1 Probabilistische Primzahltests

1.1 Quadratische Reste

Kurze Wiederholung: Quadratische Reste / Quadratische Nichtreste. Sei p eine Primzahl, a eine natürliche Zahl. Dann heißt

$$\left(\frac{a}{p}\right)$$

das **Legendre-Symbol** („a Legendre p”), und es gilt:

$$\left(\frac{a}{p}\right) = \begin{cases} +1: & a \text{ ist QR mod } p \\ 0: & p|a \\ -1: & a \text{ ist QNR mod } p \end{cases}.$$

Euler sagt nun, dass für jede ungerade Primzahl p gilt:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}.$$

Sei nun n eine beliebige ungerade Zahl, also $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$ mit ungeraden Primfaktoren p_j. Dann ist $\left(\frac{a}{n}\right)$, das **Jacobi-Symbol** („a Jacobi q”), als Verallgemeinerung des Legendre-Symbols folgendermaßen definiert:

$$\left(\frac{a}{n}\right) := \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i}.$$

Dabei sind die einzelnen $\left(\frac{a}{p_i}\right)$ *Legendre*-Symbole.

Wie kann man das Jacobi-Symbol schnell berechnen?

Zur Berechnung benutzt man 3 praktische Eigenschaften des Jacobi-Symbols:

1. $\left(\frac{2}{n}\right) = \begin{cases} +1: & n \equiv \pm 1 \pmod{8}, \\ -1: & n \equiv \pm 3 \pmod{8}. \end{cases}$

2. Wenn n ungerade ist, dann ist $\left(\frac{m_1 m_2}{n}\right) = \left(\frac{m_1}{n}\right)\left(\frac{m_2}{n}\right)$, insbesondere also $\left(\frac{2^k t}{n}\right) = \left(\frac{2}{n}\right)^k \left(\frac{t}{n}\right)$.

3. Sind m und n zwei ungerade Zahlen, dann ist
$$\left(\frac{m}{n}\right) = \begin{cases} -\left(\frac{n}{m}\right): & m \equiv n \equiv 3 \pmod{4}, \\ +\left(\frac{n}{m}\right): & \text{sonst.} \end{cases}$$

Es lässt sich nun leicht ein Algorithmus entwerfen, der durch sukzessives Herausziehen des Faktors 2 und ansonsten ein dem Euklidischen Algorithmus ähnliches Verhalten das Jacobi-Symbol in $\mathcal{O}(log^3(n))$ berechnet.

1.2 Fermat-Test

Ist p eine Primzahl, dann gilt

$$a^{p-1} \equiv 1 \pmod{p}$$

für jedes a mit $\text{ggt}(a,p) = 1$. Dies liefert einen Monte-Carlo-Algorithmus mit hoher Fehlerwahrscheinlichkeit. Sagt der Fermat-Test aber „nicht prim”, dann weiß man sofort Bescheid.

1.3 Solovay-Strassen

Der Solovay-Strassen-Algorithmus zum Primalitätstest ist ein Monte-Carlo-Algorithmus mit Bias auf „nicht prim", d.h. gibt der Algorithmus „nicht prim" zurück, dann ist die Zahl auch wirklich nicht prim. Gibt er „prim" zurück, dann ist sie wahrscheinlich prim. Die Fehlerwahrscheinlichkeit ist $\varepsilon = \frac{1}{2}$.

1. Wähle zufällig eine Zahl $1 \leq a \leq n-1$.
2. Wenn $\left(\frac{m}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod n$ ist, **dann** gib „n ist prim" zurück, **sonst** gib „n ist nicht prim" zurück.

Wenn der Solovay-Strassen-Algorithmus also m-mal gelaufen ist und jedesmal „n ist prim" geantwortet hat, dann ist n mit Wahrscheinlichkeit $p \approx 1 - 2^{-m}$ eine Primzahl.

1.4 Miller-Rabin

Der Primzahltest nach Miller und Rabin ist ebenfalls ein Monte-Carlo-Algorithmus mit Bias auf „nicht prim", d.h. wenn der Algorithmus sagt „n ist nicht prim", dann hat er immer Recht.

