DIPLOMARBEIT

Thema

Algorithmen für elliptische Kurven und ihre Implementierung in C#

Ausgeführt am Institut für

Diskrete Mathematik und Geometrie

der Technischen Universität Wien

unter der Anleitung von

Ao. Univ. Prof. Dipl. -Ing. Dr. techn. Johann Wiesenbauer

durch

Andreas Kleinbichler
Name

Harterstrasse 31, 2640 Enzenreith
Anschrift

Ich erkläre an Eides statt, dass ich meine Diplomarbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen selbständig ausgeführt habe und alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur genannt habe.

Vorwort

In den vergangenen Jahren hat sich das Interesse der Industrie und Wirtschaft an dem Einsatz von elliptischen Kurven zu kryptographischen Zwecken gesteigert. Mathematiker und Informatiker erhielten so die Möglichkeit Themengebiete, welche zuvor als rein theoretisch und ohne Anwendung charakterisiert wurden, zu einer bedeutenden Applikation umzusetzen. Da ich beruflich Software entwickle und große Begeisterung für Zahlentheorie und Algebra empfinde konnte ich dieser Herausforderung nicht widerstehen.

Ziel dieser Diplomarbeit war es Algorithmen für elliptische Kurven in C# zu implementieren. Begonnen wurde mit verschiedenen Variationen der ECM (Elliptic Curve Method) und mit dem Punktezählen auf elliptischen Kurven. Anschließend wendete ich mich den Primalitätsbeweisen von Goldwasser und Kilian zu. Zum Abschluß habe ich noch kryptographische Anwendungen im Rahmen der Verschlüsselung und digitalen Signatur nach ElGamal ausprogrammiert. Das Produkt dieser Diplomarbeit ist eine C# - Klassenbibliothek über welche Algorithmen und Datenstrukturen für die genannten Themengebiete zu Verfügung gestellt werden.

Mein Dank gebührt Ao. Univ. Prof. Dipl. -Ing. Dr. techn. Johann Wiesenbauer für die Vergabe des interessanten Themas und die Betreuung während der Fertigstellung.

Weiters möchte ich meiner Familie danken, die mir während meines Studiums moralische Unterstützung gegeben hat.

Meinen besonderen Dank möchte ich meiner Lebensgefährtin Marion Haiden aussprechen, die durch den Verzicht auf meine Unterstützung in familiären und privaten Angelegenheiten, eine Parallelisierung von Beruf und Studium möglich gemacht hat.

Inhaltsverzeichnis

Kapitel 1

Die Arithmetik der elliptischen Kurven

Ziel dieses Abschnitts ist es die Grundlagen elliptischer Kurven über endlichen Primkörpern $\mathbb{F}_p (p \in \mathbb{P} \wedge p > 3)$ einzuführen. Zuerst werden affine und projektive kubische Kurvengleichungen vorgestellt und deren für uns wichtigsten Eigenschaften diskutiert. Dann betrachten wir die Lösungsmengen dieser Gleichungen und geben passende Koordinatendarstellungen für deren Elemente, die wir auch als Punkte der Kurve bezeichnen, an. Aufbauend auf den bis dahin erarbeiteten Eigenschaften, können wir elliptische Kurven als speziellen Kurventyp definieren und ein Gruppengesetz auf der Punktmenge angeben. Den rein theoretischen Teil dieses Kapitels schließen wir mit dem Begriff der getwisteten elliptischen Kurve und dem Satz von Hasse ab. Der folgende praktische Teil beschäftigt sich mit der Implementierung des oben angesprochenen Gruppengesetzes. Im Zuge dessen werden die grundlegenden Datenstrukturen entworfen und performancekritische Überlegungen zur Abbildung der Arithmetik angestellt. C# Codebeispiele zeigen eine Möglichkeit zur Umsetzung der Ergebnisse aus der vorangegangenen Analysephase. Als Literaturgrundlage zu diesem Kapitel diente mir [C & P] und [Werner].

1.1 Elliptische Kurven

Elliptische Kurven über $\mathbb{F}_p$ sind oberflächlich betrachtet Teilmengen des zweidimensionalen affinen (bzw. projektiven) $\mathbb{A}^2(\mathbb{F}_p)$ (bzw. $\mathbb{P}^2(\mathbb{F}_p)$) Raumes. Diese beiden Grundmengen werden an dieser Stelle definiert.

Definition 1.1.1 (Affiner und projektiver Raum) $\mathbb{K}$ stehe für einen beliebigen Körper. Die Menge

$$\mathbb{A}^2(\mathbb{F}_p) := \mathbb{K} \times \mathbb{K}$$

wird als der *zweidimensionale affine Raum über* $\mathbb{F}_p$ bezeichnet und ihre Elemente nennen wir affine Punkte.
$\sim$ bezeichne eine Äquivalenzrelation auf der Menge $\mathbb{K} \times \mathbb{K} \times \mathbb{K} \backslash \{(0,0,0)\}$, sodass für je zwei Elemente (x,y,z) und (x',y',z') aus $\mathbb{K} \times \mathbb{K} \times \mathbb{K} \backslash \{(0,0,0)\}$

$$(x,y,z) \sim (x',y',z') \Leftrightarrow \exists t \in \mathbb{K} \backslash \{0\} : (x,y,z) = (tx', ty', tz')$$

erfüllt ist. Die Menge $\mathbb{P}^2(\mathbb{F}_p) := (\mathbb{K} \times \mathbb{K} \times \mathbb{K} \backslash \{(0,0,0)\})/\sim$ wird als der *zweidimensionale projektive Raum über* $\mathbb{F}_p$ bezeichnet. Die Äquivalenzklassen $[x,y,z]$ bezüglich $\sim$ nennt man projektive Punkte.

Elliptische Kurven sind aus dem Betrachtungswinkel der algebraischen Geometrie spezielle affine oder projektive Kurven in den oben definierten Räumen $\mathbb{A}^2(\mathbb{F}_p)$ und $\mathbb{P}^2(\mathbb{F}_p)$. Im Allgemeinen werden algebraische Kurven zusammen mit einer Garbe von Funktionen definiert, was aber im Rahmen dieser Arbeit nicht weiter benötigt wird. Die folgende Definition ist ausreichend und zielgerichteter in Bezug auf die hier behandelte Thematik.

Definition 1.1.2 (affine/projektive kubische Kurve). Gegeben sei ein Polynom $f \in \mathbb{F}_p[x][y]$ der Gestalt

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

Die zugehörige Nullstellenmenge

$$C_f(\mathbb{F}_p) := \{(x,y) \in \mathbb{A}^2(\mathbb{F}_p) | f(x,y) = 0\}$$

wird im Folgenden als eine *affine ebene kubische Kurve* bezeichnet.
Durch Transformation von f zu einem homogenen Polynom $F \in \mathbb{F}_p[x][y][z]$ in drei Variablen und vom Grad 3 (d.h.: zu jedem Term $x^i y^j$ von f wird $z^{3-(i+j)}$ hinzumultipliziert) erhält man

$$F(x,y,z) := -zy^2 + x^3 + axz^2 + bz^3.$$

Die Menge der Nullstellen von F aus $\mathbb{P}^2(\mathbb{F}_p)$

$$C_F(\mathbb{F}_p) := \{(x,y,z) \in \mathbb{P}^2(\mathbb{F}_p) | F(x,y,z) = 0\}$$

definiert eine *projektive ebene kubische Kurve.*

Die Einbettung des $\mathbb{A}^2(\mathbb{F}_p)$ in den $\mathbb{P}^2(\mathbb{F}_p)$ wird durch die injektive Abbildung

$$\mathrm{j} : \mathbb{A}^2(\mathbb{F}_p) \longrightarrow \mathbb{P}^2(\mathbb{F}_p), (a,b) \longmapsto [a,b,1]$$

bewerkstelligt. Zu jedem affinen $(x, y) \in C_f(\mathbb{F}_p)$ ist durch $[x, y, 1]$ eine Lösung von $C_F(\mathbb{F}_p)$ gegeben. Umgekehrt kann aus einer projektiven Lösung $[x, y, z]$ mit $z \neq 0$ durch Bildung von $(x/z, y/z)$ eine Lösung von $C_f(\mathbb{F}_p)$ gewonnen werden. Für die Punkte $[x, y, z]$ mit $z = 0$, diese werden auch als unendlich ferne Punkte bezeichnet, gibt es keine korrespondierenden affinen Lösungen, und daher bezüglich $\mathfrak{j}$ keine Urbilder in $\mathbb{A}^2(\mathbb{F}_p)$.

Es ist auch bemerkenswert, dass sich jede affine Gleichung der Form

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

durch die lineare Substitution in x

$$x \mapsto x - \frac{c}{3}$$

in die Gestalt aus Definition 1.1.2 überführen lässt. Für uns stellt es daher keine wesentliche Einschränkung dar, theoretische Betrachtungen auf diesen speziellen Gleichungstyp zu beschränken.

Eine weitere wichtige Eigenschaft dieser algebraischen Kurven ist das Singularitätsverhalten. Dazu folgende Definition:

Definition 1.1.3 (Singularitätsverhalten affiner und projektiver Kurven). Die affine (projektive) Kurve $C_f(\mathbb{F}_p)$ ($C_F(\mathbb{F}_p)$), mit

$$f(x, y) = -y^2 + x^3 + ax + b \quad (F(x, y, z) = -zy^2 + x^3 + az^2x + z^3b)$$

heißt singulär in dem Punkt $P := (a, b) \in \mathbb{A}^2(\mathbb{F}_p)$ ($(a, b, c) \in \mathbb{P}^2(\mathbb{F}_p)$) wenn

$$\frac{\partial f}{\partial x}(a, b) = \frac{\partial f}{\partial y}(a, b) = 0 \quad (\frac{\partial F}{\partial x}(a, b, c) = \frac{\partial F}{\partial y}(a, b, c) = \frac{\partial F}{\partial z}(a, b, c) = 0)$$

gilt. Ist $C_f(\overline{\mathbb{F}}_p)$ ($C_F(\overline{\mathbb{F}}_p)$) in keinem Punkt singulär, dann nennt man $C_f(\mathbb{F}_p)$ ($C_F(\mathbb{F}_p)$) eine nicht singuläre Kurve.

Für $C_f(\mathbb{F}_p)$ ist die fehlende Singularität äquivalent zu $\triangle := 4a^3 + 27b^2 \neq 0$, wobei $\triangle$ auch als die Diskriminante der Kurve $C_f(\mathbb{F}_p)$ bezeichnet wird. Weiters gilt für einen nicht unendlich fernen Punkt P einer projektiven Kurve $C_F(\mathbb{F}_p)$ für den es ein $Q \in C_f(\mathbb{F}_p)$ mit $\mathfrak{j}(Q) = P$ gibt : $C_F(\mathbb{F}_p)$ ist singulär in $P = \mathfrak{j}(Q)$ genau dann wenn die affine Kurve $C_f(\mathbb{F}_p)$ singulär in Q ist. F ist dabei die Transformation von f zu einem homogenen Polynom nach Definition 1.1.2.

Wir haben nun die notwendigen Basisbegriffe eingesammelt und können die vollständige Definition einer elliptischen Kurve angeben.

Definition 1.1.4 (Elliptische Kurve) Gegeben sei eine nicht leere affine ebene kubische Kurve $C_f(\mathbb{F}_p) \subset \mathbb{A}^2(\mathbb{F}_p)$ mit $f(x,y) = -y^2+x^3+ax+b \quad mit \quad a,b \in \mathbb{F}_p$. Weiters gelte die Bedingung $\triangle := 4a^3 + 27b^2 \neq 0$, welche zur fehlenden Singularität der Kurve äquivalent ist. Die Menge $E_{a,b}(\mathbb{F}_p) := C_f(\mathbb{F}_p) \cup \mathcal{O}$ bezeichnet man als elliptische Kurve über $\mathbb{F}_p$. $\mathcal{O}$ steht hier für den aus dem projektiven Fall bekannten, unendlich fernen Punkt.

Für die Elemente aus $E_{a,b}(\mathbb{F}_p)$ läßt sich ein Gruppengesetz mit einem breitem Anwendungsgebiet angeben. Die binäre Gruppenoperation bezeichnen wir mit + und die Gruppeninversion eines Elementes mit dem vorangestelltem unären Operator −. Diese Abbildungen werden jetzt definiert.

Definition 1.1.5 (Gruppengesetz für elliptische Kurven).
Die Gruppeninversion - :

$$E_{a,b}(\mathbb{F}_p) \longrightarrow E_{a,b}(\mathbb{F}_p),$$

$$(x,y) \longmapsto -(x,y) = \begin{cases} (x,-y), & falls \quad (x,y) \neq \mathcal{O} \\ (x,y), & falls \quad (x,y) = \mathcal{O} \end{cases}$$

Die Gruppenaddition +:

$$E_{a,b}(\mathbb{F}_p) \times E_{a,b}(\mathbb{F}_p) \longrightarrow E_{a,b}(\mathbb{F}_p),$$

$$((x_1,y_1),(x_2,y_2)) \longmapsto (x_1,y_1) + (x_2,y_2) =: (x_3,y_3),$$

$Fall (x_1,y_1) = \mathcal{O}) \vee ((x_2,y_2) = \mathcal{O})$:

$$(x_3,y_3) = \begin{cases} (x_1,y_1), & falls \quad (x_2,y_2) = \mathcal{O} \\ (x_2,y_2), & falls \quad (x_1,y_1) = \mathcal{O} \end{cases}$$

$Fall (x_1,y_1) \neq \mathcal{O}) \wedge ((x_2,y_2) \neq \mathcal{O})$:

$$(x_3,y_3) = \begin{cases} (m^2 - x_1 - x_2, -m(x_3 - x_1) - y_1), & falls \quad y_1 \neq -y_2 \\ \mathcal{O}, & falls \quad y_1 = -y_2 \end{cases}$$

$$mit \quad m = \begin{cases} \frac{y_2-y_1}{x_2-x_1}, & falls \quad x_1 \neq x_2 \\ \frac{3x_1^2+a}{2y_1}, & falls \quad x_2 = x_1 \end{cases}$$

$\mathcal{O}$ erfüllt per Definition die Eigenschaften des neutralen Gruppenelementes. Die Existenz der Inversen ist durch den y^2 Term in der Kurvengleichung, der das einzige Vorkommen der Variable y in der zugrunde liegenden Gleichung ist, gewährleistet ($\Leftarrow ((x,y) \in E_{a,b}(\mathbb{F}_p) \Rightarrow (x,-y) \in E_{a,b}(\mathbb{F}_p))$). Das Kommutativgesetz kann direkt aus der Definition heraus nachgerechnet werden. Am aufwendigsten ist der Beweis der Assoziativität, der hier entfallen soll.

In den meisten Anwendungen des Gruppengesetzes wird die Potenzbildung in der Gruppe $E_{a,b}(\mathbb{F}_p)$, welche in der Literatur als elliptische Multiplikation bezeichnet wird, exzessiv verwendet.

Definition 1.1.6 (Elliptische Multiplikation). Die Potenzbildung in der Gruppe $E_{a,b}(\mathbb{F}_p)$ wird als elliptische Multiplikation bezeichnet. Es gilt für einen Punkt $P \in E_{a,b}(\mathbb{F}_p)$ und ein $n \in \mathbb{N}$

$$[n]P := \underbrace{P + \ldots + P}_{n}$$

Ein weiterer wichtiger Begriff ist der der getwisteten elliptischen Kurve.

Definition 1.1.6 (Quadratischer Twist) Gegeben sei eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$ und $g \neq 0 \in \mathbb{F}_p$. Die Gleichung $gy^2 = x^3 + ax + b$ definiert eine elliptische Kurve, die man als den quadratischen Twist von $E_{a,b}(\mathbb{F}_p)$ bezüglich g bezeichnet.

Die affine Darstellung des quadratischen Twists bezüglich g hat die Gleichung $Y^2 = X^3 + g^2aX + g^3b$ unter Berücksichtigung von $Y = g^2y$ und $X = gx$. Weiters existiert bis auf Isomorphie, nur ein quadratischer Twist, der nicht isomorph zu $E_{a,b}(\mathbb{F}_p)$ ist. Dieser eindeutige Twist lässt sich mit einem g, das kein Quadrat in $\mathbb{F}_p$ ist, erzeugen.

Ein in den folgenden Kapiteln häufig verwendetes Resultat über $\#E_{a,b}(\mathbb{F}_p)$ ist der Satz von Hasse.

Satz 1.1.7 (Hasse) Für die Kardinalität von $E_{a,b}(\mathbb{F}_p)$ gilt

$$p + 1 - 2\sqrt{p} \quad < \quad \#E_{a,b}(\mathbb{F}_p) \quad < \quad p + 1 + 2\sqrt{p}.$$

Das Intervall $(p+1-2\sqrt{p}, \quad p+1+2\sqrt{p})$ nennt man auch das *Hasse-Intervall* der elliptischen Kurve $E_{a,b}(\mathbb{F}_p)$.

Ergänzend zu dem Satz von Hasse möchte ich noch ein Resultat über die Anzahl von Punkten auf elliptischen Kurven bringen. $E_{a,b}(\mathbb{F}_{q^n})$ bezeichne eine elliptische Kurve mit Koeffizienten $a, b \in \mathbb{F}_p$ und Punkten aus $\mathbb{F}_{q^n} \times \mathbb{F}_{q^n}$. Weiters gelte $N_n := \#E_{a,b}(\mathbb{F}_{q^n})$ für $n \in \mathbb{N}, n \geq 1$. Für jede elliptische Kurve über $\mathbb{F}_p$ sei die zugehörige Zeta-Funktion durch

$$\zeta(T) = exp\Big(\sum_{n\geq 1} \frac{N_n}{n} T^n\Big)$$

definiert. Kennt man nun N_1 einer Kurve, so kann man für die assoziierte Zeta-Funktion eine einfache Darstellung finden, welche die Berechnung von N_n für $n \geq 2$ ermöglicht. (vgl. [Zulfaj, S 92]). Schließlich gelangt man zu der Aussage

$$\#E_{a,b}(\mathbb{F}_{q^n}) = q^n + 1 - c_n \text{ für } n \geq 1,$$

wobei die c_n aus der Rekursion

$$c_n = c_1 c_{n-1} - q c_{n-2},\ c_0 = 2, c_1 = q + 1 - N_1$$

berechnet werden können. c_1 bezeichnet man auch als die Spur des Frobenius für q (vgl. Kapitel 3).

