Elliptische Kurven und ihre Bedeutung in der Kryptographie

Andreas Eisler, 9926092

Studium: Wirtschaftsinformatik, E175

Betreuer:

Ao.Univ.Prof. Dr. Johann Wiesenbauer

Institut für Diskrete Mathematik und Geometrie

TU Wien

Inhaltsverzeichnis

Vorwort

Verschlüsselung

Seit der Antike ist es für verschiedene Personengruppen von großem Interesse Nachrichten an Verbündete und Partner zu senden, ohne fürchten zu müssen, dass ein Gegner die Nachricht abfängt und die darin enthaltenen Informationen zu seinem Vorteil nutzt. Während in der Antike und im Mittelalter das Versenden verschlüsselter Botschaften noch eine auf Militär und Politik beschränkte Notwendigkeit war, ist in den letzten beiden Jahrhunderten die Verschlüsselung auch für Unternehmen und Firmen zu einem unverzichtbaren Werkzeug geworden.

Im Zeitalter des Computers und der Telekommunikation gibt es kaum noch Möglichkeiten auf Verschlüsselung zu verzichten. Während das Abfangen und gezielte Durchforsten von Nachrichten noch vor 60 Jahren eine sehr aufwendige, in manchen Fällen fast unmögliche, Aufgabe war, ist es heute bereits einem begabten Studenten möglich, Datenströme von anderen Personen aufzuzeichnen. Sollte diese Person ihre Nachrichten nicht verschlüsseln und somit für einen Unbefugten unbrauchbar machen, könnte ihr dadurch großer Schaden entstehen.

Viele Firmen haben diese Lektion bereits auf schmerzliche Art und Weise gelernt. Aus einer einzigen durch Unachtsamkeit nicht verschlüsselten Nachricht kann ein Konkurrent sich ein Bild über die Strategie eines Gegners machen und diesen bei Verhandlungen um einen Auftrag ausstechen. Noch viel offensichtlicher ist die Gefahr, die durch Abfangen von Nachrichten entstehen kann im militärischen und politischen Bereich. Besonders deutlich kann dies am historischen Beispiel des deutsch- englischen „Chiffrierkrieges“ im zweiten Weltkrieg gezeigt werden. Bis heute ist es fraglich ob die Alliierten die Schlacht im Atlantik, ohne das Entschlüsseln der deutschen Marinecodes, für sich entschieden hätte.

Chiffrierung ist somit zu einem der wichtigsten Aspekte der modernen Kommunikation geworden. Über die Summen die von Firmen und Regierungen jährlich in Forschung und Praxis der Verschlüsselung investiert werden, kann nur spekuliert werden. Staatliche Sicherheitsdienste und private Unternehmen, die sich fast ausschließlich mit dem Abhören oder Sichern von fremdem, aber auch eigenem Nachrichtenverkehr, beschäftigen suchen laufend nach neuen, besseren und schnelleren Verfahren zur Verschlüsselung von Botschaften und haben somit in den letzen Jahrzehnten einen neuen schnell wachsenden Forschungszweig für Mathematiker, und ins besondere für Zahlentheoretiker, geschaffen.

Elliptische Kurven

Die Anwendungsbereiche von elliptischen Kurven sind sehr vielfältig. Sie werden nicht nur in der Kryptographie, sondern auch in vielen anderen Bereichen der Zahlentheorie verwendet. Obwohl sich diese Arbeit ausschließlich mit digitaler Verschlüsselung und Signatur von Nachrichten beschäftigt, wollen wir kurz die wichtigsten anderen Verwendungszwecke von elliptischen Kurven aufzählen:

- **Faktorisieren mit Hilfe von elliptischen Kurven:** Die Lestra-Faktorisierung, auch Elliptic Curve Factorisation Method (ECM) genannt, ist eine sehr effiziente Methode zur Faktorisierung.

- **Primzahlen- Tests mit elliptischen Kurven**: Das Elliptic Curve Primality Proving- Verfahren (ECPP) ist eine schnelles Verfahren, das eine Laufzeit von $O\left(\ln^{5+\varepsilon} n\right)$ hat.

Diese Arbeit wird sich jedoch nur mit der Bedeutung von elliptischen Kurven in der Kryptographie beschäftigen. Daher müssen diese Anwendungsbereiche, auch wenn sie indirekt ebenfalls in der Kryptographie verwendet werden, unbehandelt bleiben.

In den folgenden Kapiteln werden die Grundlagen zum Arbeiten mit elliptischen Kurven geschaffen. Mit dem CAS DERIVE werden Programme implementiert, welche es ermöglichen die theoretischen Grundlagen praktisch anzuwenden. Der Leser wird jedoch nicht nur einfache Methoden, sondern auch komplexe Verfahren wie den Schoof-Algorithmus zum Bestimmen der Ordnung einer elliptischen Kurve, oder den Pohlig-Hellman- Algorithmus, zum Lösen des Problems des diskreten Logarithmus für elliptische Kurven, finden. Aus dem Bereich der Kryptographie werden ausgewählte Verfahren vorgestellt, und danach auf ihre Anwendbarkeit bei elliptischen Kurven untersucht.

Am Ende dieser Arbeit werden klassische kryptographische Verfahren mit der Elliptic Curve Cryptography (ECC) verglichen. Es wird gezeigt, dass ECC bei gleicher Sicherheit der Verschlüsselung weniger große Schlüssel und weniger Rechenaufwand als die klassischen Verfahren, wie zum Beispiel RSA, benötigt. Genau diese Eigenschaft macht ECC heute besonders wichtig, da in der Telekommunikation kleine Geräte mit wenig Rechen- oder Speicherkapazität, ebenfalls ein entsprechend großes Maß an Sicherheit bieten müssen.

1 Mathematische Grundlagen zu elliptischen Kurven

Richten wir zu Beginn dieser Arbeit unser Augenmerk auf elliptische Kurven und ihre besonderen Eigenschaften. Dieses Kapitel soll dem Leser einen allgemeinen Überblick über elliptische Kurven verschaffen, um elliptische Kurven praktisch für Verschlüsselungsverfahren verwenden zu können.

1.1 Arten von Kurven

Eine elliptische Kurve sei wie folgt definiert:

Definition 1.1. Eine elliptische Kurve E über einem Körper *K* ist eine nicht-singuläre projektive ebene Kurve über *K*, die durch eine homogene Gleichung vom Grad 3 gegeben ist

$$y^2z + a_1xyz + a_2yz^2 = x^3 + a_3x^2z + a_4xz^2 + a_5z^3$$

wobei $a_1, a_2, a_3, a_4, a_5 \in K$.[1]

Als Nullelement fungiert der Punkt $O[\infty,\infty]$. Dieser Punkt wird auch als der **Punkt im Unendlichen** bezeichnet.

Elliptische Kurven können somit verschiedenste Gestalt annehmen. Beispiele für elliptische Kurven sind folgende:

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

$$y^2 + cy = x^3 + ax + b \qquad \text{oder}$$
$$y^2 + xy = x^3 + ax + b$$

$$y^2 = x^3 + ax^2 + bx + c$$

In Folge ist für uns besonders der $y^2 = x^3 + ax + b$ interessant. Besonders sei an dieser Stelle noch hervorgehoben, dass es für unsere späteren Anwendungen von großer Wichtigkeit ist, dass die Diskriminante $4a^3 + 27b^2$ nicht über dem Körper *K* verschwindet.

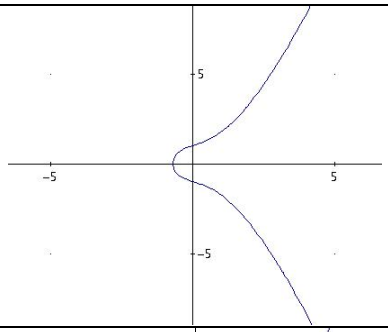
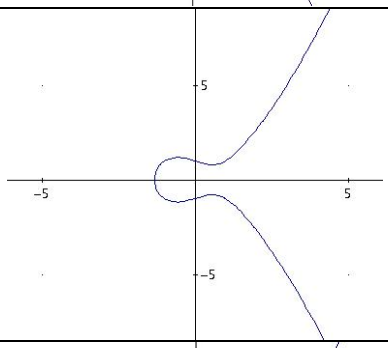
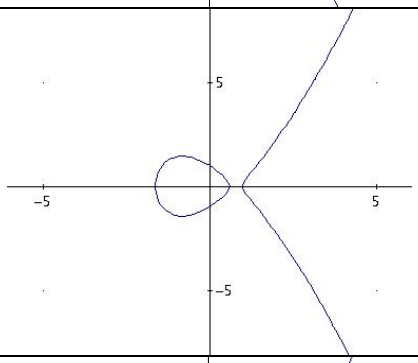
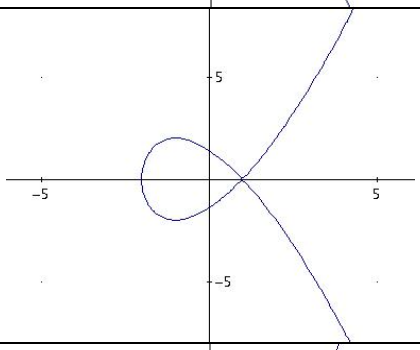
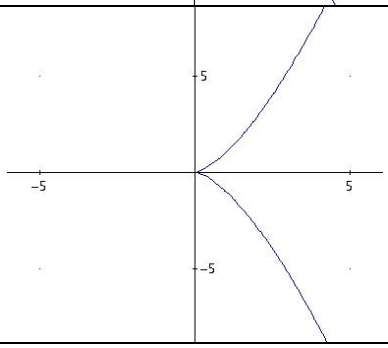
Mit einem kleinen DERIVE Programm können wir schnell überprüfen, ob eine von uns gewählte Kurve diese Bedingung erfüllt:

```
Reg?(a, b, p) := MOD(4·a^3 + 27·b^2 , p) ≠ 0

Reg?(1, 2, 7) = false
Reg?(1, 1, 7) = true
```

[1] Weierstraß- Gleichung laut Andreas Mirbach, Elliptische Kurven

Betrachten wir nun einige Kurven der Form $y^2 = x^3 + ax + b$.

$y^2=x^3+x+1$	
$y^2=x^3-x+1$	
$y^2=x^3-2x+1$	
$y^2=x^3-3x+2$	
$y^2=x^3$	

Man beachte, dass die letzten beiden Kurven nicht unsere Bedingung für die Diskriminante erfüllen. In beiden Fällen verschwindet die Diskriminante über *R*.

1.2 Addieren von Punkten elliptischer Kurven

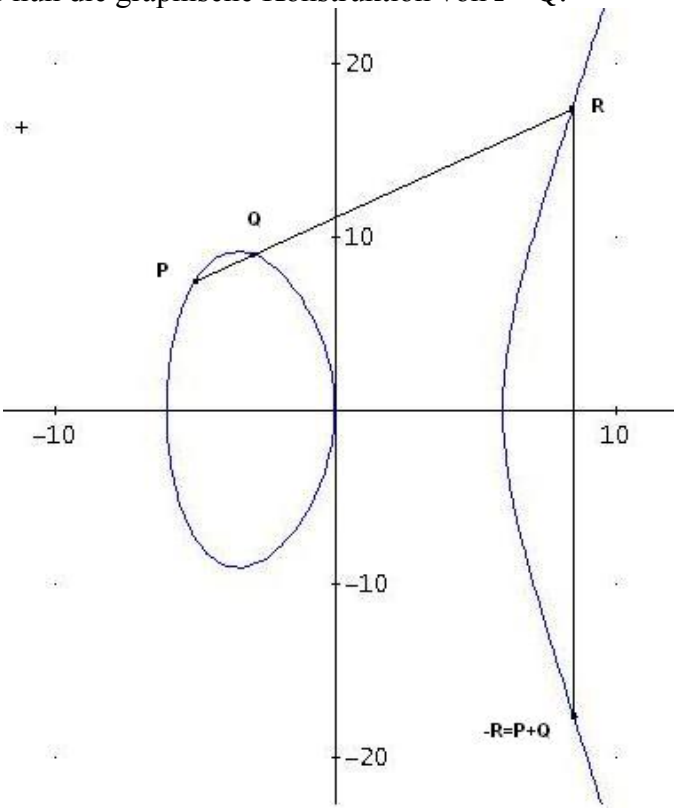
Bevor wir uns dem genauen Vorgang des Addierens zweier Punkte einer Elliptischen Kurve widmen, müssen wir einige Spezialfälle und Regeln festlegen.

Satz 1.2. Gegeben seien zwei Punkte P und Q, die beide auf der elliptischen Kurve E liegen. Für die Summe dieser beiden Punkte $P+Q$ gilt:

1. Sollte P identisch mit dem „unendlichweit entferntem Punkt“ O sein, so sei $P+Q$ gleich Q. Wir sehen hier, dass O als Nullelement unserer Menge von Punkten der elliptischen Kurve fungiert.
2. $-(x,y) = (x,-y)$, oder in Worten: $-P$ ist derselbe Punkt wie P, lediglich mit negativer y-Koordinate. Es ist somit ersichtlich, dass sobald (x,y) Punkt der Kurve ist, auch $(x,-y)$ Punkt der Kurve ist.
3. Sind P und Q zwei Punkte der Kurve E mit unterschiedlichen x-Koordinaten, so existiert eine Gerade l, die P mit Q verbindet, und die Kurve E in einem dritten Punkt R schneidet (ausgenommen l ist eine Tangente im Punkt P, was bedeuten würde dass $P = Q$ ist). Die Summe $P + Q$ sei nun definiert, als $-R$.
4. Sollte der Fall $Q = -P$ eintreten, so sei R definiert, als der „unendlich weit entfernte Punkt“ O. Als Formel: $P + Q = O$.
5. Gegeben der Fall $P = Q$, so ist l die Tangente an die Elliptische Kurve E im Punkt P, und R ist der einzige andere Schnittpunkt von l mit E. $P + Q$ sei wieder $-R$.

(siehe [Koblitz, S169])

Betrachten wir nun die graphische Konstruktion von $P+Q$.



Für eine elliptische Kurve *E* über einen Körper K, mit der Gleichung $y^2 = x^3 + ax + b$ lässt sich der Punkt *P*+*Q* folgendermaßen berechnen:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2$$

$$y_3 = -y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) * (x_1 - x_3)$$

Für die Berechnung von 2*P* erhalten wir folgende Gleichungen.

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1$$

$$y_3 = -y_1 + \left(\frac{3x_1^2 + a}{2y_1}\right) * (x_1 - x_3)$$

(Beweis siehe [Koblitz, S170])

1.3 Elliptische Kurven über F_p

Wie man leicht erkennen kann, führt das wiederholte Addieren von Punkten von elliptischen Kurven über R sehr schnell zu sehr großen Werten. Um sich immer in einem endlichen, und somit überschaubaren Bereich zu bewegen, wollen wir uns nun überlegen, wie sich elliptische Kurven über den endlichen Körper F_p verhalten.

Unter zu Hilfenahme selbst programmierter DERIVE Programme wollen wir nun die wichtigsten Rechenmethoden formalisieren.

Betrachten wir zu allererst die Anzahl der Punkte einer Kurve und überlegen, wie wir diese auflisten können. Die Anzahl der Punkte der Kurve ist von vorn herein stark eingegrenzt. Eine Kurve über den Körper F_p kann maximal $2p+1$ Punkte besitzen; q Punktepaare für jeden x-Wert 0,1,2,…,*p*-1 und den „unendlich weit entfernten Punkt“, der ja ebenfalls ein Punkt jeder elliptischen Kurve ist. Wir müssen uns jetzt nur noch überlegen welche dieser Punkte ganz- zahlige Wurzeln haben.

Mit einem kleinen DERIVE Programm ist es sehr einfach sich für bestimmte Kurven diese Punkte ausgeben zu lassen.

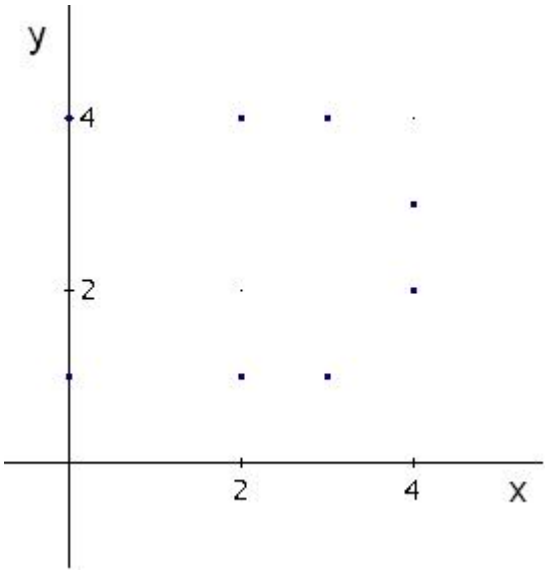
```
points(a, b, p, u_ := {}, x_ := 0, y_) :=
  Loop
    y_ := SQUARE_ROOT(x_^3 + a·x_ + b, p)
    If NUMBER?(y_)
        u_ := {[x_, y_], [x_, MOD(-y_, p)]} u u_
    x_ :+ 1
    If x_ = p
        RETURN ADJOIN([∞, ∞], SORT(u_))
```

Betrachten wir nun die Punkte der Kurve $y^2 = x^3 + x + 1$ in F_5

points(1, 1, 5)`=

$$\begin{bmatrix} \infty & 0 & 0 & 2 & 2 & 3 & 3 & 4 & 4 \\ \infty & 1 & 4 & 1 & 4 & 1 & 4 & 2 & 3 \end{bmatrix}$$

Zu besseren Übersicht tragen wir diese Punkte in ein Koordinatensystem ein.



Man kann erkennen, dass die Punkte immer paarweise auf einer x-Koordinate auftreten. Ein Punkt dieses Paares hat immer eine gerade und der andere eine ungerade y-Koordinate.

Satz 1.3. Ein Punkt *P(x,y)* einer elliptischen Kurve der Form $y^2 = x^3 + ax + b$ in F_p ist durch seine x-Koordinate und einem Wert g, mit $g \equiv y \bmod 2$, eindeutig definiert.

Wir wollen uns nun überlegen, wie wir ein Programm schreiben können, das Punkte auf einer elliptischen Kurve addiert:

```
add(u, v, a, p, k_) ≔
  Prog
    If FIRST(u) + FIRST(v) = ∞
      If FIRST(u) = ∞
        RETURN v
        RETURN u
    If FIRST(u) = FIRST(v)    MOD(u↓2 + v↓2, p) = 0
      RETURN [∞,∞]
    If u = v
      k_ ≔ MOD((3·FIRST(u)^2 + a)·INVERSE_MOD(2·u↓2, p), p)
      k_ ≔ MOD((v↓2 - u↓2)·INVERSE_MOD(FIRST(v) - FIRST(u), p), p)
    a ≔ k_^2 - FIRST(u) - FIRST(v)
    MOD([a, - u↓2 + k_·(FIRST(u) - a)], p)
```

Dieses kleine Programm deckt sämtliche Spezialfälle (welche in Satz 1.1 aufgelistet wurden) ab, und beinhaltet sowohl die Implementierung von $P+Q$ als auch die von $2 \cdot P$.

Führen wir nun ein kleines Experiment mit unserem neu entwickelten Programm aus. Wir konstruieren eine Additionstabelle für unsere bekannte Kurve $y^2 = x^3 + x + 1$. Das schaffen wir sehr einfach und elegant mit einem VECTOR Befehl:

VECTOR(VECTOR(add(u, v, 1, 5), u, points(1, 1, 5)), v, points(1, 1, 5))

Optisch ein wenig verschönert sieht das Ergebnis folgendermaßen aus:

+	0 1	0 4	2 1	2 4	3 1	3 4	4 2	4 3
0 1	4 2	∞ ∞	3 4	4 3	2 4	3 1	2 1	0 4
0 4	∞ ∞	4 3	4 2	3 1	3 4	2 1	0 1	2 4
2 1	3 4	4 2	2 4	∞ ∞	0 4	4 3	3 1	0 1
2 4	4 3	3 1	∞ ∞	2 1	4 2	0 1	0 4	3 4
3 1	2 4	3 4	0 4	4 2	0 1	∞ ∞	4 3	2 1
3 4	3 1	2 1	4 3	0 1	∞ ∞	0 4	2 4	4 2
4 2	2 1	0 1	3 1	0 4	4 3	2 4	3 4	∞ ∞
4 3	0 4	2 4	0 1	3 4	2 1	4 2	∞ ∞	3 1

Mit Hilfe dieser Tabelle können wir alle verschiedenen Punkte unserer elliptischen Kurve zueinander addieren. Hierbei stoßen wir auf eine weitere wichtige Eigenschaft von elliptischen Kurven. Addieren wir den Punkt $P(0,1)$ immer wieder, erhalten wir so alle anderen Punkte der elliptischen Kurve.

