $n = pq, p, q$ er prim, $ab \equiv 1 \mod \phi(n), \phi(n) = (p-1)(q-1)$

For en gitt $\mathcal{K}$, definer

$$e_{\mathcal{K}}(x) = x^b \mod n$$

og

$$d_{\mathcal{K}}(y) = y^a \mod n$$

[3]RSA står for Rivest, Shamir og Adleman, etter navnene på opphavsmennene.

[4]J.H. Ellis lanserte idéen i 1970 [Ell70]. C. Cocks beskrev deretter en variant av RSA i 1973 [Coc73]. Begge to jobbet for den britiske organisasjonen CESG, som valgte å hemmeligholde resultatet. Diffie og Hellman gjorde sin oppdagelse uavhengig av dette.

hvor $x, y \in \mathbb{Z}_n$. Den offentlige delen av nøkkelen består av $(n, b)$ mens den private delen av nøkkelen består av $(p, q, a)$.

Sikkerheten i RSA systemet er basert på antagelsen om at krypteringsfunksjonen $e_{\mathcal{K}} = x^b \mod n$ er en en-veis funksjon. M.a.o.: faktoriseringen $n = pq$ (dvs. beregningen av p og q) skal være vanskelig.

## Diskrete logaritmer

**Instans:** $I = (p, \alpha, \beta)$ hvor $p$ er prim, $\alpha \in \mathbb{Z}_p$ er et primitivt element og $\beta \in \mathbb{Z}_p^*$.

**Problem:** Finn $a \in \mathbb{Z}, 0 \leqslant a \leqslant p-2$, s.a. $\alpha^a \equiv \beta \pmod{p}$.

Dette problemet kalles for det diskrete logaritme problemet i $\mathbb{Z}$. Dette problemet har vært gjenstand for inngående studier. Problemet regnes for å være vanskelig dersom $p$ velges riktig. Det finnes ikke en kjent polynomiell-tids algoritme, men systemet kan likevel angripes med "bruteforce". For å hindre kjente angrep bør $p$ bestå av minst $512bits$ (enkelte mener $1024$ bits er nødvendig [Sti95]). $p-1$ bør dessuten bestå av minst en stor primfaktor.

## ElGamal krypteringssystem[Sti95]

ElGamal er et av systemene som baserer sin sikkerhet på diskrete logaritmer. Det er også et av krypteringssystemene som er blitt implementert over en elliptisk kurve.

La $p$ være et odde prim slik at det diskrete logariteproblemet i $\mathbb{Z}_p$ er vanskelig. La $a \in \mathbb{Z}_p^*$ være et primitivt element. La $P =$
$Z_p^*, A = \mathbb{Z}_p \times \mathbb{Z}_{p-1}$ og definer

$$K0\{(p, \alpha, a, \beta) : \beta \equiv \alpha^a (\mod p)\}.$$

Verdiene $p, \alpha, \beta$ er offentlige og $a$ er hemmelig. For et tilfeldig tall $k \in \mathbb{Z}_{p-1}^*$ definer

$$sig_K(x, k) = (\gamma, \delta)$$

hvor

$$\gamma = \alpha^k \mod p$$

og

$$\delta = (x - a\gamma)k^{-1} \mod (p-1)$$

Tilsvarende definerer vi

$$ver_K(x, \gamma, \delta) = OK \Leftrightarrow \beta^\gamma \gamma^\delta \equiv \alpha^x (\mod p)$$

ElGamal krypteringssystemet kan implementeres i enhver syklisk gruppe hvor det diskret logaritmeproblemet er ikke-beregnbar i polynomisk tid [Sti95, side 164]. Ettersom en elliptisk kurve tilfredsstiller dette kravet er det laget flere implementasjonsforslag som benytter seg av ElGamal (bl.a. [Men93]).

### 3.4.1 Andre problemer

**Instans:** La $U$ være en endelig mengde. La $B, K \in \mathbb{Z}^+$. Definer to funksjoner $s(u) \in \mathbb{Z}^+$ som gir oss størrelsen av $u$ og $v(u) \in \mathbb{Z}^+$ som gir oss verdien av u.

**Problem:** Finnes det en undermengde $U' \subseteq U$ slik at:
$\sum_{u \in U'} s(u) \leqslant B$ og $\sum_{u \in U'} \geqslant K$?

Dette problemet forblir $NP$-komplett dersom $s(u) = v(u) \forall u \in U$. (Det kan løses av en pseudo-polynomisk tidsalgoritme dersom $s(u) \neq v(u)$.)[GJ79, s. 246]

Sikkerheten i et kryptosystem er ikke avhengig av de ovennevnte kompleksitetsklassene. Alle forsøk på å bruke et NP-komplett problem til dette har vært mislykkede. Eksempler på dette er "Chor-Rivest" og "Merkle-Hellman Knapsack".[Sti95]

Merkle-Hellman knapsack ble knukket av Shamir tidlig på 1980-tallet [Sti95]. "Chor-Rivest" ble lenge ansett for å være sikkert. I den senere tid har dette systemet også blitt knukket av Vaudenai [Vau98] for utvalgte

parametre. Vaudenai mener angrepet kan utvides til å gjelde alle parametre. Fra dette må man konkludere med at et $NP$-komplett problem ikke nødvendigvis kvalifiserer til et sikkert krypteringssystem.

## 3.5 Digitale signaturer

En *hash-funksjon*, $h(x)$, er en enveisfunksjon som konverterer en melding av vilkårlig lengde til en s.k. hashverdi som har en fast lengde.

En *signert melding* er et tuppel

$$(x, y) \text{ hvor } y = sig_k(h(x))$$

og

En signatur er autentisk dersom *verifikasjonsalgoritmen* gir oss

$$h(x) = ver_k(y)$$

hvor $y = sig_k(h(x))$

$x$ er en melding av vilkårlig størrelse

$h(x) = z$ $h$: hashfunksjon, $z$: hashverdi

$sig_k(z) = y$ signering med nøkkel $k$

$ver_k(y) = z$ verifikasjon av signatur, $z$: hashverdi

For at en hashfunksjon skal kunne brukes i kryptografisk øyemed må den være *kollisjonsfri*. Dvs. det skal ikke være mulig å beregne en ny melding $x'$ ut i fra hashtabellen til en melding $x$. ($x' \neq x \; s.a. \; h(x') = h(x)$). Dette for å hindre at man kan forfalske innholdet i meldingen[Sti95].

De mest vanlige signatursystemene er RSA og ElGamal signaturer:

## RSA signaturer

Gitt $\mathcal{K} = \{(n, p, q, a, b) : n = pq, ab \equiv 1(\text{mod } \phi(n)\}$ hvor $p, q$ er odde primtall og $\phi(n) = (p-1)(q-1)$.

For $\mathcal{K} = (n, p, q, a, b)$ definer

$$sig_{\mathcal{K}}(x) = x^a \mod n$$

og

$$ver_{\mathcal{K}}(x, y) = sant \Leftrightarrow x \equiv y^b (\text{ mod } n)$$

hvor $(x, y \in \mathbb{Z}_n)$.

### ElGamal signaturer[Sti95, side 205]

La $p$ være et primtall slik at det diskrete logaritmeproblemet i $\mathbb{Z}_p$ er uberegnbart i polynomisk tid. La så $\alpha \in \mathbb{Z}_p^*$ være et primitivt element (dvs. $gcd(\alpha, p) = 1$). La $\mathcal{P} = \mathbb{Z}_p^*, \mathcal{A} = \mathbb{Z}_p^* \times \mathbb{Z}_{p-1}$, og definer

$$\mathcal{K} = \{(p, \alpha, a, \beta) : \beta \equiv \alpha^a \text{ ( mod } p)\}.$$

Verdiene $p, \alpha$ og $\beta$ er offentlige, og $a$ er hemmelig. For $K = (p, \alpha, a, \beta)$ og et hemmelig tilfeldig nummer $k \in \mathbb{Z}_{p-1}^*$ definer

$$sig_K(x, k) = (\gamma, \delta),$$

hvor

$$\gamma = \alpha^k \mod p$$

og

$$\delta = (x - a\gamma)k^{-1} \mod (p-1).$$

For $x, \gamma \in \mathbb{Z}_p^*$ og $\delta \in \mathbb{Z}_{p-1}$, definer

$$ver_k(x, \gamma, \delta) = sant \Leftrightarrow \beta^\gamma \gamma^\delta \equiv \alpha^x \text{ ( mod } p).$$

### 3.5.1 DSS[Sti95]

Digital signatur standard (DSS) er en modifikasjon av ElGamal signaturer. DSS bruker en undergruppe av $\mathbb{Z}_p^*$: La $p$ være et 1024-bit primtall s.a. det diskrete logaritmeproblemet i $\mathbb{Z}_p$ ikke er løselig i polynomisk tid, og la $q$ være et 160-bits primtall s.a. $q|p-1$. La $\alpha \in \mathbb{Z}_p^*$,$\alpha^q = 1(\mathrm{mod}\ p)$. Nå vil både $\gamma$ og $\beta$ være $q$-te røtter av 1, og alle eksponensieringer av $\alpha, \beta$og$\gamma$ kan reduseres modulo $q$ uten å påvirke verifikasjonen av signaturen. Definer $K = \{(p, q, \alpha, a, \beta) : \beta \equiv \alpha^a(\mathrm{mod}\ p)\}$. Verdiene $p, q, \alpha$ og $\beta$ er offentlig mens $a$ er den hemmelige verdien. Gitt $K = (p, q, \alpha, a, \beta)$ og et hemmelig tall $k, 1 \leq k \leq q-1$, definer $sig_K(x, k) = (\gamma, \delta)$ hvor $\delta = (x + a\gamma)k^{-1} \mod q$ og $\gamma = (\alpha^k \mod p) \mod q$

Verifikasjon utføres som følger: Gitt $K, (\gamma, \delta)$, $ver_k(x, \gamma, delta) =$ "$ok$" $\Leftrightarrow$ $(\alpha^{e_1}\beta^{e_2} \mod p) \mod q = \gamma$ hvor $e_1 = x\delta^{-1} \mod q$ og $e_2 = \gamma\delta^{-1} \mod q$

## 3.6 Sikkerheten i asymmetriske chifre

Det finnes pr. i dag en rekke algoritmer for å faktorisere tall (jfr. faktoriseringsproblemet). Den mest effektive er Pollards "Number Field Sieve"-metode[LV00] som finnes i en rekke varianter. Felles for disse er at de er subeksponensielle. Dette stiller dog krav til nøkkelstørrelsen. Det mest utbredte systemet pr. tiden er RSA (hvor patentet nettopp har utløpt). Problemet med dette systemet er at algoritmene for å faktorisere er relativt effektive, og dersom man tar med i beregningen at prosessorkraft har samme utvikling i fremtiden som den har hatt til nå (dvs. at hastigheten fordobles hver 18. måned[PH94]), vil nøkkelstørrelsen etter hvert vokse.

I 1985 foreslo Victor Miller og Neal Koblitz at man kunne bruke et system som baserer sikkerheten på det diskrete logaritmeproblemet på en elliptisk kurve, se [Men93]. Dette er en variant av det diskrete logaritmeproblemet som blir nærmere diskutert i kapittel 5.

Det har vist seg at selv når man har tatt hensyn til de mest effektive algoritmene (som p.t. er Pollard-$\rho$-metoden [vOW94, LV00]) vil man klare

seg med moderate nøkkelstørrelser. Et krypteringssystem basert på elliptiske kurver med nøkkelstørrelse på omlag 160 bits vil ha den samme sikkerheten som ved en 1024 bits RSA-nøkkel [LV00, IEE01]. Dette innebærer at man kan få plass til langt flere nøkler på hvert smartkort. Med tanke på alle anvendelsene man har tenkt seg for smartkort (se innledning) er dette en stor fordel. Det er sogar hovedmotivasjonen bak å bruke elliptiske kurver.

# 4

# Elliptiske kurver (EK)

I 1985 foreslo Koblitz og Miller uavhengig av hverandre at elliptiske kurver kunne brukes til kryptering[Kob87, Mil86]. Forslaget gikk ut på å implementere eksisterende asymmetriske krypteringssystemer v.h.a. elliptiske kurver.

## 4.1 Bakgrunnsteori

### 4.1.1 Kropper av odde karakteristikk

**Definisjon:** La $p$ være et odde primtall. Den endelige kroppen $\mathbb{F}_p$, kalt en kropp av odde karakteristikk, består i at mengden $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ med følgende operatorer definert:

Addisjon: $a, b \in \mathbb{F}_p, a + b = r$ hvor $r$ er addisjon modulo $p$.

Multiplikasjon: $a, b \in \mathbb{F}_p, a \circ b = s$, hvor $s$ er rest av divisjonen av $a * b$ med $p$. Dette kalles multiplikasjon modulo $p$.

### 4.1.2 Kropper av jevn karakteristikk

**Definisjon:** Den endelige kroppen $\mathbb{F}_{2^m}$ er en utvidelseskropp til primkroppen $\mathbb{F}_2$. Den har derfor karakteristikk 2 og kalles den binære kroppen av grad m.

Alle elementer $a \in \mathbb{F}_{2^m}$ er da polynomer av grad m og kan skrives på formen

$$a = \sum_{i=0}^{m-1} a_i x^i, a_i \in \{0, 1\} \tag{4.1}$$

hvor $a_i$ er koeffisienter. Mengden $K = \{x_0, x_1, \ldots, x_{m-1}\} \in \mathbb{F}_{2^m}$ kalles basisen til $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. Et hvert element i $\mathbb{F}_{2^m}$ kan da representeres som en binær vektor $(a_0, a_1, \ldots, a_{m-1})$. Det finnes to hovedtyper for å

representere disse elementene. Disse vil bli diskutert i de påfølgende avsnittene.

## 4.2 Basisrepresentasjoner

### 4.2.1 Polynomiell basis

En polynomiell basis er definert ved et reduksjonspolynom: La $f(x) = \sum_{i=0}^{m} c_i x^i, c_i \in \{0,1\}, c_m \neq 0$ være et irredusibelt polynom av grad m over $\mathbb{F}_2$. $f(x)$ kalles et reduksjonspolynom. For hvert reduksjonspolynom eksisterer en polynomiell basis. Hvert element i $\mathbb{F}_{2^m}$ korresponderer til en binært polynom av grad mindre enn $m$. Gitt $a, b \in \mathbb{F}_{2^m}$: Addisjon defineres som følger: $a + b = c = (c_{m-1} \dots c_1 c_0)$ hvor $c_i = (a_i \oplus b_i)$.[1] Multiplikasjon defineres som følger: $a * b = c = (c_{m-1} \dots c_1 c_0)$, hvor $c(x) = \sum_{i=0}^{m-1} c_i x^i$ er resten etter denne divisjonen: $\frac{(\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i)}{f(x)}$.

For å velge et reduksjonspolynom er det vanlig å bruke følgende metode: Dersom et irredusibelt trinomial $x^m + x^k + 1$ eksisterer over $\mathbb{F}_2$ velger man reduksjonspolynomet $f(x)$ til å være det trinomialet hvor $x^k$ har lavest grad. Dersom det ikke eksisterer et irredusibelt trinomial, velger man istedenfor et pentanomial $x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$ med minst mulig verdi for $k_1$. $k_2$ og $k_3$ velges deretter (med minste mulige verdi gitt $k_1$).

#### 4.2.1.1 Aritmetikk ved bruk av polynomiell basis

**Addisjon:** Addisjon over $\mathbb{F}_{2^m}$ ved bruk av polynomiell basis fungerer på bit-nivå modulo2:[BSS99]

**Algoritme 1** Addisjonsalgoritme for $\mathbb{F}_{2^m}$

INPUT: $a = (a_{m-1} \dots a_1 a_0) \in \mathbb{F}_{2^m}$ og $b = (b_{m-1} \dots b_1 b_0) \in \mathbb{F}_{2^m}$
OUTPUT: $c = a + b = (c_{m-1} \dots c_1 c_0) \in \mathbb{F}_{2^m}$
**for** $j = 0$ to $m - 1$ **do**
  $c_j \leftarrow a_j \oplus b_j$
**end for**
Return $c$

[1] $\oplus$ betegner den boolske funksjonen xor: $a \oplus b = a + b \mod 2$

**Modulær reduksjon:** Multiplikasjon over $\mathbb{F}_{2^m}$ ved bruk av polynomiell basis må reduseres modulo et irredusibelt polynom av grad $m$ (se 4.2.1). Reduksjonen er som tidligere nevnt mest effektiv dersom reduksjonspolynomet $f(x)$ er et trinomial eller pentanomial. Man kan anvende følgende algoritme for denne operasjonen.

**Algoritme 2** Modulær reduksjon,[BSS99] side 20

INPUT: $a = (a_{m-2} \dots a_{m-1} \dots a_1 a_0)$ og $f = (f_m f_{m-1} \dots f_1 f_0)$
OUTPUT: $c = a \mod f$
**for** $i = 2m-2$ to $m$ **do**
  **for** $j = 0$ to $m-1$ **do**
    **if** $f_j \neq 0$ **then**
      $a_{i-m+j} \leftarrow a_{i-m+j} \oplus a_j$
    **end if**
  **end for**
**end for**
$c \leftarrow (a_{m-1} \dots a_1 a_0)$
Return($c$)

Vi ser her at denne algoritmen har kompleksitet $\mathcal{O}(m^2)$. Den er m.a.o. tung å implementere i software.

**Kvadrering:** Gitt

$$a(x)^2 = \left( \sum_{i=0}^{m-1} a_i x^i \right)^2 = \sum_{i=0}^{m-1} a_i^2 x^{2i} = \sum_{i=0}^{m-1} a_i x^{2i}.$$

Vi kan da bruke følgende algoritme:

**Algoritme 3** Kvadrering, $a = a^2$

INPUT: $a = (a_{m-1} \dots a_1 a_0)$, $f = (f_m f_{m-1} \dots f_1 f_0)$
OUTPUT: $c = a^2 \mod f$
$t \leftarrow \sum_{i=0}^{m-1} a_i^2 x^{2i}$
$c \leftarrow t \mod f$ /* bruk reduksjonsalgorimen ovenfor */
Return($c$)

Denne algoritmen anvender seg av algoritmen for modulær reduksjon definert ovenfor. Den har derfor orden $\mathcal{O}(m^2)$.

**Multiplikasjon:** For å utføre multiplikasjon trenger vi en aritmetisk-skift mot venstre operasjonen definert som følger: Gitt $a \in \mathbb{F}_{2^m}$. Aritmetisk-skift $xa(x) \mod f(x)$ er gitt ved:

$$xa(x) \mod f(x) = \begin{cases} \sum_{j=1}^{m-1} a_{j-1}x^j & \text{for } a_{m-1} = 0 \\ \sum_{j=1}^{m-1}(a_{j-1} + f_j)x^j + f_0 & \text{for } a_{m-1} \neq 0 \end{cases}$$

**Algoritme 4** Multiplikasjon over $\mathbb{F}_{2^m}$

INPUT: $a, b \in \mathbb{F}_{2^m}, f = (f_m f_{m-1} \dots f_1 f_0)$
OUTPUT: $c = ab \mod f$
$c(x) \leftarrow 0$
**for** $j = (m-1)$ to 0 **do**
  $c(x) \leftarrow xc(x) \mod f(x)$
  **if** $a_j \neq 0$ **then**
    $c(x) \leftarrow c(x) + b(x)$
  **end if**
**end for**
Return($c$)

Denne algoritmen anvender også modulær reduksjon. Ettersom den gjør dette i en løkke har den orden $\mathcal{O}(m^3)$.

### 4.2.2 Normalbasis

En normalbasis av $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$ har formen $\{\beta, \beta^2, \dots, \beta^{2^{m-1}}\}, \beta \in \mathbb{F}_{2^m}$. Det er bevist at en slik basis alltid eksisterer[BSS99]. Alle elementer $a \in \mathbb{F}_{2^m}$ kan derfor bli skrevet som $a = \sum_{i=0}^{m-1} a_i \beta^{2^i}, a_i \in \{0, 1\}$. Elementet $a$ representeres vanligvis som bitstrengen $(a_0 a_1 \dots a_{m-1})$ (av lengde $m$). Representasjon av elementene i en normalbasis innebærer at kvadrering av elementer kan implementeres ved syklisk skifting av bitene i strengen $a$. Multiplikasjon av elementer er dog fremdeles ganske komplisert.

Gitt $a, b \in \mathbb{F}_{2^m}$, $a = \sum_{i=0}^{m-1} a_i \beta^{2^i}, b = \sum_{j=0}^{m-1} b_j \beta^{2^j}$. Vi ønsker å beregne

$$c = ab = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{2^i} \beta^{2^j}.$$

Vi kan også skrive $c$ som $c = \sum_{k=0}^{m-1} c_k \beta^{2^k}$. Dette innebærer at $\beta^{2^i}\beta^{2^j} = \sum_{k=0}^{m-1} \lambda_{ijk}\beta^{2^k}$. Vi har derfor $c_k = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} a_i b_j \lambda_{ijk}$. Dette kan skrives om til

$$c_k = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0}$$

(se [Ros99]).

Koeffisienten $\lambda_{ijk}$ kalles for multiplikasjonsmatrisen. En "optimal" normalbasis (ONB) har færrest mulig ikke-null-elementer i matrisen. For kropper av jevn karakteristikk vil multiplikasjonsmatrisen ha minimum $2m-1$ ikke-null-elementer ([Ros99]). Det finnes to typer optimale normalbasiser over $\mathbb{F}_{2^m}$: Type 1 og type 2. Hovedforskjellen mellom dem er hvordan multiplikasjonsmatrisen er generert. ONB type 1 er den enkleste å implementere (se [Ros99]), og jeg har derfor valgt å implementere den i denne oppgaven.