1. Schreibe $n - 1 = 2^k m$ mit $2 \not| m$.
2. Wähle zufällig $1 \leq a \leq n-1$.
3. Berechne $b = a^m \bmod n$.
4. Wenn $b \equiv 1$, dann „n ist prim".
5. für $i = 0$ bis $k-1$
   wenn $b \equiv -1$, dann „n ist prim",
   sonst $b := b^2 \bmod n$.
6. „n ist nicht prim".

Die Fehlerwahrscheinlichkeit des Miller-Rabin-Algorithmus ist $\varepsilon = \frac{1}{4}$.

2 Faktorisierung

Integer-Faktorisierung benötigen wir im Wesentlichen, um RSA zu attackieren. Daher soll gleich mit zwei RSA-bezogenen Faktorisierungsmöglichkeiten begonnen werden.

2.1 Faktorisierung mit Kenntnis von $\phi(n)$

Kennt man den Wert der eulerschen ϕ-Funktion zu einer Zahl $n = p \cdot q$, so kann man leicht die beiden Primfaktoren p und q bestimmen: $n = p \cdot q$ und $\phi(n) = (p-1) \cdot (q-1)$ lassen sich umformen zu $n = p \cdot q$ und

$$\phi(n) = n - p - q + 1 = n - p - \frac{n}{p} + 1 \;\Leftrightarrow\; 0 = np - \phi(n)p - p^2 - n + 1.$$

Diese quadratische Gleichung in p lässt sich problemlos lösen.

2.2 Faktorisierung mit Kenntnis der Exponenten d und e

Kennt man zu einer RSA-Konfiguration außer n noch den öffentlichen Schlüssel e und den privaten Schlüssel d, so kann man mit diesem Wissen die Faktorisierung von $n = p \cdot q$ bestimmen. Der Algorithmus ist probabilistisch (Las Vegas) mit Erfolgswahrscheinlichkeit $p \geq \frac{1}{2}$ pro Durchlauf. Insgesamt ergibt sich ein Algorithmus mit erwarteter polynomialer Laufzeit.

1. Wähle zufällig $1 \leq w \leq n-1$.
2. Berechne $x := \text{ggT}(w, n)$.
3. Wenn $1 < x < n$, dann **return** ,,Faktor gefunden: x".
4. Schreibe $ab - 1 = 2^s r$ mit $2 \not| r$.
5. Berechne $v := w^r \bmod n$.
6. Wenn $v \equiv 1$, dann **return** ,,Geht nicht".
7. Solange $w \not\equiv 1$ ist:
   $v_0 := v$, $v := v^2 \bmod n$.
8. Wenn $v_0 \equiv -1$ ist, dann **return** ,,Geht nicht",
   sonst **return** ,,Faktor gefunden: $\text{ggT}(v_0 + 1, n)$".

2.3 Faktorisierung spezieller Zahlen: Pollard p-1

Der p-1-Algorithmus von Pollard aus dem Jahr 1974 kann zur Faktorisierung von Zahlen mit einer bestimmten Eigenschaft verwendet werden. Besitzt n einen Primfaktor p, so dass für jede Primpotenz q mit $q \mid (p-1)$ gilt $q \leq B$, dann mus $(p-1) \mid B!$ gelten. $p-1$ nennt man dann auch ,,B-potenzglatt". Offenbar gilt dann:

$$2^{B!} = 2^{k \cdot (p-1)} \equiv 1 \pmod{p}.$$

Der Algorithmus läuft dann ganz straight-forward so ab:

1. $a := 2$.
2. for $j = 2$ to B do $a := a^j \bmod n$.
3. $d := \text{ggT}(a - 1, n)$.
4. Wenn $1 < d < n$ ist, dann **return** d,
   sonst **return** ,,Mist!".