1.2 Die Implementierung der Arithmetik

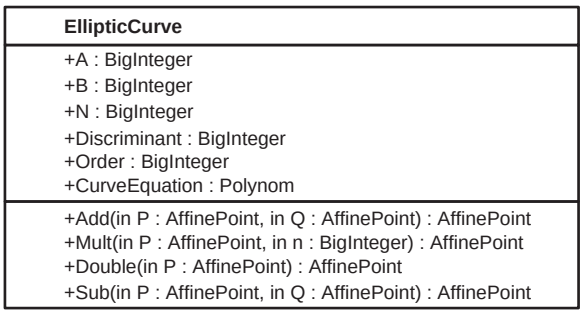
Dieser Abschnitt setzt sich mit dem Gruppengesetz der elliptischen Kurven auseinander. Wir werden Realisierungen in C♯ für die elliptische Multiplikation und Addition angeben. Als Darstellungsformen für die Punkte behandeln wir die Montgomery-Koordinaten und affine Koordinaten.

1.2.1 Das Design der Datenstrukturen

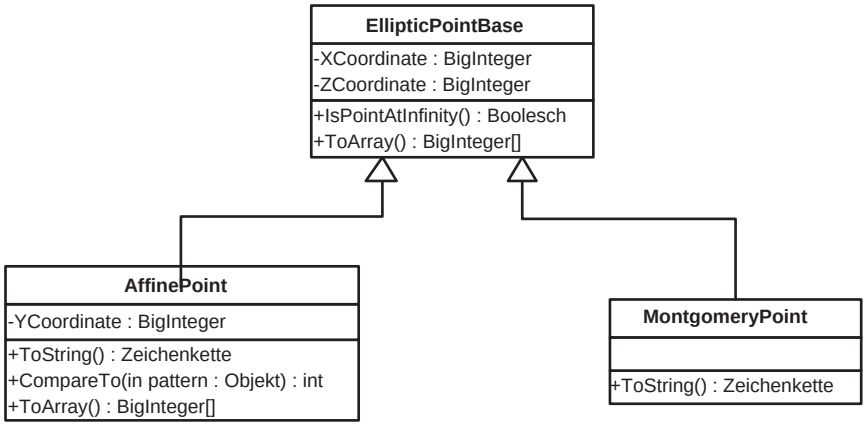
Bevor wir Algorithmen für die Arithmetik der elliptischen Kurven implementieren können, müssen noch geeignete Datenstrukturen entwickelt werden. Um eine saubere Trennung von Algorithmen und ihren Verarbeitungsdaten zu erhalten, habe ich das Gruppengesetz und die Kurvenparameter in der Klasse `EllipticCurve` gekapselt. Die Punkte werden in den Containerklassen `AffinePoint` und `MontgomeryPoint`, die von `EllipticBasePoint` erben abgebildet. Wir werden die wichtigsten Member dieser Klassen kurz vorstellen, damit sie weiter unten in C♯ Codebeispielen verwendet werden können.

EllipticCurve: Instanzierungen dieser Klasse vertreten das mathematische

Objekt einer elliptischen Kurve. Sie verfügen über Kurvenparameter und über Methoden mit denen die Arithmetik durchgeführt werden kann.



AffinePoint, MontgomeryPoint und EllipticBasePoint: Diese Klassen stellen die benötigten Datenstrukturen für die Punkte einer elliptischen Kurve zur Verfügung.



1.2.2 Affine Koordinatendarstellung

Bis jetzt haben wir immer stillschweigend affine Koordinaten zur Darstellung von Punkten ungleich $\mathcal{O}$ auf elliptischen Kurven verwendet, zu deren Realisierung Paare $(x, y) \in \mathbb{A}^2(\mathbb{F}_p)$ ausreichen würden. Für den unendlich fernen Punkt $\mathcal{O}$ würde man eine spezielle Kennzeichnung benötigen, um ihn in jedem Fall von den übrigen affinen Punkten unterscheiden zu können. Aus diesem Grund habe ich mich entschlossen, die affinen Punkte mit dem Member `ZCoordinate` zu versehen. `ZCoordinate` erhält den Wert 0 genau

dann, wenn der darzustellende Punkt gleich $\mathcal{O}$ ist, andernfalls wird 1 gesetzt.

Die **Addition** zweier affiner Punkte kann direkt aus der Definition implementiert werden, wie das folgende Codebeispiel zeigt. Es handelt sich um eine private Methode auf Instanzenebene der Klasse `EllipticCurve`, die als Eingangsparameter die Koordinaten zweier affiner Punkte hat.

```
1  private BigInteger[] AffineAdd(BigInteger X1, BigInteger Y1,
2                                 BigInteger Z1, BigInteger X2,
3                                 BigInteger Y2, BigInteger Z2)
                                      {
4    // the slope for the addition
5    BigInteger m;
6    // first point is zero - element
7    if (Z1 == BigInteger.Zero) return new BigInteger[] { X2, Y2
         , Z2 };
8    // second point is zero - element
9    if (Z2 == BigInteger.Zero) return new BigInteger[] { X1, Y1
         , Z1 };
10   // x - coordinates match
11   if (X1 == X2 && X1.isNegative == X2.isNegative) {
12         // points are inverse
13         if (((Y2 + Y1) % N) == BigInteger.Zero)
14             return new BigInteger[] { 0, 1, 0 };
15         m = (((3 * (X1 ^ (2))) + A) *
16             ((2 * Y1).nummodInverse(N))) % N;
17   } else {
18         // most common case
19         m = ((Y2 - Y1) * ((X2 - X1).nummodInverse(N))) % N;
20   }
21   // compute x - coordinate of result
22   BigInteger X3 = ((m ^ (2)) - X1 - X2) % N;
23   return new BigInteger[] { X3, (m * (X1 - X3) - Y1) % N
         , 1 };
24 }
```

Für die **elliptische Multiplikation** eines affinen Punktes P kann man verschiedene Methoden zur Potenzbildung in Gruppen heranziehen (z.B.: Square and Multiply Methode, Verwendung der NAF Darstellung des Exponenten, ...). Die Gruppe $E_{a,b}(\mathbb{F}_p)$ hat die Eigenschaft, dass die Inversenbildung sehr einfach und mit geringem Aufwand durchzuführen ist. Denn um das zu P inverse Element zu konstruieren, reicht es aus das Vorzeichen der Y-Koordinate von P zu ändern ($\Leftarrow (P = (x, y) \Rightarrow -P = (x, -y))$). Demnach kann man die Subtraktion ($P - Q = P + (-Q)$) und Addition als zwei vom Kostenstandpunkt aus gesehen äquivalente Operationen betrachten. Für Gruppen mit dieser Eigenschaft bietet sich zur Potenzbildung ein

Additions - Subtraktionsschema an.

Angenommen, wir wollen $[n]P$ für $n \in \mathbb{N}$ berechnen. Um ein Additions - Subtraktionsschema anwenden zu können, müssen wir eine geeignte vorzeichenbehaftete Zahlendarstellung für den Exponenten n finden. Da in dem konkretem Fall die Berechnung von $[2]P$ vorteilhaft ist und wir das in der Additions - Subtraktionskette verstärkt verwenden möchten, erscheint es sinnvoll eine Zahlendarstellung zur Basis 2 mit Ziffern aus $\{0, 1, -1\}$ zu verwenden. Ein beliebiges $k \in \mathbb{N}$ hat dann eine Kodierung der Form

$$k = \sum_{i \geq 0}^{\lceil ld(k) \rceil} \alpha_i 2^i \quad mit \quad \alpha \in \{0, 1, -1\}.$$

Es stellt sich nur noch die Frage, wie wir die Werte der α_i festlegen, da es dafür mehrere Möglichkeiten gibt. Eine Lösung ist, für den Exponenten n aus der einfachen Beziehung $n = \frac{3n-n}{2}$, eine Zahlenkodierung nach dem obigen Muster herzuleiten.

1. Berechnung von $3n$.

2. Bildung der Bitvektoren von $3n$ und n, wobei n durch Hinzufügen von führenden 0-Bits auf die gleiche Länge wie $3n$ gebracht wird. Anschließend wird eine modifizierte Subtraktion $3n \ominus n$ bitweise durchgeführt. Für diese modifizierte Subtraktion gilt

$$\ominus : \{0,1\}^l \times \{0,1\}^l \longrightarrow \{0,1,-1\}^l, \quad (bv1, bv2) \longmapsto bv1 \ominus bv2 =: bv3$$

$$bv3_i = sub(bv1_i, bv2_i) \quad i \in \{1, \ldots, l\} \quad mit$$

$$sub : \{0,1\} \times \{0,1\} \longrightarrow \{0,1,-1\}, (u, v) \longmapsto sub(u, v),$$

$$sub(u,v) = \begin{cases} 0 & u = 1, \quad v = 1 \\ 0 & u = 0, \quad v = 0 \\ -1 & u = 0, \quad v = 1 \\ 1 & u = 1, \quad v = 0 \end{cases}$$

   Das Resultat von $3n \ominus n$ ist eine vorzeichenbehaftete Kodierung von $2n$.

3. Durch ein Rightshift um eine Stelle in der Kodierung von $2n$ bekommt man schließlich die oben angeführte vorzeichenbehaftete Darstellung von n.

Das folgende Codebeispiel demonstriert das soeben Erörterte.

```
private BigInteger[] AffineMultSubAdd(BigInteger X1,
                                      BigInteger Y1,
                                      BigInteger Z1,
                                      BigInteger m) {
    // handle special case [0]P
    if (m == BigInteger.Zero) return new BigInteger
        [] { 0, 1, 0 };
    BigInteger[] result = new BigInteger[] { X1, Y1, Z1 };
    // handle special case [1]P
    if (m == BigInteger.One) return result;
    // compute 3m and get number of bits
    BigInteger threeTimesM = 3 * m;
    int bitCountOfM = m.bitCount();
    bool testBitThreeTimesM, testBitM;
    // loop through bits of 3m
    for (int i = threeTimesM.bitCount() - 2; i >= 1; i--) {
        result = AffineDouble(result[0], result[1], result
            [2]);
        testBitM = bitCountOfM < i ? false : m.testBit(i);
        testBitThreeTimesM = m.testBit(i);
        // perform subtraction
        if(testBitM && !testBitThreeTimesM)
            result = AffineSub(result[0], result[1], result
                [2],
                               X1, Y1, Z1);
        // perform addition
        if(!testBitM && testBitThreeTimesM)
            result = AffineAdd(result[0], result[1], result
                [2],
                               X1, Y1, Z1);
    }
    return result;
}
```

1.2.3 Projektive Koordinatendarstellung

Projektive Punkte können als Tripel $[x, y, z] \in \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p$ dargestellt werden. Der Zusammenhang zu affinen Punkten (vgl.: Einbettung j zeigt, dass auch in diesem Fall eine direkte Implementierung des Gruppengesetzes möglich ist. Vorteilhaft ist hierbei die freie Skalierbarkeit des projektiven Punktes, da man zur Codierung einen beliebigen Stellvertreter $\lambda[x, y, 1] = [\lambda x, \lambda y, \lambda]$ aus der Äquivalenzklasse wählen kann. Es ist daher möglich, die Inversenbildung bei der Addition vollständig zu umgehen, indem man λ so wählt, dass sich die zu invertierenden Elemente mit λ aufheben. Als besonders effektiv erweisen sich die Montgomery-Koordinaten, die aus projektiven Punkten durch Entfernen der Y - Koordinate hervorgehen. Für diese Punkte schreiben wir $[x, z]$.

Die fehlende Y - Koordinate verhindert eine direkte Umsetzung des Gruppengesetzes, da es im Allgemeinen zwei Alternativen für die Y - Koordinate gibt. Die Lösung dieses Problems besteht darin, bei der Addition zweier Montgomerypunkte deren Differenz als Eingangsparameter mit zu übergeben. Diese Idee stammt von Peter L. Montgomery und wird kurz beschrieben.

Satz 1.3.1 (Montgomery Identitäten) Gegeben sei eine elliptische Kurve der Form $gy^2 = x^3 + ax + b$ mit zwei affinen Punkten $P = (x_1, y_1)$ und $Q = (x_2, y_2)$ beide ungleich $\mathcal{O}$. Weiters bezeichne $x_\pm$ die X - Koordinate von $P \pm Q$. Für $x_1 \neq x_2$ gilt dann

$$x_+ x_- = \frac{(x_1 x_2 - a)^2 - 4b(x_1 + x_2)}{(x_1 - x_2)^2}. \tag{1.1}$$

und im Falle von $(x_1 = x_2) \wedge (ord(P) \geq 2)$

$$x_+ = \frac{(x_1^2 - a)^2 - 8bx_1}{4(x_1^3 + ax_1 + b)}.$$

Bemerkenswert an diesem Theorem ist, dass der Wert von g keinerlei Einfluss auf die Gleichungen hat. Erst für die Berechnung der Y - Koordinate zu einer gegebenen X - Koordinate würde man g benötigen.

Aufbauend auf diesem Theorem lassen sich Algorithmen für die Addition und Subtraktion von Montgomerypunkten herleiten. Für die folgenden Auslegungen gelten die Vereinbarungen:

1. $P := [X_1, Z_1], \quad Q := [X_2, Z_2]$ seien Punkte in Montgomerydarstellung einer elliptischen Kurve.

2. $P + Q := [X_+, Z_+]$ und $P - Q := [X_-, Z_-]$ stehen für die Ergebnisse der Addition und Subtraktion von P und Q.

Aus der durch Satz 1.3.1 gegebenen Identität lassen sich Darstellungen für die Summe von P und Q herleiten. Dazu muss die Gleichung (1.1) mit $(x_-)^{-1}$ multipliziert werden, um einen rationalen Ausdruck für die affine Koordinate x_+ zu erhalten. Anschließend bettet man den affinen Punkt $(x_+, ?)$ in $\mathbb{P}^2(\mathbb{F}_p)$ ein und erhält $[X_+, ?, 1]$. Diesen projektiven Punkt kann man mit dem Nenner von X_+ multiplizieren um die rationalen Ausdrücke in X_+ zu eleminieren. Die folgenden Formeln zeigen bereits das Endergebnis der Herleitung.

$$X_+ = Z_-((X_1 X_2 - aZ_1 Z_2)^2 - 4bZ_1 Z_2(X_1 Z_2 + X_2 Z_1)),$$

$$Z_+ = X_-(X_1Z_2 - X_2Z_1)^2.$$

Die Fälle $(P = \mathcal{O}) \wedge (Q \neq \mathcal{O})$ und $P \neq \mathcal{O} \wedge Q = \mathcal{O}$ können mit diesen Gleichungen nicht behandelt werden. In einer Implementierung sind diese abzufragen und als Ergebnis muss der Punkt ungleich $\mathcal{O}$ retourniert werden. Für den Fall $P = Q = \mathcal{O}$ können die Identitäten verwendet werden. Hier ein C# Codebeispiel:

```
private BigInteger[] MontgomeryAdd(BigInteger X1,
                                   BigInteger Z1,
                                   BigInteger X2,
                                   BigInteger Z2,
                                   BigInteger DiffX,
                                   BigInteger DiffZ) {
    // compute montgomery sum
    return new BigInteger[]{
       (DiffZ * ((X1 * X2 - A * Z1 * Z2).modPow(2, N)) -
        4 * B * Z1 * Z2 * (X1 * Z2 + X2 * Z1)) % N,
        DiffX * ((X1 * Z2 - X2 * Z1).modPow(2, N)) % N};
}
```

Für den Spezialfall $P = Q$ vereinfachen sich die Darstellungen zu

$$X_+ = (X_1^2 - aZ_1^2)^2 - 8bX_1Z_1^3$$

und

$$Z_+ = 4Z_1(X_1^3 + aX_1Z_1^2 + bZ_1^3).$$

Diese Identitäten können für eine Doubling Funktion, die für beliebiges P auf der Kurve den Punkt $[2]P$ berechnet, verwendet werden. Auch der Spezialfall $P = \mathcal{O}$ ist ohne weiteres Zutun abgedeckt.

```
private BigInteger[] MontgomeryDouble(BigInteger X1,
                                      BigInteger Z1) {
            // compute montgomery double
            return new BigInteger[]{
               (((X1.modPow(2,N) -
                  A*Z1.modPow(2,N)).modPow(2,N)) -
                  8*B*X1*Z1.modPow(2,N)) % N,
                  4*Z1*(X1.modPow(3,N) + A*X1*Z1.modPow(2,N
                      ) +
                  B*Z1.modPow(3,N)) % N};
        }
```

Zusammenfassend können wir feststellen, dass wir zur Addition zweier Montgomerypunkte P und Q mit $P \neq Q$ die Differenz $P - Q$ benötigen.

Nun beschäftigen wir uns mit der **elliptischen Multiplikation** von Montgomery Punkten, welche wir aufbauend auf den Methoden zur Addition realisieren wollen. Die Lösung dieses Problems erfolgt durch einen Algorithmus von Lucas.

Algorithmus 1.3.2 (Lucas Kette). Gegeben sei eine Folge $(x_i)_{i\in\mathbb{N}}$ deren Folgenglieder aus der Menge M stammen. Weiters gebe es zwei Abbildungen

$$\oplus : M \times M \longrightarrow M, \quad (x_j, x_{j+1}) \longmapsto x_j \oplus x_{j+1} = x_{2j+1}$$

und

$$double : M \longrightarrow M, x_j \longmapsto double(x_j) = x_{2j}.$$

$(n_{B-1}, n_{B-2}, \ldots, n_1, n_0)$ bezeichne die Binärdarstellung von n. Das Resultat des Algorithmus ist das Paar $(x_n, x_{n+1}) \in M \times M$ mit $n \in \mathbb{N}$.

