Studieretningsprojekt Roskilde Tekniske Gymnasium
December 2013 - opgaveformulering

Navn Magnus Brandt-Møller	Klasse 3.6
Fag Matematik A	Vejleder Bert van der Vegte
Fag Informationsteknologi B	Vejleder Karl Bjarnason

Opgaveformulering

Kryptografi med ECC

- Redegør for den grundlæggende matematik bag ECC, kryptering ved brug af elliptiske kurver. Sammenlign i din gennemgang gerne med andre anvendte krypteringsalgoritmer.
- Giv et eksempel på, hvordan en tekst krypteres og dekrypteres med ECC.
- Lav et program med grafisk brugerflade som kan bruge ECC eller RSA til at kryptere og dekryptere tekst. Afprøv programmet med flere tekster.
- Sammenlign ECC med RSA med henblik på sikkerhed.
- Giv et bud på, hvad status er på RSA algoritmen, og hvilken effekt det ville have, hvis den blev brudt.

Opgaven skal have et engelsk resumé på 15-20 linjer og forventes at have et omfang på ca. 8-12 normalsider a 2.500 anslag inkl. Mellemrum og ekskl. eventuelle bilag. Opgavestilleren fastlægger omfang jf. Vejledningen s. 11.

Jeg bekræfter med min underskrift, at opgavebesvarelsen er udarbejdet af mig. Jeg har ikke anvendt tidligere bedømt arbejde uden henvisning hertil.

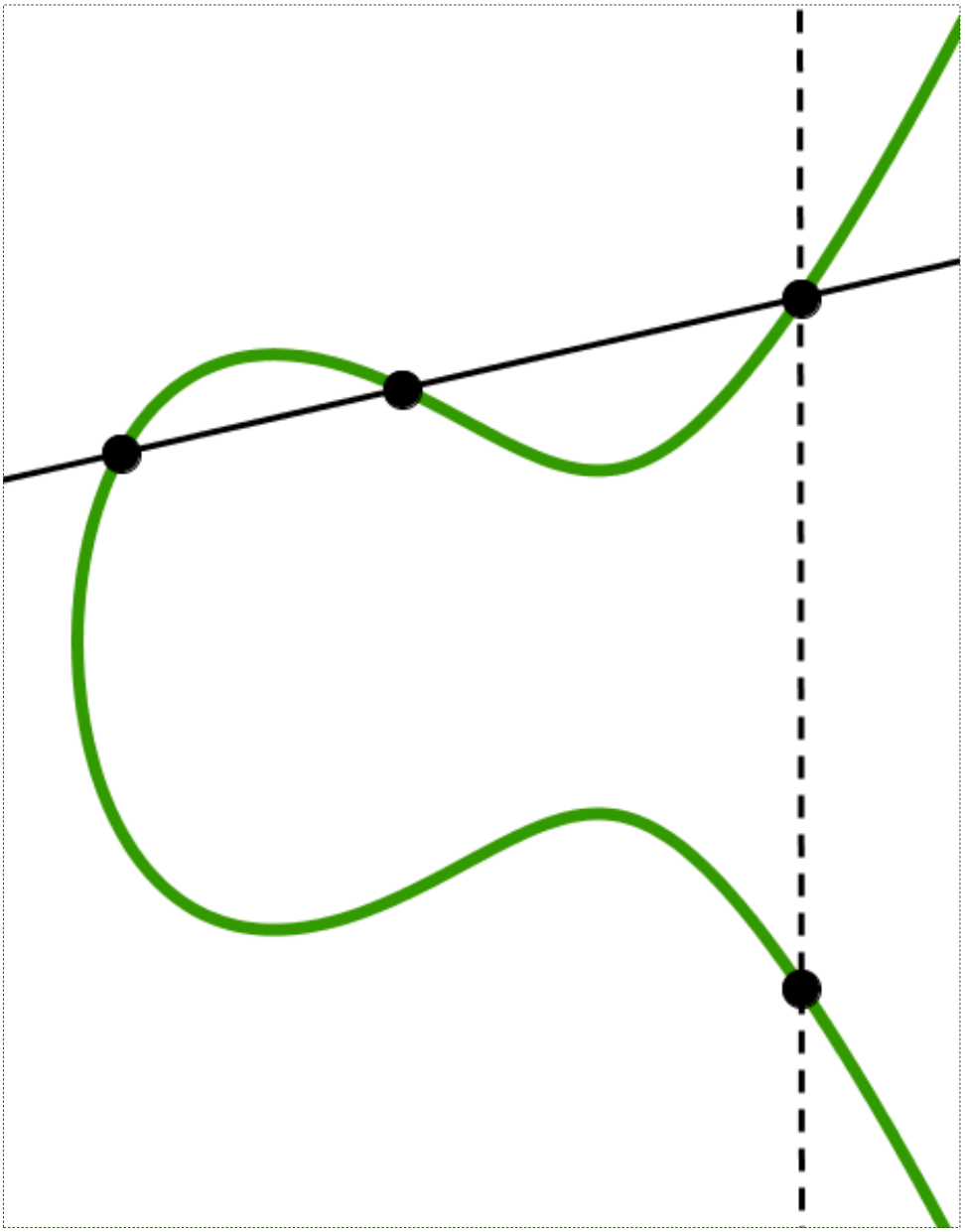
Elevens underskrift: Magnus Brandt-Møller

Studieretningsprojektet skal senest 20. december 2013 afleveres i 2 underskrevne eksemplarer og elektronisk på Fronter

Denne side indsættes som forside til besvarelsen.

KRYPTOGRAFI MED ELLIPTISKE KURVER

Studieretningsprojekt i Mat A og IT B



Vejledere/Tak til Bert van der Vegte & Karl G Bjarnason

Udarbejdet af Magnus Brandt-Møller

Afleveret d. 20/12-2013

Skole Roskilde Tekniske Gymnasium

Klasse 3.6

Abstract

This paper examines the fundamental mathematics behind the cryptosystems RSA and ECC. This includes the theory of essential modular arithmetic's and finite fields upon which RSA and ECC both rely. After studying the general mathematic problems behind ECC and RSA, respectively the generalized discrete logarithm problem and prime factorization, a practical example of ECC encryption and decryption is given. The algorithms behind RSA and Elliptic Curve Cryptography, are implemented in an accessible graphical user interface, and the usability of same is discussed. Comparisons between the two cryptosystems yield the result of the Elliptic Curve cryptosystem as the right path in the future, for the reason that there are no known efficient attacks against ECC. Experts state that RSA is becoming obsolete, and a paradigm shift towards ECC is essential within the next five years, if we are to prevent the cracking of not only our financial systems, but also our personal information. The notion of cryptography is not static, and we should be aware of aging cryptosystems.

Indhold

Indledning **1**

Kryptografi og matematik **1**

Modulær aritmetik 3
Finitte cykliske grupper 4
Kort om RSA 5
Matematikken bag Elliptiske kurver 6

Kryptering med Elliptiske Kurver **12**

Grafisk implementering af ECC og RSA **15**

Sikkerhed - ECC vs. RSA **18**

Status på RSA og fremtiden **20**

Konklusion **21**

Referencer **22**

Bøger 22
Artikler 22
Hjemmesider 22

Bilag **23**

Bilag 1 – Bevis for kryptering med RSA 23
Bilag 2 – Bevis for kryptering med Elliptiske kurver 26
Bilag 3 – Implementeringsalgoritmer 27
Bilag 4 – Implementering af ECC & RSA i C# 31

Figurer

Figur 1 – Symmetrisk kryptering 1
Figur 2 – Asymmetrisk kryptering 2
Figur 3 - Eksempel på elliptisk kurve over reelle tal $\boldsymbol{y2 = x3 - 3x + 3}$ 7
Figur 4 - Punkt addition (et geometrisk eksempel) 8
Figur 5 - Punkt fordobling (et geometrisk eksempel) 10
Figur 6 - Plot af $\boldsymbol{y2 \equiv x3 + 3x + 2 \bmod 23}$ - Via mit eget program 12
Figur 7 - Mit eget program – Kryptering med RSA 15
Figur 8 - Mit eget program - Kryptering med ECC 16
Figur 9 – Forholdet mellem ECC og RSA bit-længder 19
Figur 10 – Tidskompleksiteten bag faktorisering 24

Indledning

I denne rapport gives først et indblik i kryptografiens verden, hvorefter de grundlæggende matematiske strukturer bag kryptering i kryptosystemerne RSA og ECC beskrives. Derefter eftervises det hvordan kryptering med ECC i praksis kan realiseres, når to parter gerne vil kommunikere sikkert med hinanden. RSA og ECC kryptering implementeres i en grafisk tilgængelig brugerflade, og på baggrund af det matematiske grundlag sammenlignes sikkerheden bag de to seneste kryptosystemer RSA og ECC. Afslutningsvis gives der et bud på hvordan fremtiden vil se ud hvis kryptosystemet RSA blev brudt.

Hovedfokus i denne rapport ligger overvejende på det asymmetriske kryptosystem ECC. Der er i højere grad fokus på kryptering og dekryptering i ECC, end der eksempelvis er på nøglegenereringen i RSA. *Jeg benytter mig af punktum som decimalseparator gennem opgaven. I fodnoter henvises til litteraturlisten, hvor yderligere information om det givne værk kan findes.*

Kryptografi og matematik

Omkring begyndelsen af vores tidsregning var Romerriget, med Julius Cæsar, ved at udbrede sig drastisk. Men for at hans fjender ikke kunne finde ud af hvilke ordre han gav sin hær, måtte han på en eller anden måde gøre det umuligt for sine upålidelige budbringere at gennemskue hans besked[1]. Dog skulle modtageren af beskeden, være i stand til at finde frem til den originale besked. Det var dette grundlæggende problem, som selv kryptografien i dag bygger på. Problemet ses nedenfor:

Figur 1 – Symmetrisk kryptering

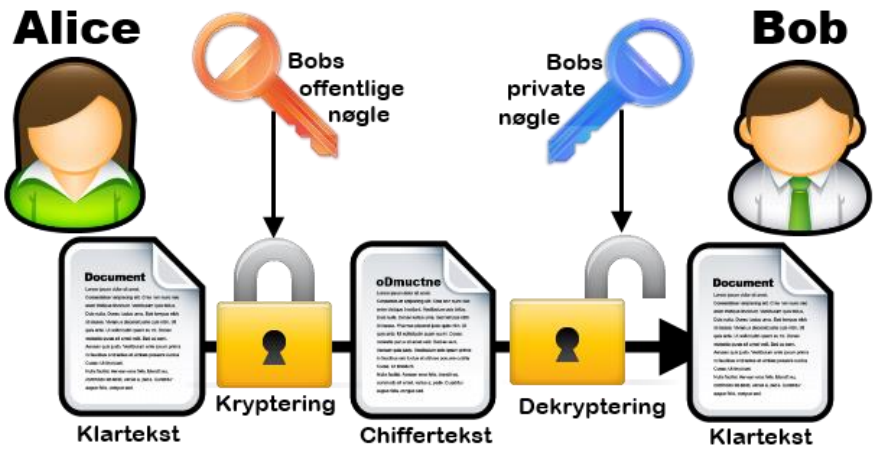
Problemet ligger altså i for Alice at kunne tage noget klartekst (tekst der endnu ikke er krypteret), kryptere dette til en chiffertekst (tekst der ulæselig), for så at dekryptere chiffereksten tilbage til klartekst af *kun* den tilsigtede modtager Bob. Alt dette ved hjælp af en hemmelig nøgle.

[1] Cruise: (2012)

Det skal bemærkes at jeg gennem rapporten kalder den der krypterer en besked for Alice, den der dekrypterer for Bob og den der utilsigtet prøver at få fat i en klartekst for "onde" Oscar. Det at kalde den der krypterer for Alice, den der dekrypterer for Bob, og den der er ond for Oscar, er en kendt krypteringskonvention[2].

Kryptografi er følgelig videnskaben om at bruge matematik til at kryptere og dekryptere data. Dette sætter os i stand til at gemme følsom information eller transmittere det over "ikke sikre" forbindelser (forbindelser hvor andre kunne tænkes at smugkigge på data), således at data ikke kan blive læst af uvedkommende. Medens kryptografi sikrer data, er kryptoanalyse det at analysere og bryde en tilsyneladende sikker kommunikation, hvilket jeg afslutningsvis i denne opgave vil bruge til at vurdere sikkerheden af to anvendte krypteringsalgoritmer.

Der findes grundlæggende to forskellige måder at kryptere på. Problemet ovenfor viser det der kaldes symmetrisk kryptografi, hvilket indebærer at krypteringen og dekrypteringen af en besked, sker med samme nøgle[3]. Det symmetriske kryptosystem er ikke hovedfokus i denne opgave, men bruges til at beskrive grundlæggende matematik bag kryptering generelt. Det asymmetriske kryptosystem går grundlæggende ud på, at kryptering og dekryptering udføres med to forskellige nøgler:



Figur 2 – Asymmetrisk kryptering

Bob offentliggør sin nøgle, hvormed Alice krypterer sin besked. Bobs private nøgle er den inverse (den omvendte) af den offentlige nøgle, hvorfor Bob er i stand til at dekryptere beskeden fra Alice. Bemærk at dette er sikkert selvom en ond person Oscar, prøver kompromittere sikkerheden og få adgang til beskeden, fordi det kræver at Oscar kender den private nøgle, som Bob aldrig har fortalt til andre.
Nedenfor vil jeg beskrive og sammenligne matematikken bag kryptografi og de mest anvendte asymmetriske krypteringsalgoritmer RSA og ECC.

[2] NetworkWorld (a): (2005)
[3] Cohen og Frey: (2006) s. 5

MODULÆR ARITMETIK

Inden for kryptering er modulær aritmetik vigtig. Et simpelt eksempel, hvilket påpeger vigtigheden af modulær aritmetik, er Cæsaralgoritmen[4]. Forestil dig, at når man vil kryptere beskeden "Hej", så erstatter man hvert bogstav i klarteksten, med et bogstav eksempelvis tre pladser længere fremme i alfabetet: "Hej" → "Khm". Dette virker i sig selv simpelt, men problemet her opstår, når man vil kryptere beskeder med de tre sidste bogstaver i alfabetet "æøå".
Efter lidt tid finder man ud af, at man blot tæller forfra i alfabetet således at "æøå" → "abc". Dette kaldes også for monoalfabetisk substitution, og betyder blot at ethvert bogstav "erstattes" af et andet bogstav, k pladser længere fremme eller tilbage i alfabetet[4]. Man skal altså finde en matematisk beskrivelse til at "tælle forfra", for at få metoden til at fungere. Samme problemstilling kender man allerede fra klokken, hvor man har en begrænset mængde tal/timer:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23\}$$

Almindelig aritmetik gælder ikke i en begrænset mængde:

$$21 + 9 = 6 \neq 30$$

Man ved godt at hvis klokken er $21{:}\,00$, og man lægger ni timer til, får man ikke at klokken er $30{:}\,00$ men $06{:}\,00$. Dette har en fin matematisk beskrivelse i modulær aritmetik, hvor man mere generelt skriver:

$$a \equiv r \bmod \mathrm{m}$$

Ovenstående udtryk kan beskrives med: Når man dividere venstre side af udtrykket a, med modulus m, hvad er resten r så. Dette regnes således:

$$r = a - \left\lfloor \frac{a}{m} \right\rfloor \cdot m$$

I eksemplet med klokken er $a = 21 + 9 = 30 \ \wedge m = 24$:

$$\left\lfloor \frac{a}{m} \right\rfloor = \left\lfloor \frac{30}{24} \right\rfloor = 1 \qquad (\boldsymbol{24\ går\ én\ gang\ op\ i\ 30})$$
$$r = 30 - 1 \cdot 24 = 6 \qquad (\boldsymbol{Resten\ er\ derfor\ 6})$$

får vi, at resten er 6. På den måde kan man, hvis klokken nu er 21:00, med modulær aritmetik beregne hvad klokken er om ni timer (06:00). Den matematiske definition bliver: Lad $a, r, m \in \mathbb{Z} \ \wedge \ m > 0$, så kan vi skrive:

$$a \equiv r \bmod m$$

Hvis m deler $a - r$. Hvor m kaldes *modulus*, og r kaldes *resten*[5].