2.4 Geburtstagsprinzip: Pollard Rho (ρ)

Sei n eine zusammengesetzte Zahl und d einer ihrer Primfaktoren. Gelingt es uns, zwei Zahlen x_i und x_j zu finden mit $x_i \equiv x_j \pmod{d}$, dann liefert $\text{ggT}(n, x_i - x_j)$ einen nicht-trivialen Teiler von n. Wir verwenden zur Konstruktion einer (hoffentlich zufälligen) Folge x_i die Vorschrift

$$x_{i+1} := f(x_i) \equiv x_i^2 + c,$$

wobei üblicherweise $c = 1$ ist.

1. $x_i := 2$, $x_j := 4 + c$.
2. $d := \text{ggT}(n, x_i - x_j)$.
3. Wenn $1 \leq d \leq n$, **return** ,,Faktor gefunden: d".
4. Wenn $d = n$, dann **return** ,,Mist!".
5. $x_i := f(x_i)$, $x_j := f(f(x_j))$.
6. Gehe zu 2.

Im Grunde handelt es sich hier um nichts anderes als einen Baby-Step-Giant-Step-Algorithmus mit effizienter Speicherverwaltung ($\mathcal{O}(1)$). Es gibt haufenweise Erweiterungen, die z.B. versuchen, den Schwanz des ρ schnell zu überspringen oder nach angemessener Zeit die Berechnungsvoschrift ändern oder ...

2.5 Dixon, QS, CFRAC

Der Standard-Trick, den fast alle moderneren Faktorisierungsverfahren benutzen, ist der folgende: Findet man zwei Zahlen x und y mit

$$x^2 \equiv y^2 \pmod{n} \quad \text{und} \quad x \not\equiv y \pmod{n},$$

dann liefert wegen $n \mid x^2 - y^2$ der $\mathrm{ggT}(n, x-y)$ einen nicht-trivialen Teiler von n.

Natürlich findet man nicht so einfach x, y mit obiger Eigenschaft. Aber man kann sie mit sub-exponentieller Laufzeit konstruieren, so dass n schließlich faktorisiert werden kann. Für Zahlen r, die kurz oberhalb von $\lfloor\sqrt{n}\rfloor$ liegen, berechnet man $f(r) = r^2 \bmod n$ und versucht, $f(r)$ über einer Basis von Primzahlen zu faktorisieren. Hat man genügend viele Paare $(r, f(r))$ gefunden, so dass $f(r)$ komplett über dieser Primzahlbasis faktorisiert werden kann, dann kann durch Linearkombination (Gauss modulo 2!) ein Paar (r', s') gebastelt werden, so dass $r'^2 \equiv s'^2 \pmod{n}$ gilt. Ist außerdem $r' \not\equiv \pm s'$ (Wahrscheinlichkeit: $\frac{1}{2}$), dann hat man einen nicht-trivialen Faktor von n gefunden.

Die erste Verbesserung des Verfahrens von Dixon, das oben beschrieben worden ist, kommt durch die Betrachtung quadratischer Reste (QR) zustande: Wenn $f(r)$ Vielfaches einer Primzahl p ist, also $f(r) \equiv 0$, dann muss wegen $f(r) = r^2 - n$ gelten:

$$n \equiv r^2 \pmod{n} \quad \Leftrightarrow \quad (\frac{n}{p}) = +1.$$

Es genügt also, solche Primzahlen in die Primzahlbasis aufzunehmen, modulo derer n ein quadratischer Rest ist. Modulo p hat n dann die zwei Wurzeln t und $-t$. Das lässt ein sehr effizientes Siebverfahren zu, da keine Probedivisionen mehr durchgeführt werden müssen, um die ganzen $f(r)$ zu in ihre Primfaktoren zu zerlegen, sondern wenn $f(r)$ ein Vielfaches von p ist, dann gilt das auch für $f(r+p-(r \bmod p))$, für $f(r+p)$, für $f(r+2p-(r \bmod p))$ usw.

Statt jetzt immer noch bei jedem $f(r), f(r+p), \ldots$ die Division durchzuführen und am Ende alle $f(r+kp)$ einzusammeln, die durch sukzessives Dividieren zu 1 geworden sind, kann man auch einfach Logarithmen subtrahieren (easy, was?!) und am Ende auf 0 checken. Im Wesentlichen war es das.