1. Initialisierung
   $(u, v) = (x_0, x_1)$;
2. Berechnung
   for(B > j $\geq$ 0) {
   if($n_j == 1$)
   $(u, v) = (u \oplus v, double(v))$;
   else
   $(u, v) = (double(u), u \oplus v)$;
   }
   return (u,v);

Definiert man die Folgenglieder, der in Algorithmus 1.3.2 benötigten Folge $(l)_{n\in\mathbb{N}}$ über $l_k := [k]P$, wobei P einen Punkt in Montgomerydarstellung bezeichnet. Verwendet man für die Abbildungen

$$l_i \oplus l_j := Addg(l_i, l_j, l_i - l_j)$$

und

$$double(l_j) := Doubleg(l_j),$$

so erhält man mit Algorithmus 1.3.2 eine Methode, mit der die elliptische Multiplikation in Montgomery-Koordinaten durchgeführt werden kann. *Addg* und *Doubleg* stehen für die Addition und *Double* - Funktion in Montgomery-Koordinaten. Zu beachtende Details können aus dem folgenden Codebeispiel entnommen werden:

```
public BigInteger[] Multh(BigInteger x, BigInteger y,
                          BigInteger m) {
    if(m == BigInteger.Zero) return new BigInteger[]{0,0};
    if(m == BigInteger.One) return new BigInteger[]{x,y};
    if(m == bigint2) return Doubleh(x, y);

    // perform elliptic multiplication via lucas chain
        algorithm
    BigInteger exponent = m.Clone();
    BigInteger[] X0 = new BigInteger[2] { x, y };
    BigInteger[] X1 = Doubleh(x, y);

    for (int i = exponent.bitCount() - 2; i > 0; i--) {
        if (exponent.testBit(i)) {
           X0 = Addh(X1[0], X1[1], X0[0], X0[1], x, y);
           X1 = Doubleh(X1[0], X1[1]);
        } else {
           X1 = Addh(X1[0], X1[1], X0[0], X0[1], x, y);
           X0 = Doubleh(X0[0], X0[1]);
        }
     }

     if (exponent.testBit(0))
         return Addh(X0[0], X0[1], X1[0], X1[1], x, y);
     else
         return Doubleh(X0[0],X0[1]);
}
```

Eine performante und gut funktionierende Implementierung der Arithmetik ist nahezu für jede Anwendung der elliptischen Kurven essentiell. Im Laufe dieser Arbeit werden noch Beispiele über den Einsatz von affinen Punkten und Montgomerypunkten gebracht und auf die Vor- und Nachteile der beiden Koordinatisierungen eingegangen.

Kapitel 2

Die Elliptic Curve Method

In diesem Abschnitt sollen die Idee, Funktionsweise und einige Implementierungen der ECM vorgestellt werden. Konstruiert wurde das Verfahren von H.W.Lenstra 1985, um Primfaktoren von natürlichen Zahlen zu finden. Nach dieser Entdeckung bestand großes Interesse an dieser neuen Methode und es wurde eifrig versucht, die ursprüngliche Version von Lenstra zu verbessern. Die bahnbrechendsten Errungenschaften gehen auf P.L. Montgomery und R. Brent zurück. Einige ihrer Ideen sind auch im Rahmen dieser Arbeit umgesetzt worden. Als Vorlage diente Lenstra die pollardsche $p-1$ Methode, die sich von der ECM durch den verwendeten Gruppentyp unterscheidet. Ein weiterer Unterschied besteht darin, dass die ECM mit verschiedenen ellip-tischen Kurven durchlaufen werden kann. So ist es möglich, nach einem erfolglosen Durchgang wieder mit einer neuen Gruppe zu starten. Diese Option ist bei der $p-1$ Methode, die ich zum Einstieg in die ECM kurz besprechen werde, nicht vorhanden. Die wichtigsten Literaturquellen zu diesem Abschnitt waren [Berger], [Brent], [C & P] und [Mont].

2.1 Die ECM als Umgestaltung der $p-1$ Methode

Wie oben bereits erwähnt wurde, handelt es sich bei der $p-1$ Methode um einen Algorithmus, der zur Faktorisierung von ganzen Zahlen verwendet wird. In Kapitel 2 bezeichnet n die zu faktorisierende Zahl und $p \in \mathbb{P} : p|n$ einen Primteiler von n. Ausgangspunkt ist die Abbildung

$$h_1 : \mathbb{Z}/n\mathbb{Z} \longrightarrow \mathbb{F}_p, a \longmapsto h_1(a) := \text{a mod p},$$

welche den Ring $\mathbb{Z}/n\mathbb{Z}$ auf den Körper $\mathbb{F}_p$ surjektiv abbildet. Findet man ein $e \in \mathbb{N}$, sodass nun für ein Element $a \in \mathbb{Z}/n\mathbb{Z}$ und dessen Bild $h_1(a) \in \mathbb{F}_p$ die

Bedingungen

$$B_1 := ord_{\mathbb{Z}/n\mathbb{Z}}(a) \nmid e$$

und

$$B_2 := ord_{\mathbb{F}_p}(h_1(a)) \mid e$$

erfüllt sind, so kann man eben für dieses a die Folgerungen

$$B_1 \Rightarrow a^e \not\equiv 1 \bmod n$$

und

$$B_2 \Rightarrow a^e \equiv 1 \bmod \text{p} \Rightarrow p|a^e - 1$$

treffen. Die Bedingungen B_1 und B_2 garantieren $p \leq ggT(a^e - 1, n) < n$, womit durch die Berechnung von $ggT(a^e - 1, n)$ ein nicht trivialer Teiler von n gefunden wird. Eine mögliche Implementierung der $p-1$ Methode besteht darin, Werte für $a \in \mathbb{Z}/n\mathbb{Z}$ und $e \in \mathbb{N}$ zu wählen und anschließend a^e mod n zu berechnen. Die Auswertung von $ggT(a^e - 1, n)$ liefert unter Gültigkeit von B_1 und B_2 einen nicht trivialen Teiler von n.

Bevor wir die $p-1$ Methode für elliptische Kurven adaptieren können, ist es zweckmäßig, den Begriff der Pseudokurve einzuführen.

Definition 2.1.1(Pseudokurve) Gegeben seien natürliche Zahlen a, b und n mit $ggT(6, n) = 1$. Die Menge

$$(\mathcal{O} \cup \{(x, y) \in \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z} : x^3 + ax + b = y^2\}) =: E_{a,b}(\mathbb{Z}/n\mathbb{Z})$$

nennt man eine elliptische Pseudokurve oder kurz Pseudokurve über dem Ring $\mathbb{Z}/n\mathbb{Z}$, wenn $ggT(4a^3 + 27b^2, n) = 1$ gilt. Ihre Elemente bezeichnet man als Punkte und $\mathcal{O}$ steht für den unendlich fernen Punkt der Pseudokurve.

Das in Definition 1.1.4 eingeführte Gruppengesetz für elliptische Kurven kann auch auf die Punkte einer Pseudokurve angewendet werden. Allerdings ist die Existenz der benötigten Inversen in dem Ring $\mathbb{Z}/n\mathbb{Z}$ nicht immer gegeben und dadurch kann es vorkommen, dass $P+Q$ für manche $P, Q \in E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ nicht definiert ist.

Im Falle der elliptischen Multiplikation eines Punktes, hängt es von dem Algorithmus zur Potenzbildung ab, ob $[n]P$ berechnet werden kann. Hierbei werden immer Zwischensummen gebildet, von deren Definierbarkeit die Durchführbarkeit einer elliptischen Multiplikation abhängt. Beispielsweise kann die Folge von Zwischensummen $P \to [2]P \to [5]P$ berechenbar sein,

während die Folge $P \to [2]P \to [4]P \to [5]P$, wegen der Undefinierbarkeit von $[2]P + [2]P = [4]P$, diese Eigenschaft nicht besitzt. Schlussendlich gilt aber für je zwei Folgen von Zwischensummen, dass diese, sofern sie kalkulierbar sind, stets dasselbe Ergebnis liefern.

Die ECM verwendet anstatt der multiplikativen Gruppen $\mathbb{Z}/n\mathbb{Z}$ und $\mathbb{Z}/p\mathbb{Z}$ die Mengen $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ und $E_{a,b}(\mathbb{F}_p)$ wobei die Punkte vorerst und ohne Beschränkung der Allgemeinheit in projektiven Koordinaten dargestellt werden ($P := (P_x, P_y, P_z) \in \mathbb{P}^2(\mathbb{F}_p)$). Hier betrachtet man die Abbildung

$$h_2 : E_{a,b}(\mathbb{Z}/n\mathbb{Z}) \longrightarrow E_{a,b}(\mathbb{F}_p), P \longmapsto h_2(P),$$

$$h_2(P) := (P_x \bmod p, P_y \bmod p, P_z \bmod p).$$

Die Koordinaten eines projektiven Punktes werden unter h_2 modulo p reduziert. Findet man nun ein $P \in E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ und ein $k \in \mathbb{N}$, die den Bedingungen

$$(B_3 := [k]P \neq \mathcal{O}) \Leftrightarrow n \nmid ([k]P)_z$$

und

$$(B_4 := [k]h_2(P) = \mathcal{O}) \Leftrightarrow p \mid ([k]h_2(P))_z$$

genügen, so liefert die Berechnung von $ggT(P_z, n)$ einen nicht trivialen Teiler von n. Die Konvertierung eines projektiven Punktes P, der die Bedingungen B_3 und B_4 erfüllt, in einen affinen Punkt ist aufgrund der Eigenschaft $ggT(n, P_z) \neq 1$ nicht möglich. Ist die Summe zweier affiner Punkte $P_{aff} + Q_{aff}$ nicht definiert, so liegt das daran, dass die Inversion bei der Berechnung von m aus Definition 1.4 nicht vollzogen werden kann. Ausschlaggebend dafür ist die fehlende Gültigkeit von $ggT(x_2 - x_1, n) = 1$ (vgl. Definition 1.1.5). Hätte man $[k]P$ in affinen Koordinaten bestimmt, so wäre die Inversion bei der Berechnung von m in Definition 1.1.5 nicht durchführbar gewesen. Verwendet man zur Ermittlung der Inversen den erweiterten euklidischen Algorithmus, so tritt der nicht triviale Faktor von n als Seitenresultat auf.

2.2 Der ECM Stage 1 Algorithmus

In diesem Teil werden wir den Grundlagenalgorithmus der ECM, welcher auch als ECM Stage 1 bezeichnet wird, theoretisch beschreiben und implementieren. Sämtliche Erweiterungen der ECM sind Algorithmen, welche die Endresultate aus dem ECM Stage 1 weiterverwenden. Der genaue Zusammenhang wird weiter unten erklärt.

Wie oben bereits erwähnt wurde, muss man einen Punkt $\overline{P}$, der die Beziehungen B3 und B4 erfüllt, finden. Dazu berechnet man $\overline{P} = [k]P$ mit

$$k = \prod_{p_i \leq B1} p_i^{a_i} \text{ mit } p_i \in \mathbb{P},\ a_i = \left\lfloor \frac{log(B1)}{log(p_i)} \right\rfloor,$$

wobei B1 $\in \mathbb{N}$ als das Stage 1 Limit bezeichnet wird. B1 stellt somit eine Schranke für den Aufwand dar und legt die Größe von k fest. Wenn nun bei der Berechnung von $[k]P$ auf einer Pseudokurve $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ eine nicht definierte Summe auftritt, dann bedeutet dies

$$h_2([k]P) = \mathcal{O} \in E_{a,b}(\mathbb{F}_p)$$

für ein $p|n$. Das impliziert auch $ord_{E_{a,b}(\mathbb{F}_p)}(P)|k$. Also speziell auch dann, wenn $\#E_{a,b}(\mathbb{F}_p)|k$ gilt kann man einen Faktor von n, unabhängig von der Wahl des Punktes P, erwarten. $\#E_{a,b}(\mathbb{F}_p)|k$ ist mit unseren Definitionen genau dann erfüllt, wenn

$$\#E_{a,b}(\mathbb{F}_p) = \prod_{i=1}^{l} p_i^{a_i} \text{ mit } p_i^{a_i} < B1,\ p_i \in \mathbb{P},\ 1 \leq i \leq l$$

gilt. Wird P in projektiver Darstellung geführt, dann gilt $P = \mathcal{O} \Leftrightarrow P_z$ mod p $= 0$. Eine Evaluierung von $ggT(P_z, n)$ liefert einen nicht trivialen Teiler $p \geq n$. Unter Verwendung von affinen Koordinaten für P äußert sich die illegale Operation dahingehend, dass die Berechnung von m, die eine Inversenbildung in $\mathbb{Z}/n\mathbb{Z}$ enthält, fehlschlägt. Als Nebenresultat der Inversion, die in der Regel über den erweiterten euklidischen Algorithmus realisiert wird, erhält man dann p. Man kann daher den ECM Stage 1 für projektive und affine Koordinaten umsetzen. In der Praxis weit verbreitet ist der Einsatz von Montgomery-Koordinaten, die sehr performante Arithmetik ermöglichen. Bei einer solchen Implementierung wird ein Punkt P in Montgomery-Darstellung gewählt. Im Anschluss berechnet man $[k]P$ und wertet $ggT([k]P_z, n)$ aus. Erhält man auf diese Weise keinen Faktor von n, so kann der ECM Stage 1 mit einer neuen Pseudokuve angestoßen werden.

Ich möchte an dieser Stelle Codebeispiele für die beiden Möglichkeiten zur Implementierung des ECM Stage 1 mit der `EllipticCurves` Libary geben. `EllipticCurves` bietet zwar im Namespace `ECM` bereits eine fertige Klasse `ECMBase` an, deren Code allerdings aus Performancegründen nicht gut lesbar ist und sich nicht zu repräsentativen Zwecken eignet. Deswegen habe ich mich hier entschieden, die Algorithmen der beiden ECM Stage 1 Alternativen als

Szenario der `EllipticCurves` Bibliothek darzustellen.

In beiden Varianten werden die Kurve und der initiale Punkt zufällig generiert. Das Verfahren wird solange wiederholt, bis ein Faktor gefunden wurde. Die Klasse `PrimenumberInformationCenter` kann durch Angabe eines Intervalls $[a, b]$ mit einer aufsteigend sortierten Liste

$$L([a, b]) := \{p \in \mathbb{P} | a \leq p \leq b\}$$

initialisiert werden. Eine genauere Erklärung von `PrimenumberInformationCenter` kann aus der Dokumentation der `BigInteger` Bibliothek in Kapitel 6 entnommen werden. Jetzt genügt es zu verstehen, dass über eine Instanz von `PrimenumberInformationCenter` die Primzahlen in einem Intervall durchiteriert werden können.

In den Methoden wird jeweils eine Pseudokurve $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ und ein Punkt $P \in E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ zufällig gewählt, um anschließend $[k]P$ berechnen zu können. Das erste Codebeispiel zeigt einen ECM Stage 1 in Montgomery-Arithmetik. Wie oben besprochen, kann ein Faktor von n gefunden werden, indem man in einem finalen Schritt $ggT(n, ([k]P)_z)$ berechnet.

```
1  public BigInteger ExecuteMontECMStage1(BigInteger n, int B1)
       {
2    EllipticCurve currCurve;
3    MontgomeryPoint M;
4    PrimenumberInformationCenter pic =
5       new PrimenumberInformationCenter(0, B1);
6    BigInteger g;
7    int currPrimePower;
8    while (true) {
9        currCurve = EllipticCurve.GetRandomCurve(n);
10       M = currCurve.FindMontgomeryPoint();
11       for (int i = 0; i < pic.Primes.Count; i++) {
12           currPrimePower = pic.Primes[i];
13           do {
14               M = currCurve.MultMont(M, pic.Primes[i]);
15           } while ((currPrimePower *= pic.Primes[i]) <= B1);
16       }
17       if ((g = M.ZCoordinate.gcd(n)) != BigInteger.One)
18           return g;
19   }
20 }
```

Das zweite Codebeispiel zeigt einen ECM Stage 1 in affinen Koordinaten. Hier ist es notwendig, die elliptische Multiplikation des affinen Punktes P zu

überwachen, da das Auftreten einer illegalen Kurvenoperation mittels einer Exception abgehandelt wird. Diese Exception ist vom Typ `NotInvertible-Exception` und kann von der Methode `BigInteger->nummodInvert()`, die versucht multiplikativ inverse Elemente in $\mathbb{Z}/n\mathbb{Z}$ zu berechnen, geworfen werden. Das Ereignis ist äquivalent zu der fehlenden Existenz eines Inversen a^{-1} in $\mathbb{Z}/n\mathbb{Z}$, wobei $a \in \mathbb{Z}/n\mathbb{Z}$ das ursprünglich zu invertierende Element bezeichnet. Eine Instanzierung von `NotInvertibleException` führt in dem Member `NotInvertibleException->Factor` das Ergebnis von $ggT(n, a)$ mit, welches entweder gleich n ist oder einen nicht trivialen Faktor von n enthält. Genau dieser Wert wird von der folgenden Implementierung zurückgegeben.

```
1  public BigInteger ExecuteAffineECMStage1(BigInteger n, int B1
       ) {
2       EllipticCurve currCurve;
3       AffinePoint P;
4       PrimenumberInformationCenter pic = new
5            PrimenumberInformationCenter(0, B1);
6       BigInteger g;
7       int currPrimePower;
8       while (true) {
9         currCurve = EllipticCurve.GetRandomCurve(n);
10        P = currCurve.FindPoint(true);
11        for (int i = 0; i < pic.Primes.Count; i++) {
12             currPrimePower = pic.Primes[i];
13             do {
14                 // execute affine arithmetic operation,
15                 try {
16                      P = currCurve.Mult(P, pic.Primes[i]);
17                 } catch (Exception ex) {
18                    if (ex is NotInvertibleException) return
19                        ((NotInvertibleException)ex).GcdResult
                              ;
20                    throw ex;
21                  }
22              } while ((currPrimePower *= pic.Primes[i]) <= B1
                    );
23         }
24    }
25 }
```

2.3 Die Komplexität der ECM

Die Einführung des ECM Stage 1 möchte ich noch mit einer Komplexitätsbetrachtung abschließen. Die erwartete Zeit für das Auffinden eines Faktors

p einer natürlichen Zahl n ist durch

$$T(p) = F_1 * \underbrace{e^{(\sqrt{2}+o(1))(\sqrt{ln(p)ln(ln(p))})}}_{F_2} \quad mit$$

$$o(1) \to 0 \text{ für } p \ \to \infty$$

gegeben. Wobei F_1 für die Kosten einer Multiplikation modulo n steht und F_2 für die Häufigkeit derselben. Die Herleitung dieses Ausdrucks kann in [Berger, S 29 - 35] nachgelesen werden. Der Faktor F_1 wird durch die Arithmetik Modulo n bestimmt, während F_2 von der Implementierung des ECM Stage 1 beeinflußt wird. Das Ausmaß von F_2 hängt für ein

$$n = \prod_{i=1}^{l} p_i \text{ mit } p_1 \geq p_2 \geq \ldots \geq p_l,\ p_i \in \mathbb{P},\ 1 \leq i \leq l$$

nur von der Größe des kleinsten Primteilers p_l ab. Die Stellenanzahl von n wirkt sich alleine auf den Faktor F_1 aus.