[4] Sauerberg: (2011) s. 71
[5] Paar, Christof og Pelzl, Jan: (2010) s. 16

FINITTE CYKLISKE GRUPPER

Vi kan se hvordan modulær aritmetik spiller en vigtig rolle i kryptering, og ydermere lægger vi mærke til at brugen af modulær aritmetik danner en matematisk struktur, en ring af tal Z_m, der starter forfra igen og igen[6]. Eksempelvis kan heltals-ringen Z_3 med 3 tal dannes med modulus 3:

X	0	1	2	3	4	5	6	7	8
X mod 3	0	1	2	0	1	2	0	1	2

Tabel 1

Hvor:

$$x \in \mathbb{Z}_m = \{0,1,\dots,m-1\} \rightarrow x \in \mathbb{Z}_3 = \{0,1,2\}$$

For at man kan kalde det en matematisk ring, skal man kunne udføre en række operationer som multiplikation og addition og få et resultat, der ligger inde for ringens mængde Z_m:

$$3 + 2 \equiv 2 \bmod 3$$
$$4 \cdot 4 \equiv 1 \bmod 3$$

Som det kan ses er resultatet altid indenfor mængden Z_3, og den kan derfor kaldes en ring. For at man kan undersøge om en sådan ring kan bruges i kryptografi, skal den kunne beskrives mere abstrakt, som en matematisk gruppe. En matematisk gruppe G er et sæt elementer (tal), der har en matematisk operation betegnet $\circ$ som kombinerer to elementer i G. En sådan gruppe skal have egenskaberne[7]:

1. Gruppeoperationer skal være lukkede, enhver operation $a \circ b = c \in G$
2. Gruppeoperationer skal være associative, rækkefølgen er ligegyldig: $a \circ (b \circ c) = (a \circ b) \circ c$
3. Der skal forefindes et neutralt element (identitet) således at $a \circ 1 = 1 \circ a = a$ for alle $a \in G$
4. For hver element i G, $a \in G$, skal der eksistere en invers så at: $a \circ a^{-1} = a^{-1} \circ a = 1$
5. En gruppe er kommutativ hvis det gælder at: $a \circ b = b \circ a$ for alle $a, b \in G$

Liste 1

Fra liste 1, skal det bemærkes at $\circ$ betegner både multiplikation og addition. Og at en gruppe ikke nødvendigvis er finit (har et begrænset antal elementer).

Da eksemplet med mængden Z_3 opererer med "almindelige" regneoperationer, kan man nemt se at alle fem ovenstående regler gælder i Z_3, da den ingen aritmetiske begrænsninger har. Det specielle ved denne gruppe, er at den er finit, fordi den har et begrænset antal elementer. Størrelsen af gruppen kaldes gruppens kardinalitet. I Z_{11} er kardinaliteten eksempelvis $|Z_{11}| = 10$ (Bemærk at det sidste element ikke tælles med fordi 11 er et primtal).
Det interessante er nu at undersøge, om man også kan kalde den en cyklisk gruppe. Hvis ikke dette

[6] Gadegaard, Henrik G. og Hansen, Johan P: (2002) s. 65-66
[7] Paar, Christof og Pelzl, Jan: (2010) s. 209

er sandt, starter den ikke forfra, og kan derfor ikke bruges inden for kryptografi[8]. Måden man undersøger dette er ved at se, om der er et element med en maksimal orden, hvor ordenen af et element defineres:

Ordenen $ord(a)$ af et element af en gruppe G er det mindste positive heltal k så at[9]:

$$a^k = \underbrace{a \circ a \circ \ldots \circ a}_{k\ gange} = 1$$

Lad os eksempelvis undersøge Z_{11} for elementet $a = 2$:

$$a \equiv 2 \bmod 11$$
$$a^2 \equiv 4 \bmod 11$$
$$a^3 \equiv 8 \bmod 11$$
$$a^4 \equiv 5 \bmod 11$$
$$a^5 \equiv 10 \bmod 11$$
$$a^6 \equiv 9 \bmod 11$$
$$a^7 \equiv 7 \bmod 11$$
$$a^8 \equiv 3 \bmod 11$$
$$a^9 \equiv 6 \bmod 11$$
$$a^{10} \equiv 1 \bmod 11$$
$$a^{11} \equiv 2 \bmod 11$$
$$a^{12} \equiv 4 \bmod 11$$
$$\ldots$$

Det kan ses, at når man opløfter elementet a i eksponenten 10, får man 1, og derefter gentager mønsteret sig. Man siger at ordenen af a er 10: $ord(a) = 10$. Og da dette er den maksimale orden, er denne gruppe per definition cyklisk[10]. Det specielle ved grupper med n antal elementer, hvor n er et primtal er, at alle andre tal end 1, er primitive elementer. Med primitive elementer menes der, at elementet kan generere de tal som gruppen består af. Lad os se om eksponenter af elementet 2 generer alle tal i gruppen Z_{11}:

k	1	2	3	4	5	6	7	8	9	10
$2^k \bmod 11$	2	4	8	5	10	9	7	3	6	1

Tabel 2

Man kan hurtigt se at $a = 2$ er et primitivt element, fordi det generere alle elementer i Z_{11}. Vi bemærker samtidigt at rækkefølgen af de genererede tal ser ret tilfældig ud. Dette arbitrære forhold mellem eksponenten k og de generede elementer, er basis for de fleste kryptosystemer.

KORT OM RSA

Det asymmetriske kryptosystem RSA, benytter sig af finitte cykliske grupper. Det går ud på at Bob genererer en offentlig nøgle e og en privat nøgle d, samt kardinaliteten af den cykliske gruppe, typisk kaldet $N = pq$. Kryptering foregår ved at opløfte beskeden man vil sende M, repræsenteret som et tal, i den offentlige nøgle e, hvilket resulterer i en chiffertekst C. Dekrypteringen foregår ved at opløfte chifferteksten C til den private nøgle d, hvilket resulterer i den originale besked M:

[8] Paar, Christof og Pelzl, Jan: (2010) s. 210
[9] Paar, Christof og Pelzl, Jan: (2010) s. 211
[10] Hankerson, Darrel og Menezes, Alfred og Vanstone, Scott: (2004) s. 29

Kryptering: $M^e \equiv C \bmod pq$
Dekryptering: $C^d \equiv M \bmod pq$

Hvor p og q er primtal, e den offentlige nøgle og d den private nøgle[11]. For at have et solidt matematisk sammenligningsgrundlag mellem RSA og ECC beviser jeg hvorfor RSA virker i Bilag 1. Ud fra beviset kan følgende konklusion på RSA gives: Da kun produktet af p og q er kendt, skal Oscar altså faktorisere $N = pq$ i to primtal, for at finde den private nøgle d. Dette er grundlaget for RSA's virke: primtalsfaktorisering eller primtalsopløsning. Sikkerheden ligger i at det er meget svært (tidsmæssigt) at primtalsfaktorisere, hvis tallene er store nok[12]. I bilaget Figur 10 – ses, hvor lang tid det faktisk tager for en computer at faktorisere.

MATEMATIKKEN BAG ELLIPTISKE KURVER

På baggrund af ovenstående gennemgang af grundlæggende matematik bag kryptering generelt, er det nu muligt at introducere Elliptiske kurver og deres anvendelse inden for kryptografi. En af de største forskelle på RSA og ECC (Elliptic Curve Cryptography), er forskellen i de grundlæggende matematiske problemer de bygger på. Medens RSA som ovenfor vist, bygger på, at det er svært at primtalsfaktorisere, bygger ECC på en generalisering (ECDLP) af det Diskrete Logaritme Problem (DLP)[26]:
Givet er en finit cyklisk gruppe $\mathrm{Z_p}$ med det primitive element $\mathrm{a} \in \mathrm{Z_p}$, og et andet element $\mathrm{b} \in \mathrm{Z_p}$. Det det diskrete logaritme problem er så at finde tallet x, således at følgende ligning tilfredsstilles:

$$\mathrm{a^x} \equiv \mathrm{b} \bmod \mathrm{p}$$

x kaldes den diskrete logaritme af b til roden a: $\mathrm{x} \equiv \log_{\mathrm{a}}(\mathrm{b}) \bmod \mathrm{p}$

Et eksempel ville være: $5^{\mathrm{x}} \equiv 41 \bmod 47$ (**Det er svært at finde x**)

Selv med så små tal er det svært at se løsningen er $\mathrm{x} = 15$. Det er altså nemt at udregne 5 opløftet i x modulus 47 hvis man kender x, men det er rigtig svært hvis man kun har resultatet, at finde eksponenten x, der tilfredsstiller ovenstående ligning. Man kalder det for en envejsfunktion (oneway-function). Den er nem at udregne den ene vej, men rigtig svær at udregne den anden vej. Dette udnytter kryptosystemer der eksempelvis er bygget på DLP[13]. I beviset for ECC Bilag 2, uddyber jeg hvordan det diskrete logaritme problem, kommer til udtryk med elliptiske kurver.

Af navnet kunne man tro at en elliptisk kurve er af formen: $\frac{\mathrm{x}^2}{\mathrm{a}^2} + \frac{\mathrm{y}^2}{\mathrm{b}^2} = 1$, altså er en ellipse. Dog forholder det sig langt fra sådan (se figur 3). En elliptisk kurve E, er en speciel form for polynomium, og har den generelle formel:

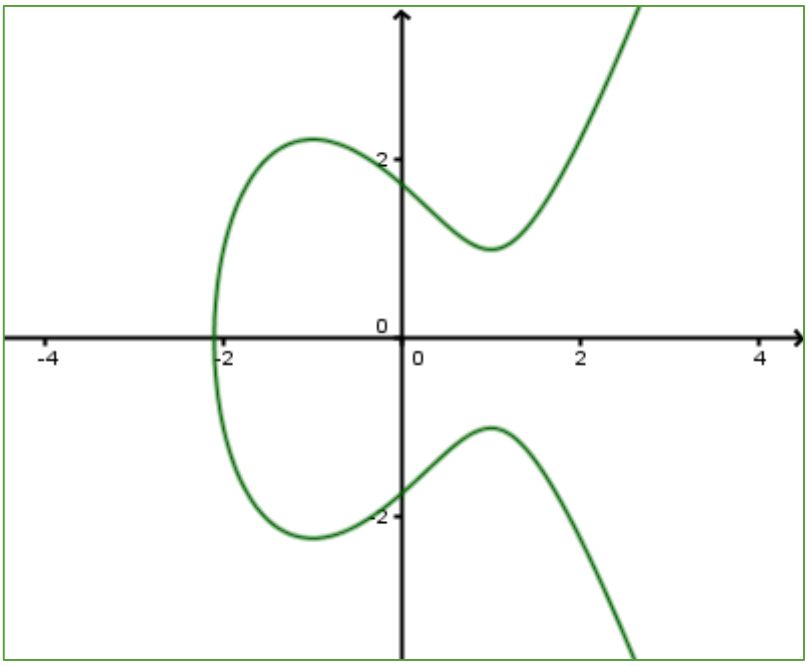
[11] Cohen, Henri og Frey, Gerhard: (2006) s. 7
[12] Nichols, Randall K. og Lekkas, Panos C: (2002) s. 228
[13] Paar, Christof og Pelzl, Jan: (2010) s. 153

$$E: y^2 = x^3 + ax + b$$

For at gøre denne brugbar indenfor kryptografi, er målet at finde en kurve med en stor cyklisk gruppe Z_p, ligesom med RSA, men hvor elementerne af den cykliske gruppe i RSA er tal, er alle elementer i en elliptisk cyklisk gruppe, en mængde af punkter $(x, y) \in \mathbb{Z}$, der tilfredsstiller nedenstående ligning[14] [15]:

$$E: y^2 \equiv x^3 + ax + b \bmod p$$

Hvis man indtegnede ovenstående elliptiske kurve over den cykliske gruppe Z_p, ville man ikke få noget der lignede en kurve (se figur 6), hvorfor mit eksempel på en elliptisk kurve $y^2 \equiv x^3 - 3x + 3$, for forståelsens skyld, starter med et plot over alle reelle tal:



Figur 3 - Eksempel på elliptisk kurve over reelle tal $y^2 = x^3 - 3x + 3$

Det er vigtigt at bemærke, at grafen for en elliptisk kurve ikke må skære sig selv, da den derfor ikke kan danne en cyklisk gruppe. Dette tjekker man ved at kigge på diskriminanten, som ligesom ved et andengradspolynomium beskriver hvor mange rødder polynomiet har:

$$D = 4 \cdot a^2 + 27 \cdot b^2 \neq 0 \bmod p$$

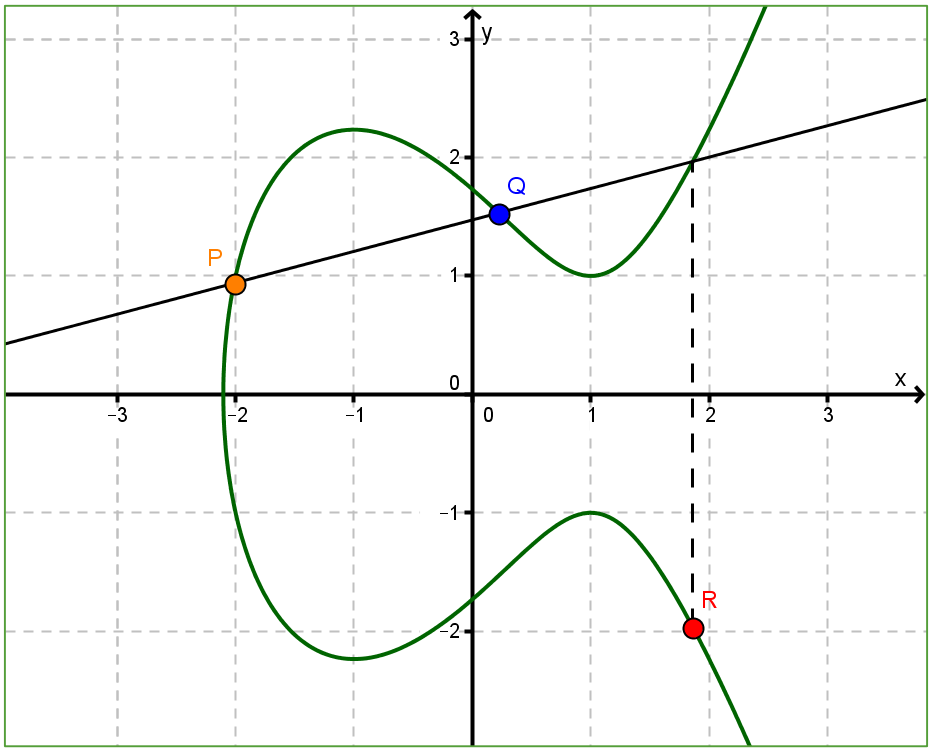
Hvis $D < 0 \ \vee D > 0$, kan man benytte den elliptiske kurve man har fundet til at danne en cyklisk gruppe, idet den ikke har mere end én rod[14].