Der Morrison-Brillhart-Algorithmus arbeitet mit Kettenbrüchen (Continued Fraction, CFRAC), um eine Reihe von r zu konstruieren, so dass die $f(r)$ möglichst nahe bei $\sqrt{n}$ liegen. Das geht ganz gut, führt zu $f(r)$, die mit größerer Wahrscheinlichkeit komplett faktorisiert werden können, verhindert aber das Sieben. Asymptotisch nehmen sich CFRAC und QS nichts.

3 Diskrete Logarithmen

Beim Diskreten Logarithmus (DL) haben wir es im Allgemeinen mit folgendem Problem zu tun: Sei G eine zyklische Gruppe mit Erzeuger g und $a \in G$. Gesucht wird nach der eindeutig bestimmten Zahl $0 \le k \le |G|-1$ mit $g^k = a$, wobei g^k die k-fache Anwendung des Erzeugers g auf das neutrale Element bedeuten soll. Der naive Ansatz des Durchprobierens liefert eine exponentielle Laufzeit (linear in $|G|$), die natürlich inakzeptabel ist. Wir suchen nach besseren Verfahren.

3.1 Baby-Step-Giant-Step von Shanks

Das Baby-Step-Giant-Step-Verfahren nutzt aus, dass sich jedes $0 \le k \le n-1$ darstellen lässt als $k = m \cdot i + j$ mit $m = \lceil\sqrt{n}\rceil$, $i, j < m$. Nach Berechnung des m (wenn $n = |G|$ bekannt ist, sonst wird es schwierig!) berechnet man sich schnell $g^{i \cdot m}$ für alle $0 \le i < m$, also ungefähr $\sqrt{n}$ viele Gruppenelemente, definiert sich irgendeine Ordnung auf ihnen und sortiert sie (z.B. $\sqrt{n} \cdot ln(\sqrt{n})$ mit Heapsort). Dann wird für jedes $0 \le j < m$ das Element $a \cdot g^j$ berechnet (wieder ungefähr $\sqrt{n}$ viele Berechnungen) und in der Tabelle (binäre Suche!) nachgeschaut, ob es ein i gibt mit

$$a \cdot g^j = g^{m \cdot i}.$$

Ist das der Fall, dann folgt sofort:
$$a = g^{m \cdot i - j}.$$
Wir haben also: $log_g(a) = m \cdot i - j$. Zwar besser als die Probiermethode, aber immer noch mit exponentiellem Aufwand!

3.2 Pollard Rho (ρ)

Das Pollardsche Rho-Verfahren für diskrete Logarithmen ist seinem Bruder für die Faktorisierung sehr ähnlich. Wieder konstruieren wir uns eine Folge (x_i, b_i, c_i) mit
$$x_i = g^{b_i} \cdot a^{c_i}.$$
Finden wir schließlich ein Paar (x_i, x_j) mit $x_i = x_j$ und $i \neq j$, so gilt
$$g^{b_i} \cdot a^{c_i} = g^{b_j} \cdot a^{c_j} \quad \Leftrightarrow \quad a^{c_i - c_j} = g^{b_j - b_i}$$
und damit
$$a = g^{\frac{b_j - b_i}{c_i - c_j}}.$$
Um invertieren zu können, muss man allerdings die Gruppenordnung n kennen.

Ist die obige Folge möglichst zufällig (was auch immer das heißen mag), dann kann wieder das Geburtstagsprinzip ausgenutzt werden, und man findet mit hoher Wahrscheinlichkeit nach $\approx \sqrt{n}$ vielen Schritten eine Lösung.

3.3 Wenn die Gruppenordnung glatt ist: Silver-Pohlig-Hellman

Hat die Gruppenordnung n nur relativ kleine Primfaktoren (bzw. Primpotenzen), dann kann man, wenn man die Faktorisierung
$$n = \prod_k p_k^{e_k}$$
kennt, diskrete Logarithmen auch in den Untergruppen $G^{\frac{n}{q_k}}$ $(q_k = p_k^{e_k})$ lösen im Anschluss mit Hilfe des chinesischen Restsatzes den Logarithmus in der vollen Gruppe G bestimmen.