ITERATES(add(u, [0, 1], 1, 5), u, [0, 1])`=

$$\begin{bmatrix} 0 & 4 & 2 & 3 & 3 & 2 & 4 & 0 & \infty & 0 \\ 1 & 2 & 1 & 4 & 1 & 4 & 3 & 4 & \infty & 1 \end{bmatrix}$$

Satz 1.4. Die Punkte einer elliptischen Kurve über F_q bilden eine abelsche Gruppe. Diese abelsche Gruppe ist nicht notwendigerweise zyklisch. Man kann jedoch zeigen, dass sie immer das Produkt von zwei zyklischen Gruppen ist.
(vgl. [Koblitz, S174])

Wir verfügen somit über eine funktionierende Methode zur Addition von Punkten einer elliptischen Kurve. Nun wollen wir uns überlegen, wie wir aufbauend auf dieser, Punkte miteinander multiplizieren können.
Ausgehend von unserer Methode add(), mit der wir Punkte addieren und somit auch verdoppeln können, konstruieren wir eine Methode mult(), mit der wir eine Multiplikation in Verdopplungen und Additionen zerlegen, und diese an add() übergeben.

Beispiel: Um 23P zu berechnen, zerlegen wir die Multiplikation in 23P=P+2(P+2(P+2(2P))). Wir müssen also nur mehr 4 Verdopplungen und 3 Additionen durchführen.

Ausprogrammiert in DERIVE hat unsere mult() Methode nun folgendes Aussehen:

```
mult(u, n, a, p, s_ := [∞,∞]) :=
   If n < 0
      mult([u↓1, - u↓2], -n, a, p)
      Loop
         If n = 0
            RETURN s_
         If ODD?(n)
            s_ := add(s_, u, a, p)
         u := add(u, u, a, p)
         n := FLOOR(n, 2)
```

(siehe [Wiesenbauer2004, S48])

Diese Methode ist nicht nur sehr einfach, sondern auch extrem schnell im Berechnen der Werte. Um dies zu verdeutlichen nun einige Berechnungen und deren Laufzeiten.

$mult([0, 1], 10^{100}, 1, 5) = [0, 1]$ 0.062 sec

$mult([0, 1], 10^{1000}, 1, 5) = [0, 1]$ 6.92 sec

$mult([0, 1], 10^{100000}, 1, 5) = [0, 1]$ 108.3 sec

Wir sehen, dass wir Multiplikationen mit bis zu 100.000 -stelligen Multiplikatoren in einer annehmbaren Zeit berechnen können! Aufbauend auf dieser sehr schnellen Methode können wir nun erste Versuche unternehmen, EC für Verschlüsselungsverfahren zu verwenden.

2 Berechnung der Ordnung einer elliptischen Kurve

Für die weitere Vorgehensweise ist besonders eine Eigenschaft elliptischer Kurven von besonders großer Wichtigkeit: ihre Ordnung. Um effizient mit elliptischen Kurven arbeiten zu können, müssen wir über schnelle Methoden ihre Ordnung, also die Anzahl der Punkte der Kurve, zu berechnen verfügen. Im letzten Kapitel wurde dies lediglich oberflächlich behandelt.
Im Laufe dieses Kapitels werden Schritt für Schritt mehrere, immer effizientere Methoden vorgestellt dieses Problem zu lösen. Beginnen wir gleich mit dem einfachsten denkbaren Ansatz.

2.1 Triviale Suche

Im vorigen Kapitel haben wir die points(a,b,p) Methode vorgestellt. Diese lieferte uns einen Vektor mit allen Punkten einer elliptischen Kurve als Rückgabewert. Mit folgender Methode haben wir somit einen ersten sehr einfachen Lösungsansatz gefunden.

```
order(a, b, p) := DIM(points(a, b, p))
```

Diese Methode ist jedoch noch ungemein langsam und keinesfalls auf Probleme realistischer Größe anwendbar.

order(1, 1, 181) = 190	0,36sec
order(1, 1, 1201) = 1262	26,1sec

Wir müssen uns daher eine bessere Methode zur Berechnung der Ordnung suchen.

2.2 Das Eulersche Kriterium

Wir wollen nun die Theorie von den quadratischen Resten zu Hilfe nehmen, und nun nicht mehr einfach sämtliche Punkte der elliptischen Kurve berechnen und dann abzählen, wie viele wir gefunden haben.

Satz 2.1: Eulersches Kriterium: Für alle Primzahlen p und Zahlen $a \in \{1,...., p-1\}$ gilt:

$$a^{\frac{p-1}{2}} \bmod p = \left(\frac{a}{p}\right)$$

wobei $\left(\frac{a}{p}\right)$ das Legendre Symbol ist.

Ersetzen wir nun *a* durch unsere Kurvengleichung, so ist die Menge der möglichen Lösungen von $y^2 \equiv x^3 + ax + b \mod p$ gegeben durch $1+\left(\left(x^3+ax+b\right)^{\frac{p-1}{2}} \bmod p\right)$.

Als DERIVE- Programm sieht unsere neue order- Methode folgendermaßen aus:

```
order(a, b, p) := p + 1 + ∑(MODS((x^3 + a·x + b)^((p-1)/2) , p), x, 0, p - 1, 1)
```

(siehe [Wiesenbauer2005, S47])

Betrachten wir nun, wie viel Laufzeit wir durch unsere Verbesserungen gewonnen haben.

order(1, 1, 181) = 190	0,001sec
order(1, 1, 1201) = 1262	0,032sec

Für größere Probleme ist diese Methode aber immer noch zu langsam. Bereits bei *p* der Größe 10^5 benötigen wir schon wieder mehrere Sekunden zur Berechnung der Ordnung.

order(1, 1, 100003) = 100181	6,34sec

Betrachten wir als nächstes die Theorie zu elliptischen Kurven etwas genauer, um mit ihrer Hilfe eine neue Methode zu finden.

2.3 Babystep- Giantstep Methode

Betrachten wir nun eine Methode die direkt aus der Theorie über elliptische Kurven resultiert.

Um diese Methode zu entwickeln benötigen wir:

Satz 2.2. Hasse-Theorem: Sei *N* die Anzahl der Punkte auf einer elliptischen Kurve *E* über F_q, dann gilt

$$|N-(q+1)| \le 2\sqrt{q}$$

(vgl. [Koblitz, S174 f])

Legen wir das Theorem auf unser Problem um, so sehen wir, dass es, für einen beliebigen Punkt *P* der Kurve, in einem $2\sqrt{p}$ Umkreis von *(p+1)P* ein *xP = O* geben **muss.** Dieses erstbeste *x*, das wir finden, ist jedoch nicht zwingend identisch mit der Ordnung des Punktes *P*. Wir müssen daher noch so oft wie möglich *xP* um Primfaktoren p_i reduzieren, so lange $\frac{x}{p_i}P = O$ gilt.

Wir wissen außerdem, dass in einer endlichen Gruppe die Ordnung eines Elementes immer ein Teiler der Ordnung der Gruppe sein muss. Finden wir also für einen beliebigen Punkt *P* seine Ordnung, so ist die Ordnung der gesamten Kurve ein Vielfaches der Ordnung des Punktes *P*. Ist die Ordnung des Punktes *P* groß genug, so ist seine Ordnung identisch mit der Ordnung der gesamten Kurve. Die Frage ist nun nur noch, wann die Ordnung des Punktes

groß genug ist. Diese Frage ist ganz einfach zu beantworten, denn wenn die Ordnung von *P* größer oder gleich der Länge des Hasse- Intervalls, also $4\sqrt{p}$ ist, so ist *xP* die einzige Möglichkeit für die Ordnung der Kurve.

Wir müssen also in einem relativ kleinen Bereich um *p*+1 einen Multiplikator *x* finden, welcher *xP* = *O* erfüllt. Dann müssen wir gegebenfalls noch aus diesem *x* Primfaktoren herausheben, und dann überprüfen, ob die Ordnung des Punktes groß genug ist, um sie auch als Ordnung der Kurve zu identifizieren.

Das Problem, eine Methode order() zu entwickeln, welche auf dem Hasse Theorem aufbaut, gliedert sich also in zwei Unterprobleme. Für das erste Problem wählen wir die **Babystep- Giantstep Methode** von Shanks: Sei *r* die kleinste ganze Zahl $\geq p^{1/4}$. Man berechnet man zunächst alle *sP* mit *s = (p+1)+2kr* für $|k| \leq r$. Danach berechet man der Reihe nach $\pm vP$ für *v* = 0, 1,…., *r* bis man auf eine Gleichung $\pm vP = (p+1+2kr)P$ stößt. Dann ist $m = p+1+2kr \mp v$ ein Vielfaches der Ordnung von *P*. (vgl. [Förster, S170f])

```
BSGS1(u, a, p, j_, k_ := 0, r_, s_, t_, v_, x_) :=
   Prog
      r_ := CEILING(p^(1/4))
      s_ := mult(u, p + 1, a, p)
      t_ := mult(u, 2·r_, a, p)
      v_ := ITERATES(add(u, w_, a, p), w_, u, r_)
      x_ := v_ COL 1
      Loop
         If MEMBER?(FIRST(s_), x_)
            Prog
               j_ := POSITION(FIRST(s_), x_)
               If s_ = ELEMENT(v_, j_)
                  j_ := -j_
               RETURN p + 1 + 2·k_·r_ + j_
         s_ := add(s_, t_, a, p)
         k_ := k_ + 1
```

Nun reduzieren wir das gefundene *xP* so weit wie möglich um die Ordnung des Punktes zu erhalten.

```
ordP(u, a, p, bg_, f_, i_) :=
   Prog
      bg_ := BSGS1(u, a, p)
      f_ := FACTORS(bg_)
      Loop
         If f_ = [ ]
            RETURN bg_
         i_ := bg_/FIRST(FIRST(f_))
         If INTEGER?(i_) mult(u, i_, a, p) = [∞, ∞]
            bg_ := i_
            f_ := REST(f_)
```

Mit diesen beiden recht einfachen Methoden können wir jetzt die Ordnung eines Punktes berechnen. Stellen wir jetzt unsere neue und fortgeschrittenste order()- Methode zusammen. Wir verwenden dazu alle bis jetzt vorgekommenen Ideen und Theorien.

```
order(a, b, p, g_ := 1, h_, s_ := 1, x_, y_) :=
  Prog
    If p ≤ 229
      RETURN p + 1 + ∑(MODS((x^3 + a·x + b)^((p - 1)/2), p), x, 0, p - 1, 1)
    Loop
      g_ :+ 1
      If JACOBI(g_, p) = -1 exit
    Loop
      Loop
        x_ := RANDOM(p)
        s_ := JACOBI(x_^3 + a·x_ + b, p)
        If s_ = ±1 exit
      h_ := g_^((1 - s_)/2)
      y_ := SQUARE_ROOT(h_·(x_^3 + a·x_ + b), p)
      h_ := ordP([MOD(h_·x_, p), MOD(h_·y_, p)], MOD(h_^2·a, p), p)
      If h_ > 4·√p
        RETURN (p + 1)·(1 - s_) + s_·CEILING(p + 1 - 2·√p, h_)·h_
```

(siehe [Wiesenbauer2005, S49])

Betrachten wir nun die Laufzeit dieser neuen order() Methode.

order(1, 1, 100003) = 100181 0,016sec

order(1, 1, NEXT_PRIME(10^8)) = 100016024 0.172sec

Leider hängt die Laufzeit dieser Methode davon ab, wie schnell wir einen passenden Punkt P finden. Haben wir Pech und finden erst nach einiger Zeit ein P dessen Ordnung groß genug ist, sieht die Laufzeit viel schlechter aus.

order(1, 1, NEXT_PRIME(10^8)) = 100016024 80,1sec

Wir sind also immer noch nicht am Ziel. Alle bis jetzt entwickelten Methoden haben eine exponentielle Laufzeit. Im nächsten Kapitel wollen wir nun eine Methode mit subexponentieller Laufzeit vorstellen.

3 Der Schoof- Algorithmus

Der Schoof- Algorithmus ist eine Methode, die Anzahl der Punkte einer elliptischen Kurve in einer subexponentiellen Laufzeit zu berechnen. Wegen seiner Bedeutung in der Praxis ist ihm und seiner Weiterentwicklung, dem SEA- Algorithmus, dieses gesamte Kapitel gewidmet.

3.1 Isogenie und Endomorphismen

Wir haben uns nun mit elliptischen Kurven über endlichen Körpern und ihren Punktegruppen beschäftigt. Bevor wir uns jedoch weiter in der Materie vertiefen, und beginnen, den Schoof- Algorithmus zu implementieren, müssen wir noch ein wichtiges Kapitel abhandeln. Im weiteren Verlauf sind so genannte Isogenien, zwischen zwei elliptischen Kurven E_1 und E_2, und weiters die Endomorphismen auf der Punktegruppe einer elliptischen Kurve E von großer Wichtigkeit.

Definition 3.1. Seien *E, E_1* und *E_2* jeweils elliptische Kurven über *F_p*. Ein Morphismus $\Phi : E_1 \to E_2$ mit $\Phi(O) = O$ heißt Isogenie. *E_1* und *E_2* heißen isogen, falls eine nicht-triviale Isogenie Φ zwischen *E_1* und *E_2* existiert, d.h. eine Isogenie Φ mit $\Phi(E_1) \neq \{O\}$. Eine Isogenie $\Phi : E \to E$ heißt auch Endomorphismus. Die Endomorphismen auf einer elliptischen Kurve *E* bilden den Endomorphismenring $End_{F_p}(E)$.
(vgl. [Mirbach, S19])

Für die Eigenschaften von Isogenien gilt des Weiteren:

Satz 3.2. Sei $\Phi : E_1 \to E_2$ eine Isogenie, dann gilt:

$$\Phi(P+Q) = \Phi(P) + \Phi(Q)$$

für alle $P, Q \in E_1$ (vgl. [Mirbach, S20])

Wir haben nun Isogenien und Morphismen definiert und auch ihre wichtigsten Eigenschaften dargelegt. Es ist somit an der Zeit die wichtigsten Beispiele für Isogenien und Morphismen vorzustellen. Bei Isogenien ist dies die Multiplikationsabbildung [l] einer elliptischen Kurve E auf sich selbst. Bei Morhismen ist es der Frobenius- Endomorphismus φ, den wir in Folge, der Einfachheit wegen, Frobenius nennen werden.

Definition 3.3. Seien $l, n \in N$, *p* prim, *E* eine elliptische Kurve über *F_p* und $P = (x, y) \in E$. Dann definiert

$$[l] : E \to E,\ P \to l * P$$

einen Endomorphismus auf *E*, wobei mit $l \cdot P$ die *l*-fache Addition des Punktes *P* auf *E* gemeint ist.

Weiters definiert die Abbildung

$$\varphi : E(F_p) \to E(F_p),\ (x,y) \to (x^p, y^p)$$

den **Frobenius** auf *E*. (vgl. [Mirbach, S20])

Betrachten wir nun den Frobenius etwas genauer. Im weiteren Verlauf wird für uns besonders eine Eigenschaft des Frobenius von großer Wichtigkeit sein.

Satz 3.4. Sei *E* eine elliptische Kurve über dem endlichen Körper F_p und $\varphi \in End_{F_p}(E)$ der Frobenius. Dann gilt im $End_{F_p}(E)$ die Relation

$$\varphi^2 - t\varphi + p = 0$$

wobei $|t| \le 2\sqrt{p}$ auch Spur des Frobenius genannt wird. (Beweis siehe [Mirbach, S24f])

3.2 Torsionspunkte und Divisionspolynome

Definition 3.5. Sei *E* eine elliptische Kurve über F_p der Form $E : y^2 = x^3 + ax + b$. Dann besteht die Menge der *l*-Torsionspunkte genau aus den Punkten $P \in E(\overline{F_p})$ deren Ordnung *l* ist. Die *l*-Torsionspunkte werden somit durch

$$E[l] = \{P \in E(\overline{F_p}) : [l]P = O\} \subseteq E(\overline{F_p})$$

gegeben.
Die Untergruppe von $E(\overline{F_p})$, welche die *l*-Torsionspunkte bilden, wird auch *l*-Torsionsgruppe genannt. (vgl. [Mirbach, S31])

Besonders wichtig wird in der Folge das *l*-te Divisionspolynom einer elliptischen Kurve sein.

Definition 3.6. Sei $E : y^2 = x^3 + ax + b$ eine elliptische Kurve über F_p. Dann wird das *l*-te Divisionspolynom $\psi_l(x,y) \in F_p[X,Y]$ auf rekursive Weise durch

$$\begin{aligned}
&\psi_{-1}(x,y) = -1,\\
&\psi_0(x,y) = 0,\\
&\psi_1(x,y) = 1,\\
&\psi_2(x,y) = 2y,\\
&\psi_3(x,y) = 3x^4 + 6ax^2 + 12bx - a^2,\\
&\psi_4(x,y) = 4y(x^6 + 5ax^4 + 20bx^3 - 5a^2x^2 - 4abx - 8b^2 - a^3)\\
&\psi_{2l}(x,y) = \frac{\psi_l(x,y)\left(\psi_{l+2}(x,y) * \psi_{l-1}^2(x,y) - \psi_{l-2}(x,y) * \psi_{l+1}^2(x,y)\right)}{2y}\\
&\psi_{2l+1}(x,y) = \psi_{l+2}(x,y) * \psi_l^3(x,y) - \psi_{l-1}(x,y) * \psi_{l+1}^3(x,y)
\end{aligned}$$

definiert. (vgl. [Mirbach, S32])

Satz 3.7. Sei E eine elliptische Kurve über F_p und $P = (x, y) \in E$. Dann gilt:

$$P \in E[l] \Leftrightarrow \psi_l(x, y) = 0$$

Beweis: [Elliptische Kurven, Mirbach S34]

Satz 3.8. Sei das *l*-te Divisionspolynom $\psi_l(x, y) \in F_p[X, Y]$ gegeben und sei mit $\psi_l' \in F_p[X, Y]/(y^2 - x^3 - ax - b)$ das Polynom bezeichnet, das entsteht, wenn in ψ_l alle y² durch die Kurvengleichung $E : y^2 = x^3 + ax + b$ ersetzt werden. Sei weiterhin das Polynom $\widetilde{\psi}_l$ definiert durch:

$$\widetilde{\psi}_l = \begin{cases} \dfrac{\psi_l'}{\psi_2} & \text{für gerade } l \\ \psi_l' & \text{für ungerade } l \end{cases}$$

.

Des weiteren gilt $\widetilde{\psi}_l \in F_p[X]$ und

$$\operatorname{ord} \widetilde{\psi}_l(x) = \begin{cases} \dfrac{l^2 - 1}{2} & \text{für ungerade } l \neq p \\ \dfrac{l^2 - 4}{2} & \text{für gerade } l \neq p \end{cases}$$

(vgl. [Mirbach, S35])

Mittels DERIVE lässt sich $\widetilde{\psi}_l$ folgendermaßen berechnen:

```
ψ(n, a, b, p, x, y, u_) :=
 Prog
  If n < 3
     RETURN n·y^MAX(0, n - 1)
  If n = 3
     RETURN POLY_MOD(3·x^4 + 6·a·x^2 + 12·b·x - a^2, p)
  If n = 4
     RETURN POLY_MOD((x^6 + 5·a·x^4 + 20·b·x^3 - 5·a^2·x^2 - 4·a·b·x - 8·b^2 - a^3)·4·y, p)
  If EVEN?(n)
     Prog
        u_ := ψ(n/2 + 2, a, b, p, x, y)·ψ(n/2 - 1, a, b, p, x, y)^2
        u_ :- ψ(n/2 - 2, a, b, p, x, y)·ψ(n/2 + 1, a, b, p, x, y)^2
        POLY_MOD(ψ(n/2, a, b, p, x, y)·u_·(p + 1)/(2·y), p)
     Prog
        u_ := ψ((n + 3)/2, a, b, p, x, y)·ψ((n - 1)/2, a, b, p, x, y)^3
        u_ :- ψ((n + 1)/2, a, b, p, x, y)^3·ψ((n - 3)/2, a, b, p, x, y)
        POLY_MOD(REMAINDER(u_, y^2 - x^3 - a·x - b, y), p)
```

(siehe [Wiesenbauer2007, S50])

Satz 3.9. Weiters gilt für jeden Punkt $P = (x, y) \in E(\overline{F}_p)$ und $l \in \mathrm{N}$ mit $[l]P \neq O$:

$$[l]P = \left(x - \frac{\psi_{l-1}\psi_{l+1}}{\psi_l^{\,2}}, \frac{\psi_{l+2}\psi_{l-1}^2 - \psi_{l-2}\psi_{l+1}^2}{4y\psi_l^3} \right)$$

(vgl. [Mirbach S36])

Das DERIVE- Programm um die x- Koordinate zu berechnen sieht folgendermaßen aus:

```
nPx(n, a, b, p, x, s_, t_, y_) :=
  Prog
    s_ := ψ(n - 1, a, b, p, x, y_)·ψ(n + 1, a, b, p, x, y_)
    If ODD?(n)
      s_ :* (x^3 + a·x + b)/y_^2
    t_ := ψ(n, a, b, p, x, y_)^2
    If EVEN?(n)
      t_ :* (x^3 + a·x + b)/y_^2
    POLY_MOD(x·t_ - s_, p)/POLY_MOD(t_, p)
```

Die Berechnung der y- Koordinate erfolgt äquivalent:

```
nPy(n, a, b, p, x, y, s_, t_) :=
  Prog
    s_ := ψ(n + 2, a, b, p, x, y)·ψ(n - 1, a, b, p, x, y)^2
    s_ := POLY_MOD(s_ - ψ(n - 2, a, b, p, x, y)·ψ(n + 1, a, b, p, x, y)^2, p)
    t_ := 4·y·ψ(n, a, b, p, x, y)^3
    If EVEN?(n)
      t_ :* (x^3 + a·x + b)/y^2
    s_/POLY_MOD(t_, p)
```

(vgl. [Wiesenbauer2007, S50])

Wir verfügen nun über die Mittel, mit Divisionspolynomen zu arbeiten, und können uns nun auf den Algorithmus konzentrieren.