[14] Paar, Christof og Pelzl, Jan: (2010) s. 241
[15] Hankerson, Darrel og Menezes, Alfred og Vanstone, Scott: (2004) s. 76

For rent praktisk at danne en cyklisk gruppe med elliptiske kurver, må man kigge på hvilken gruppeoperation man kan udføre. På en elliptisk kurve kan man *kun* "addere", hvilket betyder, at hvis man har to punkter og deres givne koordinater, $\mathbf{P} = (x_1, y_1) \wedge Q = (x_2, y_2)$, kan man beregne koordinaterne af et tredje punkt R så det gælder[16]:

$$\mathbf{P} + Q = R$$
$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

For at lægge to punkter sammen er der to scenarier at beskrive. Det første scenarie er, når de to punkter man lægger sammen er forskellige $\mathbf{P} \neq Q$ og i dette tilfælde kalder man det for "punkt addition". Det interessante ved elliptiske kurver er, at gruppeoperationen kan beskrives geometrisk i det kartesiske koordinatsystem:



Figur 4 - Punkt addition (et geometrisk eksempel)

Ovenfor kan det ses hvordan man lægger punktet $\mathbf{P}$ til punktet Q og får punktet R. Per definition opstiller man en linje gennem de to punkter man vil addere, derefter spejler man skæringspunktet med kurven over x-aksen og får punktet $R(x_3, y_3)$. Læg mærke til at dette kun kan lade sig gøre, fordi en elliptisk kurve altid er symmetrisk om x-aksen. Dette vil jeg nu matematisere.

Jeg ved at hældningen λ for den rette linje er differensen mellem de to punkters y-værdier, divideret med differensen mellem deres x-værdier:

[16] Paar, Christof og Pelzl, Jan: (2010) s. 244

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

Denne ligning kan jeg omforme til et udtryk for en ret linje givet punktet $P(x_1, y_1)$:

$$y - y_1 = \lambda(x - x_1)$$
$$y = \lambda(x - x_1) + y_1$$

Dette udtryk kan generaliseres yderligere til:

$$y = \lambda x + \beta$$

hvor β er skæringspunktet med y-aksen. For at finde skæringspunktet med den elliptiske kurve, kvadrerer jeg ligningen for linjen og sætter den lig kurven:

$$y^2 = (\lambda x + \beta)^2$$
$$y^2 = x^3 + ax + b$$

$$(\lambda x + \beta)^2 = x^3 + ax + b$$

Herefter benytter jeg kvadratsætningen til at udvide parenteserne og sætter alle led lig nul:

$$\lambda^2 x^2 + \beta^2 + 2\lambda x\beta = x^3 + ax + b$$
$$0 = x^3 - \lambda^2 x^2 - 2\lambda x\beta - \beta^2 + ax + b$$

Jeg har nu et tredjegradspolynomium hvor det gælder følgende for rødderne[17]: $x_1 + x_2 + x_3 = -\frac{b}{a}$ i forhold til den generelle formel for et tredjegradspolynomium: $ax^3 + bx^2 + cx + d$. Her er $a = 1 \ \wedge b = -\lambda^2$

$$x_1 + x_2 + x_3 = -\frac{-\lambda^2}{1}$$
$$x_1 + x_2 + x_3 = \lambda^2$$
$$\leftrightarrow x_3 = \lambda^2 - x_1 - x_2$$

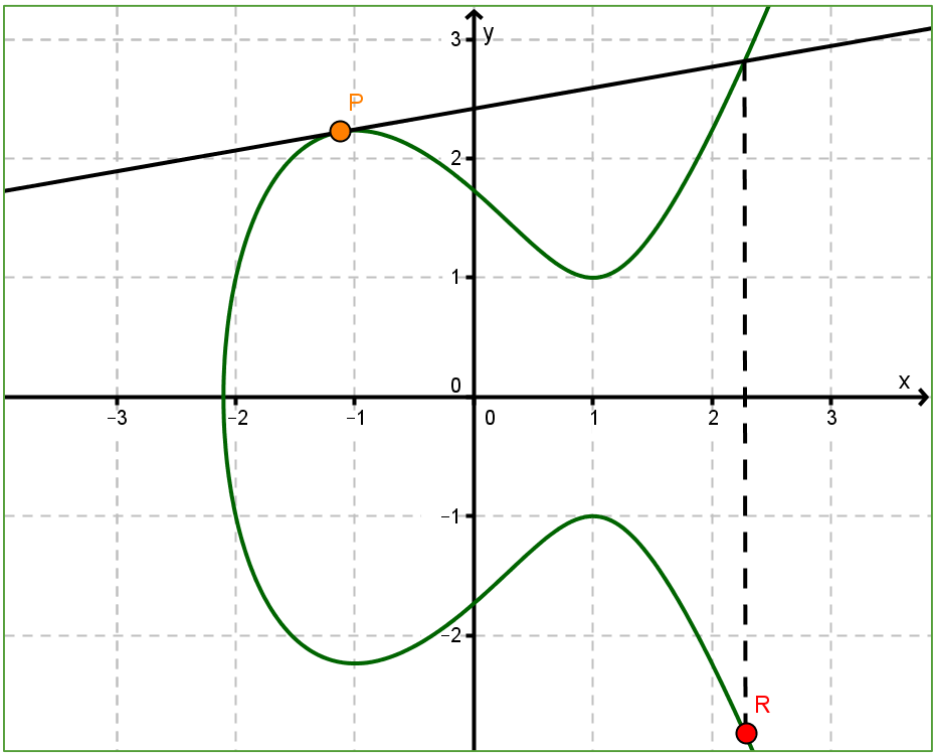
Dette udtryk for x-koordinaten x_3 af det resulterende punkt R, kan jeg indsætte i den originale ligning for linjen $y = \lambda(x - x_1) + y_1$, og derefter gange med -1 (spejle over x-aksen) for at få et udtryk for y-koordinaten y_3 af punktet R:

$$y_3 = \lambda(x_3 - x_1) + y_1$$
$$y_3 = -(\lambda(x_3 - x_1) + y_1) = \lambda(x_1 - x_3) - y_1$$

Ved punkt addition, er det nye punkt R derfor givet ved:

$$R = P + Q = (x_1, y_1) + (x_2, y_2) = (\underbrace{\lambda^2 - x_1 - x_2}_{x_3}, \underbrace{\lambda(x_1 - x_3) - y_1}_{y_3})$$

[17] Pedersen, Lars: (2010) s.103



Figur 5 - Punkt fordobling (et geometrisk eksempel)

Det andet scenarie er, at de to punkter man skal addere er ens: $P = Q$. I dette tilfælde er det ikke muligt at finde hældningen af den rette linje med differensen mellem to punkter, derfor finder man i stedet hældningen af tangenten i punktet P. Givet den elliptiske kurve, er det blot at differentiere kurven med kædereglen for at få hældningen $\lambda = y'$:

$$y^2 = x^3 + ax + b \leftrightarrow y = \sqrt{x^3 + ax + b} = (x^3 + ax + b)^{\frac{1}{2}}$$

$$y' = \frac{d(x^3 + ax + b)^{\frac{1}{2}}}{d(x^3 + ax + b)} \cdot \frac{d(x^3 + ax + b)}{dx} = \frac{1}{2} \cdot (x^3 + ax + b)^{\frac{1}{2}-1} \cdot (3 \cdot x^{3-1} + a) = \frac{3x^2 + a}{2\sqrt{x^3 + ax + b}}$$

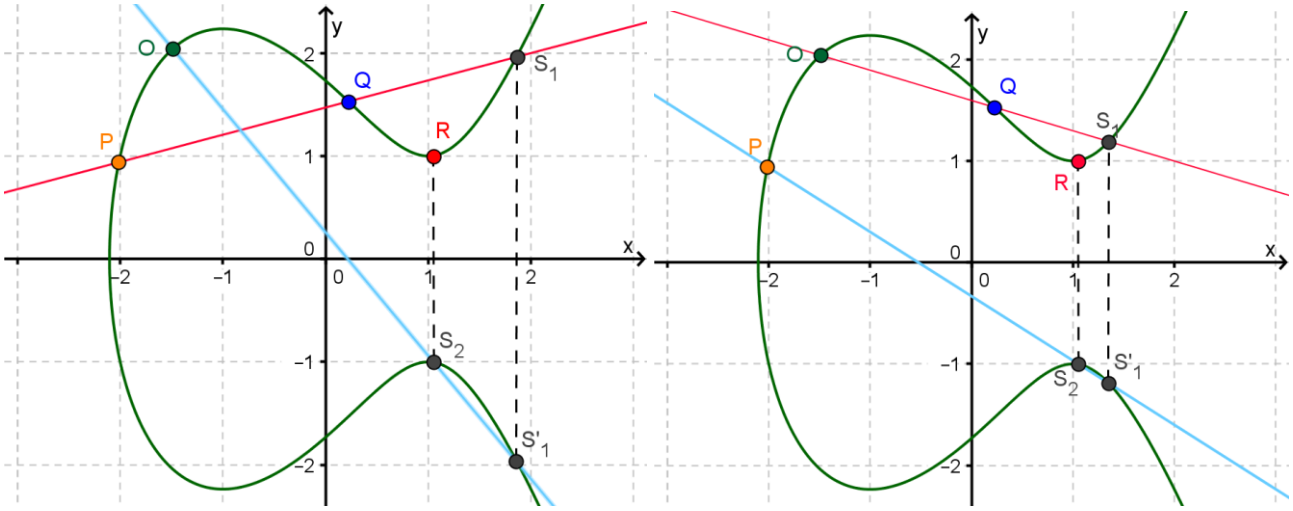
Fra ovenstående udtryk kan man se at x og $y = \sqrt{x^3 + ax + b}$ indgår, hvorfor man blot med ét punkt (x_1, y_1) kan bestemme hældningen af tangenten:

$$\lambda = \frac{3x^2 + a}{2\sqrt{x^3 + ax + b}} = \frac{3x_1^2 + a}{2y_1}$$

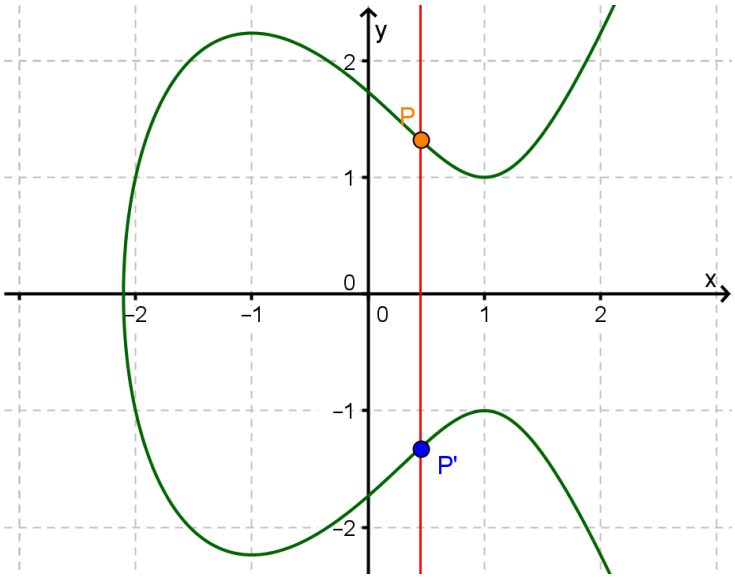
Da jeg allerede har udledt formlen for det resulterende punkt, når man kender hældningen λ og to punkter (her er de to punkter blot ens), undlader jeg at skrive den op endnu engang. Det skal dog bemærkes at når $P = Q$, så hedder gruppeoperationen punkt fordobling. Spørgsmålet er nu om den cykliske gruppe man har dannet med en elliptisk kurve, overholder de generelle regler for en gruppe (Liste 1).

Den første regel er at operationer i en gruppe skal være lukkede, hvilket for den elliptiske kurve betyder, at ethvert nyt punkt man finder, er på kurven. Med ovenstående regler for punkt addition og punkt fordobling, vil der altid være et nyt punkt R.

Den anden regel er, at operationerne skal være associative, altså at rækkefølgen er ligegyldig. Dette kan hurtigt bevises geometrisk:



Jeg vil prøve at addere tre punkter: $\mathbf{O}$, $\mathbf{P}$, og $\mathbf{Q}$. Til venstre ses udregningen: $(\mathbf{P}+\mathbf{Q})+\mathbf{O}$ og til højre ses udregningen $(\mathbf{O}+\mathbf{Q})+\mathbf{P}$. Det kan hurtigt ses, at det resulterende punkt $\mathbf{R}$ er ens i begge tilfælde, hvorfor gruppen er associativ. Fra ovenstående kan det også bemærkes at $\mathbf{P}+\mathbf{Q}=\mathbf{Q}+\mathbf{P}$ hvorfor den også er kommutativ. De to sidste regler for at elliptiske kurver kan defineres som en cyklisk gruppe vi kan benytte til kryptografi er, at der skal findes et neutralt element (identitet) og at der altid skal eksistere et inverst element til et punkt på kurven:

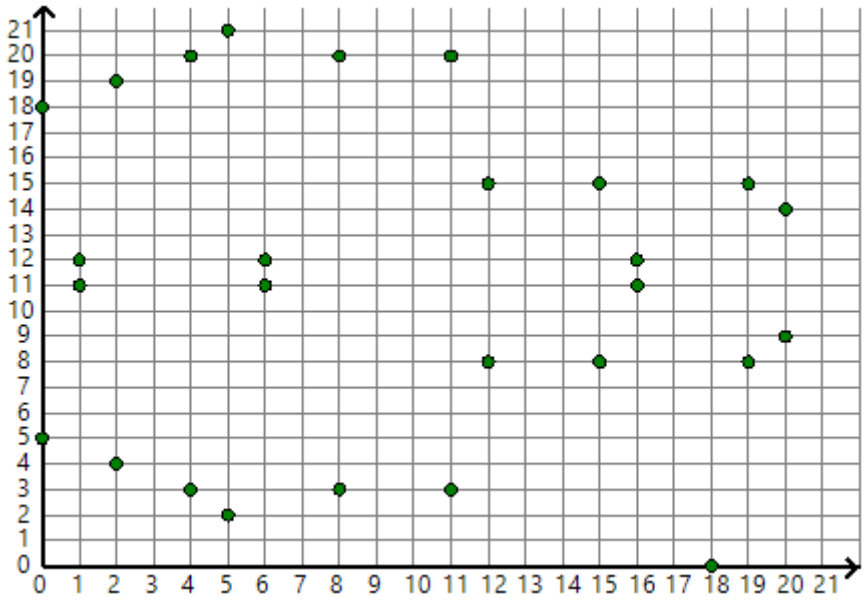


Til venstre kan det ses, at hvis man lægger uendelig til punktet P:$(\mathbf{P}+\infty)$, får man skæringspunktet P'. Dette punkt reflekteres per definition over x-aksen, hvorfor $\mathbf{P}+\infty=\mathbf{P}$. Der findes altså et neutralt element.

Derudover kan det også ses, at hvis man lægger $\mathbf{P}+\mathrm{P}'$ sammen får man uendelig. Da refleksionen af uendelig er uendelig må det gælde at: $\mathbf{P}+\mathrm{P}'=\infty$. Det betyder altså at refleksionen af et punkt er punktets inverse element:
$-\mathbf{P}=(x_1,-y_1)$.

Kryptering med Elliptiske Kurver

Indtil videre har jeg udelukkende vist elliptiske kurver over alle reelle tal $\mathbb{R}$. I kryptografi er det mere ønsket at have elliptiske kurver over finitte grupper F_p[14]. Fordi kurverne dermed består af færre punkter, er det ikke klart hvordan de geometriske relationer kan anvendes (altså hvordan man skal forbinde punkterne). Det gør dem perfekte til at kryptere med[18]. Nedenfor ses et plot af den elliptiske kurve: $y^2 \equiv x^3 + 3x + 2 \bmod 23$:



Figur 6 - Plot af $y^2 \equiv x^3 + 3x + 2 \bmod 23$ - Via mit eget program

Det man hurtigt lægger mærke til er, at der ligesom med plottet over alle reelle tal forefindes en symmetriakse. I dette tilfælde ligger den ved $y = 11.5$. Med denne finitte cykliske mængde defineret ved hjælp af elliptiske kurver, kan jeg vise én af de måder man kan bruge ECC til at kryptere med. Det skal dog bemærkes at asymmetrisk kryptografi sjældent bruges til at kryptere klartekst, fordi symmetriske algoritmer er hurtigere til dette[19]. Nedenfor vil jeg vise hvordan man krypterer med "Elliptic Curve Encryption Scheme"[20]:

Jeg tager udgangspunkt i eksemplet med den elliptiske kurve $y^2 \equiv x^3 + 3x + 2 \bmod 23$, hvor $n = 27$ er antallet af punkter og $a = 3$. Der vælges nu et primitivt element i gruppen, et punkt kaldet $P = (0,5)$. Dette punkt genererer en cyklisk undergruppe således:

$$\{\infty, P, 2P, 3P, \dots, (n-1)P\} \rightarrow \{\infty, (0;5), (4;3), (2;19), \dots, (4;20)\}$$

Ovenstående notation betyder at P skal adderes med sig selv x antal gange for at danne punktet.