For å bruke en ONB må man først sjekke om kroppen det eksisterer en ONB: Det eksisterer en type 1 ONB dersom $m+1$ er prim og 2 er et primitivt element i $\mathbb{Z}_{m+1}$.

Det eksisterer en type 2 ONB dersom $2m+1$ er prim og enten $2$ er et primitivt element i $\mathbb{Z}_{2m+1}$ *eller* $2m+1 \equiv 3 \mod 4$ og $2$ genererer alle kvadratiske rester[2] i $\mathbb{Z}_{2m+1}$.

#### 4.2.2.1 Aritmetikk ved bruk av normalbasis

Generering av multiplikasjonsmatrisen er beskrevet i [IEE01, s.109] og [Ros99]. Algoritme 5 som jeg anvender er hentet fra sistnevnte. Kort forklart: Som vi husker fra avsnittet ovenfor har vi $\beta^{2^i}\beta^{2^j} = \sum_{k=0}^{m-1} \lambda_{ijk}\beta^{2^k}$. Vi må m.a.o. løse ligningene $\beta^{2^i}\beta^{2^j} = 1$ og $\beta^{2^i}\beta^{2^j} = 0$. Dette gjøres ved å løse $2^i + 2^j = 1 \mod m+1$ og $2^i + 2^j = 0 \mod m+1$. Se algoritme 5.

[2] Se avsnitt 4.5.1

**Algoritme 5** Algoritme for å generere multiplikasjonsmatrisen $\lambda_{i,j}$

INPUT $m$ (graden til binærkroppen $\mathbb{F}_{2^m}$)
OUTPUT Multiplikasjonsmatrise $\lambda_{i,j}$
**for** $i = 0$ to $m - 1$ **do**
  $log2_i \leftarrow -1$
**end for**
$twoexp \leftarrow 1$
**for** $i = 0$ to $m - 1$ **do**
  $log2_{twoexp} \leftarrow i$ /* Løp igjennom alle bits i m, venstreskift er som vi husker fra avsnittet ovenfor kvadrering av elementet */
  $twoexp \leftarrow leftshift(twoexp) \mod m + 1$
**end for**
/* $log2_i$ er nå en logaritmetabell */
$n \leftarrow m/2$ /* matrisen er symmetrisk, vi behøver bare beregne halvparten */
$\lambda_{0,0} \leftarrow n$
**for** $i = 1$ to $m - 1$ **do**
  /* Hvert element er 1 større enn det foregående */
  $\lambda_{0,i} \leftarrow (\lambda_{0,i-1} + 1) \mod m$
**end for**
$\lambda_{1,0} \leftarrow -1$ /* aldri i bruk */
$\lambda_{1,1} \leftarrow n$
$\lambda_{1,n} \leftarrow 1$
**for** $i = 2$ to $n$ **do**
  /* $2^i + 2^j \equiv 1 \mod m + 1$ */
  $idx \leftarrow log2_i$
  $log \leftarrow log2_{m-i+2}$
  $\lambda_{1,idx} \leftarrow log$
  $\lambda_{1,log} \leftarrow idx$
**end for**
$\lambda_{1,log2_{n+1}} \leftarrow log2_{n+1}$
Return($\lambda$)

Denne algoritmen har en orden $\mathcal{O}(m)$ og er relativt enkel å beregne.

Etter vi har generert multiplikasjonsmatrisen er multiplikasjon en relativt enkel operasjon. Se algoritme 6. Denne algoritmen er også hentet fra [Ros99].

**Algoritme 6** Multiplikasjonsalgoritme for type 1 ONB

INPUT: Multiplikasjonsmatrise $\lambda_{i,j}$, $a, b \in \mathbb{F}_{2^m}$
OUTPUT: $c = a * b$
$atmp_0 \leftarrow a$ /* $atmp$ er en vektor av lengde $m$ med elementer fra $\mathbb{F}_{2^m}$ */
$btmp \leftarrow RotateRight(b)$
**for** $i = 1$ to $m - 1$ **do**
  /* genererer alle roterte versjoner av $a$ */
  $atmp_i \leftarrow RotateRight(atmp_{i-1})$
**end for**
**for** $i = 1$ to $m - 1$ **do**
  /* $\times$ står for den binære AND operatoren */
  $c_i \leftarrow btmp \times (atmp_{\lambda_{0,i}} \oplus atmp_{\lambda_{1,i}})$
**end for**
Return($c$)

Denne algoritmen har orden $\mathcal{O}(m)$ men bruker mye plass: SPACE($m^2$).

### 4.2.3 Affine og projektive koordinater

I tilfeller hvor inversjoner i den underliggende kroppen er mye tyngre enn multiplikasjoner ([BSS99]) er det mulig å implementere projektive koordinater. Vi har da koordinatene $(x, y, z)$ som tilsvarer de affine koordinatene $(x/z^2, y/z^3)$ for $z \neq 0$. Ettersom inversjon ikke er en spesielt tung operasjon i en normalbasis har jeg anvendt affine koordinater i implementasjonen i denne oppgaven.

## 4.3 Definisjon av en elliptisk kurve

Bakgrunnsteorien for elliptiske kurver er noe komplisert og går langt utenfor hva denne oppgaven skal behandle. Jeg vil likevel gi en kort innføring i begreper som er av nytte for denne oppgaven. De som er interessert i å gå dypere ned i teorien henvises til annen litteratur.

En elliptisk kurve $E(K)$ over kroppen $K$ er definert ved parametrene $a_i \in K$ og består av løsningsmengden[3]

$$E(K) = \{(x, y) \in K \times K\} \cup \{\mathcal{O}_\infty\}$$

[3]Dvs. en mengde av punkter $P = (x, y)$

til ligningen

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{4.2}$$

$\mathcal{O}_\infty$ betegner "punktet i det uendelig fjerne". Vi krever også at kurven ikke skal ha singulære punkter. Dette betyr at den deriverte skal være veldefinert i alle punktene.

Ligning 4.2 kalles den "lange Weierstrass" ligningen til en elliptisk kurve.

Gitt følgende konstanter (hvor $a_i$ fremkommer i ligningen ovenfor):

$$\begin{cases} b_2 = a_1^2 + 4a_2, b_4 = a_1a_3 + 2a_4, b_6 = a_3^2 + 4a_6, \\ b_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2, \\ c_4 = b_2^2 - 24b_4, c_6 = -b_2^3 + 36b_2b_4 - 216b_6 \end{cases}$$

Diskriminanten er da definert ved $\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6$[BSS99]. Kurven er ikke-singulær dersom $\Delta \neq 0$. Det er også vanlig å definere en $j - invariant$ definert ved $j(E) = c_4^3/\Delta$ som naturligvis kun er gyldig for $\Delta \neq 0$.

### 4.3.1 Kurver i kropper av karakteristikk $p > 3$

Anta at $K = \mathbb{F}_q$ hvor $q = p^n$ for et primtall $p, p > 3$. Dersom vi modifiserer variablene i Weierstrassligningen ved å bruke konstanten $b_2$ kan vi forenkle ligningen.

La

$$x = x' - \frac{b_2}{12}$$
$$y = y' - \frac{a_1}{2}(x' - \frac{b_2}{12}) - \frac{a_3}{2}$$

Denne omskiftningen av variable gir oss den "korte" Weierstrassligningen.

$$y^2 = x^3 + ax + b \tag{4.3}$$

Ved å bruke den korte Weierstrassligningen kan vi uttrykke diskriminanten ved $\Delta = -16(4a^3 + 27b^2)$ og $j$-invarianten til $j(E) = -1728(4a)^3/\Delta$.

Ettersom vi ønsker at kurven skal være ikke-singulær[4] må m.a.o. følgende være tilfredstilt: $4a + 27b^2 \equiv 0 \mod p$

### 4.3.2 Kurver i kropper av jevn karakteristikk

Anta at $K = \mathbb{F}_q$ hvor $q = 2^m, m \geq 1$. Vi får nå $j$-invarianten $j(E) = a_1^{12}/\Delta$. Dersom $j(E) = 0$ (dvs. $a_1 = 0$) har vi en supersingulær kurve. Dette ønsker vi å unngå (se kapitlet om disrkete logaritmer for en forklaring). Vi krever derfor at $j(E) \neq 0$. Dette gir oss følgende ligning:

$$\mathbb{F}_{2^m} : y^2 + xy = x^3 + ax^2 + b \tag{4.4}$$

### 4.3.3 Elliptiske kurver over $\mathbb{F}_p$

Det kan defineres en addisjonsoperasjon over mengden $E(\mathbb{F}_p)$. Det har blitt vist at $(E, +)$ kan danne en abelsk gruppe med $\mathcal{O}_\infty$ som identitetselement dersom addisjon defineres som følger:

1. $\forall P \in E(\mathbb{F}_p)$
   $P + \mathcal{O}_\infty = \mathcal{O}_\infty + P = P$

2. $P = (x, y) \in E(\mathbb{F}_p)$
   $-P = (x, -y)$

3. Gitt $P(x, y) \in E(\mathbb{F}_p)$
   $P - P = (x, y) + (x, -y) = \mathcal{O}_\infty$

4. Gitt $P = (x_1, y_1) \in E(\mathbb{F}_p), Q = (x_2, y_2) \in E(\mathbb{F}_p), P \neq \pm Q$.
   $P + Q = (x_3, y_3)$ hvor $\begin{cases} x_3 = \mu^2 - x_1 - x_2 \\ y_3 = \mu(x_1 - x_3) - y_1 \\ \mu = \frac{y_2 - y_1}{x_2 - x_1} \end{cases}$

[4]For å kunne utføre aritmetikk på punkter, i dette tilfellet $P + P$, må det finnes en tangent i hvert punkt. (Se avsnitt 4.3.3)

5. Gitt $P = (x_1, y_1) \in E(\mathbb{F}_p)$

$$P + P = 2P = (x_3, y_3) \text{ hvor } \begin{cases} x_3 = \mu^2 - 2x_1 \\ x_3 = \mu(x_1 - x_3) - y_1 \\ \mu = \frac{3x_1^2 + a}{2y_1} \end{cases}$$

Addisjon av punktene $P$ og $Q$ gjøres ved å trekke en linje igjennom begge punktene. Der linjen skjærer kurven får vi et nytt punkt $R$. Dersom vi speiler dette punktet om $x$-aksen får vi et nytt punkt. Dette er definert ti lå være $P+Q$. Dersom $P = Q$ bruker vi i stedenfor en tangent, og beregner $P + P$ på tilsvarende måte. Se figur nedenfor.

**Algoritme 7** Addisjonsalgoritme for punkter på $E(\mathbb{F}_p)$

INPUT: En elliptisk kurve $E(\mathbb{F}_p)$ med parametre $a, b \in \mathbb{F}_p$ og punktene $P_1 = (x_1, y_1)$ og $P_2 = (x_2, y_2)$.
OUTPUT: $Q = P_1 + P_2$
**if** $P_1 = \mathcal{O}_\infty$ **then**
  Return($Q \leftarrow P_2$)
**end if**
**if** $P_2 = \mathcal{O}_\infty$ **then**
  Return($Q \leftarrow P_1$)
**end if**
**if** $x_1 = x_2$ **then**
  **if** $y_1 = y_2$ **then**
    $\mu \leftarrow (4x_1^2 + a)/(2y_1) \mod p$
  **else**
    Return($Q \leftarrow \mathcal{O}_\infty$)
  **end if**
**else**
  $\mu \leftarrow (y_2 - y_1)/(x_2 - x_1) \mod p$
**end if**
$x_3 \leftarrow \mu^2 - x_1 - x_2 \mod p$
$y_3 \leftarrow \mu(x_1 - x_3) - y_1 \mod p$
Return($Q \leftarrow (x_3, y_3)$)

Denne algoritmen utfører ingen tunge operasjoner, og har heller ingen løkker. Kompleksiteten er $\mathcal{O}(k)$. Dvs. den kjører i konstant tid gitt en fast $p$. uavhengig av input. Ved implementasjon i software vil man derimot få en høyere orden da de aritmetiske operasjonene må gjøres i flere steg. Ordenen på funksjonen blir da $\mathcal{O}(\log n)$ hvor $n = max(x_1, x_2, y_1, y_2)$.

### 4.3.4 Elliptiske kurver over $\mathbb{F}_{2^m}$

På samme måte som for kurver over $\mathbb{F}_p$ kan man danne en abelsk gruppe dersom man definerer addisjon som følger:

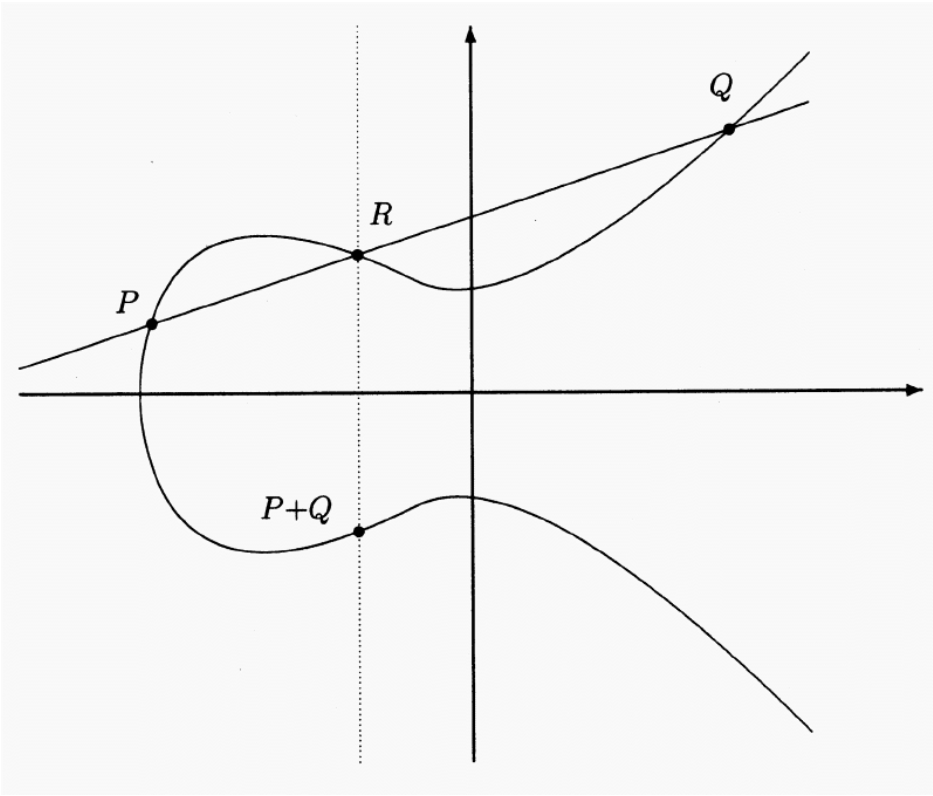
1. $\forall P \in E(\mathbb{F}_{2^m}) : P + \mathcal{O}_\infty = \mathcal{O}_\infty + P = P$
2. Gitt $P(x, y) \in E(\mathbb{F}_p)$
   $P - P = (x, y) + (x, -y) = \mathcal{O}_\infty$
3. Gitt $P = (x_1, y_1) \in E(\mathbb{F}_{2^m}), Q = (x_2, y_2) \in E(\mathbb{F}_{2^m}), P \neq \pm Q$
   $$P + Q = (x_3, y_3) \text{ hvor } \begin{cases} x_3 = \mu^2 + \mu + x_1 + x_2 + a \\ y_3 = \mu(x_1 + x_3) + x_3 + y_1 \\ \mu = \frac{y_2 + y_1}{x_2 + x_1} \end{cases}$$
4. Gitt $P = (x_1, y_1) \in E(\mathbb{F}_{2^m})$
   $$P + P = 2P = (x_3, y_3) \text{ hvor } \begin{cases} x_3 = \mu^2 + \mu + a \\ y_3 = \mu(x_1 + x_3) + x_3 + y_1 \\ \mu = x_1 + \frac{x_1}{y_1} \end{cases}$$

### 4.3.5 Nyttige definisjoner

**Skalar multiplikasjon:** Elliptisk kurve kryptering baserer seg på skalar multiplikasjon: Gitt $k \in \mathbb{Z}$ og et punkt $P \in E(\mathbb{F}_p)$. $kP$ er resultatet av å addere $P$ til seg selv $k$ ganger.

**Et punkts orden:** Ordenen til et punkt $P$ på en elliptisk kurve er det minste positive heltall $r$ s.a. $rP = \mathcal{O}_\infty$. Gitt to heltall $k, l$: Dersom $k \equiv l(\mod r)$ så er $kP = lP$. Det er ønskelig å finne et punkt med orden lik et stort primtall da dette vanskeliggjør beregningen av diskrete logaritmer.

**En kurves orden:** Antall punkter på $E(\mathbb{F}_p)$, betegnet ved $\#E(\mathbb{F}_p)$ kalles kurvens orden. Antall punkter kan beregnes i polynomiell tid med Schoofs og Satohs algoritmer. En punkttellingsalgoritme er nødvendig for å velge en kurve slik at ordenen til kurven inneholder et stort




Figur 4.1: Addisjon av to punkter: $P + Q$

**Algoritme 8** Algoritme for addisjon av to punkter på $E(\mathbb{F}_{2^m})$

INPUT: En elliptisk kurve, $E(\mathbb{F}_{2^m})$, med parametre $a, b \in \mathbb{F}_{2^m}$ og punktene $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$
OUTPUT: $Q = P_1 + P_2$
**if** $P_1 = \mathcal{O}_\infty$ **then**
  Return($Q \leftarrow P_2$)
**end if**
**if** $P_2 = \mathcal{O}_\infty$ **then**
  Return($Q \leftarrow P_1$)
**end if**
**if** $x_1 = x_2$ **then**
  **if** $y_1 = y_2$ **then**
    $\mu \leftarrow x_1 + y_x/x_1$
    $x_3 \leftarrow \mu^2 + \mu + a$
  **else**
    Return($Q \leftarrow \mathcal{O}_\infty$) /* $y_2 = y_1 + x_1$ */
  **end if**
**else**
  $\mu \leftarrow (y_2 + y_1)/(x_2 + x_1)$
  $x_3 \leftarrow \mu^2 + \mu + x_1 + x_2 + a$
**end if**
$y_3 \leftarrow \mu(x_1 + x_3) + x_3 + y_1$
Return($Q \leftarrow (x_3, y_3)$

Denne algoritmen har forskjellig kompleksitet avhengig av om man benytter seg av polynomiell basis eller en normalbasis. Multiplikasjon i polynomiell basis har som tidligere nevnt $\mathcal{O}(m^3)$. Dette innebærer at algoritmen vil få den samme orden ved bruk av denne basisen. Tilsvarende vil den få en orden $\mathcal{O}(m^2)$ ved bruk av en normalbasis.

primtall. Dette gjøres for å sikre at det diskrete logaritmeproblemet ikke skal kunne løses i polynomisk tid/rom.

## 4.4 Domeneparametre

I forbindelse med asymmetriske krypteringssystemer på en elliptisk kurve over en endelig kropp trenger vi "domeneparametre". Domeneparametrene er gitt ved dette tuplet:

$$T = (q, FR, a, b, G, n, h)$$

hvor $q$ spesifiserer den underliggende kroppen ($q = p$ eller $q = 2^m$), $FR$ ("Field Representation") som beskriver hvordan elementene i $\mathbb{F}_q$ er representert, to elementer $a, b \in \mathbb{F}_q$ som spesifiserer ligningen vi bruker ($y^2 = x^3 + ax + b$ for $p > 3$ og $y^2 + xy = x^3 + ax^2 + b$ for $p = 2$), en generator $G = (x_g, y_g) \in E(\mathbb{F}_q)$, et primtall $n = \#G$ (dvs ordenen til $G$) og et heltall $h$ som er gitt ved $h = \#E(\mathbb{F}_q)/n$.

Det finnes flere foreslåtte algoritmer for generering og validering av disse parametrene (dvs. finne "sikre" kurver), men siden sikkerheten i hovedsak baserer seg på størrelsen av $n$, defineres nøkkellengden til å være bit-lengden til $n$.

Hensynene man må ta ved generering av domeneparametrene er spredt litt rundt i oppgaven. Jeg vil for oversiktens skyld lage en kort oppsummering her:

- Weierstrass ligningen må danne en elliptisk kurve. Dette innebærer at $a$ og $b$ må kontrolleres slik det er beskrevet i avsnitt 4.3.
- Kurven vi velger må dessuten *ikke* være supersingulær. Årsaken til dette diskuteres i avsnitt 5.1.1. Dette medfører at vi må kjenne kurveordenen, og derfor trenger en punkttellingsalgoritme.
- Kurven vi velger må i tillegg *ikke* være anomal. Årsaken til dette diskuteres i avsnittet 5.1.2.
- Kurveorden må bestå av minst en stor primfaktor, se avsnitt 5.1.4.

- Punktordenen bør være et stort primtall for å vanskeliggjøre det diskrete logaritmeproblemet (se avsnitt 5.1 for en beskrivelse av problemet).

### 4.4.1 Nøkkellengde

| Symmetrisk chiffer nøkkellengde | Algoritme | Elliptisk kurve nøkkellengde | DSA/RSA-nøkkellengde |
|---|---|---|---|
| 80 | skipjack | 160 | 1024 |
| 112 | Trippel-DES | 224 | 2048 |
| 128 | 128-bit AES | 256 | 3072 |
| 192 | 192-bit AES | 384 | 7680 |
| 256 | 256-bit AES | 512 | 15360 |

Tabell 4.1: EKK,DSA og RSA-nøkkellengder

Denne tabellen viser nøkkellengden i forhold til sikkerheten for NIST-kurver[5](se A. Menezes: [BHLM01]). Det er tatt hensyn til de kjente angrepene vi har per i dag. Som det går frem vil man selv med kjente parametre ha relativ god sikkerhet i et elliptisk kurve krypteringssystem selv for små nøkkellengder.

