Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Digitale Handtekeningen: een experimentele vergelijking

D.E. Wilschut

Modelling, Analysis and Simulation (MAS)

MAS-N0001 August 31, 2000

Report MAS-N0001
ISSN 1386-3703

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Digitale Handtekeningen: een Experimentele Vergelijking

D.E. Wilschut
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
(1 februari – 30 juni 2000)

ABSTRACT

In dit onderzoek zijn twee digitale handtekeningen-algoritmen experimenteel vergeleken: Digital Signature algorithm (DSA), gebaseerd op het discrete logaritmeprobleem in $\mathbb{F}_p^*$ en Elliptic Curve DSA, de elliptische krommenvariant van DSA, gebaseerd op de discrete logaritme in $E(\mathbb{F}_p)$. Gekeken is welke bij gelijke veiligheid, dus gelijke kans te worden gekraakt, het snelste is.

Om de snelheid van beide systemen bij gelijke veiligheid te kunnen vergelijken is het nodig te bepalen wanneer beide systemen even veilig zijn. Dit is een open probleem in de cryptografie. De algemene methode is een verhouding te geven voor de parameter-bitlengtes van beide systemen, die uitdrukt hoeveel veiliger het ene systeem is dan het andere. Hoe die verhouding moet worden bepaald is nog niet uitgekristalliseerd. In dit onderzoek wordt gebruik gemaakt van een artikel van Arjen Lenstra en Eric Verheul, waarin de auteurs een eerste schematische behandeling van dit onderwerp nastreven voor de vier grote cryptosystemen, waaronder de hier gebruikte discrete logaritme in F_p en elliptische krommen.

Het belangrijkste resultaat is dat elliptische krommen voor de nu gebruikte bitlengtes sneller zijn, maar voor toenemende veiligheid verliezen van DSA. Er lijkt altijd een omslagpunt te zijn: ECDSA reageert slechter op een toename van de bitlengte dan DSA.

2000 Mathematics Subject Classification: Primary 11Y99. Secondary 94A60.
1998 ACM Computing Classification System: E.3.
Keywords and Phrases: digital signature algorithm (DSA), discrete logarithm, elliptic curve DSA, MD4 message digest algorithm.
Note: Verslag ter afronding van afstudeerstage studie Bedrijfskunde aan de Hogeschool Holland te Diemen (voor nadere gegevens, zie p. 53). De afstudeerstage werd op het CWI uitgevoerd binnen project MAS2.2 "Computational number theory and data security".

Table of Contents

Hoofdstuk 1
Inleiding

Pleitbezorgers van de Nieuwe Economie schetsen een paradijs op aarde als we ook via internet onze handtekening kunnen zetten. Nu wordt voor belangrijke transacties nog altijd een papieren overeenkomst getekend, terwijl partners elkaar in deze globale markt wellicht nooit hebben ontmoet. Deze papieren contracten staan de ware revolutie van telecommunicatie in de weg, menen de voorstanders van legalisering.

In dit onderzoek worden twee digitale signeeralgoritmen experimenteel vergeleken. Preciezer gezegd is het een vergelijking van twee wiskundige functies die aan de basis staan van dergelijke algoritmen: de *subgroep discrete logaritme* en *elliptische kromme discrete logaritme*.

Waarom deze twee?
Al sinds de ontdekking in 1985 door Miller en Koblitz wordt er veel verwacht van de cryptografische mogelijkheden van elliptische krommen. Sceptici zeggen dat deze krommen slechts veelbelovend zijn, omdat er nog onvoldoende onderzoek naar is verricht. Echter, naarmate de tijd verstrijkt lijken zij ongelijk te krijgen: de elliptische kromme, een algebraïsche structuur die een eindige, cyclische groep voortbrengt, houdt zich goed staande tussen het intensieve, geavanceerde onderzoek naar efficiënte kraakmethoden. Tot dusver zijn slechts op kleine, zeer specifieke groepen krommen succesvolle aanvallen gedaan, de meerderheid lijkt ongevoelig.

De functie die het mag opnemen tegen deze nieuwkomer is de subgroep discrete logaritme in de groep $\mathbb{F}_p$, zeker niet de eerste de beste. Als variant op de eerste public-key cryptosystemen kan de subgroep discrete logaritme bogen op een lange onderzoeksgeschiedenis en is ze door de wol geverfd. Hoewel er in de loop der jaren vele algoritmen zijn ontwikkeld om de discrete logaritme in $\mathbb{F}_p$ op te lossen, zijn zelfs de beste niet heel erg goed. Zij geldt dan ook als het beste wiskundig fundament voor de digitale handtekening, uitgezonderd – wellicht – de elliptische kromme.

In dit onderzoek wordt middels een experiment geprobeerd te bepalen welke de snelste is. Hierbij ligt de nadruk op het epxerimentele verschil, dus het verschil in benodigde rekentijd, en niet op de theoretische beloften van beide systemen.

Voor het experiment is gekozen twee algoritmen te gebruiken die veel op elkaar lijken: het *Digital Signature Algorithm* (DSA), voor de subgroep discrete logaritme, en *Elliptic Curve Digital Signature*

Algorithm (ECDSA), voor de elliptische krommen. Zoals de naam al doet vermoeden is ECDSA de elliptische kromme-variant van DSA.

In het eerste hoofdstuk wordt een beknopte beschrijving gegeven van het vakgebied cryptografie. Digitale signeermethoden maken deel uit van het cryptografisch onderzoek; de gebruikte algoritmen zijn nauw verwant aan public-key cryptosytemen. Daarna wordt, in hoofdstuk twee, ingegaan op het wiskundige probleem dat ten grondslag ligt aan de hier gesimuleerde digitale handtekeningen-systemen, het discrete logaritme-probleem. In hoofdstuk drie volgt een theoretische verkenning van de digitale handtekening.

Daarna volgt, in hoofdstuk vier, een beschrijving en bespreking van het onderzochte algoritme, Digital Signature Algorithm (DSA). In hoofdstuk vijf wordt de eerste implementatie van DSA, in $\mathbb{F}_p$, langsgelopen. Hoofdstuk zes is een theoretische inleiding over elliptische krommen. In hoofdstuk zeven wordt eerst ECDSA, het elliptische kromme analogon van het DSA-algoritme besproken. Daarna volgt een bespreking van het tweede simulatieprogramma, ECDSA. Hoofdstuk acht beschrijft het experiment en de resultaten. In het laatste hoofdstuk, hoofdstuk negen, volgen tot slot de conclusies en worden aanbevelingen gedaan voor verder onderzoek.

Hoofdstuk 2
Cryptografie

Tot halverwege de twintigste eeuw was cryptografie vooral een zaak van regeringen, het leger en criminelen. Met de komst van de computer in de jaren zestig en zeventig ontstond er ook in de commerciële en particuliere sector vraag naar beveiligingstechnieken voor deze nieuwe digitale communicatie. Tegenwoordig, in de tijd van internet, e- en m-commerce, heeft iedereen privacy-behoeften en er wordt dan ook veel onderzoek gedaan naar goedkope, snelle en eenvoudige cryptografische methoden.

In dit hoofdstuk zal kort een beeld worden geschetst van cryptografie. Eerst worden de drie cryptografische hoofdpersonen geïntroduceerd. In de tweede paragraaf wordt een algemene beschrijving van cryptografie gegeven. Daarna volgt een uitleg over public-key en private-key cryptografie[1]. Tot slot wordt één van de hoofdingrediënten van een public-key cryptosysteem besproken, de one-way functie.

1. Hoofdpersonen

In de cryptografische theorie worden sinds jaar en dag drie personen ten tonele gevoerd: Alice, Bob en Eve, die gedrieën uitwisseling van geheime informatie naspelen.

Alice Alice wil iets versturen aan Bob, bijvoorbeeld een geheime of gesigneerde boodschap.

Bob Bob ontvangt berichten van Alice. Hij wil controleren of het bericht echt van Alice komt, hij wil het bericht kunnen lezen en hij wil kunnen zien of de boodschap onderweg is veranderd.

Eve Eve is de spion. Zij wil de informatie die tussen Bob en Alice wordt uitgewisseld kunnen lezen en manipuleren, bijvoorbeeld om Bob berichten te sturen die ze met 'Alice' ondertekent.

[1] Voor verschillende begrippen is gekozen de Engelse benaming te handhaven, hetzij omdat de Engelse naam is ingeburgerd, hetzij omdat er geen adequate vertaling voorhanden is.

Alice en Bob zouden natuurlijk in een donkere achterkamer kunnen afspreken en op gedempte toon hun geheimen kunnen uitwisselen. Ze willen echter niet verplicht zijn samen te komen en tijdens zo'n bijeenkomst bovendien alert te zijn op spionnen. Ze willen in alle openheid hun gesprek voeren, communiceren via een *onveilig*[2] kanaal.

Een onveilig kanaal wil zeggen dat Eve alles kan horen wat Alice en Bob zeggen en dat zij bovendien allerlei informatie aan het gesprek toe kan voegen. Beide vormen van infiltratie moeten Alice en Bob voorkomen. Het maakt overigens niet uit dat Eve alles kan horen, als ze het maar niet kan verstaan; voor een geheim gesprek hoeven twee Chinezen in een gemiddelde amsterdamse metro niet te fluisteren.

2. Cryptografie

Cryptografie is de wetenschap die zich bezig houdt met de beveiliging van communicatie. De beveiliging bestaat uit het omzetten van geheime informatie naar een gecodeerde boodschap alvorens die te versturen. De codering van informatie gebeurt met een *sleutel*. Om de boodschap te kunnen ontcijferen moet Bob natuurlijk informatie hebben over de door Alice gebruikte codeertechniek. Er wordt daarom vaak gesproken over een *sleutelpaar*, de codeersleutel van Alice en de bijbehorende ontcijfersleutel van Bob.

Geleid door de mogelijkheden die public-key cryptografie (zie paragraaf 3) biedt, wordt binnen het huidige onderzoek gekeken naar de volgende vier eigenschappen, of doelen, van cryptografische methoden:

geheimhouding Als Alice Bob een boodschap stuurt garandeert de eigenschap *geheimhouding* dat niemand behalve Bob de informatie kan lezen.

data-integriteit Als Bob een boodschap van Alice krijgt garandeert *data-integriteit* dat Eve geen informatie aan het bericht heeft toegevoegd of uit het bericht heeft verwijderd.

authenticiteit Als Alice met Bob gaat praten garandeert *authenticiteit* dat Bob niet stiekem Eve is die zegt dat ze Bob is (entiteit-authenticiteit). Ook garandeert *authenticiteit* dat Bob kan zien dat een boodschap die 'zegt' van Alice te komen ook echt van Alice komt (afzender-authenticiteit).

erkenning Als Alice een boodschap heeft verstuurd aan Bob en dit later ontkent garandeert *erkenning* dat Bob aan derden kan aantonen dat Alice en niemand anders deze boodschap aan hem heeft verstuurd.

Voor digitale handtekeningen-algoritmen zijn *data-integriteit*, *authenticiteit* en *erkenning* van belang. In hoofdstuk 4 worden deze doelen uitgebreider behandeld.

3. Public-key versus private-key

We stand today on the brink of a revolution in cryptography.

Zo openen Whitfield Diffie en Martin E. Hellman in 1976 hun beroemde artikel *New Directions in Cryptography* [5] waarin zij public-key cryptografie voorstellen. Met recht een revolutie.

De cryptografie biedt in essentie twee manieren om de veiligheid van communicatiekanalen te garanderen: *private-key* en *public-key* cryptografie. Het verschil tussen beide wordt precies door hun namen verteld: public-key systemen maken gebruik van een openbare sleutel terwijl private-key systemen vereisen dat de betrokken partijen de sleutel geheim houden. Ook de op het eerste gezicht wellicht ondoorzichtige namen 'symmetrisch' voor geheime, en 'asymmetrisch' voor openbare sleutels, worden in de literatuur regelmatig gebruikt. Deze symmetrie is de symmetrie van de sleutels: private-key gebruikt voor coderen en ontcijferen dezelfde sleutel, terwijl public-key voor beide operaties verschillende sleutels heeft.

[2] 'onveilig' is een vertaling van het engelse *insecure*.

In de praktijk wordt vaak een combinatie van beide gebruikt, omdat public-key systemen veel langzamer zijn. Via een onveilig kanaal wisselen Alice en Bob hun sleutel uit, met behulp van public-key cryptografie, een zogenaamd *public-key sleutel-distributiesysteem*. Deze sleutel gebruiken ze daarna in een symmetrisch cryptosysteem.

4. One-way functies

Een van de belangrijkste pijlers van een public-key cryptosysteem is de one-way functie. One-way functies zijn functies die de ene kant op (relatief) eenvoudig, de andere kant op zeer moeilijk te berekenen zijn. Meer wiskundig gezegd: voor een one-way functie f is het bij gegeven x eenvoudig $f(x)$ te bepalen, maar zeer moeilijk voor gegeven $f(x)$ het orgineel x, ofwel $f^{-1}(f(x))$, te berekenen.

Een goed analogie van een one-way functie is het breken van een glas. Een glas in meer dan honderd stukjes kapot gooien is eenvoudig; van die stukjes weer een glas lijmen is een stuk moeilijker, zo niet onmogelijk.

Intuïtief is dit een duidelijke afbakening. Strikt wiskundig ligt het echter een stuk ingewikkelder. Het is nooit bewezen dat dergelijke functies bestaan, maar ook niet dat ze niet bestaan. Er is wel een aantal functies dat aan deze eis *lijkt* te voldoen. Het zijn deze kandidaatfuncties die aan de basis liggen van public-key cryptosystemen.

Wat is er cryptografisch dan interessant aan zulke functies?
Deze functies zijn op zich voor de cryptografie nog niet bruikbaar. Als je een boodschap met f codeert kan niemand hem ooit ontcijferen! Echter, er is een bijzondere klasse one-way functies, zogenaamde *trapdoor one-way functies*, die een sluiproute, *trapdoor*, in zich bergen[3]. Als je wat extra informatie hebt is het ineens heel eenvoudig om vanuit $f(x)$ x te bepalen.

Een goede vergelijking is het schudden van een stapel speelkaarten. Het is makkelijk de speelkaarten goed te schudden, maar moeilijk ze weer in de oorspronkelijke volgorde terug te leggen... tenzij je die eerst ergens had genoteerd.

Wat zijn dat voor mysterieuze functies?
De bekendste trapdoor one-way functie is het vermenigvuldigen van twee grote getallen en de bijbehorende inverse functie factorisatie, waar het veel gebruikte cryptosysteem RSA op is gebaseerd. RSA maakt gebruik van de moeilijkheid de priemfactoren van een groot getal N, van bijvoorbeeld 200 cijfers, te vinden. Als deze factoren op hun beurt zelf redelijk grote priemen zijn, van bijvoorbeeld 100 cijfers, is het factoriseren van N met de huidige technieken een bijna onmogelijke opgave. RSA gebruikt de afbeelding $x \rightarrow x^e \bmod N$, met e en N openbaar en N het produkt van twee grote getallen. De inverse functie, $\leftarrow$, is eenvoudig als je de priemfactoren van N kent (zie voor een uitgebreide beschrijving van RSA bijvoorbeeld [21]).

De 'sluiproute' van RSA is aan mensen die toestemming hebben een boodschap te ontcijferen deze factoren te geven.

[3] Strikt genomen zijn dit geen one-way functies, maar vanwege hun verwantschap worden ze wel zo genoemd. De benaming *trapdoor* is overigens ook ongelukkig gekozen; 'sluiproute' dekt mijns inziens beter de lading.

Hoofdstuk 3
Discrete Logaritme-probleem

Modulair machtsverheffen – en zijn inverse de discrete logaritme – was de eerste wiskundige invulling van het theoretische begrip one-way functie. Diffie en Hellman presenteerden in [5] behalve public-key cryptografie ook een sleutel-distributiesysteem gebaseerd op de discrete logaritme (zie voor een beschrijving van dit protocol paragraaf 1). De kracht van het systeem berust op het discrete logaritme-probleem.

Eerst wordt een definitie gegeven van de one-way functie modulair machtsverheffen en het bijbehorende discrete logaritme-probleem (DLP). Vervolgens zullen enkele algoritmen voor het DLP worden besproken. De nadruk ligt hierbij op het onderliggende idee; er is niet geprobeerd een volledige beschrijving van ieder algoritme te geven.

1. Logaritme

De wiskundige functie *logaritme* bepaalt voor een x en een grondtal g tot welke macht m je g moet verheffen om x te krijgen:

$$\log_g x = m \Leftrightarrow g^m = x.$$

Een bekend voorbeeld is de natuurlijke logaritme, $\ln = \log_e$.

De *discrete* logaritme is nu de logaritmische functie in een eindige, cyclische groep G. De cryptografisch interessante groepen zijn de multiplicatieve groepen $\mathbb{F}_q^*$ van het eindige lichaam $\mathbb{F}_q$. In dit onderzoek wordt naar $\mathbb{F}_p$, met p priem, gekeken.

definitie discrete logaritmeprobleem

Het discrete logaritme-probleem (DLP) speelt zich af in een eindige, cyclische groep G van orde n met een voortbrenger α. Voor voortbrenger α en een groepselement $\beta \in G$ geldt de volgende equivalentie

$$\beta = \alpha^x \Leftrightarrow x = log_\alpha \beta.$$

Het discrete logaritme-probleem is nu: Gegeven β en α, vind het unieke natuurlijk getal x, $0 \leq x \leq n \Leftrightarrow 1$, zó dat $\alpha^x = \beta$.

2. Algoritmen voor de discrete logaritme

Sinds het beroemde artikel van Diffie en Hellman is er veel onderzoek gedaan naar mogelijke oplossingsmethoden voor DLP. De nu bekende algoritmen zijn grofweg in twee groepen in te delen (voor

een specifiekere indeling zie [19])

1. *square root*-methoden (ze danken hun naam aan hun complexiteit van $O(\sqrt{(n)})$), methoden die voor iedere cyclische groep werken en
2. algoritmen die gebruik maken van een speciale groeps- of subgroepeigenschap. (In de praktijk betekent dit dat er extra kennis van de (sub)groep voorhanden is.)

In deze paragraaf zullen vier methoden worden besproken: twee square root-methoden en vervolgens twee methoden die de onderliggende groepsstructuur gebruiken.

De meest eenvoudige methode is simpelweg alle machten van α te doorlopen totdat β is gevonden, de zogenaamde *exhaustive search*. Al vrij snel is deze methode zeer tijdrovend en voor grote x, of liever voor groepen met grote orde, zelfs praktisch onmogelijk. Voor cryptografisch grote getallen, van bijvoorbeeld 100 cijfers, kan deze methode niet worden gebruikt.

Baby-step giant-step algoritme (Shanks)
Dit algoritme lijkt enigszins op *exhaustive search*. Echter, in plaats van *alle* machten van α te berekenen wordt een geschikt gekozen rij groepselementen doorlopen. Het idee is

$$\beta = \alpha^x \Leftrightarrow \beta = \alpha^{im+j} = \alpha^{im}\alpha^j$$

waar $m = \lceil\sqrt{n}\rceil$ en met $0 \leq i, j < m$. Ofwel

$$\beta(\alpha^{-m})^i = \alpha^j.$$

Er wordt van tevoren een tabel gemaakt van α^j voor alle j, $0 \leq j \leq m$, de zogenaamde *baby steps*. De te doorlopen rij is $\beta\alpha^0$, $\beta\alpha^{-m}$, $\beta\alpha^{-2m}$ ... $\beta\alpha^{-im}$, de *giant steps*. Na iedere *giant step* wordt gekeken of het resultaat terug te vinden is in de tabel, dus of $\exists\, j : \alpha^j = \beta\alpha^{-im}$. Zo ja, dan wordt x gelijk aan $j + im$.

Pollard's rho-algoritme
Dit algoritme maakt gebruik van Floyd's *cycle-finding*-algoritme: Zij gegeven een rij getallen $x_1, x_2 \ldots$, $0 \leq x_i < m$ voor zekere m. Zij verder $f(x)$ een functie met de eigenschap dat $0 \leq f(x) < m$ als x tussen 0 en m. Bekijk de rij voortgebracht door $x_{i+1} = f(x_i)$. Floyd zegt nu dat er een $j > 0$ is zó dat $x_j = x_{2j}$. Dit kan eenvoudig worden ingezien als je bedenkt dat een dergelijke rij x_i's altijd periodiek is. Je vindt deze j, of liever dit paar (x_j, x_{2j}), door achtereenvolgens paren x_l, x_{2l} uit te rekenen, voor $l = 0, 1, 2 \ldots$. Om de beschrijving eenvoudig te houden wordt aangenomen dat de orde van de groep n priem is.

De groep G wordt verdeeld in drie disjuncte deelverzamelingen S_1, S_2 en S_3 van ongeveer gelijke grootte op basis van een eenvoudig te verifiëren eigenschap, bijvoorbeeld restklassen modulo 3. Er wordt een rij x_i als volgt gedefinieerd

$$x_{i+1} = f(x_i) = \begin{cases} \beta \cdot x_i, & \text{als } x_i \in S_1, \\ x_i^2, & \text{als } x_i \in S_2, \\ \alpha \cdot x_i, & \text{als } x_i \in S_3, \end{cases}$$

Ieder element x_j in deze rij is van de vorm $\alpha^{a_j}\beta^{b_j}$, voor bekende $a_j, b_j \in N$.

Floyd's algoritme kan nu worden gebruikt om twee rijelementen x_i en x_{2i} te vinden zó dat $x_i = x_{2i}$. Dan geldt $\alpha^{a_i}\beta^{b_i} = \alpha^{a_{2i}}\beta^{b_{2i}} \pmod{n}$. Herschikken en logaritme met grondtal α nemen levert dan

$$(b_i \Leftrightarrow b_{2i}) \cdot \log_\alpha \beta = a_{2i} \Leftrightarrow a_i \pmod{n}.$$

Aangenomen dat $b_i \neq b_{2i} \pmod{n}$ kan nu x worden berekend. De kans dat $b_i = b_{2i}$ is verwaarloosbaar klein.