[18] Certicom Research (b)
[19] Paar, Christof og Pelzl, Jan: (2010) s. 154
[20] Hankerson, Darrel og Menezes, Alfred og Vanstone, Scott: (2004) s. 14

Det første der skal gøres er, at Bob genererer en offentlig og en privat nøgle. Han starter med tilfældigt at vælge en privat nøgle d:

$$d \in_R [1, n-1]$$

Så længe dette heltal d er større end 0 og mindre end gruppens kardinalitet (størrelse), kan d bruges (nærmere beskrevet i beviset Bilag 2). Herefter beregner han den offentlige nøgle Q, et punkt som Alice skal benytte til at kryptere en besked til Bob. Dette gør han ved at addere punktet P med sig selv $d = 3$ gange. Med formlerne udledt for punkt addition og fordobling udregnes punktet $Q = dP = 3P$, nu i den finitte cykliske mængde F_{23}:

$2P$ udregnes ved punkt fordobling idet de to punkter der skal adderes er ens:

$$\lambda = \frac{3x_1^2 + a}{2y_1} = (2 \cdot 5)^{-1} \cdot (3 \cdot 0^2 + 3) \equiv 7 \cdot 3 \equiv 21 \bmod 23$$

$$R = \left(\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1\right)$$
$$R_{x,1} = 21^2 - 0 - 0 = 441 \equiv 4 \bmod 23$$
$$R_{y,1} = 21(0 - 4) - 5 = -89 \equiv 3 \bmod 23$$

$$2P = (\underbrace{4}_{x_2}, \underbrace{3}_{y_2})$$

$3P$ udregnes ved punkt addition $2P + P$, hvor $P(x_1, y_1) = (0,5)$:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = (y_2 - y_1) \cdot (x_2 - x_1)^{-1} = (3 - 5) \cdot (4 - 0)^{-1} = -2(4)^{-1} \equiv -2 \cdot 6 \equiv 11 \bmod 23$$

$$R_{x,2} = 11^2 - 4 - 0 = 117 \equiv 2 \bmod 23$$
$$R_{y,2} = 11(0 - 2) - 5 = -27 \equiv 19 \bmod 23$$

$$Q = 3P = (2,19)$$

Dette var tænkt som et eksempel på hvordan man benytter formlerne, fremover viser jeg blot resultatet. Derudover kan algoritmen til at finde den modulære inverse ses i Bilag 3.

Nu skal Alice repræsenterer sin besked, i dette eksempel beskeden ”HEJ”, som punkter i den finitte cykliske gruppe. For at repræsenterer en besked i et punkt i den elliptiske kurves finitte cykliske gruppe, er der flere fremgangsmåder. Den typiske er at finde et punkt hvor forskellen mellem y og x værdien repræsenterer det bogstav man vil sende:

Punktdifferens	$(2,4) = 4 - 2 = 2$	$(12,15) = 15 - 12 = 3$	$(0,5) = 5 - 0 = 5$
Bogstav	$H = 2$	$E = 3$	$J = 5$

Tabel 3

Ovenfor har jeg udtaget tre punkter fra Figur 6, hvor differensen mellem y og x værdierne (punktdifferensen) er beregnet. Derefter har jeg symboliseret disse punktdifferenser som de bogstaver Alice godt vil sende H, E og J. På den måde har jeg lavet en slags "ordbog", hvor man eksempelvis kan spørge den: Hvilket bogstav har du, hvis jeg giver dig tallet 3? Svaret vil her være: "E". Beskeden "HEJ" kan altså nu repræsenteres som punkterne $M_1 = (2,4), M_2 = (12,15)\ og\ M_3 = (0,5)$.

Alice vælger nu tilfældigt et heltal $k = 5$, hvor $k \in_R [1, n-1]$. Derefter beregner hun to punkter $C_1 = kP\ \wedge C_2 = M + kQ$, som repræsenterer chifferteksten for hver bogstav hun vil sende:

$$C_1 = 5P = (0,5) + (0,5) + (0,5) + (0,5) + (0,5) = (12,15)$$
$$C_{2,1} = M_1 + 5Q = (2,4) + 5(2,19) = (19,8)$$
$$C_{2,2} = M_2 + 5Q = (12,15) + 5(2,19) = (11,20)$$
$$C_{2,3} = M_3 + 5Q = (0,5) + 5(2,19) = (19,15)$$

Chiffer teksten fra Alice består nu af følgende punkter: $(12,15), (19,8), (11,20), (19,15)$.

Bob kan dekryptere beskeden fra Alice ved at beregne $M_n = C_{2,x} - dC_1$ for hver besked, og derefter ekstrahere bogstaverne derfra:

$$dC_1 = 3C_1 = 3(2,15) = (6,12)$$

$$\begin{aligned} M_1 = C_{2,1} - dC_1 &= (19,8) - (6,12) \\ &= (19,8) + (6, -12 \bmod 23) \\ &= (19,8) + (6,11) = (\mathbf{2}, \mathbf{4}) \end{aligned}$$

$$\begin{aligned} M_2 = C_{2,2} - dC_1 &= (11,20) - (6,12) \\ &= (11,20) + (6,11) = (\mathbf{12}, \mathbf{15}) \end{aligned}$$

$$\begin{aligned} M_3 = C_{2,3} - dC_1 &= (19,15) - (6,12) \\ &= (19,15) + (6,11) = (\mathbf{0}, \mathbf{5}) \end{aligned}$$

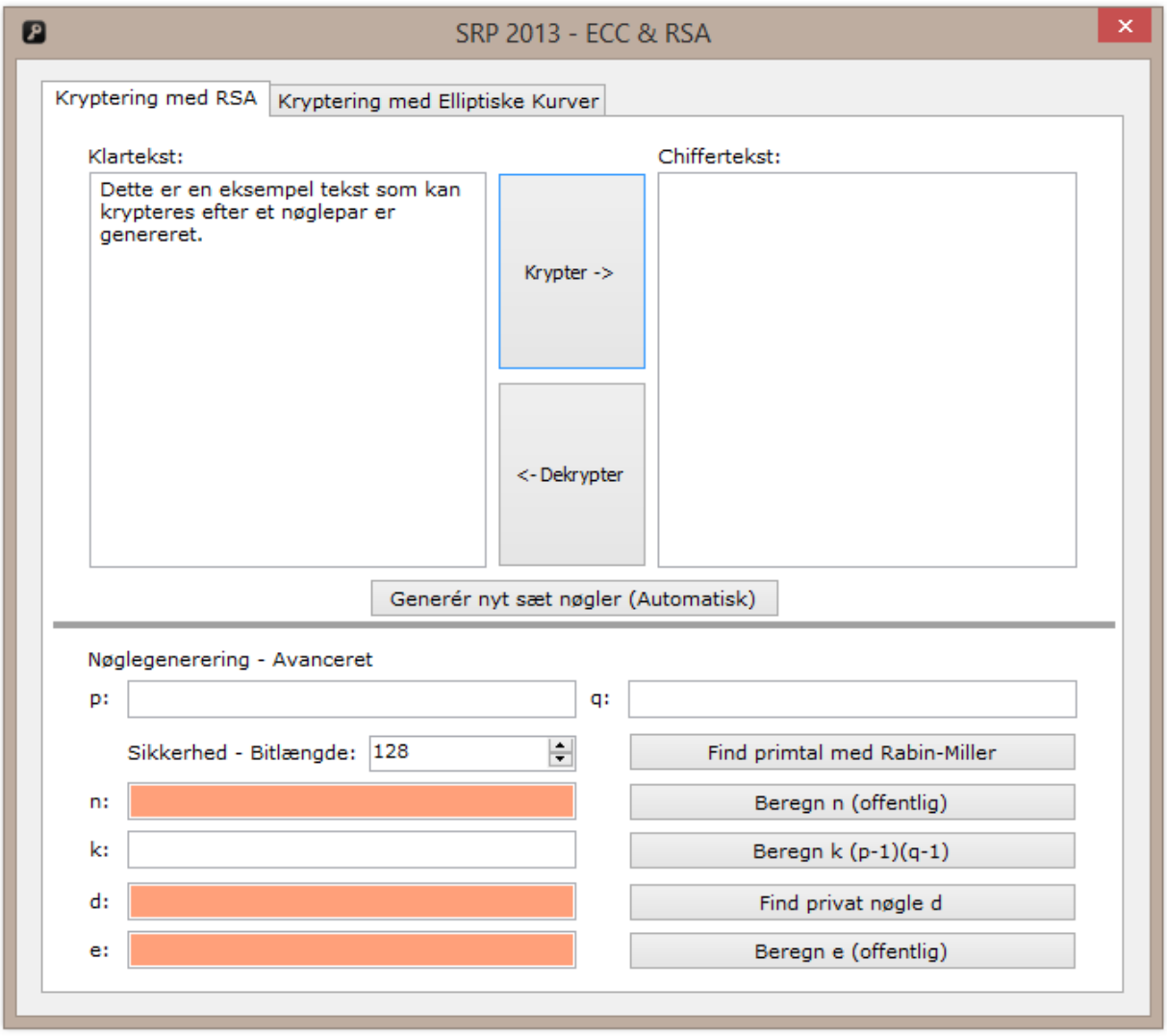
Som det ses har bob nu fået de samme punkter som Alice krypterede, og han kan nu beregne differensen mellem y og x-koordinaterne for at finde frem til de tal, og med hjælp fra ordbogen (Tabel 3), de bogstaver som Alice sendte til ham:

$$M_{1_y} - M_{1_x} = 4 - 2 = 2 = "H"$$
$$M_{2_y} - M_{2_x} = 15 - 12 = 3 = "E"$$
$$M_{3_y} - M_{3_x} = 5 - 0 = 5 = "J"$$

I praksis er den finitte cykliske gruppe flere tusinde gange større, her har jeg blot valgt små tal for forståelsens skyld. Beviset for at kryptering og dekryptering i ECC fungerer er beskrevet i Bilag 2.

Grafisk implementering af ECC og RSA

Nu har jeg gennemgået grundlæggende matematik bag kryptering og ECC, og vist hvordan man rent praktisk kan bruge samme til at kryptere en hemmelig besked. At udføre ovenstående trin for at kryptere enten i RSA eller ECC er langsommeligt og ikke særlig tilgængeligt for den gængse bruger. Derfor er det nærliggende at programmere et program med grafisk brugerflade der gør kryptering med både RSA og ECC lettere. Implementeringen af disse vil give en dybere RSA og ECC forståelse, samt bidrage til en senere sammenligning og vurdering af kryptosystemerne.

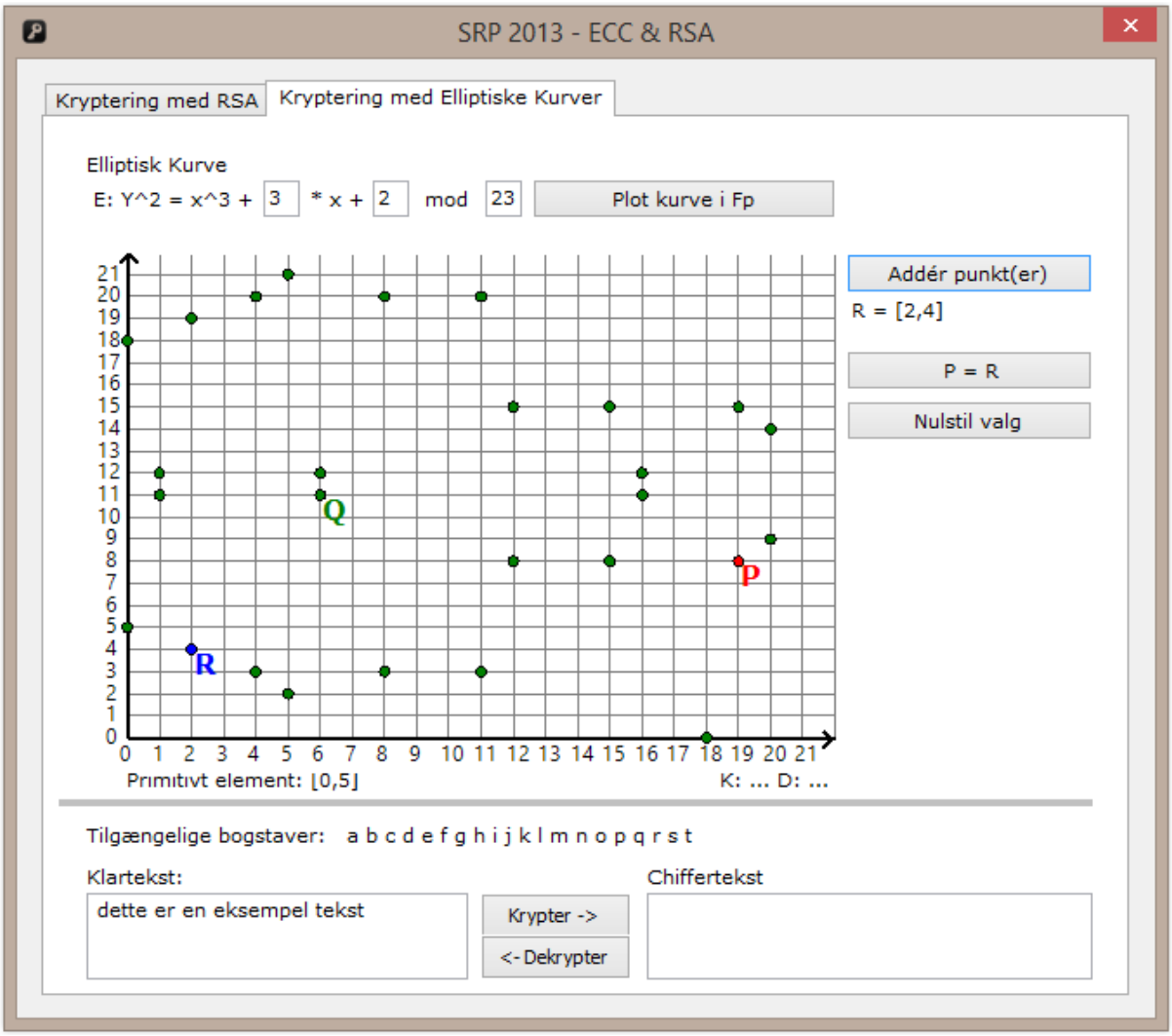


Figur 7 - Mit eget program – Kryptering med RSA

Ovenfor ses mit program med grafisk brugerflade, koden og et link til programmet kan findes i Bilag 4. Det består af to faner ”Kryptering med RSA”, og ”Kryptering med Elliptiske Kurver”, hvilket gør det let for brugeren at vælge hvilket kryptosystem han/hun vil benytte sig af. Hvis en bruger

nu med det samme vælger at trykke på knappen "Krypter ->", vil de felter der mangler at blive udfyldt få samme rødlige farve. Her gælder altså gestaltloven "Loven om lighed"[21], hvilket sætter brugeren i stand til hurtigt at identificere sine fejl. Knappen "Generér nyt sæt nøgler (Automatisk)", er til brugeren der ikke har så megen viden om RSA, men blot gerne vil kryptere sin klartekst til chiffertekst, medens den avancerede bruger under den grå horisontale linje, selv kan eksperimentere med forskellige nøglepar. Den grå linje er derfor med til at indramme programmet i dele, så det er let for brugeren at se hvor han/hun hører hjemme (jf. loven om lukkethed)[21].