### 4.4.2 Hvordan velge en kurve

Ettersom det har blitt vist at en rekke typer elliptiske kurver ikke egner seg for kryptering er man nødt til å undersøke om kurven egner seg for kryptering før man tar den i bruk. Et av de viktigste problemene i så måte er å finne en kurve med mange nok punkter. Man må deretter forsikre seg om at man ikke har valgt seg en kurve hvor det diskrete logaritmeproblemet for elliptiske kurver er lett. Det sistnevnte behandles i kapittel 5. Antall punkter på kurven vil bli behandlet i de påfølgende avsnittene.

[5]National Institute of Standards and Technology (NIST) har publisert data på "anbefalte" kurver, dvs kurver man antar er sikre

#### 4.4.2.1 Antall punkter på en kurve

For at vi skal kunne bruke en kurve er det viktig at den har nok punkter. Dette regnes for å være et av de vanskeligste problemene ved elliptisk kurve kryptering.

For en kurve av odde karakteristikk vil ordenen til kurven, designert ved $\#E(\mathbb{F}_q)$, være ca $q$.

La $E(x)$ betegne Weierstrass-ligningen[6] hvor verdien $a_2$ er substituert med variabelen $x$, og la $\frac{E}{x}$ være $1$ mindre enn antall løsninger $y \in \mathbb{F}_p$ for Weierstrassligningen. E.g. $E : y^2 + xy = x^3 + 1$ og $K = \mathbb{F}_2$. $E(0)$ er da ligningen $y^2 = 1$ som har en rot hvilket gir oss $E(0)/2 = 0$. $E(1)$ gir ligningen $y^2 + y = 0$ som har to røtter hvilket gir oss $E(1)/2 = 1$.

Antall punkter på $E$ ("1+" fordi $\mathcal{O}_\infty$ også må legges til) definert over $\mathbb{F}_p$ er derfor

$$|E(\mathbb{F}_p)| = 1 + \sum_{x \in \mathbb{F}_p} \left(1 + \left(\frac{E(x)}{q}\right)\right) = q + 1 + \sum_{x \in \mathbb{F}_p} \left(\frac{E(x)}{q}\right) \tag{4.5}$$

Dette skrives ofte $q + 1 - t$ hvor $t$ betegner Frobeniustrasen:

$$t = t(E, q) = -\sum_{x \in \mathbb{F}_p} (\frac{E(x)}{q})$$

.

Hasses teorem setter en grense på hvor mange punkter man kan finne på en kurve:

$$q + 1 - 2\sqrt{q} \leq \#E \leq q + 1 + 2\sqrt{p} \tag{4.6}$$

hvilket setter følgende begrensning på $t$: $|t| \leq 2\sqrt{q}$.

**Punkttellingsalgoritmer**

Frem til ganske nylig ble den såkalte SEA-algoritmen (også kalt Schoofs punkttellingsalgoritme) regnet for å være den mest effektive. Takakazu Satoh introduserte i 2000 en ny punkttellingsalgoritme. Denne har siden blitt forbedret og optimalisert og i de nyeste implementasjonene (se

[6]Dvs. ligning 4.3 for kurver over kropper av odde karakteristikk og ligning 4.4 for kurver over kropper av jevn karakteristikk

[FGH00]) har den fått orden $\mathcal{O}(n^k), k < 4$. Det finnes kommersielle implementasjoner tilgjengelig som man kan kjøpe dersom man velger å bruke tilfeldige kurver i implementasjonen.

#### 4.4.2.2 NIST-kurver

National Institute of Standards and Technology (NIST) har anbefalt et utvalg av elliptiske kurver og underliggende kropper (se [oCIoST00]). De anbefaler følgene kroppstørrelser til bruk av føderale myndigheter.

| Symmetrisk chiffer nøkkellengde | Algoritme | Bit-lengde av $p$ i primkropper ($\mathbb{F}_p$) | Dimensjon $m$ av binærkropper ($\mathbb{F}_{2^m}$) |
|---|---|---|---|
| 80 | SKIPJACK | 192 | 163 |
| 112 | Trippel-DES | 224 | 233 |
| 128 | AES | 256 | 283 |
| 192 | AES | 384 | 409 |
| 256 | AES | 521 | 571 |

Tabell 4.2: NISTs anbefalte kroppstørrelser for bruk av føderale myndigheter

NIST har også publisert tilhørende domeneparametre og anbefalte nøkkelstørrelser[7].

Disse kurvene har vist seg å være sikre, selv om samtlige domeneparametre har vært kjent (se [BHLM01]), og bør derfor trygt kunne brukes i en implementasjon. Man er derfor ikke avhengig av å ha en punkttellingsalgoritme.

## 4.5 Elliptisk kurve primitiver

En nøkkel for elliptisk kurve kryptering fungerer kun dersom domeneparametrene (se 4.4) er kjent. Det har vært foreslått at man kan bruke *faste* parametre som er kjent for både avsender og mottaker, eller eventuelt inkludere domeneparametrene med nøkkelen. Fordelen med å bruke faste parametre (som f.eks. NIST-kurvene) er at man vet at de er sikre ettersom

[7] Se avsnitt 4.4.1

de har vært utsatt for alle tenkelige angrep (se 4.4.1 for oversikt over sikkerhet i forhold til nøkkellengde) og at man får mindre nøkler. Det siste er en aktuell problemstilling for anvendelser på smartkort hvor man ønsker å operere med minst mulig nøkkelstørrelse.

Dersom man velger å bruke forskjellige domeneparametre for hver nøkkel trenger man en nøkkelvalideringsalgoritme og en algoritme for å finne passende domeneparametre.

**Nøkkelgenerering:** Denne algoritmen forutsetter at domeneparametrene er kjent. Se avsnittet om domeneparametre for informasjon om hvordan disse bør velges.

**Algoritme 9** Nøkkelgenereringsalgoritme

INPUT: Domeneparametre $T = (q, FR, a, b, G, n, h)$[8]
OUTPUT: Et nøkkelpar $(d, Q)$ hvor $d$ er den hemmelige og $Q$ den offentlige nøkkelen.
$d = rnd(1, n-1)$ /* d = et tilfeldig tall i intervallet $[1, n-1]$
$Q \leftarrow dG$
Return($(d, Q)$)

Nøkkelvalideringsalgoritmen sørger for at en offentlig nøkkel tilfredstiller de aritmetiske kravene til en elliptisk kurve nøkkel (se [Sec99]). En offentlig nøkkel $Q = (x_Q, y_Q)$ sammen med domeneparametrene $T = (q, FR, a, b, G, n, h)$ valideres med følgende metode:

**Eksplisitt Nøkkelvalidering:** [9]

### 4.5.1 Hvordan finne punkter på kurven

Metoden som anvendes for å finne et punkt på kurven er avhengig av om man anvender en kurve over $\mathbb{F}_p$ eller $\mathbb{F}_{2^m}$. Algoritmen som presenteres her

[9] Dersom man hopper over det siste kravet ($nQ = \mathcal{O}_\infty$) kalles algoritmen en delvis validering.

**Algoritme 10** Nøkkelvalideringsalgoritme

INPUT: Domeneparametre $T = (q, FR, a, b, G, n, h)$ og en offentlig nøkkel $Q = (x_q, y_q)$
OUTPUT: "OK" eller "IKKE OK"
**Ensure:** $Q \neq \mathcal{O}_\infty$
**Ensure:** $(x_q, y_q) \in \mathbb{F}_q$
**Ensure:** $Q$ ligger på kurven $E(\mathbb{F}_q)$ gitt ved $a, b$
**Ensure:** $nQ = \mathcal{O}_\infty$
**if** Alle krav ovenfor er møtt **then**
  Return("OK")
**else**
  Return("IKKE OK")
**end if**

er beregnet på kurver over $\mathbb{F}_p$. En tilsvarende algoritme for kurver over $\mathbb{F}_{2^m}$ finnes i [BSS99].

Den enkleste måten å finne et punkt på kurven er å finne et tilfeldig $x$-koordinat og deretter løse Weierstrassligningen med hensyn på $y$. Dette kan gjøres med algoritme 13.

**Algoritme 11** Finne et punkt på kurven, se [BSS99]

INPUT: Elliptisk kurve E(K)
OUTPUT: et punkt $P \in E(K)$
**repeat**
  La $x$ være et tilfeldig tall slik at $x \in K$.
  Forsøk å løse lignign 4.3 mhp. $y$ ved å bruke metoden for å beregne kvadratrøtter modulo p (se nedenfor).
  dersom man finner en $y$ la $P = (x, y)$.
**until** et punkt P er funnet
Returnér $P$.

### Kvadratisk rest

**Definisjon:** Gitt et primtall $p$. $a \in \mathbb{Z}_p^*$ kalles en kvadratisk rest modulo $p$ hvis og bare hvis $x^2 \equiv a (\mathrm{mod}\ p)$ har en løsning.

**Legendresymbolet**

**Definisjon:** La $p$ være prim. $\forall a \in \mathbb{Z}_p^*$. *Legendre* symbolet $\left(\frac{a}{p}\right)$ er definert ved:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & a \equiv 0(\mod p) \\ 1 & a \text{ er en kvadratisk rest modulo p} \\ -1 & a \text{ er } \textit{ikke} \text{ en kvadratisk rest modulo p} \end{cases} \tag{4.7}$$

Vi ser at dette henger sammen med følgende teorem:

**Teorem ("Eulers kriterium"):**

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2}(\mod p) \tag{4.8}$$

**Kvadratisk rest**

**Teorem:** La $p$ være prim og $\alpha \in \mathbb{Z}_p^*$ en generator. $a$ er en kvadratisk rest hvis og bare hvis $a = \alpha^i (\mod p)$ hvor $2|i$.

### 4.5.2 ElGamal på elliptiske kurver

Som tidligere nevnt baserer sikkerheten i elliptiske kurver seg på diskrete logaritmer. Som vi husker fra avsnitt 3.4 er ElGamal en slik algoritme. Ettersom både multiplikasjon og kvadrering er definert for en elliptisk kurve kan vi gjøre dette uten store problemer:

Gitt domeneparametre $T$. Gitt en nøkkel, $dQ$, (generert med algoritmen ovenfor). Bob ønsker å sende meldingen $P$ til Alice og genererer et tilfeldig tall $r \leftarrow rnd(1, n-1)$ og sender $(rG, P + r(dQ)$ til Alice. Alice vet at $d(rQ) = r(dQ)$, og kan derfor multiplisere sin "hemmelige verdi", $d$, med det første elementet i tuplet. Deretter kan hun trekke resultatet fra det andre elementet i tulpet for å få $P$: $P = P + r(dQ) - d(rQ) = P$.

**Algoritme 12** Algorithme for å beregne Legendresymbolet

INPUT: a og p.
OUTPUT: $\left(\frac{a}{p}\right) \in \{1, 0, -1\}$.
**if** $a \equiv 0 (\mathrm{mod}\ p)$ **then**
  return $0$
**end if**
$x \leftarrow a, y \leftarrow p, L \leftarrow 1$
**loop**
  $x \leftarrow x (\mathrm{mod}\ y)$
  **if** $x > y/2$ **then**
    $x \leftarrow y - x$,
    **if** $y \equiv 3 (\mathrm{mod}\ 4)$ **then**
      $L \leftarrow -L$
    **end if**
    **while** $x \equiv 0 (\mathrm{mod}\ 4)$ **do**
      $x \leftarrow x/2$
      **if** $y \equiv \pm 3 (\mathrm{mod}\ 8)$ **then**
        $L \leftarrow -L$
      **end if**
    **end while**
  **end if**
  **if** $x = 1$ **then**
    return $L$
  **end if**
  **if** $x \equiv 3 (\mathrm{mod}\ 4)$ *and* $y \equiv 3 (\mathrm{mod}\ 4)$ **then**
    $L \leftarrow -L$
  **end if**
  $x, y \leftarrow y, x$
**end loop**

**Algoritme 13** Algoritme for å beregne kvadratisk rot modulo p

1: INPUT: $a$ og $p$ s.a. $\left(\frac{a}{p}\right) = 1$.
2: OUTPUT: $x$ s.a. $x^2 \equiv a(\text{mod } p)$
3: Vel en tilfeldig $n$ til følgende er oppfylt $\left(\frac{n}{p}\right) = -1$
4: La $e, q \in \mathbb{Z}_p^*$ s.a. $q$ er odde og $p - 1 = 2^\alpha q$
5: $y \leftarrow n^q(\text{mod } p), r \leftarrow e, x \leftarrow a^{p-1}/2$
6: $b \leftarrow ax^2(\text{mod } p, x \leftarrow ax(\text{mod } p)$
7: **while** $b \not\equiv 1(\text{mod } p)$ **do**
8: Finn den minste $m$ s.a. $b^{2^m} \equiv 1(\text{mod } p)$
9: $t \leftarrow y^{2^{r-m-1}}(\text{mod } p), y \leftarrow t^2(\text{mod } p), r \leftarrow m$
10: $x \leftarrow xt(\text{mod } p), b \leftarrow by(\text{mod } p)$
11: **end while**
12: Return($x$)

#### 4.5.2.1 Eksempel[Sti95]

Den elliptiske kurven $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ er løsningsmengden $\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p\}$ til kongruensen

$$y^2 \equiv x^3 + ax + b(\text{mod } p)$$

(Eksemplet er hentet fra boken "Cryptography - Theory and Practice" av Stinson:). Vi setter $a = 1$ og $b = 6$. Vi genererer punktene ved å iterere over tillatte $x$-verdier og løse kongruensen ovenfor mhp. y. Dette gir følgende punkter:
$\{(2,4),(2,7),(3,5),(3,6),(5,2),(5,9),(7,2),(7,9),(8,3),(8,8),(10,2),(10,9)\}$
I tillegg har vi identitetselementet $\mathcal{O}_\infty$. Det eneste vi mangler er en generator for kurven. Ettersom vi i dette eksemplet har valgt en kropp av odde karakteristikk, $\mathbb{F}_p$, og $p$ er prim vil samtlige punkter på kurven være en generator (bortsett fra $\mathcal{O}_\infty$).

La $\alpha = (2, 7)$ være vår generator og $a = 3$. Vi får da $\beta = a * \alpha = 3 * \alpha =$

$\alpha + \alpha + \alpha$. Vi må først beregne $\alpha + \alpha$:

$$\begin{aligned}\mu &= (3 * x^2 + a)/(2 * y) = (3 * 2^2 + 1)(2 * 7)^{-1} \mod 11 \\ &= 2 * 3^{-1} \mod 11 \\ &= 2 * 4 \mod 11 \\ &= 8\end{aligned}$$

Vi får da:

$$\begin{aligned}x' &= \mu^2 - x - x = 8^2 - 2 - 2 \mod 11 = 5 \\ y' &= \mu(x - x') - y \\ &= 8(2 - 5) - 7 \mod 11 \\ &= 2\end{aligned}$$

Dvs. $2\alpha = (5, 2)$.

Vi må så beregne $2\alpha + \alpha = (5, 2) + (2, 7)$ på tilsvarende måte:

$$\begin{aligned}\mu &= (7 - 2)(2 - 5)^{-1} \mod 11 \\ &= 5 * 8^{-1} \\ &= 5 * 7 \mod 11 \\ &= 2\end{aligned}$$

Dette gir oss:

$$\begin{aligned}x'' &= 2^2 - 5 - 2 \mod 11 \\ &= 8\end{aligned}$$

og

$$\begin{aligned}y'' &= 2(5 - 8) - 2 \mod 11 \\ &= 3\end{aligned}$$

hvilket gir oss Bobs offentlige nøkkel: $\beta = 3\alpha = (8, 3)$. For å kunne bruke dette systemet må man i tillegg vite hvilken elliptisk kurve man har brukt (E) samt hvilken modulus man har brukt ($q = 11$). Alice ønsker å kryptere meldingen $M = (10, 9)$. Hun må gjøre følgende:

1. Velg en tilfeldig verdi $k$
2. Beregn x:$k * \alpha \mod p$

3. Beregn y:$(M + \beta * k) \mod p$

4. send $(x, y)$ til bob

Alice velger $k = 5$. $x = 5 * \alpha = (3, 6)$. og $y = M + \beta * k = (10, 9) + (8, 3) * 5 = (10, 9) + (5, 2) = (5, 9)$.

Bob kan så dekryptere meldingen for å få tilbake M:

$$\begin{aligned} M &= x - a * y \mod p \\ &= (5, 9) - a(3, 6) \\ &= (5, 9) - (5, 2) \\ &= (5, 9) + (5, 9) \\ &= (10, 9) \end{aligned}$$

Det eksisterer noen praktiske problemer ved implementasjon av ElGamal kryptering på en elliptsk kurve. ElGamal har vanligvis en meldingsekspansjonsfaktor på to. Ved implementasjon på en elliptisk kurve får man en faktor på fire. Et annet problem er at alle ukrypterte tekster må plasseres på kurven. Det finnes ikke noen opplagt entydig måte å gjøre denne avbildingen på. Begge disse problemene kan løses ved å anvende Menezes-Vanstone krypteringssystemet som tillater at både ukryptert og kryptert tekst ikke behøver å ligge på en kurve. Vi får fremdeles en meldingsekspansjon, men denne har kun en faktor på to [Sti95].

### 4.5.3 Elliptic Curve Diffie-Hellman (ECDH)

ECDH er en versjon av Diffie-Hellman nøkkelutvekslingsprotokollen. Den fungerer ved at det genereres en hemmelig verdi fra en privat nøkkel eid av Alice og en offentlig nøkkel eid av Bob.

$P$ kan også beregnes slik: $P \leftarrow hd_A Q_B$. Dette kalles for "elliptisk kurve kofaktor Diffie-Hellman". Dette gir beskyttelse for en del angrep[Sec99].

**Algoritme 14** Elliptic Curve Diffie-Hellman

INPUT: Domeneparametre $T = (q, FR, a, b, G, n, h)$, en privat nøkkel eid av Alice, $d_A$, og en offentlig nøkkel eid av Bob: $Q_B$.
OUPUT: en hemmelig verdi $z$
**repeat**
  $P \leftarrow d_A Q_B = (x_P, y_P)$
**until** $P \neq \mathcal{O}_\infty$
$z \leftarrow z = x_P$

### 4.5.4 DSS for elliptiske kurver (EC-DSA)[JM97]

Det er utarbeidet en variant av DSS (se avsnitt 3.5.1) som kalles for "Elliptic Curve Digital Signature Algorithm" (ECDSA)[IEE01]. Dette er en versjon av ElGamal-signaturer som er utgangspunktet for de allerede omtalte anbefalingene til NIST (se [oCIoST00]).

**Algoritme 15** EC-DSA

INPUT: EK-parametrene $q, a, b, n, G$ (se 4.4),tilhørende nøkkel $Q$ samt melding $M$.
OUTPUT: Signatur: $(c, d)$
**repeat**
  Generer et nøkkelpar $(u, V)$ i samme domene som den private nøkkelen
  **repeat**
    $V \leftarrow (x_v, y_v)$
  **until** $V \neq 0$
  $x_v \leftarrow FE2IP(x_v)$
  $c \leftarrow i \mod n$
  $d \leftarrow (f + Qc)u^{-1} \mod n$
**until** $c \neq 0$ and $d \neq 0$
return $(c, d)$

### 4.5.5 Elliptic Curve Authenticated Encryption Scheme(ECAES)

For å bruke denne algoritmen må både A og B kjenne funksjonene $MAC$[10],$ENC$[11], og $KDF$[12].

---

**Algoritme 16** ECAES - kryptering

---

For å kryptere en melding $m$ for oversendelse til B må A bruke følgende algoritme:

INPUT: Domeneparametre $T = (q, FR, a, b, G, n, h)$ samt en offentlig nøkkel $Q_B$ tilhørende Bob samt meldingen $m$.
OUTPUT: Kryptert melding $m'$
Velg et tilfeldig tall $r \in [1, n-1]$
$R \leftarrow rG$
**repeat**
  $K = hrQ_B = (K_x, K_y)$
**until** $K \neq \mathcal{O}_\infty$
$k_1||k_2 \leftarrow KDF(K_x)$
$c \leftarrow ENC_{k_1}(m)$
$t \leftarrow MAC_{k_2}(c)$
$m' \leftarrow (R, c, t)$
Return($m'$)

---

[10]MAC står for Message Authentication Code, e.g. HMAC[KBC97]

[11]ENC betegner en symmetrisk krypteringsalgoritme som f.eks trippel-DES

[12]KDF står for Key Derivation Function; en funksjon for å utveksle nøkler fra en delt hemmelighet. (Se ECDH ovenfor)

**Algoritme 17** ECAES – dekryptering

For å dekryptere meldingen fra A må B bruke følgende algoritme:

INPUT: De samme domeneparametrene ($T$) som nevnt ovenfor samt den krypterte meldingen $m' = (R, c, t)$.
OUTPUT: Den dekrypterte meldingen $m$.
Valider nøkkelen $R$ (se algoritmen under avsnittet om ECDH)
**repeat**
  $K \leftarrow hd_B R = (K_x, K_y)$
**until** $K \neq \mathcal{O}_\infty$
$k_1 || k_2 = KDF(K_x)$
**if** $t \neq MAC_{k_2}(c)$ **then**
  Abort
**end if**
$m \leftarrow ENC_{k_2}^{-1}(c)$
Return($m$)

# 5

# Diskrete logaritmer

Sikkerheten til elliptisk kurve kryptering er basert på den tilsynelatende kompleksiteten til det diskrete logaritmeproblemet for elliptiske kurver som er en variant av problemet beskrevet nedenfor.
**Generalized Discrete Logarithm Problem (GLDLP):**

**Instans:** Gitt en endelig syklisk gruppe $G$ av orden $n$, en generator $\alpha$ og et element $\beta \in G$.

**Problem:** Finn et heltall $x, 0 \leq x \leq n-1$ slik at $\alpha^x = \beta$. Vi skriver da $x = log_\alpha \beta$

## 5.1 Det diskrete logaritmeproblemet for en elliptisk kurve (ECDLP)

Gitt et punkt $P \in E(K)$. $P$ vil da generere en syklisk undergruppe av $E(K)$. Det finnes derfor et tilsvarende problem som nevnt ovenfor for elliptiske kurver. Det diskret logaritmeproblemet anses (som tidligere nevnt) som et mye vanskeligere problem enn det tilsvarende problemet for $\mathbb{Z}$[Men95]:

**Instans:** For definisjon av elliptiske kurver, se 4. Gitt en elliptisk kurve $E(\mathbb{F}_q)$, et punkt $P \in E(\mathbb{F}_q)$ av orden $n$ og et punkt $Q \in E(\mathbb{F}_q)$.