Pohlig-Hellman algoritme
Pohlig-Hellman gebruikt de factorisatie van de groepsorde om de Chinese reststelling te kunnen toepassen. Deze ontbinding is niet a priori bekend.

Zij de factorisatie van de groepsorde $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$. Voor alle i wordt $x_i = x \bmod p_i^{e_i}$ berekend. De Chinese reststelling zegt dat voor getallen die paarsgewijs relatief priem zijn, hier de priemmachten $p_i^{e_i}$, de volgende rij congruenties een unieke oplossing modulo $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ heeft
$x_1 \equiv x \pmod{p_1^{e_1}}$
$x_2 \equiv x \pmod{p_2^{e_2}}$
$\vdots$
$x_r \equiv x \pmod{p_r^{e_r}}$.
Gauss' algoritme kan nu worden gebruikt om uit deze x_i's het getal x te berekenen.
Er is overigens voor cryptografisch grote getallen een beter algoritme bekend om x uit deze x_i's te berekenen, Garner's algoritme [21, bladzijden 612–613] .

Index-calculus methode
Deze methode is de meest krachtige die op dit moment bekend is. Net als in baby-step giant-step wordt een tabel gecontrueerd met daarin geschikt gekozen groepselementen. Ook hier wordt deze tabel vervolgens gebruikt om te proberen 'nieuwe' machten uit te drukken in 'bekende machten', de tabel-elementen.

Voor deze tabel wordt een relatief kleine deelverzameling S van G gekozen, de zogenaamde factorbasis, zó dat een voldoende groot deel van de elementen van G geschreven kan worden als produkt van elementen uit S. De keuze van een geschikte basis zal hier niet verder worden besproken.

Vervolgens worden lineaire vergelijkingen gevormd van de logaritmen van de elementen van S. Er worden $t + c$ vergelijkingen gezocht, waarbij t het aantal elementen in S is en c een relatief kleine constante die de kans op een oplossing van het stelsel voldoende groot moet maken. In het resulterende stelsel vergelijkingen wordt aan beide kanten $\log_\alpha$ genomen. Uit het stelsel kunnen nu de logaritmen van de elementen uit de factorbasis worden berekend. Om x te bepalen wordt geprobeerd $\log_\alpha \beta$ te schrijven als lineaire combinatie van de logaritmen van de factorbasiselementen.

Voor een volledige beschrijving van deze algoritmen zie [21].

Hoofdstuk 4
Digitale Handtekeningen

In de laatste klas van de basisschool moest ik mijn handtekening creëren; er was een vrijdagmiddag voor uit getrokken. Ik kreeg een stapel A4-tjes en op het bord stonden de eisen waaraan mijn handtekening diende te voldoen: hij moest reproduceerbaar zijn (maar dat kwam vanzelf verzekerde de meester ons), ik moest in ieder geval mijn voorletter en mijn achternaam erin verwerken en het zou handig zijn als ik de echte van vervalsingen zou kunnen onderscheiden door bijvoorbeeld altijd aan de D een extra 'kringeltje' te maken. De hele middag heb ik eraan gewerkt; alleen de D, ook de E [1], een slinger na de 't' of een streep onder de hele handtekening? Beslissingen, beslissingen, beslissingen.

Die middag is mijn handtekening geboren. Intussen doe ik alweer 12 jaar met die handtekening en ik moet bekennen dat hij minder en minder op zichzelf gaat lijken. Ook zou ik hem absoluut niet van een serieuze vervalsing kunnen onderscheiden. En mijn voorletters en achternaam zijn al in de beginjaren door ongedefinieerde lussen verdrongen.

In dit hoofdstuk zal eerst geprobeerd worden af te bakenen wat een handtekening is en welke functie hij vervult. Daarna wordt een aantal eigenschappen van (digitale) handtekeningen besproken en een algemene beschrijving gegeven van een digitaal signeeralgoritme. In de tweede paragraaf gaat over hashfuncties, een belangrijk hulpmiddel bij digitaal signeren.

1. Handtekening: geschreven en digitaal

Een handtekening is een manier om je identiteit met informatie te verbinden. Het drukt goedkeuring, bijvoorbeeld bij een koopcontract, of auteurschap, bijvoorbeeld bij een schilderij, van de ondertekenaar uit. Omdat een handtekening een verbinding is tussen informatie en een persoon heeft hij alleen betekenis in combinatie met de plaats waar hij zich bevindt, bijvoorbeeld onderaan een contract; een handtekening op een leeg vel papier betekent niets.

Waarom is een handtekening zo handig? Aan een handtekening worden – juridisch – de volgende eigenschappen toegedicht:

1. Een handtekening is authentiek. De handtekening overtuigt de ontvanger dat de ondertekenaar de informatie willens en wetens heeft gesigneerd.

2. Een handtekening is niet te vervalsen. De handtekening bewijst dat de ondertekenaar, en niemand anders, de informatie heeft gesigneerd.

[1] van Eleonore.

3. De gesigneerde informatie kan niet worden veranderd. De handtekening is een bewijs dat de informatie nog dezelfde vorm heeft als toen het werd ondertekend.

4. De handtekening kan niet worden ontkend. Immers, alleen de rechtmatige 'eigenaar' van de handtekening kan deze hebben geplaatst.

Natuurlijk voldoen geschreven handtekeningen in de praktijk niet aan deze eisen: handtekeningen worden vervalst, het is mogelijk een contract na signeren te veranderen en een handtekening kan – met wat moeite – best van het ene document naar het andere worden gekopieerd. Dat we toch bereid zijn handtekeningen te vertrouwen is te danken aan enerzijds de grote moeite die iemand zich moet getroosten om een handtekening te vervalsen en anderzijds het risico dat een vervalser loopt.

Je zou graag een digitaal equivalent van de geschreven handtekening willen met dezelfde eigenschappen. In hoofdstuk 2 over cryptografie is al even kort genoemd welke eigenschappen een handtekening moet hebben:

data-integriteit Als Bob een boodschap van Alice krijgt kan hij controleren of Eve geen informatie aan het bericht heeft toegevoegd of uit het bericht heeft verwijderd.

authenticiteit Voordat Alice met Bob gaat praten wil ze zeker weten dat Bob niet stiekem Eve is die zegt dat ze Bob is. Daarnaast wil Bob kunnen zien dat een boodschap die zegt van Alice te komen ook echt van Alice komt en niet van Eve.

erkenning Als Alice een boodschap heeft verstuurd naar Bob en dit later ontkent kan Bob dankzij deze eigenschap aan derden aantonen dat Alice en niemand anders deze boodschap aan hem heeft verstuurd.

Bij een geschreven handtekening bepaalt de plaats, bijvoorbeeld onderaan een contract, waarop deze van toepassing is. Bij een digitale handtekening is dat echter minder duidelijk. Hoe kun je een digitale handtekening met digitale informatie verbinden? Of meer praktisch, hoe kun je twee bitstrings met elkaar verbinden zodat het niet eenvoudig is ze los te koppelen en/of afzonderlijk te kopiëren of te veranderen? Een letterlijke omzetting van een geschreven handtekening naar digitale vorm is niet voldoende. Een bit-representatie van een geschreven handtekening die achter een document is geplakt kan worden 'los geknipt' en aan andere documenten worden vastgemaakt. Bovendien kan een dergelijke handtekening eenvoudig worden gekopieerd. Daarenboven kun je het betreffende document daarna nog veranderen zonder sporen achter te laten.

Je zoekt het equivalent van de fysieke verbinding die een geschreven handtekening heeft met de gesigneerde informatie; de handtekening moet onlosmakelijk met de informatie worden verbonden. Omdat in digitale vorm alles uiteindelijk als bitstring wordt gerepresenteerd wordt aangenomen de te signeren informatie zowel als de handtekening een rij nullen en enen is. Zo bezien zijn het dus twee getallen en dringt de volgende oplossing zich op.

De handtekening wordt rekenkundig *op* de boodschap geplaatst, door een wiskundige bewerking uit te voeren met beide strings als invoer. De handtekening hoort daardoor specifiek bij deze en geen andere informatie.

Welke wiskundige functies komen hiervoor in aanmerking?
In de afgelopen jaren zijn veel wiskundige functies voorgesteld, waarbij vooral de one-way functies van public-key cryptosystemen opvallen. Het lijkt erop dat de eisen die aan een digitale signeermethode worden gesteld overeenkomen met die voor een cryptosysteem. In hoofdstuk 6, waarin de eerste van de twee signeeralgoritmen uit dit onderzoek wordt beschreven, wordt de overeenkomst tussen public-key cryptografie en digitale signeermethoden uitgebreid behandeld. In dit hoofdstuk worden geen specifieke wiskundige functies besproken.

Om digitaal te kunnen signeren moet gebruik worden gemaakt van een zogenaamd digitale handtekeningenalgoritme. Zo'n algoritme moet enerzijds Alice in staat stellen informatie in digitale vorm te onder-

tekenen en anderzijds Bob de gelegenheid bieden de handtekening van Alice te verifiëren. Digitaal signeren – en de bijbehorende methode – bestaat daarom uit drie stappen:

handtekening creëren Alice moet eerst een handtekening maken.

signeren Alice moet de informatie kunnen ondertekenen. Ze moet dus haar handtekening aan de informatie kunnen koppelen.

verifiëren Bob moet – liefst eenvoudig en snel – kunnen controleren dat de boodschap inderdaad door Alice is ondertekend.

2. Hashfuncties

Een belangrijk hulpmiddel bij digitale handtekeningen is de zogenaamde hashfunctie.

> *There is no definition for this word – nobody knows what hash is.* Ambrose Bierce (The Devil's dictionary, 1906)

Een hashfunctie wordt gebruikt[2] om een bitstring van willekeurige, eindige lengte, de boodschap, om te zetten in een zogenaamde *hashwaarde* van vaste, kleine bitlengte.

Waarom hashen?
Zoals in de vorige paragraaf is beschreven wordt de digitale handtekening *op* de boodschap geplaatst. In de praktijk is die boodschap echter vaak heel groot. Wanneer met deze volledige boodschap gerekend moet worden kost dat veel tijd. In plaats daarvan wordt daarom een bitstring kenmerkend voor de boodschap, de hashwaarde, ondertekend. [6]

Behalve de eerdergenoemde eigenschap dat een hashfunctie iedere input moet kunnen reduceren of samenpersen tot een bitstring van vaste lengte, moet een hashfunctie ook 'eenvoudig' te berekenen zijn. Het moet niet zo zijn dat de hashfunctie bepalend wordt voor de snelheid van het algoritme; dan wordt de winst van het ondertekenen van de hashwaarde in plaats van de boodschap immers teniet gedaan.

Kijkend naar de beschrijving van een hashfunctie valt direct op dat deze functie meerdere orginelen op hetzelfde beeldelement afbeeldt. Hieronder zijn een aantal eigenschappen van hashfuncties gedefinieerd die deze 'dubbele' projectie, *collision*, aan banden legt.

1. Voor bijna alle elementen $h(m)$ uit het bereik van de hashfunctie h is het zo goed als onmogelijk om de orginele boodschap in het domein te vinden. Ofwel de hashfunctie moet one-way zijn.

2. Gegeven een boodschap m en de bijbehorende hashwaarde $h(m)$ is het zo goed als onmogelijk een tweede element m' te vinden zodat $h(m) = h(m')$.

3. Het is zo goed als onmogelijk twee verschillende boodschappen m_1 en m_2 te vinden zo dat $h(m_1) = h(m_2)$.

Hoewel intuïtief duidelijk, zijn de zinsdelen "voor bijna alle elementen" en "zo goed als onmogelijk" vaag. Duidelijk is dat *enige* ruis best is toestaan, maar hoeveel? Je kunt je afvragen of je deze begrippen niet naar strikte wiskundige regels kunt vertalen: Is 'voor bijna alle' uit te drukken in een kans? Zegt 'zo goed als onmogelijk' iets over de complexiteit van het onderliggende algoritme?
Bij de bespreking van de gebruikte hashfunctie MD4 zullen 'harde' cijfers worden gebruikt, maar voor de rest van dit hoofdstuk zijn de hierboven gegeven, nonchalante definities voldoende. We bevinden ons daarmee overigens in goed gezelschap:

> *We will call a task "computationally infeasible" if its cost as measured by either the amount of memory used or the runtime is finite but impossibly large.*
> Diffie en Hellman in New Directions in Cryptography (1976) [5].

[2] Als in dit verslag over hashfuncties wordt gesproken, worden alleen hashfuncties bedoeld die gebruikt worden of zouden kunnen worden in digitale handtekeningen-algoritmen.

Hoe ziet zo'n hashfunctie eruit?
Hashfuncties zijn vaak iteratieve functies die de boodschap omvormen door iedere keer een bitstring van vaste lengte, een blok, te 'hashen' totdat de hele boodschap is verwerkt. Een belangrijk kenmerk van een hashfunctie is de voortschrijdende uitvoer: ieder nieuw blok dat wordt ingelezen bepaalt samen met het tussentijds resultaat de nieuwe hashwaarde.

Een hashfunctie start met een hashwaarde H_0, de hashwaarde van een lege boodschap zogezegd. De eerste stap is de boodschap m een lengte te geven die een veelvoud t is van de lengte van een blok. Soms wordt nog ruimte gelaten om informatie over de oorspronkelijke boodschap op te nemen; in MD4 wordt bijvoorbeeld 64 bits gereserveerd voor de *lengte* van de oorspronkelijke boodschap. Voor de lengte van een blok wordt meestal een macht van 2 gekozen, 512 ($=2^9$) bits bijvoorbeeld in MD4. Het verlengen van de bitstring van m gebeurt door middel van 'padding': aan het eind van de string worden net zolang nullen [3] toegevoegd tot deze de gewenste lengte heeft. Dan wordt het eerste blok x_1 ingelezen. Dit blok dient samen met startwaarde H_0 als invoer voor een iteratieve compressiefunctie die een nieuwe hashwaarde H_1 retourneert. Deze stap wordt herhaald totdat de hele boodschap m is ingelezen. Het resultaat H_t van het laatste blok x_t gehasht met H_{t-1} is de hashwaarde $h(m)$ van de boodschap.

De interne functie is zoals gezegd iteratief. Ze bestaat uit een aantal ronden, waarin telkens een complexe bitmanipulatie, een combinatie van de logische functies *and*, *or* en *exclusive or*, wordt uitgevoerd op de bitstrings van het huidige blok en het laatste resultaat. Zie de beschrijving van MD4 in bijlage 13 voor een voorbeeld van dergelijke bitmanipulatie.

[3] In principe kan iedere bitstring worden gebruikt om te completeren; in de praktijk wordt vaak voor nullen gekozen.

Hoofdstuk 5
Digital Signature Algorithm

Digital Signature Algorithm is een algoritme voor het plaatsen van digitale handtekeningen gebaseerd op het discrete logaritme-probleem in de groep $\mathbb{F}_p$. Het bouwt voort op een eerder digitale handtekeningen-algoritme van Tahar ElGamal [7]. In zijn artikel uit 1984 presenteert El Gamal '*a new signature scheme; the security of the system relies on the difficulty of computing discrete logarithms over finite fields.*'. Het artikel beschrijft eerst een implementatie van een public-key systeem gebaseerd op het sleutel-distributiesysteem van Diffie en Hellman [5]. El Gamal gebruikt dit public-key cryptosysteem vervolgens om een digitale handtekeningen-algoritme op te zetten.

Dit hoofdstuk begint met een korte beschrijving van de hierboven genoemde systemen van Diffie en Hellman en ElGamal, voorlopers van huidige discrete logaritme-systemen vormen. In de vierde paragraaf volgt dan een volledige beschrijving van het DSA-algoritme[1]. Daarna wordt het algoritme opnieuw doorgelopen en waar nodig uitvoeriger besproken.

1. Diffie-Hellman sleuteldistributie

Het sleutel-distributiesysteem van Diffie en Hellman was een antwoord op een belangrijke onderzoeksvraag binnen private-key cryptografie: de distributie van de geheime sleutel.

In private-key cryptografie moeten Alice en Bob vóór ze kunnen communiceren eerst een sleutel overeenkomen, omdat deze voor codering en ontcijfering gelijk is. Een van de grote problemen bij het gebruik van een private-key cryptosysteem is de distributie van die sleutel. Om geheimhouding te garanderen moet de sleutel over een veilig kanaal worden verzonden. Je zou echter liever gebruik willen kunnen maken van een onveilig kanaal om een sleutel overeen te komen. Diffie en Hellman doen in hun artikel een voorstel voor zo'n distributiesysteem, gebaseerd op de discrete logaritme in de groep $\mathbb{F}_p^*$ met p priem. Hun voorstel was een verbetering van een al bestaand distributiesysteem van Merkle; dat algoritme zal hier echter niet worden besproken.

Het systeem gaat uit van de groep $\mathbb{F}_p^*$ met p priem en een voortbrenger α. Alice genereert een random getal x_A uniform gekozen uit het interval $[1, p \Leftrightarrow 1]$. Ze houdt x_A geheim, maar plaatst

$$y_A = \alpha^{x_A} \bmod p$$

in het publieke domein. Bob doet hetzelfde en genereert x_B en plaatst $y_B = \alpha^{x_B} \bmod p$ in het publieke

[1] Hoewel niet correct, wordt DSA in de literatuur vaak aangeduid als DSA-algoritme. Deze gewoonte zal hier worden gevolgd.

domein. Als ze nu willen communiceren fungeert

$$K_{AB} = \alpha^{x_A x_B} \bmod p$$

als hun geheime sleutel. Alice en Bob kunnen beiden deze sleutel eenvoudig construeren met behulp van hun eigen geheime sleutel en de informatie van de ander in het publieke domein. Alice berekent bijvoorbeeld

$$\begin{aligned} K_{AB} &= y_B^{x_A} \bmod p \\ &= (\alpha^{x_B})^{x_A} \bmod p = (\alpha^{x_A})^{x_B} \bmod p. \end{aligned}$$

Eve, die alleen y_A en y_B tot haar beschikking heeft, kan K_{AB} alleen construeren door bijvoorbeeld

$$K_{AB} = y_A^{\log_\alpha y_B} \bmod p$$

te berekenen, waarvoor ze de discrete logaritme $\log_\alpha y_B$ op moet lossen. Diffie en Hellman concluderen – iets voorzichtiger:

> *We thus see that if logs mod p are easily computed the system can be broken. While we do not currently have a proof of the converse (i.e., that the system is secure if logs mod p are difficult to compute), neither do we see any way to compute K_{AB} from y_A en y_B without first obtaining either x_A or x_B.*

2. ElGamal public-key systeem

ElGamal baseert zijn public-key systeem op het hierboven beschreven public-key distributiesysteem. Hij neemt aan dat Alice en Bob – zoals hierboven – ieder een geheime en een publieke sleutel hebben ontwikkeld.

Stel Alice wil Bob een boodschap m zenden, $0 \leq m \leq p \Leftrightarrow 1$.[2] Vervolgens kiest Alice een random getal k tussen $0 < k < p \Leftrightarrow 1$. Alice berekent de sleutel

$$K = y_B^k \bmod p.$$

De gecodeerde boodschap is nu het getallenpaar (c_1, c_2) met

$$c_1 = \alpha^k \bmod p, \qquad c_2 = K \cdot m \bmod p.$$

De vermenigvuldiging in c_2 kan worden vervangen door ieder andere inverteerbare bewerking – of een combinatie daarvan – zoals optelling modulo p.

Bob ontcijfert het bericht nu als volgt. Eerst berekent hij K uit c_1. K was $y_B^k \bmod p$ dus

$$\begin{aligned} c_1^{x_B} \bmod p &= (\alpha^k)^{x_B} \bmod p \\ &= (\alpha^{x_B})^k \bmod p \\ &= y_B^k \bmod p \qquad = K. \end{aligned}$$

Vervolgens deelt Bob c_2 door K en vindt hij de boodschap m. Het kraken van dit systeem is hetzelfde als het kraken van Diffie-Hellman distributie: om K uit te rekenen moet Eve k kennen en die kan alleen worden berekend uit $c_1 = \alpha^k \bmod p$.

3. ElGamal's digitale handtekeningen-algoritme

Dit algoritme werkt in de groep F_p^* met voorbrenger α. De boodschap m ligt weer tussen 0 en $p \Leftrightarrow 1$. In het publieke domein staan nog altijd de sleutels y_A en y_B van Alice en Bob.

[2] ElGamal neemt niet aan dat p priem is; hij eist alleen dat $p-1$, de orde van de groep, tenminste 1 grote priemfactor heeft.

> *To sign a document , Alice should be able to use her secret key x_A to find a signature for m in such a way that all users can verify the authenticity of the signature using her public key y_A (together with α and p), and no one can forge her signature without knowing her secret key x_A.*
> ElGamal in de inleiding van [7].

Een handtekening is bij ElGamal een getallenpaar (r, s), $0 \leq r, s < p \Leftrightarrow 1$, dat voldoet aan de vergelijking

$$\alpha^m = y^r r^s \bmod p,$$

waar y de publieke sleutel van Alice is. Als Alice nu een boodschap wil signeren kiest ze een random getal k, $0 < k \leq p \Leftrightarrow 1$, $\text{ggd}(k, p \Leftrightarrow 1) = 1$ en berekent

$$r = \alpha^k \bmod p.$$

De eerste vergelijking kan nu als volgt worden herschreven

$$\alpha^m = \alpha^{xr} \alpha^{ks} \bmod p$$

met x de geheime sleutel van Alice. Deze vergelijking geeft een oplossing s via

$$m = xr + ks \bmod p \Leftrightarrow 1,$$

waar $p \Leftrightarrow 1$ de orde van α is. Nota bene, dit kan alleen als k en $p \Leftrightarrow 1$ inderdaad copriem zijn, zoals bij de keuze van k werd geëist.

Alice stuurt Bob nu haar boodschap m en haar handtekening (r, s). Als Bob deze handtekening wil verifiëren gaat hij als volgt te werk. Hij gebruikt de relatie

$$\alpha^m = y^r r^s \bmod p.$$

Hij berekent α^m en kijkt of dit gelijk is aan $y^r r^s = \bmod p$. Alleen als deze twee gelijk zijn, accepteert Bob de handtekening.