3.3 Die Idee des Algorithmus

Wir wissen bereits, das nach dem Satz von Hasse die Anzahl der rationalen Punkte einer elliptischen Kurve der Form $E : y^2 = x^3 + ax + b$ über einen endlichen Körper F_p, in einem Intervall von $2\sqrt{p}$ um p+1 liegt. Oder anders geschrieben gilt für die Anzahl der Punkte auf E: $\#E(F_p) = p + 1 - t$ mit $|t| \leq 2\sqrt{p}$.

Mit dem Schoof- Algorithmus versucht man nun die Werte von t modulo l zu berechnen, wobei l kleine Primzahlen sind. Gelingt dies für alle l = 2,3,5,… bis das Produkt aller l größer als die Länge des Hasse Intervalls ist, so kann man t mit dem Chinesischen Restklassensatz eindeutig bestimmen.

Sei τ_l der Wert für den gilt: $\tau_l \equiv t \bmod l$, so sucht man mittels der Divisionspolynome ψ_l für ein festes l einen Wert $\tau = 0,...,l-1$ der die Gleichung des Frobenius $\varphi^2 - \tau\varphi + p = 0$ erfüllt. Gelingt dies, wie bereits oben beschrieben wurde, für eine ausreichende Anzahl von verschiedenen *l*, so können wir die Anzahl der Punkte berechnen.

Der Spezialfall $l = 2$

Für den Fall $l = 2$ gibt es nur 2 Möglichkeiten; entweder $\#E(F_p)$ mod 2 ist 1 oder 0. Wir können diesen einfachen Sonderfall getrennt von den anderen Fällen betrachten. Wir nehmen zur Bestimmung dieses Falls folgenden Satz zu Hilfe.

Satz 3.10. Es gilt die Relation $\#E(F_p) \equiv 0 \bmod 2$ genau dann, wenn die Polynome $x^3 + ax + b$ und $x^p - x$ in $F_p[X]$ nicht teilerfremd sind.
(Beweis: siehe [Mirbach S37])

Berechnung für $l \geq 3$

Wir wollen uns nun überlegen, auf welche Art wir τ_l für ein festes *l* mit $l \geq 3$, *l* prim und $l \neq p$ bestimmen können. Um die theoretischen Grundlagen zur Berechnung dieser τ_l zu schaffen benötigen wir folgenden Satz.

Satz 3.11. Sei $E[l]$ die *l*-Torsionsgruppe und φ_l der auf $E[l]$ eingeschränkte Frobenius. Dann erfüllt φ_l für alle $P \in E[l]$ die Gleichung

$$\left(\varphi_l^2 - \tau_l \varphi_l + p\right) \cdot (P) = O$$

wobei $\tau_l \equiv t \bmod l$ ist. (siehe [Mirbach S38])

Aus Satz 3.11. und Satz 3.9. kann man nun direkt folgenden Satz herleiten:

Satz 3.12. Die Gleichung aus Satz X.7. ist genau dann für ein $l \geq 3$ und einen Punkt $P = (x, y) \in E(l)^x$ erfüllt, wenn gilt:

$$\left(x^{p^2}, y^{p^2}\right) + \left(x - \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^{\ 2}}, \frac{\psi_{p_l+2}\psi_{p_l-1}^2 - \psi_{p_l-2}\psi_{p_l+1}^2}{4y\psi_{p_l}^3} \right)$$

$$= \begin{cases} O & \text{falls } \tau_l = 0 \\ (\tilde{x}, \tilde{y}) & \text{falls } \tau_l \neq 0 \end{cases}$$

wobei gilt

$$\tilde{x} = x^p - \left(\frac{\psi_{\tau_l - 1} \psi_{\tau_l + 1}}{\psi_{\tau_l}^{\ 2}} \right)^p$$

und

$$\tilde{y} = \left(\frac{\psi_{\tau_l + 2} \psi_{\tau_l - 1}^2 - \psi_{\tau_l - 2} \psi_{\tau_l + 1}^2}{4y \psi_{\tau_l}^3} \right)^p$$

und

$$p_l = p \bmod l$$

(Herleitung vgl. [Mirbach S40])

Um nun ein passendes $\tau_l \equiv t \bmod l$ für ein festes l zu finden, müssen wir nun überprüfen für welche τ aus $\tau \in \{0,1,...,l-1\}$ die Gleichung aus Satz 3.12. erfüllt wird. Wir unterscheiden daher ab nun, für $l \geq 3$, immer folgende Fälle: $\varphi_l^2(P) = \pm p_l(P)$ (Fall A) und $\varphi_l^2(P) + p_l(P) = \tau_l \varphi_l(P)$ (Fall B). Wir müssen daher zu erst untersuchen, welcher der beiden Fälle vorliegt.

Betrachten wir die Gleichung für die x-Koordinate aus Satz 3.12 für den Fall A, so erhalten wir:

$$x^{p^2} = x - \frac{\psi_{p_l - 1} \psi_{p_l + 1}}{\psi_{p_l}^{\ 2}}$$

Umgeformt erhalten wir:

$$\left(x^{p^2} - x \right) \psi_{p_l}^{\ 2} + \psi_{p_l - 1} \psi_{p_l + 1} = 0$$

Ersetzen wir nun in dieser Gleichung alle y² durch die Kurvengleichung, so erhalten wir die Gleichung g(x).

Ist nun der Grad des ggT(g(x), $\psi_l(x)$) größer als 0, so befinden wir uns im Fall A, denn es existiert mindestens ein Punkt P aus $E[l]^x$ für den gilt:

$$\varphi_l^2(P) = \pm p_l(P)$$

Ist der Grad des ggT(g(x), $\psi_l(x)$) gleich 0, so befinden wir uns im Fall $\tau_l \neq 0$.
(Beweis vgl. [Elliptische Kurven, Mirbach, S40f])

Fall A: $\varphi_l^2(P) = \pm p_l(P)$

Wir wissen, dass es einen Punkt P aus $E[l]^x$ geben **muss**, für den $\varphi_l^2(P) = \pm p_l(P)$ gilt. Wir müssen uns nun genau überlegen, ob wir uns in dem Unterfall $\varphi_l^2(P) = +p_l(P)$ oder $\varphi_l^2(P) = -p_l(P)$ befinden.

Der Fall $\varphi_l^2(P) = -p_l(P)$ ist sehr schnell überprüft. Sollte das Legendre- Symbol $\left(\frac{p_l}{l}\right) = -1$, also p_l kein Quadrat modulo l, sein, so befinden wir uns im Fall $\varphi_l^2(P) = -p_l(P)$. Wir können also an dieser Stelle sofort abbrechen und als Lösung $\tau_l = 0$ zurückgeben.

Nehmen wir aber einmal an, dass wir uns im Fall $\varphi_l^2(P) = +p_l(P)$ befinden. Dann ist p_l ein Quadrat modulo l, und wir können die Quadratwurzel von p_l modulo l mit $w \in Z$ und $0 < w < l$ bezeichnen. Setzen wir w nun in $\varphi_l^2(P) = +p_l(P)$ ein, so erhalten wir $\varphi_l^2(P) = w^2(P)$ und somit $\varphi_l(P) = \pm w(P)$.

Genau so, wie schon bei der letzten Überprüfung, setzen wir in die Gleichung für die x- Koordinate aus Satz 3.12 ein und erhalten somit die Gleichung h(x)=0:

$$\left(x^p - x\right)\psi_{p_l}{}^2 + \psi_{p_l-1}\psi_{p_l+1} = 0$$

Ist der Grad des ggT(h(x), $\psi_l(x)$) gleich 0 so befinden wir uns doch im Fall $\varphi_l^2(P) = -p_l(P)$. Ist der Grad jedoch größer als 0, so wissen wir, dass es ein P mit $\varphi_l(P) = \pm w(P)$ gibt. Wir müssen jetzt nur mehr überprüfen, ob wir uns im Fall $\varphi_l(P) = +w(P)$ oder im Fall $\varphi_l(P) = -w(P)$ befinden.

Dies können wir sehr schnell schaffen, indem wir nun in die Gleichung für die y-Koordinate aus Satz 3.12 einsetzen. Wir erhalten somit für $\varphi_l(P) = +w(P)$ die Gleichung:

$$y^p - \frac{\psi_{w+2}\psi_{w-1}^2 - \psi_{w-2}\psi_{w+1}^2}{4y\psi_w^3} = 0$$

Ersetzen wir nun wieder alle y² durch die Gleichung der Elliptischen Kurve, so erhalten wir die Gleichung k(x,y)=0. Ist der Grad des ggT(k(x,y), $\psi_l(x)$) nun größer als 1, so ist $\varphi_l(P) = +w(P)$ die richtige Lösung. Wir erhalten:

$$\varphi_l(P) = \frac{2p_l}{\tau_l} \cdot P = \frac{2w^2}{\tau_l} \cdot P,$$

und somit $\tau_l \equiv 2w \bmod l$.

Äquivalent gilt für den Fall $\varphi_l(P) = -w(P)$ die Relation $\tau_l \equiv -2w \bmod l$. (vgl. [Mirbach, S42f])

Fall B: $\varphi_l^2(P) + p_l(P) = \tau_l \varphi_l(P)$

Ist nun der Grad des ggT(g(x), $\psi_l(x)$) gleich 0 so befinden wir uns im Fall B. Wir wissen, dass $\tau_l \neq 0$ und müssen nun ermitteln, welches τ_l aus (1,…, *l*-1) die Gleichung aus Satz 3.12 erfüllt. Dazu überprüfen wir für jedes τ_l , beginnend mit τ_l=1, die Gleichung der x-Koordinate:

$$x^{p^2} + x - \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^{\,2}} = x^p - \left(\frac{\psi_{\tau_l-1}\psi_{\tau_l+1}}{\psi_{\tau_l}^{\,2}} \right)^p$$

Wir erhalten durch Umformen und Ersetzen der y² durch die Gleichung der elliptischen Kurve die Gleichung $h_x(x) = 0$.

Ist nun der Grad des ggT($h_x(x), \psi_l(x)$) größer als 0, so überprüfen wir noch die Gleichung der y-Koordinate aus Satz 3.12:

$$y^{p^2} + \frac{\psi_{p_l+2}\psi_{p_l-1}^2 - \psi_{p_l-2}\psi_{p_l+1}^2}{4y\psi_{p_l}^3} = \left(\frac{\psi_{\tau_l+2}\psi_{\tau_l-1}^2 - \psi_{\tau_l-2}\psi_{\tau_l+1}^2}{4y\psi_{\tau_l}^3} \right)^p$$

Wieder erhalten wir durch Umformen und Ersetzen der y² die Gleichung $h_y(x) = 0$.

Gilt auch hier, dass der Grad des ggT($h_y(x), \psi_l(x)$) größer als 0 ist, so haben wir das passende τ_l gefunden. Sollte der Grad des ggT gleich 0 sein, so müssen wir weitersuchen. (vgl. [Mirbach, S44f])

Wir verfügen nun über sämtliche theoretischen Grundlagen, den Schoof- Algorithmus zu implementieren. Nun, endlich, können wir uns ganz dem eigentlichen Algorithmus widmen, und diesen implementieren.

3.4 Der Algorithmus

Wir haben in den vorangegangenen Abschnitten sämtliche Grundlagen, die notwendig sind, um den Schoof- Algorithmus zu implementieren, geschaffen. Es wurde genau erklärt, wie der Algorithmus nun im Detail funktioniert, aber bevor wir beginnen können eine Version des Algorithmus in DERIVE zu programmieren, müssen wir noch ein paar kleine Hilfsfunktionen erstellen.

Es hat sich im Laufe des Entwickelns des Schoof- Algorithmus gezeigt, dass die von DERIVE zur Verfügung gestellten Methoden zu langsam sind, um mit großen Termen zu arbeiten. Daher müssen wir auf einige kleine Hilfsroutinen zurückgreifen, mit denen wir große Terme multiplizieren und potenzierten können[2]. Beginnen wir mit zwei sehr einfachen Methoden, die ein Polynom in einen Vektor, beziehungsweise einen Vektor zurück in ein Polynom konvertieren.

```
polytovec(u, p, x) :=
  Prog
    u := POLY_MOD(u, p)
    If u = 0
      RETURN []
    u := MOD(SUBST(TERMS(u + p·(x^POLY_DEGREE(u) - 1)/(x - 1)), x, 1), p)
    Loop
      If FIRST(u) > 0
        RETURN u
      u := REST(u)


vectopoly(u, x) := u·VECTOR(x^j_, j_, DIM(u) - 1, 0, -1)
```

Weiters eine Methode, um zwei Vektoren miteinander zu multiplizieren.

```
vecmult(u, v, p, s_) :=
  Prog
    If DIM(u) < DIM(v)
      [s_ := u, u := v, v := s_]
    s_ := 0·u
    v := REVERSE(v)
    Loop
      If v = [] exit
      s_ := ADJOIN(0, s_ + FIRST(v)·u)
      u := APPEND(u, [0])
      v := REST(v)
    s_ := MOD(s_, p)
    Loop
      s_ := REST(s_)
      If FIRST(s_) ≠ 0
        RETURN s_
```

[2] Sämtliche grundlegenden Methoden zum Rechnen mit Vektoren wurden freundlicherweise vom Betreuer dieser Arbeit, Ao.Univ.Prof. Dr. Johann Wiesenbauer, zur Verfügung gestellt.

Im Laufe der Berechnung der einzelnen Gleichungen des Schoof- Algorithmus müssen wir die einzelnen Polynome immer um das jeweilige Divisionspolynom $\psi_l(x)$ reduzieren, um sicher zu stellen, dass die einzelnen Ausdrücke nicht zu groß werden. Wir benötigen daher auch eine Methode zum Bilden des Rests einer Division von zwei Vektoren.

```
vecrem(u, v, p, k_ := 0) :=
  Prog
    If DIM(u) < DIM(v)
      RETURN u
    Loop
      If DIM(v) = DIM(u) exit
      k_ :+ 1
      v := APPEND(v, [0])
    Loop
      u := MOD(u - FIRST(u)·v, p)
      u := REST(u)
      k_ :- 1
      If k_ < 0 exit
        v := DELETE(v, -1)
    Loop
      If FIRST(u) > 0
        RETURN u
        0
        RETURN u
     u := REST(u)
```

Außerdem benötigen wir noch eine Methode zur Berechnung des ggT von zwei Vektoren.

```
polygcd(u, v, p, x, k_ := 0, t_, v_) :=
  Prog
    u := polytovec(u, p, x)
    v := polytovec(v, p, x)
    If DIM(u) < DIM(v)
      [t_ := u, u := v, v := t_]
    Loop
      If v = []
        RETURN vectopoly(u)
      t_ := INVERSE_MOD(FIRST(v), p)
      v := MOD(t_·v, p)
      Loop
        If DIM(v) < DIM(u)
          Prog
            k_ :+ 1
            v := APPEND(v, [0])
          exit
      t_ := u
    Loop
      If FIRST(t_) > 0
        t_ := MOD(t_ - FIRST(t_)·v, p)
      t_ := REST(t_)
      If k_ = 0 exit
      v := DELETE(v, -1)
      k_ :- 1
    u := v
    v := t_
      Loop
        If v = []
          exit
          If FIRST(v) > 0 exit
        v := REST(v)
```

Da diese Methode extrem lang ist, wurde sie geteilt und die Teile nebeneinander dargestellt.

Wir verfügen nun über alle notwendigen Hilfsmethoden, um mit Hilfe von Vektoren Polynome zu multiplizieren, zu reduzieren und ihren ggT zu bilden. Wir verwenden nun genau diese Methoden um Polynome zu potenzieren und zu vereinfachen.

```
power(u, k, r, p, x, y, a := 0, b := 0, e_, u_ := [1]) :=
   Prog
      e_ := k·POLY_DEGREE(u, y)
      If e_ = k
         If ODD?(k)
            e_ := power(x^3 + a·x + b, (k - 1)/2, r, p, x)·y
            e_ := power(x^3 + a·x + b, k/2, r, p, x)
         e_ := 1
      u := SUBST(u, y, 1)
      u := polytovec(u, p, x)
      r := polytovec(r, p, x)
      Loop
         If k = 0
            RETURN vectopoly(u_)·e_
         If ODD?(k)
            u_ := vecrem(vecmult(u, u_, p), r, p)
         u := vecrem(vecmult(u, u, p), r, p)
         k := FLOOR(k, 2)
```

```
polymult(u, v, a, b, p, x, y, e_, f_) :=
  Prog
    e_ := POLY_DEGREE(u, y) + POLY_DEGREE(v, y)
    e_ := REMAINDER(y^e_, y^2 - x^3 - a·x - b)
    u := polytovec(SUBST(u, y, 1), p, x)
    v := polytovec(SUBST(v, y, 1), p, x)
    f_ := vecmult(u, v, p)
    POLY_MOD(vectopoly(f_)·e_, p)
```

```
polyrem(u, k, p, x, e_) :=
   Prog
      e_ := POLY_DEGREE(u, y)
      u := SUBST(u, y, 1)
      u := polytovec(u, p, x)
      k := polytovec(k, p, x)
      u := vecrem(u, k, p)
      vectopoly(u)·y^e_
```

```
reduce(u, a, b, p, r, x, y, e_) :=
   Prog
      u := POLY_MOD(REMAINDER(POLY_MOD(u, p), y^2 - x^3 - a·x - b, y), p)
      e_ := POLY_DEGREE(u, y)
      u := SUBST(u, y, 1)
      POLY_MOD(polyrem(u, r, p, x) ·y^e_, p)


polyreduce(u, p, n_, z_, m_ := 1) :=
  Prog
   If POLY_DEGREE(NUMERATOR(u), y) > 0
     Prog
      u :/ y
      m_ := y
   If POLY_DEGREE(DENOMINATOR(u), y) > 0
     Prog
      u :* y
      m_ := 1/y
   n_ := polygcd(NUMERATOR(u), DENOMINATOR(u), p, x)
   z_ := polyquot(NUMERATOR(u), n_, p)
   n_ := polyquot(DENOMINATOR(u), n_, p)
   RETURN z_/n_·m_
```

Man beachte, dass bei diesen Methoden, die auf die Vektor- Methoden zurückgreifen, auch einzelne multiplikative y in den Polynomen vorkommen können. Dies ist für das einwandfreie Funktionieren des Schoof- Algorithmus unbedingt notwendig.

Die reduce()- Methode reduziert ein einzelnes Polynom und ersetzt alle y^2 durch die Gleichung der Kurve, wären polyreduce() einen Bruch in F_p kürzt.

Wegen seiner Länge wurde der Schoof- Algorithmus auf die nächsten beiden Seiten aufgeteilt.