Im Detail geht das so: Für alle k bestimmt man
$$a^{\frac{n}{q_k}} = g^{x_k \cdot \frac{n}{q_k}}$$
mit dem naiven Verfahren, Baby-Step-Giant-Step oder Pollard ρ. Diese Logarithmen x_k sind modulo q_k eindeutig bestimmt, so dass danach der Chinese gestartet werden kann, um aus allen x_k das große $x \pmod n$ zusammenzubasteln.

3.4 Diskrete Logarithmen in $\mathbb{F}_p^*$: Index Calculus

Ein für große Gruppen sehr gutes Verfahren ist der Index-Kalkül (Index Calculus), der ähnlich funktioniert wie Dixon/QS beim Faktorisierungsproblem. Er kann jedoch nur für die Einheitengruppe $\mathbb{F}_p^*$ verwandt werden, weil er spezielle Eigenschaften der natürlichen Zahlen ausnutzt. Für elliptische Kurven gibt es nach derzeitigem Wissensstand kein besseres Verfahren als Pollard ρ.

Wie beim Quadratischen Sieb bauen wir uns auch hier zunächst eine Primzahlbasis, wobei jedoch jetzt nicht auf $(\frac{n}{p}) = 1$ oder Ähnliches geachtet werden muss.

Warum können wir den Index Calculus nicht auf beliebige Gruppen, z.B. elliptische Kurven, anwenden? Weil die Primfaktorzerlegung dort nicht so einfach ist wie in den natürlichen Zahlen. Dann wählen wir uns zufällig eine Horde $1 \leq b_i < n$, $(1 \leq i \leq m)$ berechnen g^{b_i} und die Primfaktorzerlegung
$$g^{b_i} = \prod_i p_i^{a_{ij}}.$$

Haben wir genügend Paare (b_i, g^{b_i}) mit zerlegbarem g^{b_i} gefunden, geht es weiter: Statt

$$\prod_{j=1}^{n} p_j^{a_{ij}} \equiv g^{b_i}, \ 1 \leq i \leq m,$$

schreiben wir $p_i \equiv g^{l_i}$ und erhalten das lineare Gleichungssystem

$$\sum_{j=1}^{n} a_{ij} l_j \equiv b_i, \ 1 \leq i \leq m,$$

über dem Ring $\mathbb{Z}/(p-1)\mathbb{Z}$, das sich wunderbar lösen lässt.

Wir haben jetzt also die Logarithmen $log_g(p_i)$ der Primzahlen in der Faktorbasis gefunden. Um den Logarithmus einer Zahl a zur Basis g zu bestimmen, muss nicht mehr viel getan werden: Wir wählen zufällig ein b, bis wir eine Relation

$$a \cdot g^b \equiv \prod_j p_j^{e_j}$$

erhalten, die Zahl $ag^b \bmod p$ also faktorisiert werden kann. Dann ist

$$ag^b \equiv g^{\sum_j e_j l_j} \quad \Leftrightarrow \quad log_g(a) \equiv \sum_j (e_j l_j) - b,$$

und wir haben den diskreten Logarithmus gefunden!

4 Diverse Kryptographieverfahren

4.1 Diffie-Hellman-Schlüsselaustausch

Zwei Teilnehmer, A und B, einigen sich auf eine große Primzahl p und eine Zahl $2 \leq g \leq p-2$. Wenn Sie einen Kommunikationskanal öffnen wollen, wählen sie jeweils eine zufällige, geheime Zahl: K_A und K_B. A schickt dann g^{K_A} an B, B schickt g^{K_B} an A. A berechnet nun

$$K := (g^{K_B})^{K_A} = (g^{K_A})^{K_B} = g^{K_A K_B},$$

B verfährt ebenso. K ist dann ihr gemeinsamer Schlüssel, den außer A und B niemand kennt und der z.B. als Session-Key für ein symmetrisches Verfahren eingesetzt werden kann.