2.4 Die Wahl der Kurven

Um die ECM Stage 1 mehrmals hintereinander durchführen zu können, muss in jedem Schritt eine neue Pseudokurve erzeugt werden. Es bietet sich die Möglichkeit an die Parameter a und b von $E_{a,b}(\mathbb{F}_p)$ zufällig zu generieren, was aber den Nachteil mit sich bringt, dass man keinen Einfluß auf die Ordnung von $E_{a,b}(\mathbb{F}_p)$ nehmen kann. Eine Verbesserung in dieser Hinsicht ist es, nach einem Satz von Brent vorzugehen, mit dem sich eine Kurve, deren Ordnung durch 12 teilbar ist, konstruieren lässt.

Satz 2.4.1 (Kurvenkonstruktion). Man konstruiere die affine Kurve

$$C_\sigma(\mathbb{F}_p) := C_f(\mathbb{F}_p) \quad mit \quad f(x,y) = -y^2 + x^3 + C(\sigma)x^2 + x,$$

wobei für den von $\sigma \in \mathbb{F}_p \backslash \{0,1,5\}$ abhängigen Kurvenparamter C,

$$u = \sigma^2 - 5$$

$$v = 4\sigma$$

$$C(\sigma) = \frac{(v-u)^3(3u+v)}{4u^3v} - 2$$

gilt. Definiert man über $C_\sigma(\mathbb{F}_p)$ eine elliptische Kurve $E_\sigma(\mathbb{F}_p)$, so ist $\#E_\sigma(\mathbb{F}_p)$

durch 12 teilbar und entweder auf $E_\sigma(\mathbb{F}_p)$ oder deren quadratischen Twist existiert ein affiner Punkt P mit $P_x = u^3v^{-3}$.

Dieses Resultat ermöglicht es uns von einem Wert σ ausgehend, eine Kurve zu erzeugen. Von dem initialen Punkt $(u^3v^{-3}, ?)$ in affiner Darstellung können wir nicht direkt sagen, ob er der Kurve oder deren quadratischem Twist angehört. Wie in Kapitel 1 aus dem Satz 1.3.1 über die Montgomery-Identitäten hervorgeht, ist die Arithmetik für Punkte in Montgomery-Darstellung unabhängig von dem Parameter g. Aus diesem Grund ist es nicht notwendig, bei der Verwendung von Montgomery-Koordinaten die genauen Parameter der zugrunde liegenden Kurve zu kennen. Dieser Umstand prädestiniert die Montgomerypunkte für den Einsatz in dem ECM Stage 1, wenn die benötigten Kurven nach Satz 2.4.1 generiert werden. Da diese Kurven die Gleichung $y^2 = x^3+C(\sigma)x^2+x$ haben, muss die Arithmetik dementsprechend angepasst werden (vgl.: [C & P, S 296 - 297]).

2.5 Die Stage 2 Continuations

Um mit dem ECM Stage 1 sicher einen Faktor p zu finden, muss die Ordnung von $E_{a,b}(\mathbb{F}_p)$ in Primzahlpotenzen kleiner B1 zerfallen, was eine für die Praxis zu strenge Bedingung darstellt. Um diesem Problem entgegenzuwirken, kann man das Endresultat $[k]P$ eines ECM Stage 1 in einem nachfolgenden Verfahren weiterverarbeiten. Die in dieser Arbeit dazu behandelten Methoden fasst man unter dem Sammelbegriff ECM Stage 2 Continuations zusammen. Sie können dann einen Faktor von n liefern, wenn die Bedingung

$$\#E_{a,b}(\mathbb{F}_p) = q\prod_{i=1}^{l} p_i^{a_i} \text{ mit } p_i^{a_i} \leq B1,\ q > B1 \tag{2.1}$$

$$q,\ p_i \in \mathbb{P},\ a_i \in \mathbb{N} \text{ für } 1 \leq i \leq l,\ q > B1$$

gilt. $\#E_{a,b}(\mathbb{F}_p)$ hat also genau einen Primteiler, der größer als $B1$ ist. Die Erfolgsbedingung eines Verfahrens, das aus einer Kombination aus ECM Stage 1 und ECM Stage 2 Continuation hervorgeht, ist somit häufiger erfüllt als jene des ECM Stage 1.

Im Folgenden werden Continuations, die den Ideen von Brent, Montgomery und Crandall folgen behandelt. Das `EllipticCurves` Framework bietet Datenstrukturen und Controllerklassen zur Implementierung diverser Continuations auf einem höheren Abstraktionsniveau an. Es gibt aber auch fertige

Implementierungen der oben erwähnten Continuations, die in einer gemeinsamen Vererbungshierarchie stehen. Als Basisklasse aller ECM Stage 2 Continuations dient die Klasse `ECMBase`, die eine Implementierung des ECM Stage 1 enthält. In diesem Zusammenhang wird in der Methode `ECMBase->ExecuteStage1(...)` Satz 2.4.1 zur Generierung von Kurven und passenden Startpunkten in Montgomery-Darstellung verwendet. Weiters enthält ECM eine öffentliche Methode `ECMBase->ExecuteECM(...)`, mit der eine von `ECMBase` abgeleitete ECM Implementierung aufgerufen werden kann. Dazu muss in der entsprechenden Ableitung die virtuelle Methode `ECMBase->ExecuteStage2(...)` überschrieben werden. Das kann als Schnittstelle zur Einbindung einer Continuation genutzt werden. In dem folgendem Codebeispiel wird die Methode `ECMBase->ExecuteECM(...)`, die von jeder von `ECMBase` ableiteteten Continuation durchlaufen wird, vorgestellt.

Eingangsparameter ist das in Satz 2.4.1 definierte σ, das zur Erzeugung der Kurve verwendet wird. Anschließend wird die ECM Stage 1 für dieses σ ausgeführt. Das Ergebnis wird dann von einer Stage 2 Continuation, die einen spezifischen Code in einer Überschreibung von `ECMStage1->ExecuteStage2()` einbindet, weiter verarbeitet. Wird an einer Stelle des Verfahrens ein Faktor gefunden, dann werfen die entsprechenden Instanzierungen Exceptions.

```
1 public virtual void ExecuteECM(BigInteger sigma) {
2             try {
3                 ExecuteStage2(ExecuteStage1(sigma));
4             } catch (Exception ex) {
5                 // all revealed factors are comming up here
6                 if(!(ex is ThreadAbortException)) throw ex;
7             }
8 }
```

Bei den hier behandelten Continuations kann zwischen zwei verschiedenen Grundtypen unterschieden werden. Unterscheidungsmerkmal ist die Verwendung von affinen Koordinaten oder Montgomery-Koordinaten bei der ECM Stage 2 Implementierung. Für die erste Gruppierung gibt es eine gemeinsame Oberklasse `ECMAffineExtension`, über welche die affine Arithmetik bereitgestellt wird. Die Continuations von Montgomery und Brent sind hier zugeordnet. Stage 2 Continuations, die affine Koordinaten verwenden, müssen das in Montgomery-Koordinaten gegebene Resultat aus dem vorangegangenem ECM Stage 1 Durchlauf konvertieren. Dazu sind die hier angeführten Schritte notwendig.

1. Aus dem Stage 1 Resultat $Q = [X, Y]$ in Montgomery-Koordinaten wird der Wert $X_{aff} = X * Z^{-1}$ des affinen Punktes (X_{aff}, Y_{aff}) be-

rechnet. Anschließendes Einsetzen von X_{aff} in the aktuelle Kurvengleichung vom Typ

$$y^2 = x^3 + C(\sigma)x^2 + x$$

liefert einen Wert den ich etwas vorgreifend als g^{-1} bezeichne.

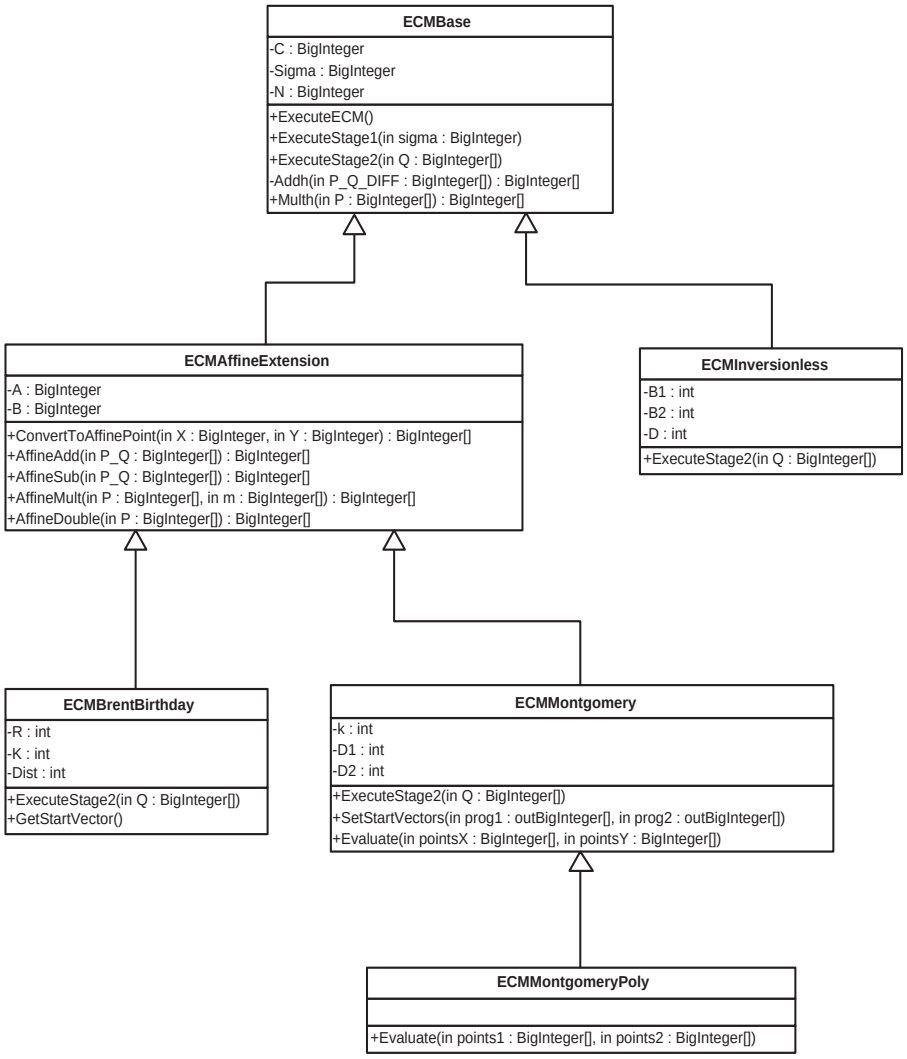
2. Setzt man Y_{aff} gleich 1, dann gilt vorerst

$$(X_{aff}, Y_{aff}) = (X_{aff}, 1) \in C_f(\mathbb{F}_p) \quad mit$$

$$f(x, y) = -gy^2 + x^3 + C(\sigma)x^2 + x$$

3. Jetzt müssen der Punkt und die Kurvengleichung noch transformiert werden, sodass g verschwindet. Diese Konvertierung wurde in Kapitel 1 im Anschluß an die Definition 1.1.6 besprochen.

Eine Continuation, die Montgomery-Koordinaten verwendet, ist durch die nach Crandall implementierte `ECMInversionless` Klasse gegeben. Hier kann das Stage 1 Endergebnis ohne zusätzliche Konvertierung weiterverwertet werden. Das `EllipticCurves` Framework stellt die Klassen `ECMMultiThreadMananger` und `ECMThreadContainer` für einen Parallelbetrieb sämtlicher `ECMBase` Ableitungen bereit. Eine Instanz von `ECMThreadContainer` referenziert höchstens ein Objekt vom Typ `ECMBase`. Die dadurch gegebene ECM Continuation wird dann in einem, von `ECMThreadContainer` zugeordneten Thread, ausgeführt und die gefundenen Faktoren werden in einer Queue abgelegt. Für die Parallelverarbeitung benötigt man mehrere Instanzen von `ECMThreadContainer`, welche von `ECMThreadManager` verwaltet werden. Da `ECMThreadContainer` mit allen Ableitungen von `ECMBase` umzugehen vermag, können mit dieser Schnittstelle Contuinations mit unterschiedlichen Typen parallel gestartet werden. Im Rahmen dieser Arbeit wurde eine Multithreadingfunktion umgesetzt, bei der die oben erwähnten Continuations nebeneinader gestartet werden.Das angeführte statische Klassendiagramm dokumentiert die Vererbungshierarchie der ECM Klassen und liefert einen Überblick über die implementierten Continuations.



2.5.1 Die Inversionless Continuation

Die Inversionless Continuation wird von R. Crandall und Carl Pomerance in ihrem Buch [C & P, S 310] angeführt. In der Methode sind verschiedene Ideen von Brent, Montgomery und Zimmermann umgesetzt. Grundlegende Strategie ist es, den einzelnen Teiler q von $\#E_{a,b}(\mathbb{F}_p)$ mit $q > B1$ durch sukzessives Probieren ausfindig zu machen. Aus Performancegründen werden sämtliche Berechnungen in Montgomery-Koordinaten durchgeführt. Für die anschließenden Überlegungen bezeichne Q das Resultat eines erfolglosen ECM Stage 1 Durchlaufs und $E_{a,b}(\mathbb{F}_p)$ die elliptische Kurve von Q.

Ziel der ECM Inversionless ist es, möglichst ökonomisch bei Gültigkeit der Stage 2 Erfolgsbedingung (2.1) jenes q zu entdecken, für das $[q]Q = \mathcal{O} \in E_{a,b}(\mathbb{F}_p)$ gilt. Da man von vornherein keine Primzahl größer B1 ausschließen kann, ist man gezwungen $[\overline{p}]Q$ für diverse $\overline{p} > B1, \overline{p} \in \mathbb{P}$ solange auszurechnen, bis q gefunden wird. Entscheiden wir uns dafür sukzessive vorzugehen, so gibt es Möglichkeiten, die Kosten der benötigten elliptischen Multiplikationen zu senken. In der hier behandelten Version der Inversionless Continuation wurden zwei Ideen umgesetzt. Eine davon ist die Reduktion einer Anzahl von elliptischen Multiplikationen zu Additionen, indem man über eine Liste auf bestimmte Vielfache des Punktes Q zugreift. Die zweite Idee führt zu einer performanten Überprüfung von $[q]Q = \mathcal{O} \in E_{a,b}(\mathbb{F}_p)$. Im Anschluß werden diese beiden Ideen genauer erklärt und eine darauf aufbauende Implementierung angegeben.

Betrachtet man nun ein Intervall $[a, b]$ mit $a, b \in \mathbb{N}$, so kann von $[a]Q$ ausgehend für jede Primzahl $\overline{p} \in [a, b]$ der Punkt $[\overline{p}]Q$ berechnet werden, indem man

$$[\overline{p}]Q = [a]Q + [\overline{p} - a]Q \quad \overline{p} \in [a, b]$$

ermittelt. Weiters ist $\overline{p} - a$ nach oben hin beschränkt durch $b - a$. Angenommen man bestimmt die Punkte

$$[\Delta_l]Q := [l]Q \quad 1 \leq l \leq b - a$$

im Vorfeld und speichert diese in der Liste L_{Δ_l}, so reicht die einfache elliptische Addition

$$[\Delta_k]Q + [a]Q \text{ mit } k = \overline{p} - a$$

zur Berechnung von $[\overline{p}]Q$ für $\overline{p} \in [a, b]$ aus. Da uns nur die Primzahlen in dem Intervall $[a, b]$ interessieren und wir für a ohne Beschränkung der Allgemeinheit eine ungerade Zahl heranziehen können, ist es ausreichend, l über alle geraden Zahlen aus $[2, b - a]$ laufen zu lassen.

Zur Überprüfung von

$$[\overline{p}]Q = \mathcal{O} \in E_{a,b}(\mathbb{F}_p) \text{ mit } \overline{p} \in \mathbb{P} \cap [a, b]$$

muss nun gezeigt werden, dass

$$1 < ggT(([\overline{p}]Q)_z, n) \leq n$$

gilt. Die Z-Koordinate von $[\overline{p}]Q$ muß durch einen Teiler ungleich 1 von n teilbar sein. Verwendet man jetzt die oben eingeführte Darstellung von

$$[\overline{p}]Q = [a]Q + [\Delta_l]Q \text{ mit } [a]Q = [X_a, Z_a],\ [\Delta_l]Q = [X_{\Delta_l}, Z_{\Delta_l}],$$

dann bietet sich an, anstatt der Z-Koordinate von $[\overline{p}]Q$ den vereinfachten Ausdruck $X_a Z_{\Delta_l} - X_{\Delta_l} Z_a$ als linkes Argument für den ggT zu verwenden. Das ist nicht ganz exakt, weil die Z-Koordinate von Q durch $X_{a-\Delta_l}(X_a Z_{\Delta_l} - X_{\Delta_l} Z_a)^2$ gegeben ist. Wir vernachlässigen also den Faktor $X_{a-\Delta_l}$, der für die X-Koordinate von $[a]Q - [\Delta_l]Q$ steht.