4. Beschrijving DSA-algoritme

Het DSA-algoritme lijkt in vorm en inhoud sterk op zijn voorganger, het ElGamal systeem. Het is ontworpen door het Amerikaanse National Institute of Standards and Technology (NIST) en in augustus 1991 voorgedragen als mogelijke standaard. In 1994 is het algoritme – uitgebreid met een aantal gebruiksvoorwaarden – aangenomen onder de naam Digital Signature Standard (DSS).

In deze paragraaf wordt een beschrijving van het DSA-algoritme gegeven. Daarna volgt een meer precieze bespreking van de verschillende stappen. In tegenstelling tot ElGamal, geeft DSA ook een methode om de groep $\mathbb{F}_p^*$ en een voortbrenger α van die groep te vinden. Deze *sleutelgeneratie* wordt als eerste beschreven, daarna volgen *signeren* en *verifiëren*.

sleutel genereren

Iedere deelnemer maakt een publieke sleutel en een corresponderende geheime sleutel. Om een sleutelpaar te maken doet Alice het volgende:

1. Ze kiest een priemgetal q zó dat $2^{159} < q < 2^{160}$.
2. Ze kiest t zó dat $0 \leq t \leq 8$, en kies een priemgetal p met $2^{511+64t} < p < 2^{512+64t}$, met de eigenschap dat $q \mid (p \Leftrightarrow 1)$.
3. (Ze kiest een voortbrenger α van de unieke, cyclische groep van orde q in $\mathbb{F}_p^*$.)
   3.1 Ze kiest een element $g \in \mathbb{F}_p^*$ en berekent $\alpha = g^{(p-1)/q} \bmod p$.

3.2 Als $\alpha = 1$, kiest ze een nieuwe g (stap 3.1).

4. Ze kiest een random getal a zó dat $1 \leq a \leq q \Leftrightarrow 1$.
5. Ze berekent $y = \alpha^a \bmod p$.

Alice's publieke sleutel is (p, q, α, y); haar geheime sleutel is a.

signeren
Alice signeert een boodschap met behulp van haar geheime sleutel. Om Bob bovendien te kunnen garanderen dat de boodschap onderweg niet is veranderd, versleutelt ze de hashwaarde van de boodschap in haar handtekening.

1. Ze kiest een random getal k, $0 < k < q$.
2. Ze berekent $r = (a^k \bmod p) \bmod q$.
3. Ze berekent $k^{-1} \bmod q$.
4. Ze berekent $s = k^{-1}\{h(m) + ar\} \bmod q$.

Alice's handtekening voor boodschap m is het getallenpaar (r, s).

verifiëren
Om de handtekening van Alice te verifiëren controleert Bob het volgende:

1. Hij haalt Alice's publieke sleutel (p, q, α, y) op.
2. Hij verifieert dat $0 < r < q$ en $0 < s < q$; zo niet, dan weigert hij de handtekening.
3. Hij berekent $w = s^{-1} \bmod q$ en $h(m)$ (aangenomen $s \neq 0$).
4. Hij berekent $u_1 = w \cdot h(m) \bmod q$ en $u_2 = rw \bmod q$.
5. Hij berekent $v = (\alpha^{u_1} y^{u_2} \bmod p) \bmod q$.

Bob accepteert de handtekening dan en slechts dan als $v = r$.

Nota bene, s^{-1} bestaat alleen als $s \neq 0$. Om er zeker van te zijn dat Bob de handtekening kan verifiëren zou Alice bij het genereren van de handtekening kunnen controleren of $s \neq 0$. De kans dat $s = 0$ is erg klein; omdat alle getallen random gegenereerd worden kan s alle waarden tussen 0 en q aannemen. De kans dat $s = 0$ is daarom $1/(q \Leftrightarrow 1)$, wat voor grote q verwaarloosbaar klein is.

waarom het werkt
Waarom accepteert Bob de handtekening alleen wanneer $v = r$?
Als (r, s) een legitieme handtekening van Alice is voor boodschap m dan moet gelden (stap 4 signeren)

$$s = k^{-1}\{h(m) + ar\} \bmod q$$

ofwel $h(m) = \Leftrightarrow ar + ks \bmod q$.
Vermenigvuldigen met $w(=s^{-1})$ en herschikken geeft $w \cdot h(m) + arw = k \bmod q$. Maar dat is precies $u_1 + au_2 = k \bmod q$. Verheffen tot de macht α aan beide kanten geeft

$$(\alpha^{u_1} y^{u_2} \bmod p) \bmod q = (\alpha^k \bmod p) \bmod q.$$

Dus $v = r$.
Dit bewijst dat als $v \neq r$ de handtekening onmogelijk van Alice kan zijn.

5. Bespreking DSA-algoritme

De beschrijving van het DSA-algoritme in de vorige paragraaf is zoals die in de literatuur te vinden is. Zo 'recht door zee' als het hier gebracht wordt is het in de praktijk echter niet. Hoe kom je bijvoorbeeld aan zo'n priempaar p en q? En waarom kun je bij het zoeken van een voortbrenger weer opnieuw beginnen als $\alpha = 1$?

In deze paragraaf wordt op deze twee vragen antwoord gegeven. Verder zal in hoofdstuk 6 over de implementatie van DSA nog gesproken worden over random getallen en efficiënt modulorekenen. In hoofdstuk 9, over de vergelijking van de twee systemen, zullen de intervalseisen voor p en q worden toegelicht.

Het vinden van twee geschikte priemen

Het vinden van een priemgetal is niet eenvoudig: zoveel zijn er niet en bovendien is niet precies bekend hoe ze verdeeld zijn over de natuurlijke getallen. Zijn er wel genoeg? Dat wel, er zijn oneindig veel priemgetallen.

De priemgetallenstelling zegt dat

$$\pi(x) \sim \frac{x}{\ln x}, \qquad x \to \infty,$$

waar $\pi(x)$ het aantal priemen is tussen 0 en x. Bovendien zijn ze redelijk uniform verdeeld. De kans dat een willekeurig getal in de buurt van x priem is, is dus ongeveer $1/(\ln(x)$.

Het genereren van een sleutel begint met het vinden van een priempaar (p, q) met bovendien de eis dat $q \mid (p \Leftrightarrow 1)$. (Waarom q een deler moet zijn van $p \Leftrightarrow 1$ wordt verderop uitgelegd.) Het is al snel duidelijk dat je eerst die q zult moeten vinden en vervolgens proberen of een van de veelvouden van deze q plus één, $k \cdot q + 1$, in het gewenste interval een priemgetal is. Eerst een priemgetal q dus, tussen 2^{159} en 2^{160}, ofwel van 160 bits.

Voor het vinden van priemen kun je gebruik maken van zogenaamde *priemgeneratoren*. Deze doorlopen het volgende algoritme:

1. Genereer een kandidaatpriem n, die oneven is en van de gewenste bitlengte.
2. Test of n priem is.
3. Zo ja, n is de gezochte priem, zo nee, ga terug naar stap 1.

Het vinden van priemen valt dus uiteen in twee stappen: het vinden van een kandidaatpriem en bepalen of het gevonden getal inderdaad priem is. De combinatie van deze twee stappen heet – verwarrend – priemgeneratie. De eerste stap zal *kandidaatgeneratie* worden genoemd.

Testen of een getal priem is of samengesteld gebeurt met een zogenaamde *priemtest*. Aan de wieg van alle priemtesten staat de volgende stelling.

Kleine stelling van Fermat

Als n een priemgetal is, a een natuurlijk getal tussen 1 en $n \Leftrightarrow 1$ en $ggd(a, n) = 1$, dan geldt

$$a^{n-1} \equiv 1 \pmod{n}.$$

Bewijzen dat een kandidaatpriem samengesteld is, is dus hetzelfde als een a vinden waarvoor deze stelling niet geldt. Echter, bewijzen dat een getal *priem* is, blijkt nog niet mee te vallen. Want hoewel Fermat kan aantonen dat een getal samengesteld is, dat een getal priem is staat – met alleen deze stelling als hulpmiddel – nooit helemaal vast. Sterker, er zijn getallen, de zogenaamde *Carmichael getallen*, waarvoor bovenstaande congruentie geldt voor iedere a, maar die tóch samengesteld zijn. Het kleinste Carmichaelgetal is $c = 561 = 3 \cdot 11 \cdot 17$. Gelukkig zijn er heel weinig Carmichaelgetallen, beduidend minder dan er priemen zijn, al is recent wel bewezen dat het er oneindig veel zijn.

Er zijn twee soorten priemtests: echte (*provable*) en pseudopriemtests (*probable*). De eerste soort stelt met wiskundige zekerheid vast dat een getal priem is, de laatste maakt het (zeer) aannemelijk.

Het bestaan van Carmichaelgetallen maakt het ontwerpen van echte priemtests bijzonder lastig. Echte priemtests worden in de praktijk nauwelijks gebruikt, hooguit *na* een pseudopriemtest, omdat ze veel tijd kosten en daarom duur zijn. De priemgeneratoren in de cryptografische wereld maken overigens bijna allemaal gebruik van *pseudopriemtests*. De bekendste en meest gebruikte test is de zogenaamde Miller-Rabin test, die ook voor DSA wordt aangeraden. Er wordt hier geen volledige beschrijving van deze test gegeven (zie daarvoor [21]). De Miller-Rabin test is gebaseerd op een verfijning van de kleine stelling van Fermat:

> Zij n een oneven priem, en $n \Leftrightarrow 1 = 2^s r$ voor zekere s en oneven r. Als a nu een natuurlijk getal is met $\gcd(a, n) = 1$ dan geldt óf $a^r \equiv 1 \pmod{n}$ óf $a^{2^j} \equiv \Leftrightarrow 1 \pmod{n}$ voor zekere j, $0 \leq j \leq s \Leftrightarrow 1$.

Het algoritme test nu voor een random grondtal a of het – voor te onderzoeken getal n – voldoet aan een van deze twee equivalentierelaties. Als dit niet het geval is, is n samengesteld. Als Miller-Rabin 'samengesteld' retourneert is de invoer zeker niet priem, omdat voor priemgetallen bovenstaande stelling geldt. Evenzo zal het algoritme als n priem is, altijd 'priem' retourneren. Maar hoe vaak heeft Miller-Rabin ongelijk? Ofwel, hoe groot is de kans dat het algoritme een samengesteld getal 'priem' noemt?

Je kunt uitrekenen dat de kans dat een samengesteld getal 'priem' wordt genoemd hooguit $\frac{1}{4}$ is. Herhalen we de test voor t onafhankelijk van eerder gekozen random grondtallen a, dan is de kans dat Miller-Rabin een samengesteld getal 'priem' blijft noemen kleiner dan $(\frac{1}{4})^t$. Door deze t, de zogenaamde veiligheidsparameter, te variëren kan de gebruiker zelf bepalen welk risico hij of zij loopt.

Zij tot slot nog opgemerkt dat het handiger is om eerst voor een zekere grens B 'met de hand' te testen of het getal deelbaar is door priemen $< B$, omdat de kans dat een random getal een kleine priemdeler heeft relatief groot is.

Vóór de priemtest moet echter eerst een kandidaatpriem worden gegenereerd. Zoals hierboven werd gezegd is het aantal priemen tot x ongeveer gelijk aan $\frac{x}{\ln(x)}$. Om een k-bits priem te vinden zou je dus herhaaldelijk een oneven k-bits getal kunnen testen totdat je er een vindt die 'priem' wordt genoemd. Dit wordt *random search* genoemd. Om de kans op een priem te vergroten is in de loop der jaren veel onderzoek gedaan naar 'slimmere' genereeralgoritmen. Bovendien worden er in de cryptografie vaak extra eisen gesteld aan de gezochte priemen, zoals in DSA bijvoorbeeld waar een priem*paar* (p, q) wordt gezocht met de eigenschap dat $q \mid (p \Leftrightarrow 1)$. NIST heeft speciaal voor DSA een priemgenerator voor p en q ontwikkeld, die voor het testgedeelte gebruik maakt van Miller-Rabin. Een volledige beschrijving van dit algoritme wordt hier niet gegeven (zie hiervoor [21]). In dit algoritme wordt – zoals hierboven al opgemerkt – eerst q van bitlengte g 'gemaakt'. Dit gebeurt met behulp van een hashfunctie h, die een getal van g bits retourneert, en een random startwaarde s. Er wordt een getal u berekend door bitsgewijs $h(s)$ en $h(s, g)$ (de hashwaarde van een bepaalde combinatie van s en g) op te tellen. De priemgenerator retourneert voor q de string u, waarvan de eerste en de laatste bit gelijkgesteld zijn aan 1 (waardoor het een oneven getal van gewenste bitlengte wordt.). Deze q wordt getest met behulp van Miller-Rabin. Vervolgens wordt een p geconstrueerd door de concatenatie van verschillende hashwaarden, totdat p de vereiste bitlengte heeft. Aan het eind van deze berekening wordt door modulo rekenen gegarandeerd dat $p \Leftrightarrow 1$ inderdaad wordt gedeeld door q. Na een pseudopriemtest van deze p is het paar gevonden.

Het vinden van de q-ondergroep

Nadat p en q bekend zijn moet de unieke cyclische ondergroep van $\mathbb{F}_p^*$ van orde q worden gevonden, waarin de rest van het algoritme wordt gerekend. Dat deze ondergroep bestaat zal hier niet worden bewezen (zie hiervoor [16]). Er wordt gesteld dat als $\alpha (= g^{(p-1)/q} \bmod p) \neq 1$ voor een $g \in \mathbb{F}_p^*$, α een voortbrenger is van de cyclische ondergroep van orde q. Waarom is dat zo?

De hoofdgroep $\mathbb{F}_p^*$ heeft orde $p \Leftrightarrow 1$ omdat p priem is. Deze orde $p \Leftrightarrow 1$ is te schrijven als qd voor

twee positieve getallen q en d. Als α nu een voortbrenger is van $\mathbb{F}_p^*$, dan wordt de ondergroep van orde q voortgebracht door $\alpha^d = \alpha^{n/q}$. De vraag verplaatst zich dus naar de oorspronkelijke groep: is de willekeurig gekozen g een voortbrenger van $\mathbb{F}_p^*$? Als deze g geen voortbrenger is, dan is α dat ook niet van de ondergroep, anders wel. Deze g is een voortbrenger als g tot de macht $\frac{p-1}{q}$ ongelijk is aan 1.

Hoofdstuk 6
Simulatie van DSA in F_p

In dit hoofdstuk wordt het eerste simulatieprogramma DSA in $\mathbb{F}_p^*$ beschreven. Een aantal opmerkingen en verdediging van keuzes geldt ook voor het tweede simulatieprogramma (DSA over elliptische krommen) zoals bijvoorbeeld het gebruikte multilengte-softwarepakket en de hashfunctie. Deze keuzes worden aan het begin van dit hoofdstuk toegelicht; bij de bespreking ECDSA in hoofdstuk 8 worden ze bekend verondersteld.

Paragraaf één licht kort de keuze voor de programmeertaal C toe, paragraaf twee de keuze voor multilengte-softwarepakket LIP. In paragraaf drie volgt een korte beschrijving van de hashfunctie MD4. In de laatste paragraaf wordt het computerprogramma beschreven waarbij de belangrijkste stappen worden besproken. De programmacode is als bijlage opgenomen (bijlage 11.

1. Programmeertaal C

C is een derde generatie imperatieve programmeertaal. Ontworpen in 1978 door Kernighan en Ritchie, is de taal de afgelopen twintig jaar uitgegroeid tot een van de meest 'gesproken' talen. Wildgroei van toegevoegde functies, en vele, elkaar snel opvolgende versies maakten een standaard noodzakelijk. In 1988 is daarom ANSI-C, een gestandaardiseerde versie van C, ingevoerd.

In tegenstelling tot wat je wellicht zou denken is in de wetenschappelijke wereld het relatief oude (ANSI-)C de voertaal. Jongere, meer object-georiënteerde talen als C++ en Java worden slechts door een selecte groep gebruikt; de noodzaak over te stappen naar een nieuwe taal lijkt niet te bestaan. Om het programma voor een zo groot mogelijke groep 'verstaanbaar' te maken en omdat in de CWI-projectgroep *Computational Number Theory* veelal in C wordt geprogrammeerd, is voor deze taal gekozen.

2. Long Integer Programming

In beide cryptografische systemen moet gerekend worden met getallen van tenminste 160 bits. In C is het niet mogelijk met zulke grote getallen te rekenen; *long* is het grootst mogelijke variabeletype en kan 32 bits bevatten. Voor dit onderzoek was dus aanvullende software nodig die rekenen met zeer grote getallen mogelijk zou maken.

In beide programma's is gebruik gemaakt van het softwarepakket Long Integer Programming (LIP). Dit pakket is geschreven door Ajren K. Lenstra in 1991. Het is volledig in C en speciaal geschreven voor cryptografische applicaties. Er is een commerciële en een freeware versie; in dit onderzoek is de

freeware versie, freeLIP[1], gebruikt.

In LIP is een multilengte variabeletype *verylong* gedefinieerd. Dit type is intern gerepresenteerd als een willekeurig lange array van variabelen van het C-type *long*. De lengte van een variabele is dus dynamisch; de geheugenallocatie (en -teruggave) die eventueel nodig is bij bewerkingen, is binnen de functies geregeld. Omdat het een nieuw type is moeten alle standaardfuncties zoals 'inlezen', 'afdrukken', maar ook 'optellen' en 'aftrekken' opnieuw worden geïmplementeerd.

LIP is een ondersteunend pakket: het stelt de gebruiker in staat in zijn of haar C-programma's gebruik te maken van multilengte variabelen. De routines in LIP zijn dus basisfuncties – uitgebreid met functies voor cryptografische doeleinden – die werken met multilengte-variabelen. In tabel 5.1 is een aantal routines van LIP beschreven, om een indruk te geven van de mogelijkheden.

ziszero(a)	retourneert 1 als $a = 0$, anders 0.
zsmul(a, b, c)	c krijgt de waarde $a \cdot b$.
zhread(a)	a krijgt de waarde van het hexadecimale getal, ingelezen van het scherm.
zexpmod(a, b, c, d)	d krijgt de waarde a^b mod c.
zrandomb(a, b)	retourneert random getal b, $0 \leq b \leq a$.

Tabel 6.1: Functies uit multilengte-softwarepakket LIP.

3. Hashfunctie MD4

MD4 is in 1984 geschreven door Ronald Rivest, bij 'het grote publiek' bekend van zijn aandeel in het public-key cryptosysteem *R*SA. De letters MD zijn een afkorting van *Message Digest*, een andere naam voor de hashwaarde van een boodschap, '4' is het serienummer. Hoewel dus onderdeel van een familie is gekozen hier alleen MD4 te bespreken. De hashfunctie MD4 retourneert voor een boodschap m een 128 bits outputstring, de hashwaarde $h(m)$. MD4 volgt zeer strak het 'hashfunctie format' zoals dat in hoofdstuk 4 is besproken. Er wordt hier geen volledige beschrijving van MD4 (zie daarvoor bijlage 13).

In de keuze van de hashfunctie is afgeweken van de NIST-standaard DSS. De door NIST geëiste hashfunctie SHA-I, een opvolger van MD4, is in dit experiment niet gebruikt. MD4 was in tegenstelling tot SHA-I vrij beschikbaar en het doel van dit onderzoek, het vergelijken van twee simulatiesystemen, werd door een 'mindere' hashfunctie niet geschaad.

MD4 werd niet geaccepteerd, omdat de functie zijn beloften niet waar kon maken. De veiligheid van MD4 was vanuit de aanvaller gedefinieerd: deze zou tenminste 2^{64} operaties nodig hebben om twee boodschappen te vinden met dezelfde hashwaarde en zelfs 2^{128} operaties om bij van een gegeven hashwaarde een bijbehorende boodschap te vinden. Deze aantallen zijn ten hoogste 2^{20}.

4. Programmacode DSA

In het computerprogramma wordt de indeling van DSA, zoals beschreven in het vorige hoofdstuk, strikt gevolgd. Er zijn drie hoofdroutines: *sleutel*, *handtekening* en *verificatie*. Het hoofdprogramma doorloopt achtereenvolgens deze routines. In deze paragraaf wordt de programmacode globaal doorgenomen, zie voor de volledige code bijlage 11.

variabelen

Eerst wordt een aantal constanten gedefinieerd:

```
#define blp 1024       /*bitlengte p; NIST 768<=blp<=1024*/
#define blq 521        /*bitlengte q; NIST:160*/
#define aantaltests 40 /*aantal iteraties pseudopriemtest*/
```

[1] FreeLIP kan gratis van internet worden gedownload. Het www-adres staat in de literatuurlijst onder "Lenstra".

De variabele *aantaltests* is de veiligheidsparameter van Miller-Rabin, die bepaalt hoe vaak de interne testloop wordt doorlopen (zie hoofdstuk 5). NIST eist dat de kans dat een getal samengesteld is en 'priem' wordt genoemd kleiner is dan $(\frac{1}{2})^{80}$. De kans hierop na één iteratie van Miller-Rabin was $\frac{1}{4}$, dus 40 tests is voldoende. De bitlengtes van p en q worden – zo zal in hoofdstuk 9 blijken – tijdens het exeriment gevarieeerd.

Vervolgens wordt een aantal globale variabelen gedeclareerd, die dezelfde betekenis hebben als in de beschrijving van DSA in hoofdstuk 5.

```
verylong p=0,q=0,quot=0,alpha=0,a=0,y=0;  /*sleutel*/
verylong m=0;                             /*boodschap*/
verylong r=0,s=0;                         /*handtekening*/
verylong w=0,u1=0,u2=0,v=0;               /*verificatie*/
```

Deze variabelen zijn alle van het multilengte-type *verylong*. Vanwege de variabele grootte van verylongs is het noodzakelijk ze voor gebruik gelijk te stellen aan nul; in het programma wordt dit direct bij declaratie gedaan. Tot slot worden drie tijdmeting-variabelen gedeclareerd.

main

In het hoofdprogramma *main* wordt eerst gevraagd om een 'seed' voor de randomgenerator. Deze seedwaarde wordt in de rest van het programma gebruikt als startwaarde voor randomfuncties.

> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*
> John von Neumann – 1951.

In deel I van Knuth's *The Art of Computer Programming* [11] staat een prachtige inleiding over randomgeneratoren, met als belangrijkste observatie dat een goede randomgenerator alles behalve random in elkaar moet zitten. Hoewel een wiskundig interessant terrein, waar menig Don Quichotte nog altijd eeuwige roem probeert te behalen, zal er in deze paragraaf alleen gesproken worden over de randomgenerator in LIP.

LIP kent één interne random functie, waar de andere randomroutines van zijn afgeleid: `zrandom`. De functie `zrandom` vraagt als invoer een geheel getal g – en intern ook de seedwaarde – dat als bovengrens dient, en retourneert een randomwaarde tussen 0 en g. De seedwaarde wordt vervolgens vervangen door $s \cdot g \bmod p$ met $p = 2^{107} \Leftrightarrow 1$ priem en $g = 3^{121}$, voortbrenger van $(\mathbb{Z}/p\mathbb{Z})^*$. Dit wordt net zo lang herhaald totdat $s < p \Leftrightarrow (p \bmod g)$. De functie `zrandom` retourneert nu $s \bmod g$. Merk op dat de seed na een aanroep van `zrandom` is veranderd.
De oplettende lezer zal het niet zijn ontgaan dat deze random functie niet werkt voor een grens g die groter is dan p. De hier gebruikte functie `zrandomb` past intern de berekening aan als $g > p$.

sleutel

Zoals gezegd, is LIP speciaal geschreven voor cryptografische doeleinden. De commerciële versie bevat zelfs een DSA-simulatieprogramma, volledig opgebouwd uit LIP-routines. Omdat hier is gewerkt met FreeLIP moest een eigen versie van DSA worden geschreven, waarbij evenwel wel gebruik gemaakt kon worden van een aantal specifieke DSA-routines van LIP. De eerste hiervan wordt gebruikt in de functie *sleutel*: `zrandomqprime`. De aanroep van `zrandomqprime` ziet er als volgt uit:

```
zrandomqprime( blp, blq, aantaltests, &p, &q, zrandomb);
```

De naam verraadt het enigszins: deze functie retouneert twee (pseudo)priemen p (van lengte `blp`) en q (van lengte `blq`), met $q \mid (p \Leftrightarrow 1)$. Deze priemen zijn gegenereerd met behulp van de NIST-priemgenerator en getest met Miller-Rabin en veiligheidsparameter `aantaltests`. De functie `zrandomb` wordt gebruikt voor de randomwaarden (beide priemen worden random gegenereerd).

Vervolgens wordt in de routine *sleutel* de voortbrenger α van de ondergroep gekozen. Hier – en aan het eind van *sleutel* – wordt `zexpmod` aanroepen:

```
zexpmod(g, quot, p, &alpha).
```

Deze functie retourneert – in `alpha` – g^{quot} mod p. De berekening van een modulaire macht gebeurt volgens binair machtsverheffen. In hoofdstuk 8 over elliptische krommen-DSA wordt deze methode uitgebreid besproken.

De functie *sleutel* heeft als uitvoer de publieke en geheime sleutel van Alice:

```
SLEUTEL
publieke sleutel
p is 819732357255140239048384672080547687764809602355620 3\
2375042611500698420712193797244478299209047018385302652 28\
913555511479596518895689433790806802472040791
q is 81174395387823116609762481879757817245044387241 1
alpha is 413566898332803548449257151399385693337846428124\
8201202280935905939778657316944624450921665780884962449 61\
771439248815370294263339162704730412338323578523 0
y is 188242354928128011683640984736170554042545716945629 1\
316911946024060573879868049649909843331450137086581462070\
099509373846146727839009111462647762704290281

geheime sleutel
a is 692474738099084647510772103242099562048092799155
```

handtekening

In de functie *handtekening* wordt, voor een zekere boodschap m, het getallenpaar (r, s) (de handtekening) geretourneerd. Zoals in hoofdstuk 4 is verteld wordt de handtekening niet op de boodschap zelf, maar op de hashwaarde $h(m)$ geplaatst. Het hashen gebeurt met de hierboven beschreven functie MD4. De aanroep `md4( );` vraagt een boodschap in te voeren van het scherm en retourneert de hashwaarde in de variabele m. De boodschap zelf wordt in dit programma dus niet bewaard! Waarom dat allemaal zomaar kan wordt uitgelegd in hoofdstuk 9. De uitvoer van *handtekening* ziet er bijvoorbeeld zo uit:

```
HANDTEKENING
handtekening (r,s)
r is 660599113799696747088677099156912385539501327
s is 573664308107154822382076345843439960891935552007
geheime deel (niet langer nodig)
k is 568326528403222920396946547433382296486345853741
```

verificatie

In de functie *verificatie* wordt de gegenereerde handtekening geverifieerd. Het zal niemand verbazen dat het programma zijn eigen handtekening altijd accepteert. De verificatie wordt hier dan ook alleen gebruikt om de rekentijden te kunnen meten. Als eerste wordt gecontroleerd of r en s beide kleiner zijn dan q. Deze controle is nodig omdat het programma in het vervolg aanneemt dat beide waarden kleiner zijn en omdat de NIST-standaard dit eist. Als iemand een handtekening wil vervalsen zal hij of zij er voor zorgen voor dat de vervalsing tenminste uiterlijk op de echte lijkt, en dus r en s kleiner kiezen dan q. Vervolgens wordt v berekend volgens DSA en gecontroleerd dat $v = r$. De uitvoer van *verificatie* is dan:

```
VERIFICATIE
v 660599113799696747088677099156912385539501327
Handtekening geaccepteerd.
```

In het hoofdprogramma en in *sleutel* worden tijdmetingen gedaan. Deze worden besproken in hoofdstuk 9.

Hoofdstuk 7
Elliptische krommen

Elliptische krommen duiken links en rechts in de wiskunde op. Hun meest memorabele optreden is waarschijnlijk in het bewijs van de laatste stelling van Fermat door Wiles. In 1985 hebben Victor Miller en Neal Koblitz onafhankelijk van elkaar voorgesteld elliptische krommen te gebruiken voor cryptografische doeleinden. Wat elliptische krommen zo interessant maakt is dat ze – over eindige lichamen – een bijna onuitputtelijke bron van eindige, abelse groepen zijn, die bovendien een rijke algebraïsche structuur hebben. Ze lijken sterk op eindige lichamen, maar hebben meer 'keuzemogelijkheden'.

In dit hoofdstuk wordt een inleiding over elliptische krommen gegeven. De nadruk ligt hierbij op het begrijpelijk maken van de theorie en niet op volledige bewijsvoering. De behandeling laat zich het beste als volgt samenvatten: wel correct, niet volledig. Bovendien blijft de bespreking grotendeels beperkt tot de kromme die nodig is voor ECDSA, de elliptische kromme-variant van DSA, die in het volgende hoofdstuk zal worden besproken.

In de eerste paragraaf wordt een definitie gegeven van elliptische krommen. Daarna volgt een beschrijving van een elliptische kromme over het lichaam $\mathbb{R}$, de verzameling der reële getallen. In de derde paragraaf wordt de elliptische kromme over $\mathbb{F}_p$ ($E(\mathbb{F}_p)$) beschreven, die wordt gebruikt in de simulatie van ECDSA. Tot slot wordt het discrete logaritme-probleem in $E(\mathbb{F}_p)$ besproken.

1. Ellips?

Elliptische krommen hebben niet veel met ellipsen te maken. Hun naam is afgeleid van zogenaamde elliptische functies, de 'inverse' van de elliptische integraal. In de inleiding van deel II is de elliptische kromme een *algebraïsche structuur, die een eindige cyclische groep voortbrengt* genoemd. Deze definitie zal in deze paragraaf worden aangescherpt.

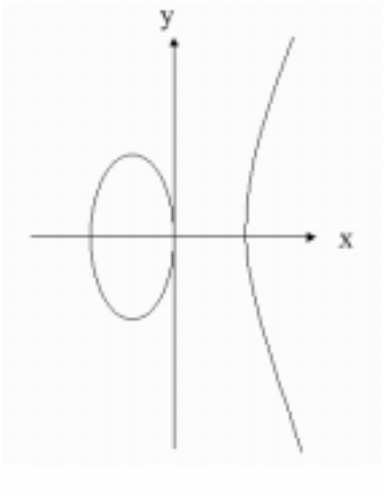
Een elliptische kromme bestaat niet op zichzelf, hij is gebaseerd op een lichaam K. De algemene vorm van een elliptische kromme wordt gegeven door de zogenaamde Weierstrassvergelijking:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \quad a_i \in K$$

De elliptische kromme (over K), $E(K)$, is nu de verzameling punten (x, y), $x, y \in K$, die voldoen aan deze vergelijking. De precieze vorm van de kromme wordt bepaald door het karakteristiek van het onderliggende lichaam (deze afhankelijkheid zal hier niet worden uitgelegd). Voor een elliptische kromme over een lichaam K met karakteristiek > 3 geldt de volgende, afgeslankte vergelijking:

$$y^2 = x^3 + ax + b, \quad a, b \in K.$$

In figuur 7.1 is de kromme over $\mathbb{R}$ te zien met $a = \Leftrightarrow 7$ en $b = 0$. De verzameling punten (x, y) die voldoen aan de vergelijking vormt een abelse groep onder optelling (zie paragraaf 6.2). Als eenheidselement geldt het zogenaamde 'point at infinity' O, dat verderop wordt besproken.



Figuur 7.1: Elliptische kromme $y^2 = x^3 \Leftrightarrow 7x$ over $\mathbb{R}$.

2. Elliptische krommen over $\mathbb{R}$

Aangezien de groepsbewerking 'optelling' voor de punten op een elliptische kromme meetkundig is gedefnieerd, zal eerst de elliptische kromme over $\mathbb{R}$ worden besproken.
De groepsbewerking van $E(\mathbb{R})$ laat zich als volgt samenvatten:

De som van de drie snijpunten van een lijn met de kromme is O.

Zij E een elliptische kromme over de reële getallen $\mathbb{R}$ en zij P en Q twee punten op E. Het tegengestelde van P en de som $P + Q$ zijn gedefinieerd volgens de volgende regels:

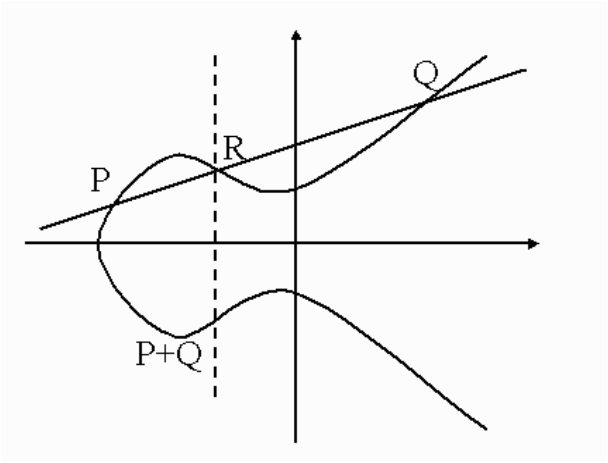
1. Als P het *point at infinity* O is, dan is $\Leftrightarrow P$ ook O en is $P + Q$ gelijk aan Q. Het point at infinity fungeert dus – zoals hierboven al even genoemd – als eenheidselement. In 2.–5. wordt aangenomen dat P en Q niet O zijn.

2. Het tegengestelde punt $\Leftrightarrow P$ van P is het punt op de kromme met dezelfde x-coördinaat, maar de tegengestelde y-coördinaat, dus $\Leftrightarrow(x, y) = (x, \Leftrightarrow y)$. Aan de vergelijking is te zien dat E symmetrisch is om de x-as en er dus op deze wijze voor ieder punt een tegengestelde is gedefinieerd.

3. Als P en Q verschillende x-coördinaten hebben snijdt de lijn l door P en Q de kromme in nog precies één punt R. De som $P + Q$ is nu het tegengestelde van dit punt R: $P + Q = \Leftrightarrow R$. Mocht de lijn l maar twee snijpunten hebben met de kromme, omdat l de raaklijn aan de kromme in P is, dan geldt $R = P$. (Zie figuur 7.2 voor de meetkundige constructie van $P + Q$.)

4. Als $P = \Leftrightarrow Q$ (dus gelijke x-, maar ongelijke y-coördinaat) dan is $P + Q = O$, vanwege regel 2.

5. De laatste mogelijkheid is dat $P = Q$. In dat geval is de lijn l door P en Q de raaklijn aan de kromme in het punt P. Het enig andere punt op de kromme waar l door gaat is dan R en $P + Q$ is weer $\Leftrightarrow R$. Als de raaklijn in P de kromme alleen in P snijdt, dus als P een buigpunt, dan is R gelijk aan P en de som $P + Q$ wederom gelijk aan $\Leftrightarrow R$.



Figuur 7.2: Optellen van de punten P en Q op een elliptische kromme.

Hoewel deze meetkundige beschrijving verhelderend is, moeten – om met deze groep te kunnen rekenen – formules worden afgeleid die de coördinaten van de som geven als functie van de coördinaten van de twee op te tellen punten. Hieronder zullen de coördinaten van $P+Q$ uitgedrukt worden in de coördinaten van P en Q. Verder wordt getoond waarom er nog een derde snijpunt is van de lijn door P en Q met de kromme. Laat (x_1, y_1) en (x_2, y_2) en (x_3, y_3) de coördinaten van respectievelijk P, Q en $P+Q$ zijn, dan willen we x_3 en y_3 uitdrukken in x_1, x_2, y_1, y_2.

In eerste instantie wordt gekeken naar de situatie als in regel 3. Zij $y = \alpha x + \beta$ de vergelijking van de lijn l door P en Q. Dan is $\alpha = (y_2 \Leftrightarrow y_1)/(x_2 \Leftrightarrow x_1)$ en $\beta = y_1 \Leftrightarrow \alpha x_1$. Een punt op l, dus een punt $(x, \alpha x + \beta)$, ligt op de kromme dan en slechts dan als $(\alpha x + \beta)^2 = x^3 + ax + b$. Twee oplossingen van deze vergelijking zijn al bekend, namelijk de x-coördinaten x_1 en x_2 van P en Q. Omdat de som van de wortels van een monisch polynoom gelijk is aan de coefficiënt van de op één na hoogste macht[1], is de derde wortel: $x_3 = \alpha^2 \Leftrightarrow x_1 \Leftrightarrow x_2$. Dit geeft een uitdrukking voor x_3, en dus voor $P + Q = (x_3, \Leftrightarrow(\alpha x_3 + \beta))$, uitgedrukt in x_1, x_2, y_1 en y_2:

$$x_3 = (\frac{y_2 \Leftrightarrow y_1}{x_2 \Leftrightarrow x_1})^2 \Leftrightarrow x_1 \Leftrightarrow x_2;$$

$$y_3 = \Leftrightarrow y_1 + (\frac{y_2 \Leftrightarrow y_1}{x_2 \Leftrightarrow x_1})(x_1 \Leftrightarrow x_3).$$

Als P gelijk is aan Q is de lijn l de raaklijn in het punt P, ofwel α is de afgeleide in het punt P. Differentiëren van $y^2 = x^3 + ax + b$ geeft $\alpha = (3x_1^2 + a)/2y_1$. Vervangen van α door deze nieuwe uitdrukking geeft voor de coördinaten van $P + Q$:

$$x_3 = (\frac{3x_1^2 + a}{2y_1})^2 \Leftrightarrow 2x_1;$$

$$y_3 = \Leftrightarrow y_1 + (\frac{3x_1^2 + a}{2y_1})(x_1 \Leftrightarrow x_3).$$

Voorbeeld Op de kromme $y^2 = x^3 \Leftrightarrow 36x$ liggen de punten $P = (\Leftrightarrow 3, 9)$ en $Q = (\Leftrightarrow 2, 8)$. Bereken $P+Q$. Substitutie van $x_1 = \Leftrightarrow 3, y_1 = 9, x_2 = \Leftrightarrow 2, y_2 = 8$ in de eerste vergelijking geeft $x_3 = (\Leftrightarrow 1/1)^2 + 3 + 2 = 6$. Substitutie in de tweede vergelijking geeft dan $y_3 = \Leftrightarrow 9 + \Leftrightarrow 1 \cdot (\Leftrightarrow 3 \Leftrightarrow 6) = 0$. $P + Q$ is dus $(6, 0)$. Voor $2P$ substitueer je $x_1 = \Leftrightarrow 3, y_1 = 9$ en $a = \Leftrightarrow 36$ in de vergelijking. Voor de x-coördinaat geeft dit

[1] Waarom dat zo is zal hier niet worden uitgelegd.

25/4, voor de y-coördinaat $\Leftrightarrow 35/8$.
Dat deze definities van $P + Q$ de verzameling punten op E tot een abelse groep maken zal hier niet worden bewezen (zie hiervoor bijvoorbeeld [14]).

Point at Infinity O
Dit mysterieuze punt is in de eerste paragraaf al even genoemd en in de definities is duidelijk geworden dat het de rol van eenheidselement vervult in de abelse groep voortgebracht door een elliptische kromme. Maar waar ligt dat punt?

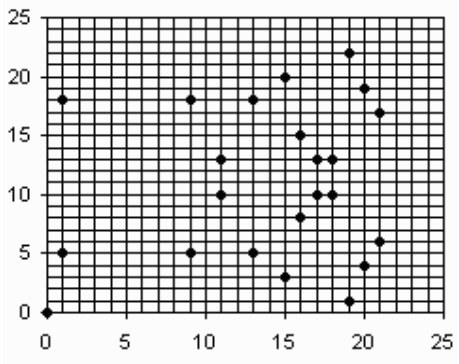
De som van twee tegengestelden is het *point at infinity*: $P + \Leftrightarrow P = O$. $\Leftrightarrow P$ is gedefinieerd als het punt met dezelfde x-coördinaat als P, maar tegengestelde y-coördinaat. De som van P en $\Leftrightarrow P$ is het tegengestelde van R, het derde snijpunt met de kromme. Maar de lijn door P en $\Leftrightarrow P$ loopt verticaal en snijdt de kromme nergens meer! Dit 'derde snijpunt' is nu gedefinieerd in $y = \infty$. Deze definitie is natuurlijk niet helemaal uit de lucht gegrepen. Kijk naar de punten $P(x_P, y_P)$ en $Q(x_P + \delta, y_Q)$. Zij R nu het derde snijpunt met de kromme voor $\delta \neq 0$ en laat $\delta \to 0$. Dan nadert x_Q tot x_P en y_Q tot $\Leftrightarrow y_P$, ofwel $Q \to \Leftrightarrow P$ en R nadert naar oneindig. Als nu $\delta = 0$ dan is l de verticale lijn door P en heeft ieder punt op deze lijn, in het bijzonder de som van P en $\Leftrightarrow P$, x-coördinaat x_P. Maar wat is er met de y-coördinaat van R gebeurd? Deze was op weg naar oneindig. R ligt dus op de lijn l in het oneindige, en $\Leftrightarrow R$ dus op l met $y = \Leftrightarrow\infty$. Dat ietwat merkwaardige punt is nu O. Wat is daar dan zo merkwaardig aan? De som van *alle* tegengestelde punten op de kromme is O. Het point at infinity is dus eigenlijk de verzameling punten met een x-coördinaat gelijk aan tenminste één ander punt op de kromme en y-coördinaat 'oneindig'. Al deze punten zijn topologisch equivalent. Een iets minder abstracte definitie kan worden gegeven als de kromme in het projectieve vlak wordt bekeken; O is dan namelijk de equivalentieklasse van triples (x, y, z) met $x = z = 0$ (voor een uitgebreide beschrijving zie bijvoorbeeld [14] of [3]). Belangrijk is te onthouden dat O fungeert als eenheidselement in de groep $E(R)$.

3. Elliptische krommen over F_p

In het experiment is een elliptische kromme gebruikt met coefficiënten uit $\mathbb{F}_p$:

$$y^2 = x^3 + ax + b, \quad a, b \in \mathbb{F}_p.$$

De groep voortgebracht door een elliptische kromme over $\mathbb{F}_p$, $E(\mathbb{F}_p)$, is de verzameling punten (x, y), $x, y \in \mathbb{F}_p$, die voldoet aan de vergelijking tezamen met het punt O. De groepsbewerking volgt dezelfde definities als in $\mathbb{R}$ met dien verstande dat alles modulo p wordt berekend.



Figuur 7.3: De 23 groepselementen van $\mathbb{F}_{23}$.

voorbeeld $E(\mathbb{F}_{23})$
De elliptische kromme over $\mathbb{F}_{23}$ bestaat uit de verzameling punten (x, y) die voldoet aan de volgende vergelijking:

$$y^2 = x^3 + x.$$

Nota bene: Hier is $a = 1$ en $b = 0$, maar iedere keus van $a, b \in F_p$ is toegestaan.
Het punt $(1, 5)$ is een element van $E(\mathbb{F}_{23})$ omdat geldt

$$\begin{aligned} y^2 \bmod p &= x^3 + x \bmod p \\ 5^2 \bmod 23 &= 1^3 + 1 \bmod 23 \\ 25 \bmod 23 &= 2 \bmod 23 \\ 2 &= 2 \end{aligned}$$

De 23 punten in $E(\mathbb{F}_{23})$ zijn (zie ook figuur 7.3)
$(0,0)$ $(1,5)$ $(1,18)$ $(9,5)$ $(9,18)$ $(11,10)$ $(11,13)$ $(13,5)$ $(13,18)$ $(15,3)$ $(15,20)$ $(16,8)$ $(16,15)$ $(17,10)$ $(17,13)$ $(18,10)$ $(18,13)$ $(19,1)$ $(19,22)$ $(20,4)$ $(20,19)$ $(21,6)$ $(21,17)$.
Nota bene: het is slechts toeval dat het aantal punten in $E(\mathbb{F}_{23})$ gelijk is aan p (zie hoofdstuk 8).