Ohne Kenntnis von K_A bzw. K_B und ohne die Fähigkeit, diskrete Logarithmen zu berechnen, ist es i.A. nicht möglich, an den Schlüssel K heranzukommen.

4.2 ElGamal

4.2.1 Setup

Jeder Beteiligte wählt sich eine große Primzahl p und eine Zahl $2 \leq g \leq p-2$. Jeder Teilnehmer A wählt sich eine geheime (zufällige) Zahl $2 \leq d_A \leq p-2$, seinen privaten Schlüssel, und berechnet $e_A \equiv g^{d_A}$, seinen öffentlichen Schlüssel.

4.2.2 Ver- und Entschlüsselung

B besorgt sich den öffentlichen Schlüssel e_A von A und erzeugt eine Zufallszahl $1 \leq z \leq p-1$. Um die Nachricht m zu verschlüsseln, berechnet er $b \equiv g^z$ und $c \equiv m \cdot e_A^z$. Das Paar (b, c) schickt er an A.

A empfängt (b, c) und berechnet

$$b^{-d_A} \cdot c \equiv g^{-d_A z} \cdot m \cdot g^{d_A z} \equiv m.$$

Die Entschlüsselung funktioniert also.

4.2.3 Bemerkungen

Das Verfahren ist total von der Sicherheit des Zufallszahlengenerators abhängig. Eine Übertragung auf elliptische Kurven ist sehr einfach, da ausschließlich Gruppenoperationen verwendet werden.

4.3 RSA

4.3.1 Setup

Jeder Teilnehmer wählt sich zwei große Primzahlen p und q mit bestimmten Eigenschaften ($p-1$ nicht glatt, etc.), berechnet $n = pq$ und $\phi(n) = (p-1)(q-1)$. Dann wählt er ein zufälliges d, seinen privaten Schlüssel, und berechnet $e \equiv d^{-1} \pmod{\phi(n)}$, den öffentlichen Schlüssel.

4.3.2 Ver- und Entschlüsselung

Will jemand eine Nachricht an A schicken, besorgt er sich dessen öffentlichen Schlüssel e_A und den Modulus n_A. Eine Nachricht m_p wird dann verschlüsselt zu $m_c \equiv m_p^{e_A} \pmod{n}$. Will A die Nachricht wieder entschlüsseln, berechnet er einfach $m_c^{d_A} \equiv m_p^{e_A d_A} \equiv m \pmod{n}$.

4.4 Digitale Signatur mit ElGamal

4.4.1 Setup

Wie im Fall des ElGamal-Verschlüsselungsverfahrens.

4.4.2 Signieren einer Nachricht

A wählt sich eine Zufallszahl $0 < z < q$, $\mathrm{ggT}(z, p_A - 1) = 1$, und berechnet damit

$$r = g_A^z \bmod p_A.$$

Danach berechnet er

$$s \equiv \frac{1}{z}(h - e_A \cdot r),$$

wobei h der Hashwert der zu signierenden Nachricht ist. Die Signatur der Nachricht ist dann das 320 Bit lange Paar (r, s), $0 \leq r, s < q$.

4.4.3 Verifikation einer Signatur

B berechnet den Hashwert h der Nachricht, testet, ob $1 \leq r \leq p_A - 1$ ist und schließlich, ob $g_A^h \equiv f_A^r \cdot r^s \pmod{p_A}$ gilt. Ist das der Fall, wird die Signatur akzeptiert.

4.5 Digital Signature Algorithm (DSA)

DSA wurde im Januar 2000 vom NIST zum Standard für digitale Unterschriften erkoren. Ich finde das seltsam, weil es ohnehin schon genügend asymmetrische Verfahren gibt, die problemlos für Signaturschemata verwendet werden können.

4.5.1 Setup

Jeder Teilnehmer A wählt sich eine 160 Bit große Primzahl q und eine weitere, 512 bis 1024 Bit große Primzahl p mit $p \equiv 1 \pmod{q}$. A bestimmt einen Erzeuger der zyklischen Untergruppe der Ordnung q von $\mathbb{F}_p^*$: Ist g_0 mit $g_0^{\frac{p-1}{q}} \not\equiv 0, 1 \pmod{p}$, so liefert $g \equiv g_0^{\frac{p-1}{q}} \pmod{p}$ einen solchen Erzeuger.