Den anden fane giver som sagt brugeren mulighed for at kryptere med ECC:



Figur 8 - Mit eget program - Kryptering med ECC

Ovenfor kan fanen "Kryptering med Elliptiske Kurver" ses valgt. Det første man skal gøre er at trykke på "Plot kurve i F_p", derefter vil alle nøgler, primitive elementer mv. blive genereret. Den elliptiske kurves værdier: $a = 3 \ \wedge b = 2 \ \wedge p = 23$ er valgt på forhånd til den ikke-avancerede

[21] Gamborg, Niels.

bruger, men kan selvfølgelig indstilles af den avancerede. Igen adskilles det avancerede felt hvor man kan eksperimentere, og der hvor man blot kan kryptere, med en grå horisontal linje. Plottet har jeg gjort interaktivt, hvilket vil sige at man ved at venstre klikke på et punkt vælger punktet P, og ved at højreklikke vælger punktet Q. Derefter kan man lægge de to punkter sammen ved at trykke på "Addér punkt(er)". Eksemplet på billedet er taget fra krypteringseksemplet med ECC hvor følgende udregn laves: $R = P + \boldsymbol{Q} = (19{,}8) + (\boldsymbol{16, 11}) = (2{,}4)$.

Nede under den grå linje kan man se de bogstaver man har til rådighed, hvilket er begrænset af den cykliske gruppe F_p som kurven er defineret over. Da jeg i eksemplet med ECC har benyttet mig af en lille gruppe F_{23}, gør jeg brugeren opmærksom på hvilke tegn han/hun har til rådighed.

Det at det generelt gøres let for brugeren at benytte programmet, er med til at sænke guessability-gulfen (den tid det tager at gætte sig til hvordan man skal benytte et program)[22], og dermed også at gøre det mere tilgængeligt for den gængse bruger.

I tabellen herunder ses forskellige tekster krypteret med henholdsvis ECC og RSA via mit program:

ECC $k = 10 \wedge d = 13$ $y^2 \equiv x^3 + 3x + 2 \bmod 23$		**RSA (16-bit)** $n = 25283 \wedge e = 7229 \wedge d = 21269$	
Klartekst	**Chiffertekst**	**Klartekst**	**Chiffertekst**
eksempeltekst	[8,20]|[15,8] [0,0] [4,3] [15,8] [8,20] [16,11] [15,8] [4,20] [19,8] [15,8] [0,0] [4,3] [19,8]	eksempeltekst	23674 17661 6146 23674 11279 112 23674 2014 23817 23674 17661 6146 23817
tester lige med space	[8,20]|[19,8] [15,8] [4,3] [19,8] [15,8] [1,11] [5,2] [4,20] [12,15] [19,15] [15,8] [5,2] [8,20] [15,8] [6,12] [5,2] [4,3]	"æøå" virker det?	4966 594 22810 25648 4966 20065 12585 18273 25973 15814 31509 25973 20065 20908 31509 29241 1930
dasdbhjcr	[8,20]|[6,12] [16,12] [4,3] [6,12] [20,14] [5,21] [11,3] [18,0] [1,11] [16,12] [4,3] [19,15] [5,21] [6,12]	#=sldæ&/(	11158 23146 18866 31911 20908 594 12257 11002 27069

Tabel 4 – Afprøvning af mit program med flere tekster

Læg mærke til at krypteringen med ECC ikke har et lige så stort antal tilgængelige bogstaver grundet den mindre cykliske mængde den er defineret over. Derfor er der ikke helt ens eksempeltekst. Under implementeringen af modulær aritmetik i C#, var der en del algoritmer der skulle være på plads. Især under kryptering med RSA hvor man beregner tal opløftet i enorme eksponenter. Hertil bruges algoritmen ”Eksponentiering ved kvadrering”. Derudover skulle der både til ECC og RSA bruges en

[22] Jordan, Patrick W: (1998) s. 11-12

metode til at beregne den modulære inverse til et element, algoritmen hertil hedder "Euklids udvidede algoritme". Begge algoritmer, samt interessante kodestykker ses forklaret i Bilag 3.

Sikkerhed - ECC vs. RSA

For at kunne vurdere hvilket kryptosystem der er sikrest, er det essentielt at definere hvad der forstås ved begrebet sikkerhed, inden for kryptering. Hertil findes Kerckhoffs princip som siger at: Sikkerheden bag en krypteringsalgoritme ikke skal være baseret på at algoritmen bag systemet er hemmelig, men alene på at den private nøgle er ukendt. Det er fristende at lave et kryptosystem hvor implementeringsdetaljerne er skjulte, og man derfor tror at systemet er mere sikkert (sikkerhed via obskuritet). Dog har militære erfaringer og omvendt konstruktion (Reverse Engeneering) vist at disse systemer er svage[23]. Selvom der har været patent på RSA, og stadig er patent på nogle former for ECC kryptering, er metoden/algoritmen bag krypteringsformerne velkendt[24]. Både ECC og RSA's sikkerhed beror på den private nøgle d's hemmelighed, hvorfor de begge er sikre i henhold til Kerchhoffs princip.

Når man måler sikkerhedsniveauet af krypteringsalgoritmer, ser man på hvor svært det er at bryde sikkerheden for det bedst kendte angreb mod algoritmen. RSA's algoritme er som sagt bygget på, at det er svært at primtalsfaktorisere[25], medens ECC bygger på det diskrete logaritme problem[26]. Lige nu forholder det sig sådan, at der findes kendte angreb så som index-calculus metoden[27], som kan løse almindelige diskrete logaritme problemer (ikke ECDLP). Derudover kan primtalsfaktoriseringen som RSA bygger på, regnes effektivt i sub-eksponentiel tid, ved hjælp at ECC[28]. Sikkerheden af RSA er altså mindre end ECC fordi der ikke findes kendte angreb på ECC hvis parametrene af den elliptiske kurve vælges omhyggeligt[29].

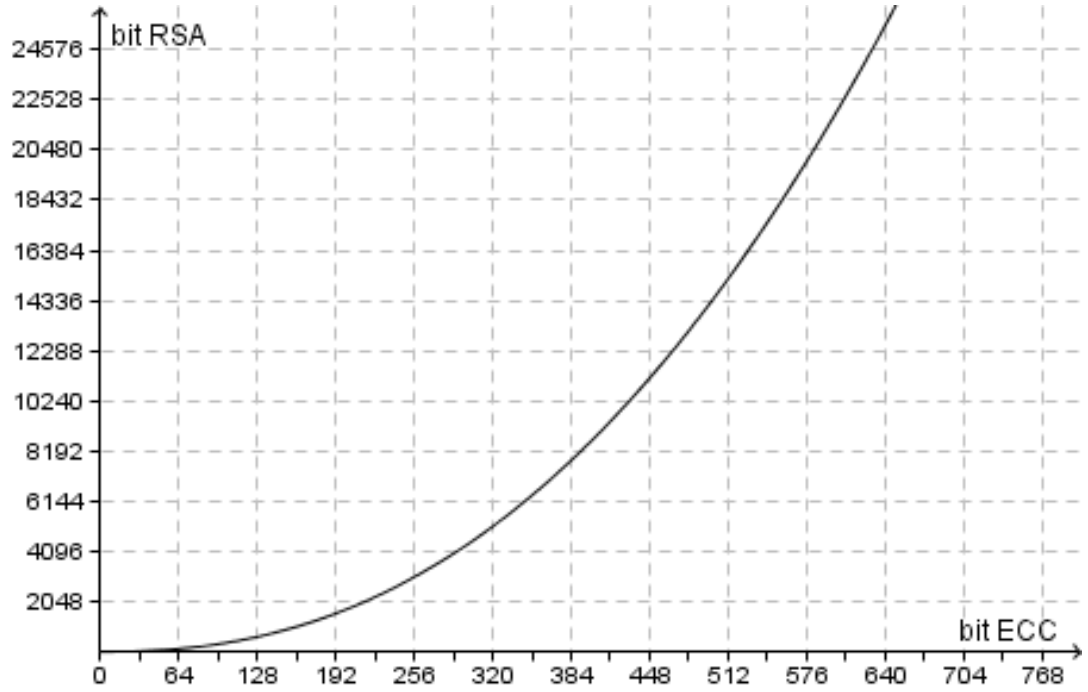
Mere specifikt kan sikkerhedsniveauet være angivet i n bits for en symmetrisk algoritme, og opgøres i det antal trin det bedste angreb på algoritmen tager[30]: 2^n trin. Da både RSA og ECC ikke er symmetriske algoritmer, måler man det i stedet relativt i forhold til symmetriske algoritmer:

Algoritme familie	Kryptosystem	Sikkerhedsniveau (bit)			
		80	128	192	256
Primtalsfaktorisering	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Diskrete logaritme	DH, DSA, ELgamal	1024 bit	3072 bit	7680 bit	15360 bit
Ellipstiske kurver	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetrisk-nøgle	AES, 3DES	80 bit	128 bit	192 bit	256 bit

Tabel 5 – ***Paar, Christof og Pelzl, Jan: (2010) - Table 6.1***

[23] Paar, Christof og Pelzl, Jan: (2010) s. 11, 6
[24] Paar, Christof og Pelzl, Jan: (2010) s. 174
[25] Nichols, Randall K. og Lekkas, Panos C: (2002) s.228
[26] Nichols, Randall K. og Lekkas, Panos C: (2002) s.232
[27] Hankerson, Darrel og Menezes, Alfred og Vanstone, Scott: (2004) s. 165
[28] Link til beregner: http://alpertron.com.ar/ECM.HTM
[29] Paar, Christof og Pelzl, Jan: (2010) s. 155

Ovenstående tabel viser hvor store de tal man arbejder med i de forskellige kryptosystemer skal være, for at få et tilsvarende sikkerhedsniveau. Man kan hurtigt se, at asymmetriske algoritmer kræver langt større bit-længder (antal bits i et givent tal) end de symmetriske, for samme sikkerhed. En konklusion på denne tabel er derfor: Det er en uønsket effekt ved asymmetriske algoritmer, at for at de skal være sikre, skal deres bit-længder være meget store. Dette er uønsket fordi de dermed er ekstremt beregningstunge. Man vil gerne undgå, at det tager så lang tid at kryptere en given klartekst, at en implementering i praksis ikke er brugbar. Hvad mere interessant er, at 256-bit ECC svarer til 3072 bit RSA[30]. Via regressionsanalyse af ovenstående tabel kan forholdet mellem ECC og RSA visualiseres:



Figur 9 – Forholdet mellem ECC og RSA bit-længder

Det er nu lettere at bemærke, hvordan RSA hurtigt skal bruge meget større tal, for at opnå samme sikkerhedsniveau som ECC. Dette sikkerhedsniveau illustreres i Figur 8. Man kan se hvor svært det er at forestille sig hvordan man er kommet fra et punkt P til et andet punkt R, altså hvor mange "hop" der skal til for at komme fra et punkt til et andet. ECC kan altså ved langt mindre bitlængder være lige så sikker som RSA med sine store beregningstunge bitlængder. Selvom der stadig findes flest computere der benytter RSA og DLP, begynder alle nyere sikkerhedssystemer at være baseret på ECC. Fordi ECC er så sikkert selv med små tal, gør det ECC perfekt til mobile enheder, hvis processorkraft ikke rækker ligeså langt som desktop-computere[31] [32].

[30] Paar, Christof og Pelzl, Jan: (2010) s. 156
[31] Paar, Christof og Pelzl, Jan: (2010) s. 253
[32] Nichols, Randall K. og Lekkas, Panos C: (2002) s.237

Status på RSA og fremtiden

Ud fra ovenstående analyse af sikkerheden bag RSA og ECC, kan det opsummerende siges, at med en fremtid med flere mobile enheder, og med et evigt stigende krav til højere og højere sikkerhed, ser elliptisk kurve kryptografi mest lovende ud. Spørgsmålet er nu hvor RSA placerer sig i forhold her til.

I løbet af det seneste årti er verden blevet mere globaliseret. Vi lever i en verden hvor det er nemt at udveksle forskningsresultater med hinanden, hvilket har sat rigtig skub i udviklingen og forskningen af blandt andet kryptosystemer[33]. For at give et bud på hvor vi står i dag med henblik på det nuværende mest anvendte kryptosystem RSA, giver det mening at kigge tilbage i tiden for ikke blot at kunne sige noget om status nu, men også om fremtiden.

Det smarte ved et globalt samfund er også, at der er flere til at teste og som sagt udveksle idéer og angreb på eksempelvis RSA. I 1996 var det muligt at faktorisere en 130-bit nøgle fra RSA[34], men blot 14 år efter kunne man faktorisere tal med næsten 6 gange større bitlængder (768-bit RSA[35]). D. 18 december 2013 blev det offentliggjort, at man nu havde fundet en metode til at uddrage hele 4096-bit RSA dekrypteringsnøgler fra forskellige computere[36]. Taget i betragtning af at den generelle anbefaling til langsigtet sikkerhed er at bruge 3072-bit nøgler i RSA[37], kan man godt se tendensen: at processorkraften i moderne computere er ved at hale ind på RSA.

Ved en Black Hat konference (konference for hackere) i August i år, udtalte eksperter sig, at indenfor 5 år vil RSA kryptosystemet være forældet. Hovedbudskabet var at enden på RSA er uundgåelig. Denne advarsel er kommet på baggrund af at matematikere og kryptoanalytikere har gjort store fremskridt i form af studier i at bryde RSA[38]. The National Security Agency i USA har endda udeladt RSA fra deres liste over anbefalede kryptosystemer, hvilket indikerer at tiden for RSA er ved at løbe ud[39].

Da RSA er indbygget i næsten alle af nutidens applikationer, vil det have store konsekvenser for verden hvis RSA brydes. Selvfølgelig kan RSA blive ved med at være sikker hvis man øger bitlængden af nøglerne til 16384-bit (tal med 4933 cifre). Dette vil dog som forventet resultere i langsommere kryptering og længere svar tider. Og fordi RSA er en del af TLS (Transport Layer Security)[40], som næsten al online handel benytter sig af (typisk brugt under den velkendte https:// protokol), vil en skræmmende konsekvens være, at der kunne snydes med al online handel baseret på RSA. Hvis

[33] Green, Tim: (2013)
[34] Dobbertin, Hans: (1996)
[35] Kleinjung, Thorsten mf.: (2010)
[36] Genkin, Daniel og Adi Shamir og Eran Tromer: (2013)
[37] Paar, Christof og Pelzl, Jan: (2010) s. 288
[38] Mimoso, Michael: (2013)
[39] NSA: (2013)
[40] Sarkar, Pratik & Shawn Fitzgerald: (2013)

verden ikke når at skifte til et mere sikkert kryptosystem som ECC inden da, vil have implikationerne være af helt uoverskuelige dimensioner.

Det er svært at forstille sig situationen, men et bud herpå ville have en særdeles hypotetisk karakter. Det vil være muligt at få adgang til andre folks bankkonti og andre personlige oplysninger, hvorved hele finansmarkedet vil kollapse. Tilliden til penge vil være ikke eksisterende, fordi de ikke mere har nogen fortrolighed. Personlige oplysninger om folk, liggende hos kommunen, politiet eller hos staten, vil kunne udnyttes. Alt det vi troede blev behandlet fortroligt, ville nu være frit tilgængeligt, og vi ville være nødt til at bygge en ny digital samfundsstruktur op.

Konklusion

Der findes grundlæggende to forskellige måder at kryptere på, asymmetrisk kryptering hvor to forskellige nøgler bliver brugt til at kryptere og dekryptere, og symmetrisk kryptering hvor nøglerne er ens. Inden for kryptering er modulær aritmetik vigtig, fordi det sætter os stand til at beskrive en begrænset mængde af tal, hvorved finitte cykliske grupper dannes. Disse består af et primitivt element hvilket opløftet i en eksponent giver alle gruppens elementer i en tilsyneladende arbitrær rækkefølge, hvilket er basis for de fleste kryptosystemer.