**Problem:** Bestem $k \in \mathbb{F}, 0 \leq k \leq n-1$ s.a. $Q = kP$ (dersom en k i det hele tatt eksisterer).

### 5.1.1 EK må være ikke-supersingulær

Menezes, Okamato og Vanstone (se [MVO91]) viste at det diskrete logaritmeproblemet for elliptiske kurver kan reduseres til det diskrete logaritmeproblemet i $\mathbb{F}_{q^k}$. Dette er opphavet til MOV-algoritmen (etter opphavsmennene). Denne er svært effektive på supersingulærekurver (se også Harazawa [HSSI99]). Dette er bare praktisk mulig for små verdier av $k$. For klassen av supersingulære kurver (trasen til en kurve over $\mathbb{F}_q$, $t(E)$, er delelig med karakteristikken til $\mathbb{F}_q$, $t(E) = q - \#E(\mathbb{F}_q)$) har vi $k \leq 6$. Det er derimot vist[BK98] at kurver med tilfeldige parametre som regel har $k > log^2 q$. For å unngå dette angrepet må man sjekke at $n$, den største primfaktoren i ordenen til kurven,ikke deler $q^k - 1$ for en $k$ hvor det diskret logaritmeproblemet i $\mathbb{F}_{q^k}$ er løselig. I praksis gjøres dette for $1 \leq k \leq 30$. Hele dette problemet kan unngås dersom man velger ikke-supersingulære kurver.

### 5.1.2 Anomale kurver

**Lemma:** En kurve kalles anomal dersom $t(E, p) = 1$ (hvilket betyr at $\#E(K) = p$)[BSS99, s35].

Det finnes flere polynomiell-tid algoritmer ($f \in FP$) for å løse ECDLP i undergupper på en anomal kurve[BSS99, s79]. Disse angrepene ser ikke ut til å virke for andre kurver. For å unngå dette angrepet må man sjekke at ordenen til kurven ikke er identisk med kardinaliteten til den underliggende kroppen.

### 5.1.3 Pohlig Hellman

Pohlig og Hellmann har laget en algoritme som kan brukes for å angripe systemer som er basert på en orden bestående av små primfaktorer[Sti95, s.187]. Det er derfor viktig at man velger en gruppe hvor kurveordenen

inneholder minst en stor primfaktor. [BSS99, ss.80] beskriver dette angrepet.

Pohlig Hellman for elliptiske kurver er en parallell med Pohlig Hellman for $\mathbb{Z}$. Vi ønsker å finne $m$ gitt $Q$ der $Q = mP$. Gitt gruppeordenen $n$ og en generator $G$ kan vi m.a.o. løse problemet for undergrupper med orden $n_i, n_0 n_1 \ldots n_r = n$ og deretter bruke det kinesiske restleddteoremet for å beregne $m$.

Dersom vi setter $n_c = n/p^{c-1}$ hvor $p^c$ er ordenen til generatoren i undergruppen, kan vi bruke følgende algoritme:

**Algoritme 18** Pohlig Hellman for en elliptisk kurve

1: Finn primfaktorene til $n_c$ til $n$
2: beregn $\gamma_k = n_k G, 0 \leq k \leq n-1$
3: **for** $j$ from 0 to $c-1$ **do**
4: $\quad \delta_j \leftarrow n_k P$
5: $\quad$ finn $i$ slik at $\delta_i = \gamma_k + Q$
6: $\quad m_j \leftarrow i$
7: $\quad R_j = (Q - m_j P)$
8: **end for**

### 5.1.4 Pollard-$\rho$ metoden

J. Pollard [Men93, Pol78] fant en metode for å beregne logaritmer som er probabilistisk, men som fjerner nødvendigheten til å regne ut en liste over logaritmer. Denne metoden har senere blitt modifisert til å løse ECDLP. Algoritmen har orden $\mathcal{O}(\sqrt{n\pi/2})$ ([BSS99]), hvilket er relativt effektivt.

Som tidligere ønsker vi å finne $m$ gitt $Q$ hvor $Q = mP$

Denne metoden kalles ofte for "ville og tamme kenguruer". Den fungerer ved at man velger et tilfeldig punkt på kurven ("kenguruer") og utfører en "tilfeldig tur" (random walk). Dette er utføres av både den "tamme kenguruen" (hvor man kjenner $Q' = m'P$) og for den ville kenguruen. Før eller siden den ville kenguruen havne i det samme sporet som den tamme. Gitt startpunktene $g_0$ og $h_0$ for hhv. den tamme og den ville

kenguruen:

$$g_0 = x_0 P + x'_0 Q$$
$$h_0 = y_0 + y'_0 Q$$

Gitt at $g_k = h_k$ (dvs. at kenguruene møtes i et punkt etter $k$ iterasjoner: Vi har da

$$x_k P + x'_k Q = g_k$$
$$h_k = y_k P + y'_k Q$$

hvilket innebærer at $(x_k - y_l)P = (y'_l - x'_k)Q = (y'_l - x'_k)mP$

Det skulle nå være mulig å løse det diskrete logaritmeproblemet.

Flere standardiseringsforslag har foreslått anvendelsen av $E(\mathbb{F}_{2^m}), m = 163$ og at man skal bruke kurver hvis orden er dividerbar med et primtall på minst 160 bits. Dette ble gjort i den tro at det ikke ville være mulig å beregne logaritmer i mindre enn $2^{80}$ operasjoner [WZ98, Com98]. [WZ98] viser derimot at antall operasjoner kan reduseres med en faktor på $\sqrt{2m}$ noe som gir ca $2^{77}$ operasjoner. Dette er under "sikkerhetsmarginen" som er foreslått av standardene. Legg merke til at det her ikke er snakk om noe annet enn en forbedring av et eksponensielt problem med en liten faktor.

# 6

# Metode

I denne oppgaven har jeg benyttet meg av to metoder. Disse er henholdsvis kompleksitetsteori og modelleringsteori.

## 6.1 Unified Modelling Language (UML)

UML er et modelleringsspråk som ikke bare er beregnet på modellering av datastrukturer/klasser og funksjoner. I denne oppgaven har jeg dog kun brukt den til den siste delen. Selv om implementasjonen i denne oppgaven er svært liten, er det nyttig både for oversiktens og for implementasjonens skyld å danne seg et godt bilde av hva man skal implementere. Valget av UML som modelleringsverktøy var ganske innlysende da dette er "de-facto" standarden for IT-industrien. Det finnes selvfølgelig flere andre modelleringsspråk, men ingen av disse har de samme muligheter når det gjelder å utrykke prosesser og interaksjoner mellom objekter.

## 6.2 Kompleksitetsteori

For å kunne bedømme hvilke algoritmer som egner seg for implementasjon på forskjellige platformer er det viktig å vite hva slags kompleksitet en algoritme har. Dette gjelder både med henseende til hvor effektiv den kan implementeres, og med tanke på hva som lar seg implementere gitt begrensninger på minne- og ressursbruk. Kompleksitetsteori er definert over en abstrakt platform kalt en "Turingmaskin". Turingmaskinen har navn etter opphavsmannen, Alan Turing, som beskrev denne på begynnelsen av 1920-tallet. En Turingmaskin er en universell beskrivelse av en datamaskin som skal kunne utføre alle programmer. Poenget med en Tur-

ingmaskin er at kompleksiteten et problem har på denne maskinen gjelder for alle platformer og datamaskiner. Ettersom alle platformer kan simuleres av en Turingmaskin vil kompleksiteten på en platform nødvendigvis gjelde for de andre platformene.[1] Dette innebærer at man kan bruke et universelt sett med regler for å beskrive kompleksiteten til et problem. Ingen av algoritmene diskutert i denne oppgaven tilhører dog noen annen klasse enn $P$, men enkelte av dem har grad $\mathcal{O}(n^2)$ og $\mathcal{O}(n^3)$ hvilket betyr at de for selv moderate verdier av $n$ vil ta vesentlig lengre tid å beregne. I forbindelse med kryptografi-algoritmer snakker man som regel om svært store tall, og enkelte av algoritmene vil derfor være praktisk talt ubrukelige. $\mathcal{O}$-notasjon er derfor en av de beste måtene å vurdere ulike algoritmer i forhold til effektivitet. Se forøvrig[Wei95, GJ79, Pap95] for en diskusjon av dette.

## 6.3 Hastighetsmålinger av programmer

Ved hastighetsberegningene i implementasjonen i denne oppgaven har jeg anvendt meg av programmet gprof. Dette programmet er nøye tilpasset platformen man anvender og er avhengig av at kompilatoren legger inn informasjon den kan bruke når hastighet og kall-grafer i programmet skal genereres. Alle testene i denne oppgaven ble gjort på FreeBSD 4.5S på en Pentium Celeron 750MHz. Hastighetsberegningene til gprof regnes i $1/100$-sekunder. Jeg har derfor valgt å kalle funksjonene av interesse $1000$-ganger for å frem forholdstall. Ettersom det mest interessante for en implementasjon er kompleksiteten til funksjonene (gitt ved $\mathcal{O}$-notasjon) har jeg valgt å se på forholdet i kjøretiden mellom de forskjellige funksjonskallene for å se hvor "skoen trykker": Hastighetsmålinger i $\mu$sekunder er svært avhengig av hvilken prosessor man bruker. Slike målinger vil ikke gi leseren noe særlig forhold til hastigheten.

[1] Det finnes flere typer Turingmaskiner, og disse er alle opphav til egne kompleksitetsklasser; det vi her i første grad snakker om er ordenen til en funksjon. Se avsnitt 3.1

# 7

# Implementasjon

## 7.1 Valg av matematikkbibliotek

Når man skal lage kryptografirutiner er det viktig at man har effektive rutiner for multipresisjonsmatematikk.Under arbeidet med oppgaven prøvde jeg ut de fleste bibliotekene. Det viste seg at svært mange av bibliotekene inneholdt til dels stygge syntaksfeil i kildekoden (f.eks. $crypto{+}{+}$ som det refereres til fra IEEE1363 siden) som gjorde at de ikke lot seg kompilere på Unix med egcs/gcc uten store endringer. Disse problemene skyldtes at $C{+}{+}$-standarden ikke var ferdig da bibliotekene ble laget eller at forfatterne ofte brukte konstruksjoner som bare ble støttet av en kompilator.

### 7.1.1 GMP

Jeg endte til slutt opp med å bruke GMP ("GNU Multiple precision arithmetic library"). Dette biblioteket er gratis og er av høy kvalitet. Det støtter vilkårlig presisjon og er i tillegg et av de raskeste bibliotekene tilgjengelig [Ber99]. Biblioteket inneholder i tillegg funksjoner som regner ut Legendre og Jacobi symboler og er lagt opp for bruk mot kryptografi.

### 7.1.2 APfloat

APfloat biblioteket innehold mange syntaks-feil som gjorde det vanskelig å kompilere. Det inneholdt heller ikke alle funksjonene for lavnivå manipulasjon av bits som var nødvendig for en del av funksjonene som ble implementert i forbindelse med oppgaven.

### 7.1.3 Crypto++

Crypto++ består av en mengde funksjoner for vilkårlig presisjonsaritmetikk samt innebygget funksjoner for flere krypteringssystemer.

I tillegg til de ovennevte problemene med kompileringen av dette biblioteket var det også lisensproblemer: Forfatteren krever å få alle endringer og utvildelser av biblioteket, og vil i tillegg at man skal fraskrive seg opphavsretten til koden. I motsetning til en del andre lisenser som er i bruk medfører denne lisensen at forfatteren av biblioteket får rettighet til all kommersiell bruk av programkoden. Dette anså jeg som uakseptabelt.

### 7.1.4 CLN

CLN var et av de raskeste bibliotekene jeg testet. Dette inneholdt dessverre heller ikke funksjoner for lavnivå manipulering av bits i versjonen som ble testet.

### 7.1.5 Miracl

Miracl er et kommersielt[1] bibliotek. I tester jeg utførte i forbindelse med oppgaven var det ikke særlig raskt sammenlignet med en del av de andre bibliotekene.

## 7.2 Beskrivelse av lagdeling i implementasjon

Ved implementasjon av elliptiske kurver må målet fra et modeleringsstandpunkt være å skjule den underliggende matematikken mest mulig for programmereren. Ved bare å lage et grensesnitt bestående av funksjoner har man etter min mening ikke gjort dette. Den rette måten å gjøre dette på måtte derfor bli å bruke et objektorientert språk. Det finnes da flere varianter å velge mellom. De mest nærliggende var Java og C++. Ettersom Java har en god del innebyggede funksjoner som er beregnet på

[1] gratis for "non-profit" bruk

kryptering var dette en god kandidat, men ettersom effektivitet og fullstendig kontroll helt ned til lavnivå var nødvendig for en del operasjoner (som f.eks. multiplikasjon i $\mathbb{F}_{2^m}$) falt valget på C++. Fordelen med C++ var at jeg da også kunne implementere aritmetiske operatorer på de selvkonstruerte typene slik at de oppfører seg helt som de innebyggede typene. Den eneste ulempen jeg kan se er at det blir en del kopiering av data ettersom man ikke bør endre innholdet i et objekt i en operator. (Dersom man eksempelvis utfører operasjonen $a = b+c$ bør man ikke endre på operandende på høyre side av uttrykket). Dette medfører at operasjonen krever at man returnerer et nytt objekt som resultat av operasjonen. I implementasjonen har jeg valgt å konstruere flere lag som det går frem av figuren nedenfor:


Figur 7.1: UML-diagram over klassene i denne implementasjonen

Klassene mlib og gmp er selvstendige klasser hvor det er implementert en rekke operatorer slik at disse kan fungere uavhengig av klassen ec som implementerer selve aritmetikken for elliptisk kurve operasjoner. Både mlib og gmp kan brukes som innebyggede typer. Klassen mlib implementerer kun aritmetikk i $\mathbb{F}_{2^m}$ og ble laget for å demonstrere hvordan aritmetikk i $\mathbb{F}_{2^m}$ kan implementeres i mangel av multipresisjonsfunksjoner, som kan være tilfelle på enkelte smartkort. (De fleste har dog en egen multipresisjons prosessor med tilhørende funksjoner innebygget i opera-

tivsystemet). Klassen gmp er en overbygning for biblioteket Gnu MP. Dette biblioteket regnes for å være det raskeste multipresisjonsbiblioteket tilgjengelig på markedet, og er i tillegg gratis. Se forøvrig avsnitt 7.1.

Jeg har valgt å implementere elliptisk-kurve operasjoner klassen ecelement. Denne klassen inneholder de statiske domeneparametrene i tillegg til koordinatene til punktet man opererer på. For å ha muligheten til å kunne bytte ut de underliggende bibliotekene er klassen konstruert som en "template". Dvs. at klassen er parametrisert slik at den kan ha elementer av en generisk type. Dersom man skal opprette et objekt av denne klassen må man derfor først instansiere klassen med en parameterklasse. Det er først ved selve instansieringen at det skjer noen form for sjekk av klassen som er parameter: Template klassen er allerede kompilert; og det blir opprettet en spesialversjon av kompilator/linker. Under denne operasjonen undersøker kompilatoren om funksjonene som på den generiske typen er definert i parameterklassen. Dette er også grunnen til at jeg har valgt å la være å definere en abstrakt superklasse: Det vil ikke være mulig å håndheve at klassene som brukes som parameter til ecelement er subklasse av en gitt klasse. Kompilatoren og linkeren sørger dessuten selv for å sjekke om denne klassen har implementert de nødvendige funksjonene og operatorene.

## 7.3 Kjøretid og effektivitet

Effektiviteten i implementasjonen er svært avhengig av hastigheten i de underliggende lagene. Kjøring av gprof (se metodekapittel) avslørte at forholdet mellom systemkall (f.eks. minneallokering og deallokering) og kall på rutiner i implementasjonen var så mye som 52:1. Tidsmessig brukte system så mye som $90\%$ av tiden på minnehåndtering som følge av at de fleste operasjonene returnerer et nytt objekt.

Ved anvendelse av et objektorientert språk til implementasjon av ekkryptering vil det derfor være nødvendig å gjøre noen justeringer: Man bør unngå å konstruere objekter under punktaritmetikk. Dette gjelder både i det underliggende matematikkbiblioteket, de underliggende kroppene

og i ecelement-klassen.

Jeg har ikke foretatt noen store målinger av hastigheten bortsett fra å se på forholdstallene til de forskjellige funksjonskallene og en liten sammenligning i hastigheten for den tyngste operasjonen[2]. Hastigheten til operasjonene på en kurve påvirkes i stor grad av kompleksiteten til operasjonene i den underliggende kroppen. Det er derfor et mål å minimalisere antall ganger de tyngste operasjonene må utføres. Her må man m.a.o. ta hensyn til hvilke operasjoner som utføres i den underliggende kroppen når man velger en optimalisering.

### 7.3.1 Kurver over $\mathbb{F}_p$

Hastigheten til operasjonene i $\mathbb{F}_p$ er avhengig av registerstørrelsen på prosessoren man anvender. Generelt kan man si at hastigheten i maskinvare er $\mathcal{O}(1)$, dvs. hastigheten er konstant uavhengig av størrelsen på tallene man opererer med[3]. I software vil man derimot ha en annen situasjon: Registerstørrelsen vil i stor grad påvirke hastigheten. Det finnes dog flere metoder for likevel å sikre at operasjonene går raskt:

Ved å anvende flere algoritmer som utvelges avhengig av størrelsen på tallene man opererer garanterer GMP asymptotisk vekst på en rekke tunge operasjoner. Det anvendes f.eks. fire forskjellige algoritmer for multiplikasjon. Enkelte av disse krever f.eks. forhåndsberegninger som gjør at de er uegnet til operasjoner på små tall, mens de likevel har en garantert asymptotisk vekst når tallene blir store nok. Tilsvarende vil man ha algoritmer som kun er effektive på små tall.

### 7.3.2 Aritmetikk på kurver over kropper av jevn karakteristikk

For kurver av jevn karakteristikk er det derimot stor forskjell på hvor tunge de forskjellige operasjonene er. Multiplikasjon, invers ($a = a^{-1}$) og addisjon av elementer har forskjellig kostnad avhengig av hvilken basis

[2]Begrunnelse finnes i beskrivelsen av kompleksitetsklasser og i metodekapitlet.

[3]Dette forutsetter at man anvender en RISK-prosessor.

man har valgt. Multiplikasjon over $\mathbb{F}_{2^m}$ i en normalbasis har kompleksitet $\mathcal{O}(m)$. Selv om dette i seg selv regnes som en polynomiell algoritme, er det svært minne- og tidkrevende selv med moderate verdier av $m$ da operasjonen krever $m^2$-minne. Dette plasserer problemet i **SPACE**$(m^2)$ hvilket betyr at algoritmen er i klassen $PSPACE$ (se avsnitt 3.1.2). Algoritmen kan løses med konstant bruk av minnet, men vil da kreve $m^2 \log m$ beregninger (jfr. forholdet mellom $SPACE$ og $TIME$ kompleksitetsklassene [Pap95]). Multiplikasjon kan implementeres i maskinvare, men vil innebære en relativt stor krets (jfr. $SPACE$-kompleksiteten). Implementasjon av disse rutinene i et multipresisjonsbibliotek kan være relativt tungvindt ettersom det innebærer bitmanipulasjon på lavnivå. Dette er dessverre ikke støttet av alle multipresisjonsbiblioteker. (Se forøvrig avsnitt 7.1).

#### 7.3.2.1 Normalbasis

Ved bruk av normalbasis har man fordelen at kvadrering ($a = a^2$) og invers kan utføres ved rotasjon av bit'ene. Dette er en svært enkel operasjon i maskinvare, og går relativt raskt i software. Hastigheten til operasjonen er selvfølgelig avhengig av ord-størrelsen til maskinen den utføres på, men vil kunne utføres i $\mathcal{O}(m)$.

Addisjon fungerer med XOR ("Exclusive OR"), og går også svært raskt både i maskin- og software. I maskinvare vil man få en konstant hastighet mens hastigheten i software vil være avhengig av registerstørrelsen.