Nun ist es endlich soweit, und wir können eine fertige Implementierung des Algorithmus vorstellen:

```
schoof(a, b, p, n := 0, p_, q_ := [2], e_, g_, h_, i_, j_, r_, s_, t_, τ := 0, u_, v_, w_, x_, y_, z_) :=
  Prog
    If n = 0
      Loop
        p_ := ∏(q_)
        If p_ > 4·√p
          Prog
            t_ := CRT(VECTOR(schoof(a, b, p, q__), q__, q_), q_)
            RETURN p + (p_ + 1)^IF(t_ > 2·√p) - t_
        q_ := ADJOIN(NEXT_PRIME(FIRST(q_)), q_)
    If n = 2
      Prog
        n := power(x, p, x^3 + a·x + b, p, x)
        RETURN 0^POLY_DEGREE(polygcd(n - x, x^3 + a·x + b, p, x), x)
    r_ := ψ(n, a, b, p, x)
    r_ := POLY_MOD(INVERSE_MOD(n, p)·r_, p)
    p_ := MOD(p, n)
    s_ := (power(x, p^2, r_, p, x) - x)·ψ(p_, a, b, p, x, y)^2
    s_ :+ ψ(p_ - 1, a, b, p, x, y)·ψ(p_ + 1, a, b, p, x, y)
    s_ := REMAINDER(s_, y^2 - x^3 - a·x - b, y)
    Prog
      If POLY_DEGREE(polygcd(s_, r_, p, x), x) < 1 exit
      q_ := SQUARE_ROOT(p_, n)
      If q_ = ?
        RETURN 0
      s_ := (power(x, p, r_, p, x) - x)·ψ(q_, a, b, p, x, y)^2
      s_ :+ ψ(q_ - 1, a, b, p, x, y)·ψ(q_ + 1, a, b, p, x, y)
      s_ := REMAINDER(s_, y^2 - x^3 - a·x - b, y)
      If POLY_DEGREE(polygcd(s_, r_, p, x), x) < 1
        RETURN 0
      s_ := 4·ψ(q_, a, b, p, x, y)^3·power(x^3 + a·x + b, (p + 1)/2, r_, p, x)
      s_ :- ψ(q_ + 2, a, b, p, x, y)·ψ(q_ - 1, a, b, p, x, y)^2
      s_ :+ ψ(q_ - 2, a, b, p, x, y)·ψ(q_ + 1, a, b, p, x, y)^2
      s_ := REMAINDER(s_, y^2 - x^3 - a·x - b, y)
      s_ := SUBST(s_, y, 1)
      If POLY_DEGREE(polygcd(s_, r_, p, x), x) > 0
        RETURN MOD(2·q_, n)
        RETURN MOD(- 2·q_, n)
```

Dieser erste Teil des Algorithmus behandelt den Fall A. Die zweite Hälfte des Algorithmus, die den Fall B behandelt befindet sich auf der nächsten Seite.

```
t_ := polyreduce(nPx(MOD(p, n), a, b, p, x), p)
s_ := polyreduce(nPy(MOD(p, n), a, b, p, x, y), p)
x_ := power(x, p, r_, p, x)
u_ := power(x_, p, r_, p, x)
y_ := power(x^3 + a·x + b, (p - 1)/2, r_, p, x)·y
v_ := power(y_/y, p + 1, r_, p, x)·y
i_ := polymult(NUMERATOR(u_), DENOMINATOR(t_), a, b, p, x, y)
i_ :- polymult(NUMERATOR(t_), DENOMINATOR(u_), a, b, p, x, y)
h_ := polymult(NUMERATOR(v_), DENOMINATOR(s_), a, b, p, x, y)
h_ :- polymult(NUMERATOR(s_), DENOMINATOR(v_), a, b, p, x, y)
i_ := polymult(i_, polymult(DENOMINATOR(v_), DENOMINATOR(s_), a, b, p, x, y), a, b, p, x, y)
h_ := reduce(h_, a, b, p, r_)
h_ := polymult(h_, polymult(DENOMINATOR(u_), DENOMINATOR(t_), a, b, p, x, y), a, b, p, x, y)
s_ := reduce(h_, a, b, p, r_)
w_ := reduce(i_, a, b, p, r_)
i_ := power(s_, 2, r_, p, x, y, a, b)/power(w_, 2, r_, p, x, y, a, b)
w_ := s_/w_
t_ := FACTOR(i_ - u_ - t_, Trivial)
s_ := reduce(NUMERATOR(t_), a, b, p, r_)
t_ := reduce(DENOMINATOR(t_), a, b, p, r_)
s_ :/ t_
t_ := (u_ - s_)
t_ :*w_
t_ := reduce(NUMERATOR(t_), a, b, p, r_)/reduce(DENOMINATOR(t_), a, b, p, r_)
t_ := FACTOR(t_ - v_, Trivial)
g_ := reduce(NUMERATOR(t_), a, b, p, r_)
 i_ := reduce(DENOMINATOR(t_), a, b, p, r_)
Loop
    τ :+ 1
    t_ := polyreduce(nPx(τ, a, b, p, x), p)
    t_ := nPxpower(τ, a, b, p, r_, x, x_)
    t_ := FACTOR(t_, Trivial)
    j_ := polymult(reduce(NUMERATOR(s_), a, b, p, r_), reduce(DENOMINATOR(t_), a, b, p, r_), a, b, p, x, y)
    t_ := polymult(reduce(DENOMINATOR(s_), a, b, p, r_), reduce(NUMERATOR(t_), a, b, p, r_), a, b, p, x, y)
    t_ := j_ - t_
    If POLY_DEGREE(polygcd(t_, r_, p, x), x)_ > 0
        Prog
            h_ := polyreduce(nPy(τ, a, b, p, x, y), p)
            j_ := reduce(power(DENOMINATOR(h_), p, r_, p, x, y, a, b), a, b, p, r_)
            h_ := reduce(power(NUMERATOR(h_), p, r_, p, x, y, a, b), a, b, p, r_)
            t_ := polymult(g_, j_, a, b, p, x, y),
            t_ :- polymult(i_, h_, a, b, p, x, y),
            t_ := POLY_MOD(REMAINDER(t_, y^2 - x^3 - a·x - b, y), p)
            t_ := SUBST(t_, y, 1)
            If POLY_DEGREE(polygcd(t_, r_, p, x), x) > 0
                RETURN τ
```

Wobei die Methode nPxpower() eine kleine Hilfsmethode ist, welche die rechte Seite der Gleichung für die x-Koordinate aus Satz 3.12 berechnet. Ich habe dies als notwendig erachtet, da der Algorithmus sonst noch länger und unhandlicher geworden wäre.

```
nPxpower(n, a, b, p, r, x, xp, s_, t_, w_, y_) :=
  Prog
    s_ := ψ(n - 1, a, b, p, x, y_)·ψ(n + 1, a, b, p, x, y_)
    If ODD?(n)
      s_ :* (x^3 + a·x + b)/y_^2
    t_ := ψ(n, a, b, p, x, y_)^2
    If EVEN?(n)
      t_ :* (x^3 + a·x + b)/y_^2
    t_ := power(t_, p, r, p, x)
    w_ := polymult(xp, t_, a, b, p, x, y)
    POLY_MOD(w_ - power(s_, p, r, p, x), p)/POLY_MOD(t_, p)
```

3.5 Laufzeit und Ergebnisse des Algorithmus

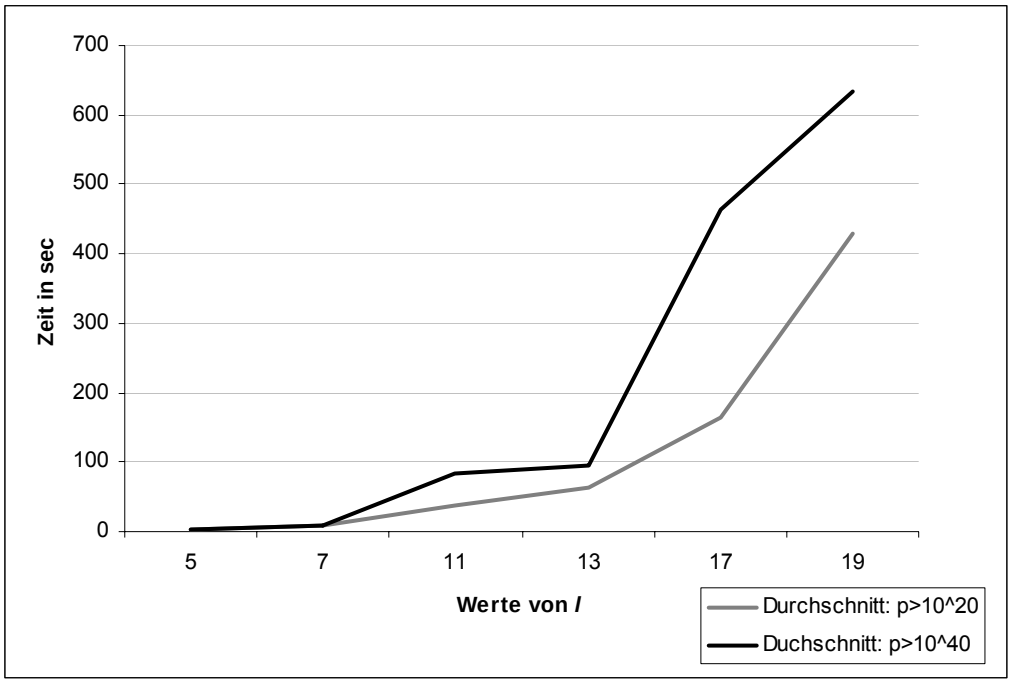
Wie schon zu Anfangs erwähnt hat der Algorithmus eine subexponentielle Laufzeit und ist somit unserer alten order() Methode, die den Babystep-Giantstep Algorithmus verwendet überlegen. Bei kleinen Beispielen mit $p<10^{12}$ war die alte order() Methode schneller, wird jedoch dann von unserer Version des Schoof- Algorithmus überholt.
Doch wie groß sind die Probleme, die wir mit unserer Version des Schoof- Algorithmus noch lösen können?

SOLVE(4·√p = 2·3·5·7·11·13·17·19·23·29·31, p)
p = 2.514031887·10^21

NEXT_PRIME(10^{20}) = 100000000000000000039
schoof(1, 2, 10^{20} + 39, 2) = 0
schoof(1, 2, 10^{20} + 39, 3) = 2
schoof(1, 2, 10^{20} + 39, 5) = 3
schoof(1, 2, 10^{20} + 39, 7) = 2
schoof(1, 2, 10^{20} + 39, 11) = 6
schoof(1, 2, 10^{20} + 39, 13) = 1
schoof(1, 2, 10^{20} + 39, 17) = 1
schoof(1, 2, 10^{20} + 39, 19) = 14
schoof(1, 2, 10^{20} + 39, 23) = 4
schoof(1, 2, 10^{20} + 39, 29) = 17
schoof(1, 2, 10^{20} + 39, 31) = 12

Wir sehen, dass wir problemlos mit elliptischen Kurven bis zu einer Größenordnung von $p=10^{21}$ arbeiten können. Doch die letzten Berechnungen für $l = 29$ und $l = 31$. dauern schon sehr lange. Auf meinem PC dauerten diese Berechnungen bereits eine halbe Stunde. Die Laufzeit des Schoof- Algorithmus in der von uns Implementierten Version hat immer noch eine Laufzeit von $O\left(\log^8 p\right)$. (siehe [Mirbach, S55])
Eine vollständige Berechnung der Anzahl der Punkte einer elliptischen Kurve dauert also für realistisch große Beispiele immer noch zu lange.

Werfen wir einen Blick auf den Anstieg der Laufzeit bei größer werdenden *l* und *p*.



Die oben zu sehende Grafik zeigt die durchschnittlichen Laufzeit des Schoof-Algorithmus für $y^2 \equiv x^3 + x + 2 \bmod p_i$ mit p_i den ersten fünf Primzahlen größer als 10^{20} beziehungsweise 10^{40}.

Man kann jedoch den Algorithmus bei einem *l* einer bestimmten Größe abbrechen, und die berechneten Kongruenzen dazu verwenden, um die Schrittweite des BSGS Algorithmus zu erhöhen. Gelingt es uns zum Beispiel nur den Fall $l = 2$ zu berechnen, so haben wir die Laufzeit von BSGS bereits halbiert. Gelingt uns auch noch die Berechnung für das nächst größere l = 3, so haben wir die Geschwindigkeit von BSGS bereits versechsfacht. Betrachten wir nun ein Beispiel realistischer Größe und brechen wieder bei der ersten Berechnung, die über 30 Minuten benötigt, ab:

NEXT_PRIME(10^{50}) = 100000000000000000000000000000000000000000000000151
schoof(1, 2, 10^{50} +151 , 2) = 0
schoof(1, 2, 10^{50} +151, 3) = 2
schoof(1, 2, 10^{50} +151, 5) = 2
schoof(1, 2, 10^{50} +151, 7) = 1
schoof(1, 2, 10^{50} +151, 11) = 10
schoof(1, 2, 10^{50} +151, 13) = 12
schoof(1, 2, 10^{50} +151, 17) = 13
schoof(1, 2, 10^{50} +151, 19) = 3
schoof(1, 2, 10^{50} +151, 23) = 3
schoof(1, 2, 10^{50} +151, 29) = 25

Mit dem oben angeführten Beispiel haben wir nun gezeigt, dass wir in einer annehmbaren Zeit Lösungen für realistische Probleme, bis zu einem $l = 29$ finden können. Wir könnten somit den BSGS- Algorithmus um einen Faktor von

$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 = 6.469.693.230$

beschleunigen.

In der Praxis sind derartige hybride Verfahren, zur Berechnung der Anzahl der Punkte einer elliptischen Kurve, sehr gebräuchlich und liefern auch sehr gute Ergebnisse.

Die Laufzeit des Schoof- Algorithmus kann jedoch noch extrem verbessert werden. Wir wollen nun kurz einen Blick auf eine Verbesserungsmöglichkeit werfen.

3.6 Der SEA- Algorithmus

In der Praxis ist es so, dass sich Berechnungen für $l > 31$ in keiner Implementierung des Schoof- Algorithmus, vom investierten Zeitaufwand her, rentieren. Die immer noch sehr hohe Laufzeit resultiert aus der Tatsache, dass die für die Berechnungen verwendeten Polynome nur nach dem entsprechenden Divisionspolynom $\psi_l(x)$ reduziert werden, und daher die Polynome einen Grad von $\frac{l^2-1}{2}$ haben.

Die Idee des von N. Elkies und A. O. L. Atkins entwickelten, verbesserten Schoof-Algorithmus, in Folge SEA- Algorithmus genannt, ist es nun, die Ordnung der Polynome zu reduzieren. Man kann dies erreichen, indem man nicht mehr nach $\psi_l(x)$ reduziert, sondern nach einem Teiler f_l von $\psi_l(x)$, der nur mehr die Ordnung $\frac{l-1}{2}$ hat.

Wir erinnern uns an Satz 3.4 und erweitern diesen auf folgende Form:

$$\left(\varphi_l^2 - \tau_l \varphi_l + p_l\right) \cdot (P) = O$$

wobei $\tau_l \equiv t \mod l$ und $p_l \equiv p \mod l$. Wir erinnern uns außerdem, dass wir t als **Spur des Frobenius** auf E bezeichnet haben.
Weiters gilt, dass die Gleichung

$$f(X) = X^2 - \tau_l X + p_l \qquad \in F_l[X]$$

dem charakteristischen Polynom χ_{φ_l} von φ_l entspricht.
(Herleitung und Beweis siehe [Mirbach, S57ff])

Um die Idee des Sea- Algorithmus zu verstehen, müssen wir an dieser Stelle noch kurz den Begriff der Untergruppen einführen

Satz 3.13. Seien l und p teilerfremd. Dann gibt es in $E[l]$ genau $l + 1$ Untergruppen der Ordnung l. Diese bezeichnen wir als $C_1, \ldots, C_{l+1}$. Weiters gilt:

$$\bigcup_{i=1}^{l+1} C_i = E[l]$$

Definition 3.14. Eine endliche Untergruppe *C* der Ordnung *l* von *E[l]* heißt invariant unter einem Endomorphismus φ, falls gilt: $\varphi(C) \subseteq C$. Falls φ nicht der Null-Endomorphismus ist, gilt sogar $\varphi(C) = C$, denn *l* ist prim.
(siehe [Mirbach, S61f])

Betrachten wir nun den eingeschränkten Frobenius und das charakteristische Polynom genauer. Wir müssen nun zwischen zwei speziellen Fällen unterscheiden. Entweder das charakteristische Polynom $f(X)$ hat zwei verschiedene Nullstellen α und β, oder es hat die doppelte Nullstelle α. Weiters müssen wir unterscheiden, ob im Falle von zwei verschiedenen Nullstellen, beide Nullstellen Elemente von F_l oder von $F_{l^2} - F_l$ sind.
Hierzu stellen wir folgenden Satz auf:

Satz 3.15. Sei *E* eine elliptische Kurve über F_p und *l* prim, $l \neq p$. Sei weiters φ_l der auf die *l*-Torsionsgruppe *E[l]* eingeschränkte Frobenius und $f(X) = X^2 - \tau_l X + p_l$ sein charakteristisches Polynom. Dann können folgende drei Fälle unterschieden werden:

1) Zerfällt $f(X)$ in $f(X) = (X - \alpha)^2$ mit $\alpha \in F_l$, so folgt: entweder sind alle Untergruppen der Ordnung *l* von *E[l]* invariant unter φ_l, oder genau eine ist invariant unter φ_l^l; in beiden Fällen gilt für die Spur des Frobenius $\tau_l \equiv 2\alpha \bmod l$, mit $\alpha^2 \equiv p_l \bmod l$, $0 < \alpha < l$.

2) Zerfällt $f(X)$ in $f(X) = (X - \alpha)(X - \beta)$ mit $\alpha, \beta \in F_l$, $\frac{\alpha}{\beta} = \zeta_r$, $ord(\zeta_r) = r > 1$, so folgt: genau zwei Untergruppen der Ordnung *l* von *E[l]* sind invariant unter φ_l und alle anderen sind invariant unter φ_l^l, *r* minimal, *r* teilt (*l* – 1); für die Spur des Frobenius gilt: $\tau_l \equiv \pm(\zeta_r + 1) * \sqrt{\frac{p_l}{\zeta_l}} \bmod l$, $\zeta_r \in F_l$.

3) Zerfällt $f(X)$ in $f(X) = (X - \alpha)(X - \beta)$ mit $\alpha, \beta \in F_{l^2} - F_l$, $\frac{\alpha}{\beta} = \zeta_r$, $ord(\zeta_r) = r > 1$, so folgt: keine Untergruppe der Ordnung *l* von *E[l]* ist invariant unter φ_l und alle sind invariant unter φ_l^l, *r* minimal, *r* teilt (*l* + 1); für die Spur des Frobenius gilt: $\tau_l \equiv \pm(\zeta_r + 1) * \sqrt{\frac{p_l}{\zeta_l}} \bmod l$, $\zeta_r \in F_{l^2}$.

(Herleitung und Beweis siehe [Elliptische Kurven, Mirbach, S63ff])

Befinden wir uns in den Fällen 1) oder 2), so heißt *l* **Elkies- Primzahl**; im Fall 3) handelt es sich bei *l* um eine **Atkins- Primzahl**.
Wir wollen nun einen kurzen Blick auf die Möglichkeiten zur Verbesserung des Schoof-Algorithmus werfen, die sich durch diese Unterscheidung ergeben.

3.6.1 Elkies- Algorithmus

Handelt es sich bei l um eine Elkies- Primzahl, so wissen wir, dass das passende charakteristische Polynom $f(X)$ des Frobenius φ_l zwei verschiedene Nullstellen α und β in F_l besitzt. Somit gilt:

$$f(X) = (X - \alpha)(X - \beta) = X^2 - (\alpha + \beta)X + \alpha \cdot \beta$$

Gelingt es uns einen der beiden Werte der Nullstellen zu bestimmen, so kennen wir auch τ_l. Berechnen wir zum Beispiel α, so gilt:

$$\tau_l \equiv \alpha + \frac{p_l}{\alpha} \bmod l .$$

Um α zu finden, suchen wir uns einen beliebigen Punkt $P = (x, y) \in C$ und setzen diesen in die Gleichung

$$\varphi_l(x, y) = (x^p, y^p) = \alpha \cdot (x, y)$$

ein. Wir testen diese Gleichung für alle α aus (1,…, l-1), bis wir einen passenden Wert gefunden haben. Man kann sehen, dass dieses Verfahren dem originalen Schoof- Algorithmus sehr ähnlich ist. Doch wir verwenden hier nicht das l-te Divisionspolynom, sondern das Polynom $F_l(x)$. Man kann zeigen, dass für dieses Polynom gilt:

$$ord(F_l(x)) = \frac{l-1}{2} .$$

(Beweis vgl.[Elliptische Kurven, Mirbach, S83ff])

3.6.2 Atkins- Algorithmus

Handelt es sich bei der Primzahl l um eine Atkins- Primzahl, so sind die Nullstellen von $f(X)$ die Werte $\alpha, \beta \in F_{l^2} - F_l$. Des weiteren gilt $\alpha^r = \beta^r$. Betrachten wir nun das Element $\frac{\alpha}{\beta} = \zeta_r$ in F_{l^2}. Wegen $\alpha^r = \beta^r$ ist ζ_r ein Element der Ordnung r und stammt aus einer Menge M mit $\varphi(r)$ Elementen. Bestimmen wir nun alle Elemente der Ordnung r in F_{l^2}, so erhalten wir die Menge M. Da τ_l aus einer Menge T_l stammt, die ebenfalls $\varphi(r)$ Elemente haben muss, sind die Mengen M und T_l identisch.

Wir bestimmen also einen Erzeuger g von $F_{l^2}^{x}$ und berechen danach für $t_i = g^{\frac{i}{r}\left(l^2-1\right)}$ für alle i aus (1,…, r-1) und i teilerfremd zu r. Dadurch erhalten wir die Menge T_l in der sich unser gesuchtes τ_l befinden muss.