Nun wählt A sich einen zufälligen privaten Schlüssel $1 < e_A < q$ und veröffentlicht q, p, g und $f_A \equiv g^{e_A}$.

4.5.2 Signieren einer Nachricht

A wählt sich eine Zufallszahl $0 < z < q$ und berechnet damit

$$r = (g^z \bmod p) \bmod q.$$

Danach berechnet er

$$s \equiv \frac{1}{z}(h + e_A \cdot r),$$

wobei h der Hashwert der zu signierenden Nachricht ist. Die Signatur der Nachricht ist dann das 320 Bit lange Paar (r, s), $0 \leq r, s < q$.

4.5.3 Verifikation einer Signatur

B besorgt sich den öffentlichen Schlüssel (q, p, g, f_A) und berechnet den Hashwert der Nachricht. Ist eine der Bedingungen $0 < r < q$, $0 < s < q$ verletzt, so wird die Signatur nicht akzeptiert (wichtig!). B berechnet nun

$$u_1 = s^{-1}h \bmod q \quad \text{und} \quad u_2 = s^{-1}r \bmod q$$

und schließlich

$$v = (g^{u_1} f_A^{u_2} \bmod p) \bmod q.$$

Gilt $v = r$, so wird die Unterschrift akzeptiert.

4.5.4 Bemerkungen

- Was passiert, wenn zwei Nachrichten mit der selben Zufallszahl z signiert werden? Wegen $s_i \equiv \frac{1}{z}(h + e_A r_i) \pmod q \Leftrightarrow h \equiv s_i z - e_A r_i \pmod q$ ergibt sich das lineare Gleichungssystem

  $$s_1 \cdot z - r \cdot e_A \equiv h_1 \pmod q, s_2 \cdot z - r \cdot e_A \equiv h_2 \pmod q.$$

  Mit Kenntnis von h_1, h_2 und s_1, s_1 lässt sich nun der private Schlüssel berechnen!

- Haben zwei Signaturen (r, s_1) und (r, s_2) die gleiche erste Komponente, so kann man daraus keinen Nutzen ziehen. Dies ist anders als bei der ElGamal-Signatur.

5 Kryptographie auf elliptischen Kurven

5.1 EC-DH

Diffie-Hellman lässt sich problemlos auf elliptische Kurven übertragen: Man wählt zunächst eine Primzahl p, eine elliptische Kurve über $\mathbb{F}_p$ und einen Punkt P auf der Kurve. A und B wählen sich dann je einen geheimen Schlüssel (natürliche Zahl) k_A und k_B. A schickt an B die Nachricht $k_A \cdot P$ und B an A die Nachricht $k_B \cdot P$.

Das Produkt $k_A k_B \cdot P$ kann dann – eventuell nach Umwandlung in eine andere Darstellung – als Session-Key für die Kommunikation zwischen A und B verwendet werden.

5.2 EC-ElGamal

Wie bereits beim ordinären $\mathbb{Z}_p$-ElGamal erwähnt, ist die Umstellung von ElGamal auf elliptische Kurven überhaupt kein Problem.

5.2.1 Setup

Man einigt sich auf eine Primzahl p und eine elliptische Kurve über $\mathbb{F}_p$. Statt des Generators g wählt man nun einen Punkt P auf der Kurve mit möglichst großer Ordnung. Ein Teilnehmer A wählt sich einen geheimen Schlüssel $k_A \in \mathbb{N}$ und berechnet seinen öffentlichen Schlüssel $Q_A = k_A \cdot P$.

5.2.2 Ver- und Entschlüsselung

Zum Verschlüsseln eines Texts muss dieser zunächst in Häppchen zerlegt werden. Jeder Happen wird nach einem vorher festgelegten Verfahren in einen Kurvenpunkt M umgewandelt. Der Sender B besorgt sich Q_A, den öffentlichen Schlüssel von A, und wählt sich eine Zufallszahl z. Er berechnet $B = z \cdot P$ und $C = M + z \cdot Q_A$. Das Paar (B, C) sendet er an A.