Jetzt müssen die beiden Ideen zu einer ECM Stage 2 Continuation integriert werden. Die Anzahl der durchzuführenden euklidischen Algorithmen sollte möglichst gering gehalten werden. Daher empfiehlt es sich, nicht für jeden Kandidaten $\overline{p}$ den ggT mit den oben angeführten Parametern aufzurufen. Um für eine Menge von Vielfachen von Q $\{[p_i]Q | i \in I\}$ die Bedingungen $[p_i]Q = \mathcal{O} \in E_{a,b}(\mathbb{F}_p)$ überprüfen zu können, ist es ausreichend $ggT(\prod_{i \in I}([p_i]Q)_z, n)$ zu berechnen. $I \subset \mathbb{N}$ steht hier für die Indexmenge der Laufvariablen i. Selbstverständlich wird auch hier die Z-Koordinate jedes Punktes aus $\{[p_i]Q | i \in I\}$ durch den zuvor eingeführten vereinfachten Ausdruck ersetzt. Das liefert vorerst die Erfolgsbedingung

$$1 < ggT(\prod_{i \in I}(X_a Z_{\Delta_{l_i}} - Z_a X_{\Delta_{l_i}}), n) < n,\ [a]Q + [\Delta_{l_i}]Q = [p_i]Q. \tag{2.2}$$

Geht man nach dieser Methode vor, so benötigt man für jeden Punkt $[a + \Delta_l]Q$ aus L_{Δ_l} 2 Multiplikationen (Modulo n), um den entsprechenden Faktor von (2.2) zu bilden. Es besteht allerdings auch hier die Möglichkeit, Multiplikationen (Modulo n) einzusparen. Dazu betrachten wir die Identität

$$(X_a Z_{\Delta_l} - Z_a X_{\Delta_l}) = (X_a - X_{\Delta_l})(Z_a + Z_{\Delta_l}) + X_{\Delta_l} Z_{\Delta_l} - X_a Z_a.$$

Entschließen wir uns in einem initialen Schritt, $X_a Z_a$ zu speichern und eine Liste der Produkte $X_{\Delta_l} Z_{\Delta_l}$ für gerade $l \in [a, b]$ anzulegen, so kann mit nur einer Multiplikation (Modulo n) für jeden Punkt aus L_{Δ_l} der Faktor für (2.2) konstruiert werden. Die Liste bezeichnen wir in weiterer Folge mit L_{prod}.

Als nächstes muss die Größe der Menge I festgelegt werden. Hier bietet sich an, die Primzahlen, nach deren Zugehörigkeit zu Intervallen mit einer fixen Länge D, zu gruppieren. Man konstruiert zu diesem Zweck eine Überdeckung des Bereichs

$$\{\overline{p} \in \mathbb{P} | B1 < \overline{p} < B2\} \subset \bigcup_{i \geq 1}^{v} [a_i, a_{i+1}] \text{ mit}$$

$$a_1 = B1 - 1,\ a_{i+1} = D + a_i,\ v := min\{j \in \mathbb{N} : a_j \geq B_2\}.$$

Nun arbeitet man diese Intervalle schrittweise ab, indem man für alle Primzahlen eines Intervalls die Bedingung (2.2) formuliert und deren Erfülltheit testet.

Im Folgenden stelle ich die konkrete Implementierung der `ECMInversionless` vor. Da die gesamte Methode `ECMInversionless->ExecuteStage2(...)` sehr umfangreich ist, habe ich mich an dieser Stelle entschlossen, nur die aussagekräftigsten Codefragmente zu präsentieren.

Das erste Fragment zeigt den initialen Schritt des Verfahrens. Wie bereits oben erklärt wurde, werden die Listen L_{Δ_l} und L_{prod} erzeugt. Dazu gibt es die Arrays `storeX[]` und `storeZ[]`, deren Einträge die X- und Z-Koordinaten der Punkte aus L_{Δ_l} enthalten. In dem Array `storeXZ[]` werden die Produkte aus L_{prod} abgelegt. Über eine Schleife werden die geforderten Werte berechnet.

```
...
// get D
D = step / 2;
BigInteger[] tmp;
storeX[0] = Q[0];
storeZ[0] = Q[1];
tmp = Doubleh(Q[0],Q[1]);
storeX[1] = tmp[0];
storeZ[1] = tmp[1];
int j, l, i;
for (i = 2, j = 1, l = 0; i < D; i++, j++, l++) {
      // compute all Values [2i]Q and produkt
      tmp = Addh(storeX[j], storeZ[j], Q[0], Q[1],
             storeX[l], storeZ[l]);
      storeX[i] = tmp[0];
      storeZ[i] = tmp[1];
      storeXZ[i] = storeX[i] * storeZ[i] % N;
}
...
```

Das zweite Codebeispiel zeigt die Verarbeitung der Primzahlen in den Intervallen. `r` bezeichnet die linke Grenze des aktuellen Intervalls und die Integer Größe `step` beinhaltet die fixe Intervallgrenze. In `V[]` wird der Montgomerypunkt $[r]Q$ abgelegt. Es wird für die Primzahlen je Intervall $[r, r + step]$ die Bedingung (2.2) formuliert und überprüft. Bei Erfolg wird eine `ECMRevealedFactorException` geworfen. Liefert die Abarbeitung eines Intervalls keinen Faktor, so werden die Variablen für einen Durchlauf des nächsten Intervalls gesetzt. Die Abbruchbedingung für die äußere Schleife ist

durch die Größe B2 gegeben, welche auch in der Literatur als das Stage 2 Limit bezeichnet wird. Ein Erreichen von B2 bedeutet, dass der letzte Durchlauf erfolglos gewesen ist und kein Faktor gefunden werden konnte.

```
...
while (r < B2){
    a = V[0] * V[1] % N;
    while (currPrime <= r + step && currPrime > r + 1) {
         q = (currPrime - r) / 2;
         g = (g * (storeXZ[q - 1] - a + ((V[0] - storeX[q
              - 1]) *
              (V[1] + storeZ[q - 1])))) % N;
         primeIndex++;
         currPrime = pc.Primes[primeIndex];
    }
    if ((g = g.gcd(N)) != 1)
      throw new ECMRevealedFactorException(g, sigma,
         ECMStageType.Stage2, ECMVariants.Inversionless);
    // reset g
    g = 1;
    // Set Points for next Loop
    tmp = V;
    V = Addh(V[0], V[1], storeX[d - 1], storeZ[d - 1], T[0],
        T[1]);
    T = tmp;
    r += step;
}
...
```

Die folgende Tabelle zeigt den Erfolg der Inversionless Continuation in Abhängigkeit der Parameter B1 und B2 (B2 = 100*B1 [vgl.: C & P, S 309]). Getestet wurde mit 40-stelligen Zahlen, die Primteiler mit 10, 15 oder 20 Stellen haben. Es wurden für jede Kombination aus B1 und B2 200 verschiedene Werte für σ verwendet. Die Tabelle zeigt die Anzahl der erfolgreichen σ - Werte.

B1	**B2**	$\#\sigma(log(p) = 10)$	$\#\sigma\ (log(p) = 15)$	$\#\sigma\ (log(p) = 20)$
2000	20000	69	5	0
3000	30000	90	10	0
4000	40000	90	11	2
5000	50000	90	16	2
6000	60000	92	17	2
7000	70000	93	18	3
8000	80000	96	20	3
9000	90000	96	21	3
10000	100000	97	21	3
15000	150000	97	21	3
20000	200000	97	21	3

2.5.2 Die Montgomery Continuation

Peter L. Montgomery veröffentlichte in seiner Dissertation *An FFT Extension of the Elliptic Curve Method of Factorization* eine Reihe von Ideen zu einer effizienten Implementierung der ECM. Ausgangspunkt für Montgomery ist die Standardcontinuation, bei der am Anfang eines Durchlaufs zwei Listen von Punkten $L_1 := \{[n_i]Q | i \in I\}$ und $L_2 := \{[m_j]Q | j \in J\}$ erzeugt werden. Die Mengen I und J sind über $I := \{v \in \mathbb{N} | 0 \leq d1\}$ und $J := \{v \in \mathbb{N} | 0 \leq d2\}$ gegeben. Q steht wie gewohnt für den in dem vorangegangenem erfolglosen ECM Stage 1 Durchlauf berechneten Punkt und $(n_i)_{i \in I}$ beziehungsweise $(m_j)_{j \in J}$ bezeichnen endliche Folgen natürlicher Zahlen. Zu beachten an dieser Stelle ist, dass der Punkt Q in affinen Koordinaten geführt wird. Die X-Koordinaten der Punkte in L_1 werden mit denen in L_2 verglichen. Stimmen diese für ein $P \in L_1$ und $\overline{P} \in L_2$ (Modu-lo p) überein, so hat die Differenz $P_x - \overline{P}_x$ einen Teiler ungleich 1 mit n gemeinsam. Montgomery beschäftigte sich damit, erfolgsversprechende Eigenschaften der Folgen $(n_i)_{i \in I}$ und $(m_j)_{j \in J}$ zu finden. Seine Ergebnisse listet er als *Desired Properties* in [Mont, S 49-50] auf. Anschließend konstruierte er Folgen, die den *Desired Properties* möglichst gerecht wurden. Weitere von Montgomery publizierte Möglichkeiten zur Effizienzsteigerung betreffen die Berechnung der Listen L_1 und L_2. In diesem Zusammenhang wird eine performante Auswertung von Polynomen an Stellen, die Folgenglieder einer arithmetischen Folge sind, eingesetzt. Hierzu geht man nach dem Schema der finiten Differenzen von Polynomen vor. Zum Vergleich der Listen kann ebenfalls Polynomarithmetik herangezogen werden. Montgomery implementiert in diesem Zusammenhang ein auf FFT basierendes Verfahren, was im Einklang mit dem Titel der Dissertation steht.

Zuerst diskutieren wir Eigenschaften der Folgen $(n_i)_{i \in I}$ und $(m_j)_{j \in J}$, die den Erfolg der Montgomery-Continuation begünstigen. Über $(n_i)_{i \in I}$ und $(m_j)_{j \in J}$ werden die Folgen von X-Koordinaten $([n_i]Q)_x$ und $([m_j]Q)_x$ gebildet, wobei der Faktor p von n gewöhnlich dann gefunden wird, wenn

$$\exists (i,j) \in I \times J : q | n_i \pm m_j \tag{2.3}$$

oder

$$\exists (i_1, i_2) \in I \times I, i_1 \neq i_2 : q | m_{i_1} \pm m_{i_2} \tag{2.4}$$

gilt. (2.3) ist äquivalent zu $[n_i \pm m_j]Q = \mathcal{O}$, $Q \in E_{a,b}(\mathbb{F}_p)$ und bedeutet, dass durch Bildung von

$$ggT(([m_j]Q)_x \pm ([n_i]Q)_x, n) \tag{2.5}$$

ein nicht trivialer Faktor von n aufgedeckt werden kann. Verwendet man als Argument für den ggT die Differenz der X-Koordinaten der Punkte, so überprüft man durch Auswertung von (2.5) die beiden Fälle $[n_i + m_j]Q = \mathcal{O}$ und $[n_i + m_j]Q = \mathcal{O}$ in einem Zug. Da sich die Punkte $[m_j]Q$ und $-[m_j]Q$ nur durch das Vorzeichen der Y-Koordinate unterscheiden, muss in beiden Fällen bei der Berechung von m aus Definition 1.1.5 der Wert $(x_2 - x_1)^{-1}$ in $\mathbb{Z}/n\mathbb{Z}$ invertiert werden. Das steigert die Erfolgswahrscheinlichkeit des Verfahrens. Formulieren wir jetzt einige der für uns interessanten erfolgsbegünstigenden Eigenschaften von $(n_i)_{i\in I}$ und $(m_j)_{j\in J}$.

1. **(DP1)** Die meisten kleineren Primzahlen q sollten einige der $m_i \pm n_j$ oder $m_{i_1} \pm m_{i_2}$ teilen. Es reicht auch aus, dass diese q einige der m_i oder n_j teilen, da in dem Fall bei der Kalkulation von $[m_i]Q$ oder $[n_j]Q$ der Faktor p ohnehin aufgedeckt wird.

2. **(DP2)** Die $m_{i_1} \pm m_{i_2}$ und $m_i \pm n_j$ sollen im Allgemeinen zusätzlich zu denen in *(DP1)* geforderten Teilern noch weitere besitzen. Auszuschließen sind in jedem Fall Summen und Differenzen, die 0 ergeben, sowie für einzelne Folgenglieder nicht $n_i = 0$ beziehungsweise $m_j = 0$ gelten soll. Die n_j sollen unterschiedlich sein.

3. **(DP3)** Es wäre von Vorteil, wenn die Folgen $(n_i)_{i\in I}$ und $(m_j)_{j\in J}$ so beschaffen wären, dass die Kalkulationen von $[n_i]Q$ und $[m_j]Q$ möglichst wenig Multiplikationen (Modulo n) benötigen.

Montgomery gibt noch weitere *Desired Properties* in seiner Arbeit an. Diese habe ich, um im Rahmen zu bleiben, nicht berücksichtigt. Beispielsweise werden Bedingungen zur Implementierung der Methode auf einem Cluster mit mehreren Prozessoren gestellt, was für unsere PC - kompatible Software nicht umsetzbar ist.

Um **(DP2)** gerecht zu werden, kann man Polynome $\in \mathbb{Z}/n\mathbb{Z}[x]$ zur Erzeugung von $(n_i)_{i\in I}$ und $(m_j)_{j\in J}$ verwenden. Für die Praxis bedeutet dies: es müssen Folgen $(N_i)_{i\in I}$, $(M_j)_{j\in J}$ gefunden und ein Polynom $P(x) \in \mathbb{Z}/n\mathbb{Z}[x]$ definiert werden, aus denen die Folgen $(n_i)_{i\in I}$ und $(m_j)_{j\in J}$ durch die Identitäten

$$n_i = P(N_i) \quad \textit{und} \quad m_j = P(M_j)$$

gebildet werden. Wählt man als Polynom $P(x) = x^k$ mit einem k für das $2k$ in viele Primfaktoren zerfällt, so kann man folgern, dass

$$(P(x) - P(y))(P(x) + P(y)) = x^{2k} - y^{2k}$$

$d(2k)$ verschiedene irreduzible Faktoren besitzt. $d(2k)$ bezeichnet die Anzahl der Teiler von $2k$ inklusive 1 und $2k$ (vgl.: [Mont, S 50]).

Montgomery gibt als Alternative zu x^k sogenannte Dicksonpolynome an, die ebenfalls eine große Anzahl an irreduziblen Faktoren haben (vgl. [Mont, S 51]). Für unsere Implementierung hat sich das Polynom $P(x) = x^4$ bewährt, was im Folgenden genauer begründet wird.

Unter Berücksichtigung von **(DP1)** kann man für die Folgen $(N_i)_{i\in I}$ und $(M_j)_{j\in J}$ die arithmetischen Bildungsgesetze

$$N_j = 6d_1(j+1) \quad j \in J$$

und

$$M_i = 6i + 1 \quad i \in I$$

verwenden. Im Falle von $ggT(q, 6) = 1$ und $6d_1 < q < 6d_1d_2$ lässt sich q als $q = 6d_1(j+1) \pm (6i+1)$ mit $(i, j) \in I \times J$ darstellen. Das folgt aus der Eigenschaft, dass für alle Primzahlen $q > 3$ prinzipiell

$$q \equiv a \bmod 6 \ \Rightarrow a \in \{1, 5\}$$

gilt. Der Faktor d_1 in den Folgengliedern von $(N_j)_{j\in J}$ und die Schranke d_2 sorgen dafür, dass q in dem oben festgelegten Bereich liegen muss. Bei der Wahl von $k = 4$ folgt sogar

$$q|(M_i + N_j)(M_i - N_j) \Rightarrow q|M_i^4 - N_j^4,$$

was die Berücksichtigung von **(DP1)** bedeutet.

Die Wahl einer arithmetischen Folge für die M_i und N_i bringt einen weiteren Vorteil für die Berechnung der Listen L_1 und L_2. Über das Schema der finiten Differenzen können Polynome an einer Anzahl von Stellen $(x_l)_{l\in\mathbb{N}}$ kostengünstig evaluiert werden. Die x_l müssen dazu eine arithmetische Folge bilden. In unseren Fall haben wir das Polynom $x^k = P(x)$ und suchen die Werte $P(N_i)$ für $i \in I$ und $P(M_j)$ für $j \in J$. An dieser Stelle wird ein Schema zur Bildung und Verwertung von finiten Differenzen erklärt.

Wir zeigen den Aufbau des Schemas anhand des Beispiels $P(x) = x^4$ und der oben definierten Folge $(N_i)_{i\in I}$. Ziel ist es, die Werte $P(N_i)\ \forall i \in I$ zu erhalten. Dazu erstellt man die folgende Tabelle:

0		1		16		81		256		625
	1		15		65		175		369	
		14		50		110		194		
			36		60		84			
				24		24				

In der ersten Zeile stehen die Funktionswerte des Polynoms $P(x) = x^4$ an den Stellen N_i für $0 \leq i \leq 5$. Jede der folgenden Zeilen setzt sich aus den Differenzen aufeinanderfolgender Elemente in der darüberliegenden Zeile zusammen. Für die Differenz wird das linke Element von seinem rechten Nachbar subtrahiert. Aus einem Satz [Knuth, S 431] folgt nun, dass die 5. Zeile konstant ist. Ausgehend von der ersten vollen Diagonale

$$[24,\ 60,\ 110,\ 175,\ 256]$$

kann durch Additionen die nächste volle Diagonale berechnet werden. Der Wert in der ersten Zeile der Diagonale liefert den Funktionswert von P an der Stelle N_6. Auf dieselbe Weise lässt sich aus der Diagonale zu N_6 jene von N_7 ausrechnen. Dadurch erhält man auch $P(N_7)$. Auf diese Weise kann $P(x)$ entlang der arithmetischen Folge N_i ausgewertet werden. Zur Adaption dieses Verfahrens auf die Punktlisten L_1 und L_2 berechnet man wie eben erklärt wurde, die Tabelle bis zur ersten vollen Diagonale. Dann ersetzt man die Zahlen in der Diagonale durch die entsprechenden Vielfachen des Punktes Q. Nun kann man analog zu oben durch elliptische Additionen die nächsten Punktdiagonalen kalkulieren. Damit hat man **(DP3)** ebenfalls berücksichtigt.

Als letzter Baustein zu einer vollwertigen ECM Continuation fehlt noch die Suche nach Paaren aus $L_1 \times L_2$ mit

$$1 < ggT(([n_i]Q)_x - ([m_j]Q)_x, n) < n \quad \vee$$

$$1 < ggT(([m_{i_1}]Q)_x - ([m_{i_2}]Q)_x, n) < n.$$

Allgemeiner formuliert müssen wir 2 Listen mit Elementen aus $\mathbb{Z}/n\mathbb{Z}$ nach Übereinstimmung (Modulo p) durchtesten. Wenn p bekannt wäre, bräuchte

man nur die Elemente in den Listen zu reduzieren (Modulo p), anschließend auf- oder absteigend zu sortieren und zuletzt in den geordneten Listen zu suchen. Die Schwierigkeit in unserer Anwendung ergibt sich aus der Unbekanntheit von p. Eine Lösung wäre das Produkt über alle zu überprüfenden Differenzen (Modulo n) zu bilden und dessen größten gemeinsamen Teiler mit n zu berechnen. Das Bedeutet die Auswertung von

$$D := ggT(\prod_{i=0}^{d_1} \prod_{j=i+1}^{d_2} (([n_i]Q)_x - ([m_j]Q)_x), n) \tag{2.6}$$

am Schluss der Continuation, wobei die beiden Produkte Modulo n aufzufassen sind.