De to seneste asymmetriske kryptosystemer RSA og ECC, er begge opbygget ved hjælp af finitte cykliske grupper, men det grundlæggende matematiske problem de bygger på er forskelligt. RSA bygger på, at det er svært tidsmæssigt at primtalsfaktorisere store tal, medens ECC bygger på det generaliserede diskrete logaritme problem: Hvis kun start og slutpunkt på en elliptisk kurve er kendt, hvor mange gange ”hoppede” man så rundt, for at komme fra start til slut på kurven.

Via et program med grafisk brugerflade gøres det let for den gængse bruger at eksperimentere med kryptering, i de ellers forholdsvist matematisk tunge kryptosystemer RSA og ECC. Og med et evigt stigende krav til højere og højere sikkerhed, ser elliptisk kurve kryptografi mest lovende ud. RSA skal hurtigt bruge meget større nøgler, for at opnå samme sikkerhedsniveau som ECC. Derudover er sikkerheden af ECC større på baggrund af, at der ikke findes nogle kendte angreb mod det generaliserede diskrete logaritme problem (ECDLP) som ECC bygger på.

Både eksperter og forskningsresultater viser, at enden på RSA er uundgåelig, at processorkraften i moderne computere er ved at hale ind på RSA. Hvis dette sker inden ECC har erstattet RSA, vil det få uanede implikationer. Alt det vi troede var fortroligt (stats-, kommune- og bankoplysninger), vil nu kunne tilgås af enhver der kan bryde RSA, og vi vil være nødt til at bygge en ny digital samfundsstruktur op. Kryptografi er ikke et statisk begreb, og derfor skal man være på vagt overfor aldrende kryptosystemer.

Referencer

BØGER

- Cohen, Henri og Frey, Gerhard. *Elliptic and Hyperelliptic Curve Cryptography*. Chapmon & Hall/CRC, 2006
- Gadegaard, Henrik G. og Hansen, Johan P. *Algebra og talteori*. 1. udg. Gyldendal, 2002
- Hankerson, Darrel og Menezes, Alfred og Vanstone, Scott. *Guide to Elliptic Curve Cryptography*. Springer, 2004
- Hansen, Johan P. *Tal og mængder – Begreber, metoder og resultater*. Aarhus Universitetsforlag, 2012
- Jordan, Patrick W. *An Introduction to Usability.* CRC Press, 1998
- Nichols, Randall K. og Lekkas, Panos C. Wireless Security. MacAllister Publishing Services, 2002
- Pedersen, Lars. *Matematik 112 – Førstehjælp til formler*, 3.udgave, 2.rettede oplag, 2010
- Paar, Christof og Pelzl, Jan. *Understanding Cryptography – A Textbook for Students and Practitioners*. Springer, 2010 (***Hovedbog***)
- Riber, Peter og Frandsen, Jesper (red.). *Kryptering*. 2. udg. Systime, 2008

ARTIKLER

- Certicom Research (a). *SEC1: Elliptic Curve Cryptography*, 2000, http://www.secg.org/collateral/sec1_final.pdf, besøgt d. 17/12-2013
- Dobbertin, Hans. *CryptoBytes.* RSA labatories, 1996 ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf, besøgt d. 16/12-2013
- Genkin, Daniel og Adi Shamir og Eran Tromer. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. 2013, http://www.cs.tau.ac.il/~tromer/acoustic/, besøgt d. 19/12-2013
- Green, Tim. *Black Hat: Elliptic curve cryptography coming as smarter algorithms threaten RSA*. NetworkWorld (b), 2013 http://www.networkworld.com/news/2013/080213-black-hat-elliptical-curve-cryptography-272476.html, besøgt d. 18/12-2013 https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf, besøgt d. 18/12-2013
- Kleinjung, Thorsten mf. *Factorization of a 768-bit RSA modulus.* 2010, http://eprint.iacr.org/2010/006.pdf, besøgt d. 18/12-2013
- Mimoso, Michael. *CRYPTO GAINS RAMP UP CALLS TO GET AHEAD OF INEVITABLE RSA ALGORITHM DOWN-FALL*. Threatpost, 2013, http://threatpost.com/crypto-gains-ramp-up-calls-to-get-ahead-of-inevitable-rsa-algorithm-down-fall/101560, besøgt d. 17/12-2013
- NetworkWorld (a), *Security's inseparable couple*. 2005, http://www.networkworld.com/news/2005/020705widernetaliceandbob.html, besøgt d. 14/12-2013
- Sarkar, Pratik & Shawn Fitzgerald. *Attacks on SSL*. iSEC Partners, 2013.

HJEMMESIDER

- Bauer, Johannes. *Elliptic Curve Cryptography Tutorial*. Johannes Bauer, http://www.johannes-bauer.com/compsci/ecc/, besøgt d. 15/12-2013
- Certicom Research (b). *ECC Tutorial*. http://www.certicom.com/index.php/ecc-tutorial, besøgt d. 14/12-2013
- Cruise, Brit. *The Caesar cipher*. Khan Academy, 2012, https://www.khanacademy.org/math/applied-math/cryptography/crypt/v/caesar-cipher, besøgt d. 13/12-2013
- Gamborg, Niels. Designteori. http://nielsgamborg.dk/
- NSA. *Suite B Cryptograhy*. 2013, http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml, besøgt d. 15/12-2013
- Sauerberg, Jim. *Monoalphabetic Ciphers*. Saint Mary's College of California, 2011, http://ftp.stmarys-ca.edu/jsauerbe/m10s11/chapter5.pdf, besøgt d. 13/12-2013

Bilag

BILAG 1 – BEVIS FOR KRYPTERING MED RSA

Kryptering og dekryptering i RSA sker med følgende matematiske funktioner:

$$\textbf{Kryptering}: M^e \equiv C \bmod pq \qquad (\mathbf{1.1})$$
$$\textbf{Dekryptering}: C^{\boldsymbol{d}} \equiv M \bmod pq \qquad (\mathbf{1.2})$$

Hvor p og q er primtal, e den offentlige nøgle og d den private nøgle[41]. Det gælder at:

$$ed \equiv 1 \bmod (p-1)(q-1) \qquad (\mathbf{1.3})$$

Læg mærke til at når der regnes i en finit cyklisk mængde med modulus, betegner lig-med-tegnet med tre streger blot at det på venstre side er kongruent til det på højre side:

$a \equiv b \bmod n$ betyder at $(a-b)$ er delelig med n.

Hvis RSA kryptering og dekryptering skal fungere, altså at dekryptering af M^e giver klarteksten, må det gælde at kryptering og dekryptering skal være inverse operationer i den finitte cykliske gruppe F_{pq}:

$$(M^e)^d = M^{ed} \equiv M \bmod pq \qquad (\mathbf{1.4})$$

Jeg skal altså vise at den oprindelige besked M, opløftet til den offentlige nøgle multipliceret med den private nøgle, modulus produktet af de to primtal, giver den oprindelige besked. Jeg starter med at kigge på udtrykket $(\mathbf{1.3})$ der skal gælde:

$$ed \equiv 1 \bmod (p-1)(q-1) \qquad (\mathbf{1.5})$$

Modulus kan fjernes og skrives som en konstant k multipliceret med modulus m, adderet med resten r:

$$a \equiv r \bmod m \rightarrow a = r + k \cdot m \qquad (\mathbf{1.6})$$

grundet definitionen af restregning (modulær aritmetik)[42]. I dette tilfælde er $a = ed \ \wedge \ r = 1 \ \wedge \ m = (p-1)(q-1)$:

$$a = r + k \cdot m \rightarrow ed = 1 + k(p-1)(q-1) \ \textbf{hvor} \ k \in \mathbb{Z} \qquad (\mathbf{1.7})$$

Dette nye udtryk for ed, som er fremkommet på baggrund af definitionen af modulær aritmetik, kan jeg nu indsætte i $(\mathbf{1.4})$, og multiplicere med M på begge sider af lighedstegnet**:**

$$M^{ed} \equiv 1 \bmod pq \qquad (\mathbf{1.8})$$
$$M \cdot M^{ed} \equiv M \bmod pq$$
$$M \cdot M^{k(p-1)(q-1)} \equiv M \bmod pq$$

[41] Cohen, Henri og Frey, Gerhard: (2006) s.7
[42] Paar, Christof og Pelzl, Jan: (2010) s. 14

Da jeg ved at regnereglen $(a^b)^c \rightarrow a^{bc}$, kan jeg også regne den anden vej som set i $(\mathbf{1.9})$. Først undersøger jeg mod det ene primtal p:

$$M \cdot (M^{p-1})^{k(q-1)} \equiv M \bmod p \qquad (\mathbf{1.9})$$

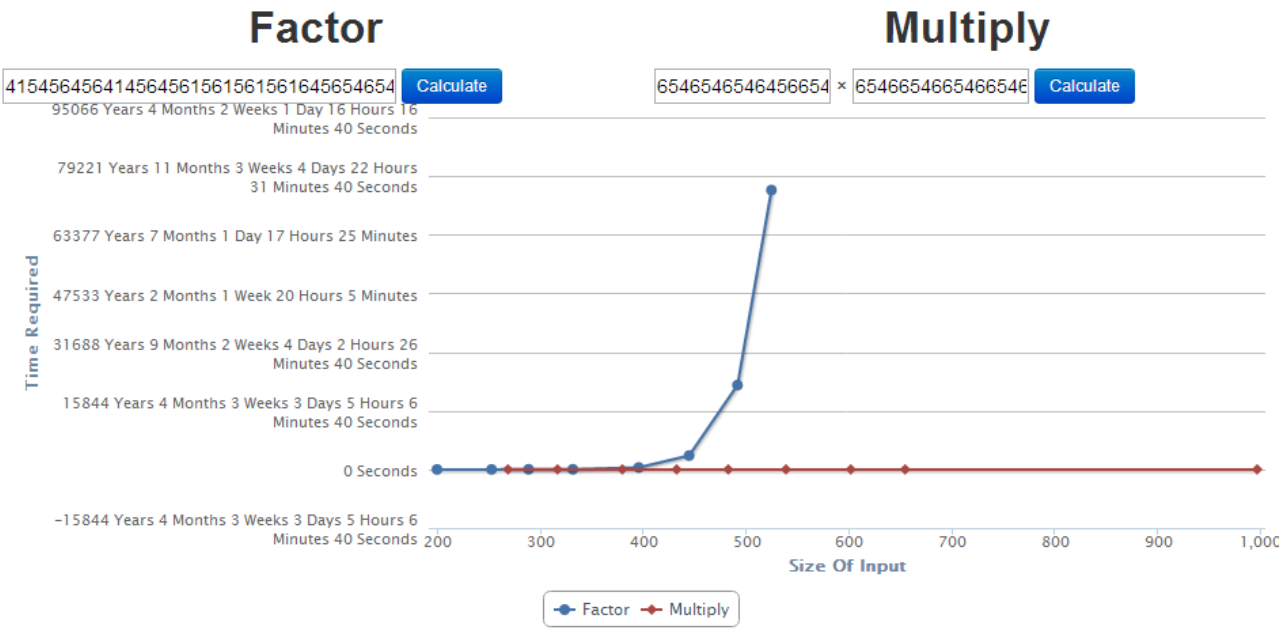
Fermats lille sætning siger at hvis p er et primtal, så vil det for ethvert helt tal a, som ikke er et multiplum (produktet af multiplikation) af p, gælde at: $a^{p-1} \equiv 1 \bmod p$, hvilket jeg kan indføre i $(\mathbf{1.9})$:

$$M \cdot (1)^{k(q-1)} \equiv M \bmod p \qquad (\mathbf{2.0})$$
$$M \cdot 1 \bmod p$$
$$M \bmod p$$

Og dermed må $M^{ed} \equiv M \bmod p$. Ved samme procedure findes også at $M^{ed} \equiv M \bmod q$. Via den kinesiske restklasse sætning[43], som siger at hvis $a \equiv b \bmod p \ \wedge a \equiv b \bmod q \rightarrow a \equiv b \bmod pq$, kan jeg finde ud af at dekryptering fungerer. Vi har at dekrypteringen af M^e giver klarteksten M i det finitte felt F_{pq}:

$$M^{ed} \equiv M \bmod pq \qquad (\mathbf{2.1})$$

Fra $(\mathbf{1.8})$ og $(\mathbf{1.4})$ kan det ses at Bob kan dekryptere en besked M, sendt fra Alice, ved at beregne: $c^d \equiv M^{ed} \equiv M^{1+k(p-1)(q-1)} \bmod pq$. Da kun produktet af p og q er kendt, skal Oscar (den onde der vil kompromittere sikkerheden) altså faktorisere $N = pq$ i to primtal, for at finde den private nøgle d. Dette er grundlaget for RSA's virke: primtalsfaktorisering, eller primtalsopløsning. Sikkerheden ligger i at det er meget svært (tidsmæssigt) at primtalsfaktorisere, hvis tallene er store nok[44]. Dette kan ses på nedenstående graf:



Figur 10 – Tidskompleksiteten bag faktorisering[45]

[43] Cohen, Henri og Frey, Gerhard: (2006) s.196
[44] Nichols, Randall K. og Lekkas, Panos C: (2002) s. 228
[45] Link til graf: https://www.khanacademy.org/labs/explorations/time-complexity

På grafen er x-aksen størrelsen af tallet I cifre, medens y-aksen er den tid det vil tage for en computer med nuværende teknologi, at udregne faktoriseringen af tallet (den blå kurve), og produktet af to store tal (den røde kurve). Det kan ses at hvis tallet bliver stort nok, jf. at den offentlige og private nøgle bliver stor nok (ca. 520 cifre lang), vil det tage næsten 79221 år at faktorisere nøglen. Som computere bliver kraftigere og kraftigere i henhold til Moores lov (at antallet af it-komponenter i en integreret kredsløb vil blive fordoblet hvert andet år), vil behovet for større bitlængder være nødvendig for at opnå den samme (tidsmæssige) sikkerhed.

BILAG 2 – BEVIS FOR KRYPTERING MED ELLIPTISKE KURVER

For at bevise at "Elliptic Curve Encryption Scheme" fungerer, skal jeg ligesom med RSA beviset, bevise at kryptering og dekryptering er inverse operationer. Kryptering og dekryptering foregår således (jf. Kryptering med Elliptiske Kurver):

$$(\mathbf{Bob})\ \mathbf{Offenligt\ punkt}\text{: } Q = dP \qquad (\mathbf{2.2})$$
$$(\mathbf{Alice})\ \mathbf{Kryptering}\text{: } C_1 = kP\ \wedge C_2 = M + kQ \qquad (\mathbf{2.3})$$
$$(\mathbf{Bob})\ \mathbf{Dekryptering}\text{: } M = C_2 - dC_1 \qquad (\mathbf{2.4})$$

hvor M er beskeden der skal sendes, k er et tilfældigt heltal inden for den elliptiske kurve mængde Z_p, og d er et (privat) heltal ligesom k, tilfældigt valgt indenfor Z_p. Det skal dog bemærkes at k og d ikke må være lig 1 eller længden af den cykliske mængde, da der derfor slet ikke bliver krypteret: $C_1 = 1P = P$. Som med alle andre asymmetriske kryptosystemer, er det Bob der generere en offentlig nøgle, hvilken Alice benytter til at krypterer sin besked til Bob. Bob kan herefter med sin private nøgle, dekrypterer beskeden fra Alice og se hvad der står i samme.

Jeg skal vise at Bobs beregning af beskeden $C_2 - dC_1$ rent faktisk giver M. Jeg starter med at se på hvad Bob udregner når kan udregner dC_1:

$$dC_1 = d(kP) = k(dP) = kQ \qquad (\mathbf{2.5})$$

Det kan ses at Bobs udregning giver den hemmelige konstant k som Alice benyttede til at kryptere, multipliceret med den offentlige nøgle. Læg mærke til at Bob grundet den associative lov, kan beregne kQ uden at kende k. Nu har Bob følgende:

$$M = C_2 - dC_1 = C_2 - kQ \qquad (\mathbf{2.6})$$

Herefter ser jeg på hvad punktet C_2, som er en del af chiffer teksten, består af:

$$C_2 = M + kQ \qquad (\mathbf{2.7})$$

Bob udregner altså følgende, hvor jeg substituerer $(\mathbf{2.5})$ og $(\mathbf{2.7})$ ind i $(\mathbf{2.4})$:

$$M = C_2 - dC_1 = M + kQ - kQ = M \qquad (\mathbf{2.8})$$
$$M = M$$

Det kan altså nu ses at Bob rent faktisk udregner den oprindelige besked sendt fra Alice.