Will A den Text wieder entschlüsseln, muss er nur

$$C - k_A \cdot B = M + z \cdot Q_A - zk_A \cdot P = M + zk_A \cdot P - zk_A \cdot P = M$$

berechnen und den so gewonnenen Kurvenpunkt wieder in die Nachricht zurückumwandeln.

5.3 Menezes-Vanstone-Verfahren

Menezes-Vanstone ist cool, weil es keine Umwandlung der Nachricht in Kurvenpunkte erfordert. Diese Umwandlung ist in den meisten Fällen unsicher, speicherplatzfressend und daher schlecht. Hier können wir darauf verzichten.

5.3.1 Setup

Man einigt sich auf eine Primzahl p, eine elliptische Kurven über $\mathbb{F}_p$ und einen Punkt P auf der Kurve. Jeder Teilnehmer A erzeugt eine zufällige Zahl k_A, seinen privaten Schlüssel, und berechnet den öffentlichen Schlüssel $Q_A = k_A \cdot P$.

5.3.2 Ver- und Entschlüsselung

Der Sender B besorgt sich Q_A und übersetzt die Nachricht in eine Folge von Punkten (m_{i1}, m_{i2}). Diese Punkte müssen keine Kurvenpunkte sein! Für jeden dieser Punkte wählt er eine Zufallszahl z_i und berechnet nacheinander

- $R_i = (r_{i1}, r_{i2}) = z_i \cdot P$,
- $z_i \cdot Q_A = (s_{i1}, s_{i2})$ und
- $(t_{i1}, t_{i2}) = (m_{i1}s_{i1}, m_{i2}s_{i2}) \bmod p$.

Die verschlüsselte Nachricht ist dann das Tupel $(r_{i1}, r_{i2}, t_{i1}, t_{i2})$.

Will A die Nachricht entschlüsseln, berechnet er $S = (s_{i1}, s_{i2}) = k_A R_i$ und dann (wegen $\phi(p) = p - 1$ wissen wir, wie wir multiplikative Inverse berechnen) $M_i = (m_{i1}, m_{i2}) = (t_{i1} \cdot s_{i1}^{-1}, t_{i2} \cdot s_{i2}^{-1})$.

Easy? Denke ich auch!

5.4 EC-DSA

Wie auch in den bisherigen Fällen unterscheidet sich EC-DSA im Wesentlichen nicht von seinem $\mathbb{F}_p^*/\mathbb{F}_q^*$-Bruder.

5.4.1 Setup

Als systemweite Parameter wählt man eine elliptische Kurve über $\mathbb{F}_p$ und einen Kurvenpunkt P der Ordnung $q \approx p$ (q prim). Jeder Teilnehmer A wählt sich eine zufällige Zahl $1 < e_A < q - 1$, seinen privaten Schlüssel, und berechnet $Q_A = e_A \cdot P$. Der öffentliche Schlüssel von A ist Q_A.

5.4.2 Signieren einer Nachricht

A wählt eine Zufallszahl $1 < z < q - 1$ und berechnet $z \cdot P = (x_1, y_1)$ und $r \equiv x_1$ mit $0 < r \leq q - 1$. Dann berechnet er den Hashwert h der Nachricht und $s \equiv \frac{1}{z}(h + e_A \cdot r) \bmod q$. Die Signatur der Nachricht ist dann das Paar (r, s).

5.4.3 Verifikation einer Signatur

B überprüft, ob die Zahlen r und s aus der Signatur im Intervall $[1, q-1]$ liegen und berechnet nacheinander

$$u_1 \equiv s{-}1h \bmod q, \quad u_2 \equiv s1r \bmod q, \quad (x_0, y_0) = u_1 \cdot P + u_2 \cdot Q_A, \quad v \equiv x_0 \bmod q.$$

Gilt $v = r$, so wird die Signatur akzeptiert.