Je größer der Wert $\varphi(r)$ für eine Atkins- Primzahl ist, je größer ist auch die Menge T_l, und aufwendiger wird die Berechnung von τ_l. Es stellt sich nun die Frage, in wie weit es überhaupt sinnvoll ist Atkins- Primzahlen speziell zu behandeln. Man könnte einen Kompromiss eingehen und bis zu einer gewissen Größe von $\varphi(r)$ Atkins- Primzahlen mit dem Atkins- Algorithmus behandeln, und größere Werte einfach unterschlagen.
(vgl. [Mirbach, S81f])

3.7 Die Laufzeit des SEA- Algorithmus

Die Kombination der berechneten Werte für die Elkies- und Atkins- Primzahlen erfolgt wie beim ursprünglichen Schoof- Algorithmus mit Hilfe des Chinesischen Restsatzes. Wie bereits weiter oben beschrieben, ist es nicht notwendig, für alle l den Wert τ_l zu bestimmen. Man kann einzelne Werte auslassen (da sie unter Umständen zu kompliziert zu berechnen sind) oder auch bei einem Wert einer bestimmten Größe abbrechen. Die gewonnenen Informationen, über die Ordnung der Kurve, lässt man dann einfach in den BSGS Algorithmus einfließen, und erhöht somit dessen Geschwindigkeit. Es ergibt sich somit für den SEA- Algorithmus eine Laufzeit von $O\left(\log^6 p\right)$..
(siehe [Mirbach, S98])

4 Public- Privat- Key Verschlüsselung

In diesem Kapitel wollen wir nun, da wir über die mathematischen Grundlagen verfügen, erklären warum man überhaupt Zahlentheorie in die Verschlüsselungstechnik einbeziehen muss, und wie wir in der Praxis mit elliptischen Kurven verschlüsseln können.

4.1 Das Problem der Schlüsselverteilung

Nach dem zweiten Weltkrieg entwickelten sich verschiedenste Chiffriersysteme, die den neu erfundenen Computer zur Durchführung des Verschlüsselns nutzten. Es entstanden dadurch sehr sichere Verschlüsselungsverfahren, doch lange konnte man sich nicht auf eines von ihnen als Standard einigen. Schließlich wurde 1976 der Data Encryption Standard (DES) als Verschlüsselungsstandard in den USA übernommen. DES basiert auf einem komplizierten Verfahren, bei dem die Nachricht in Blöcke von 64 Zahlen gespaltet wird, welche für sich verschlüsselt werden. Diese Blöcke werden in Halbblöcke zu 32 Zahlen geteilt, um danach die Zahlen einer Seite nach einem bestimmten Verfahren zu substituieren. Danach wird die Seite mit den substituierten Zahlen zu der unveränderten Seite addiert. Die Blöcke werden danach vertauscht und man beginnt von neuem. Insgesamt wird dieser Schritt 16mal wiederholt. Singh vergleicht das DES- Verfahren sehr passend mit einem Teig, der immer wieder durchgeknetet und danach wieder ausgerollt wird. Der Schlüssel von DES beschreibt die genauen Einzelheiten dieses „Durchknetens“ der Nachricht. Ein anderer Schlüssel bedeutet daher eine komplett andere Art der Verschlüsselung. DES ist wie man bereits erahnt ein sehr sicheres Verschlüsselungsverfahren. Doch es hatte ein großes Problem, welches sehr schnell unüberschaubar zu werden drohte.

Man musste dem Empfänger auf irgendeine Art und Weise die Schlüssel zum Entschlüsseln der Nachricht zukommen lassen. So kam es, dass zu Monats oder Wochenbeginn Banken und Unternehmen die Schlüssel für diese Woche oder diesen Monat ihren Kunden schicken mussten. Dieser Verkehr an Schlüsseln nahm binnen kurzer Zeit gewaltige Ausmaße an und verschlang Unmengen an Ressourcen. Während Staat und Militär mit diesem Problem noch einigermaßen fertig wurden, gerieten private Unternehmen immer mehr ins Hintertreffen. Man begann sich beim Schlüsseltransport auf Dritte zu verlassen, was die Sicherheit des Verfahrens natürlich schwächte.

Weniger die Sicherheit der Verfahren an sich, als die Verteilung der Schlüssel, war somit zum Hauptproblem der Chiffrierung geworden. Man musste eine bessere Lösung für dieses Problem finden, oder Unternehmen würden sich angesichts des schnell wachsenden Datenverkehrs schnell inmitten eines logistischen Alptraumes wieder finden.

4.2 Diffie- Hellman Schlüsselaustausch

Für das Problem des Schlüsselaustausches lieferte die Zahlentheorie schließlich Lösungen, welche die Verschlüsselung revolutionierten. Erste Erfolge erreichten die beiden amerikanischen Mathematiker Whitfield Diffie und Martin Hellman im Jahr 1976. Sie entwickelten ein Verfahren zum Schlüsselaustausch, welches sicherstellte, dass eine dritte Person, welche den Nachrichtenverkehr zwischen den Kommunikationspartnern abhörte, den Schlüssel nicht stehlen konnte.

Das Prinzip dieses Schlüsselaustausches war genau so einfach, wie es brillant war. Allgemein bekannt sind:

- die endliche abelsche Gruppe G
- ein Element P von G
- n, die Ordnung von P

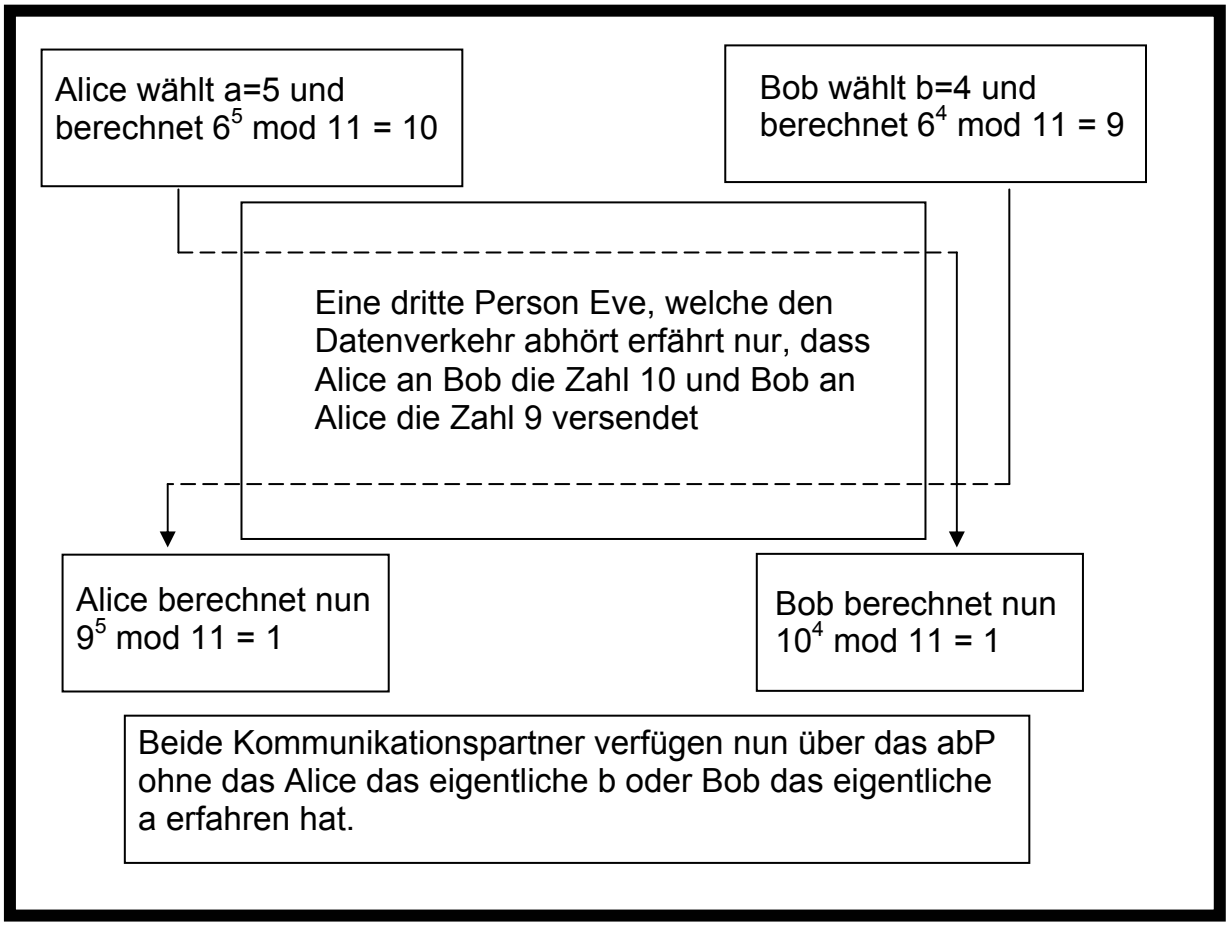
Will nun Alice eine verschlüsselte Nachricht an Bob senden, so muss sie Bob diesen Schlüssel vorher zukommen lassen.

Das Verfahren funktioniert nun folgendermaßen:

1) Alice wählt eine Zahl a aus $\{1,2,\ldots,n\text{-}1\}$ und bildet mit a das Gruppenelement aP, welches Sie an Bob übermittelt.
2) Auch Bob wählt eine Zahl b aus $\{1,2,\ldots,n\text{-}1\}$ und schickt an Alice das Gruppenelement bP
3) Sowohl Alice als auch Bob bilden nun $b(aP) = abP$ bzw. $a(bP) = abP$. Somit verfügen beide über abP, ohne das Bob das a von Alice kennt oder Alice das b von Bob.

Überlegen wir uns dies nun anhand eines einfachen praktischen Beispiels.

Alice und Bob einigen sich auf $P = 6$ und $G = Z_{11}$. Die Funktion mit der Alice und Bob nun arbeiten ist daher die folgende: $6^x \bmod 11$.



Eine dritte Person Eve, welche den Datenaustausch zwischen Alice und Bob abhört, steht nun vor dem Problem, dass sie die Gleichung $6^a \bmod 11 = 10$, bzw. $6^b \bmod 11 = 9$ lösen muss, um

den gemeinsamen Schlüssel zu ermitteln. Dieses Problem nennt man Diffie- Hellman-Problem (DH- Problem).

Diffie- Hellman- Problem: Berechne zu zwei Elementen *kP* und *lP* in $\langle P\rangle = \{kP : k \in Z\}$ das Element *klP* in $\langle P\rangle$.

Das Verfahren beruht also auf dem Prinzip von mathematischen **Einwegfunktionen**; also Funktionen, die sehr schnell und leicht in eine Richtung zu rechnen sind, aber sehr schwer umkehrbar. Das Lösen des Diffie- Hellman- Problems läuft für Eve, wie man sehen kann, auf das Problem des diskreten Logarithmus hinaus. Kann Eve das DL- Problem lösen, so hat sie auch das Diffie- Hellman- Problem gelöst. Es ist jedoch noch nicht bekannt, ob dies auch in die andere Richtung gilt, und somit das zu einem schwer lösbaren DL- Problem passende DH- Problem auch schwer lösbar ist.

Überlegen wir uns nun an einem kleinen Beispiel, wie wir dieses Verfahren auf elliptische Kurven anwenden können.
Als Gruppe *G* wählen wir die Kurve $y^2 = x^3 + x + 1 \bmod 11$. Als Punkt P wählen wir P[1,5].

```
Reg?(1, 1, 11) = true
                    ⎡ ∞ 0  0  1  1 2  3  3  4  4  6  6  8 8 ⎤
points(1, 1, 11)`=  ⎢                                        ⎥
                    ⎣ ∞ 1 10  5  6 0  3  8  5  6  5  6  2 9 ⎦

P := [1, 5]

ordP(P, 1, 11) = 14
```

Alice wählt nun a=5 und Bob wählt b=9

```
mult(P, 5, 1, 11) = [4, 6]
mult(P, 9, 1, 11) = [4, 5]
```

Nun tauschen Alice und Bob ihre Punkte aus und jeder multipliziert den empfangenen Punkt mit seiner geheimen Zahl.

```
mult([4, 5], 5, 1, 11) = [8, 2]
mult([4, 6], 9, 1, 11) = [8, 2]
```

Beide verfügen nun über den selben Punkt [8,2], ohne dass sie diesen an den anderen versendet haben. Wir sehen also, dass der Diffie- Hellman Schlüsselaustausch auch auf elliptischen Kurven problemlos funktioniert.
Werfen wir nun im nächsten Abschnitt einen Blick auf das bekannteste und derzeit auch meist verwendete Verschlüsselungssystem.

4.3 RSA- Verfahren

Parallel zu Diffie und Hellman arbeitete ein zweites Team von Mathematikern ebenfalls an einer Lösung für das Problem der immer größer werdenden Menge an Schlüsseln, welche regelmäßig dem jeweiligen Gesprächspartner geschickt werden mussten. Während Diffie und Hellman eine Lösung für den Austausch der Schlüssel fanden, entwickelte dieses zweite Team ein völlig neues Verschlüsselungsverfahren.
Die drei Mathematiker Ron Rivest, Adi Shamir und Leonard Adleman entwickelten 1977 das erste so genannte Public- Private- Key Verschlüsselungsverfahren. Nach ihnen RSA-Verfahren genannt, beruhte es auf einer genau so einfachen, wie genialen Idee. Alice verfügt nicht mehr über einen einzigen Schlüssel sondern über zwei verschiedene. Den einen Schlüssel, den Public- Key, gibt Alice bekannt, und jeder kann mit diesem Schlüssel Nachrichten verschlüsseln und an Alice schicken. Den anderen Schlüssel behält sich Alice, denn nur mit diesem zweiten Schlüssel, dem Private- Key, kann man die Nachricht wieder entschlüsseln.

Das RSA- Verfahren beruht auf zwei zahlentheoretischen Sätzen:

Satz 4.1. Satz von Fermat: Ist p eine Primzahl und a eine natürliche Zahl, die nicht durch p geteilt werden kann, so gilt:

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

(Beweis siehe [Koblitz,S20])

Die zweite Grundlage des Verfahrens bildet eine Verallgemeinerung des Satzes von Fermat.

Satz 4.2. Satz von Euler: Es sei m eine natürliche Zahl und $\varphi(m)$ die Anzahl der zu m teilerfremden natürlichen Zahlen, die kleiner als m sind. Weiters sei a eine zu m teilerfremde ganze Zahl, dann gilt:

$$a^{\varphi(m)} \equiv 1 \mod m$$

(vgl. [Koblitz, S12])

Die Funktion $\varphi(m)$ heißt eulersche Funktion. Für den Spezialfall das m eine Primzahl ist gilt:

$$\varphi(p) = p - 1 .$$

Wir wollen nun den Fall behandeln, in dem wir ein n in zwei Primzahlen p und q teilen können, sodass gilt $n = pq$. Für die eulersche Funktion gilt dann:

$$\varphi(n) = \varphi(p \cdot q) = \varphi(p) \cdot \varphi(q) = (p-1) \cdot (q-1)$$

Wir verfügen nun über alle mathematischen Grundlagen, und wollen nun das RSA- Verfahren vorstellen:

1.) Alice wählt zwei große Primzahlen p und q und berechnet sich daraus $n = pq$ und $\varphi(n) = (p-1) \cdot (q-1)$
2.) Nun wählt sich Alice eine Zahl e (Encryptor), mit $1 < e < n$ und e teilerfremd zu $\varphi(n) = (p-1) \cdot (q-1)$. Weiters berechnet Alice sich eine Zahl d (Decryptor) sodass gilt: $e \cdot d \equiv 1 \mod (p-1) \cdot (q-1)$ (dies macht Alice mit dem erweiterten euklidischen Algorithmus)
3.) Alice gibt nun das Paar (n,e) als öffentlichen Schlüssel bekannt und behält sich das Paar (n,d) als privaten Schlüssel.

4.) Bob will nun eine Nachricht *m* an Alice schicken und besorgt sich aus dem öffentlichen Verzeichnis den Schlüssel (*n,e*) von Alice. Er berechnet nun $c = m^e \mod n$ und schickt Alice die so verschlüsselte Nachricht *c*.

5.) Alice bildet nun mit ihrem privaten Schlüssel $c^d \mod n = \left(m^e\right)^d \mod n = m^{ed} \mod n$.
Da Alice *e* und *d* so gewählt hat, dass $e \cdot d \equiv 1 \mod (p-1)\cdot(q-1)$ gilt $e \cdot d = 1 + k(p-1)(q-1)$.
Daraus folgt $m^{ed} \mod n = m^{1+(p-1)(q-1)*k} \mod n = m\left(m^{(p-1)(q-1)*k}\right) \mod (p-1)(q-1) = m$.

Wir wollen uns nun wieder an einem praktischen Beispiel ansehen, wie das Verfahren funktioniert:

(p := NEXT_PRIME(RANDOM(2^{64}))) = p := 10825955976732494197

(q := NEXT_PRIME(RANDOM(2^{64}))) = q := 9457521030628019621

(n := p·q) = n := 1023867063266006673219506887339846393337

(e := NEXT_PRIME(RANDOM(2^{64}))) = e := 8175545344570931989

EXTENDED_GCD((p - 1)·(q - 1), e) =
[1, [-1258820410078248696, 157648780861085025344953092411739841 89]]

d := 157648780861085025344953092411739841 89

m := 123456789

(c := MOD(m^e, n)) = c := 990126610759846450840009513051764391 89

MOD(c^d, n) = 123456789

Man kann sehen, wie einfach und problemlos das Generieren von Schlüsseln und das Anwenden des Verfahrens funktioniert.

Eine dritte Partei, welche versucht die Nachricht zu entschlüsseln, muss entweder ein DL-Problem lösen, oder versuchen den privaten Schlüssel zu ermitteln. Dies gelingt jedoch nur, falls man n in seine Faktoren zerlegen kann.

$$\text{FACTORS}(n) = \begin{bmatrix} 9457521030628019621 & 1 \\ 10825955976732494197 & 1 \end{bmatrix}$$

Bereits bei diesem sehr einfachen und kleinen Beispiel dauerte das Faktorisieren von *n* 294sec. Verwendet man für das RSA- Verfahren weitaus größere Primzahlen, als wir in unserem Beispiel, so kann das Faktorisieren von *n* sogar Jahrzehnte dauern.

Wir verfügen also mit dem RSA- Verfahren über ein sehr sicheres Verschlüsselungssystem, welches einen ungemeinen Vorteil gegenüber DES und anderen klassischen Verfahren besitzt: die Schlüssel müssen nicht mehr von auf einem sicheren Weg von Alice an Bob weitergegeben werden. Sie liegen öffentlich auf und dennoch kann eine dritte Partei nur unter enormen Schwierigkeiten unbefugt eine Nachricht entschlüsseln.

4.4 ElGamal Verfahren

Wir haben mit dem RSA Verfahren im letzten Abschnitt das Standardbeispiel für ein Public- Privatekey Verfahren vorgestellt. Nun wollen wir ein neueres Verfahren vorstellen, welches das RSA Verfahren schon in einigen Bereichen abgelöst hat.

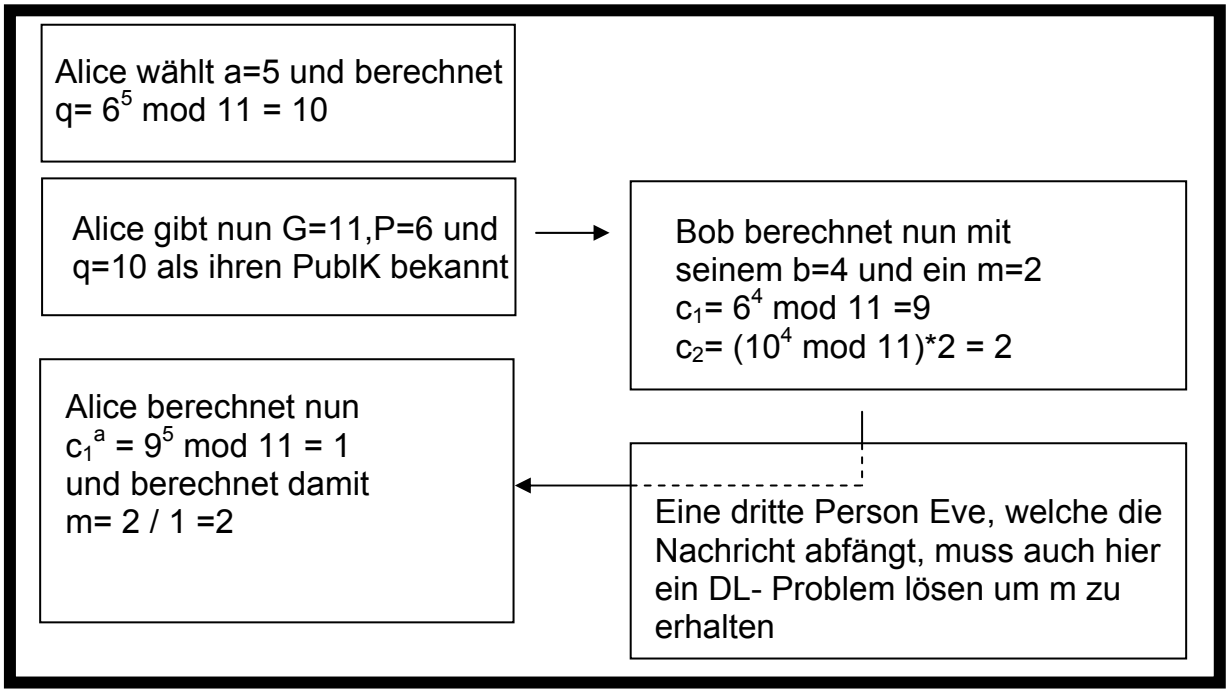
4.4.1 ElGamal Verschlüsselung

Das von Taher ElGamal 1984 entwickelte Verfahren basiert, wie man sehen wird, auf dem Diffie- Hellmann Schlüsselaustausch und hat einige Vorteile gegenüber dem RSA-Verfahren. Allgemein bekannt sind:

- die zyklische Gruppe G
- n, die Ordnung von G
- g, ein Element von G

1.) Alice wählt nun ein zufälliges a aus (1,…,n-1) und bildet damit $q = g^a$. Dieses q gibt Alice nun als ihren Public- Key bekannt, während a als Privat-Key geheim bleibt.
2.) Bob will Alice die Nachricht m zukommen lassen und konvertiert dazu m in ein Element von G.
3.) Bob wählt nun seinerseits ein zufälliges b aus (1,…,n-1) und berechnet daraus ein $c_1 = g^b$ und $c_2 = mq^b$.
4.) Dieses $[c_1, c_2]$ sendet er nun an Alice.
5.) Alice bildet $\frac{c_2}{c_1^a} = \frac{m \cdot q^b}{\left(g^b\right)^a} = \frac{m \cdot g^{ab}}{g^{ab}} = m$ um die ursprüngliche Nachricht zu erhalten.