Montgomery gibt eine Alternative, die sich auf Polynomarithmetik stützt, an. Diese setzt allerdings eine Implementierung der Algorithmen *PolyGCD* und *HalfGCD* voraus, die aber im Rahmen dieser Diplomarbeit nicht umgesetzt wurden (vgl.: Mont S 24-32]). Auch Brent stellt in seiner Arbeit *Some Integer Factorization Algorithms using Elliptic Curves* Möglichkeiten vor. Diese Ideen wurden von mir zur Finalisierung der Montgomery Continuation umgesetzt.

Bilden wir aus der Liste L_1 das Polynom

$$G(x) := \prod_{i \in I} (x - a_i) \quad a_i \in L_1, \forall i \in I$$

und evaluieren wir anschließend

$$g := ggT(n, \prod_{j \in J} G(b_j)) \quad b_j \in L_2, \forall j \in J.$$

Gilt $n > g > 1$, so existiert ein Paar $(a_i, b_j) \in L_1 \times L_2$ mit $p | b_j - a_i$ und somit $p | ggT(n, b_j - a_i)$. Der Faktor p von n wird dann extrahiert. Jetzt muss man nur noch eine Überprüfung für den Fall

$$\exists\ a_{i_1},\ a_{i_2} \in L_1 \text{ mit } p | a_{i_1} - a_{i_2}$$

finden. Dazu betrachtet man den Ausdruck $\overline{D}$

$$\overline{D} = \prod_{i=1}^{d_1 - 1} \prod_{j=i+1}^{d_1} (a_i - a_j),\ a_i, a_j \in L_1, \forall i \in I$$

und hält fest, dass für das Polynom $G(x)$

$$\overline{D}^2 = \prod_{i \in I} G'(a_i) \quad a_i \in L_1, \forall i \in I$$

gilt, wobei $G'(x)$ die formale Ableitung von $G(x)$ nach x bezeichnet. Die Gültigkeit dieser Identität ist durch einen Vergleich der Faktoren einzusehen. Denn es gilt die für $a_{i_1} - a_{i_2} | \overline{D}^2$ sofort $(a_{i_1} - a_{i_2}) | G'(a_{i_1})$. Daraus erhält man $\overline{D}^2 | \prod_{i \in I} G'(a_i)$. Betrachtet man umgekehrt ein $G'(a_i)$ mit $a_i \in L_1$, so folgt aus

$$G'(a_i) = \prod_{i \in I, i \neq l} (a_i - a_l) | D^2,$$

was die Gleichheit der beiden Ausdrücke beweist. Die Evaluierung von $ggT(\overline{D}^2, \prod_{i \in I} G'(a_i))$ liefert somit genau dann einen Faktor von n, wenn es in der Liste L_1 mindestens zwei Elemente a_{i_1} und a_{i_2} gibt, für die $p | a_{i_1} - a_{i_2}$ gilt.

Im Folgenden wird auf die Implementierung der beiden Klassen `ECMMontgomery` und `ECMMontgomeryPoly` eingegangen. Die Klassen unterscheiden sich nur durch die Evaluierung der Listen L_1 und L_2. Aus diesem Grund wird in `ECMMontgomery` die Methode Evaluate als `virtual` deklariert und in `ECMMontgomeryPoly`, die von `ECMMontgomery` erbt, mittels `override` überschrieben. Die anderen behandelten Codefragmente betreffen daher beide Implementierungen.

Die Berechnung der ersten vollen Diagonale in dem Differenzenschema für x^k und der Folgen $(N_i)_{i \in I}$ und $(M_j)_{j \in J}$ erfolgt durch

```
1  private void SetStartVectors(out BigInteger[] prog1,
2                               out BigInteger[] prog2) {
3     // M(i) = 6*i + 1 for 0 <= i <= D1
4     prog1 = new BigInteger[k + 1];
5     BigInteger tmp = 1;
6     prog1[0] = 0;
7     for (int i = 1; i < prog1.Length; i++) {
8         tmp += 6; prog1[i] = tmp ^ k;
9     }
10    // N(j) = 6*d1*(j + 1) for 0 <= j <= D2
11    prog2 = new BigInteger[k + 1];
12    BigInteger incr = 6 * d1;
13    tmp = 1;
14    prog2[0] = incr;
15    for (int j = 1; j < prog2.Length; j++) {
16         prog2[j] = tmp ^ k; tmp += incr;
17    }
```

```
18    // get initial upward diagonal for fast evaluation on M(i)
19    for (int i = 0; i < 4; i++) {
20         for (int j = 0; j < 4 - i; j++) {
21              prog1[j] = prog1[j + 1] - prog1[j];
22              prog2[j] = prog2[j + 1] - prog2[j];
23         }
24    }
25 }
```

Es folgt die Kalkulation der Punktlisten L_1 und L_2. Hier wird um Redundanz im Text zu vermeiden, nur die Erzeugung einer Liste dargestellt

```
1  for (int i = 0; i < d1; i++) {
2       for (int j = 1; j <= k; j++) {
3            currPoints1[j] = (AffineAdd(currPoints1[j - 1][0],
4                                        currPoints1[j - 1][1],
5                                        currPoints1[j][0],
6                                        currPoints1[j][1]));
7       }
8       randomPointsX1[i] = currPoints1[k][0];
9       randomPointsY1[i] = currPoints1[k][1];
10 }
```

Die Evaluierung der Listen L_1 und L_2 passiert in `ECMMontgomery` durch

```
1  protected virtual void Evaluate(BigInteger[] randomPointsX1,
2                                  BigInteger[] randomPointsX2)
                                       {
3     BigInteger D = new BigInteger(1);
4     // build product for evaluation
5     for (int i = 0; i < d1; i++)
6          for (int j = 0; j < d2; j++)
7              D = (D * (randomPointsX1[j] - randomPointsX2[i]))
                    % N;
8     // Check GCD
9     if ((D = D.gcd(N)) != 1) {
10         throw new ECMRevealedFactorException(D, sigma,
11             ECMStageType.Stage2, ECMVariants.Montgomery);
12    }
13 }
```

und in `ECMMontgomeryPoly` durch

```
1  ...
2  // Get polynomials F and G with the
3  // roots randomPointsX1, RandomPointsX2
4  BigInteger[] F = PolynomArrayArithmetic.GetOutOfRoots(
5             randomPointsX1, N);
6  // check the two Lists of agreement mod N
7  BigInteger g = 1;
```

```
8  foreach (BigInteger currBi in randomPointsX2)
9     g = (g * PolynomArrayArithmetic.Evaluate(F, currBi, N))
          % N;
10 factor = g.gcd(N);
11 if (factor != BigInteger.One)
12     throw new ECMRevealedFactorException(factor, sigma,
13                                          ECMStageType.Stage2
                                                );
14 // check in randomPoints1 list for agreement mod N
15 // differentiate F
16 F = PolynomArrayArithmetic.Differentiate(F, N);
17 g = 1;
18 foreach (BigInteger currBi in randomPointsX1)
19     g = (g * PolynomArrayArithmetic.Evaluate(F, currBi, N))
           % N;
20 factor = g.gcd(N);
21 if (factor != BigInteger.One)
22     throw new ECMRevealedFactorException(factor, sigma,
23                                          ECMStageType.Stage2
                                                );
24 ...
```

Obwohl die Vorschläge von P.L Montgomery für IBM Großrechner gedacht sind, kann man mit der Implementierung in C# durchaus erfolgreich sein.

2.5.3 Die Brent Continuation

Bei der klassischen Continuation von R. Brent wird eine Liste L_1 von Punkten erstellt. Wesentlicher Unterschied zu der Listenerzeugung bei Montgomery ist, dass die Wahl der Punkte aus L_1 nach dem Zufallsprinzip erfolgt. Genauer gesagt, muss eine Pseudozufallsfolge $(\overline{n_i})_{i\in I}$ generiert werden. Danach wird die Liste mit denselben Methoden wie bei Montgomery nach einem Paar aus $L_1 \times L_1 : [a_{i_1} - a_{i_2}]Q = \mathcal{O} \in E_{a,b}(\mathbb{F}_p)$ durchsucht. Erfolgsversprechend wirkt diese Methode aufgrund des Geburtstagsparadoxon, wonach die Übereinstimmung des Geburtsdatums zweier Personen aus einer kleinen Gruppe eine überraschend hohe Wahrscheinlichkeit hat. Denselben Effekt erhofft man sich bei der zufällig generierten Folge von Punkten aus L_1, wenn man nach der Übereinstimmung $([a_{i_1}]Q_{i_1})_x \equiv ([a_{i_2}]Q_{i_2})_x$ mod p sucht.

Die Pseudozufälligkeit der Folge L_1 läßt sich auf verschiedene Arten realisieren. Wichtig ist es hier zwischen Performanceeinbußen aufgrund von aufwendigen Zufallsexperimenten und der Qualität der Pseudozufallsfolgen abzuwägen. Brent gibt in seiner Arbeit beispielsweise die Folge

$$Q_1 = Q$$

$$Q_{j+1} = \begin{cases} [2]Q_j & \text{mit Wahrscheinlichkeit } 0.5 \\ [3]Q_j & \text{mit Wahrscheinlichkeit } 0.5 \end{cases}$$

an und bemerkt im Anschluß die Ineffizienz derselben. Die bis jetzt angeführten Überlegungen werden in der klassischen Version der Brent Continuation integriert, die sich so in der Praxis gegenüber der Montgomery Continuation nicht durchsetzen konnte.

Weiter unten in [Brent, S 10] gibt Brent eine seiner Meinung nach bessere Strategie für die Erzeugung der Punktlisten an. In einem initialen Schritt werden zwei pseudozufällige arithmetische Folgen erzeugt. Das bedeutet die randomisierte Wahl von a, b zu $a + ib$ für $0 \leq i \leq s$. Um die Analogie zu Montgomeries Methodik zu dokumentieren, bezeichnen wir diese Folgen mit $(\overline{n_i})_{i \in I}$ und $(\overline{m_i})_{i \in I}$. Anschließend kann man so vorgehen wie im finalen Schritt der `ECMMontgomery`, beziehungsweise der `ECMMontgomeryPoly` und die beiden Listen nach Übereinstimmung durchkämmen. Der ausführende Code wurde bereits weiter oben angegeben, weshalb an dieser Stelle nur ein Codebeispiel zur Implementierung einer zufällig generierten arithmetischen Folge angeführt wird.

```
1  private BigInteger[] GetStartVector() {
2      BigInteger[] tableStart = new BigInteger[K + 1];
3
4      //  get random arithmetic sequence
5      Random rand = new Random(DateTime.Now.Millisecond);
6      int seq = rand.Next(int.MaxValue / K);
7
8      // set first line of difference table
9      for (int i = 0; i <= K; i++)
10           tableStart[i] = Power(i*seq, K);
11
12     // determine the startvector for the arithmetic
           progression
13     // and store it in tableStart
14     for (int i = 0; i < K; i++)
15        for (int j = 0; j < K - i; j++)
16           tableStart[j] = tableStart[j + 1] - tableStart[j];
17
18     return tableStart;
19 }
```

Kapitel 3

Die Kalkulation von $\#E_{a,b}(\mathbb{F}_p)$

In diesem Kapitel werden Verfahren zur Bestimmung der Punktezahl einer elliptischen Kurve besprochen. Begonnen wird mit einer naiven Methode, welche für jedes Element $x \in \mathbb{F}_p$ seine Zugehörigkeit zu der Kurve überprüft. Anschließend stelle ich den Algorithmus von Shanks und Mestre vor. Dieser ist schon wesentlich performanter als die zuvor behandelte Methode und kann schon für deutlich höhrere p eingesetzt werden. Danach folgt ein theoretischer Teil, welcher die Grundlagen für den Schoof-Algorithmus zusammenfasst. Mit diesem Wissen wird dann eine konkrete Implementierung des Verfahrens entwickelt. Ein Hybridverfahren, das eine Kombination aus den Ideen von Schoof, Shanks und Mestre ist, bildelt den Abschluss der hier diskutierten Möglichkeiten zum Punktezählen.

Die Notwendigkeit, die Anzahl der Punkte von $E_{a,b}(\mathbb{F}_p)$ zu kennen, ergibt sich aus den Anforderungen kryptographischer Anwendungen elliptischer Kurven. In diesem Zusammenhang ist es wichtig, die genaue Punktezahl zu kennen, um die Sicherheit bei der Verwendung einer Kurve gewährleisten zu können. Diese Thematik wird in Kapitel 5 eingehend behandelt. Quasi als Anhang zu diesem Kapitel, erkläre ich noch ein effektives Verfahren zur Erzeugung von Kurven mit bekannter Ordnung.

3.1 Naives Punktezählen

Für jeden Punkt $(\overline{x}, \overline{y}) \neq \mathcal{O}$ einer elliptischen Kurve $E_{a,b}(\mathbb{F}_p)$ gilt

$$\left(\frac{\overline{x}^3 + a\overline{x} + b}{p}\right) = 1, \tag{3.1}$$

da die X-Koordinate ein Quadrat in $\mathbb{F}_p$ sein muss. Über das Legendresymbol kann also für beliebige Paare $(x, y) \in \mathbb{A}^2(\mathbb{F}_p)$ die Zugehörigkeit zu einer elliptischen Kurve festgestellt werden. Aufbauend auf dieser Erkenntnis kann man die Identität

$$\#E_{a,b}(\mathbb{F}_p) = p + 1 + \sum_{x \in \mathbb{F}_p} \left(\frac{x^3 + ax + b}{p}\right) \tag{3.2}$$

herleiten. Für jedes $\overline{x} \in \mathbb{F}_p$ liefert die Evaluierung des Legendresymbols aus (3.1) einen Wert aus $\{1, -1, 0\}$. Wenn 1 auftritt so gilt, dass $\overline{x}$ ein Quadrat in $\mathbb{F}_p$ ist. Man erhält dann zwei mögliche Werte für $\overline{y}$. Liefert die Auswertung von (3.1) -1, so gibt es keinen Punkt auf der elliptischen Kurve mit $\overline{x}$ als X-Koordinate. In dem seltenen Fall, wo das Legendresymbol aus (3.1) für ein bestimmtes $\overline{x} \in \mathbb{F}_p$ gleich 0 ist, hat man mit $\overline{x}$ eine Nullstelle von $x^3 + ax + b$. In diesem Fall erhält man den Punkt $(\overline{x}, 0) \in E_{a,b}(\mathbb{F}_p)$. Zusammenfassend kann man sagen, dass ein $x \in \mathbb{F}_p$ entweder 2, 1 oder gar keinen Punkt liefern kann. Addiert man nun zu $p = char(\mathbb{F}_p)$ die Werte $(\frac{x^3+ax+b}{p})$ für alle $x \in \mathbb{F}_p$, so bekommt man $\#(E_{a,b}(\mathbb{F}_p)\backslash\{\mathcal{O}\})$. Damit ist die Identität (3.2) erklärt.

Die Klasse `EllipticCurve` definiert die Instanzmethode `EllipticCurve->CountingPoints()`, die nach dem soeben erörterten Verfahren die Anzahl der Punkte ermittelt. Die Methode wird auch über den Getter des `EllipticCurve` Members `Order` aufgerufen. Hier ist die C#-Implementierung dieser Methode angegeben.

```
1  // Counts the point of the elliptic curve,
2  // by evaluating the Jacobi symbol
3  public BigInteger CountingPoints() {
4     BigInteger tmpOrder = N + 1;
5     // evaluate Jacobi symbol
6     for (BigInteger currElement=0;currElement<N;currElement
          +=1)
7        tmpOrder += Eval(currElement).Jacobi(N);
8
9     order = tmpOrder;
10    return tmpOrder;
11 }
```

Für $p > 10000$ ist diese Implementierung quasi nicht mehr einsetzbar. Um $\#E_{a,b}(\mathbb{F}_p)$ für $p > 10000$ bestimmen zu können muss man die Methoden von Schoof beziehungsweise Shanks-Mestre heranziehen.

3.2 Der Algorithmus von Shanks und Mestre

Bei der Shanks-Mestre Methode wird versucht, über die Ordnung einzelner Punkte aus $E_{a,b}(\mathbb{F}_p)$, die Ordnung der gesamten Gruppe zu bestimmen. In diesem Zusammenhang suchen wir Vielfache eines Punktes $P \in E_{a,b}(\mathbb{F}_p)$, sodass

$$(3.3) \quad [n]P = \mathcal{O} \text{ für } n \in (p+1-\sqrt{2}, p+1+\sqrt{2})$$

gilt. Nach dem Satz von Hasse erhält man für jedes P mindestens ein Ergebnis. Findet man tatsächlich nur ein $\overline{n}$, das (3.3) genügt, so handelt es sich dabei um die Ordnung der Kurve selbst. Dieser Fall tritt dann ein, wenn $ord_{E_{a,b}(\mathbb{F}_p)}(P)$ ein eindeutiges Vielfaches im Hasseintervall hat, also die Bedingung

$$(3.4) \quad \exists!\ v \in \mathbb{N} \text{ mit } v * Ord_{E_{a,b}(\mathbb{F}_p)}(P) \in (p+1-\sqrt{2}, p+1+\sqrt{2})$$

erfüllt ist. Das ist wiederum dann der Fall, wenn die Ordnung von P größer $4\sqrt{p}$ (d.h größer als die Länge des Hasseintervalls) ist. Shanks und Mestre geben ein Verfahren an, in dem bestimmte Vielfache eines Punktes berechnet werden, um eine Situation wie in (3.3) zu provozieren.

Angenommen für ein $P \in E_{a,b}(\mathbb{F}_p)$ gilt die Bedingung

$$(3.5) \quad \overline{P} = [v]P \ \vee\ \overline{P} = -[v]P,\ \overline{P} := [p+1+u]P$$

dann stimmen die X-Koordinaten von $[v]P$ und $\overline{P}$ überein und wir erhalten

$$(3.6) \quad [p+1+u+v]P = \mathcal{O} \ \vee\ [p+1+u-v]P = \mathcal{O}$$

Einer der beiden Faktoren muss dann ein Vielfaches der Ordnung von P sein und ist vielleicht sogar die Ordnung der Kurve selbst.