#### 7.3.2.2 Polynomiell basis

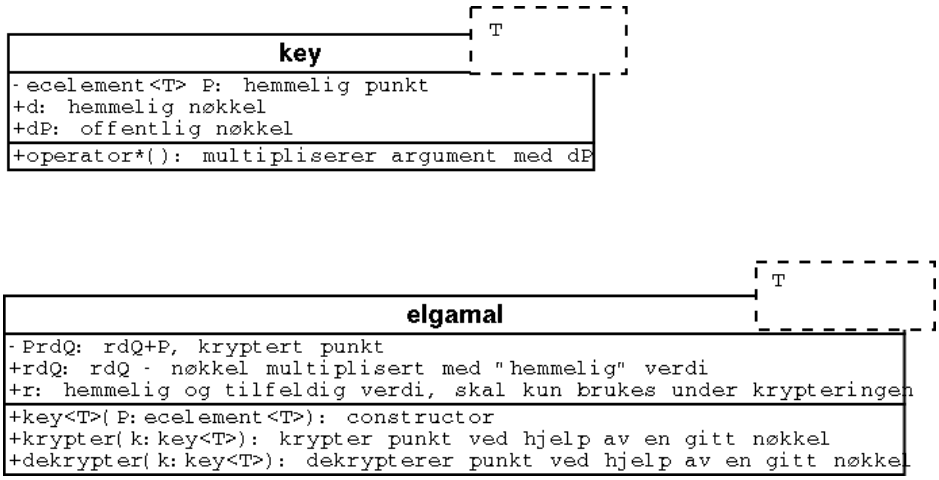
Ved valg av polynomiell basis kan vi fremdeles benytte oss av XOR for addisjon av elementer (se [Ros99, LD, BSS99]). Multiplikasjonsalgoritmen er relativ enkel, men krever dog en reduksjonsalgoritme. Begge disse algoritmene er relativt enkle å implementere. Addisjonsalgoritmen har orden $\mathcal{O}(m)$ og reduksjonsalgoritmen har orden $\mathcal{O}(m^2)$.[4] Kvadrering og in-

[4]M.Brown m.fl (se [BHLM01]) har diskutert flere algoritmer for rask reduksjon, men disse høyt spesialiserte. Disse kan selvfølgelig brukes dersom man ønsker å bruke en av NIST-kurvene, men de er selvfølgelig uinteressante for en generell implementasjon som i denne oppgaven.

vertering av elementer har kompleksitet $\mathcal{O}(m)$ men benytter seg også av modulær reduksjon. Av denne grunn har jeg valgt å benytte meg av en normalbasis.

## 7.4 Beskrivelse av implementasjon av EK primitiver

Klassen ecelement har et enkelt grensesnitt: Man setter karakteristikk og eventuell eksponent ved å kalle på funksjonen $setbasis()$; Deretter kan man sette $a$ og $b$ (dvs. $a_2$ og $a_6$, se 4.2) med funksjonene $setA()$ og $setB()$. $a, b$ og $karakteristikk$ er statiske medlemmer i objektet. Dvs. at de er felles for alle objekter som opprettes. Dette medfører at man kun behøver å sette disse verdiene én gang. Når domeneparametrene er satt opp kan man fritt gjøre aritmetikk med punkter. Implementasjonen inneholder skalarmultiplikasjon og punktaddisjon. (Se forøvrig eget avsnitt om skalarmultiplikasjon lengre ut i dette kapitlet). Det er åpenbart at implementasjon av krypteringsalgoritmer på toppen av dette er svært enkelt. Som eksempel kan brukes ElGamal for elliptiske kurver (se avsnitt 4.5.2).




Figur 7.2: UML-diagram over ElGamal-implementasjon

Eksempel på anvendelse av ElGamal-klassene og ecelement-klassen finnes i avsnitt A.7. Som det går frem av eksempelet blir kryptering svært

enkelt for en bruker når implementasjonsdetaljene skjules. Dette gjelder for så vidt også for implementasjon av diverse algoritmer på toppen av ecelement-klassen.

### 7.4.1 Konverteringsalgoritmer

For å lagre informasjon om punkter/kurver samt overføring av data trenger vi et fast format. IEEE-P1363-standarden har spesifisert en rekke funksjoner for å konvertere punkter og domeneparametre frem og tilbake mellom et fast lagringsformat. Standarden spesifiserer at all data skal lagres som "oktetter".

Oktettene lagres slik:

$$x = x_{l-1}256^{l-1} + x_{l-2}256^{l-2} + \ldots + x_l 256 + x_0$$

hvor $0 \leq x_i < 256$

Implementasjonen i denne oppgaven avviker noe fra denne fremstillingen (som er hentet fra [Sec99]). Dette er gjort da de fleste konverteringer skjer ved konstruksjon av objekter. Ved f.eks. å opprette et koordinat med en tekststreng blir automatisk riktig rutine kalt. Tilsvarende vil riktig rutine bli kalt dersom man bruker objektet i en kontekst som krever konvertering til et annet format. Originalnavnene er tatt med for å gjøre det enklere å skille mellom de forskjellige operasjonene.

**I2OSP** (Integer to Octet String Conversion Primitive) brukes for å konvertere et heltall til en oktett. Denne kan brukes til å lagre domeneparametre.

**Algoritme 19** I2OSP

INPUT: En positivt heltall $a = a_{mlen-1}2^{8(mlen-1)} + a_{mlen-2}2^{8(mlen-2)} + \ldots x_1 2^8 0 x_0$ sammen med ønsket oktettlengde $mlen$
OUTPUT: Oktettstreng $M = M_0 M_1 \ldots M_{mlen-1}$ av lengde $mlen$ oktetter.
**for** $i \leftarrow 0 to (mlen - 1)$ **do**
  $M_0 \leftarrow a_{mlen-1-i}$
**end for**
Return($M$)

**OS2IP** (Octet String to Integer Conversion Primitive) brukes for å konvertere en oktett til et heltall og er inversen av funksjonen ovenfor.

**Algoritme 20** OS2IP

INPUT: En oktettstreng $M$ av lengde $mlen$ oktetter
OUTPUT: Heltall $x$
$x \leftarrow 0$
**for** $i \leftarrow 0 to (mlen - 1)$ **do**
  $x \leftarrow x + 2^{8(mlen-1-i)} M_i$
**end for**
Return($x$)

**FE2OSP** (Field Element to Octet String Conversion Primitive) brukes for å konvertere et punkt til en oktett.

**Algoritme 21** FE2OSP

INPUT: $a \in \mathbb{F}_q$, $a = a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$
OUTPUT: Oktettstreng $M = M_0 M_1 \ldots M_{mlen-1}$ av lengde $mlen = \lceil log_2 q/8 \rceil$ oktetter
**if** $q = p, p > 2$ **then**
  Return(I2OSP(a,mlen))
**end if**
**if** $q = 2^m$ **then**
  **for** $0 < i \leq mlen - 1$ **do**
    $M_i \leftarrow a_7 + 8(mlen - 1 - i) a_6 + 8(mlen - 1 - i) \ldots a_8 (mlen - 1 - i)$
  **end for**
  $M_0 \leftarrow 00_{16} a_{m-1} a_{m-2} \ldots a_{8(mlen)-8}$
  Return($M$)
**end if**

**OS2FEP** (Octet String to Field Element Primitive) er inversen av funksjonen ovenfor.

**ECP2OSP** (Elliptic Curve Point to Octet String Primitive): For lagring av punkter på en datafil må vi konvertere punktene:

**Algoritme 22** OS2FEP

INPUT: Domeneparametrene $T$ samt en oktettstreng $M$
OUTPUT: Et element $a \in \mathbb{F}_q$
**if** $q = p, p > 2$ **then**
  Return($a \leftarrow OS2IP(M)$)
**else**
  /* $q = 2^m$ */
  **for** $i \leftarrow 0 to (m-1)$ **do**
    $a_i \leftarrow M_{mlen-1-\lfloor i/8 \rfloor}^{7-i+8(\lfloor i/8 \rfloor)}$
  **end for**
**end if**

**Algoritme 23** ECP2OSP

INPUT: Et punkt $P = (x_p, y_p)$ på kurven samt domeneparametre $a, b$
OUTPUT: Oktetstreng $M$ av lengde $mlen = 1$ dersom $P = \mathcal{O}_\infty, mlen = \lceil (log_2 q)/8 \rceil + 1$ dersom $P \neq \mathcal{O}_\infty$.
**if** $P = \mathcal{O}_\infty$ **then**
  Return $M = 00_{16}$
**end if**
**if** $P = (x_p, y_p) \neq \mathcal{O}_\infty$ **then**
  $X \leftarrow FE2OSP(x_p, a, b, mlen)$
**end if**
$X \leftarrow FE2OSP(x_p)$
$Y \leftarrow FE2OSP(y_p)$
Return($M \leftarrow 04_{16} || X || Y$)

**OS2ECP** (Octet String to Elliptic Curve Point Primitive): Denne algoritmen er inversen av algoritmen ovenfor og brukes til å konvertere en datastrøm til punkter.

**Algoritme 24** OS2ECP

INPUT: En elliptisk kurve over $\mathbb{F}_q$ definert ved elementene $a, b$ samt en oktettstreng.
OUTPUT: Et punkt $P$
**if** $M = 00_{16}$ **then**
  Return($P \leftarrow \mathcal{O}_\infty$)
**end if**
Splitt opp $M$ i $W||X||Y$ hvor $W$ er den første oktetten etterfulgt av $X$ og $Y$ (begge av lengde $\lceil (log_2 q)/8 \rceil$)
**if** $W \neq 04_{16}$ **then**
  Return $''Feil''$
**end if**
$x_p \leftarrow OS2FE(X)$
$y_p \leftarrow OS2FE(Y)$
Return($P \leftarrow (x_p, y_p)$)

## 7.5 Skalar multiplikasjon

Implementasjon av primitiver som ECDH, ECDSA og ECAES krever skalar multiplikasjon. Dvs.

$$Q = kP = \underbrace{P + P + \ldots + P}_{k\text{ganger}}$$

hvor $P \in E, k \in [1, \#P]$. $P$ kan enten være et fast punkt som genererer en stor undergruppe på $E(\mathbb{F}_q)$, eller $P$ kan selv være et tilfeldig element i en slik undergruppe. For å velge ut et slikt punkt kan man benytte seg av en kjent kurve hvor ordenen til generatoren er kjent.

Ved simuleringer var det i særdeleshet skalarmultiplikasjon som utpekte seg som den tyngste operasjonen. Dette gjaldt både for kurver over

$\mathbb{F}_{2^m}$ og $\mathbb{F}_p$.[5] Det er derfor viktig å finne en effektiv metode for å optimalisere denne operasjonen.

### 7.5.1 "Double-and-add" algoritmen

Det finnes mange algoritmer for rask eksponensiering i en multiplikativ gruppe som kan modifiseres for bruk på en elliptisk kurve (se f.eks. [MvOV97]). Det finnes en god oversikt i [LD]. En del av disse fungerer kun på spesielle kurver.

For implementasjon av skalar multiplikasjon er den enkleste effektive metoden "double-and-add"[MRS01]:

**Algoritme 25** "Double-and-add"

INPUT: Et punkt $P$, et heltall $k$, $n = lg_2(k)$
OUTPUT: $R = kP$
**if** $k_0 = 0$ **then**
  $R \leftarrow P$
**else**
  $R \leftarrow 2P$
**end if**
**for** $i = 1$ to $n$ **do**
  $P \leftarrow 2P$
  **if** $k_i = 1$ **then**
    $R \leftarrow R + P$
  **end if**
**end for**
Return($R$)

Denne algoritmen krever $w(k) - 1$ addisjoner (hvor $w(k)$ er "vekten" av $k$, dvs. antall bits som er satt) og $n - 1$ kvadreringer. Denne algoritmen kalles også for "binærmetoden". Denne algoritmen bruker gjennomsnitlig $8\frac{1}{3}\log_2 k$ multiplikasjoner og $\frac{4}{3}\log_2 k$ inversjoner jfr. [LD99]. Som det går frem av avsnitt 4.3.3 består addisjonsalgoritmen for punkter på kurver over $\mathbb{F}_p$ av addisjon og multiplikasjoner ($\mathrm{mod}\ p$). Ettersom ingen

[5] Det var dog (ikke uventet p.g.a. kompleksiteten til operasjonene i de underliggende kroppene) *store* forskjeller i simuleringstid (se forøvrig avsnitt 7.6).

av disse operasjonene er spesielt krevende for kurver av odde karakteristikk vil "double-and-add" algoritmen ovenfor være en god optimering dersom $k$ ikke er for stor. I implementasjonen i denne oppgaven benyttes "double-and-add" dersom $w(k) \leq 64$, ellers benyttes "addisjon-substraksjons"-algoritmen beskrevet nedenfor. Formålet med dette er å garantere en asymptotisk vekst i ordenen til skalarmultiplikasjonsalgoritmen. NAF-Window-algoritmen krever en forhåndsberegning. Tiden dette tar kan overskride tiden "double-and-add" algoritmen bruker for små verdier av $k$. Dette er tanken bak å benytte seg av to algoritmer.

### 7.5.2 NAF-representasjon

I tillegg til "double-and-add"-algoritmen ovenfor er det mulig å redusere antall addisjoner ved å bruke en "Signed Digit (SD)" representasjon av $k$. Den minimale SD-representasjonen av $k$ kalles for en "non adjacent form" (NAF)[BSS99]). NAF har lengden $l/3$ hvor $l = w(k)$ som sammen med spesialkonstruert algoritmer vil gi en god effekt dersom $w(k)$ er stor nok. Formålet med dette er som med double-and-add algoritmen ovenfor å redusere antall kurveoperasjoner. Ettersom tiden det tar å kalkulere $NAF(k)$ er neglisjerbar i forhold til de andre operasjonene[MRS01], vil man kunne spare mye tid.

**Algoritme 26** Algoritme for å beregne NAF [BSS99]

INPUT: Heltall $k = \sum_{j=0}^{l-1} k_j 2^j, k_j \in \{0, 1\}$
OUTPUT: $NAF(k) = \sum_{j=0}^{l} s_j 2^j, s_j \in \{-1, 0, 1\}$
$c_0 \leftarrow 0$
**for** $j = 0$ to $l$ **do**
  $c_{j+1} \leftarrow \lfloor (k_j + k_{j+1} + c_j)/2 \rfloor$
  $s_j \leftarrow k_j + c_j - 2c_{j+1}$
**end for**
Return($s_l s_{l-1} \ldots s_0$)

#### 7.5.2.1 Addisjon-substraksjons algoritmen

Vi kan nå bruke en analog av "double-and-add"-metoden som kalles "addisjon-substraksjons"-metoden. Denne algoritmen trenger $l$ kvadreringer og $l/3$ addisjoner. Dette tilsvarer en forbedring på $14\%$ i forhold til "double-and-add"-metoden [LD].

**Algoritme 27** Addisjon-substraksjons metoden

INPUT: Et heltall $k$ og et punkt $P = (x, y) \in E(\mathbb{F}_q)$
OUTPUT: $Q = kP \in E(\mathbb{F}_q)$
$j = NAF(k)$
$Q \leftarrow \mathcal{O}_\infty$
**for** $j$ from $l - 1$ downto $0$ **do**
  $Q \leftarrow 2Q$
  **if** $u_j = 1$ **then**
    $Q \leftarrow Q + P$
  **end if**
  **if** $u_j = -1$ **then**
    $Q \leftarrow Q - P$
  **end if**
**end for**
Return($Q$)

#### 7.5.2.2 "Width-w window" algoritmen

Det finnes også flere metoder som benytter seg av NAF. Blant disse er "vindus"-metoden[6] en av de mest kjente. Denne algoritmen reduserer antall addisjoner mens antall kvadreringer er den samme som for "addisjon-substraksjons"-metoden[LD]. Ettersom addisjoner innebærer multiplikasjon i $\mathbb{F}_2$ kan dette gi en god gevinst: Hastigheten er avhengig av kostnadsforholdet mellom inversjon og multiplikasjon og hvorvidt man bruker affine eller projektive koordinater (se [BSS99]). I denne oppgaven har jeg valgt å bruke affine koordinater p.g.a. enklere implementasjon av aritmetikken.

[6] "Width-w window method", se [BSS99] side 69-70 og [LD]

### 7.5.3 Spesialtilpassede algoritmer

Som allerede nevn stilles det store krav til effektivitet ved skalarmultiplikasjon. Det har derfor det blitt utviklet flere algoritmer som forsøker å unngå å bruke de tyngste operasjonene. Disse algoritmene er derfor som regel høyt spesialiserte, og enkelte av dem fungerer kun med en gitt basis og med avgrensede domeneparametre. Eksempler på dette er [MRS01] hvor en algoritme som bare fungerer på Kobliz-kurver[7] presenteres. Man skal være klar over at en del av egenskapene som utnyttes for å oppnå bedre effektivitet i disse algoritmene er de samme som brukes ved angrep. Flere av disse kurvene er man frarådet fra å bruke (se avsnitt 5 for en oversikt over "svake" kurver).

## 7.6 Simuleringsresultat

Som allerede nevnt går det med mye tid til minnehåndtering i implementasjonen min. Dette gjelder også for kurver over kropper av jevn karakteristikk. Simuleringene ble foretatt med NIST-kurver (se [oCIoST00]). Simuleringsprogrammet finnes i avsnitt A.7.

| Underliggende kropp | Minnehåndtering | Kroppoperasjoner |
|---|---|---|
| $\mathbb{F}_p$ | $0.06 * 10^{-3}s$ | $0.01 * 10^{-3}s$ |
| $\mathbb{F}_{2^m}$ | $0.13 * 10^{-3}s$ | $2.5 * 10^{-3}s$ |

Tabell 7.1: Tidsforbruk per skalarmultiplikasjon i hhv. $fo$ og $fp$.

## 7.7 Smartkort-implementasjoner av EK-kryptering

Et av de viktigste spørsmålene ved implementasjon på smartkort er hvorvidt man trenger spesialkonstruerte smartkort for å utføre beregningene. I lys av diskusjonen i avsnitt 7.5 og beskrivelsen av aritmetikken i kapittel 4 er det innlysende at operasjoner på $E(\mathbb{F}_{2^m})$ er vesentlig tyngre enn på $E(\mathbb{F}_p)$. Dette innebærer at man kan anvende et "standard" smartkort med

[7] Kobliz-kurver kalles også "anomale" kurver[LD]

innebygget støtte for multipresisjonsaritmetikk for kurver av odde karakteristikk. Dette vil være en stor besparelse da spesiallagede smartkort kan være kostbare. Det er også mulig å implementere algoritmer for multiplikasjon på elementer i $\mathbb{F}_{2^m}$, men dette er såpass plasskrevende (p.g.a. at man må generere alle permutasjoner av bitmønstret til et av punktene, se 4.2.2) at det praktisk talt blir umulig å få til uten et spesialkonstruert smartkort. Dette er m.a.o. tungtveiende grunner til å velge kurver over $\mathbb{F}_p$ ved en implementasjon både på smartkort og i vanlig software.

### 7.7.1 Montgomerys algoritme

Et av de viktigste hensyn man må ta ved implementasjon på smartkort er å redusere register/minnebruken. Minnet er som regel svært begrenset på disse kortene (ca. 16k som også brukes av de installerte nøklene/programmene). Med tanke på dette har det dukket opp flere algoritmer som er designet for å minimalisere minnebruk. Montgomery ([LD99, BSS99] har laget en algoritme som minimaliserer register/minnebruken. Ulempen med denne algoritmen er at den ikke reduserer antall multiplikasjoner, som er en tidkrevende prosess for et smartkort med begrenset regnekraft. Dette gjelder også for et evt. spesialdesignet smartkort. López m.fl. [LD99] har presenteret en variant av Montgomerys algoritme som bruker $2\lfloor \log_2 k \rfloor + 4$ multiplikasjoner og $4\lfloor \log_2 k \rfloor + 1$ inversjoner. Dette vil dette føre til en stor økning i hastigheten jfr. hastighetsbereginger i [LD99], og vil derfor være helt nødvendig dersom man skal anvende et smartkort.

For å unngå at alle nøkler (og tilhørende applikasjoner) på smartkortet skal måtte ha en kopi av de samme algoritmene vil det være nødvendig å lage en spesialversjon av operativsystemet som inneholder algoritmer for punktaddisjon og skalarmultiplikasjon. Dette burde ikke ta stor plass, og kan eksempelvis erstatte rutiner for RSA-kryptering som finnes i flere av disse operativsystemene. Man kan selvfølgelig ha støtte for både RSA og EK-kryptering dersom man må ha nøkler av begge typer på kortet, men da blir hele poenget med plassbesparelser borte.

**Algoritme 28** Montgomery skalarmultiplikasjon

INPUT: Et heltall $k = (k_{l-1}, \ldots k_1 k_0), k \geq 0$ og et punkt $P = (x, y) \in E$
OUTPUT: $Q = kP$
**if** $k = 0$ or $x = 0$ **then**
  Return($Q = \mathcal{O}_\infty$)
**end if**
$x_1 \leftarrow x$
$x_2 \leftarrow x^2 + b/x^2$
**for** $i = l - 2$ downto $0$ **do**
  $t \leftarrow \frac{x_1}{x_1 0 x_2}$
  **if** $k_i = 1$ **then**
    $x_1 \leftarrow x + t^2 + t$
    $x_2 \leftarrow x_2^2 + b/x_2^2$
  **else**
    $x_1 \leftarrow x_1^2 + b/x_1^2$
    $x_2 \leftarrow x + t^2 + t$
  **end if**
**end for**
$r_1 \leftarrow x_1 + x$
$r_2 \leftarrow x_2 + x$
$y_1 \leftarrow r_2(r_1 r_2 + x^2 + y)/x + y$
Return($Q \leftarrow (x_1, y_1)$)

### 7.7.2 Dedikert eller standarisert maskinvare

Som nevnt i de forutgående avsnittene er operasjoner i $\mathbb{F}_{2^m}$ tung i software. Dersom man ønsker å anvende underliggende kropper av jevn karakteristikk vil det derfor være påkrevet å ha effektiv og tilpasset maskinvare. Dette gjelder selv for enkle operasjoner som signering av en hash, som er den typiske operasjonen som gjøres i et smartkort. For verifikasjon av signaturer får man det samme problemet: Dersom en kortutsteder, som f.eks. en bank, skal verifisere alle transaksjoner fortløpende vil dette også kreve spesialisert maskinvare.

Som allerede nevnt har multiplikasjon over $\mathbb{F}_{2^m}$ en rom-kompleksitet $SPACE(m^2)$. Dette vil også gjelde i maskinvare. Dette innebærer at størrelsen på kretsen i en spesialdesignet krets vil vokse kvadratisk med nøkkelstørrelsen.