De groepsbewerking in $E(\mathbb{F}_{23})$ volgt de definities van $E(\mathbb{R})$, maar dan gerekend modulo p. De tegengestelde van een punt P is het punt $\Leftrightarrow P$ met gelijke x, maar tegengestelde y (modulo p). De tegengestelde van $(1, 5) \in E(\mathbb{F}_{23})$ is bijvoorbeeld $(1, \Leftrightarrow 5 \bmod p) = (1, 18)$. De coördinaten van de som van twee punten $P(x_1, y_1)$ en $Q(x_2, y_2)$, $P \neq Q$, wordt gegeven door

$$\begin{aligned} \alpha &= \left(\tfrac{y_2 - y_1}{x_2 - x_1}\right) \bmod p \\ &= (y_2 \Leftrightarrow y_1) \cdot (x_2 \Leftrightarrow x_1)^{-1} \bmod p \\ x_3 &= \alpha^2 \Leftrightarrow x_1 \Leftrightarrow x_2 \bmod p \\ y_3 &= \Leftrightarrow y_1 + \alpha(x_1 \Leftrightarrow x_3) \bmod p. \end{aligned}$$

Voor twee gelijke punten, ofwel een verdubbeling, wordt α vervangen door

$$\alpha = (3x_1^2 + a)/(2y_1) \bmod p.$$

Problemen ontstaan als de y-coördinaat van P gelijk is aan nul: het is dan onmogelijk de inverse modulo p te bepalen. De verdubbeling van P laat zich in dit geval als volgt uitrekenen:
Zij $P = (x_p, 0)$. De tegengestelde van P is dan $\Leftrightarrow P = (x_p, \Leftrightarrow y_p) = (x_p, 0) = P$. Maar dan geldt $2P = P + P = P + \Leftrightarrow P = O$.

Voorbeeld $E(\mathbb{F}_{23})$
Zij $P = (11, 10)$ en $Q = (21, 6)$, beide elementen van $E(\mathbb{F}_{23})$. De som $P + Q$ wordt dan als volgt berekend

$$\begin{aligned} \alpha &= (y_2 \Leftrightarrow y_1) \cdot (x_2 \Leftrightarrow x_1)^{-1} \bmod p \\ &= (6 \Leftrightarrow 10)(21 \Leftrightarrow 11)^{-1} \bmod 23 \\ &= \Leftrightarrow 4 \cdot 10^{-1} \bmod 23 \\ &= (\Leftrightarrow 4 \bmod 23 \cdot 10^{-1} \bmod 23) \bmod 23 \\ &= (19 \cdot 7) \bmod 23 \\ &= 18. \\ x_3 &= \alpha^2 \Leftrightarrow x_1 \Leftrightarrow x_2 \bmod p \\ &= 18^2 \Leftrightarrow 11 \Leftrightarrow 21 \bmod 23 \\ &= 324 \Leftrightarrow 32 \bmod 23 \\ &= 16. \\ y_3 &= \Leftrightarrow y_1 + \alpha(x_1 \Leftrightarrow x_3) \bmod p. \\ &= \Leftrightarrow 10 + 18(11 \Leftrightarrow 16) \bmod 23 \\ &= \Leftrightarrow 100 \bmod 23 \\ &= 15. \end{aligned}$$

De som van $(11, 10)$ en $(21, 6)$ is dus $(16, 15)$, inderdaad een punt in $E(\mathbb{F}_{23})$.

4. Discrete logaritme-probleem in $E(F_p)$

Het digitale handtekeningenalgoritme waar DSA in dit onderzoek mee wordt vergeleken, Elliptic Curve DSA (ECDSA), is ook gebaseerd op de discrete logaritme, in de groep $E(\mathbb{F}_p)$.

De discrete logaritme was de logaritmische functie in een eindige, cyclische groep. Ook in de groep voortgebracht door een elliptische kromme kan derhalve een discrete logaritme worden gedefinieerd. De beschrijving wijkt iets af van hoofdstuk 3 omdat daar de multiplicatieve notatie werd gehanteerd, terwijl de groep voortgebracht door een elliptische kromme additief is.

Definitie discrete logaritmeprobleem in $E(\mathbb{F}_p)$

Dit probleem wordt in de literatuur vaak aangeduid met ECDLP: *Elliptic Curve Discrete Logarithm Problem.*

Gegeven een eindige, abelse groep E voortgebracht door een elliptische kromme over het lichaam $\mathbb{F}_p$, en een basispunt P van $E(\mathbb{F}_p)$ met orde q, en een groepselement $Q \in E(\mathbb{F}_p)$. De discrete logaritme van Q bij het basispunt P is het natuurlijke getal x, $0 \leq x \leq q \Leftrightarrow 1$, zó dat $xP = Q$.

Voorbeeld $E(\mathbb{F}_{23})$

Wat is de discrete logaritme x van $Q(4,5)$ als $P(16,5)$ het basispunt is?

Een – naïeve – manier om x te vinden is net zolang P bij zichzelf optellen tot Q gevonden is, de eerdergenoemde *exhaustive search*. $P = (16,5) 2P = (20,20) 3P = (14,14) 4P = (19,20) 5P = (13,10) 6P = (7,3) 7P = (8,7) 8P = (12,17) 9P = (4,5)$. Omdat $9P = (4,5) = Q$ is 9 de discrete logaritme van Q voor basispunt P.

Hoofdstuk 8
Simulatie van Elliptic Curve DSA

Met de opkomst van het onderzoek naar elliptische krommen, is ook het aantal toepassingen van elliptische krommen in de cryptografie sterk toegenomen. Sinds 1999 bestaat er een gestandaardiseerde elliptische kromme variant van het DSA-algoritme: Elliptic Curve DSA (ECDSA). ECDSA is sinds begin dit jaar ook NIST-standaard.

In dit hoofdstuk wordt eerst kort het verschil tussen ECDSA en zijn voorganger DSA besproken. Daarna volgt in paragraaf twee een beschrijving van het ECDSA-algoritme en een bespreking van de belangrijkste stappen. In de derde paragraaf zal gekeken worden naar het zogenaamde ordeprobleem, een probleem dat zich voordoet bij het gebruik van ECDSA en andere op elliptische krommen gebaseerde systemen. In de vierde paragraaf wordt tot slot de programmacode van de simulatie van ECDSA besproken. Ook hier is niet de volledige code opgenomen; daarvoor wordt verwezen naar bijlage 12.

1. ECDSA en DSA

De opbouw van de DSA- en ECDSA-algoritmen is gelijk, ECDSA wijkt alleen af als de onderliggende groep $E(\mathbb{F}_p)$ dat vereist. Ze verschillen in hun groepselementen (punten in ECDSA en natuurlijke getallen in DSA) en in hun groepsbewerking (additief in ECDSA en multiplicatief in DSA). Bij de beschrijving van ECDSA in de literatuur worden voor bepaalde variabelen vaste letters gebruikt die afwijken van DSA. Na de beschrijving van ECDSA is een tabel opgenomen met de belangrijkste verschillen, inclusief namen van 'equivalente' parameters.

2. Beschrijving ECDSA-algoritme

ECDSA bestaat net als zijn voorganger uit drie hoofdroutines: *sleutel genereren*, *signeren* en *verifiëren*. Deze routines worden achtereenvolgens beschreven. De krommen die in ECDSA worden gebruikt zijn alle over $\mathbb{F}_p$. De vergelijking die wordt gebruikt staat daarmee vast: $y^2 = x^3 + ax + b$, met a, b in $\mathbb{F}_p$.

sleutel genereren

Zij E een elliptische kromme over $\mathbb{F}_p$ en zij P een punt van priemorde q in $E(\mathbb{F}_p)$; P is het basispunt. Alice kiest een random getal x, $0 < x < q \Leftrightarrow 1$, en berekent $Q = xP$.

Alice's publieke sleutel is nu Q, x haar geheime sleutel. Nota bene: de elliptische kromme-coëfficiënten en het basispunt P zijn ook openbaar. Deze horen echter niet specifiek bij deze boodschap en worden daarom vaak weggelaten.

signeren
Om een handtekening te plaatsen op haar te versturen boodschap doet Alice het volgende.

1. Ze kiest een random integer k, $1 < k < q \Leftrightarrow 1$.
2. Ze berekent $kP = (x_1, y_1)$ en $r = x_1 \bmod q$ (waar q de orde van het basispunt P is). Als $r = 0$ kiest ze een nieuwe k, omdat bij $r = 0$ de geheime sleutel x uit s wegvalt (zie berekening van s bij (punt 4.)).
3. Ze berekent $k^{-1} \bmod q$.
4. Ze berekent tot slot $s = k^{-1}\{h(m) + xr\} \bmod q$, waar $h(m)$ de hashwaarde van de boodschap m is. Ook hier geldt dat wanneer $s = 0$ opnieuw moet worden begonnen, dus met een nieuwe k.

De handtekening van Alice voor de boodschap m is het getallenpaar (r, s).

verifiëren
Om de handtekening (r, s) van Alice op de boodschap m te verifiëren doet Bob het volgende.

1. Bob haalt Alice's publieke sleutel Q op, evenals de openbare parameters p, q en de vergelijking van de kromme (liever: de coefficiënten a en b en het basispunt P).
2. Hij kijkt of r en s beide kleiner zijn dan q, zo niet dan verwerpt hij direct de handtekening.
3. Hij berekent $w = s^{-1} \bmod q$ en $h(m)$.
4. Hij berekent $u_1 = h(m)w \bmod q$ en $u_2 = rw \bmod q$.
5. Dan berekent hij $u_1P + u_2Q = (x_2, y_2)$ en $v = x_2 \bmod q$.

Bob accepteert de handtekening ook hier dan en slechts dan als $v = r$.

In tabel 7.1 staat een opsomming van de belangrijkste verschillen tussen DSA en ECDSA, specifiek voor dit onderzoek. Er wordt bijvoorbeeld uitgegaan van een kromme over $\mathbb{F}_p$ en niet een kromme van de algemene Weierstrassvorm.

	DSA	**ECDSA**
Groep	F_p	$E(\mathbb{F}_p)$
Groepselementen	natuurlijke getallen	punten (x, y) op E samen met O
groepsbewerking	vermenigvuldiging modulo p	optellen van punten
DLP	Gegeven $\alpha \in F_p$ en $y = \alpha^a \bmod p$, bepaal x.	Gegeven $P \in E(\mathbb{F}_p)$ en $Q = xP$, bepaal x.
notatie	p	orde $E(\mathbb{F}_p)$
	q	orde q van basispunt P
	a	geheime sleutel x

Tabel 8.1: verschil DSA – ECDSA.

3. Ordeprobleem in ECDSA
Het basispunt P, dat een priemorde q moet hebben, brengt de ondergroep voort waarin vervolgens wordt gerekend, net als in DSA. Hoe vind je zo'n punt P? Er wordt in ECDSA geen algoritme wordt gegeven om P te vinden. Dat is niet voor niks: het vinden van zo'n punt P is alles behalve eenvoudig.

De orde van een punt is een deler van de groepsorde, het aantal punten in $E(\mathbb{F}_p)$. De groepsorde is echter niet a priori bekend. Er is wel een belangrijke stelling die een afbakening geeft van het aantal punten.

stelling van Hasse
Het aantal punten N op een elliptische kromme E over F_q ligt in het interval

$$q + 1 \Leftrightarrow 2\sqrt{q} \leq N \leq q + 1 + 2\sqrt{q}.$$

De orde van de groep verschilt dus ten hoogste $2\sqrt{q}$ van de orde van de onderliggende groep $\mathbb{F}_q$, en is dus ongeveer even groot. De maximale (absolute) afwijking van N t.o.v. $q + 1$, $2\sqrt{q}$, wordt in de literatuur vaak aangeduid met t.

Het bekendste algoritme om de groepsorde uit te rekenen is van Schoof [19]. Schoof's algoritme is ingewikkeld en lastig te implementeren. In dit experiment is daarom gebruik gemaakt van reeds bestaande krommen, in plaats van dit – of een ander – algoritme te implementeren. In toepassingen van elliptische kromme-systemen wordt dit vaak gedaan. Er zijn dan ook krommen 'in omloop', bijvoorbeeld in boeken [3] en op internet (Certicom), waarvan er in het simulatieprogramma een aantal is gebruikt.

4. Programmacode ECDSA

Net als DSA is ook dit programma opgebouwd uit de drie routines *sleutel*, *handtekening* en *verificatie*. Het belangrijkste verschil met DSA is dat de groepselementen punten zijn, in plaats van getallen. In dit programma is een nieuwe variabeletype gedefinieerd met behulp van het C-type *struct*. Een struct is een variabele met meerdere velden, mogelijk van verschillend type, vergelijkbaar met een `record` in Pascal. De struct `punt` ziet er als volgt uit

```
struct punt/*punt kromme; x- en y-coordinaat in Fp*/
{
    verylong x;  /*x coordinaat*/
    verylong y;  /*y coordinaat*/
    int oneindig;/*boolean:punt==point at infinity --> :=1*/
};
```

Een `punt` bestaat uit twee coördinaten `x` en `y` en een 'label' `oneindig`. Dit label is een booleanvariabele: 1 als het punt point at infinity O is, en anders 0. Als `oneindig`=1 dan worden de coördinaten in `x` en `y` genegeerd. Structs worden in het programma aangegeven met hoofdletters, verylongs nog steeds met kleine letters.

Omdat de opbouw van het programma het ECDSA-algoritme strikt volgt en bovendien veel lijkt op de eerder beschreven DSA-code worden hier alleen nog dec volgende twee onderdelen van ECDSA behandeld: optellen en vermenigvuldigen van punten op een kromme en 'handmatig moduleren'.

optellen en vermenigvuldigen
Op verschillende punten in ECDSA worden veelvouden van punten uitgerekend, bijvoorbeeld bij het generen van de publieke sleutel Q, die een veelvoud x is van basispunt P. Ook wordt in ECDSA éénmaal opgeteld, in de functie *verificatie* bij de berekening van $u_1P + u_2Q$. Deze twee functies heten in het programma – toegegeven weinig verrassend – `optellen` en `vermenigvuldig`.

In de functie `optellen` is de definitie van de coördinaten van het derde punt gevolgd, zoals beschreven bij de theorie over elliptische krommen. Een aanroep van `optellen` ziet er als volgt uit:

```
optellen(A,B,&C);
```

De functie retourneert voor twee gegeven punten A en B de som $C = A + B$. Zoals in hoofdstuk 7 is beschreven wordt 'onderscheid' gemaakt tussen de volgende gevallen:

1. A, B of beide zijn O
2. $A = \Leftrightarrow B$

3. $A = B$
4. alle andere gevallen

Dit onderscheid wordt in de functie `optellen` gemaakt door een reeks if-statements. In de afzonderlijke if-blokken volgt de definitie zoals die bij de bespreking van elliptische krommen is gegeven.

De functie `vermenigvuldig` leunt voor een belangrijk deel op de functie `optellen`. Een aanroep van `vermenigvuldig` ziet er als volgt uit

```
vermenigvuldig (x,A,&B);
```

De functie retourneert in B het veelvoud x van A: $B = xA$. Deze berekening moet – voor de grote getallen waar hier mee wordt gerekend – natuurlijk zo efficiënt mogelijk worden geïmplementeerd. De eenvoudigste methode om dit 'slim' te doen is met behulp van de binaire representatie van x. Om xA uit te rekenen wordt eerst de binaire representatie van x bepaald. Vervolgens wordt deze van links naar rechts doorlopen, waarbij bij ieder bit j het volgende wordt gedaan:

1. $som = 2 \cdot som$.
2. Als het j-de bit gelijk is aan 1: $som = som + A$.

Het eerste bit kan worden overgeslagen omdat dat altijd gelijk is aan 1. De startwaarde van som is daarom A. Het aantal stappen om xA uit te rekenen is dus gelijk aan het aantal bits van x, dus van de orde $\log x$.

voorbeeld veelvoudberekening
Voor de berekening van $9 \cdot 3$ wordt dus eerst de binaire representatie van 9 bepaald: 1001. Vervolgens wordt deze string van links naar rechts doorlopen.

startwaarde som=3
stap 1. $som = 2 \cdot som = 6$
stap 2. 2-de bit is $0 \rightarrow$ volgende bit; som blijft 6.
stap 1. $som = 2 \cdot som = 12$.
stap 2. 3-de bit is $0 \rightarrow$ volgende bit; som blijft 12.
stap 1. $som = 2 \cdot som = 24$.
stap 2. 4-de bit is $1 \rightarrow som = som + 3 = 27$.

De hele bitstring van 9 is nu doorlopen, dus $9 \cdot 3 = 27$.
Op dergelijke wijze is de functie `vermenigvuldig` opgebouwd. Bij iedere aanroep van de for-loop wordt dus tenminste eenmaal de functie `optellen` aangeroepen.

moduleren
In beide programma's, maar in het bijzonder in ECDSA, worden veel groepsbewerkingen uitgevoerd in de onderliggende groep $\mathbb{F}_p$. Deze modulaire berekeningen zijn duur; ze kosten vergeleken met de overige bewerkingen veel rekentijd. Geprobeerd is daarom modulair rekenen waar mogelijk te vervangen door bij te houden in welk interval het resultaat zich ten hoogste bevindt. Een voorbeeld hiervan is te vinden in de functie `optellen`.

```
if(zcompare(D->x,0)==-1)
{
    if(zcompare(D->x,minp)==-1)/*(D.x<(-p)-->D.x:=D.x+p*/
        zadd(D->x,p,&(D->x));
    zadd(D->x,p,&(D->x));
}
```

Hier is bekend dat `D->x` kleiner kan zijn dan nul, maar groter is dan $\Leftrightarrow 2p$. Dit zogenaamde handmatig moduleren is natuurlijk alleen rendabel als het interval niet te groot is, dus bijvoorbeeld na een optelling.

bestaande krommen
Omdat gebruik is gemaakt van bestaande krommen, bestaat de functie `kromme` uit het inlezen van waarden uit de file *getallen.c* (bijlage 14). In deze file zijn zeven krommen opgeslagen met de volgende bitlengtes voor modulus p: 160, 192, 224, 230, 256, 384 en 521. Van iedere kromme zijn de volgende gegevens beschikbaar:

modulus p
krommecoefficiënt a
krommecoefficiënt b
orde q van het punt P
x-coördinaat van basispunt P
y-coördinaat van basispunt P

voorbeeld kromme-160 uit file getallen.c

```
1461501637330902918203684832716283019655932542983
10
134363276215009249970163743897076481852807556507 8
1461501637330902918203683518218126812711137002561
916268978016867856409840773281569781084143028751
672457782153227759280123736012922325822259884444
```

De krommen 160, 192 en 230 zijn afkomstig uit [3], de andere uit een artikel van Johnson en Menezes, waarin ECDSA wordt beschreven [9]. De krommen moeten – helaas handmatig – worden geselecteerd met behulp van

```
#define nummerkromme 7 /*Welke kromme wordt ingelezen?*/
```

Bovendien moet de bitlengte van p, `blp`, nodig voor de binaire representatie in `vermenigvuldig`, handmatig worden aangepast.

Zij tot slot opgemerkt dat er een globale booleanvariabele `genereer` gedefinieerd is die aangeeft of het programma de kromme zelf moet genereren of dat een kromme wordt ingelezen. De optie 'zelf genereren' werkt echter niet omdat het bepalen van de orde van P nog niet is geïmplementeerd. De variabele `genereer` is daarom 0.

Hoofdstuk 9
Experimentele Vergelijking DSA–ECDSA

En dan eindelijk waar het allemaal voor gedaan is: de experimentele vergelijking van beide systemen. Welke is de beste? Digitale handtekeningen worden vaak op grote schaal gebruikt, denk aan transacties van banken of op internet. Snelheid van een signeeralgoritme lijkt – na veiligheid – dan ook de belangrijkste eigenschap. Daarom zullen de systemen worden vergeleken bij *gelijke veiligheid*; dan kun je heel duidelijk iets zeggen over het verschil in snelheid, en dus welk systeem het beste is.

In dit hoofdstuk zal eerst worden besproken wat 'gelijke veiligheid' precies inhoudt. In de tweede paragraaf wordt het artikel van Lenstra en Verheul [17] geïntroduceerd, dat in de opzet van het experiment als leidraad heeft gediend. In paragraaf drie wordt dan het experiment beschreven en in de laatste paragraaf worden de resultaten gepresenteerd.

1. Gelijke Veiligheid

De veiligheid van een cryptosysteem wordt bepaald door de mogelijkheid het te kraken: een makkelijk te kraken systeem is onveilig. Maar hoe weet je of een syteem makkelijk of moeilijk te kraken is?

Je kunt het natuurlijk gewoon proberen en kijken hoe moeilijk het is. In de cryptografische wereld is dat ook precies hoe de veiligheid van systemen wordt gedefinieerd: vanuit de aanvaller. Hoe makkelijk is het voor een aanvaller om het systeem te kraken? Welke methoden staan hem of haar ter beschikking?[1] Vooral de resultaten van RSA-challenges en sinds een aantal jaren ook EC-challenges, uitgeschreven door Certicom[2], zijn een goede indicatie hoe moeilijk het is deze systemen te kraken.

De twee systemen die in dit onderzoek worden vergeleken zijn beide gebaseerd op de discrete logaritme en het gaat hier dan ook om de volgende vraag:

Hoe goed is een aanvaller in staat het discrete logaritmeprobleem op te lossen?

Het belangrijkste verschil tussen ECDSA en DSA is dat – voor een geschikt gekozen kromme – $E(\mathbb{F}_p)$ alleen exponentiële kraakalgoritmen kent, terwijl voor DLP in F_p^* diverse sub-exponentiële algoritmen bekend zijn. Hier komt de scepsis die in de inleiding van deel II werd genoemd om de hoek kijken: de reden dat er slechts exponentiële algoritmen voor elliptische krommen bekend zijn, is – volgens de sceptici – dat er nog onvoldoende naar het probleem is gekeken.

[1] In de cryptografische theorie wordt bij analyse van systemen aangenomen dat het algoritme bij de aanvaller bekend is. De aanvaller weet dus welk probleem hij of zij op moet lossen.