Um für ein P, das die Bedingung (3.4) erfüllt, auch sicher den eindeutigen anullierenden Faktor zu finden, muss man alle natürlichen Zahlen aus dem Hasseintervall durchprobieren. Dazu bietet sich an, zwei Listen L_1 und L_2 mit Vielfachen des Punktes P zu führen. Eine Möglichkeit bestünde darin, für ein $P \in E_{a,b}(\mathbb{F}_p)$ und $W = \lceil p^{\frac{1}{4}}\sqrt{2} \rceil$

$$L_1 := \{[p+1+\beta]P : \beta \in I\} \text{ mit } I = \{k \in \mathbb{N} : 0 \leq k \leq W-1\}$$

$$L_2 := \{[\gamma W]P : \gamma \in J\} \text{ mit } J = \{k \in \mathbb{N} : 0 \leq k \leq W\}$$

zu wählen. Für jedes $k \in \mathbb{Z}$ mit $|k| < 2\sqrt{p}$ existiert ein Paar $(\overline{\beta}, \overline{\gamma})$ aus $I \times J$ mit $k = \overline{\beta} + \overline{\gamma}W$. Hat man diese Listen einmal gebildet, dann kann man

$$(3.7) \quad [p + 1 + \beta \pm \gamma W]P = \mathcal{O} \text{ für } (\beta, \gamma) \in I \times J$$

überprüfen. Findet man genau ein Paar $(\overline{\beta}, \overline{\gamma}) \in I \times J$, das (3.7) genügt, so ist mit $p + 1 + \overline{\beta} + \overline{\gamma}W$, beziehungsweise $p + 1 + \overline{\beta} - \overline{\gamma}W$ die Ordnung von $E_{a,b}(\mathbb{F}_p)$ gegeben. Gibt es allerdings mehrere Paare aus $I \times J$, die (3.7) genügen, so ist es nicht möglich, auf $\#E_{a,b}(\mathbb{F}_p)$ zu schließen. Der folgende Satz von Mestre garantiert für gewisse $p \in \mathbb{P}$ die Existenz eines Punktes $P \in E_{a,b}(\mathbb{F}_p)$, für den das obige Verfahren die Ordnung von $E_{a,b}(\mathbb{F}_p)$ liefern kann.

Satz 3.1(Mestre) Für eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$ und deren quadratischen Twist $E_{g^2a,g^3b}(\mathbb{F}_p)$ gilt

$$\#E_{a,b}(\mathbb{F}_p) + \#E_{g^2a,g^3b}(\mathbb{F}_p) = 2p + 2 \text{ mit } (\frac{g}{p}) = -1$$

Auf mindestens einer der beiden Kurven $E_{a,b}(\mathbb{F}_p)$ oder $E_{g^2a,g^3b}(\mathbb{F}_p)$ liegt ein Punkt dessen Ordnung größer als $4\sqrt{p}$ ist, wenn $p > 457$ gilt. Weiters gilt für $p > 229$, dass mindestens eine der besagten Kurven einen Punkt P mit

$$\exists!\ v \in \mathbb{N} \text{ mit } [v]P = \mathcal{O}, v \in (p + 1 - \sqrt{2}, p + 1 + \sqrt{2})$$

besitzt. v ist dann die Ordnung der aktuellen Kurve.

Das Resultat $\#E_{a,b}(\mathbb{F}_p) + \#E_{g^2a,g^3b}(\mathbb{F}_p) = 2p + 2$ ist leicht einzusehen, wenn man bedenkt, dass jedes $x \in \mathbb{F}_p$ mit $x^3 + ax + b \neq 0$ entweder für $E_{a,b}(\mathbb{F}_p)$ oder für $E_{g^2a,g^3b}(\mathbb{F}_p)$ zwei Punkte liefert. In dem Fall wo x eine Nullstelle der Kurvengleichung ist gibt es auf $E_{a,b}(\mathbb{F}_p)$ und $E_{g^2a,g^3b}(\mathbb{F}_p)$ jeweils einen Punkt mit X-Koordinate x.

Nun kann man aufbauend auf diesen theoretischen Grundlagen eine Methode zur Bestimmung von $\#E_{a,b}(\mathbb{F}_p)$ entwerfen. Aus dem Satz von Mestre folgt, dass über die Kenntnis von $\#E_{g^2a,g^3b}(\mathbb{F}_p)$ auf $\#E_{a,b}(\mathbb{F}_p)$ geschlossen werden kann. Für jeden Durchlauf des Shanks-Mestre Verfahrens wird ein Punkt $P = (\overline{x}, \overline{y}) \in E_{a,b}(\mathbb{F}_p) \cup E_{g^2a,g^3b}(\mathbb{F}_p)$ benötigt. Wählt man dazu ein $x \in [0, p-1]$ zufällig, so gilt $(\frac{x^3+ax+b}{p}) = \pm 1$. Falls $x^3 + ax + b$ ein quadratischer Rest Modulo p ist, liegt P auf $E_{a,b}(\mathbb{F}_p)$. Handelt es sich bei $x^3 + ax + b$ um einen quadratischen nicht Rest Modulo p, dann gilt $P \in E_{g^2a,g^3b}(\mathbb{F}_p)$. Der

Algorithmus muss dann mit der entsprechenden Kurve abgearbeitet werden. Das angeführte Codesample zeigt diesen initialen Teil der in `EllipticCurves` realisierten Implementierung der Methode.

```
1     // loop until x is found
2     do {
3        // choose random x
4        x = BigInteger.GetRandom(Curve.N - 1);
5        // evaluate random x and compute Jacobi symbol
6     } while ((sigma = Curve.Eval(x).Jacobi(Curve.N)) == 0);
7
8     // henceforth we will use the curve
9     // or its quadratic twist
10    // in dependency of sigma
11    if (sigma == 1) {
12       // use the original curve
13       currCurve = Curve;
14    } else {
15        // case sigma == -1
16        currCurve = Curve.GetTwistCurve(g);
17        // transform x-coordinate
18        x = (x * g) % Curve.N;
19    }
20
21    // gets a point with the input x coordinate
22    P = currCurve.GetPointOfXCoordinate(x);
```

Hat man einen Punkt P gefunden und seine Kurve als Parameter gesetzt, dann können die oben bereits angesprochenen Listen L_1 und L_2 erzeugt und durchsucht werden. In der Methode `ShanksMestre->GetPointList(...)` werden die Listen aufgebaut. In dem affinen Punkt `tmp` werden die Zwischenresultate abgelegt, um sie über den Member `AffinePoint.Tag` vom Typ `BigInteger` markieren zu können. Diese Kennzeichnung wird dann in weiterer Folge beim Vergleich der Listen auf Übereinstimmung benötigt.

```
1
2 protected virtual void GetPointLists(AffinePoint P,
3                                      ref BigInteger w,
4                       out List<AffinePoint> pointsA,
5                       out List<AffinePoint> pointsB) {
6
7   pointsA = new List<AffinePoint>();
8   pointsB = new List<AffinePoint>();
9
10   // form two lists A and B of points
11  // get list A
12  AffinePoint tmpP;
13  int i = 0;
14  for (BigInteger tmp = 0; tmp < w; tmp += 1) {
15       tmpP = CurrCurve.Mult(P, Curve.N + 1 + tmp);
16       tmpP.Tag = i++;
17       pointsA.Add(tmpP);
18   }
19
20   // get List B
21   i = 0;
22   for (BigInteger tmp = 0; tmp <= w; tmp += 1) {
23       tmpP = CurrCurve.Mult(P, tmp * w);
24       tmpP.Tag = i++;
25       pointsB.Add(tmpP);
26   }
27 }
```

Als nächstens muss der Durchschnitt $L_1 \cap L_2$ gebildet werden. Das wird über die Methode `ShanksMestre->GetIntersection(...)` realisiert. Die affinen Punkte in den Listen werden nach ihrer X-Koordinate sortiert. Dazu implementiert die Klasse `AffinePoint` das Interface `ICompareable` und stellt die Methode `AffinePoint->CompareTo(...)` zur Verfügung. Aufgrund der Sortierung, die $O(ln(N)N)$ Operationen benötigt, können die Listen in $O(N)$ Schritten abgearbeitet werden. N steht hier für die Länge der Listen. Die Methode retourniert nur dann einen Wert ungleich `null` wenn $\#L_1 \cap L_2 = 1$ gilt.

```
1
2 private AffinePoint GetIntersection(List<AffinePoint> pointsA
    ,
3         List<AffinePoint> pointsB,
4         out BigInteger indexI, out BigInteger indexJ) {
5   // the intersection
6   AffinePoint intersection = null;
7   // default values of indexI, indexJ
8   indexI = indexJ = -1;
9   // cancel values in ListA and ListB down to
```

```
10    // sort List
11    pointsA.Sort();
12    pointsB.Sort();
13    // compute intersection
14    int i = 0, j = 0;
15    while (i < pointsA.Count && j < pointsB.Count) {
16       if (pointsB[j] > pointsA[i])
17           i++;
18       else if (pointsB[j] < pointsA[i])
19           j++;
20       else {
21           if (intersection == null &&
22               pointsB[j].ZCoordinate == pointsA[i].ZCoordinate
                     ) {
23               indexI = pointsA[i].Tag;
24               indexJ = pointsB[j].Tag;
25               intersection = pointsA[i];
26               } else {
27                   return null;
28               }
29               i++;
30       }
31 }
```

Bekommt man von der Methode `GetIntersection(...)` eine Instanzierung von `AffinePoint` zurück, die ungleich `null` ist, dann hat man mit dem verwendeten Punkt P eine Möglichkeit zur Bestimmung von $\#E_{a,b}(\mathbb{F}_p)$ gefunden. Das Paar $(\overline{\beta}, \overline{\gamma})$ mit

$$[p+1+\overline{\beta}+\overline{\gamma}W]P = \mathcal{O} \ \vee \ [p+1+\overline{\beta}-\overline{\gamma}W]P = \mathcal{O}$$

wird über die `out` Parameter `indexI` und `indexJ` zurückgegeben. Das Vorzeichen in $\overline{\beta} \pm \overline{\gamma}W$ wird durch Probieren ermittelt. Tritt der seltene Fall ein, dass

$$[p+1+\overline{\beta}+\overline{\gamma}W]P = [p+1+\overline{\beta}-\overline{\gamma}W]P = \mathcal{O}$$

gilt, so kann das Ergebnis nicht weiter verwertet werden. Andernfalls gilt dann

$$\#E_{a,b}(\mathbb{F}_p) = p+1+\sigma t \text{ mit } t = \beta \pm \gamma W, \ \sigma = \left(\frac{x^3+ax+b}{p}\right) \in \{1,-1\}$$

Damit hat man sein Ziel erreicht und die Ordnung der elliptischen Kurve bestimmt.

Die Komplexität dieses Verfahrens setzt sich aus der Komplexität der Durchschnittsbildung $L_1 \cap L_2$ und der Listengeneration zusammen. Für die Durch-

schnittsbildung benötigt man $O(p^{\frac{1}{4}}ln(p))$ und für die Berechnung der Listen $O(p^{\frac{1}{4}})$ Operationen. Das ergibt insgesamt einen Aufwand von $O(p^{\frac{1}{4}+\epsilon})$.

3.3 Der Schoof Algorithmus

Die in dem Artikel *Elliptic curves over finite fields and the computation of square roots mod p* von R. Schoof vorgestellte Methode liefert einen deterministischen Algorithmus zur Bestimmung der Punktezahl einer elliptischen Kurve mit polynomialer Laufzeit $O(ln(p)^k)$ (vgl. [C & P, S 317] und [Mirbach, S 47]). Wie man aus dem Satz von Hasse entnehmen kann, gilt

$$\#E_{a,b}(\mathbb{F}_p) = p + 1 + t \text{ mit } t \in (-2\sqrt{p}, 2\sqrt{p}).$$

Schoof hat eine Möglichkeit gefunden, für beliebige $q \in \mathbb{P}$ den Wert $\tau_q \equiv t \bmod q$ zu ermitteln. Kennt man genügend Gleichungen der Form

$$G_i := (\tau_{q_i} \equiv t \bmod q_i) \text{ mit } i \in I, q_i \in \mathbb{P}$$

so kann das Gleichungssystem linearer Kongruenzen durch Anwendung des chinesischen Restsatzes nach t aufgelöst werden. Genügend G_i bedeutet konkret, dass die q_i die Bedingung

$$\prod_{i \in I} q_i > 4\sqrt{p} \text{ mit } I = \{i \in \mathbb{N} : 0 \leq i \leq l_{max}\} \tag{3.9}$$

efüllen. l_{max} ist die kleinste natürliche Zahl für die (3.9) erfüllt ist. Man erhält somit die Ordnung von $E_{a,b}(\mathbb{F}_p)$.

3.3.1 Der Fall t mod 2

Die Berechnung von t mod 2 erfolgt auf eine andere Weise als bei den übrigen Primzahlen. Als Ergebnis können dabei die Werte 0 und 1 auftreten. Wenn $t \bmod 2 = 1$ gilt, so bedeutet dies, dass die Ordnung von $E_{a,b}(\mathbb{F}_p)$ gerade ist. Andernfalls ist $\#E_{a,b}(\mathbb{F}_p)$ nicht durch 2 teilbar. Allgemein gilt, dass die Ordnung einer Gruppe genau dann gerade ist, wenn sie ein Element mit Ordnung 2 besitzt. Die Punkte P aus $E_{a,b}(\mathbb{F}_p)$ mit $Ord_{E_{a,b}(\mathbb{F}_p)}(P) = 2$ sind immer von der Gestalt $(x, 0)$ $x \in \mathbb{F}_p$, da für sie immer $-P = P$ gelten muss. Der folgende Satz liefert eine weitere Charakterisierung dieser Punkte.

Satz 3.2 (Punkte der Ordnung 2) Für die Polynome $f(x) := x^3 + ax + b$, $f_p = x^p - x$ aus $\mathbb{F}_p[x]$ und die Ordnung von $E_{a,b}(\mathbb{F}_p)$ gilt

$$deg(ggT(f, f_p)) > 0 \Leftrightarrow \#E_{a,b}(\mathbb{F}_p) \equiv 0 \bmod 2.$$

Beweis Die Bedingung $deg(ggT(f, f_p)) > 0$ ist genau dann erfüllt, wenn das Polynom $f(x)$ Nullstellen in $\mathbb{F}_p$ hat. Jede dieser Nullstellen entspricht einem Punkt, dessen Y-Koordinate gleich 0 ist. Wie oben gesagt, handelt es sich dabei genau um die Punkte mit Ordnung 2. Aus diesem Grund reicht es aus, den ggT von f mit dem charakteristischen Polynom $f_p(x)$ von $\mathbb{F}_p$ zu berechnen.

Zur Umsetzung dieses Spezialfalls braucht man keine weiteren theoretischen Erkenntnisse mehr. Problematisch ist die Berechnung von $ggT(f, f_p)$, da p im Allgemeinen groß sein kann. Abhilfe schaft hier, vor dem ggT das Polynom

$$x^p \bmod f = x^p \bmod x^3 + ax + b$$

zu berechnen und dann

$$ggT(f, f_p) = ggT(x^3 + ax + b, (x^p \bmod x^3 + ax + b) - x) \tag{3.10}$$

auszuwerten. Aufgrund der niedrigen Graddifferenz der Argumente (ist immer kleiner als 3) ist dann mit keinem Performanceproblemen zu rechnen.

Die Klasse `Polynom` aus der Bibliothek `BigInteger` implementiert ein Polynom in einer Variablen. Als grundlegende Datenstruktur werden hier generische Listen aus `System.Collections.Generic` verwendet. Auf diese Weise wird nur so viel Speicher allokiert, wie die Koeffizienten der Polynome tatsächlich benötigen. Die statische Klasse `PolynomKernel` stellt sämtliche Algorithmen für die Arithmetik von Polynomen mit einer oder zwei Veränderlichen zur Verfügung. Für den aktuellen Fall wird die statische Methode `PolynomKernel->GCD(...)` zur Berechnung von (3.10) verwendet. Objekte von `SchoofBase` besitzen den `Polynom` Member `CharPoly`, der das charakteristische Polynom von $\mathbb{F}_p$ darstellt und über `CurveEquation` vom Typ `Polynom` kann bei einer Instanzierung von `EllipticCurve` auf die Gleichung der aktuellen Kurve zugegriffen werden. Da man auf diese Weise die beiden benötigten Polynome bequem ansprechen kann und deren Graddifferenz niedrig ausfällt, ist der Einsatz von `PolynomKernel->GCD(...)` gerechtfertigt. Die Potenzierung x^p mod x^3+ax+b erfolgt über eine statische Methode von `PolynomArrayArithmetik`. Hier werden Polynome als `BigInteger` Arrays dargestellt. Das bringt beim Zugriff auf die Elemente einen maßgeblichen Performancevorteil gegenüber der in Listen strukturierten Instanzen von `Polynom`. Allerdings wird auf die spezielle Gestalt mancher Polynome bei der Speicherverwaltung keine Rücksicht genommen. Da aber durch permanente Reduktionen (Modulo x^3+ax+b) während der Berechnung von x^p der Grad der Polynome niedrig gehalten wird, hält sich der verschwendete Spei-

cher in Grenzen. Das folgende Codebeispiel zeigt die Umsetzung des soeben Erörterten über die Polynomarithmetik in der `BigInteger` Bibliothek. Die erste Methode zeigt die Berechung von $x^p - x$.

```
/// Creates the characteristic polynomial
/// of the underlying field
private void SetCharPoly() {
   // the curve equation is needed for mod reduction of X^p
   charPoly = Polynom.FromArray(
                  PolynomArrayArithmetic.PolyPowerMod(
                    new BigInteger[] { 0, 1 },
                    Curve.N, Curve.CurveEquation.ToArray(),
        Curve.N));
   // = x^p - x
   charPoly.InsertMonom(new Monom(1, Curve.N - 1));
}
```

Dieser Ausschnitt aus der Implementierung des Schoof-Algorithmus präsentiert die Abhandlung des Falls t mod 2.

```
// Work off special case l ==2
if (l == 2) {
    // polynomial mod reduction of CharPoly by
    // calculating x^p - 1 mod CurveEqu
    Polynom g = PolynomKernel.GCD(CharPoly,
    Curve.CurveEquation, Curve.N);

    if (g.IsOnePoly())
       return 1;
    else
       return 0;
}
```

3.3.2 Der Fall t mod q für $q > 2$

Für die Berechnung von $\tau_q \equiv t \bmod q$ für $q > 2$ wird der Frobenius - Endomorphismus oder kurz **Frobenius**

$$\phi : E_{a,b}(\overline{\mathbb{F}}_p) \to E_{a,b}(\overline{\mathbb{F}}_p),\ (x,y) \mapsto \phi((x,y))$$

$$\phi((x,y)) = \begin{cases} (x^p, y^p) & \text{für } (x,y) \neq \mathcal{O} \\ \mathcal{O} & \text{für } (x,y) = \mathcal{O} \end{cases}$$

benötigt. Die Potenzbildung x^p in $\overline{\mathbb{F}}_p$ ist ein Körperautomorphismus, der die Elemente aus $\mathbb{F}_p$ fest lässt. Daraus erkennt man vorerst, dass

$$\phi(P) \in E_{a,b}(\overline{\mathbb{F}}_p)\ \forall\ P \in E_{a,b}(\overline{\mathbb{F}}_p)$$

und

$$\phi(P) = P \text{ für } P \in E_{a,b}(\mathbb{F}_p)$$

gilt. Bedenkt man nun weiter, dass bei der Arithmetik der elliptischen Kurven nur rationale Ausdrücke in den X- und Y-Koordinaten der betroffenen Punkte gebildet werden, so erkennt man die Erfülltheit der Homomorphiebedingung von ϕ.