Kostnadene ved å anvende kort med en egen prosessor for multipresisjonsaritmetikk i forhold til kostnadene med å produsere et spesialdesignet kort vil være avgjørende for hva som blir standarden. Pr. dags dato er det RSA-kryptering (se avsnitt 3.4) som anvendes på smartkortene. Som det allerede er blitt antydet ovenfor er det mulig å implementere EK-kryptering på et av disse smartkortene dersom man velger å benytte en underliggende kropp av odde karakteristikk. Det er derfor tenkelig at det kan produseres smartkort med støtte for både RSA- og EK-kryptering. Dette vil gjøre det mulig å gradvis utfase allerede eksisterende RSA-nøkkelhierarkier etterhvert som EK-kryptering blir mer og mer vanlig. Det vil også være enklere for brukeren da begge typer nøkler kan lagres på det samme kortet.

# 8

# Konklusjon

## 8.1 Oppsummering

Jeg har i denne oppgaven sett på hvilke hensyn som bør tas ved en softwareimplementasjon av elliptisk kurve krypteringssystemer.

For å underbygge argumentasjonen ble kompleksitetsteori presentert i kapittel 3.

I kapittel 4 ble teorien for elliptiske kurver, nøkkellengder, antall punkter på en kurve og gruppestruktur på kurvene presentert og diskutert. Konklusjonene som ble tatt var:

- Det må være mange nok punkter på kurven til at det diskrete logaritmeproblemet ikke kan løses i polynomisk tid/rom
- Kurver hvor antall punkter er kjent (som f.eks. NIST-kurvene) gir god nok sikkerhet selv om alle parametre om dem er kjent (se [BHLM01] samt avsnitt 4.4.1).

Deretter ble angrepsmetoder på elliptiske kurver og det diskret logaritmeproblemet presentert i kapittel 5, og på bakgrunn av dette ble det satt ytterlige krav til kurvene:

- Supersingulære og anomale kurver bør unngås.
- Kurveordenen må bestå av minst en stor primfaktor

I kapittel 7 ble algoritmer for effektiv skalarmultiplikasjon presentert. Her ble det også diskutert hvorvidt implementasjon av kurver over $\mathbb{F}_{2^m}$ og $\mathbb{F}_p$ lar seg gjøre på smartkort og hvilke hensyn som må tas ved en slik implementasjon.

## 8.2 Konklusjon

I lys av kompleksiteten av en del av operasjonene er det åpenbart at en software implementasjon, det være seg i smartkort eller på en vanlig datamaskin, bør bruke kurver over $\mathbb{F}_p$ av hensyn til effektiviteten: Softwareimplementasjon av aritmetikk i $\mathbb{F}_{2^m}$ (spesielt multiplikasjon) krever mye minne dersom man ønsker effektivitet. En del av aritmetikken er også *tung* for ikke spesialisert maskinvare.

I lys av diskusjonen i avsnitt 4.4.1 kan man trygt velge en kurve hvor domeneparametrene er kjente uten at sikkerheten påvirkes i noen særlig grad. Implementasjon av elliptisk kurve aritmetikk over $\mathbb{F}_{2^m}$ i maskinvare skulle ikke by på noen problemer, men noen stor hastighetsgevinst i forhold til å bruke en vanlig prosessor med støtte for multipresisjonsaritmetikk på kurver over $\mathbb{F}_p$ er ikke å forvente da de aritmetiske operasjonene går svært raskt i software. Ved implementasjon på smartkort må man dog bruke algoritmer som reduserer register og minnebruken da dette er en ressurs som det er stor mangel på.[1] Eksempel på dette er Montgomerys algoritme som finnes i avsnitt 7.7.1.

Anvendelsen av C++ for å implementere ek-kryptering avslørte at man må ta spesielle hensyn for å oppnå effektivitet da mesteparten av kjøretiden til funksjonene bestod i minnehåndtering p.g.a. objektgenerering.

[1] Smartkort har som nevnt innledningsvis typisk 16k med minne som skal anvendes både til programmer og nøkler.

# 9

# Referanseliste

## Referanser


[Ber99] D.J. Bernstein, *Integer multiplication benchmarks*, Tech. report, ftp://koobera.math.uic.edu/www/speed/mult.html, November 1999.

[BHLM01] Michael Brown, Darrel Hankerson, Julio Lopez, and Alfred Menezes, *Software implementation of the NIST elliptic curves over prime fields*, CT-RSA, 2001, pp. 250–265.

[BK98] R. Balasubramanian and N. Koblitz, *The improbability that an elliptic curve has a sub-exponential discrete log problem under the menezes-okamoto-vanstone algorithm*, no. 11, Springer-Verlag, New York, Inc., 1998.

[BSS99] Ian Blake, Gadiel Seroussi, and Nigel Smart, *Elliptic curves in cryptography*, London Mathematical Society Lecture Note Series, no. 265, Cambridge University Press, 1999.

[Coc73] C. C. Cocks, *A note on "non-secret encryption"*, november 1973.

[Com98] "Ansi X9.63 Committee", *Ansi x9.63 – public key cryptography for the financial services industry: Elliptic curve key agreement and key transport schemes*, ANSI X9.63 Working Draft - Version 2.0, July 1998.

[DH76] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, 1976.

[Ell70] J.h. Ellis, *The possibility of secure non-secret digital encryption*, januar 1970.

[FGH00] M. Fouquet, P. Gaudry, and R. Harley, *An extension of satoh's algorithm and its implementation.*, J. Ramanujan Math. Soc. **15** (2000), 281–318.

[GB96] Shafi Goldwasser and Mihir Bellare, *Lecture notes on cryptography*, http://theory.lcs.mit.edu/ shafi, juli 1996.

[GJ79] Michael R. Garey and David S. Johnson, *Computers and intractability. a guide to the theory of np-completeness*, W. H. Freeman and Company, 1979.

[Hov99] Asbjørn Hovstø, *Arkitektur for digitale sertifikater*, Seminar om kommunikasjonssikkerhet i regi av NSF, 7-8 januar 1999.

[HSSI99] Ryuichi Harasawa, Junji Shikata, Joe Suzuki, and Hideki Imai, *Comparing the mov and fr reductions in elliptic curve cryptography*, IEICE Trans. on Foundamentals A, Vol. 82-A, No. 8, August 1999.

[IEE01] IEEE Standards Department, *Standard specifications for public key cryptography.*, IEEE, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA, d10 ed., 2001.

[JM97] Don B. Johnson and Alfred J. Menezes, *Elliptic curve dsa (ecdsa): An enhanced dsa*, 1997.

[KBC97] H. Krawczyk, M. Bellare, and R. Cannetti, RFC 2104, Februar 1997.

[Kha67] David Khan, *The codebreakers. the story of secret writing*, revised edition, 1996 ed., Scribner. 1230 Avenue of the Americas. New York, NY10020, 1967.

[Kob87] N. Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation 48, Mathematics of Computation 48, 1987, pp. 203-209.

[LD99] Julio Lopez and Ricardo Dahab, *Fast multiplication on elliptic curves over $\mathbb{F}_{2^m}$ without precomputation*, Cryptographic Hardware and Embedded Systems, no. Generators, 1999, pp. 316–327.

[LD] Julio López and Ricardo Dahab, *An overview of elliptic curve cryptography*.

[LV00] Arjen K. Lenstra and Eric R. Verheul, *Selecting cryptographic key sizes*, Public Key Cryptography, 2000, pp. 446–465.

[Men93] Alfred J. Menezes, *Elliptic curve public key cryptosystems*, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, 1993.

[Men95] Alfred Menezes, *Elliptic curve cryptosystems*, CryptoBytes **1** (1995), no. 2, 1–4.

[Mil86] V.S Miller, *Use of elliptic curves in cryptography*, Advances in Cryptology – CRYPTO '85 Proceedings, Springer-Verlag, 1986.

[MRS01] Tommi Meskanen, Ari Renvall, and Paula Steinby, *Efficient scalar multiplication of elliptic curves*, mars 2001.

[MVO91] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto, *Reducing elliptic curve logarithms to logarithms in a finite field*, ACM (1991), 80–89.

[MvOV97] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.

[oCIoST00] U.S. Department of Commerce/National Institute of Standards and Technology, *Digital signature standard (dss), fips pub 186-2*, februar 2000.

[Pap95] Christos H. Papadimitriou, *Computational complexity*, Addison- Wesley Publishing Company, 1994,1995.

[PH94] David A. Patterson and John L Hennessy, *Computer organization & design. the hardware / software interface*, Morgan Kaufmann Publishers, Inc., 1994.

[Pol78] J. Pollard, *Monte carlo methods for index computation mod p*, Mathematics of Computation **32** (1978), 918–924.

[PTVF92] William H. Press, Saus A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in c*, second edition ed., Cambridge University Press, 1992.

[Ros99] Michael Rosing, *Implementing elliptic curve cryptography*, Manning Publications Co., 32 Lafayette Place, Greenwich, CT 06830, 1999.

[RSA77] Ronald L. Rivest, Adi Shamir, and Len Adelman, *On digital signatures and public key cryptosystems, technical memorandum 82*, 1977.

[Sec99] Sec, *Elliptic curve cryptography*, Standards for Efficiency Cryptography Group, September 1999.

[Sti95] Douglas R. Stinson, *Cryptography, theory and practice*, CRC Press Series on Discrete Mathematics and Its Applications, CRC Press inc., 2000 Corporate Blvd., N.W., Boca Raton, Florida 33431, 1995.

[Ukj99] *Elektroniske pasientkort i nork helsetjeneste – vurdering av bruksområder og nasjonale tiltak*, Sosial- og helsedepartementet. Forebyggings- og utviklingsavdelingen, Desember 1999.

[Vau98] S. Vaudenay, *Cryptanalysis of the chor-rivest cryptosystem*, Advances in Cryptology – CRYPTO '98, Lecture Notes in Computer Science, vol. 1462, Springer-Verlag, 1998, pp. pp. 243–256.

[vOW94] Paul C. van Oorschot and Michael J. Wiener, *Parallel collision search with application to hash functions and discrete logarithms*, Proceedings of the 2nd ACM Conference on Computer and communications security, pages 210-218, CCS '94, 1994.

[Wei95] Mark Allen Weiss, *Data structures and algorithm analysis*, The Benjamin/Cummings Publishing Company, inc., 1995.

[WZ98] Michael J. Wiener and Robert J. Zuccherato, *Faster attacks on elliptic curve cryptosystems*, Tech. report, Entrust Technologies, 750 Heron Road, Ottawa, Ontario. Canada KIV 1A7, april 1998.

# A

# Kildekode

## A.1 Makefile

```
# $Id: Makefile,v 1.6 2002/04/19 11:39:00 andreasd Exp $

CXX=g++

CXXFLAGS=-fexceptions -ggdb -Wall -iso -pedantic\
        -fpic -pipe -finline-functions -funroll-loops \
        -I/usr/local/include -ggdb -pg

DEST1=gmptest
DEST2=libgmpwrapper.a
DEST3=libgmpwrapper.so.1
DEST4=libmlibwrapper.a
DESTOBJ4=mlib.o mlibinst.o
DESTS=$(DEST1) $(DEST2) ectest elgamaltest test_f2 test_fp
DESTOBJ2=gmpwrapper.o ecinst.o
DESTOBJ1=gmptest.o
LFLAGS=-L. -L/usr/local/lib -pg
LIBS=-lgmpwrapper -lgmp -lmlibwrapper


.SUFFIXES: $(.SUFFIXES) .cc


all: $(DESTS)
```

```
clean:
	rm -f *~ *.o *.core *.gmon $(DESTS)


.cc.o:
	$(CXX) -c $(CXXFLAGS) $<


$(DEST1): $(DESTOBJ1) $(DEST2) $(DEST4)
	$(CXX) -o $@ $(DESTOBJ1) $(LFLAGS) $(LIBS)
	strip $@

elgamaltest: elgamaltest.o elgamal.o $(DEST2) $(DEST4)
	$(CXX) -o $@ elgamaltest.o $(LFLAGS) $(LIBS) elgamal.o

test_f2: elgamaltest
	ln -f $> $@

test_fp: elgamaltest
	ln -f $> $@

# $> fungerer bare med bsd-make ser det ut til...
$(DEST2): $(DESTOBJ2)
	ar cru $@ $>
	ranlib $@

$(DEST4): $(DESTOBJ4)
	ar cru $@ $>
	ranlib $@


# for generering av ELF-bibliotek.
```

```
# Biblioteket blir mer enn dobbelt så stort som .a
# biblioteket!
$(DEST3): $(DESTOBJ2)
	gcc –shared –fpic –W,–lsoname=$* –o $@ $^ $(LFLAGS) $(LIBS)
	strip –g $@

#NB: på Solaris er denne switchen (–g) udokumentert
#    Her har man en switch kalt –x
#    Sjekk om denne finnes på FreeBSD

ectest:ectest.o $(DEST2)
	$(CXX) –o $@ ectest.o ecinst.o $(LFLAGS) $(LIBS)


# for å fjerne debug og andre "unødvendige" ting:
#	strip –g $@



#gmpwrapper.o: gmpwrapper.cc gmpwrapper

ectest.o: ectest.cc ec $(DEST2)
ecinst.o: ecinst.cc ec
# DO NOT DELETE
```

## A.2 ec

```
// -*- C++ -*-
//
// $Id: ec,v 1.10 2002/04/28 20:12:48 andreasd Exp $
//

#ifndef __EC__
#define __EC__ 1

#include "gmpwrapper"
#include <utility>

/*
 * Rutiner for elliptisk-kurve kryptografi over GF(p^n)
 *
 */

namespace ec{
  template<class T> class ecelement
  {
    T x,y; // koordinatene til selve punktet
    static ecelement<T> inf;// point at infinity
    static T a, b; // kurven - statisk, dvs. identisk for alle punkter
    static T karakteristikk, kexp; // kexp brukes kun hviss karakteristikk == 2
    static unsigned long kexp_l;// maxbit = høyeste bit hviss karakteristikk == 2
    static int * lambda[2];

  public:
    ecelement(const T&, const T&);
    ecelement(const int&, const int&);
    ecelement(const ecelement<T>&);
    ecelement(void);
```

```
  T& kar(void){
    return karakteristikk;
  }

  unsigned long& karexp(void){
    return kexp_l;
  }

  const T& rnd() const; // hent tilfeldig tall

  void setBasis(const int&);
  void setBasis(const int&, const int&);
  void setBasis(const T&); // overloading for q != 2
  void setBasis(const T&, const T&);
  void setA(const T&);
  void setA(const int&);
  void setB(const T&);
  void setB(const int&);
  const bool operator==(const ecelement<T> &) const;
  //      const bool operator!=(const ecelement<T> &); // avledes av operator==
  const ecelement<T>& operator+(const ecelement<T> &);
  const ecelement<T>& operator+=(const ecelement<T> &);
  const ecelement<T>& operator-() const; // -P
  const ecelement<T>& operator-(const ecelement<T>&);
  void genlambda(void);
  ecelement& operator=(const ecelement<T> &);
  const bool isInf(void) const; // is this the point at infinity ?
  const ecelement<T>& operator*(const T&) const;
  const ecelement<T>& double_and_add(const unsigned int&) const;
  int *naf(const unsigned long&) const;
  const ecelement<T>& add_sub(int *, const int&) const;
  ecelement<T>& finn_punkt(void);
};
```

```
template<class T>
void ecelement<T>::setA(const T& nyA)
{
  a = nyA;
}

template<class T>
void ecelement<T>::setB(const T& nyB)
{
  b = nyB;
}

template<class T>
void ecelement<T>::setA(const int& nyA)
{
  a = nyA;
}

template<class T>
void ecelement<T>::setB(const int& nyB)
{
  b = nyB;
}

template<class T>
const T& ecelement<T>::rnd(void) const
{
  T *retval = new T();
  if(karakteristikk != 2)
    return retval->rnd(karakteristikk);
  else {
```

```
    T *tmp = new T(kexp_l);
    tmp = (T*)&(retval->rnd(*tmp));
    delete tmp;
    return *retval;
  }
}

template<class T>
void ecelement<T>::setBasis(const T& k, const T& ny_kexp)
{ // se IEEE P1363 A9.6
  if(k != 2) {
    // ulovlig verdi
    throw((string)"void ecelement<T>::"
      "setBasis(const T& k, const T& ny_kexp): kalt med ulovlig verdi");
  }
  karakteristikk=k;
  kexp=ny_kexp;
  kexp_l = kexp.get_long();
  inf=ecelement<T>(0,0); // setter inf til et punkt som
                         // *IKKE* ligger på kurven
  genlambda();
}

template<class T>
void ecelement<T>::setBasis(const T& k)
{
  if(k < 3) {
    // ulovlig verdi
    std::cout << "void ecelement<T>::setBasis(const T& k): kalt med ulovl
    std::cout << k;
    std::cout<< std::flush << std::endl;
    exit(-1);
  }
```

```
  karakteristikk=k;
  // inf settes slik det er foreslått i IEEE P1363 A9.6
  inf=(a==0)?ecelement<T>(0,1):ecelement<T>(0,0);
}

template<class T>
void ecelement<T>::setBasis(const int& k)
{
  setBasis(T(k));
}

template<class T>
void ecelement<T>::setBasis(const int& k, const int &ny_kexp)
{
  setBasis(T(k), T(ny_kexp));
}

template<class T>
const bool ecelement<T>::isInf() const
{
  return *this == inf;
}


template <class T>
const bool ecelement<T>::operator==(const ecelement<T> &p2) const
{
  return (x == p2.x && y == p2.y);
}

template<class T>
ecelement<T>& ecelement<T>::operator=(const ecelement<T> &p)
```

```
{
  x = p.x;
  y = p.y;
  return *this;
}



template<class T>
ecelement<T>::ecelement(const T& xx, const T& yy)
{
  x = xx;
  y = yy;
}

template<class T>
ecelement<T>::ecelement(const int& xx, const int& yy)
{
  x = T(xx);
  y = T(yy);
}

template<class T>
ecelement<T>::ecelement(const ecelement<T> &e2)
{
  x = e2.x;
  y = e2.y;
}

template<class T>
ecelement<T>::ecelement()
{
  *this = inf; // udefinert punkt – setter det til inf
```

```
}


template<class T>
const ecelement<T>& ecelement<T>::operator+=(const ecelement<T>& e2)
{
  *this = *this + e2;
  return *this;
}

template<class T>
const ecelement<T>& ecelement<T>::operator+(const ecelement<T>& e2)
{
  T mu;
  ecelement<T> *retval = new ecelement<T>();

  if(this->isInf()){
    //  O_inf + Q
    return e2;
  }
  if(e2.isInf()){
    //  P + O_inf
    return *this;
  }

  // dette skal ikke skje!
  if( y == 0) {
    std::cout<<"y-koordinat == 0" << std::endl << std::flush;
  }


  // plus-operasjonen er forskjellig fra jevn karakteristikk til
  // odde karakteristikk
```

```
if(karakteristikk == 2) { //F_{2^m}
  if(*this == e2) {// P = Q, P+Q = 2P
    mu = x ^ x ^ y.ror(kexp_l);
    // x_3 = μ^2 + μ + a_2
    retval->x = mu.f2mul(mu ^ mu ^ a, kexp_l, lambda);
    //y_3 = μ*(x_1 + x_3) + x_3 + y_1
    retval->y = mu.f2mul(x ^ retval->x, kexp_l, lambda) \
      ^ retval->x ^ y;
  } else { //  P ≠ Q
    if(*this == -e2) {
      //  P + Q = P - P = O_∞
      return inf;
    } else {
      mu = x ^ e2.x; //μ = x_2 + x_1
      mu = mu.ror(kexp_l); //μ = μ^{-1}
      mu*= y ^ e2.y; //  μ = (y_2+y_1)/(x_2+x_1)
      //x_3 = μ^2 + μ + x_1 + x_2 + a_2
      retval->x = mu.f2mul(mu ^ mu ^ x ^ e2.x ^ a,  kexp_l, lambda);
      //y_3 = μ(x_1 + x_3) + x_3 + y_1
      retval->y = mu.f2mul(x ^ retval->x, kexp_l,lambda) \
        ^ retval->x ^ y;
    }
  }
  return *retval;
} else { // odde karakteristikk
  if(x == e2.x) {
    if((y == e2.y) && (y != 0)) {
        return inf;
      //  μ = 3x_1^2 + a_2
      mu = (3 * x * x) + a;
      //  μ = μ/(2*y_1)
      mu /= 2 * y;
      //  μ = μ mod p
```

```
      mu %= karakteristikk;
    } else {
      return inf;
    }
  } else {
    // μ = (y2 − y1)
    mu = (e2.y - y);
    // μ = μ/(x2 − x1)
    mu /= (e2.x - x);
  }
  // x3 = μ^2 − x1 − x2
  retval->x = mu*mu - x - e2.x;
  // x3 = x3 mod p
  retval->x %= karakteristikk;
  // y3 = mu(x1 − x3) − y1
  retval->y = mu*(x - retval->x) - y;
  // y3 = y3 mod p
  retval->y %= karakteristikk;
  return *retval;
 }
}


template<class T>
const ecelement<T>& ecelement<T>::operator-(const ecelement<T> &p2)
{
 ecelement<T> *tmp = new ecelement<T>();
 if(this->isInf()){
  tmp->x = p2.x;
  tmp->y = -p2.y;
  return *tmp;
 }
 if(karakteristikk == 2) { //F_{2^m}
```

```
    //x3 = x2
    tmp->x = p2.x;
    //y3 = x2 + y2
    tmp->y = p2.x ^ p2.y;
    return *tmp + *this;
  } else { // odde karakteristikk
    *tmp = -p2;
    return *this + *tmp;
  }
}

template<class T>
const ecelement<T>& ecelement<T>::operator-() const
{
  ecelement<T> *retval = new ecelement<T>();
  if(this->isInf()) {
    return *this;
  }
  retval->x=x;
  retval->y=-y;
  return *retval;
}


template<class T>
void ecelement<T>::genlambda()
{
  //      unsigned long *tmp = new unsigned long(2*maxbit);
  //      unsigned long *lambda[2] = (unsigned long *[2])tmp;
  int *log2;
  unsigned long mykexp = kexp.get_long() + 1;
  log2 = new int[mykexp];
  lambda[0] = new int[mykexp];
```

```
  lambda[1] = new int[mykexp];
  for(unsigned long i=0;i<mykexp;i++)
    log2[i]= -1;
  int twoexp = 1;
  for(unsigned long i = 0;i<mykexp;i++) {
    log2[twoexp] = i;
    twoexp = (twoexp << 1) % (mykexp + 1);
  }
  unsigned long n = mykexp / 2;
  lambda[0][0] = n;
  for(unsigned long i = 1; i < mykexp; i++) {
    lambda[0][i] = (lambda[0][i-1] + 1) % mykexp;
  }
  lambda[1][0] = -1;
  lambda[1][1] = n;
  lambda[1][n] = 1;
  for(unsigned long i = 2;i<=n;i++){
    int idx = log2[i];
    int log = log2[mykexp - i + 2];
    lambda[1][idx] = log;
    lambda[1][log] = idx;
  }
  lambda[1][log2[n+1]] = log2[n+1];

}



// skalar multiplikasjon, med _store_ tall (2^32>)
//  se avsnitt 7.5.2.1
template<class T>
const ecelement<T>& ecelement<T>::operator*(const T& k) const
{
```

```
size_t num_limbs = k.size();
size_t limbsize = 8*sizeof(unsigned long);

// allokerer _nok_ plass, gidder ikke beregne w(k)
if(num_limbs > 1){ // _mer_ enn 32 bits
  int *ktmp = new int[num_limbs];
  int *s = new int[num_limbs*limbsize];
  int *c = new int[num_limbs*limbsize];
  ecelement<T> *retval;
  T tmp = k;
  for(unsigned int i = 0;i < num_limbs; i++) {
    ktmp[i] = tmp.get_long();
    tmp = tmp >> limbsize;
  }
  c[0] = 0;
  for(unsigned int i = 0; i < num_limbs;i++) {
    for(unsigned int j=0;j<limbsize;j++) {
      c[i+j+1] = ((ktmp[i] & (1<<j)) \
                  + (ktmp[i] & (1<<(j+1))) + c[i+j])/2;
      s[j+j] = (ktmp[i] & (1<<j)) + c[i+j] – 2*c[i+j+1];
    }
  }
  retval = (ecelement<T> *)&add_sub(s,num_limbs);
  delete []ktmp;
  delete []s;
  delete []c;
  return *retval;
} else { // < 32 bits, betyr at vi kan bruke versjon for 32–bit
  if(karakteristikk != 2)
    return double_and_add(k.get_long());
  else
    return add_sub(naf(k.get_long()),1);
}
```

```
}


// versjon av add_sub som tar et tall på non–adjacent–form som input.
template<class T>
const ecelement<T>& ecelement<T>::add_sub(int *s, \
                      const int& num_limbs) const
{
  ecelement <T> *retval = new ecelement<T>();
  int j;
  int limb_size = sizeof(unsigned long) * 8;
  // finn første element != 0

  for(j=num_limbs*limb_size - 1;s[j] == 0;j--)
    ;
  for(;j>=0;j--) {
    if(s[j] == 1) {
      *retval = *retval + *this;
    }
    if(s[j] == -1){
      *retval = *retval - *this;
    }
    *retval += *retval;
  }
  delete []s;
  return *retval;
}




// se avsnitt 7.5.2
template<class T>
```

```
int * ecelement<T>::naf(const unsigned long& k) const
{
  // beregn "non–adjacent–form"
  static int s[sizeof(unsigned long)*8]; // mer enn god nok plass;
  int c[sizeof(unsigned long)*8];

  c[0] = 0;

  for (unsigned long i=0;i<=sizeof(unsigned long)*8 – 2;i++) {
    c[i+1] = ((k & (1 << i)) + (k & (1<<(i+1))) + c[i])/2;
    s[i] = (k & (1<<i)) + c[i] – 2*c[i+1];
  }
  return s;
}

// se avsnitt 7.5.1
template<class T>
const ecelement<T>& ecelement<T>::double_and_add( \
                      const unsigned int& k) const
{
  ecelement<T> *retval = new ecelement<T>();
  ecelement<T> *p = new ecelement<T>(*this);

  int i = k;
  if(i & 1)
    *retval = *this;
  else
    *retval = inf;
  for(i>>=1; i; i>>=1) {
    *p += *p;
    if(i & 1)
      *retval += *p;
  }
```

```
  return *retval;
}

template<class T>
ecelement<T>& ecelement<T>::finn_punkt(void)
{
  // returnér et tilfeldig punkt på kurven
  ecelement<T> *P;
  do {
    T x = rnd(); // finn en tilfeldig x–verdi
    T y;
    // y^2 = x^3 + ax + b
    y = x*x*x + a * x + b;
    // beregner legendresymbolet...
    if(y.legendre(karakteristikk)==1){
      // kvadratrot modulo p...
      P = new ecelement<T>(x,y.sqrt(karakteristikk));
      return *P;
    }
  } while(1);
}


// allocate space for static members:

  template<class T>
    T ecelement<T>::a;

  template <class T>
    T ecelement<T>::b;

  template <class T>
    unsigned long ecelement<T>::kexp_l;
```

```
  template<class T>
   int * ecelement<T>::lambda[2];

  template<class T>
    ecelement<T> ecelement<T>::inf;

  template<class T>
    T ecelement<T>::karakteristikk = T(0); // gjør det enkelt å sjekke om
                                           // variabelen er initiert (0 er
                                           // ulovlig verdi).

  template<class T>
    T ecelement<T>::kexp;

} // end namespace ec

typedef ec::ecelement<mp::gmp> pt;

#endif
```