Elliptisk kurve kryptografi bygger altså på "The elliptic curve discrete logarithm problem": Givet er en elliptiske kurve E defineret i den cykliske mængde $\mathrm{F_p}$ med kardinaliteten n og et punkt Q som ligger på kurven. Problemet ligger nu i at finde $l \in [0, n-1]$ således at:

$$Q = lP$$

Hvor heltallet l kaldes den diskrete logaritme til Q to basen P: $l = \log_p(Q)$ [46].

[46] Hankerson, Darrel og Menezes, Alfred og Vanstone, Scott: (2004) s. 153

BILAG 3 – IMPLEMENTERINGSALGORITMER

Eksponentiering ved kvadrering

Under implementeringen af RSA og ECC, løb jeg hurtigt ind i problemer. Det første problem var at kunne beregne en rod a opløftet til en eksponent med over 100 cifre, hvilket eksempelvis for roden $a = 2$, resulterer i et meget stort tal med 30 cifre:

$$2^{100} \approx 1.26 \cdot 10^{30}$$

Det forholder sig imidlertid sådan, at RSA og ECC i praksis arbejder med tal der er meget større, og en effektiv algoritme til at beregne sådanne potenser er derfor vigtig. For at gøre det ekstra besværligt, skal denne udregning ske i en finit cyklisk mængde eksempelvis F_{23}, hvorfor den endelige udregning bliver:

$$2^{100} \equiv 2 \bmod 23$$

Metoden til at regne dette hurtigt hedder "eksponentiering ved kvadrering" og udføres således. Først omskrives eksponenten til et binært tal, altså i base-2 systemet:

$$b = 100 = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$$
$$= 1100100_b$$

Herefter tælles antallet m, af cifre i b, og m a'er skriver op:

$$m = 7$$
$$\underbrace{a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a}_{m\ gange}$$

Hvert a opløftes nu til det tilsvarende binære ciffer i b:

$$a^1 \cdot a^1 \cdot a^0 \cdot a^0 \cdot a^1 \cdot a^0 \cdot a^0$$

Der sættes nu m venstre parenteser foran det første a, og én højre parentes efter alle a'er, og der kvadreres ved hver a undtagen det sidste:

$$(((((((a^1)^2 \cdot a^1)^2 \cdot a^0)^2 \cdot a^0)^2 \cdot a^1)^2 \cdot a^0)^2 \cdot a^0)$$

Da det gælder at $(a \cdot b) \bmod n = \big((a \bmod n) \cdot (b \bmod n) \big) \bmod n$ kan jeg nu indskrive restregningen ved hvert a:

$$(((((((a^1 \bmod n)^2 \cdot a^1 \bmod n)^2 \cdot a^0 \bmod n)^2 \cdot a^0 \bmod n)^2 \cdot a^1 \bmod n)^2 \cdot a^0 \bmod n)^2 \cdot a^0 \bmod n) \bmod n$$

Dette er smart, fordi man dermed sørger for at holde tallene man regner med nede på et fornuftigt niveau. I dette tilfælde er $a = 2 \ \wedge n = 23$, og resultatet bliver derfor (taget en udregning ad gangen), og hvor jeg for overskueligheden skyld har udeladt "mod" fra parenteserne og blot beholdt modulus ude til højre:

$$(((((((2)^2 \cdot 2)^2 \cdot 1)^2 \cdot 1)^2 \cdot 2)^2 \cdot 1)^2 \cdot 1) \bmod 23$$
$$((((((8)^2 \cdot 1)^2 \cdot 1)^2 \cdot 2)^2 \cdot 1)^2 \cdot 1) \bmod 23$$

$$(((((18)^2 \cdot 1)^2 \cdot 2)^2 \cdot 1)^2 \cdot 1) \bmod 23$$
$$((((2 \cdot 1)^2 \cdot 2)^2 \cdot 1)^2 \cdot 1) \bmod 23$$
$$(((4 \cdot 2)^2 \cdot 1)^2 \cdot 1) \bmod 23$$
$$((18 \cdot 1)^2 \cdot 1) \bmod 23$$
$$(2 \cdot 1) \bmod 23$$
$$2 \bmod 23$$

Som det kan ses fra mellemregninger bliver tallene aldrig større end modulus, og det kræver blot syv trin. Sammenlignet med den almindelige måde at gøre det på ved først at udregne 2^{100} og derefter dividerer med modulus for at finde resten, er denne algoritme meget effektiv[47]. Dette har jeg implementeret i C# kode og funktionen kan ses nedenfor:

```
public static BigInteger powModN(this BigInteger a, BigInteger b, BigInteger n)
{
    BigInteger x = BigInteger.Parse("1"),y=a; // Initialiser x og y
    while(b > 0){ // Kør alle bits i eksponenten igennem
        if (b % 2 == 1) // Hvis bit'en er sat
            x = (x * y) % n; // så ganger vi resultatet med y mod n
        y = (y * y) % n; // Kvadrer altid resultatet mod n
        b /= 2; // Divider med 2 er det samme som at shifte bit'sne til højre
    }
    return x % n; // Retunér resultatet x mod n
}
```

Man bruger funktionen ved at kalde således:

```
BigInteger.Parse("2").powModN(BigInteger.Parse("100"),BigInteger.Parse("23")) -> 2
```

Euklids udvidede algoritme

Fra eksemplet i "Kryptering med Elliptiske Kurver", fås følgende udregning der blot blev udført:

$$-2 \cdot (4)^{-1} \equiv -2 \cdot 6 \equiv 11 \bmod 23$$

Det kan ses at $4^{-1} \bmod 23$ gav 6. Seks er altså den modulære inverse til 4, og med Euklids udvidede algoritme kan man nemt beregne den modulært inverse af selv store tal[48]. Herunder viser jeg et papir og blyant eksempel:

Først ser jeg at den inverse kan repræsenteres som $4^{-1} = \frac{1}{4}$, men da man ikke i modulær aritmetik kan have brøker, må jeg finde en ligning der beskrive dette forhold:

$$4 \cdot x \equiv 1 \bmod 23$$

Hvor x er lig med den inverse til 4. Dette kan hurtigt eftervises er sandt, ved at indsætte den inverse til 4 på x plads:

[47] Riber: (2008) s. 33-34
[48] Gadegaard, Henrik G. og Hansen, Johan P: (2002) s. 17

$$4 \cdot \frac{1}{4} \equiv 1 \bmod 23$$

Via Euklids algoritme som siger at vi kan repræsenterer modulus n som en konstant q multipliceret med en anden konstant m adderet med en rest r, kan jeg opstille følgende sammenhæng:

$$n = qm + r$$

Hvor q er lig roden af eksponenten af den inverse: 4^{-1}, og $n = 23$:

$$23 = 4 \cdot m + r$$

Nu skal man så finde to tal m og r der tilfredsstiller ligningen, man kan se at 4 går $m = 5$ gange op i 23, hvorved resten bliver $r = 23 - 20 = 3$

$$23 = 4 \cdot (5) + 3$$

Herefter siger algoritmen at $n = q \ \wedge q = r$, hvilket jeg udfører indtil resten bliver 1:

$$4 = 3 \cdot (1) + 1$$

Resten bliver 1 med det samme, og ligningerne omformes nu, så resten r står for sig selv:

$$4 - 3 \cdot (1) = 1$$
$$23 - 4 \cdot (5) = 3$$

Derefter tager jeg udtrykket for tallet 3 og substituerer det i den første ligning og reducerer, hvor jeg hele tiden skriver resultatet multipliceret med et andet heltal (i parentes):

$$4 - (23 - 4 \cdot (5)) \cdot 1 = 1$$
$$4 - (23 \cdot (1) - 4 \cdot (5)) = 1$$
$$4 + 23 \cdot (-1) + 4 \cdot (5) = 1$$
$$23 \cdot (-1) + 4 \cdot (6) = 1$$

Jeg skal nu huske at jeg arbejder i modulus 23, hvorfor leddet $23 \cdot (-1) \equiv 0 \bmod 23$ (forsvinder):

$$4 \cdot (6) \equiv 1 \bmod 23$$

Jeg har nu et udtryk der ligner det jeg startede ud med: $4 \cdot x = 1 \bmod 23$, men nu blot en værdi for x, altså har jeg fundet den modulære inverse til $4 \bmod 23$ til 6.

Dette har jeg også kodet i C# og kan ses nedenfor:

```
public static BigInteger ModInverse(this BigInteger a, BigInteger mod){
            BigInteger b = mod; BigInteger dividend = a % b; BigInteger divisor = b;
            BigInteger last_x = BigInteger.One; BigInteger curr_x = BigInteger.Zero;
            while (divisor.Sign > 0){
                BigInteger quotient = dividend / divisor;
                BigInteger remainder = dividend % divisor;
```

```
            if (remainder.Sign <= 0)
                break;
            BigInteger next_x = last_x - curr_x * quotient;
            last_x = curr_x; curr_x = next_x;
            dividend = divisor; divisor = remainder;
        }
        return (curr_x.Sign < 0 ? curr_x + b : curr_x);
    }
```

Funktionen benyttes således:

```
ModInverse(BigInteger.Parse("4"), BigInteger.Parse("23")); -> 6
```

Addition af to punkter på en Eliptisk Kurve

Under afsnittet "Matematikken bag Elliptiske Kurver" viste jeg hvordan man lagde to punkter sammen, for de to scenarier: $P = Q \ \wedge P \neq Q$. Dette har jeg programmeret således i programmeringssproget C#:

```
public Point AddPoints(Point p1, Point p2){
    if (p1.X == 0 && p1.Y == 0 || p2.X == 0 && p2.Y == 0){// En af punkterne er uendelig
        return p1.X == 0 && p1.Y == 0 ? p2 : p1;
    }else{
        if ((p1.X == p2.X) && (p1.Y != p2.Y)) { return new Point(0, 0); } //Punktet uendelig
        if (p1.X == p2.X && p1.Y == p2.Y){ // Punkt fordobling
            if (p1.Y == 0 || p2.Y == 0) // Hvis y er 0, 2P = uendelig
                return new Point(0, 0);
            Point R = GetR(Slope(p1), p1, p2);
            return R.IsOnCurve() ? R : GetR(Slope(p2), p1, p2);
        }else{// Punktaddition
            Point R = GetR(Slope(p1, p2), p1, p2);
            return R.IsOnCurve() ? R : GetR(Slope(p2, p1), p1, p2);
        }
    }
}
```

Det kan ses at funktionen tager to punkter som parametre. Hvis én af punkterne befinder sig i (0,0) må det være punktet uendelig (altså det neutrale element på kurven). Derfor returnerer jeg det punkt hvis x- og y koordinater er forskellig fra nul.

Hvis x-værdierne er ens for de to punkter, samtidigt med at deres y-værdier er forskellige, må de ligge lodret over hinanden. Og da den elliptiske kurve er symmetrisk omkring x-aksen må de være hinandens inverse elementer. At lægge to punkter sammen der ligger lige over hinanden giver en hældning af linjen på uendelig, og resultatet bliver derfor det neutrale element (identiteten).

Hvis hverken punkterne ikke gav det neutrale element, men de to punkter er ens, udregner jeg punkt fordoblingen, ellers må det være en punkt addition.

BILAG 4 – IMPLEMENTERING AF ECC & RSA I C#

Min kode består af to klasser "RSA.cs" og "EllipticCurve.cs", og derudover en Windows Form (Grafisk brugerflade) "Form1.cs". Form1 klassen er den der sætter de to klasser RSA og EllipticCurve i spil på en grafisk tilgængelig brugerflade, mens klasserne er selve implementeringen af kryptosystemet. Programmet kan downloades her: http://www.magnusbm.dk/Projekter/Skole/SRP/ECCogRSA.zip
Koden i disse filer kan findes herunder:

RSA klassen

```
using System;
using System.Numerics;

namespace SRP___ECCnRSA{
public class RSA{
public class RSAKeys{
    public BigInteger n;
    public BigInteger e;
    public BigInteger d;
    public RSAKeys(BigInteger n, BigInteger e, BigInteger d)
            { this.n = n; this.e = e; this.d = d; }
}
private class PrimeLimits{
    public BigInteger Lower; public BigInteger Upper;
    public PrimeLimits(BigInteger l, BigInteger u) { Lower = l; Upper = u; }
}
public int BitLength{
    get { return this.bitLength; }
    set{
        if (value >= 16 && value <= 3072){
            this.primeLimits = new PrimeLimits(BigInteger.Pow(BigInteger.Parse("2"), value /
2 - 1), BigInteger.Subtract(BigInteger.Pow(BigInteger.Parse("2"), value / 2), BigInte-
ger.Parse("1")));
            this.bitLength = value;
        }
    }
}
public RSAKeys Keys = new RSAKeys(0, 0, 0);
private PrimeLimits primeLimits;
private int bitLength = 128;
private Random random = new Random();
public RSA(int bitLength) { this.BitLength = bitLength; }
public BigInteger findN(BigInteger p, BigInteger q) { return p * q; }
public BigInteger findK(BigInteger p, BigInteger q) { return (p - 1) * (q - 1); }
public BigInteger findD(BigInteger k) { return k.CoPrime(); }
public BigInteger findE(BigInteger d, BigInteger k) { return d.ModInverse(k); }
public void OpenKeys(RSAKeys keys) { this.Keys = keys; }
public void CreateKeys(){
    BigInteger p = RandomPrime(), q = RandomPrime(); this.Keys.n = p * q;
    BigInteger k = (p - 1) * (q - 1); this.Keys.d = k.CoPrime();
    this.Keys.e = Keys.d.ModInverse(k);
}
public string Encrypt(string mes){
    var cTxt = "";
    for (var i = 0; i < mes.Length; i++)
        cTxt += BigInteger.ModPow(BigInteger.Parse(((int)mes[i]).ToString()), this.Keys.e,
this.Keys.n).ToString() + " ";
    return cTxt;
}
```

```
public string Decrypt(string mes){
    var parts = mes.Split(' '); var mTxt = "";
    for (var i = 0; i < parts.Length - 1; i++)
        mTxt += (char)BigInteger.ModPow(BigInteger.Parse(parts[i]), this.Keys.d,
this.Keys.n);
    return mTxt;
}
public BigInteger RandomPrime() { return RandomProbPrime(this.primeLimits.Lower, this.prime-
Limits.Upper); }
private BigInteger RandomProbPrime(BigInteger low, BigInteger high){
    BigInteger bigGuess = low.Random(high);
    while (!bigGuess.IsProbablePrime(100)) { bigGuess = low.Random(high); }
    return bigGuess;
}}}
```