[2] Een bedrijf dat toepassingen van elliptische krommen verkoopt ... met overigens een fabelachtig goede webpagina over cryptografie in het algemeen en elliptische krommen in het bijzonder: *www.certicom.com* .

Een tweede vraag is: *hoe* moeilijk moet het voor een aanvaller zijn? Het moge duidelijk zijn dat de moeilijk van het kraken van DSA – behalve van de beschikbare kraakalgoritmen – afhankelijk is van de gebruikte bitlengte voor p en q. De grootte van parameters p en q in DSA en p in ECDSA wordt de sleutellengte genoemd. Deze sleutellengte dient in de theorie over veiligheid als meetlint.

Om tot een experimenteel vergelijk te komen moet de volgende vraag beantwoord worden: Welke sleutellengte in DSA correspondeert met een gegeven sleutellengte in ECDSA? Ofwel, hoe vertaal je gelijke veiligheid van DSA en ECDSA in sleutellengtes? In dit experiment is hiervoor het artikel van Lenstra en Verheul gebruikt.

2. Artikel Lenstra & Verheul

In november 1999 is een artikel verschenen van de hand van Arjen Lenstra en Eric Verheul:*Selecting Cryptographic Key Sizes* [17].

> *Key size recommendations are scattered throughout the cryptographic literature or may, for a particular cryptosystem, be found in vendor documentation. Unfortunately it is often hard to tell on what premises (other than marketability) the recommendations are based. As far as we know this article is the first uniform, clearly defined, and properly documented treatment of this subject for the most important generally accepted cryptosystems.*
> Lenstra&Verheul – Selecting Cryptographic Key Sizes [17].

Inderdaad, de meeste artikelen over geschikte sleutellengtes komen van commerciële cryptografiebedrijven zoals Certicom en RSA. Wetenschappelijke (niet-commerciële) artikelen gaan vaak alleen in op de veiligheid of vereiste sleutellengte van één systeem. Omdat dit artikel enig in zijn soort is, en omdat Lenstra en Verheul beiden gerenommeerde onderzoekers zijn, is besloten hun artikel als leidraad te nemen. Zij direct opgemerkt dat het artikel niet zonder tegenstanders is; vooraanstaande cryptografen zoals Silvermann [3] hebben de conclusies van Lenstra-Verheul aangevallen.

Model Lenstra & Verheul

In *Selecting Cryptographic Key Sizes* wordt een beoordelingsmodel opgebouwd. De auteurs willen aanbevelingen kunnen doen over de sleutellengtes die in de toekomst voor diverse cryptosystemen nodig zijn om de gewenste mate van veiligheid te kunnen garanderen. Cryptosystemen worden verdeeld in vier groepen: symmetrische (DES), klassiek asymmetrische (RSA), subgroep discrete logaritme (DSA) en elliptische krommen-cryptosystemen. De keuze voor sleutellengtes hangt volgens Lenstra en Verheul af van de volgende vier factoren:

1. Levensduur: de verwachte periode dat de informatie beveiligd moet blijven.
2. Veiligheidsmarge: een acceptabele moeilijkheidsgraad van een succesvolle aanval.
3. Rekenomgeving: de verwachte verandering in hoeveelheid rekenapparatuur die krakers ter beschikking zal staan.
4. Crypto-analyse: de verwachte ontwikkeling van cryptografisch onderzoek.

Dat het model vooral als gebruikershandleiding moet dienen blijkt uit de presentatie van 'defaultwaarden'. Lenstra en Verheul bieden voor alle factoren die zij van belang achten defaultwaarden aan waarop hun advies gebaseerd is, maar stellen door hun openbare afweging de gebruiker in staat zijn of haar sleutellengtes naar eigen inzicht te berekenen, door één of meerdere parameters te wijzigen. Er wordt bijvoorbeeld een definitie van de mate van ontwikkeling van rekenkracht (chipcapaciteit) gegeven. Hierbij wordt de zogenaamde wet van Moore gebruikt die stelt dat de dichtheid van componenten per *integrated circuit* iedere 18 maanden verdubbelt. Deze 18 is de defaultwaarde voor m, het aantal maanden dat het gemiddeld duurt tot de processorsnelheid en geheugencapaciteit is verdubbeld. De gebruiker staat het echter vrij bijvoorbeeld $m = 9$ te kiezen.

[3] Het betreffende artikel van Silverman, *A Cost-based Security Analysis of Symmetric and Asymmetric KeyLentghs*, is te vinden op de RSA-website www.rsa.com .

De belangrijkste factor die wordt geïntroduceerd is de veiligheid van het meest gebruikte symmetrische cryptosysteem DES. De parameter s geeft aan in welk jaar DES – door de gebruiker! – nog veilig werd geacht; defaultwaarde van s is 1982. De reden dat Lenstra en Verheul deze definitie hanteren is omdat veel mensen in de cryptografische wereld bekend zijn met DES, en derhalve vast een eigen mening hebben over de veiligheid van dat systeem.

Het volledige model wordt hier niet besproken. Hieronder worden alle definities en bijbehorende defaultwaarden opgesomd, in de hoop zo een indruk te geven van de afwegingen van Lenstra en Verheul.

s eerder besproken veiligheidsmarge – het jaar tot wanneer de gebruiker DES veilig achtte; defaultwaarde: 1982.

m aantal maanden dat het gemiddeld duurt voordat een verdubbeling van de processorsnelheid en geheugencapaciteit optreedt; defaultwaarde: 18.

t binaire variabele, die aangeeft hoe m geïnterpreteerd moet worden:
$t = 1$ betekent dat m het aantal maanden is dat het gemiddeld duurt voordat de hoeveelheid rekenkracht en *random access memory* (RAM) die voor één dollar kan worden gekocht is verdubbeld,
$t = 0$ betekent dat de hoeveelheid rekenkracht en RAM – die de aanvaller ter beschikking staan – verdubbelt, ongeacht de kosten; defaultwaarde: 1.

b aantal jaren dat het gemiddeld duurt voordat het budget, dat de aanvaller ter beschikking staat, is verdubbeld; defaultwaarde: 10.

r aantal maanden dat het gemiddeld duurt voordat cryptoanalytische ontwikkelingen van klassieke asymetrische systemen tweemaal zo effectief zijn, ofwel verwacht wordt dat r maanden van nu een aanval op een klassiek asymmetrisch cryptosysteem half zoveel rekentijd kost; defaultwaarde: 18 (gebaseerd op ontwikkelingen in afgelopen 20 jaar).

c aantal maanden dat het gemiddeld duurt totdat crypto-analytische ontwikkelingen van elliptisch krommesytemen in effectiviteit verdubbelen; defaultwaarde: 0 (Dit betekent dat Lenstra en Verheul geen vooruitgang verwachten, voor geschikt gekozen krommen).

P prijs in dollars van een *'stripped down'* PC met tenminste 64 MB RAM. Met een *stripped down* PC wordt een 450MHz pentium II processor bedoeld, met een moederbord en communicatiehardware; defaultwaarde : 100.

Tabel Lenstra & Verheul
Lenstra en Verheul geven aan het eind van hun artikel een tabel met ondergrenzen van sleutellengtes voor de verschillende cryptosyteemgroepen, waarbij elliptische krommen bovendien zijn uitgesplitst naar 'wel' en 'geen vooruitgang' (in cryptografisch onderzoek). De tabel is hier niet volledig opgenomen. Bij de bespreking van het experiment is een deel van de tabel gegeven.

Een rij in de tabel hoort bij een bepaald jaar. De betekenis van de bitlengte bij een bepaald cryptosysteem in een bepaald jaar is: Als je informatie wilt beschermen met dit cryptosysteem en tot dit jaar, dan moet je je sleutel minstens deze bitlengte geven.

3. Experiment

Op basis van de tabel van Lenstra en Verheul is het mogelijk invulling te geven aan het begrip gelijke veiligheid. De bitlengtes voor de verschillende systemen in een rij (dus bij een bepaald jaar) betekenen gelijke veiligheid voor alle systemen. In de rij bij jaartal 2035 bijvoorbeeld, is af te lezen dat een bitlengte van 184 in een elliptische krommesysteem dezelfde veiligheid biedt als een sleutellengte van 230 in een subgroep discrete logaritmesysteem.

In dit experiment worden DSA en ECDSA vergeleken. Van beide systemen is een computersimulatie gemaakt, zoals beschreven in hoofdstukken 6 en 8. Deze systemen zijn voor verschillende waarden van de parameters (p en q in DSA en p in ECDSA) gedraaid, waarbij de keus is gebaseerd op de tabel van Lenstra en Verheul.

Omdat gewerkt is met bestaande krommen is de tabel van Lenstra en Verheul pragmatisch gebruikt: de bitlengtes van de zeven beschikbare krommen zijn opgezocht in de twee EC-kolommen, elliptische kromme met en zonder crypto-analytische vooruitgang. Voor waarden die niet in de tabel stonden – Lenstra/Verheul loopt tot 2051 – is doorgerekend. Voor elk van de waarden is vervolgens de bijbehorende rij uit de tabel van Lenstra en Verheul genomen, met als resultaat tabel 8.1 .

Jaar	EC zonder	EC met progressie	Subgroep progressie	Klassiek
2010	**160**	146	138	1369
2019	185	**160**	150	1825
2022	**192**	164	154	1995
2033	**224**	181	169	2698
2035	**230**	184	172	2840
2041	247	**192**	180	3292
2045	**256**	198	186	3616
2062	307	**224**	210	–
2067	320	**230**	215	–
2084	376	**256**	240	–
2088	**384**	262	245	–
3029	**521**	332	311	–
3061	625	**384**	360	–
4048	893	**521**	489	–

Tabel 9.1: Ondergrenzen sleutellengtes cryptosystemen–Lenstra en Verheul.

Nota bene: de bijbehorende jaartallen zijn ook opgenomen om een indruk te geven van de veiligheid ten opzichte van de andere waarden, dit jaartal is *niet* van belang voor het experiment. Hetzelfde geldt voor de kolom *Klassiek*, waarin de bitlengtes voor klassieke asymmetrische systemen zijn gegeven.

De in de tabel gegeven bitlengtes voor DSA zijn lengtes voor q, de orde van de ondergroep. Voor de modulus p van de hoofdgroep adviseren Lenstra en Verheul de equivalente bitlengte van klassieke cryptosystemen. Dit is een geaccepteerde keus voor p, behalve dat juist de aanbevelingen van Lenstra en Verheul voor klassieke asymmetrische systemen ter discussie staan. Gekozen is daarom DSA te simuleren voor de volgende drie bitlengtes: 768, 1024 en 1536. De eerste twee zijn gebaseerd op de NIST-bitlengte bij DSA; geeïst wordt tenminste 768 en maximaal 1024 te gebruiken. Omdat dit in vergelijking met Lenstra en Verheul kort is, wordt bovendien naar 1536 gekeken. Om een indruk te geven van de invloed die de modulus p heeft op de snelheid van DSA is het programma voor vaste bitlengte $q = 138$ en verschillende moduli p gedraaid.

Voor beide systemen is gemeten hoeveel tijd ze nodig hebben voor de drie hoofdroutines: sleutel genereren, handtekening plaatsen en verifiëren. Ter voorbereiding van het experiment is een aantal extra meetpunten gekozen om te kunnen zien wat – binnen een routine – veel tijd kost om zo tijdens het experiment beter te kunnen vergelijken. Op één punt is uiteindelijk afgeweken van het routine-meten: in de eerste routine *sleutel genereren*.

Zoals bij de bespreking van het programma is verteld wordt in de ECDSA-routine *sleutel* gebruik gemaakt van reeds bestaande krommen met een gegeven basispunt P. De routine *sleutel* bestaat daarmee alleen nog uit het inlezen van deze krommen uit de file, iets dat verwaarloosbaar weinig tijd

kost, en het berekenen van de publieke sleutel Q. Voor een eerlijke vergelijking is daarom in DSA ook de sleutelgeneratie en de berekening van de publieke sleutel opgesplitst en bij de vergelijking alleen gekeken naar de berekening van de publieke sleutel. Een vergelijking van de sleutelgeneratie in beide systemen zou erg interessant zijn, maar valt buiten dit experiment vanwege de eerdergenoemde moeilijkheid van het berekenen van de orde van het basispunt P.

De hashfunctie MD4 doet zijn naam eer aan: de aanroep `md4();` kost verwaarloosbaar weinig tijd, minder dan 1/1000 seconde. Ook de lengte van de boodschap – mits natuurlijk niet heel lang – beïnvloedt de snelheid niet merkbaar. In het experiment is iedere keer een boodschap van een of enkele regels ingelezen van het scherm en mag worden aangenomen dat `md4()` geen rol speelt in de verschillen tussen beide systemen.

De rekentijden zijn gemeten met behulp van de C-functie `clock`. Deze functie gedraagt zich als een stopwatch: vanaf de aanroep houdt zij bij hoeveel processortijd het programma vraagt, gemeten in MHz. Voor en na een functie of routine zijn meetpunten aangebracht. Het verschil tussen deze twee is de rekentijd die voor deze functie is genoteeerd. De rekentijden zijn gegeven in seconden, door de functie `clock` te corrigeren met behulp van de functie `clocks_per_second`.

De experimenten zijn in eerste instantie vijfmaal herhaald, wat kan omdat een seedwaarde wordt meegegeven. Omdat duidelijk werd dat de waarden van iedere run nauwelijks verschilden is dit teruggebracht tot driemaal. De boodschap is iedere keer van ongeveer gelijke grootte.

4. Resultaten

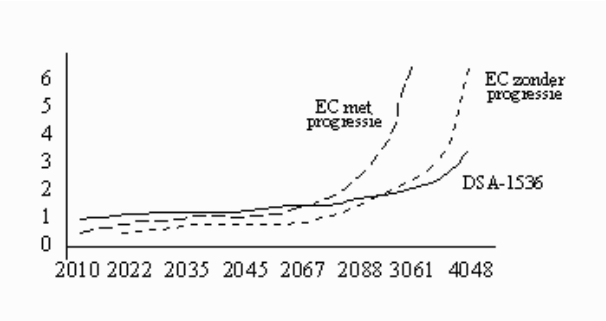
Het belangrijkste resultaat van het experiment is de vergelijking van de totale rekentijden van ECDSA en DSA. De resultaten staan in Tabel 8.2.

Jaar	EC-met progressie	EC-zonder progressie	DSA-768	DSA-1024	DSA-1536
2010	0,498	–	0,25	0,403	0,86
2019	–	0,498	0,2667	0,454	0,93
2022	0,694	–	0,2833	0,4597	0,96
2033	0,934	–	0,3027	0,5063	1,06
2035	0,984	–	0,3037	0,524	1,07
2041	–	0,694	0,3133	0,5403	1,09
2045	1,24	–	0,3363	0,5593	1,2
2062	–	0,934	0,365	0,617	1,34
2067	–	0,984	0,39	0,6403	1,347
2084	–	1,24	0,43	0,7137	1,54
2088	3,036	–	0,43	0,7203	1,54
3029	6,296	–	0,555	0,9267	1,95
3061	–	3,036	0,64	1,0697	2,27
4048	–	6,296	0,865	1,4733	3,08

Tabel 9.2: Rekentijden in seconden DSA en ECDSA bij gelijke veiligheid.

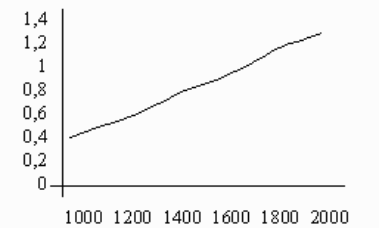
In de tabel is te zien dat DSA-768 en DSA-1024 vanaf de kleinste veiligheid sneller zijn dan ECDSA. Voor DSA-1536 ligt dat omslagpunt bij de veiligheid gerepresenteerd door 2045; voor grotere veiligheid is DSA sneller.

Het lijkt erop dat ECDSA slechter reageert op toenemende bitlengtes (zie figuur 8.1). De rekentijden van DSA stijgen lineair, terwijl ECDSA kwadratisch lijkt toe te nemen. De rekentijden van EC (zonder progressie) worden van 2088 naar 3029 bijna verdubbeld, DSA stijgt voor elke modulus met ongeveer 25%.



Figuur 9.1: De rekentijden van ECDSA en DSA-1536.

Modulus
Zoals in de vorige paragraaf is aangekondigd is ook gekeken naar de rekentijden van DSA-138 voor verschillende moduli. Zo kan gekeken worden naar de invloed van de modulus p van de hoofdgroep $\mathbb{F}_p$ op de rekentijd. De resultaten staan in figuur 8.2 . Het is duidelijk dat de modulus de rekentijd beïnvloedt; de toename lijkt bovendien lineair.



Figuur 9.2: De rekentijden van DSA-138 voor verschillende waarden van p.

Hoofdstuk 10
Conclusies & Aanbevelingen

Welke is nou beter? Hoewel verleidelijk kan geen eenduidig antwoord worden gegeven. Elliptische krommen zijn niet altijd sneller en ook DSA komt niet als grote winnaar uit de bus.

In de eerste paragraaf worden de conclusies gegeven. In paragraaf twee en drie volgen dan nog kanttekeningen bij deze conclusies en aanbevelingen voor verder onderzoek. De nadruk ligt hierbij op het verbeteren van de huidige keuzes in het experiment.

1. Conclusies

De eerste en voornaamste conclusie is dat naarmate grotere veiligheid wordt geëist en dus de bitlengtes voor p (in ECDSA) en q (in DSA) omhoog gaan, DSA in toenemende mate sneller is dan ECDSA. ECDSA reageert slechter op een toename van de bitlengte dan DSA. Een tweede kwalitatieve conclusie is dat de rekentijd van DSA lineair afhangt van de modulus p.

Voor kwantitatieve conclusies wordt uitgegaan van modulus 1536 in DSA. Van de drie gesimuleerde moduli heeft deze waarde het grootste draagvlak; niemand zal 768 als minimum betwisten, terwijl er voldoende mensen te vinden zijn, in het bijzonder Lenstra en Verheul, die 1536 aan de lage kant vinden. Bij modulus 1536 is ECDSA voor relatief lage veiligheid sneller dan DSA. Het omslagpunt ligt voor ECDSA met progressie bij 2035 (ECDSA:230 – DSA:172) en voor ECDSA zonder progressie bij 2084 (ECDSA:256 – DSA:240). Dus tot de veiligheid gerepresenteerd door bitlengte 256 in ECDSA en 240 in DSA, is ECDSA sneller. Wil je een grotere veiligheid dan is DSA sneller. Omdat de veiligheid die bitlengte 256 in ECDSA biedt zeer groot is, hij beschermt informatie tot 2035, kan geconcludeerd worden dat – in ieder geval op dit moment – ECDSA sneller is dan DSA.

2. Kanttekeningen

De belangrijkste kanttekening die bij dit – en ieder ander vergelijkend experiment – kan worden geplaatst is **het probleem van gelijke veiligheid**. Welke bitlengtes representeren voor de twee systemen gelijke veiligheid? In dit experiment is gekozen voor Lenstra-Verheul, maar er zijn ook voldoende andere keuzes denkbaar. Daarenboven is de tabel van Lenstra en Verheul niet zonder kritiek.

Een tweede punt van discussie is de keus van de modulus p in DSA, bij gegeven modulus q. De algemene consensus is dat deze gelijk gekozen moet worden aan de modulus in RSA. Deze modulus staat echter zelf ter discussie.

Een opmerking die specifiek voor dit experiment geldt is **het gebruik van bestaande krommen** als gevolg van de moeilijk te implementeren **ordeberekening van het basispunt P**. De keuze van de te simuleren parameters in ECDSA wordt door het gebruik van bestaande krommen zeer beperkt. Daarnaast kunnen de sleutel-routines van beide systemen niet worden vergeleken, omdat het in het ECDSA-simulatieprogramma niet mogelijk is eigen krommen te maken en dus een sleutel te genereren. De vergelijking in dit experiment zou dus op dit punt onvolledig genoemd kunnen worden.

Zij tot slot opgemerkt dat de simulatieprogramma's niet altijd gebruik maken van de efficiënste implementatie mogelijk.

3. Aanbevelingen

De aanbevelingen die worden gedaan op basis van dit experiment, volgen de kanttekeningen. Er zijn vier aanbevelingen voor verder onderzoek:

1. Onderzoek naar een methode om 'gelijke veiligheid' in bitlengtes te kunnen uitdrukken, in het bijzonder voor de hier gesimuleerde systemen.
2. Onderzoek naar de verhouding tussen de moduli p en q in DSA.
3. Programmeren van de volledige routine *sleutel* in ECDSA. Ofwel, programmeren van de ordeberekening van het basispunt P.
4. Evaluatie van beide simulatieprogramma's waarin wordt gekeken of en hoe ze efficiënter geïmplementeerd kunnen worden.

Daarnaast wordt geadviseerd te kijken naar grotere bitlengtes voor p in DSA, in het bijzonder de ondergrenzen voor klassieke asymmetrische systemen uit de tabel van Lenstra en Verheul.

Referenties

1. G. Alberts. *Jaren van berekening: toepassingsgerichte initiatieven in de Nederlandse wiskundebeoefening 1945-1960.* Proefschrift, Universiteit van Amsterdam, 1998.
2. G. Alberts, F. van der Blij, and J. Nuis. *Zij mogen uiteraard daarbij de zuivere wiskunde niet verwaarlozen.* C.W.I., 1987. Uitgegeven t.g.v. het 40-jarig bestaan van het CWI.
3. Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart. *Elliptic curves in cryptography.* Cambridge University Press, 1999.
4. T. H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* M.I.T. Press, 5th edition, 1991.
5. W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, pages 644 – 654, 1976.
6. D.W.Davies and W.L. Price. The application of digital signatures based on public key cryptosystems. In *Proceedings of the Fifth International conference on computer communication*, pages 525 – 530, 1981.
7. T. ElGamal. A public key cryptosystem and signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, pages 469 – 472, 1985.
8. S. Cavallar et al. Factoring RSA-155. In *Advances in Cryptology – Eurocrypt 2000*, pages 1–18, 2000.
9. B. Johnson and A. Menezes. The elliptic curve digital signature algorithm (ECDSA). *technical report CORR 99 – 34 (www.cacr.math.uwaterloo.ca)*, 2000.
10. D. E. Knuth. *Selected papers on computer science.* C.S.L.I., 1996.
11. Donald E. Knuth. *Seminumerical Algorithms.* The Art of Computer Programming – deel II. Addison-Wesley, Reading, MA, 3nd edition, 1998.
12. Donald E. Knuth. *Sorting and searching.* The Art of Computer Programming – deel III. Addison-Wesley, Reading, MA, second edition, 1998.
13. N. Koblitz. *Introduction to elliptic curves and modular forms.* Springer, 1984.
14. N. Koblitz. *A course in number theory and cryptography.* Springer, 1987.
15. N. Koblitz. *Algebraic aspects of cryptography.* Springer, 1998.