## A.3 ecinst.cc

```
// Instansiering av ecelement:
// $Id: ecinst.cc,v 1.1.1.1 2000/08/13 14:05:18 andreasd Exp $
//

#include "gmpwrapper"
#include "ec"

template class ec::ecelement<mp::gmp>;
```

## A.4 ectest.cc

```
// -*- C++ -*-
// $Id: ectest.cc,v 1.1.1.1 2000/08/13 14:05:18 andreasd Exp $


#include "ec"
#include "gmpwrapper"
#include <iostream.h>


int main(void)
{

   pt x, y;
   try{
     x.setBasis(2,156);


   }catch(string s){
     std::cerr << s <<endl <<flush;
   }
}
```

## A.5 elgamal

```
// -*- C++ -*-
// $Id: elgamal,v 1.3 2002/04/19 11:39:00 andreasd Exp $

#ifndef __EC_KEY_
#define __EC_KEY_ 1

#include "gmpwrapper"
#include "ec"
#include "mlib"
#include <utility>

namespace ec{

  template<class T> class key
  {
    T d; // hemmelig verdi
    ecelement<T> P; // "hemmelig" punkt, privat nøkkel
    ecelement<T> dP; // "offentlig nøkkel"
  public:
    key(ecelement<T>&);
    const key& operator=(ecelement<T>&); // sett hemmelig nøkkel
    const key& operator=(T &); // sett "hemmelig verdi"
    // multipliser rQ med d
    const ecelement<T>& operator*(const ecelement<T>& rQ) const;
  };


  template<class T> class elgamal
  {
    ecelement<T> rQ;
    ecelement<T>  PrdQ;
```

```
  public:
    elgamal(const ecelement<T>&, const key<T>&);
    elgamal(void);
    const ecelement<T>& dekrypter(const key<T>&) const;
  };


}

#endif
```

## A.6 elgamal.cc

```
#include "elgamal"
#include "mlib"

namespace ec {

  template <class T>
  elgamal<T>::elgamal(const ecelement<T>& P,const key<T> &k)
  {
    rQ = P * P.rnd();
    PrdQ = k * rQ;
  }

// dekrypter med nøkkel k
  template <class T>
  const ecelement<T>& elgamal<T>::dekrypter(const key<T>& k) const
  {
    ecelement<T> *retval = new ecelement<T>(PrdQ);
    *retval = *retval – k*rQ;
    return *retval;
  }


  template <class T>
  key<T>::key(ecelement<T> &P1)
  {
    P = P1;
    d = P.rnd();
    dP = P*d;
  }
```

```
  // multipliser rQ med d
  template <class T>
  const ecelement<T>& key<T>::operator*(const ecelement<T>& rQ) const
  {
    ecelement<T> *retval = new ecelement<T>(rQ * d);
    return *retval;
  }


  // instansiering av templates:
  template class key<mp::gmp>;
  template class key<mlib::mlib>;
  template class elgamal<mp::gmp>;
  template class elgamal<mlib::mlib>;
}
```

## A.7 elgamaltest.cc

```
#include "elgamal"
#include "gmpwrapper"
#include "ec"
#include "mlib"

using mp::gmp;
using ec::ecelement;
using ec::key;
using ec::elgamal;
using mlib::mlib;

//typedef ecelement<gmp> punkt;
//typedef key<gmp> nokkel;
int main(int argc, char *argv[]);
void test_F2(void);
void test_Fp(void);
void test(ecelement<gmp> &P, ecelement<gmp> &Q, key<gmp> &k);
void test(ecelement<mlib> &P, ecelement<mlib> &Q, key<mlib> &k);
int main(int argc, char *argv[])
{
  if(strstr(*argv,"test_f2"))
    test_F2();
  else
    test_Fp();
  return 0;
}

void test_F2(void)
{
  //t^163 + t^7 + t^6 + t^3 + 1, se [oCIoST00, s.36]
```

```
  mlib x,y, tmp;
  x = "0x05679b353caa46825fea2d3713ba450da0c2a4541";
  y = "0x235b7c6810050689906bac39d9dec76835591edb2";
  ecelement<mlib> P(x,y);
  //t^163 + t^7 + t^6 + t^3 + 1, se [oCIoST00, s.36]
  P.setBasis(2,163);
  P.setA(1); // dette er en "anomal–kurve"; den er fremdeles anbefalt av NIST
  P.setB(27);
  mlib konst = P.rnd();
  ecelement<mlib> Q = P*konst;
  // opprett nøkkel.., det blir automatisk valgt en "hemmelig" verdi.
  key<mlib> k(P);

  std::cout << "TestF2"<< std::endl << std::flush;

  // for å få frem forhåndstall mellom de forskjellige
  // funksjonskallene i implementasjonen kaller jeg constructor i
  // elgamal 1000–ganger.
  for (int i = 0; i < 1000; i++) {
    test(P,Q,k);
  }
}

void test_Fp(void)
{
  /* NIST–kurve P − 192, se [oCIoST00, s.29] */
  gmp x, y, b, p;
  x = "0x188da80eb03090f68cbf20eb43a18800f4ff0afd82ff1012";
  y = "0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811";
  b = "0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1";
  p = "6277101735386680763835789423207666416083908700390324961279";
  ecelement<gmp> P(x, y);
  P.setBasis(p);
```

```
  P.setA(-3);
  P.setB(b);
  gmp konst = P.rnd();
  ecelement<gmp> Q = P*konst;
  // opprett nøkkel.., det blir automatisk valgt en "hemmelig" verdi.
  key<gmp> k(P);

  std::cout << "TestFp"<< std::endl << std::flush;

  // for å få frem forhåndstall mellom de forskjellige
  // funksjonskallene i implementasjonen kaller jeg constructor i
  // elgamal 1000–ganger.
  for (int i = 0; i < 1000; i++)
    test(P,Q,k);
}

void test(ecelement<gmp> &P, ecelement<gmp> &Q, key<gmp> &k)
{
// Objektet som e refererer til inneholder nå tuplene r*Q og
  elgamal<gmp> e(Q,k);
  //   e.dekrypter(k);
}

void test(ecelement<mlib> &P, ecelement<mlib> &Q, key<mlib> &k)
{
// Objektet som e refererer til inneholder nå tuplene r*Q og
  elgamal<mlib> e(Q,k);
  //   e.dekrypter(k);
}
```

## A.8 gmpwrapper

```
/* -*- C++ -*-
 * $Id: gmpwrapper,v 1.8 2002/04/28 20:12:49 andreasd Exp $
 */

#ifndef __GMPWRAPPER__
#define __GMPWRAPPER__ 1

extern "C"{
#include <gmp.h>
}
#include <string>

namespace mp{

  /*
   * class gmp: C++ wrapper for Gnu MP
   * Inneholder kun funksjoner jeg tror vil bli brukt i oppgaven...
   * Skal "renskes" opp (dvs. vil fjerne ting som ikke brukes etterhvert)
   */

  class gmp{
    mpz_t i;
  public:
    gmp();
    gmp(const gmp*);
    gmp(const gmp&);
    gmp(const mpz_t);
    gmp(const int&);
    ~gmp();
    void operator=(const gmp&);
    void operator=(const int);
```

```
void operator=(const mpz_t);
void operator=(mpz_t);
void operator=(const char*);
const gmp& rnd(const gmp&) const;
const int legendre(const gmp&) const;
const gmp& sqrt(gmp &) const;
const bool operator<(const int&) const;
const gmp& operator*=(const gmp&);
const gmp& operator*(const gmp&) const;
const gmp& exp(const gmp&, const gmp&) const;
const gmp& operator*=(int);
const gmp& operator*(int) const;
const gmp& operator>>(const int&) const;
const gmp& operator/=(const gmp&);
const gmp& operator/(const gmp&) const;
const gmp& operator/=(const int&);
const gmp& operator/(const int&) const;
const gmp& operator-() const; // negasjon...
const gmp& operator+=(const gmp&);
const gmp& operator+(int) const;
const gmp& operator+=(int);
const gmp& operator+(const gmp&) const;
const gmp& operator-=(const gmp&);
const gmp& operator-(int) const;
const gmp& operator-=(int);
const gmp& operator-(const gmp&) const;
const gmp& operator%=(const gmp&);
const gmp& operator%(const gmp&) const;
const gmp& operator%=(int);
const gmp& operator%(int) const;
const gmp& operator&=(const gmp&); // bit-wise and
const gmp& operator&(const gmp&) const;
const gmp& operator|=(const gmp&);
```

```
  const gmp& operator|(const gmp&) const;
  const gmp& operator^=(const gmp&); // bitwise xor
  const gmp& operator^(const gmp&) const; // bitwise xor
  const gmp& operator~() const; // bitwise NOT
  const gmp& operator<<=(const unsigned int); // bit'ene vil _ikke_
  const gmp& operator<<(const unsigned int);  // rotere – størrelsen
                                              // på tallet vil *øke*...
  const bool operator==(const gmp&) const;
  const bool operator==(const unsigned int) const;
  const bool operator!=(const unsigned int) const;
  operator string() const;// FE2OSP
  //invers (ONB) x^-1
  const gmp& ror(const gmp&) const;
  //invers (ONB) x^-1
  const gmp& ror(unsigned long) const;
  //kvadrering (ONB) x^-1
  const gmp& rol(unsigned long) const;
  //multiplikasjon for ONB
  const gmp& f2mul(const gmp &, unsigned long, int *[2]) const;
  //multiplikasjon for ONB
  const gmp& f2mul(const gmp &b, gmp &, int *[2]) const;
  // operator unsigned long skaper problemer med operator=(gmp&, int&)
  const unsigned long& get_long() const;
  // returner størrelsen på i...
  const size_t& size() const;
};

const gmp& operator*(const int&, const gmp&);

ostream& operator<<(ostream& o, const gmp& g);
}
```

```
#endif
```

## A.9 gmpwrapper.cc

```
/*
 * $Id: gmpwrapper.cc,v 1.9 2002/04/28 20:12:49 andreasd Exp $
 */

#include "gmpwrapper"

#define lmbda(l,size,offa,offb) (l+size*offa+offb)

namespace mp{

  gmp::gmp()
  {
    mpz_init(i);
  }

  gmp::gmp(const int& j)
  {
    mpz_init(i);
    mpz_set_si(i,j);
  }

  gmp::gmp(const gmp* j)
  {
    // tør ikke kalle på gmp(); ettersom dette medførte ganske stygge
    // bugs sist gang jeg prøvde. Muligens en bug i g++ 2.93–pre?
    // kaller derfor på mpz_init() i alle constructorene
    mpz_init(i);
    mpz_set(i,j->i);
  }

  gmp::gmp(const gmp& j)
```

```
{
  mpz_init(i);
  mpz_set(i,j.i);
}

gmp::gmp(const mpz_t j)
{
  mpz_init(i);
  mpz_set(i,j);
}

gmp::~gmp()
{
  mpz_clear(i);
}

void gmp::operator=(const gmp& j)
{
  mpz_set(i,j.i);
}

void gmp::operator=(const char* s)
{
  mpz_set_str(i,s,0);
}

void gmp::operator=(int j)
{
  mpz_set_ui(i,j);
}

gmp::operator string() const
{
```

```
  //      std::cout<<"operator string()" << std::endl << std::flush;
  return mpz_get_str(NULL, 10, i);
}

const size_t& gmp::size() const
{
  size_t *retval = new size_t;
  *retval = mpz_size(i);
  return *retval;
}

const int gmp::legendre(const gmp& p) const
{
  return mpz_legendre(this->i, p.i);
}

// kvadratrot modulo p
const gmp& gmp::sqrt(gmp &p) const
{

  gmp x,y,e,q,n,b;
  do{
    n = rnd(p);
  }while(n.legendre(p) != -1);
  q = (p - 1) % 2;
  e = (p - 1) / q;
  y = n.exp(q,p);
  x = exp((q-1)/2,p);
  b=*this*x*x % p;
  x = *this*x % p;
  gmp m;
  while(b%p != 1) {
    m = m + 1;
```

```
    b = b*b;
    gmp t = y.exp(e-m-1,p);
    y = t*t % p;
    e = m;
    x = x*t % p;
    b = b*y % p;
  }
  return *(new gmp(x));
}

// x^n mod p
const gmp& gmp::exp(const gmp& n, const gmp& p) const
{
  gmp *retval = new gmp();
  mpz_powm(retval->i,i,n.i,p.i);
  return *retval;
}

const gmp& gmp::rnd(const gmp &max) const
{
  gmp *retval = new gmp();
  gmp_randstate_t state;
  gmp_randinit (state,  GMP_RAND_ALG_DEFAULT, 4);
  mpz_urandomm(retval->i, state, max.i);
  gmp_randclear(state);
  return *retval;
}

const bool gmp::operator<(const int& j) const
{
  return(mpz_cmp_si(i,j)<0);
}
```

```
const gmp& gmp::operator*=(int j)
{
  mpz_mul_ui(i,i,j);
  return *this;
}

const gmp& gmp::operator*(int j) const
{
  gmp *res = new gmp(i);
  return *res *= j; // her blir 'res' returnert...
}

const gmp& gmp::operator*=(const gmp& j)
{
  mpz_mul(i,i,j.i);
  return *this;
}

const gmp& gmp::operator*(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res *= j;
}


const gmp& gmp::operator/=(const gmp& j)
{
  mpz_div(i,i,j.i);
  return *this;
}

const gmp& gmp::operator/(const gmp& j) const
{
```

```
  gmp *res = new gmp(i);
  return *res /= j;
}

const gmp& gmp::operator/=(const int& j)
{
  gmp *tmp = new gmp(j);
  //      mpz_div_ui(i,i,j);
  return *this /= *tmp;
}

const gmp& gmp::operator/(const int& j) const
{
  gmp *res = new gmp(i);
  return *res /= j;
}



const gmp& gmp::operator+=(int j)
{
  mpz_add_ui(i,i,j);
  return *this;
}

const gmp& gmp::operator+(int j) const
{
  gmp *res = new gmp(i);
  return *res+=j;
}

const gmp& gmp::operator+=(const gmp& j)
{
  mpz_add(i,i,j.i);
```

```
  return *this;
}

const gmp& gmp::operator+(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res += j;
}



const gmp& gmp::operator-=(int j)
{
  mpz_sub_ui(i,i,j);
  return *this;
}

const gmp& gmp::operator-(int j) const
{
  gmp *res = new gmp(i);
  return *res-=j;
}

const gmp& gmp::operator-=(const gmp& j)
{
  gmp tmp = i;
  mpz_sub(i,i,j.i);
  return *this;
}

const gmp& gmp::operator-(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res -= j;
```

```
}

const gmp& gmp::operator%=(const gmp& j)
{
  gmp tmp = i;
  mpz_mod(i,i,j.i);
  return *this;
}

const gmp& gmp::operator%(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res %= j;
}

const gmp& gmp::operator%=(int j)
{
  mpz_mod_ui(i,i,j);
  return *this;
}

const gmp& gmp::operator%(int j) const
{
  gmp *res = new gmp(i);
  return *res %= j;
}


const gmp& gmp::operator&=(const gmp& j)
{
  mpz_and(i,i,j.i);
  return *this;
}
```

```
const gmp& gmp::operator&(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res &= j;
}

const gmp& gmp::operator|=(const gmp& j)
{
  mpz_ior(i,i,j.i);
  return *this;
}

const gmp& gmp::operator|(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res |= j;
}

const gmp& gmp::operator^=(const gmp& j)
{
  mpz_xor(i, i, j.i);
  return *this;
}


const gmp& gmp::operator^(const gmp& j) const
{
  gmp *res = new gmp(i);
  return *res ^= j;
}

// bitwise negasjon, ikke verdi...
```

```
const gmp& gmp::operator~() const
{
  gmp *res = new gmp();
  mpz_com(res->i,i);
  return *res;
}

const bool gmp::operator==(const gmp& j) const
{
  return mpz_cmp(i,j.i) == 0;
}

const bool gmp::operator==(const unsigned int j) const
{
  return mpz_cmp_si(i,j) == 0;
}

const bool gmp::operator!=(const unsigned int j) const
{
  return !(*this == j);
}

const gmp& gmp::operator-() const
{
  gmp *res = new gmp();
  mpz_neg(res->i,i);
  return *res;
}

const gmp& gmp::operator<<=(const unsigned int j)
{
  mpz_mul_2exp(i,i,j);
  return *this;
```

```
}

const gmp& gmp::operator<<(const unsigned int j)
{
  // No cyclic shift I'm afraid :-/
  gmp *res=new gmp(i);
  return *res<<=j;
}

ostream& operator<<(ostream& o, const gmp& g)
{
  // svært viktig: husk å caste gmp& -> ellers blir det
  // en rekursjonsbrønn...
  return o<<(string)g;
}

const gmp& operator*(const int& i, const gmp& j)
{
  return j*i;
}

const gmp& gmp::ror(const gmp& maxbit) const
{
  return ror(mpz_get_ui(maxbit.i));
}

const gmp& gmp::ror(unsigned long maxbit) const
{
  /* denne invers-funksjonen bruker _ROTATE_, funker kun med ONB1 */
  int carry = mpz_tstbit(i, 0);
  gmp *res = new gmp(i);
  mpz_fdiv_q_2exp(res->i,i,1);
  if(carry)
```

```
    mpz_setbit(res->i, maxbit - 1);
  return *res;
}

const gmp& gmp::f2mul(const gmp &b, gmp &kexp, int * lambda[2]) const
{
  return f2mul(b,mpz_get_ui(kexp.i), lambda);
}


/* \begin{tex}Dette er multiplikasjonsalgoritmen for $\fe$.  Det er
 * fullt mulig å implementere denne i GMP-biblioteket, men det er
 * mye "overhead" ved bitmanipulasjon\end{tex}
 */
const gmp& gmp::f2mul(const gmp &b,unsigned long kexp, \
                      int * lambda[2]) const
{
  /* c = a * b */
  kexp++;
  gmp *res = new gmp();
  gmp btmp = b.ror(kexp-2);
  gmp *atmp = new gmp[kexp];
  atmp = atmp;
  atmp[0] = *this;
  for(unsigned long i=1;i<kexp + 1;i++)
    atmp[i] = atmp[i-1].ror(kexp);

  for(unsigned long i=1;i<kexp + 1;i++)
    *res ^= btmp & (atmp[lambda[0][i]] ^ atmp[lambda[1][i]]);
  delete []atmp;
  return *res;
}
```

```
  const gmp& gmp::operator>>(const int &num) const
  {
    gmp *retval = new gmp();
    mpz_tdiv_q_2exp(retval->i,i, 8*sizeof(unsigned long int));
    return *retval;
  }

  const unsigned long& gmp::get_long() const
  {
    static unsigned long res = mpz_get_ui(i);
    return res;
  }

  const gmp& gmp::rol(unsigned long maxbit) const
  {
    /* denne kvadrerings-funksjonen bruker _ROTATE_, funker kun med ONB1 */
    int carry = mpz_tstbit(i, maxbit - 1);
    gmp *res = new gmp(i);
    mpz_clrbit(res->i, maxbit-1); // slett bit...
    mpz_mul_2exp(res->i,i,1);
    if(carry)
      mpz_setbit(res->i, 1);
    return *res;
  }
}
```