EllipticCurve klassen

```
using System;
using System.Collections.Generic;
namespace SRP___ECCnRSA{
public class EllipticCurve{
public class Point{

    public static EllipticCurve Curve = null; public int X = 0; public int Y = 0;
    public Point(int _x = 0, int _y = 0) { X = _x; Y = _y; }
    public bool IsOnCurve() { return Curve.PointOnCurve(this); }
    public static Point operator +(Point p1, Point p2) { return Curve.AddPoints(p1, p2); }
    public override string ToString() { return "[" + X + "," + Y + "]"; }
}
public int P = 23; public int A = 0; public int B = 0; public int K = 0; public int D = 0;
public Point Generator; public Point Q = null; private List<Point> field = null;
public List<Point> Field{
    get{
            field = new List<Point>();
            int[] x_val = new int[P]; int[] y_val = new int[P];
            for (int n = 0; n < P; ++n){
                int nsq = n * n;
                x_val[n] = ((n * nsq) + A * n + B) % P;
                y_val[n] = nsq % P;
            }
            for (int n = 0; n < P; ++n){
                for (int m = 0; m < P; ++m)
                    if (x_val[n] == y_val[m])
                        field.Add(new Point(n, m));
            }
        return field;
}}
public List<Point> UniqueDiffField{
    get{
        var uPoints = new List<Point>();
        foreach (Point p in Field){
            bool diffExists = false;
            foreach (Point up in uPoints)
                if ((p.Y - p.X) == (up.Y - up.X))
                    diffExists = true;
            if (!diffExists)
                uPoints.Add(p);
        }
        return uPoints;
}}
```

```
public EllipticCurve(int p, int _a, int _b, Point primitive) { P = p; A = _a; B = _b; Gener-
ator = primitive; }
public void CreateDomainParameters(){
    Random r = new Random();
    int k = r.Next(1, P - 1); int d = r.Next(1, P - 1);
    K = k; D = d; Q = EllipticCurve.Point.Curve.Mult(d, EllipticCurve.Point.Curve.Genera-
tor);
}
public Dictionary<string, int> EncryptDictionary(){
    Dictionary<string, int> dic = new Dictionary<string, int>(); char c = 'a';
    for (var i = 0; i < UniqueDiffField.Count; i++){
        Point p = UniqueDiffField[i];
        if (i == 0) c = ' '; else if (i == 1) c = 'a';
        try { dic.Add(c.ToString(), p.Y - p.X); } catch { continue; }
        c++;
    } return dic;
}
public Dictionary<int, string> DecryptDictionary(){
    Dictionary<int, string> dic = new Dictionary<int, string>(); char c = 'a';
    for (var i = 0; i < UniqueDiffField.Count; i++){
        Point p = UniqueDiffField[i];
        if (i == 0) c = ' '; else if (i == 1) c = 'a';
        try { dic.Add(p.Y - p.X, c.ToString()); } catch { continue; }
        c++;
    } return dic;
}

public string Encrypt(string m, int k, Point Q){
    Dictionary<string, int> eDic = EncryptDictionary(); string cpoints = "";
    EllipticCurve.Point C1 = EllipticCurve.Point.Curve.Mult(k, Ellip-
ticCurve.Point.Curve.Generator); cpoints += C1 + "|";
    for (var i = 0; i < m.Length; i++){
        EllipticCurve.Point M = EllipticCurve.Point.Curve.WhereDif-
fEquals(eDic[m[i].ToString()]);
        EllipticCurve.Point kQ = EllipticCurve.Point.Curve.Mult(k, Q); EllipticCurve.Point
C2 = M + kQ;
        cpoints += C2 + " ";
    } return cpoints;
}
public string Decrypt(int d, string cpoints){
    Dictionary<int, string> dDic = DecryptDictionary();
    string dTxt = ""; string tmp = cpoints.Split('|')[0]; cpoints = cpoints.Split('|')[1];
    string C1x = tmp.Trim('[', ']').Split(',')[0]; string C1y = tmp.Trim('[',
']').Split(',')[1];
    Point C1 = new Point(int.Parse(C1x), int.Parse(C1y));
    foreach (string cpoint in cpoints.TrimEnd().Split(' ')){
        string[] twopoints = cpoint.Trim().Split(',');
        string C2x = twopoints[0].Trim('[', ']');
        string C2y = twopoints[1].Trim('[', ']');
        Point C2 = new Point(int.Parse(C2x),int.Parse(C2y));
        EllipticCurve.Point dC1 = EllipticCurve.Point.Curve.Mult(d, C1);
        dC1.Y = Mod(-dC1.Y, EllipticCurve.Point.Curve.P); // Inverter punktet..
        EllipticCurve.Point DM = dC1 + C2; // Så dette bliver C2 - dC1
        dTxt += dDic[((DM.Y - DM.X))];
    } return dTxt;
}
public Point WhereDiffEquals(int diff){
    foreach (Point p in Field)
        if ((p.Y - p.X) == diff)
            return p;
```

```
    return new Point(0, 0);
}
public Point Mult(int k, Point gen){
    Point p = new Point(gen.X, gen.Y);
    for (var i = 0; i < k - 1; i++)
        p = gen + p;
    return p;
}
public Point AddPoints(Point p1, Point p2){
    if (p1.X == 0 && p1.Y == 0 || p2.X == 0 && p2.Y == 0){// En af punkterne er uendelig
        return p1.X == 0 && p1.Y == 0 ? p2 : p1;
    }else{
        if ((p1.X == p2.X) && (p1.Y != p2.Y)) { return new Point(0, 0); } //Punktet uendelig
        if (p1.X == p2.X && p1.Y == p2.Y){ // Punkt fordobling
            if (p1.Y == 0 || p2.Y == 0) // Hvis y er 0, 2P = uendelig
                return new Point(0, 0);
            Point R = GetR(Slope(p1), p1, p2);
            return R.IsOnCurve() ? R : GetR(Slope(p2), p1, p2);
        }else{// Punkt addition
            Point R = GetR(Slope(p1, p2), p1, p2);
            return R.IsOnCurve() ? R : GetR(Slope(p2, p1), p1, p2);
        }
    }
}
public bool PointOnCurve(Point p) { return Mod(p.Y * p.Y, P) == Mod((p.X * p.X * p.X) + A *
p.X + B, P); }
private int Slope(Point p1, Point p2) { return (p2.Y - p1.Y) * Inverse(Mod((p2.X - p1.X),
P), P); }
private int Slope(Point p) { return Mod((3 * p.X * p.X + A), P) * Inverse(Mod(2 * p.Y, P),
P); }
public static int Inverse(int n, int p){
    int tmp = n; int x = 1; int y = 0; int a = p; int b = n; int q, t;
    while (b != 0) { t = b; q = (int)Math.Floor((float)a / t); b = a - q * t; a = t; t = x;
x = y - q * t; y = t; }
    return y < 0 ? y+p : y;
}
private int Mod(int n, int mod){ // Egen implementering af modulus og ikke "remainder"
    int frac = (int)Math.Floor((float)n / (float)mod);
    return n - (frac) * mod;
}
private Point GetR(int slope, Point p1, Point p2)
{
    int xR = Mod((slope * slope) - p1.X - p2.X, P); int yR = Mod(((slope * (p1.X - xR)) -
p1.Y), P);
    if (xR < 0) { xR = xR + P; } if (yR < 0) { yR = yR + P; }
    return new Point(xR, yR);
}}}
```

Form1 klassen

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.Numerics;
namespace SRP___ECCnRSA{

    public partial class Form1 : Form{
        Random random = new Random(); RSA rsa; EllipticCurve curve;
        public Form1(){
```

```
            InitializeComponent();
            rsa = new RSA((int)bitLength.Value); // Initialiser RSA med valgt bitlængde
            EllipticCurve.Point.Curve = new EllipticCurve(23, 3, 2, new EllipticCurve.Po-
int(0, 5)); // Initialiser en ny elliptisk kurve
        }
        private int Mod(int n, int mod)
        { // Egen implementering af rigtig modulus og ikke "remainder"
            int frac = (int)Math.Floor((float)n / (float)mod);
            return n - (frac) * mod;
        }
        #region RSA - FindButtons
        private void findPrimesButton_Click(object sender, EventArgs e){
            rsa.BitLength = (int)bitLength.Value; // Ændre bitlængden til den valgte
            pBox.Text = rsa.RandomPrime().ToString(); // Find to store primtal,
            qBox.Text = rsa.RandomPrime().ToString(); // forholdsvist tæt på hinanden
        }
        private void findKButton_Click(object sender, EventArgs e){
            try{
                kBox.Text = rsa.findK(BigInteger.Parse(pBox.Text), BigInte-
ger.Parse(qBox.Text)).ToString(); // find konstanten k
                pBox.BackColor = Color.White; qBox.BackColor = Color.White; // Hvis det kan
lade sig gøre så sæt baggrunden til hvid
            }catch{// Hvis ikke den kan lade sig gøre, er p eller q ikke til stede eller
ugyldige
                if (pBox.Text == "")
                    pBox.BackColor = Color.LightSalmon; // Sæt derfor baggrunden til lakse-
farvet for fejl
                if (qBox.Text == "")
                    qBox.BackColor = Color.LightSalmon;
        }}
        private void findNButton_Click(object sender, EventArgs e){
            try{
                nBox.Text = rsa.findN(BigInteger.Parse(pBox.Text), BigInte-
ger.Parse(qBox.Text)).ToString(); // Multiplicér p og q
                pBox.BackColor = Color.White; qBox.BackColor = Color.White;
            }catch{
                if (pBox.Text == "")
                    pBox.BackColor = Color.LightSalmon;
                if (qBox.Text == "")
                    qBox.BackColor = Color.LightSalmon;
        }}
        private void findPrivateKeyButton_Click(object sender, EventArgs e){
            try{
                dBox.Text = rsa.findD(BigInteger.Parse(kBox.Text)).ToString(); // find den
private nøgle
                kBox.BackColor = Color.White;
            }catch{
                if (kBox.Text == "")
                    kBox.BackColor = Color.LightSalmon;
        }}
        private void findEButton_Click(object sender, EventArgs e){
            try{
                eBox.Text = rsa.findE(BigInteger.Parse(dBox.Text), BigInte-
ger.Parse(kBox.Text)).ToString(); // Find offentlig nøgle
                dBox.BackColor = Color.White;
                kBox.BackColor = Color.White;
            }catch{
                if (kBox.Text == "")
                    kBox.BackColor = Color.LightSalmon;
                if (dBox.Text == "")
```

```
                    dBox.BackColor = Color.LightSalmon;
        }}
        #endregion
        #region RSA - Key Generation
        private void automaticKeyGenButton_Click(object sender, EventArgs ee){
            rsa.BitLength = (int)bitLength.Value;
            BigInteger p = rsa.RandomPrime(), q = rsa.RandomPrime();
            BigInteger n = (p * q), k = (p - 1) * (q - 1), d = k.CoPrime(); BigInteger e =
d.ModInverse(k);
            pBox.Text = p.ToString(); qBox.Text = q.ToString(); kBox.Text = k.ToString();
nBox.Text = n.ToString();
            dBox.Text = d.ToString(); eBox.Text = e.ToString();
        }
        #endregion
        #region RSA - Encrypt/Decrypt
        private void cryptButton_Click(object sender, EventArgs e){
            if (plainTextBox.Text != ""){
                ValidateKeys(); // Prøv at parse alle konstanter/nøgler i RSA opsætningen
                if (nBox.Text != "" && eBox.Text != "" && dBox.Text != ""){ // Hvis valide
så kryptér
                    chifferTextBox.Text = rsa.Encrypt(plainTextBox.Text);
                    plainTextBox.Text = ""; plainTextBox.BackColor = Color.White;
            }}else
                plainTextBox.BackColor = Color.LightSalmon; // Ellers indikér ugyldighed med
laksefarvet baggrund
        }
        private void ValidateKeys(){
            try{
                rsa.Keys.n = BigInteger.Parse(nBox.Text);
                nBox.BackColor = Color.White;
            }catch { nBox.BackColor = Color.LightSalmon; }
            try{
                rsa.Keys.d = BigInteger.Parse(dBox.Text);
                dBox.BackColor = Color.White;
            }catch { dBox.BackColor = Color.LightSalmon; }
            try{
                rsa.Keys.e = BigInteger.Parse(eBox.Text);
                eBox.BackColor = Color.White;
            }catch { eBox.BackColor = Color.LightSalmon; }
        }
        private void decryptButton_Click(object sender, EventArgs e){
            if (chifferTextBox.Text != ""){
                ValidateKeys();
                if (nBox.Text != "" && eBox.Text != "" && dBox.Text != ""){
                    plainTextBox.Text = rsa.Decrypt(chifferTextBox.Text); // Dekrypter med
privat nøgle d
                    chifferTextBox.Text = ""; chifferTextBox.BackColor = Color.White;
            }}else
                chifferTextBox.BackColor = Color.LightSalmon;
        }
        #endregion
        #region ECC - Plotting
        private void plotCurveButton_Click(object sender, EventArgs e){
            EllipticCurve.Point.Curve.A = int.Parse(aBox.Text); EllipticCurve.Point.Curve.B
= int.Parse(bBox.Text);
            EllipticCurve.Point.Curve.P = int.Parse(modBox.Text);
            plot1.XSteps = EllipticCurve.Point.Curve.P - 1; plot1.YSteps = Ellip-
ticCurve.Point.Curve.P - 1;
            EllipticCurve.Point pe = EllipticCurve.Point.Curve.Field[0]; primitiveLabel.Text
= "Primitivt element: " + pe;
```

```
            plot1.PlotPoints.Clear();
            foreach (EllipticCurve.Point p in EllipticCurve.Point.Curve.Field)
                plot1.PlotPoint(new Plot.Point(p.X,p.Y));
            string availLetters = "";
            foreach (KeyValuePair<string, int> d in EllipticCurve.Point.Curve.EncryptDic-
tionary())
                availLetters += d.Key+" ";
            availLettersLabel.Text = "Tilgængelige bogstaver: "+availLetters;
        }
        private void addPointsButton_Click(object sender, EventArgs e){
            EllipticCurve.Point R;
            if (plot1.SelectedPointP != null && plot1.SelectedPointQ != null)
                R = EllipticCurve.Point.Curve.AddPoints(new EllipticCurve.Point(plot1.Se-
lectedPointP.X, plot1.SelectedPointP.Y), new EllipticCurve.Point(plot1.SelectedPointQ.X,
plot1.SelectedPointQ.Y));
            else
                R = EllipticCurve.Point.Curve.AddPoints(new EllipticCurve.Point(plot1.Se-
lectedPointP.X, plot1.SelectedPointP.Y), new EllipticCurve.Point(plot1.SelectedPointP.X,
plot1.SelectedPointP.Y));
            rLabel.Text = "R = " + R.ToString(); plot1.SelectPointR(R.X, R.Y);
        }
        private void clearButton_Click(object sender, EventArgs e) { plot1.ClearSelec-
tions(); }
        private void pEqRButton_Click(object sender, EventArgs e) { plot1.Se-
lectPointP(plot1.SelectedPointR.X, plot1.SelectedPointR.Y); plot1.SelectedPointR = null; }
        #endregion
        #region ECC - Encrypt/Decrypt
        public bool validPlainText(){
            string[] letters = availLettersLabel.Text.Trim().Split(':')[1].Trim().Split('
'); bool valid = false;
            foreach (char c in plainTxtEcc.Text){
                valid = false;
                foreach (string l in letters)
                    if (l == c.ToString() || c.ToString() == " ")
                        valid = true;
                if (!valid) break;
            }
            if (!valid) { plainTxtEcc.BackColor = Color.LightSalmon; } else {
plainTxtEcc.BackColor = Color.White; } return valid;
        }
        private void plainTxtEcc_TextChanged(object sender, EventArgs e){validPlainText();}
        private void plainTxtEcc_KeyDown(object sender, KeyEventArgs e){validPlainText();}
        private void cryptEccButton_Click(object sender, EventArgs e){
            if (validPlainText()){
                EllipticCurve.Point.Curve.CreateDomainParameters();

                knDLabel.Text = "K: " + EllipticCurve.Point.Curve.K + " D: " + Ellip-
ticCurve.Point.Curve.D;
                chifferTxtEcc.Text = EllipticCurve.Point.Curve.Encrypt(plainTxtEcc.Text, El-
lipticCurve.Point.Curve.K, EllipticCurve.Point.Curve.Q);
                plainTxtEcc.Text = "";
        }}
        private void decryptECCButton_Click(object sender, EventArgs e){
            try { plainTxtEcc.Text = EllipticCurve.Point.Curve.Decrypt(Ellip-
ticCurve.Point.Curve.D, chifferTxtEcc.Text); chifferTxtEcc.Text = ""; }
            catch { }
        }
        #endregion
    }
}
```