16. S. Lang. *Undergraduate Algebra.* Springer-Verlag, second edition, 1990.
17. A. Lenstra and Eric Verheul. Selecting cryptographic key sizes. *www.cryptosavvy.com*, 1999.
18. A.K. Lenstra. FreeLIP – multilengte-softwarepakket. *ftp://ftp.cam.uk.eu.microsoft.com/pub/-math/software/freelip_1.1.tar.gz*, 1991.
19. A. Menezes. *Elliptic curve public key cryptosystems.* Kluwer, 1993.
20. A.J. Menezes and S.A. Vanstone, editors. *Advances in Cryptology – Eurocrypt '90.* Springer-Verlag, 1991.
21. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1997.
22. B. Preneel, editor. *Advances in Cryptology – Eurocrypt 2000.* Springer-Verlag, 2000.
23. Hans Riesel. *Prime numbers and computer methods for factorization.* Birkhäuser, Boston, etc., second edition, 1994.
24. R. Rivest. The MD4 Message Digest Algorithm. In *Advances in Cryptology – Eurocrypt '90*, pages 303 – 311, 1991.
25. B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* Wiley, 1996.

Adresgegevens

begeleiders – stagebiedende instituut – opleiding – stagiaire

Begeleiders
Begeleider **Herman te Riele**
020 – 592 4106 (CWI)

Eerste stagedocent **Gerard Koestier**
020 – 676 9746 (privé)
020 – 495 1365 (Hogeschool Holland)

Stagebiedende instituut
onderzoeksinstituut **Centrum voor Wiskunde en Informatica**
Kruislaan 413
1098 SJ Amsterdam

Opleiding
HTO-opleiding **Bedrijfswiskunde**
Hogeschool Holland
Bergwijkdreef 10
1112 XD Diemen

Stagiaire
Dagmara Wilschut
Gouden Leeuw 1036
1103 KZ Amsterdam
020 – 695 9385

Hoofdstuk 11
Programmacode DSA

```
/*dsasigalg.c*/


/*DSA volgens NIST-Digital Signature Standard.
/*programma maakt publieke en geheime sleutel:
/* publieke sleutel p, q, y, alpha
/* geheime sleutel a met y=alpha mod p
/*programma genereert voor ingevoerde boodschap m handtekening
/* (r,s).
/*handtekening wordt op h(m), de hashwaarde van m, geplaatst.
/*programma controleert of handtekening (r,s) overeenkomt met
/* publieke sleutel*/

#include <stdio.h>
#include "lip.h"   /*multi-length software pakket*/
#include <time.h>  /*functie time*/
#include "md4.h"   /*hash file*/

#define blp 786 /*bitlengte priem p; NIST 768<=blp<=1024*/
#define blq 160 /*bitlengte van priem q; NIST:160*/
#define aantaltests 40 /*aantal iteraties van pseudopriemtest
                       /*in zrandomprime in de functie sleutel
                       /*NIST: kans samengesteld < 2\^(-80)
                       /*Miller-Rabin(pseudopriemtest):
                       /* kans per iteratie 1/4*/

/*globale variabelen*/
verylong p=0,q=0,quot=0,alpha=0,a=0,y=0;    /*sleutel*/
verylong m=0;                               /*boodschap*/
verylong r=0,s=0;                           /*handtekening*/
```

```
verylong w=0,u1=0,u2=0,v=0;                /*verificatie*/
/* gelijkstellen aan nul is vereist bij type verylong*/

float Tpriem, Tvoortbrenger;         /*tijd priemen p en q
                                     /*genereren en
                                     /*voortbrenger alpha
                                     /*vinden*/

void sleutel ()
{
/* maakt sleutel (p,q,y,alpha,a)
/* kiest p van lengte blp en q van lengte blq zo dat q|p-1
/* contrueert element y uit cyclische ondergroep van orde q
/* hiertoe wordt eerst voortbrenger alpha gezocht en random
/* geheime sleutel a gekozen (0 < a < q)
/* y:=alpha^a mod q */

    verylong g=0;                  /*DSA-hulpvariabele*/
    float tp1, tp2, tv1, tv2;      /*tijdmetingvariabelen*/

    tp1 = (double)clock()/(double)CLOCKS_PER_SEC;

    /*genereert p en q priem van lengte blp en blq met q|p-1*/
    zrandomqprime(blp,blq,aantaltests,&p,&q,&quot,zrandomb);

    tp2 = (double)clock()/(double)CLOCKS_PER_SEC;
    Tpriem = tp2 - tp1;

    zone(&alpha);/*alpha:=1 (nodig voor while-loop)*/

    tv1 = (double)clock()/(double)CLOCKS_PER_SEC;
    /*voortbrenger alpha van ondergroep orde q*/
    while (ztoint(alpha)==1)/*alpha!=1 => alpha voortbrenger*/
    {
        zrandomb(p,&g);            /*kies g tussen 0 en p*/
        zexpmod(g,quot,p,&alpha);  /*alpha:=g^quot mod p*/
    };

    tv2 = (double)clock()/(double)CLOCKS_PER_SEC;
    Tvoortbrenger = tv2 - tv1;

    /*maak y = alpha^a mod p; genereer a random*/
    while (ztoint(a)==0)           /*a=0 => y=1 (laakbaar)*/
    {
        zrandomb(q,&a);            /*kies random a*/
    };

    zexpmod(alpha,a,p,&y);         /*y:=alpha^a mod p*/

    printf("\nSLEUTEL \n");
```

```
    printf("publieke sleutel \n");
    printf(" p is ");zwriteln(p);
    printf(" q is ");zwriteln(q);
    printf(" alpha is ");zwriteln(alpha);
    printf(" y is ");zwriteln(y);

    printf("geheime sleutel \n");
    printf(" a is ");zwriteln(a);
    printf("\n");

}

void handtekening()
{
/* genereert handtekening (r,s) voor boodschap m
/* handtekening wordt geplaatst op h(m), hashwaarde van m
/* (berekend mbv MD4)
/* boodschap m wordt ingelezen van het toetsenbord
/* r :=alpha^k mod p mod q voor random k: 0 < k < q
/* k voor iedere boodschap nieuw (NIST); k geheim
/* s :=k^(-1){h(m)+a*r} mod q*/

    verylong k=0;                        /*hulpvariabele DSA*/
    verylong r1=0,kmin=0,ar=0,mplusar=0;/*hulpvariabelen*/

    /*hash-functie; leest m in van toetsenbord
    /*retourneert 128 bits hashwaarde --> m:=h(m)*/
    md4();

    zrandomb(q,&k);                    /*kies k 0 < k < q*/
    zexpmod(alpha,k,p,&r1);            /*r1:=alpha^k mod p*/

    /*eerste deel handtekening: r*/
    zmod(r1,q,&r);                     /*r:=alpha^k mod q*/

    /*voorwerk s:=k^(-1){m+ar} mod q*/
    zinv(k,q,&kmin);                   /*k invers modulo q*/
    zmul(a,r,&ar);                     /*ar:=a*&r*/
    zadd(m,ar,&mplusar);               /*mplusar:=m+ar*/

    /*tweede deel handtekening: s*/
    zmulmod(kmin,mplusar,q,&s);        /*s:=k^(-1){m+ar}mod q*/

    printf("\nHANDTEKENING \n");
    printf("handtekening (r,s) \n");
    printf(" r ");zwriteln(r);
    printf(" s ");zwriteln(s);

    printf("geheime deel (niet langer nodig) \n");
    printf(" k ");zwriteln(k);
}
```

```
void verificatie()
{
/*verifieert handtekening (r,s) bij boodschap m*/

    verylong v1=0, v2=0, v3=0;         /*hulpvariabelen*/

    if(((zcompare(r,q))==1)||((zcompare(s,q))==1))
    /*r en s kleiner dan q?*/
    {
        printf("Handtekening geweiger.");
    }
    else
    {
    zinv(s,q,&w);                      /*w:=s^(-1) mod q*/
    zmulmod(w,m,q,&u1);                /*u1:=w*m mod q*/
    zmulmod(r,w,q,&u2);                /*u2:=r*w mod q*/

    /*v:=alpha^u1 * y^u2 mod p mod q; berekening opgesplitst*/
    zexpmod(alpha,u1,p,&v1);           /*v1:=alpha^u1 mod p*/
    zexpmod(y,u2,p,&v2);               /*v2:=y^u2 mod p*/
    zmulmod(v1,v2,p,&v3);              /*v3:=v1*v2 mod p*/
    zmod(v3,q,&v);                     /*v:=v3 mod q*/

    printf("\nVERIFICATIE \n");

    printf(" v ");zwriteln(v);

    /*handtekening accepteren desda v==r*/
    if ((zcompare(v,r))==0)
        printf("Handtekening geaccepteerd.\n");
    else
        printf("Handtekening geweigerd.\n");
    }

}
```

```
md4()
{
/* hash functie
/* vraagt om boodschap van toetsenbord. Mag willekeurige
/* boodschap van willekeurige lengte zijn, afgesloten met EOF
/* (hier: tweemaal <CTRL> D.
/* berekent hashwaarde van m volgens MD4-algoritme (in md4.c)
/* m krijgt waarde MD, hash-waarde van ingevoerde boodschap*/

MDstruct MD;/*struct waarin MD van m wordt 'bijgehouden';
            /*bestaat uit vier woorden (32 bits van het
            /*type unsigned int): MD.buffer[i]*/

/*programmeerhulpvariabelen*/
verylong a=0,c=0,n=0,hulp=0,d=0;
long b=32;
int j;
char X[64];
char  i;

    zone(&a);      /*a:=1*/
    z2mul(a,&a);   /*a:=2*/
    zsexp(a,b,&c); /*c:=2^32*/

    MDbegin(&MD);               /*initialisatie MDstruct MD*/

    printf("Geef de boodschap, sluit af met EOF
      (2x <CNTRL> D): \n");

    while ((i=fread(X,1,64,stdin))>1)
    /*inlezen 64 chars (512 bits)*/
        MDupdate(&MD,X,i*8);
        /*MD 'aanpassen' aan volgende 64 char*/
    MDupdate(&MD,X,0);
    /*afronden contstructie MD*/

    /* m:=MD; array[4] van unsigned ints --> verylong*/
    zuintoz(MD.buffer[3],&m);      /*m:=(MD.buffer[3])*/
    for(j=2;j>=0;j--)
    {
        zcopy(m,&hulp);
        zmul(hulp,c,&m);           /*m:=m*(2^32)*/
        zuintoz(MD.buffer[j],&n);  /*n:=(MD.buffer[j])*/
        zadd(m,n,&m);
    }
}

main()
{
```

```
verylong seed=0;
float time1, time2,time3,time4;

    printf("Geef waarde voor seed s van randomgenerator
        (s>0) \n");
    zread(&seed);
    zrstart(seed);
    /*ingevoerde seed wordt default-waarde voor zrandomb*/

    time1 = (double)clock()/(double)CLOCKS_PER_SEC;
    sleutel();
    time2 = (double)clock()/(double)CLOCKS_PER_SEC;
    handtekening();
    time3 = (double)clock()/(double)CLOCKS_PER_SEC;
    verificatie();
    time4 = (double)clock()/(double)CLOCKS_PER_SEC;

    printf("\nBenodigde tijd totaal %2.2f sec waarvan \n",
        (double)(clock())/(double)CLOCKS_PER_SEC);
    printf("sleutel \t %2.2f sec\n", time2 - time1);
    printf("handtekening \t %2.2f sec\n", time3 - time2);
    printf("verificatie \t %2.2f sec\n", time4 - time3);

    printf("\nBenodigde tijd sleutel %2.2f sec waarvan \n",
        time2 -time1);
    printf("genereren priemen p en q\t%2.2f sec \n", Tpriem);
    printf("vinden voortbrenger alpha\t%2.2f sec \n",
        Tvoortbrenger);

return(0);

}
```