## A.10 mlib

```
// -*- C++ -*-
//
// $Id: mlib,v 1.11 2002/04/28 20:12:49 andreasd Exp $
// my own little test-lib:
//

// antall bits:
#ifndef __MLIB__
#define __MLIB__ 1
#define PRECISION 300
#define NUMLONGS (PRECISION/sizeof(unsigned long))

typedef unsigned long ulong_t;

extern "C" {
#include <gmp.h>
}
#include <string>
#include <iostream>

namespace mlib{
  class mlib {
    ulong_t num[NUMLONGS];
    static ulong_t poly_prime[NUMLONGS];
  public:
    mlib();
    mlib(const int& j);
    mlib(const char *s);
    //sett divisjonspolynom.
    const mlib& set_poly_prime(const mlib&) const;
    const mlib& operator=(const mlib&);
```

```
const mlib& operator=(const int&);
const mlib& operator<<(const int&) const;
const mlib& operator&(const mlib&) const;
const mlib& operator&=(const mlib&);
const mlib& ror(unsigned long) const; // roter mot høyre
const mlib& operator^(const mlib&) const;
const mlib& operator^=(const mlib &j);
const mlib& sqrt(void) const;
const int legendre(const mlib&) const;
const mlib& div_shift(void) const;
const mlib& poly_div(const mlib& div, mlib& quot, mlib& rem) const;
const mlib& poly_inv() const;
const unsigned long int log_2(const unsigned long int&) const;
const unsigned long int grad(void) const;
const mlib& sqrt(const mlib &) const;
void poly_gcd(mlib&, mlib&) const;
void mul_x_mod(mlib&, mlib&) const;
bool is_irreducible(void) const;

operator string() const;
const unsigned long& size() const
{
  static int res = NUMLONGS;
  return res;
}

const mlib& rnd(mlib& max);

const mlib& operator-(const mlib&) const;
const mlib& operator/=(const mlib&) const;
const mlib& operator-() const;
const mlib& operator*(const mlib&) const;
const mlib& operator>>(const int&) const;
```

```
    const bool operator<(const int&) const;
    const mlib& operator*=(const mlib&);
    const mlib& operator%=(const mlib&) const;
    const bool operator==(const mlib&) const;
    const bool operator==(const int&) const;
    const bool operator!=(const int&) const;
    const mlib& operator+(const mlib&) const;
    unsigned long &get_long() const;
    const mlib& f2mul(const mlib&, unsigned long, int *[2]) const;
  };
  ostream& operator<<(ostream& o, const mlib&);

  const mlib& operator*(const int& i, const mlib& j);
}
#endif
```

## A.11 mlib.cc

```
#include "mlib"
extern "C" {
  #include<stdio.h>
}

namespace mlib {

  mlib::mlib()
  {
    bzero(num, sizeof(num));
  }


  mlib::mlib(const int& j)
  {
    num[0] = j;
  }

  mlib::mlib(const char *s)
  {
    /* bruker vsscanf
     * vsscanf(const char *str, const char *format, va_list ap);
     */
    ulong_t i = strlen(s);
    ulong_t k;
    char *tmp = (char *)s;
    char tmpbuf[9];
    if(strstr(tmp,"0x")) {
      tmp+=2;
      for(ulong_t j = 0;j<i/8;j++) {
        strncpy(tmpbuf,tmp,8);
```

```
      tmp+=8;
    /* hexadesimalt tall */
      vsscanf(tmpbuf,"%x", (char *)&k);
      num[j] = k;
    }
  } else if(strstr(tmp,"%")) {
    /* binært tall */
    tmp++;
    ulong_t m;
    for(ulong_t j = 0;j<i/(sizeof(ulong_t)*8);j++){
      m = 0;
      for(ulong_t l = 0;k<8;l++){
        char c = *tmp;
        tmp++;
        if(c=='1')
          m |= 1 << l;
      }
      num[j] = m;
    }
  } else { /* desimal–representasjon... */
    for(unsigned int j = 0;j < i;j++) {
      for (unsigned int a = 0; a < 8; a++) {
        num[j*8] = (s[j+a] >> a) & 1;
      }
    }
  }
}

/* sett reduksjonspolynomet */
const mlib& mlib::set_poly_prime(const mlib& m) const
{
  if(m.is_irreducible()) {
    for(ulong_t i=0;i<NUMLONGS;i++)
```

```
      poly_prime[i] = m.num[i];
    return *this;
  } else {
    throw "set_divpoly: Ikke irredusibelt polynom";
  }
}


void mlib::poly_gcd(mlib& u, mlib& gcd) const
{
  mlib top;
  mlib r, dummy, tmp;
  top = u;
  r = *this;
  while(r.grad() >= 0) {
    top.poly_div(r,dummy,tmp);
    top = r;
    r = tmp;
  }
  gcd = top;
}

void mlib::mul_x_mod(mlib &u, mlib &w) const
{
  mlib mulx;
  ulong_t i, deg_v;
  deg_v = grad();
  mulx = u;
  mulx = mulx << 1;
  w = mulx;
  if(w.num[NUMLONGS - 1 - (deg_v/sizeof(ulong_t))]
     & (1L<<(deg_v % sizeof(ulong_t)*8)))
    for(i=0;i<NUMLONGS;i++) {
```

```
      w.num[i] ^= num[i];
    }
}

/* kontroller om polynomet er irredusibelt */
bool mlib::is_irreducible(void) const
{
  mlib vprm, gcd, x2r, x2rx, tmp;
  mlib sqr_x[(NUMLONGS+1) * sizeof(ulong_t)*8];
  ulong_t i,r,deg_v,k;
  /* masken 0x55555555 fjerner annen-hver bit
   * Vi ønsker å se på eksponensienter av _odde_ grad,
   * dette får vi til med høyreskift og maskering.
   */
  for(i=0;i<NUMLONGS;i++)
    vprm.num[i] = (num[i] >> 1) & 0x55555555;
  poly_gcd(vprm, gcd);
  /* Sjekk om gcd(*this,vprm) != 1 */
  if(gcd.num[NUMLONGS-1] > 1)
    return false;
  /* gjør tilsvarende for alle deler av *this... */
  for(i=0;i<NUMLONGS;i++)
    if(gcd.num[i])
      return false;
  deg_v = grad();
  sqr_x[0].num[0] = 1;
  /* generer en vektor med x^2k mod v
   * som vi kan bruke til å beregne x^2^r
   */
  for(i=1;i<=deg_v;i++){
    mul_x_mod(sqr_x[i-1], tmp);
    mul_x_mod(tmp, sqr_x[i]);
  }
```

```
  /* sjekk om gcd(x^2 - x, v) == 1 */
  x2r.num[NUMLONGS-1] = 2;
  for(r=1;r<=deg_v/2;r++) {
    for(i=0;i<=deg_v;i++) {
      if(x2r.num[NUMLONGS-1-(i/sizeof(ulong_t))] & (1L<<(i%sizeof(ulong_t)
        for(k=0;i<NUMLONGS;k++)
          x2rx.num[k] ^= sqr_x[i].num[k];
    }

    x2r = x2rx;
    x2rx.num[NUMLONGS-1] ^= 2;

    poly_gcd(x2rx,gcd);
    if(gcd.num[NUMLONGS-1] > 1)
      return false;
    for(i=0;i<NUMLONGS;i++)
      if(gcd.num[i])
        return false;
  }
  /*dersom vi kommer hit er vi sikre på at dette er et irredusibelt polynom */
  return true;
}

const mlib& mlib::operator&=(const mlib& b)
{
  for(unsigned int i=0;i<NUMLONGS;i++)
    num[i] &= b.num[i];
  return *this;
}

const mlib& mlib::operator&(const mlib& b) const
{
  mlib *res = new mlib();
```

```
  for(unsigned int i=0;i<NUMLONGS;i++)
    res->num[i] = num[i] & b.num[i];
  return *res;
}

// multiplikasjon i ONB1 lambda er multiplikasjonsmatrisen, den
// genereres i funksjonen genlambda() i ecelement-klassen
const mlib& mlib::f2mul(const mlib &b,ulong_t kexp, \
                        int * lambda[2]) const
{
  /* c = a * b */
  kexp++;
  mlib *res = new mlib();
  mlib btmp = b.ror(kexp);
  mlib *atmp = new mlib[kexp];
  atmp[0] = *this;
  for(ulong_t i=1;i<kexp + 1;i++)
    atmp[i] = atmp[i-1].ror(kexp);

  for(ulong_t i=1;i<kexp + 1;i++)
    *res ^= btmp & (atmp[lambda[0][i]] ^ atmp[lambda[1][i]]);
  delete []atmp;
  return *res;
}

const mlib& mlib::ror(ulong_t maxbit) const
{
  // roter mot "høyre", maxbit forteller index "hvor" vi skal dytte inn
  // bit på venstre side
  int ubitlw = sizeof(ulong_t)*8 - 1; // øverste bit i longword
  int ulw = (maxbit / sizeof(ulong_t)*8) - 1; // øverste longword i bruk
  int bnum = (maxbit % sizeof(ulong_t)*8) - 1; // bitnummer i longword
  int lsb = num[0] & 1;
```

```
  int carry = 0, carry2 = 0;
  mlib *res = new mlib(*this);

  for(int x=ulw; x>=0; x--) {
    carry2 = res->num[x] & 1;
    res->num[x] >>= 1;
    res->num[x] |= carry << ubitlw;
    carry = carry2;
  }
  res->num[ulw] |= lsb << bnum;
  return *this;
}

const mlib& mlib::operator=(const int& j)
{
  int k = j;
  for(unsigned int i = 0; i < sizeof(int);i++){
    num[i] = k & 1;
    k >>= 1;
  }
  return *this;
}

const mlib& mlib::operator=(const mlib& j)
{
  for(unsigned int i=0;i<NUMLONGS;i++)
    num[i] = j.num[i];
  return *this;
}

const mlib& mlib::operator<<(const int &j) const
{
```

```
  mlib *res = new mlib(*this);
  if(j != 1){
    cout << "<< with shamt > 1 not implemented!\n" <<endl;
    cout << "Kall rutine flere ganger!" << endl;
    return *res;
  }else{
    // fra høyre mot venstre:
    int carry = 0;
    for(int i=NUMLONGS-1;i>=0;i--){
      res->num[i]<<=1;
      res->num[i] |= carry;
      carry = (res->num[i] >> (sizeof(ulong_t)*8-1)) & 1;
    }
  }
  return *res;
}

const mlib& mlib::operator^=(const mlib &j)
{
  *this = *this ^ j;
  return *this;
}

const mlib& mlib::operator^(const mlib &j) const
{
  mlib *res = new mlib(*this);
  for(unsigned int i=0;i<NUMLONGS;i++)
    res->num[i] = num[i] ^ j.num[i];
  return *res;
}

const mlib& operator*(const int& i, const mlib& j)
{
```

```
  return j*i;
}

ostream& operator<<(ostream& o, const mlib& m)
{
  return o << (string)m; // må konverteres...
}

mlib::operator string() const
{
  string *retval;
  retval = new string[NUMLONGS*sizeof(ulong_t) + 1];
  retval[NUMLONGS*sizeof(ulong_t)] = '\0';
  for(unsigned int i = 0; i < NUMLONGS - 1;i++) {
    for(unsigned int j = 0; j < sizeof(ulong_t) ; j+=8){
      retval[i*sizeof(ulong_t) + j] = (num[i] & (0xff << j));
    }
  }
  return *retval;
}

const mlib& mlib::rnd(mlib& max)
{
  /* generer et tilfeldig tall
   * vi bruker en random-funksjon i GMP-biblioteket
   */
  mpz_t i,j;
  gmp_randstate_t state;
  gmp_randinit (state,  GMP_RAND_ALG_DEFAULT,4);
  mpz_init(i);
  mpz_init(j);
  mpz_set_ui(j, max.num[0]);
  mpz_urandomm(i, state, j);
```

```
  mlib *retval = new mlib(mpz_get_str(NULL, 10, i));
  mpz_clear(i);
  mpz_clear(j);
  gmp_randclear(state);
    return *retval;
}


/* høyre–shift rutine for polynomdivisjon */
const mlib& mlib::div_shift(void) const
{
  mlib *res = new mlib(*this);
  int b = 0;
  for(unsigned int i=0;i<NUMLONGS;i++){
    unsigned int tmp = (res->num[i] >> 1) | b;
    b = (res->num[i] & 1) ? 1<<(sizeof(ulong_t)*8 - 1) : 0UL;
    res->num[i] = tmp;
  }
  return *res;
}

/* binærsøk etter MSB (mest signifikante bit) */
const ulong_t mlib::log_2(const ulong_t& w) const
{
  ulong_t i;
  for(i=sizeof(ulong_t)*8-1;!(w & (1 << i));i++)
    ;
  return i;
}

/* returner graden til polynomet */
const ulong_t mlib::grad(void) const
{
```

```
  /* søk etter et ikke–null element */
  ulong_t i;
  for(i=NUMLONGS;!num[i];i--)
    ;
  return i*sizeof(ulong_t)*8 + log_2(num[i]);

}

/* divisjon av to polynomer, returnerer ((*this / div) + rem) */
/*  Se [PTVF92] */
const mlib& mlib::poly_div(const mlib& div,mlib& quot, mlib& rem) const
{
  mlib *tmp = new mlib();
  mlib *top = new mlib(*this);
  ulong_t deg_tmp, deg_div, equot;
  quot = 0;
  deg_tmp = this->grad();
  deg_div = div.grad();
  if(deg_tmp < deg_div) {
    rem = *this;
    return *tmp;
  }
  *tmp = div;
  /* divisor må tilpasses divident */
  for(ulong_t i=0;i<(deg_tmp - deg_div);i++)
    *tmp = *tmp << 1;
  ulong_t topb = 1UL << (deg_tmp % sizeof(ulong_t));
  ulong_t tptr = deg_tmp / (sizeof(ulong_t)*8);

  /* for hver mulig kvotient: sjekk om vi kan trekke fra divisor */
  for(ulong_t bc = deg_tmp + 1;bc;bc-- ) {
    if(tptr & topb) {
      *top ^= *tmp;
```

```
      equot = NUMLONGS - (deg_tmp - deg_div)/sizeof(ulong_t);
      quot.num[equot] |= 1UL << (deg_div % sizeof(ulong_t));
    }
    deg_tmp--;
    tmp->div_shift();
    topb >>= 1;
    if(!topb) {
      topb = 1 << (sizeof (ulong_t)*8 - 1);
      tptr++;
    }
  }
  return *tmp;
}

const mlib& mlib::sqrt(const mlib& p) const
{
  // i polynomisk basis kan vi beregne dette ved en _høyreskift_...
  mlib *retval = new mlib();
  int lsb = 0;
  for(ulong_t i=NUMLONGS - 1;i>=0;i--) {
    retval->num[i] = num[i] >> 1;
    if(lsb) {
      retval->num[i] |= 1<<(sizeof(ulong_t)*8 - 1);
    }
    lsb = num[i] & 1;
  }
  return *retval;
}

/* inversjon av polynom, a^-1 */
const mlib& mlib::poly_inv(void) const
{
  mlib *pk, *pk1, *pk2, *rk, *rk1, *rk2, *invers, *qk, *qk1, *qk2;
```

```
  rk2 = new mlib();
  for(ulong_t i=0;i<NUMLONGS;i++)
    rk2->num[i] = poly_prime[i];
  *rk1 = *this;
  pk = new mlib();
  pk1 = new mlib();
  pk2 = new mlib();
  qk1 = new mlib();
  qk2 = new mlib();
  pk1->num[NUMLONGS - 1] = 1L;
  qk1->num[NUMLONGS - 1] = 1L;
  pk->num[NUMLONGS - 1] = 1L;
  *rk2 = rk2->poly_div(*rk1,*qk,*rk);
  while(rk->grad() >= 0){
    for(ulong_t i=0;i<NUMLONGS;i++)
      pk->num[i] = rk2->num[i] ^ pk2->num[i];
    *rk2 = *rk1;
    *rk1 = *rk;
    *qk2 = *qk1;
    *qk1 = *qk;
    *pk2 = *pk1;
    *pk1 = *pk;
    rk2->poly_div(*rk1,*qk,*rk);
  }
  *invers = *pk;
  return(*invers);
}

const mlib& mlib::operator-(const mlib& j) const
{
  cout << "Minus finnes ikke i ONB-repr" << endl;
  cout << "Bruk inv + mul?" << endl;
  return *this;
```

```
}

const mlib& mlib::operator/=(const mlib& j) const
{
  cout << "Divisjon må gjøres ved invers + multiplikasjon" << endl;
  return *this;
}

const mlib& mlib::operator-() const
{
  cout << "Negasjon finnes ikke i ONB-repr" << endl;
  cout << "Bruk inv + mul?" << endl;
  return *this;
}

const mlib& mlib::operator*(const mlib &j) const
{
  cout <<"Bruk f2mul() for ONB" << endl;
  return *this;
}

const mlib& mlib::operator>>(const int& j) const
{
  mlib *retval = new mlib(*this);
  if(j == sizeof(ulong_t)*8) {
    for(unsigned int i=0;i<NUMLONGS-1;i++) {
      retval->num[i] = retval->num[i+1];
    }
    retval->num[NUMLONGS-1] = 0;
  } else {
    for(int i=0;i<j;i++)
      *retval = retval->ror(1);
  }
```

```
  return *retval;
}

const bool mlib::operator<(const int &j) const
{
  if(j<0)
    return false;
  return(num[0] < (unsigned int)j);
}

const mlib& mlib::operator*=(const mlib& j)
{
  *this = *this * j;
  return *this;
}

/* Polynomdivisjon */
const mlib& mlib::operator%=(const mlib& j) const
{
  cout << "not valid for ONB"<<endl;
  return *this;
}

const bool mlib::operator==(const mlib& j) const
{
  unsigned int i;
  for(i=0;i<NUMLONGS && j.num[i]==num[i] ;i++)
    ;
  return (i == NUMLONGS);
}

const bool mlib::operator==(const int& j) const
{
```

```
  if(j<0)
    return false;
  return(num[0] == (unsigned int)j);
}

const bool mlib::operator!=(const int& j) const
{
  return !(*this == j);
}

const mlib& mlib::operator+(const mlib &j) const
{
  return *this ^ j;
}

ulong_t& mlib::get_long() const
{
  static ulong_t res;
  res=num[0];
  return res;
}

ulong_t mlib::poly_prime[NUMLONGS];

}
```

## A.12 mlibinst.cc

```
// Instansiering av ecelement:
// $Id: mlibinst.cc,v 1.1 2002/04/09 13:29:33 andreasd Exp $
//

#include "mlib"
#include "ec"

template class ec::ecelement<mlib::mlib>;
```