Wir wollen nun das Verfahren mit einem einfachen und leicht nachzuvollziehenden Beispiel testen:



4.4.2 EC- ElGamal Verschlüsselung

Nun wollen wir auch dieses Verfahren auf elliptische Kurven anwenden. Als Gruppe G wählen wir wieder die Kurve $y^2 = x^3 + x + 1 \bmod 11$. Als Punkt P wählen wir wieder $P[1,5]$.
Wir wissen, dass die Ordnung dieser Kurve $n = 14$ ist, und könne somit sofort mit dem Verfahren beginnen.
Wieder wählt Alice als ihren Privat- Key $a = 5$ und berechnet sich ihren Public- Key.

$Q = \text{mult}(P, 5, 1, 11) = [4, 6]$

Der Public Key besteht nun aus [$y^2 = x^3 + x + 1 \bmod 11$, $P[1,5]$, $Q[4,6]$].
Bob schlägt nun diesen Schlüssel im Verzeichnis nach und will Alice die Nachricht $m = 3$ schicken. Er wählt nun für diese eine Nachricht seinen Schlüssel $b = 9$ und berechnet sich einen Punkt $K = bQ$ und sein $c_1 = bP$.

$K = \text{mult}([4, 6], 9, 1, 11) = [8, 2]$
$c_1 = \text{mult}(P, 9, 1, 11) = [4, 5]$

Nun berechnet sich Bob sein c_2.

$c_2 = \text{MOD}(10 \cdot K_1 , 11) = 3$

Bob hat sich nun sein c_1 und sein c_2 berechnet und kann dieses nun an Alice verschicken. Wenn wir aber c_1 betrachten, so sehen wir, dass es sich bei c_1 nicht um ein Skalar, sondern um einen Punkt handelt. Wir erinnern uns an dieser Stelle an Satz 1.3. Demzufolge war ein Punkt einer elliptischen Kurve eindeutig durch seine x- Koordinate, und ein einzelnes Bit definiert, welches angibt, ob die y-Koordinate gerade oder ungerade ist.
Man kann daher den Punkt c_1 folgendermaßen mit einer einzelnen Zahl darstellen:

$c_1 := 2 \cdot c_1 + \text{MOD}(c_1 , 2) = 9$

Dieses Zusammenfassen eines Punktes nennt man **Point Compression.** Bei dem sehr einfachen Beispiel, welches wir hier berechnen, ist dies nicht unbedingt notwendig, in der Praxis kann man jedoch die Menge an versendeten Daten ungemein reduzieren.
Jetzt kann Bob seine Nachricht $C = [9,3]$ an Alice senden.

Alice muss nun zu aller erst den komprimierten Punkt c_1 wieder in seine ursprüngliche Form bringen.

$c_{11} := \text{FLOOR}(c_1 , 2) = 4$
$c_{12} := \text{ITERATE}\ (\text{IF}(\text{MOD}(y\_, 2) = \text{MOD}(c_1 , 2), y\_, p - y\_), y\_, \text{SQUARE\_ROOT}(c_{11}\text{^}3 + 1 \cdot c_{11} + 1, 11), 1) = 5$

Nun da Alice wieder über den ursprünglichen c_1 verfügt kann sie $K = ac_1$ bilden.

K = mult([4, 5], 5, 1, 11) = [8, 2]
(M := MOD(INVERSE_MOD(K_1 , 11)·c_2 , 11)) = M := 10

Somit hat Alice aus der verschlüsselten Nachricht *c* wieder den Klartext m gewonnen.

Nun ist es an der Zeit, dass wir ein praktisches Beispiel mit realistischen Werten durchrechnen. Dazu verwenden wir die folgenden Methoden:

```
ElGamalencrypt(m, k_, K_, c_) :=
  Prog
    k_ := RANDOM(r - 2) + 2
    K_ := mult(Q, k_, a, p)
    c_ := mult(P, k_, a, p)
    [2·c_↓1 + MOD(c_↓2, 2), MOD(m·K_↓1, p)]
```

m…….	Nachricht
a,b,p……	Parameter der Kurve
n……..	Ordnung der Kurve
P……..	Allgemein bekannter Punkt
r……...	Ordnung von P
Q…….	Public Key von Alice

Und zum Entschlüsseln verwenden wir:

```
ElGamaldecrypt(c, d, c_, K_) :=
  Prog
    c_ := FLOOR(c↓1, 2)
    c_ := [c_, ITERATE(IF(MOD(y_, 2) = MOD(c↓1, 2), y_, p - y_), y_,
        SQUARE_ROOT(c_^3 + a·c_ + b, p), 1)]
    K_ := mult(c_, d, a, p)
    MOD(INVERSE_MOD(K_↓1, p)·c↓2, p)
```

c…….	verschlüsselte Nachricht	a,b,p……	Parameter der Kurve
n……..	Ordnung der Kurve	P……..	Allgemein bekannter Punkt
Q…….	Public Key von Alice	d……..	Private Key von Alice

Die zu übertragende Nachricht lautet:

M	i	s	t	e	r	B	o	n	d	b	i	t	t	e	m	e	l	d	e	n
13	09	19	20	05	18	02	15	14	04	02	09	20	20	05	13	05	12	04	05	14

Ver- und Entschlüsseln wir nun diese Nachricht auf einer elliptischen Kurve realistische Größe mit den beiden oben vorgestellten Methoden.

a := -258743725126721201828274893516819894046818200963

b := 61614323777191969934469432463182689883512771206 7

p := 73373680476700040691395964317326510008842069882 3

n := 7337368047670004069139607184206408666898755132 04

P := [2, 125267436092609969799990276946351557181845598 94]

r := n/14

d := 50798681879704647710735552457342609482718448112

Q := [23773138667140638474754984621033263221301785827 3,
399553995101007850200146788626032152928847288685]

m := 13091920051802151404020920200513051204051 4

(c := ElGamalencrypt(m)) =

c := [16289030125851094410536174444102057299250816325 9,
49072216435232792055960913813112856182535864651 6]

ElGamaldecrypt(c, d) = 13091920051802151404020920200513051204051 4

Wir sehen, dass unsere Mühen sich gelohnt haben, und wir endlich über ein funktionierendes Verschlüsselungssystem auf der Basis von Elliptischen Kurven verfügen. Am Ende dieser Arbeit werden wir zeigen, dass dieses Verfahren, klassischen Verfahren, wie RSA deutlich überlegen ist.

Im nächsten Kapitel wollen wir nun einen etwas genaueren Blick auf die Sicherheit dieses Verfahrens werfen, und uns überlegen, wie man diese Verschlüsselung knacken könnte.

5 Das DL- Problem auf elliptischen Kurven

Wir wissen aus dem letzten Kapitel, dass eine dritte Partei, welche das ElGamal-Verfahren brechen will, ein diskretes Logarithmus Problem lösen muss. Die Sicherheit des Verfahrens hängt also davon ab, wie schwer dieses spezielle DL- Problem zu lösen ist. Wir wollen nun die Rolle der dritten Partei übernehmen, und versuchen Methoden zu finden um ElGamal zu entschlüsseln.

5.1 Trivialer Ansatz

Ein Angriff von außen auf das Verfahren reduziert sich auf das Problem für einen gegebenen Punkt *P*, und einen Punkt *Q* ein *k* zu finden, für das gilt $Q = k \cdot P$. Gelingt uns dies, so können wir den Private- Key von Alice bestimmen und alle Nachrichten, die für sie bestimmt sind, entschlüsseln.

Wie schon bei der Suche nach der Anzahl der Punkte einer elliptischen Kurve, beginnen wir mit der einfachsten denkbaren Methode das Problem zu lösen. Wir addieren *P* einfach so lange zu sich selbst, bis wir unser *Q* gefunden haben.

```
trivialsearch(q, u, a, p, k_ := 1, u_) :=
  Prog
    u_ := u
    Loop
      If u_ = q
        RETURN k_
      u_ := add(u_, u, a, p)
      k_ :+ 1
```

```
Reg?(1, 2, 13) = true

                    ⎡ ∞  1   1  2  2  6  6  7   7   9  9  12 ⎤
points(1, 2, 13)` = ⎢                                        ⎥
                    ⎣ ∞  2  11  5  8  4  9  1  12   5  8   0 ⎦

ordP([7, 1], 1, 13) = 12
mult([7, 1], 9, 1, 13) = [1, 11]
trivialsearch([1, 11], [7, 1], 1, 13) = 9
```

Ich denke wir brauchen an dieser Stelle nicht erwähnen, dass dieser Ansatz viel zu langsam ist, um praktisch Verwendung zu finden. Wir wollen daher gleich zur nächsten Methode weitergehen.

5.2 Babystep- Giantstep- Methode

Wir haben im Kapitel zur Bestimmung der Ordnung einer elliptischen Kurve bereits das BSGS- Verfahren vorgestellt. Wir wollen dieses nun auch auf das Problem des diskreten Logarithmus auf elliptischen Kurven anwenden. Hierzu müssen wir die bereits bekannte Methode BSGS1() ein wenig umbauen. Wir suchen nun nicht mehr im Intervall nach einem Nullelement, sondern nach unserem Punkt $Q = k \cdot P$.

Implementiert in DERIVE sieht die Methode folgendermaßen aus:

```
BSGS2(q, u, a, p, k_ := 0, r_, m_, t_) :=
  Prog
    m_ := CEILING(√(p - 1))
    r_ := ITERATES(add(u, w_, a, p), w_, u, m_ - 1)
    t_ := mult(u, -m_, a, p)
    Loop
      If MEMBER?(q, r_)
        RETURN POSITION(q, r_) + k_·m_
        q := add(q, t_, a, p)
      k_ := k_ + 1
```

Überprüfen wir nun, ob die Methode richtig funktioniert.

mult([236, 22386], 45632, 1, 65537) = [43777, 30356]
BSGS2([43777, 30356], [236, 22386], 1, 65537) = 45632 0.078sec

Zum Lösen von kleineren Problemen ist der BSGS- Algorithmus also schon ein wirklich guter Ansatz. Für große Probleme in der Größenordnung von $n>2^{64}$ ist der Algorithmus jedoch immer noch zu langsam. Wie bereits beim Anwenden der BSGS-Methode beim Finden der Ordnung der gesamten Kurve erwähnt, hat die Methode eine Laufzeit von $O\left(\sqrt{n}\right)$. Wir müssen daher eine bessere Methode finden das Q = k*P zu bestimmen.

5.3 Pollard-ρ-Algorithmus

Genau so wie die BSGS- Methode hat die Pollard-ρ-Methode eine Laufzeit von $O\left(\sqrt{n}\right)$. Sie benötigt jedoch weniger Speicherplatz als BSGS und verdient daher besondere Aufmerksamkeit.

Die Idee des Algorithmus ist recht einfach: wir bilden mit einer Funktion $x_i = f\left(x_{i-1}\right)$ so lange Pseudozufallswerte, bis wir in einen zyklischen Kreislauf kommen. Da unser zu Grunde liegende Gruppe eine Ordnung von *n* hat, geschieht dies nach maximal *n* Iterationen. Im

Schnitt geschieht dies nach $\sqrt{n}$ Operationen. Daher die Laufzeit von $O\left(\sqrt{n}\right)$ für diese Methode.

Wollen wir nun für das Problem $Q = k \cdot P$ in der Gruppe *G* der Ordnung *n* eine Lösung finden, so benötigen wir zuerst eine Funktion, welche unsere Pseudozufallswerte generiert. Wir teilen dazu die *G* in drei ungefähr gleich große Gruppen S_1, S_2, S_3. Für die Funktion *f(x)* gilt nun:

$$X_{i+1} = f\left(X_i\right) = \begin{cases} Q + X_i & falls\ X_i \in S_1 \\ 2X_i & falls\ X_i \in S_2 \\ P + X_i & falls\ X_i \in S_3 \end{cases}$$

und

$$O \notin S_2$$

Für ein beliebiges X_i gilt nun: $X_i = a_i P + b_i Q$ mit

$$a_{i+1} = \begin{cases} a_i & falls\ X_i \in S_1 \\ 2a_i \pmod{n} & falls\ X_i \in S_2 \\ a_i + 1 \pmod{n} & falls\ X_i \in S_3 \end{cases}$$

und

$$b_{i+1} = \begin{cases} b_i + 1 \pmod{n} & falls\ X_i \in S_1 \\ 2b_i \pmod{n} & falls\ X_i \in S_2 \\ b_i & falls\ X_i \in S_3 \end{cases}$$

Wir bilden nun für einen Punkt *X* so lange schrittweise $f(X)$ und $f\left(f\left(X\right)\right)$ bis wir auf ein $X_m = X_{2m}$ stoßen. Wie bereits erwähnt treten wir in einen derartigen Zyklus nach durchschnittlich $\sqrt{n}$ Iterationen ein. Sobald wir diesen Zyklus gefunden haben, können wir das DL Problem in der Gruppe *G* leicht lösen.

Auf der nächsten Seite findet der Leser eine Implementierung der Pollard-ρ-Methode in DERIVE.

```
pollard_ρ(q, u, a, p, n, k_ := 1, i_, j_, a_ := 0, b_ := 0, c_ := 0, d_ := 0, x_, y_) :=
  Prog
    x_ := [∞, ∞]
    y_ := [∞, ∞]
    Loop
      If INTEGER?(FIRST(x_))
        i_ := FIRST(x_)
        i_ := 2
      If INTEGER?(FIRST(y_))
        j_ := FIRST(y_)
        j_ := 2
      a_ := MOD(a_ + [a_, 0, 1]↓(MOD(i_, 3) + 1), n)
      b_ := MOD(b_ + [b_, 1, 0]↓(MOD(i_, 3) + 1), n)
      x_ := [add(x_, x_, a, p), add(q, x_, a, p), add(u, x_, a, p)]↓(MOD(i_, 3) + 1)
      c_ := MOD(c_ + [c_, 0, 1]↓(MOD(j_, 3) + 1), n)
      d_ := MOD(d_ + [d_, 1, 0]↓(MOD(j_, 3) + 1), n)
      y_ := [add(y_, y_, a, p), add(q, y_, a, p), add(u, y_, a, p)]↓(MOD(j_, 3) + 1)
      If INTEGER?(FIRST(y_))
        j_ := FIRST(y_)
        j_ := 2
      c_ := MOD(c_ + [c_, 0, 1]↓(MOD(j_, 3) + 1), n)
      d_ := MOD(d_ + [d_, 1, 0]↓(MOD(j_, 3) + 1), n)
      y_ := [add(y_, y_, a, p), add(q, y_, a, p), add(u, y_, a, p)]↓(MOD(j_, 3) + 1)
      If x_ = y_
        Prog
          x_ := SOLVE_MOD((b_ - d_)·x - c_ + a_, x, n)
          Loop
            If mult(u, FIRST(x_), a, p) = q
              RETURN FIRST(x_)
            x_ := REST(x_)
```

Berechnen wir nun mit unserer Methode ein kleines Beispiel, um die Funktionalität zu testen.

```
mult([20, 27501], 2474, 1, 65537) = [46236, 59317]
ordP([20, 27501], 1, 65537) = 5962
pollard_ρ([46236, 59317], [20, 27501], 1, 65537, 5962) = 2474
```

Die Pollard-ρ- Methode ist somit eine wichtige Alternative zum BSGS- Algorithmus. Die Methode benötigt weniger Arbeitsspeicher, und man kann die Anzahl der notwendigen Iterationen, bis zum Finden der Schleife, klein halten, indem man die Funktion *f(x)* weiter spezialisiert. Daher gilt die Pollard-ρ- Methode in der Regel als die beste, derzeit zur Verfügung stehende, Methode, das DL- Problem auf elliptischen Kurven zu lösen.

5.4 Pohlig- Hellman Algorithmus

Die beiden in den letzen Kapiteln vorgestellten Verfahren waren bereits enorme Verbesserungen in Bezug auf die Laufzeit, verglichen mit dem triviale Iterations- Verfahren. Doch die O-Notation der Laufzeit war immer noch eine $O(\sqrt{n})$. Bei Problemen in eine realistischen Größenordnung, mit einem n > 10^{50} wären diese Algorithmuse daher immer noch zu langsam.
Wir wollen daher ein weiteres Verfahren betrachten.

Die Idee des Pohlig- Hellman Verfahrens ist relativ einfach. Man zerlegt *n* in das Produkt seiner Primfaktoren, und zerlegt somit das Problem des DL in der Gruppe $\langle P \rangle$ in viele kleine, leichter zu lösende, Probleme in Untergruppen von $\langle P \rangle$, deren Ordnung Primteiler von *n* sind.

Es sei

$$n = \prod_{i=1}^{t} p_i^{\lambda_i}$$

die Primfaktorzerlegung von n mit Primzahlen p_i und natürlichen Exponenten $\lambda_i \geq 1$. Wollen wir nun wieder zu einem Element Q aus $\langle P \rangle$ den Wert *k*, mit $Q = k \cdot P$, berechnen, so ermitteln wir ein System von Restklassen der Form

$$\begin{aligned} k &\bmod p_1^{\lambda_1} \\ k &\bmod p_2^{\lambda_2} \\ &\ldots\ldots\ldots \\ k &\bmod p_i^{\lambda_i} \end{aligned}$$

aus welchem wir mit Hilfe des Chinesischen Restsatzes *k* berechnen können.

Betrachten wir nun ein beliebiges $p_i^{\lambda_i}$ der Primfaktorzerlegung von *n*. Um die folgenden Überlegungen einfacher zu gestalten schreiben wir einfach $p_i = p$ und $\lambda_i = \lambda$. Wie kann man nun die Zahl z, mit $z \equiv k \bmod p^{\lambda}$ und $z \in \{0,..., p^{\lambda} - 1\}$, berechnen? Werfen wir einen Blick auf die p- adische Zerlegung

$$z = z_0 + z_1 p + z_2 p^2 + ... + z_{\lambda-1} p^{\lambda-1}$$

von z. Es sei nun $R = \frac{n}{p} P$. Somit gilt

$$\frac{n}{p} Q = \frac{n}{p} kP = kR$$

Der Punkt *R* hat jedoch nur mehr eine Ordnung von *p*, sodass gilt $p \cdot R = O$. In der p-adischen Zerlegung von *z* fallen nun alle Werte, welche mit *p* multipliziert werden weg, und es gilt nur mehr

$$\frac{n}{p} Q = kR = zR = z_0 R .$$

Lösen wir also das DL- Problem in der Gruppe $\langle R \rangle$, die eine Untergruppe von $\langle P \rangle$ mit der Ordnung *p* ist, so erhalten wir z_0. (vgl. [Werner, S77ff])

Wir zerlegen somit das große Problem, für einen Punkt *P* der Ordnung *n,* ein $Q = k \cdot P$ zu finden, in viele kleine Unterprobleme, deren Ordnung nur mehr p_i^i ist. Diese viel kleineren Probleme lassen sich zum Beispiel mit der BSGS- Methode lösen.