Hoofdstuk 12
Programmacode ECDSA

```
/* elliptische kromme implementatie van DSA*/


/*programma simuleert DSA voor elliptische krommen (ECDSA).
/* Hoewel het programma nog geen 'eigen krommen' genereert
/* is die optie wel ingebouwd.In de rest van het programma
/* wordt dit in het commentaar niet meer vermeld. De
/* mogelijkheid ontbreekt de orde van het basispunt P te
/* berekenen. In plaats daarvan kunnen bestaande krommen
/* worden gebruikt.
/* De elliptische kromme zijn van de vorm:
/* y^2=x^3+ax+b
/* ECDSA begint met een elliptische kromme (bepaald door
/* coefficienten a en b) en een punt P(x,y) op die kromme.
/* Vervolgens wordt een sleutel gecreerd en een handtekening
/* geplaatst en geverifieerd.
/* sleutel
     punt Q op de kromme met x.P=Q voor geheime sleutel x
         (0 < x < p-1)
**handtekening
     kies 1<k<q-1 en bereken k.P=(x1,x2)
     r:=x1 mod q (idem DSA)
     s:=k^(-1).{h(m)+xr} mod q (idem DSA)
     handtekening = (r,s) (idem DSA)
**verificatie
     r,s< q; anders direct verwerpen (idem DSA)
     w:=s^-1 mod q (idem DSA)
     u1:=h(m).w mod q & u2:=r.w mod q(idem DSA)
     u1.P + u2.Q = (x3,x4)
```

```
     v:=x3 mod p(idem DSA)
     als v=r accepteren, anders verwerpen (idem DSA)
*/


#include <stdio.h>
#include "lip.h"      /*multi-length software pakket*/
#include <time.h>     /*tijdfuncties - library in C*/
#include "md4.h"      /*hash file*/

#define blp 160       /*bitlengte van priem p*/

#define aantaltests 40/*aantal iteraties pseudopriemtest
                      /*in zrandomprime in functie sleutel
                      /*(alleen als genereer=1)*/
#define genereer 0    /*genereer=1 --> programma genereert
                      /*kromme; orde P is dan onbekend.
                      /*random x,y en a in Fp, berekent b
                      /*kromme: E(Fp):=y^2=x^3+ax+b,
                      /*punt (x,y)
                      /*genereer=0 --> programma gebruikt
                      /*waarden uit file getallen.c:
                      /*p,a,b,q,P.x,P.y*/

#define nummerkromme 1 /*genereer=0: Bepaalt welke kromme
                       /*uit getallen.c wordt ingelezen
                       /*1<=nummerkromme<=4
                       /*kromme 1: p 160 bits
                       /*kromme 2: p 192 bits
                       /*kromme 3: p 224 bits
                       /*kromme 4: p 230 bits
                       /*blp handmatig veranderen*/

/*VARIABELEN VARIABELEN VARIABELEN VARIABELEN VARIABELEN */

struct punt/*punt op de kromme; x- en y-coordinaat in Fp*/
{
  verylong x;    /*x coordinaat*/
  verylong y;    /*y coordinaat*/
  int oneindig;
  /*boolean:punt==point at infinity --> oneindig:=1*/
};

/*globale variabelen*/
struct punt P;/*basispunt*/
struct punt Q;/*geheime sleutel*/
struct punt R, Atemp,Btemp;/*hulpstructen programma*/
struct punt V1,V2,Vsom;/*hulpstructen functie verifieren*/

/*gelijkstellen nul vereist bij type verylong*/
verylong p=0;     /*modulus*/
verylong q=0;     /*orde basispunt P*/
verylong a=0,b=0; /*coefficienten elliptische kromme*/
verylong x=0;     /*geheime sleutel*/
verylong m=0;     /*boodschap*/
verylong r=0,s=0; /*handtekening*/
```

```
long rij[blp];/*array waarin bitrepresentatie van
              /*willekeurige elementen Fp wordt opgeslagen;
              /*wordt gebruikt in functie vermenigvuldig*/
      /*STRUCT-HULPFUNCTIES    STRUCT-HULPFUNCTIES*/
void schrijfpunt (struct punt A)
{
/*functie schrijft punt naar scherm*/
    if(!A.oneindig)
    {
        printf("(");
zwrite(A.x);
printf(",");
zwrite(A.y);
printf(")\n");
    }
    else
        printf("Point at infinity\n");
}
void intpunt (struct punt *A)
{
/*functie initialiseert punt A*/
    zuintoz(0,&(A->x));
    zuintoz(0,&(A->y));
    A->oneindig=0;
}
wordt(struct punt A, struct punt *B)
{
/*functie kopieert A in B*/
    zcopy(A.x,&(B->x));
    zcopy(A.y,&(B->y));
    (B->oneindig)=A.oneindig;
}
          /*HULPFUNCTIES VOOR DRIE HOOFDROUTINES*/
void kromme()
{
/* functie berekent - afhankelijk van waarde globale
/* variabele genereer - een elliptische kromme, gegeven
/* modulus p.
** genereer: 0 --> bestaande kromme
**           1 --> zelf genereren*/

    /*hulpvariabelen*/
    verylong xc=0,yc=0;
    verylong yorgineel=0,ax=0,x3=0,a3=0,va3=0,b2=0,zb2=0;
    verylong det=0;
    verylong temp=0;
    int i;
    int klaar=0; /*boolean:determinant gegeneerde
                 /*kromme!=0 --> klaar:=1*/
```

```
    FILE *inputfile;/*pointer naar file*/
    if(genereer==1)
    {
        while (!klaar)
        {
            zrandomb(p,&a);
    zrandomb(p,&xc);
    zrandomb(p,&yc);
    /*bereken b--> b:=y^2-x^3-ax*/
    zcopy(yc,&yorgineel);
    zsqinmod(&yc,p);               /*y:=y^2 mod p*/
    zsexpmod(xc,3,p,&x3);          /*x3:=x^3 mod p*/
    zmulmod(a,xc,p,&ax);           /*ax:=ax mod p*/

    /*b:=y^2-x^3-ax*/
    zsub(yc,x3,&b);
            if(zsign(b)==-1)/*handmatig moduleren*/
        zadd(b,p,&b);
    zsub(b,ax,&b);
    if(zsign(b)==-1)
        zadd(b,p,&b);
    /*determinant kromme!=0; 4.a^3 + 27.b^2 ?=0*/
    zsexpmod(a,3,p,&a3);zsmulmod(a3,4,p,&va3);
    zsexpmod(b,2,p,&b2);zsmulmod(b2,27,p,&zb2);
    zaddmod(va3,zb2,p,&det);/*det:=4a^3+27b^2*/
    if(!(zsign(det)==0))
        klaar=1;
        }

        /*punt P krijgt berekende x en y als coordinaten*/
        zcopy(xc,&P.x);
        zcopy(yorgineel,&P.y);
    }
    else/*if (genereer==0)*/
    {
        inputfile = fopen("getallen.c", "r");
        /*getallen.c openen voor lezen*/
        if (inputfile == NULL )
            printf("File getallen.c niet gevonden.\n");
        for(i=1;i<=((nummerkromme-1)*6);i++)
{
            zfread(inputfile, &temp);
}
/*inlezen van variabelen uit getallen.c*/
        zfread(inputfile,&p);
zfread(inputfile,&a);
zfread(inputfile,&b);
zfread(inputfile,&q);
```

```
zfread(inputfile, &P.x);
zfread(inputfile, &P.y);

fclose(inputfile);
    }
}

void optellen(struct punt A, struct punt B, struct punt *D)
{
/*functie berekent D = A+B*/

    int hetzelfde;/*boolean: als A=B --> hetzelfde:=1*/
    int bijzonder;/*boolean: als bijzondere optelling
                  /*--> bijzonder:=1*/

    /*hulpvariabelen*/
    verylong labda=0,teller=0,noemer=0,minp=0;
    verylong labda2=0,hulpD=0,AminD=0,lAminD=0,x2=0,driex2=0;
    verylong x11=0,x22=0;
    verylong hA=0,hB=0;
    verylong kopieAy=0;
    verylong nul=0;/*getal nul als verylong*/

    zsub(nul,p,&minp);                /*minp:=-p (p > 0)*/

    hetzelfde=0;
    bijzonder=0;
    /*D->oneindig=0;default; D->oneindig namelijk gebruikt
    /*voor onderscheid*/

    /*A?=point at infinity, B?=point at infinity of beide?*/
    if(A.oneindig==1)
    {
        if(B.oneindig==1)
{
    D->oneindig=1;
    bijzonder=1;
        }
        else
        {
    zcopy(B.x,&(D->x));
    zcopy(B.y,&(D->y));
    D->oneindig=0;
    bijzonder=1;
}
    }
    else if(B.oneindig==1)
    {
        zcopy(A.x,&(D->x));
zcopy(A.y,&(D->y));
D->oneindig=0;
bijzonder=1;
    }
```

```
    if(!(bijzonder))
    {
        /*A?=B || A.x?=B.x*/
        if(zcompare(A.x,B.x)==0)
        {
            if(zcompare(A.y,B.y)==0)/*A=B*/
                hetzelfde=1;
            else/*A=-B*/
            {
D->oneindig=1;
bijzonder=1;
    }
        }


        /*A.y?=B.y*/
        else if (zcompare(A.y,B.y)==0)/*D=derde 'wortel'*/
        {
    zcopy(A.y,&kopieAy);
    znegate(&kopieAy);        /*kopieAy:= -kopieAy*/
    zadd(kopieAy,p,&kopieAy); /*kopieAy>0*/
            zcopy(kopieAy,&(D->y));
    /*x3:=-x1-x2*/
    zcopy(A.x,&x11);zcopy(B.x,&x22);znegate(&x11);
    zsub(x11,x22,&(D->x));
    if(zcompare(D->x,0)==-1)/*handmatig moduleren*/
    {
        if(zcompare(D->x,minp)==-1)
                /*D->x<-p --> D->x:=D->x+p*/
                    zadd(D->x,p,&(D->x));
        zadd(D->x,p,&(D->x));
            }
    bijzonder=1;
}
    }

    if(hetzelfde)
    {
        if(zsign(A.y)==0)
    D->oneindig=1;
    }

    if(!(bijzonder==1))
    {
        if(!hetzelfde)
{
            zsub(B.y, A.y, &teller);   /*teller:=(B.y-A.y)*/
            zsub(B.x, A.x, &noemer);   /*noemer:=(B.x-A.x)*/

            if (zsign(noemer)==-1)
            {
                zabs(&noemer);         /*absolute waarde*/
                zinvmod(noemer,p,&noemer);
```

```
                /*noemer:=noemer^(-1) mod p*/
                zsub(p,noemer,&noemer);/*noemer:= -noemer+p*/
            }
            else
                zinv(noemer,p,&noemer);/*noemer^-1 mod p*/
    /*labda:=(B.y-A.y)/(B.x-A.x) mod p*/
            zmulmod(teller,noemer,p,&labda);
        }
        else
        {
            zsq(A.x,&x2);               /*x2:=A.x^2*/
    zmod(x2,p,&x2);
    zsmul(x2,3,&driex2);        /*driex2:=3.(A.x)^2*/
    zadd(driex2,a,&teller);     /*teller:=3(A.x)^2+a*/
    zsmul(A.y,2,&noemer);       /*noemer:=2(A.y)*/
    zinv(noemer,p,&noemer);
    zmulmod(teller,noemer,p,&labda);
            /*labda:=(3(A.x)^2+a)/2(A.y)*/
}
        /*D.x = labda^2 - A.x - B.x*/
zsq(labda,&labda2);             /*labda2:=labda^2*/
zmod(labda2,p,&labda2);         /*reduceren modulo p*/
zcopy(A.x,&hA);
zsub(labda2,hA,&hulpD);
if(zsign(hulpD)==-1)
    zadd(hulpD,p,&hulpD);
zcopy(B.x,&hB);
zsub(hulpD, hB, &(D->x));
zcopy(hA,&A.x);
        /*handmatig moduleren; je weet -p<= (D->x) <= +p*/
if(zcompare(D->x,0)==-1)
    zadd(D->x,p,&(D->x));
        /*D.y = labda*(A.x - D.x) - A.y*/
        zsub(A.x,D->x,&AminD);
zmulmod(labda,AminD,p,&lAminD);
zsub(lAminD, A.y,&(D->y));

        /*idem D->y:  -p<= (D->y) <= +p*/
if(zcompare(D->y,0)==-1)
            zadd(D->y,p,&(D->y));
    }
}
void maakschoon()
{
/*vermenigvuldig; leegt array waarin binaire
/*representatie veelvoud wordt bijgehouden*/
    int i;

    for(i=0;i<=blp;i++)
        rij[i]=0;
}
```

```
void vermenigvuldig(verylong d,struct punt A,struct punt *B)
{
/*vermenigvuldigt punt A met d; resultaat B = d.A*/

    verylong dee=0;/*kopie van orginele d*/
    int i=0,j;
    int slaover=0;
    /*boolean; d binair < 2 bits -->slaover :=1*/
    int klaar=0;
    /*als d=0--> klaar=1 (&B=point at infinity*/

    zcopy(d,&dee);

    if(zsign(dee)==0)
    {
B->oneindig=1;
klaar=1;
    }

    if(!klaar)
    {
        while (!(zsign(dee)==0))
        {
  rij[i]=z2mod(dee);
          /*i-de element ‘rij‘ wordt dee mod 2*/
  zrshift(dee,1,&dee);
          /*shift bits van dee 1 naar rechts*/
          i++;
        }

        Atemp=A;

        /*B=A omdat eerste bit altijd 1*/
        wordt(A,B);

        if(i>=2)
            j=i-2;
        else
            slaover=1;

        while(B->oneindig==0&&(j>=0)&&(!slaover))
        {
    optellen(*B,*B,&Btemp);
    wordt(Btemp,B);
    if(rij[j]==1)
    {
        optellen(*B,Atemp,&Btemp);
        wordt(Btemp,B);
    }
    j--;
        }
        maakschoon();
    }
}
```

```
md4()
{
/* hash functie
/* vraagt boodschap van toetsenbord. Mag willekeurige
/* boodschap van willekeurige lengte zijn, afgesloten met
/* EOF (hier: tweemaal <CTRL> D. berekent message digest
/* van m volgens MD4-algoritme (in md4.c)
/* m krijgt waarde MD, hash-waarde van ingevoerde boodschap*/
    MDstruct MD;/*struct waarin MD van m wordt 'bijgehouden';
                /*bestaat uit vier woorden (32 bits van het
                /*type unsigned int): MD.buffer[i]*/
    /*programmeerhulpvariabelen*/
    verylong a1=0,c=0,n=0,hulp=0,d=0;
    long b1=32;
    int j;
    char X[64];
    char  i;
    zone(&a1);       /*a:=1*/
    z2mul(a1,&a1);   /*a:=2*/
    zsexp(a1,b1,&c); /*c:=2^32*/
    MDbegin(&MD);             /*initialisatie MDstruct MD*/

    printf("Geef de boodschap, sluit af met EOF
        (2x <CNTRL> D): \n");
    while ((i=fread(X,1,64,stdin))>1)
    /*inlezen 64 chars (512 bits)*/
        MDupdate(&MD,X,i*8);
        /*MD 'aanpassen' aan volgende 64 char*/
    MDupdate(&MD,X,0);        /*afronden contstructie MD*/
    /* m:=MD; array[4] van unsigned ints --> verylong*/
    zuintoz(MD.buffer[3],&m); /*m:=(verlong)(MD.buffer[3])*/
    for(j=2;j>=0;j--)
    {
        zcopy(m,&hulp);
        zmul(hulp,c,&m);      /*m:=m*(2^32)*/
        zuintoz(MD.buffer[j],&n);
        /*n:=(verylong)(MD.buffer[j])*/
        zadd(m,n,&m);
    }
}
/*HOOFDROUTINES*HOOFDROUTINES*HOOFDROUTINES*HOOFDROUTINES*/
/*       SLEUTEL  *  HANDTEKENING  *  VERIFICATIE       */
void sleutel ()
{
/*deels afhankelijk van variabele genereer
/*genereer=1 --> genereert priemmodulus p van bitlengte
/*               blp (define) & berekent kromme E(Fp):
/*               y^2=x^3+ax+b
```

```
/*genereer=0 --> leest priemmodulus p uit file getallen.c
/*               functie kromme leest a,b en punt P uit
/*               file getallen.c
/*
/*berekent in beide gevallen Q:=xP
/*Q publieke en x geheime sleutel*/
    if(genereer==1)
    {
        /*kies p random - groep = Fp*/
        zrandomprime(blp, aantaltests, &p, zrandomb);
        zcopy(p,&q);
    }


    kromme();
    /*Q:=x.P voor random x; x geheime sleutel, Q publiek*/
    while(zsign(x)==0)
        zrandomb(q,&x);           /*kies x tussen 1 en q-1*/
    vermenigvuldig(x,P,&Q);       /*Q:=x.P*/


    printf("\nSLEUTEL \n");


    printf("publieke sleutel \n");
    printf(" P is ");schrijfpunt(P);
    printf(" Q is ");schrijfpunt(Q);


    printf("geheime sleutel \n");
    printf(" x is ");zwriteln(x);
    printf("\n");
}
void handtekening()
{
/* genereert handtekening (r,s) voor boodschap m
/* handtekening wordt geplaatst op h(m), hashwaarde van
/* m (berekend mbv MD4)
/* boodschap m wordt ingelezen van het toetsenbord
/* afhankelijk van variabele genereer wordt gewerkt met
/* de volgende modulus:
/* genereer = 0 --> priemorde q van basispunt P uit file
/*                  getallen.c
/* genereer = 1 --> onbekende priemorde q wordt gelijk-
/*                  gesteld aan modulus p
/* random k: 0 < k < q --> k.P = R
/* k voor iedere boodschap nieuw (NIST); k geheim
/* r :=R.x mod q
/* s :=k^(-1){h(m)+a*r} mod q*/
    verylong k=0;/*DSA-variabele; geheim&eenmalig*/
    verylong kmin=0,xr=0,mplusar=0;/*hulpvariabelen*/
```

```
    intpunt(&R);

    /*hash-functie; leest m in van toetsenbord
    /*retourneert 128 bits hashwaarde h(m) --> m:=h(m)*/
    md4();
    /*loop: r,s,k alle drie ongelijk nul*/
    while(zsign(s)==0)
    {
        if(zsign(r)!=0)
    zcopy(0,&r);
        while(zsign(r)==0)
        {
    if(zsign(k)!=0)
        zcopy(0,&k);
    while(zsign(k)==0)
                zrandomb(q,&k);      /*kies k: 0<k<q*/
            vermenigvuldig(k,P,&R);  /*R:=k.P*/
            /*eerste deel handtekening: r*/
            zmod(R.x,q,&r);          /*r:=R.x mod q*/
}

        /*voorwerk s:=k^(-1){m+xr} mod q*/
        zinv(k,q,&kmin);             /*k invers modulo q*/
        zmul(x,r,&xr);               /*xr:=x*&r*/
        zadd(m,xr,&mplusar);         /*mplusar:=m+xr*/
        /*tweede deel handtekening: s*/
        zmulmod(kmin,mplusar,q,&s);  /*s:=k^(-1){m+xr} mod q*/
    }
    printf("\nHANDTEKENING \n");
    printf("handtekening (r,s) \n");
    printf(" r ");zwriteln(r);
    printf(" s ");zwriteln(s);
    printf("geheime deel (niet langer nodig) \n");
    printf(" k ");zwriteln(k);
}
void verificatie()
{
/*verifieert handtekening (r,s) bij boodschap m
**r,s<q; anders direct verwerpen
**berekent u1.P + u2.Q = Vsom voor
**u1:=h(m).w mod q & u2:=r.w mod q(idem DSA)
**w:=s^-1 mod q
**v:=Vsom.x mod q
**als v=r accepteren, anders verwerpen*/
    verylong w=0,u1=0,u2=0,v=0;      /*ECDSA-variabelen*/
```

```
    if(((zcompare(r,q))==1)||((zcompare(s,q))==1))
    /*r,s ?< q*/
    {
        printf("Handtekening geweigerd; r of s is
            kleiner dan q.\n");
    }
    else
    {
        zinv(s,q,&w);                /*w:=s^(-1) mod q*/
zmulmod(w,m,q,&u1);          /*u1:=w*m mod q*/
zmulmod(r,w,q,&u2);          /*u2:=r*w mod q*/
/*Vsom:=u1.P + u2.Q mod p; berekening opgesplitst*/
vermenigvuldig(u1,P,&V1);
        vermenigvuldig(u2,Q,&V2);
optellen(V1,V2,&Vsom);

zmod(Vsom.x,q,&v);           /*v:=Vsom.x mod q*/
printf("\nVERIFICATIE \n");
/*handtekening accepteren desda v==r*/
if ((zcompare(v,r))==0)
    printf("Handtekening geaccepteerd.\n");
else
            printf("Handtekening geweigerd.\n");
    }
}
main()
{
    verylong seed=0;
    float time1, time2,time3,time4;/*tijd-variabelen*/
    intpunt(&P);
    intpunt(&Q);
    printf("Geef waarde voor seed s van
         randomgenerator (s>0) \n");
    zread(&seed);
    zrstart(seed);
    /*ingevoerde seed wordt default-waarde voor zrandomb*/

    time1 = (double)clock()/(double)CLOCKS_PER_SEC;
    sleutel();
    time2 = (double)clock()/(double)CLOCKS_PER_SEC;
    handtekening();
    time3 = (double)clock()/(double)CLOCKS_PER_SEC;
    verificatie();
    time4 = (double)clock()/(double)CLOCKS_PER_SEC;
    printf("\nBenodigde tijd totaal %2.2f sec waarvan \n",
        (double)(clock())/(double)CLOCKS_PER_SEC);
    printf("sleutel \t %2.2f sec\n", time2 - time1);
    printf("handtekening \t %2.2f sec\n", time3 - time2);
    printf("verificatie \t %2.2f sec\n", time4 - time3);
    return(0);
}
```

Hoofdstuk 13
Beschrijving MD4-hashfunctie

De hashfunctie MD4 maakt van de boodschap m, een bitstring van willekeurige, eindige lengte, een hashwaarde $h(m)$ van 128 bits.
In zijn artikel *The MD4 Message Digest Algorithm* [24] geeft Rivest – naast een korte geschiedenis van hashfuncties in het algemeen en zijn doelstellingen – alleen een *beschrijving* van het algoritme, zonder zijn keuzes te motiveren of toe te lichten. Juist bij algoritmen die ogenschijnlijk random werken is de keus van parameters, de volgorde van functieaanroepen en constanten ondoorzichtig; toelichting is dan welkom. MD4 volgt wel zeer strak het 'hashfunctie format' uit hoofdstuk 4 en wellicht worden afwegingen bekend verondersteld.

VOORBEREIDING
De te hashen boodschap m heeft bitlengte b, waarbij b een willekeurig getal groter gelijk 0 is. De bits van m worden aangeduid met $m_0 m_1 \ldots m_{b-1}$. In onderstaande beschrijving worden een aantal definities en notaties gebruikt die eerst zullen worden uitgelegd.

byte 8 bits

woord 4 bytes

$<<<$ **a** bit-operatie: een 'rotatieverschuiving' linksom van a bits. Ieder bit wordt dus a plaatsen naar links opgeschoven; de bits die er op deze manie 'afvallen' worden rechts weer toegevoegd. Bijvoorbeeld, **010**0111 $<<<$ 3 geeft 0111**010**.

PADDING
De boodschap m wordt aangevuld zo dat de lengte congruent is met 448, modulo 512 bits. Het is dan 64 bits 'verwijderd' van een veelvoud van 512. Deze resterende bits worden gebruikt om de lengte van de oorspronkelijke boodschap in op te slaan. Mocht de boodschap groter zijn dan deze 2^{64} dan worden de laatste, minst significante bits gebruikt. Er wordt *altijd* verlengd, dus ook als toevallig $m \equiv 448 \pmod{512}$.

Padding gebeurt als volgt: aan het eind van de bitstring m wordt een '1' toegevoegd en daarna een rij nullen tot de gewenste lengte is bereikt. Deze '1' is nodig om 'hashnullen' van de oorspronkelijke string te kunnen onderscheiden.

Het resultaat is een bitstring met een lengte die een veelvoud k is van 512, en dus ook een veelvoud N van woorden, met $N = 16k$. Laat $M[0 \ldots N \Leftrightarrow 1]$ de woorden van de resulterende bitstring zijn.

ITERATIEVE COMPRESSIEFUNCTIE

Als eerste wordt de voortschrijdende hashwaarde, de MD-buffer, geïnitialiseerd. De buffer is 128 bits lang, ofwel vier woorden – aangeduid met A, B, C en D. Deze vier woorden worden met de volgende, hexadecimale waarden geïnitialiseerd

A	01	23	45	67
B	89	ab	cd	ef
C	fe	dc	ba	98
D	76	54	32	10

De iteratieve functie bestaat uit drie ronden waarin achtereenvolgens de functies f, g en h worden aangeroepen. Deze functies hebben alle drie als invoer drie woorden en als uitvoer één woord. Ze werken bitsgewijs op de drie invoerwoorden.

1. $f(X,Y,Z) = XY \vee \neg(X)Y$ conditioneel: als x dan y, anders z
2. $g(X,Y,Z) = XY \vee XZ \vee YZ$ meerderheid: tenminste twee van x,y en z
3. $h(X,Y,Z) = XxorYxorZ$ minderheid: precies één van x,y en z

De iteratieve functie is een for-loop die alle blokken $M[i]$ van 16 woorden doorloopt. Iedere ronde doorloopt in een eigen for-loop alle woorden per blok. Hierin gebeurt het eigenlijke hashen. Iedere ronde bestaat uit 16 functieaanroepen, voor ieder woord X één: functie f in ronde 1, g in ronde 2 en h in ronde 3. Vóór ronde 1 wordt de MD-buffer opgeslagen: A in AA, B in BB,C in CC en D in DD. Deze zullen na ronde 3 worden gebruikt.

Ronde 1

Laat [*A B C D i s*] de volgende operatie betekenen

$$A = (A + f(B,C,D) + X[i]) <<< s.$$

Hierin is $X[i]$ het i-de woord van het huidige blok. Teller i doorloopt het interval $[0,15]$, s is herhaaldelijk 3,7,11 en 19. De blokken A, B, C en D worden iedere aanroep één plaats naar rechts verschoven. De eerste drie aanroepen zijn
[*A B C D 0 3*]
[*D A B C 1 7*]
[*C D B A 2 11*].

Ronde 2

In ronde 2 gebeurt iets vergelijkbaars, alleen wordt hier bovendien een constante toegevoegd. De aanroep [*A B C D i s*] betekent

$$A = (A + g(B < C < D) + X[i] + 5a827999) <<< s.$$

Deze constante $5a827999$ is niet helemaal uit de lucht gegrepen: het is een 32-bits deel van de ontwikkeling van $\sqrt{2}$, afkomstig van een tabel in deel 2 van Knuth's *The Art of Computer Programming* [11]. Zijn tabellen worden regelmatig gebruikt om 'random' contstanten te vinden. Ook in deze ronde 16 aanroepen, waarin i het interval $[0,15]$ doorloopt volgens [0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15] en s herhaaldelijk 3,5,9 en 13 is. De blokken schuiven weer ieder aanroep één plaats naar rechts. Ronde 2 begint dus met [*A B C D 0 3*].

Ronde 3
In deze laatste ronde betekent $[A\ B\ C\ D\ i\ s\ ]$

$$A = (A + h(B < C < D) + X[i] + 6ed9eba1) <<< s$$

waarbij de constante $6ed9eba1$ weer afkomstig is uit Knuth; deze keer is het $\sqrt{3}$. i doorloopt zijn interval [0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15] en s – herhaaldelijk – 3,9,11 en 15. De eerste aanroep is wederom $[\ A\ B\ C\ D\ 0\ 3\ ]$.

Tot slot
Resultaat van deze rondes zijn de blokken A, B, C en D. Tot slot wordt dan nog de oorspronkelijke waarde van ieder blok bij de nieuwe waarde opgeteld
$A = A + AA$
$B = B + BB$
$C = C + CC$
$D = D + DD$.

Resultaat
De hahswaarde $h(m)$ van boodschap m is de 128 bits string $ABCD$, de concatenatie van A, B, C en D.

Hoofdstuk 14
File getallen.c – bestaande krommen

In de file getallen.c zijn zeven krommen opgeslagen. Van iedere kromme is achtereenvolgens de volgende informatie bewaard:
1. modulus p
2. coefficiënt a
3. coefficiënt b
4. orde q van het basispunt P
5. x-coördinaat van P
6. y-coördinaat van P

FILE GETALLEN.C

```
1461501637330902918203684832716283019655932542983
10
134363276215009249970163743897076481852807556507 8
1461501637330902918203683518218126812711137002561
916268978016867856409840773281569781084143028751
67245778215322775928012373601292232582225988444 4
627710173538668076383578942320766641608390870039032496127 9
-3
24551555460089438177402939151974517847691080581611912380 65
62771017353866807638357894231760590137671947731828422840 81
60204628237568865675821348058752611191669897663688468481 8
17405033229362203140485755228021941036402348892738665064 1
26959946667150639794667015087019630673557916260026308143510066298881
-3
18958286285566608000408668544493926415504680968679321075787234672564
26959946667150639794667015087019625940457807714424391721682722368061
19277929113566293071110308034699488026831934219452440156649784352033
19926808758034470970197974370888749184205991990603949537637343198772
17254365866976409468586889655692563631127772430425966387906310559498 91
```

```
7
3076062716593211670800930834288601694174418861512281754061963336251 5
5751455288992136489528963218564187831892921282154032337301807077850 93
1712025255479900821538790370285151763108545244878707903304675152736720
4444290301102136058138199264848929764840745598141279384197619947851 3
1157920892103562487626974469494075735300861434152903141955336313088 6\
7097853951
-3
4105836372515214212932612978004726840911444101599372555483525631403 94\
6740129111579208921035624876269744694940757352999695522413576034242 22\
59061068512044369
4843956129390645175905258525279791420276294952604174799584408071708 240\
4635286361342509567497957985851279195878819566111066729850150718771 982\
53568414405109
3940200619639447921227904010014361380507973927046544666794829340424 5721\
7714968703290472660882589380018616069731123 19
-3
2758019355995970587784901184038904809305690585636156852142870730198 8689\
2413098608651362607648837451077654397612305 75
3940200619639447921227904010014361380507973927046544666794690527962 7659\
3991132635693989563081522949135544336539426 43
2624703509579968926862315674456698189185292349110921338781561590092 5518\
8547380500890223880539757197866508724767320 87
8325710961489029985546751289520108179287853048861315594709205902480 5031\
9988441922443864376039294733307808651162787 1
6864797660130609714981900799081393217269435300143305409394463459185 5431\
8339765605212255964066145455497729631139148085803712198799971664381 2574\
028291115057151
-3
1093849038073734274511112390766805569936207598951683748994586394495 953\
1161507350160137087375737596232485921322967063133094384525315910129 121\
42327488478985984
6864797660130609714981900799081393217269435300143305409394463459185 5431\
8339765539424505774633321719753296399637136332111386476861244038034 037\
2808892707005449
2661740802050217063228768716723360960729859168756973147706671368418 8029\
4499642780849154508062777190235209424122506555866215711354557091681 4161\
637315895999846
3757180025770020463545507224491183603594455134769762486694567779615 5444\
7744055631669123440501294553956214444453728942852258566672919658081 0124\
344277578376784
```