Implementiert in DERIVE sieht der Algorithmus folgendermaßen aus:

```
PohlHell(q, u, a, p, n, f_, bg_, c_) :=
  Prog
    If PRIME?(n)
      RETURN BSGS2(q, u, a, p)
    If PRIME_POWER?(n)
      Prog
        f_ := FIRST(FIRST(FACTORS(n)))
        bg_ := MOD(BSGS2(mult(q, n/f_, a, p), mult(u, n/f_, a, p), a, p), f_)
        q := add(q, mult(u, -bg_, a, p), a, p)
        RETURN bg_ + f_·PohlHell(q, mult(u, f_, a, p), a, p, n/f_)
    c_ := VECTOR(u_↓1^u_↓2, u_, FACTORS(n))
    CRT(VECTOR(PohlHell(mult(q, n/r_, a, p), mult(u, n/r_, a, p), a, p, r_), r_, c_), c_)
```

FACTORS(65440) = $2^5 * 5^1 * 409^1$
PohlHell([43777, 30356], [236, 22386], 1, 65537, 65440) = 45632

Die Verbesserung der Laufzeit ist mit dieser neuen Idee enorm. . Das Pohlig- Hellman Verfahren hat in diesem Fall nur mehr eine Laufzeit mit einer $O\left(\sum_{i=1}^{t}\left(\lambda_i\left(\log n + \sqrt{p_i}\right)\right)\right)$.

Die Laufzeit hängt somit praktisch nur mehr von dem größten Faktor p_i^i der Primfaktorzerlegung von n ab.
(vgl. [Werner, S79])

5.5 Sicherer Kurven

Wir haben die gebräuchlichsten und besten Verfahren vorgestellt, die zu einem Angriff auf elliptische Kurven verwendet werde können. Nun stellen wir uns die Frage, wie wir eine elliptische Kurve wählen müssen, um sie möglichst sicher zu machen.
Wir stellen zu diesem Zweck folgende Bedingungen für eine sicher elliptische Kurve auf:

1) Wir haben gesehen, dass man das DL- Problem auf einer elliptischen Kurve sehr schnell lösen kann, falls die Ordnung der Kurve in viele kleine Primteiler zerfällt. Wir benötigen somit eine Kurve, deren Ordnung einen **möglichst großen Primteiler** *q* besitzt. In der Praxis reicht ein Teiler $q \geq 2^{160}$
2) Weiters darf die Kurve **weder supersingulär, noch anormal**[3] sein, da für beide Fälle Methoden existieren, das DL- Problem in subexponentieller Zeit zu lösen.

[3] Eine Kurve *E* über F_p heißt anormal, falls Ord(*E*) = *p*

(SSSA- Algorithmus für anormale und MOV- Algorithmus für supersinguläre Kurven)

3) Für die kleinste Zahl l mit $p^l \equiv 1 \bmod n$ sollte $l \geq 20$ gelten. Ist ***l* größer als 20** ist es nur mehr sehr schwer das DL- Problem mit dem MOV- Algorithmus lösen.

Versuchen wir nun eine passende Kurve zu finden, wählen wir zu erst ein $p \approx 2^{163}$. Damit stellen wir laut dem Satz von Hasse sicher, dass die Ordnung der elliptischen Kurve groß genug ist. Wir wählen nun zufällig ein a und ein b und bestimmen die Ordnung der somit entstehenden elliptischen Kurve $y^2 = x^3 + ax + b \bmod p$. Erfüllt die entstandene Kurve alle oben angeführten Bedingungen, so haben wir eine elliptische Kurve von passender Sicherheit gefunden. Wird eine der Bedingungen nicht erfüllt, müssen neue Koeffizienten a und b finden, und wieder mit unserem Test beginnen.

Da diese Methode, wie man erahnen kann sehr zeitaufwendig ist, verwendet man auch einen anderen Weg. Man kann elliptische Kurven mit komplexen Multiplikatoren erzeugen. Bei dieser Methode sucht man sich zu erst eine passende Ordnung, und erzeugt dann zu dieser Ordnung eine zugehörige Kurve. Erst wenn man eine Ordnung gefunden hat, die kryptographisch geeignet ist, erzeugt man die dazu passende Kurve. Man kann mit dieser Methode zwar nicht jede theoretisch mögliche elliptische Kurve erzeugen, die Menge an so generierbaren Kurven ist jedoch für derzeitige Belange ausreichend.

6 Digitale Signatur

Wir haben nun verschiedene Methoden der Kryptographie vorgestellt und wollen uns nun einem weiteren wichtigen Kapitel der modernen Verschlüsselungstechnik widmen. Die digitale Signatur ist ein weiterer wichtiger Aspekt, bei dem elliptische Kurven Verwendung finden.

6.1 Warum eine digitale Signatur?

In all unseren bis jetzt angeführten Beispielen hatte Eve, unsere dritte Partei, immer nur die Möglichkeit Nachrichten, die Alice und Bob an einander versenden mitzuhören. Eve konnte nicht in den Datenfluss eingreifen und Nachrichten abfangen, ändern oder gar fälschen.

In der Praxis ist dies jedoch leider nur all zu leicht möglich. Die Palette an Möglichkeiten, welche eine dritte Partei hat, um den Datenaustausch zwischen zwei Personen zu korrumpieren ist sehr groß. Hier nur einige wenige Beispiele:

- **Gefälschte Nachrichten**: Es gibt keinerlei Garantie für Alice, dass eine Nachricht, die sie über die Datenleitung zu Bob empfängt auch wirklich von Bob kommt.

- **Man-in-The-Middle Attacke**: Besonders Verfahren wie der Diffie- Hellman Schlüsselaustausch sind sehr anfällig für folgenden Angriff. Eve zapft die Leitung zwischen Alice und Bob an und sendet sowohl an Alice als auch an Bob einen geheimen Schlüssel. Alice glaubt, die Nachricht kommt von Bob und Bob glaubt die Nachricht kommt von Alice. Beide senden nun an den vermeintlichen Partner ihren Teil des Schlüssels zurück. Eve fängt nun alle Nachrichten ab, die zwischen den beiden hin und her gesendet werden, entschlüsselt sie mit dem eigene Schlüssel, und sendet sie dann, mit dem passenden richtigen Schlüssel wieder verschlüsselt, an Alice oder Bob weiter. So könnte Eve unbemerkt den kompletten Datentransfer abhören oder Nachrichten sogar nach Belieben ändern.

- **Chosen-Ciphertext Attacke**: Eve sendet an Alice eine verschlüsselte Nachricht, die Eve nach ganz speziellen Kriterien zusammengestellt hat. Nachdem Alice diese Nachricht durch die Entschlüsselungs- Routine geschickt hat, bemächtigt sich Eve der bearbeiteten Nachricht. Mit dieser speziell zusammengestellten Nachricht könnte es Eve gelingen den privaten Schlüssel von Alice zu ermitteln

Wir sehen also, dass es unbedingt notwendig ist, dass Alice und Bob ihre Nachrichten, auf irgendeine Art und Weise, unfälschbar signieren. Hierzu gibt es verschiedene Möglichkeiten. Die wichtigsten wollen wir nun hier vorstellen.

6.2 ElGamal- Signatur

Beginnen wir gleich mit dem von Taher ElGamal im Jahr 1984 vorgestellten, nach ihm benannten Verfahren. Die Voraussetzungen sind genau die Selben wie bei der ElGamal-Verschlüsselung. Allgemein bekannt sind wieder:

- die Gruppe *G*
- die Ordnung *n* von *G*
- ein bestimmtes Element *g* von *G*

Des weiteren müssen sich Alice und Bob auf eine kryptographische Hashfunktion h(x) einigen.

1.) Alice wählt ein zufälliges *a* aus (1,…, *n*-1) und bildet damit $q = g^a \bmod p$. Wieder gibt Alice *q* als ihren öffentlichen Schlüssel bekannt.
2.) Will nun Alice eine Nachricht *m* an Bob signieren, wählt sie zu erst ein *k* aus (1,…..,*p*-1), das zu (*p*-1) teilerfremd ist. Nun bildet Alice $r = g^k \bmod p$, und danach $s = k^{-1}\left(h(m) - ar\right) \bmod (p-1)$. Tritt der Fall *s* = 0 ein, so muss Alice ein neues *k* aussuchen und neu beginnen.
3.) Alice sendet mit der verschlüsselten Nachricht ihre Signatur (*r*,*s*) mit.
4.) Bob will nun überprüfen, ob die Nachricht wirklich von Alice kommt. Hierzu entschlüsselt er die Nachricht und überprüft nun, ob r im Intervall [1, p-1] und s im Intervall [1, *p*-2] liegt. Dann überprüft er, ob die Kongruenz $q^r r^s \equiv g^{h(m)} \bmod p$ erfüllt ist.

Kommt die Nachricht wirklich von Alice so muss gelten:

$$q^r r^s \equiv g^{ar} g^{kk^{-1}(h(m)-ar)} \equiv g^{h(m)} \bmod p$$

Wir sehen, dass diese Kongruenz nur erfüllt sein kann, wenn wirklich Alice mit Hilfe ihres privaten Schlüssels die Signatur (*r*,*s*) erstellt hat.

6.3 Digital Signature Algorithm

Im Jahr 1991 wurde vom US- amerikanischen National Institute of Standards and Technology der Digital Signature Algorithm (DSA) vorgestellt und 1993 als allgemeiner Standard angenommen. Das Verfahren beruht auf der eben vorgestellten ElGamal- Signatur und führt nun einige Verbesserungen und Richtlinien zur Schlüsselgenerierung ein.

1.) Alice wählt eine Primzahl *q* mit $2^{159} < q < 2^{160}$. Danach wählt Alice eine zweite, größere Primzahl *p* mit $2^{511+64t} < p < 2^{512+64t}$ mit *t* aus (0,1,…, 8). Die zu erst gewählte Primzahl *q* muss ein Teiler von *p* sein.
2.) Nun wählt Alice eine Primitivwurzel *x* mod *p* und berechnet daraus $g = x^{\frac{p-1}{q}} \bmod p$. Mit diesem *g* (welches die Ordnung *q* hat) und einem zufällig gewählten *a* aus (1,…, *q*-1) berechnet Alice nun $A = g^a \bmod p$. Der öffentliche Schlüssel von Alice ist somit *(p,q,g,A)*; der private ist *a*.

3.) Alice will nun einen Text *m*, den sie verschlüsselt an Bob senden will, signieren. Dazu wählt sie ein zufälliges *k* aus (1,…., *q*-1) und berechnet damit

$$r = \left(g^k \bmod p\right) \bmod q \qquad \text{und}$$

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

4.) Alice sendet mit der verschlüsselten Nachricht *c* auch ihre Signatur (*r*,*s*).

5.) Bob will nun überprüfen, ob die Nachricht wirklich von Alice ist. Er überprüft daher wieder, ob *r* im Intervall [1, *q*-1] und *s* im Intervall [1, *q*-1] liegen. Danach überprüft er, ob

$$r = \left(\left(g^{s^{-1}h(m) \bmod q} A^{rs^{-1} \bmod q}\right) \bmod p\right) \bmod q$$

gilt.

Nur wenn die Nachricht wirklich von Alice kommt und mit ihrem privaten Schlüssel generiert worden ist gilt:

$$\begin{aligned}
&\left(\left(g^{s^{-1}h(m) \bmod q} A^{rs^{-1} \bmod q}\right) \bmod p\right) \bmod q = \\
&= \left(\left(g^{s^{-1}h(m) \bmod q} g^{ars^{-1} \bmod q}\right) \bmod p\right) \bmod q = \\
&= \left(\left(g^{s^{-1}(h(m)+ar) \bmod q}\right) \bmod p\right) \bmod q = \\
&= \left(\left(g^{k*(h(m)+ar)^{-1}(h(m)+ar) \bmod q}\right) \bmod p\right) \bmod q = \\
&= \left(\left(g^k\right) \bmod p\right) \bmod q = r
\end{aligned}$$

Wir sehen also, dass das DSA Verfahren sehr eng an das ElGamal- Verfahren angelegt ist. In der Praxis benötigt es jedoch weniger Rechenaufwand und die Größe der Exponenten reduziert sich im Vergleich zu ElGamal enorm. Doch genug von DSA; wir wollen nun genau betrachten, wie man eine Nachricht mit elliptischen Kurven signieren kann.

6.4 Elliptic Curve Digital Signature Algorithm

Der Elliptic Curve Digital Signature Algorithm (ECDSA) ist die direkte Anwendung von DSA auf elliptische Kurven. Wie wir sehen werden handelt es sich dabei um ein überaus sicheres und effizientes Verfahren. Allgemein bekannt sind wieder:

- die elliptische Kurve *E* mit ihren Parametern (*a*,*b*,*p*)
- die Ordnung *n* von *E*
- ein Punkt *P* dessen Ordnung *q* ein großer Primteiler von *n* ist.

Weiters haben sich Alice und Bob auf eine kryptographische Hashfunktion h(x) geeinigt. Will Alice nun eine an Bob zu sendende Nachricht *m* signieren, geschieht dies folgendermaßen:

1.) Alice wählt ein *d* aus (2,…, *q*-2) und bildet damit $Q = dP$. Sie gibt (*E*,*P*,*q*,*Q*) als ihren öffentlichen Schlüssel bekannt und behält sich *d* als privaten Schlüssel.

2.) Alice wählt nun eine weitere zufällige Zahl *k* aus (2,…, *q*-2) und berechnet sich daraus $kP = (x_1,y_1)$ und $r = x_1 \bmod q$. Falls $r = 0$ muss sich Alice ein neues *k* suchen und neu beginnen.

3.) Alice berechnet nun $s = k^{-1}\left(h(m) + dr\right) \bmod q$. Ist $s = 0$ muss Alice wieder mit Schritt 2 beginnen.

4.) Alice sendet mit der verschlüsselten Nachricht *c* die Unterschrift (*r*,*s*).

5.) Bob überprüft nun ob r und s ganze Zahlen aus (1,…, q-1) sind und berechnet sich $w = s^{-1} \bmod q$.

6.) Nun berechnet Bob $u_1 = h(m)w \bmod q$ und $u_2 = rw \bmod q$. Dann überprüft er ob $r = x_0 \bmod q$ mit $(x_0, y_0) = u_1 P + u_2 Q$.

Wieder kann diese Bedingung nur erfüllt sein, falls die Nachricht wirklich von Alice mit Hilfe ihres geheimen Schlüssels signiert worden ist.

Wir wollen nun das Verfahren an einem Beispiel mit realistisch großen Zahlen praktisch anwenden und damit zeigen, wie das Signieren in der Praxis funktioniert.

Die Parameter der Kurve Lauten wie folgt:

a := -2587437251267212018282748935168198940468182009 63

b := 6161432377719196993446943246318268988351277120 67

p := 7337368047670004069139596431732651000884206988 23

n := 7337368047670004069139607184206408666898755132 04

P := [4241040376872502021312877283505463028166095736 10,
213293699174308002865557695857800677535626027401]

q := 4143261480313015450275228855691861537700 69

mult(P, q, a, p) = [∞, ∞]

Nun berechnet Alice Ihren Public- und ihren Private-Key:

d := 4015896554753952575674383959503423836918 60

Q := [4698643558583863350058448723186643018856455232 60,
5218688890049689680076332436240037781614592256 57]

Alice kann jetzt (E,P,q,Q), als ihren Public- Key bekannt geben, und behält sich d als ihren Private- Key. Jetzt kann Alice ihre Signatur (r,s) für m erstellen.

m := „abc“

h=todec(SHA_1(source(m)), 16) =
9682368737159886141705690735153157075667664795 17

k := 1407774941337643836317392664381746331369 75

mult(P, k, a, p) = [4865893766293716085043859987182536006993240700 72,
2118841104820027288371014050613893363688826446 12]

r := MOD(4865893766293716085043859987182536006993240700 72, q)

s := MOD(INVERSE_MOD(k, q)·(h + d·r), q)

[r, s]= [**1907937827297070678191540601206640635657 13**,
2593238358362206598937985234891747589513 17]

Bob muss nun Überprüfen, ob die Signatur (r,s) wirklich von Alice kommt, oder eine Fälschung ist:

w := INVERSE_MOD(s, q)

u1 := MOD(h·w, q)

u2 := MOD(r·w, q)

add(mult(P, u1 , a, p), mult(Q, u2 , a, p), a, p)=
[48658937662937160850438599871825360069932407007 2,
21188411048200272883710140506138933636888264461 2]

MOD(48658937662937160850438599871825360069932407007 2, q) =
190793782729707067819154060120664063565713 = r

Die Signatur ist somit gültig, und Bob kann sich sicher sein, dass die Nachricht wirklich von Alice gesendet wurde.

Nun da wir die Theorie in einem praktischen Beispiel umgesetzt haben, sehen wir, dass die digitale Signatur mit elliptischen Kurven wirklich genau so funktioniert, wie wir es erwartet haben. Im nächsten und letzten Kapitel dieser Arbeit wollen wir noch kurz die Effizienz und Sicherheit der einzelnen vorgestellten Verfahren betrachten.

7 Vergleich der verschiedenen Verfahren

Wir haben im Lauf dieser Arbeit nun mehrere Public- Key Verfahren vorgestellt, und wollen nun betrachten, wie sicher diese Verfahren im Vergleich zu einander sind.

Sicherheit

Alle vorgestellten Verfahren ziehen ihre Sicherheit aus der Schwierigkeit eines der drei folgenden Probleme zu lösen:

- Faktorisierungsproblem: Das RSA Verfahren beruht auf diesem Problem.
- (allgemeines) DL- Problem: Sowohl der Diffie- Hellman Schlüsselaustausch, als auch das ElGamal- Verfahren und der DSA basieren auf diesem Problem
- ECDL- Problem: Auf dem Problem des Lösens des DL- Problems auf einer elliptische Kurve basieren der EC- Diffie-Hellman, der EC- ElGamal oder der ECDSA- Algorithmus

Allgemein kann man sagen, dass es sowohl zum Lösen des Faktorisierungsproblems, als auch zum Lösen des allgemeinen DL- Problems bereits Methoden mit subexponentieller Laufzeit gibt. Derzeit gibt es aber noch keine derartige Methode zum Lösen des ECDL- Problems. Man kann daher annehmen, dass Verfahren, die auf elliptischen Kurven basieren, bei einem gleichen Level der Sicherheit, Schlüssel von geringerer Größe benötigen, als Verfahren, die auf den beiden anderen Problemen beruhen. Diese Tatsache wird durch folgende Tabelle veranschaulicht.

Zeit in MIPS-Jahren[4] um den Schlüssel zu knacken	RSA/ DSA Schlüssellänge in Bits	ECC Schlüssellänge in Bits	Verhältnis RSA / ECC
10^4	512	106	4,8 : 1
10^8	768	132	5,8 : 1
10^{12}	1024	160	6,4 : 1
10^{20}	2048	210	9,8 : 1

Tabelle 1[5]

Wir sehen, dass die Tatsache, dass man zum Beispiel mit der Index- Calculus- Methode ein DL-Problem in subexponentieller Zeit lösen kann, sofort die Sicherheit der klassischen Verfahren verschlechtert.

Effizienz

Wenn wir die Effizienz der einzelnen Verfahren miteinander Vergleichen wollen, so können wir dies nur unter Berücksichtigung verschiedener Kriterien.

- **Rechenaufwand**: Wenn wir zum Beispiel RSA mit ECC vergleichen, so stellt sich heraus, das RSA wegen der relativ geringen Größe des Public-Keys zwar beim Verschlüsseln schnell ist, aber durch den großen Private Key beim Entschlüsseln

[4] MIPS steht für Million Instructions Per Second
[5] Vgl. [Elliptische Kurven, Mirbach, S122f]

und Signieren viel langsamer ist als ECC. In Summe ist ECC ungefähr um den Faktor 10 schneller als RSA[2].

- **Schlüssellänge**: Wir haben bereits gesehen, dass bei identischer Sicherheit, die Länge des Schlüssels sehr variiert. Betrachten wir nun das Verhältnis etwas genauer und unterteilen die Gesamtlänge des Schlüssels noch in die Länge des jeweiligen Public- bzw. Private-Keys.

	System Parameter in Bits	Public- Key in Bits	Private- Key in Bits
RSA	---	1088	2048
DSA	2208	1024	160
ECC	481	161	160

Tabelle 2[6]

- **Bandbreite**: Als letzten Aspekt betrachten wir nun die Menge an Bits bei der Verschlüsselung einer Nachricht und deren Signatur übertragen werden müssen. Bei langen Nachrichten zeigt sich, dass sich die einzelnen Verfahren was die übertragene Datenmenge betrifft nicht wirklich von einander unterscheiden. Bei kurzen Nachrichten (wo zum Beispiel mit dem Verfahren nur der Schlüssel für ein anderes Verfahren wie DES übertragen wird) sind aber sehr wohl große Unterschiede zu bemerken.

	Größe der Signatur einer 2000 Bit Nachricht in Bit	Größe einer verschlüsselten Nachricht von 100 Bit in Bit
RSA	1024	1024
DSA/ ElGamal	320	2048
ECC	320	321

Tabelle 3[6]

Auf Grund dieser Daten kann man nun ECC als sinnvolle Alternative zu herkömmlichen Verfahren wie RSA betrachten. Besonders da derzeit niemand eine subexponentielle Methode gefunden hat das ECDL- Problem zu lösen, ist ECC in praktisch allen Bereichen RSA oder DSA überlegen.

[6] vgl. [Elliptische Kurven, Mirbach, S124]

8 Ergänzungen

In dieser Arbeit wurden eine Reihe wichtiger Bereiche, die ebenfalls noch zum Thema passen würden, nur sehr kurz angeschnitten oder gar nicht behandelt, da sonst der Umfang viele hunderte Seiten länger geworden wäre. In diesem Kapitel wollen wir nun noch einen kurzen Blick auf einige wenige Aspekte von elliptischen Kurven und ECC werfen, die zu interessant sind, um sie ganz unerwähnt zu lassen.

8.1 Elliptische Kurven über F_{2^m}

Wir haben bis jetzt elliptische Kurven immer nur über einen Restklassenring mod *p*, mit *p* als einer großen Primzahl betrachtet. Elliptische Kurven über F_{2^m} wurden noch nicht behandelt, obwohl diese Art von elliptischen Kurven in der Praxis ebenfalls ungemein wichtig ist.

8.1.1 Die Gruppe F_{2^m}

Betrachten wir zu erst die Gruppe F_{2^m} und die Darstellung ihrer Elemente. Es gibt mehrere Möglichkeiten ein *g*, mit $g \in F_{2^m}$ zu definieren. Wir entscheiden uns hier für eine Darstellung als Polynom der Form

$$g \in F_{2^m} = \left\{ a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1 x + a_o \right\}$$

mit $a_i \in \{0,1\}$.

Es gibt noch weitere Möglichkeiten der Darstellung, doch wegen der relativen Einfachheit der mathematischen Operationen, die wir in Folge benötigen werden, entscheiden wir uns für diese Version.

Satz 8.1. Die Gruppe $G = F_{2^m}$ ist zyklisch und besitzt einen Generator *g* mit $g \in G$ und $g \neq 0$. Es gilt:

$$G = \left\{ g^1, g^2, ..., g^n \right\}$$

mit $n = 2^m - 1$. (siehe [Pollard-Rho, Nicolas Lamb, S4])

Das bedeutet, dass sämtliche Elemente von G als Potenzen von *g* dargestellt werden können.

Nachdem wir definierten, wie Elemente von G auszusehen haben, betrachten wir nun die Addition und Multiplikation von Elementen in G:

a) Addition:

Da die einzelnen Elemente, wie oben definiert Bit- Strings sind ist es extrem einfach zwei Elemente *g* und *h* zu addieren. Die Rechenoperation entspricht einer Bit- weisen XOR Operation.

Betrachten wir ein einfaches Beispiel in F_{2^4}:

$g = 0110$ und $h = 0101$ $\qquad\qquad$ $g + h = 0011$

Weiters gilt, dass $g + g = g - g = 0$. Wir sehen, dass Addition und Subtraktion in dieser Gruppe äquivalent sind.

b) Multiplikation:

Ein wenig komplizierter sieht es mit der Multiplikation von zwei Elementen *g* und *h* von *G* aus. Wir haben gesehen, dass Elemente von *G* binäre Strings der Länge *m* sind. Doch was ist, wenn das Ergebnis einer Multiplikation ein binärer String mit einer größeren Länge als *m* ist?

Werfen wir zu erst einen Blick auf ein kleines Beispiel, um das Problem zu verdeutlichen.

G ist wieder F_{2^4} und $g = 0010$: somit ergibt sich für $g^2 = 0100$ und $g^3 = 1000$.

Doch wie stellen wir g^4 dar? Wir können schließlich nicht $g^4 = 10000$ annehmen, da dies nicht mehr ein Element von G ist.

Wir lösen dieses Problem, indem wir für den Fall, dass eine Potenz von *g* länger als *m* Bits ist, diese Potenz um ein *f(x)* reduzieren. Dieses *f(x)* muss eine Ordnung von *m* haben und darf nicht über *G* reduzierbar sein. Das bedeutet, dass man *f(x)* über *G* nicht in zwei Funktionen geringerer Ordnung faktorisieren kann.

Für unser Beispiel mit F_{2^4} und $g = 0010$ wählen wir $f(x) = x^4 + x + 1 = 10011$.

Daraus ergibt sich für g^n:

$g^0 = 0001$	$g^1 = 0010$	$g^2 = 0100$	$g^3 = 1000$
$g^4 = 0011$	$g^5 = 0110$	$g^6 = 1100$	$g^7 = 1011$
$g^8 = 0101$	$g^9 = 1010$	$g^{10} = 0111$	$g^{11} = 1110$
$g^{12} = 1111$	$g^{13} = 1101$	$g^{14} = 1001$	$g^{15} = 0001$

An diesem Beispiel können wir gut erkennen, dass unsere zuvor gefällten Aussagen über das zyklische Verhalten von *G* richtig sind. Weiters sehen wir dass wir tatsächlich mit Hilfe einer beliebigen Funktion der Ordnung *m*, die jedoch nicht reduzierbar sein darf, Elemente von *G* multiplizieren können.

(BSP vgl. [Lamb, S5])

Definition 8.2. Ist *g* ein Element von F_{p^m} und sind die Elemente von $\left\{g, g^2, g^{2^2}, ..., g^{2^{m-1}}\right\}$ linear unabhängig, so bilden diese m Elemente eine **Normalbasis** in F_{p^m}. Für jeden Körper F_{p^m} existiert eine Normalbasis über F_p.

Betrachten wir nun ein einfaches Beispiel. Wir wählen als unsere Gruppe wieder F_{2^4} und erhalten somit für unsere Normalbasis $\{a_1 g + a_2 g^2 + a_3 g^4 + a_4 g^8\}$. Wir wollen nun diese Normalbasis quadrieren:

$$\begin{aligned}&\left(a_1 g + a_2 g^2 + a_3 g^4 + a_4 g^8\right)^2 = \\ &= a_1{}^2 g^2 + a_2{}^2 g^4 + a_3{}^2 g^8 + a_4{}^2 g^{16} = \\ &= a_1 g^2 + a_2 g^4 + a_3 g^8 + a_4 g^{16} = \\ &= a_4 g + a_1 g^2 + a_2 g^4 + a_3 g^8\end{aligned}$$

Dies gilt, da $g^{16} = g$ und $a_i \in \{0,1\}$ wodurch $a_i{}^2 = a_i$ gilt. Wir sehen, dass das Quadrieren einer Normalbasis einfach nur dem Verschieben der Faktoren nach rechts entspricht. Die Rechenoperation des Quadrierens ist somit für Normalbasen sehr unaufwendig.

8.1.2 Punkte auf elliptischen Kurven über F_{2^m}

Bis jetzt haben wir immer elliptische Kurven über F_p betrachtet. Wir erinnern uns, dass diese Kurven immer eine Gleichung der Form:

$$y^2 = x^3 + ax + b \bmod p$$

mit *a, b* aus (1,…p-1) hatten.
Für elliptische Kurven über F_{2^m} gilt nun:

$$y^2 + xy = x^3 + ax + b$$

mit $a, b \in F_{2^m}$ und $b \neq 0$.
Äquivalent zu Kurven über den Restklassenring mod *p*, gilt auch für diese Art von Kurven, dass sie Koordinaten *x* und *y* eines Punktes *P(x,y)* der Kurve, die Kurvengleichung erfüllen müssen. Des Weiteren sind *x* und *y* Elemente von F_{2^m}**.** Auch hier gibt es einen unendlich weit entfernten Punkt *O* für den $P + O = P$ gilt.

Die **Addition von Punkten** einer elliptischen Kurve über F_{2^m} sei wie folgt definiert:

Definition 8.3. Für einen beliebigen Punkt $P(x_1, y_1)$ sei $-P(x_1, x_1 + y_1)$ und $P - P = O$. Für zwei unterschiedliche Punkte $P(x_1, y_1)$ und $Q(x_2, y_2)$ mit $Q \neq -P$ gilt:

$$P + Q = (x_3, y_3)$$

mit

$$\begin{aligned}x_3 &= \left(s^2 + s + x_1 + x_2 + a\right) \\ y_3 &= \left(s(x_1 + x_3) + x_3 + y_1\right)\end{aligned}$$

und

$$s = \left(\frac{y_1 - y_2}{x_1 + x_2}\right)$$

Für den Fall $P = Q$ gilt:

$$2P = (x_3, y_3)$$

mit

$$x_3 = (s^2 + s + a)$$
$$y_3 = (x_1 + (s+1) \cdot x_3)$$

und

$$s = x_1 + \frac{y_1}{x_1}$$

Man darf jedoch nicht vergessen die entstehenden Polynome gegebenenfalls immer mit unserem *f(x)* zu reduzieren. (vgl. [Lamb, S6])

Wir sehen, dass das Rechnen auf elliptischen Kurven über F_{2^m} sehr ähnlich abläuft, wie das in den vorangegangenen Kapiteln beschriebene Rechnen über F_p.

Um der Bedeutung dieser Art von elliptischen Kurven wirklich gerecht zu werden, müssten wir nun alle in den vorangegangenen Kapiteln besprochenen Verfahren und Probleme, auf diese Art der Kurven umlegen. Doch an dieser Stelle wollen wir abbrechen. Es wäre viel zu aufwendig die beiden Arten von elliptischen Kurven bezüglich ihrer Vor- und Nachteile miteinander zu vergleichen. Dieser kurze Einblick in das Thema muss im Hinblick auf den Umfang dieser Arbeit vorerst reichen.

8.2 Kryptographische Hashfunktionen

Ein weiterer wichtiger Themenbereich, der im Laufe des Kapitels über die digitale Signatur kurz angeschnitten wurde, ist die kryptographische Hashfunktion.. Auch über dieses Kapitel könne man mehrere Doktorarbeiten schreiben, ohne auch nur andeutungsweise alle Bereiche dieses Themas erschöpfend behandelt zu haben.
Wir wollen daher nur kurz klären, was eine Hashfunktion ist und welche Eigenschaften sie hat. Dann wollen wir SHA-1 als Beispiel für eine kryptographische Hashfunktion vorstellen.

8.2.1 Allgemeine und spezielle Hashfunktionen

Zu Beginn wollen wir klären, was eine Hashfunktion eigentlich ist. Dies tun wir wie folgt:

Definition 8.4. Eine Hashfunktion *h(S)* ist eine Funktion, welche eine Zeichenfolge beliebiger Länge auf eine Zeichenfolge bestimmter Länge abbildet. Sie ist nicht injektiv.

Ein Beispiel für eine sehr einfache Hashfunktion wäre eine Funktion, die aus einem beliebig langen binären String *S* die Summe $s_1 \oplus s_2 \oplus ... \oplus s_n$ bildet. Aus *S* = 11101001 würde somit *h(S) = 1* resultieren. Bereits hier kann man erkennen, dass es möglich ist, dass verschiedene Eingabestrings dasselbe Ergebnis liefern.

Eine kryptographische Hashfunktionen sollte außerdem noch folgende Eigenschaften haben:

1) Es sollte sich bei der Hashfunktion um eine **Einwegfunktion** handeln. Das heißt, dass es praktisch unmöglich sein sollte zu einem vorgegebenen Ergebnis der Hashfunktion *h(x) = y* den passenden Eingabewert *x* zu finden.
2) Die Funktion sollte **kollisionsfrei** sein. Es soll somit praktisch unmöglich sein, zu einem gegebenen Eingabewert *x* einen zweiten Eingabewert *y* zu finden für den gilt *h(x) = h(y)*.

Die Qualität einer Hashfunktion kann nach folgenden Kriterien gemessen werden:

1) Datenreduktion: der Ausgabewert der Funktion *h(x)* sollte weniger Speicher benötigen als der Eingabewert *x*.
2) Chaotisch: ähnliche Eingabewerte x_1 und x_2, sollten sehr unterschiedliche Ausgabewerten $h(x_1) = y_1$ und $h(x_2) = y_2$ ergeben.
3) Surjektiv: die Funktion *h(x)* sollte so beschaffen sein, dass sie kein mögliches Ergebnis ausschließt. Praktisch sollte kein Ergebniswert unmöglich sein.
4) Effizienz: die Hashfunktion *h(x)* sollte mit möglichst wenig Rechen- und Speicheraufwand arbeiten.

Im nächsten Abschnitt wollen wir nun, als wohl bekanntestes Beispiel für eine kryptographische Hashfunktion, den SHA-1 Algorithmus vorstellen.

8.2.2 SHA-1

Wir erinnern uns, dass wir zur digitalen Signatur einer Nachricht eine kryptographische Hashfunktion benötigen, der wir die Nachricht m als Eingabeparameter mitgeben. Sofort erkennen wir den Sinn der von uns oben angeführten Kriterien für eine Hashfunktion. Um über eine möglichst sichere und effiziente Hashfunktion zu verfügen, entwickelte die US-amerikanische NSA in Zusammenarbeit mit dem NIST den SHS (Secure Hash Standard). In diesem Standard wurde der SHA (Secure Hash Algorithm) spezifiziert. Diese 1994 erschienene Version, die mit SHA-0 bezeichnet wird, musste auf Grund einer Schwäche 1995 etwas Umgebaut werden. Das Ergebnis dieser Korrektur ist der heute bekannte und verwendete SHA-1.

Zur Implementierung von SHA-1 benötigen wir einige kleine Hilfsfunktionen[7].
Folgende Funktionen benötigen wir zum Umwandeln von Zahlen und Zeichenfolgen in Strings der b-adischen Darstellung, beziehungsweise der Strings der b-adischen Darstellung zurück in Zahlen.

```
convert(n, b := 2, k := 0, s_ := "", t_) :=
   Loop
      If k ≤ 0 ∧ n = 0
         RETURN s_
      t_ := MOD(n, b) + 48
      s_ := ADJOIN(CODES_TO_NAME(t_ + 7·IF(t_ > 57)), s_)
      n := FLOOR(n, b)
      k :- 1

todec(h, b := 2, s_ := 0, t_) :=
   Loop
      If h = ""
         RETURN s_
      t_ := FIRST(NAME_TO_CODES(FIRST(h)))
      t_ := IF(t_ < 65, t_ - 48, MOD(t_, 32) + 9)
      s_ := b·s_ + t_
      h := REST(h)

source(t, s_ := "") :=
   Loop
      If t = ""
         RETURN s_
      s_ := APPEND(s_, convert(FIRST(NAME_TO_CODES(FIRST(t))), 2, 8))
      t := REST(t)
```

[7] Sämtliche Hilfsfunktionen und der SHA-1 selbst wurden freundlicherweise vom Betreuer dieser Arbeit, Ao.Univ.Prof. Dr. Johann Wiesenbauer, zur Verfügung gestellt, und unverändert übernommen.

Mit folgenden Funktionen definieren wir die Bit- weisen Rechenoperationen.

```
concat(v, s_ := "") :=
   Loop
      If v = []
         RETURN s_
      s_ := APPEND(s_, FIRST(v))
      v := REST(v)

and v := concat(VECTOR(IF(v_ = 49·DIM(v)), v_, ∑(NAME_TO_CODES(v_), v_, v)))

or v := concat(VECTOR(IF(v_ > 48·DIM(v)), v_, ∑(NAME_TO_CODES(v_), v_, v)))

xor v := concat(MOD(∑(NAME_TO_CODES(v_), v_, v), 2))

not(u) := concat(VECTOR(49 - u_, u_, NAME_TO_CODES(u)))

add v := convert(MOD(∑(todec(v_), v_, v), 2^32 ), 2, 32)
```

Für die Rechenoperationen in der Methode benötigen wir noch:

```
rotl(u, s := 1) := APPEND(u[s + 1, ..., DIM(u)], u[1, ..., s]           )

fsub(i, u) :=
   Prog
      If i ≤ 20
         RETURN or(and(u↓2, u↓3), and(not(u↓2), u↓4))
      If i ≤ 40 v i > 60
         RETURN xor(u↓2, u↓3, u↓4)
      If i ≤ 60
         or(and(u↓2, u↓3), and(u↓2, u↓4), and(u↓3, u↓4))
```

Die SHA-1 Methode sieht somit folgendermaßen aus:

```
SHA_1(m, b := 16, a_, c_, h_, k_, l_, t_, w_) :=
   Prog
      l_ := DIM(m)
      m := ITERATE(APPEND(m_, "0"), m_, APPEND(m, "1"), MOD(447 - l_, 512))
      m := APPEND(m, convert(l_, 2, 64))
      k_ := ["5a827999", "6ed9eba1", "8f1bbcdc", "ca62c1d6"]
      k_ := VECTOR(convert(todec(j, 16), 2, 32), j, k_)
      h_ := ["67452301", "efcdab89", "98badcfe", "10325476", "c3d2e1f0"]
      h_ := VECTOR(convert(todec(j, 16), 2, 32), j, h_)
```

Den zweiten Teil des Algorithmus findet der Leser auf der nächsten Seite.

```
Loop
   w_ := VECTOR(m↓[32·j + 1, ..., 32·j + 32], j, 0, 15)
   m := m↓[513, ..., DIM(m)]
   c_ := 16
   Loop
      t_ := xor(w_↓(c_ - 2), w_↓(c_ - 7), w_↓(c_ - 13), w_↓(c_ - 15))
      w_ := APPEND(w_, [rotl(t_)])
      c_ :+ 1
      If c_ = 80 exit
   a_ := VECTOR(h_↓j, j, 1, 5)
   c_ := 1
   Loop
      t_ := add(rotl(a_↓1, 5), a_↓5, fsub(c_, a_), w_↓c_, k_↓CEILING(c_, 20))
      [a_↓5 := a_↓4, a_↓4 := a_↓3, a_↓3 := rotl(a_↓2, 30), a_↓2 := a_↓1, a_↓1 := t_]
      c_ :+ 1
      If c_ = 81 exit
   h_ := VECTOR(add(h_↓j, a_↓j), j, 1, 5)
   If m = ""
       Prog
         h_ := concat(h_)
         If b = 2
            RETURN h_
         If b = 10
            RETURN todec(h_)
         If b = 16
            RETURN convert(todec(h_), b, 40)
            RETURN convert(todec(h_), b)
```

Genau diese Implementierung von SHA-1 wurde auch für die in den vorangegangen Kapiteln vorgestellten Verfahren zur digitalen Signatur verwendet.
Werfen wir zum Abschluss noch einen kurzen Blick auf die Ergebnisse dieser Hashfunktion.

SHA_1(source(Mister Bond sofort melden)) = 4FD65553163F47A1957D94A389536944DE7F5713
SHA_1(source(mister bond sofort melden)) = B54123AE993608523A60515C1394687D05978C0A
SHA_1(source(mister bond sofort melden!)) = 293B81B8A09FD8123C19038289727DCD5B1275AD

Wir sehen, dass minimale Änderungen des Inputs sofort zu komplett anderen Ergebnissen führen.

An dieser Stelle muss noch erwähnt werden, dass im Februar 2005 ein erster erfolgreicher Angriff auf SHA-1 unternommen wurde. Es ist gelungen einzelne Kollisionen von SHA-1 in einer realisierbaren Laufzeit zu berechnen. Seither sind weitere Schwachpunkte von SHA-1 entdeckt worden, was dazu geführt hat, dass der Algorithmus nicht mehr als 100%ig sicher angesehen werden kann.

Literaturliste

[Buchmann] Johannes Buchmann; Einführung in die Kryptographie; 3., erweiterte Auflage; Springer- Verlag, 2004

[Förster] Otto Förster; Algorithmische Zahlentheorie; Vieweg Verlag; 1996

[Koblitz]: Neal Koblitz; A Course in Number Theory; Second Edition; Springer-Verlag; 1998

[Lamb]: Nicholas Lamb; An Investigation for Pollard's Rho method for attacking Elliptic Curve Crypto Systems; April 2002
Link: http://www.cs.ualberta.ca/~nlamb/PollardRhoDiscussion.pdf

[Mirbach]: Andreas Mirbach; Elliptische Kurven, Die Bestimmung Ihrer Punktezahl und Anwendungen in der Kryptographie; Verlagshaus Monsenstein und Vannerdat; 2003

[Singh] Simon Singh; Geheime Botschaften; 5. Auflage; Deutscher Taschenbuch Verlag; 2004

[Werner]: Annette Werner; Elliptische Kurven in der Kryptographie; Springer-Verlag; 2002

[Wiesenbauer2004] Johann Wiesenbauer; Titbits 29; Derive Newsletter #55; Seite 39-56; Dezember 2004; Link: http://www.austromath.at/dug/dnl55.pdf

[Wiesenbauer2005] Johann Wiesenbauer; Titbits 30; Derive Newsletter #58; Seite 40-51; Juni 2005; Link: http://www.austromath.at/dug/dnl58.pdf

[Wiesenbauer2007] Johann Wiesenbauer; Titbits 34; Derive Newsletter #67; Seite 46-50; September 2007; Link: http://www.austromath.at/dug/dnl67.pdf