Ein weiterer wichtiger Begriff in diesem Zusammenhang ist die l-Torsionsgruppe einer elliptischen Kurve.

Definition 3.3.1(l-Torsionsgruppe) Gegeben sei eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$. Die Menge

$$E_{a,b}(\mathbb{F}_p)[l] := \{P \in E_{a,b}(\overline{\mathbb{F}}_p) : [l]P = \mathcal{O}\}$$

für ein gegebenes $l \in \mathbb{N}$ vereinigt die *l-Torsionspunkte* von $E_{a,b}(\overline{\mathbb{F}}_p)$. $E_{a,b}(\mathbb{F}_p)[l]$ selbst wird als die *l-Torsionsgruppe* von $E_{a,b}(\overline{\mathbb{F}}_p)$ bezeichnet.

Der folgende Satz zeigt die für den Schoofalgorithmus wichtige Eigenschaft des Frobenius bei Restriktion auf eine l-Torsionsgruppe.

Satz 3.3.2 (Die Identität des Frobenius) $E_{a,b}(\mathbb{F}_p)[l]$ sei eine l-Torsionsgruppe und

$$\phi_l := E_{a,b}(\mathbb{F}_p)[l] \to E_{a,b}(\mathbb{F}_p)[l],\ (x,y) \mapsto (x^p, y^p)$$

bezeichne die Restriktion des Frobenius auf dieselbe. Für alle $P \in E_{a,b}(\mathbb{F}_p)[l]$ gilt

$$(\phi^2(P) - [\tau_l](\phi_l(P)) + [p](P)) = \mathcal{O}$$

wobei $\tau_l \equiv t \bmod l$ erfüllt.

$\phi^2 = \phi \circ \phi$ bedeutet in diesem Zusammenhang die Hintereinanderausführung des Frobenius. Der Beweis dieses Satzes folgt aus einem etwas allgemeineren Resultat über Endomorphismen von elliptischen Kurven (vgl. [Mirbach, S 24]) und dem Eindeutigkeitsnachweis von $\tau_l \in \{0, \ldots, l-1\}$ (vgl. [Mirbach, S 38-39]).

Nachdem die theoretische Bedeutung der Werte τ_l erklärt wurde, muss nur noch ein Weg zu deren Berechnung gefunden werden. Dazu gehen wir wieder von der Gleichung aus Satz 3.3.2 aus. Gesucht ist also jenes eindeutig bestimmte $\tau_l \in \{0, \ldots, l-1\}$ mit

$$\phi_l^2(P) + [\tau_l]\phi(P) + [p]P = \mathcal{O} \quad (3.11)$$

$$\forall\ P \in E_{a,b}(\mathbb{F}_p)[l]^x := E_{a,b}(\mathbb{F}_p)[l] \backslash \{\mathcal{O}\}.$$

Der Ausdruck (3.11) lässt sich noch etwas vereinfachen. Man kann nämlich $[p]P$ durch $[p_l]P$ mit $p \equiv p_l \bmod l$ ersetzen, da P zu der l-Torsionsgruppe gehört. Weiters können die Ergebnisse von $\phi^2(x,y) = (x^{p^2}, y^{p^2})$ und $\phi(x,y) = (x^p, y^p)$ direkt eingesetzt werden. Das liefert uns den zur Weiterverwendung brauchbaren Ausdruck

$$(x^{p^2}, y^{p^2}) + [p_l]P = [\tau_l](x^p, y^p)\ \forall P = (x,y) \in E_{a,b}(\mathbb{F}_p)[l]^x. \tag{3.12}$$

Zur performanten Auswertung von (3.12) verwendet man spezielle Polynome, über die das Ergebnis der elliptischen Multiplikation $[p_l]P$ ausgedrückt werden kann. In der folgenden Definition werden zu diesem Zweck die Divisionspolynome eingeführt.

Definition 3.3.3(Divisionspolynome einer elliptischen Kurve) Gegeben sei eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$, der man die Divisionspolynome

$$\psi_n \in \mathbb{F}_p[x][y]/(y^2 - x^3 - ax - b) \text{ für } n \geq -1$$

zuordnen kann. Die ψ_n sind wie folgt definiert:

$$\psi_{-1} = -1,$$

$$\psi_0 = 0,$$

$$\psi_1 = 1,$$

$$\psi_2 = 2y,$$

$$\psi_3 = 3x^4 + 6ax^2 + 12bx - a^2,$$

$$\psi_4 = 4y(x^6 + 5ax^4 + 20bx^3 - 5a^2x^2 - 4abx - 8b^2 - a^3)$$

und alle weiteren ergeben sich aus

$$\psi_{2n} = \psi_n(\psi_{n+2}\psi_{n-1}^2 - \psi_{n-2}\psi_{n+1}^2)/2y \quad (n > 2),$$

$$\psi_{2n+1} = \psi_{n+2}\psi_n^3 - \psi_{n+1}^3\psi_{n-1} \quad (n \geq 2).$$

Die Nullstellen von $\psi_l(x,y)$ sind genau die nicht trivialen Punkte der l-Torsionsgruppe. Alle bei der Berechnung der Divisionspolynome nach Definition 3.3.3 vorkommenden Potenzen in y werden über die Gleichung $y^2 = x^3 + ax + b$ reduziert. Polynome mit einem geraden Index haben die Form $yp(x)$ mit $p(x) \in \mathbb{F}_p[x]$, während jene mit einem ungeraden Index nur von x abhängen. Der nächste Satz zeigt eine für den Schoofalgorithmus essentielle

Eigenschaft der Divisionspolynome auf.

Satz 3.3.4(Divisionspolynome) Gegeben sei eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$. Für alle $P \in E_{a,b}(\overline{\mathbb{F}}_p)$ die nicht auch in $E_{a,b}(\mathbb{F}_p)[n]$ liegen gilt

$$[n](x,y) = (x - \frac{\psi_{n-1}\psi_{n+1}}{\psi_n^2}, \frac{\psi_{n+2}\psi_{n-1}^2 - \psi_{n-2}\psi_{n+1}^2}{4y\psi_n^3}).$$

Die Kalkulation der Divisionspolynome in der `EllipticCurve` Library wird von Instanzierungen der Klasse `DivPolyGenerator` durchgeführt. Diese referenziert die Parameter a, b und p einer elliptischen Kurve $E_{a,b}(\mathbb{F}_p)$. Die Startwerte der Folge aus Definition 3.3.3 werden in einem initialen Schritt erzeugt. Mittels der Methode `DivPolyGenerator->GetDivPolynomByIndex(...)` kann man Divisionspolynome mit beliebigem Index berechnen. Über die gesamte Lebensdauer des Objektes führt `DivPolyGenerator` ein Dictionary mit den bisher benötigten Divisionspolynomen. Somit werden redundante Berechnungen innerhalb einer Instanz zur Gänze vermieden. Die Polyome werden als Objekte vom Typ `Dim2Polynomial` geführt. Die weiter oben schon erwähnte Klasse `PolynomKernel` stellt die arithmetischen Operationen für die Polynome in 2 Veränderlichen bereit. Hier die private Methode `DivPolyGenerator->ComputeNextPoly()`, welche über die in Definition 3.3.3 angegebene Rekursion ein Divisionspolynom berechnet.

```
1  private Dim2Polynomial ComputeNextDivPoly() {
2    // select highest available polynomial
3    int index = divPolynomials.Count + 5, l;
4    if (index % 2 == 0) {
5       l = index / 2;
6       Dim2Polynomial tmp1, tmp2, result;
7       tmp1 = PolynomKernel.Mult(GetDivPolynomByIndex(l+2),
8                                 GetDivPolynomByIndex(l-1), n);
9       tmp1 = PolynomKernel.Mult(tmp1,
10                                GetDivPolynomByIndex(l-1), n);
11      tmp2 = PolynomKernel.Mult(GetDivPolynomByIndex(l-2),
12                                GetDivPolynomByIndex(l+1), n);
13      tmp2 = PolynomKernel.Mult(tmp2,
14                                GetDivPolynomByIndex(l+1), n);
15      tmp2 = PolynomKernel.Sub(tmp1,tmp2,n);
16      result = PolynomKernel.Mult(
17                                GetDivPolynomByIndex(l),
18                                tmp2, n);
19      result = Reduce(result, new Dim2Monom(new BigInteger(0),
20                new BigInteger(1), new BigInteger(2)), n);
21      result.ReduceMod(n);
22      divPolynomials.Add(index, result);
```

```
23      return result;
24    } else {
25      l = (index - 1) / 2;
26      Dim2Polynomial tmp1, tmp2, result;
27      tmp1 = PolynomKernel.Mult(GetDivPolynomByIndex(l + 2),
28                                GetDivPolynomByIndex(l),n);
29      tmp1 = PolynomKernel.Mult(tmp1,
30                                GetDivPolynomByIndex(l),n);
31      tmp1 = PolynomKernel.Mult(tmp1,
32                                GetDivPolynomByIndex(l),n);
33      tmp2 = PolynomKernel.Mult(GetDivPolynomByIndex(l+1),
34                                GetDivPolynomByIndex(l+1),n);
35      tmp2 = PolynomKernel.Mult(tmp2,
36                                GetDivPolynomByIndex(l+1),n);
37      tmp2 = PolynomKernel.Mult(tmp2,
38                                GetDivPolynomByIndex(l-1),n);
39      result = PolynomKernel.Sub(tmp1, tmp2, n);
40      result = ReplaceQuadraticYTerms(result,
41                                      UnderlyingCurve, n);
42      result.RemoveZeroMonoms();
43
44      divPolynomials.Add(index,result);
45      return result;
46    }
47  }
```

Verwendet man Satz 3.3.4 für die Berechnung von $[p_l]P$ aus (3.12) und möchte man die Koordinaten der Punkte ungebrochen darstellen, so ist eine Inversenbildung in $\mathbb{F}_p[x]$ für Polynome notwendig. Das kann zwar über den erweiterten euklidischen Algorithmus für Polynome bewerkstelligt werden, ist aber durch eine geschickte Koordinatisierung der Punkte vermeidbar. Die Lösung besteht darin, einen Punkt P von $E_{a,b}(\mathbb{F}_p)$ in der Form

$$P = \Big(\frac{U(x)}{V(x)}, y\frac{G(x)}{H(x)}\Big) \text{ mit } U(x), V(x), G(x), H(x) \in \mathbb{F}_p[x] \tag{3.13}$$

anzuschreiben. Der Faktor y der Y-Koordinate ist in der hier behandelten Implementierung des Schoof-Algorithmus immer vorhanden. Ansonsten stellt er schon eine Beschränkung der Allgemeinheit dar. Das `EllipticCurves` Framework stellt für die Verwaltung von rationalen Ausdrücken der Form $\frac{p(x)}{q(x)}$ mit $p(x), q(x) \in \mathbb{F}_p[x]$ die Klassen `RationalPoint` und `RationalExpression` zur Verfügung. `RationalPoint` implementiert einen Punkt von $E_{a,b}(\mathbb{F}_p)$ in der Darstellung (3.13). Die elliptische Arithmetik für zwei rationale Punkte und diverse Vergleichsoperationen sind als statische Methoden dieser Klasse umgesetzt.

Für die endgültige Implementierung des Schoof-Algorithmus in `EllipticCurve` seien die folgenden Richtlinien ausschlaggebend (vgl. [C & P, S 319]).

1. Die Koordinaten der Punkte sind rationale Ausdrücke in $\mathbb{F}_p[x][y]$.
2. Die höchste vorkommende Potenz von y ist 1, da alle Polynome Modulo $y^2 - x^3 + ax + b$ reduziert werden. Für Punkte aus $E_{a,b}(\mathbb{F}_p)[n]$ wird laufend die Reduktion Modulo ψ_n vorgenommen.
3. Große Potenzen von x werden mittels *Square & Multiply* berechnet, wobei eine permanente Reduktion Modulo dem entsprechenden Polynom erfolgt.

Wie oben schon erwähnt, wird im Schoofalgorithmus durch Probieren das richtige τ_l aus (3.12) bestimmt. Das folgende Codefragment zeigt die Berechnung von $P_0 = (x^p, y^p)$ und $P_1 = (x^{p^2}, y^{p^2})$ und stellt diese Modulo dem aktuellen Divisionspolynom `divPoly_l` dar.

```
// get necessary division polynomial
BigInteger[] divPoly_l = divPG.GetDivPolynomByIndex(l).
    ToArray();
// get reduction of p mod l
BigInteger[] u = PolynomArrayArithmetic.PolyPowerMod(
                      new BigInteger[] { 0, 1 }, Curve.N,
                         divPoly_l, Curve.N);
// = curveEqu ^ (p-1)/2 mod divPoly(l)
BigInteger[] v = PolynomArrayArithmetic.PolyPowerMod(
                       Curve.CurveEquation.ToArray(),
           (Curve.N - 1) / 2,
                       divPoly_l, Curve.N);
// = (X^p,Y^p) = P0 (in rational coordinates)
RationalPoint P0 = new RationalPoint(
              new RationalExpression(u,
                                     new BigInteger[] { 1 }),
              new RationalExpression(v,
                                     new BigInteger[] { 1 }))
                                         ;
// = u^p mod divPoly(l)
BigInteger[] uPower = PolynomArrayArithmetic.PolyPowerMod(u,
                                        Curve.N,
                                        divPoly_l, Curve.N);
// = v^p mod divPoly(l)
BigInteger[] vPower = PolynomArrayArithmetic.PolyPowerMod(v,
                                        Curve.N+1,
                                        divPoly_l, Curve.N);
// = (x^(p^2), Y^(p^2)) = P1 (in rational coordinates)
RationalPoint P1 = new RationalPoint(
```

```
28                      new RationalExpression(uPower,
29            new BigInteger[] { 1 }),
30                      new RationalExpression(vPower,
31                                  new BigInteger[] { 1 }));
```

Anschließend berechnet man $[p_l]P$ über die in Satz 3.3.4 festgehaltene Identität. Dazu implementiert die Klasse `RationalPoint` die Methode `RationalPoint->GetMultiplier(...)`, die als Eingangsparameter unter anderem auch ein Objekt vom Typ `DivPolyGenertor` übernimmt. Auf diese Weise kann auf die benötigten Divisionspolynome zugegriffen werden. Das nächste Codebeispiel zeigt die Kalkulation von $P_2 = [p_l](x, y)$.

```
1 // get reduction of p mod l
2 BigInteger _p = Curve.N % new BigInteger(l);
3 // = [_p](X,Y) = P2
4 RationalPoint P2 = RationalPoint.GetMultiplier(_p, Curve.N,
5                                                  divPG);
```

Nun, da die Punkte P_0, P_1 und P_2 in der Darstellung (3.9) berechnet worden sind, kann man das τ_l durch Probieren ausfindig machen. Eine der beiden Bedingungen

1. $P_1 + P_2 = \mathcal{O} \Rightarrow \tau_l = 0$
2. $\exists! k \in \{1, ..., l-1\} : P_1 + P_2 = [k]P_0 \Rightarrow k = \tau_l$

muss dann gelten und liefert das gesuchte τ_l. Der folgende Code überprüft die beiden Bedingungen und retourniert den Wert für τ_l. Falls der Schoofalgorithmus nicht terminiert, was bei einer elliptischen Kurve nach Definition (1.1.4) nicht der Fall sein kann, wird eine `SchoofUnterminatedException` geworfen. In dem nächsten Kapitel über ECPP wird diese Exception genutzt werden, indem ich den Schoof-Algorithmus auf eine Pseudokurve $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ anwende, von der ich nicht weis, ob n eine Primzahl ist.

```
1  if (RationalPoint.CheckIfSumIsPointAtInfinty(P1, P2,
       divPoly_l,
2                                                 Curve.N))
3         return 0;
4  // compute P1 + P2
5  RationalPoint sum = RationalPoint.Add(P1, P2, Curve.A,Curve.B
       ,
6                                        Curve.N, divPoly_l);
7  sum.ExceedModul(divPoly_l, Curve.N);
8  RationalPoint P3 = P0;
9  for (int i = 1; i <= (l / 2); i++) {
10     if (RationalPoint.CheckIfXCoordinatesMatch(sum, P3,
           divPoly_l,
```

```
11          Curve.N)) {
12        if (RationalPoint.CheckIfYCoordinatesMatch(sum, P3,
              divPoly_l,
13                                                   Curve.N))
14                        return i;
15        return l - i;
16     }
17     P3.ExceedModul(divPoly_l, Curve.N);
18     P0.ExceedModul(divPoly_l, Curve.N);
19     P3.RemoveLeadingZeros();
20     P0.RemoveLeadingZeros();
21     P3 = RationalPoint.Add(P3, P0, Curve.A, Curve.B, Curve.N
           ,
22                            divPoly_l);
23 }
24 throw new SchoofUnterminatedException(l);
```

Zum Abschluss des Schoof-Algorithmus möchte ich noch ein Praxisbeispiel bringen. Versuchen wir die Ordnung der Kurve $E_{a,b}(\mathbb{F}_p)$ mit

$$a = 69771859804340235254,\ b = 10558409492409151218,\ p = 10^{20} + 39$$

zu bestimmen. Die folgende Tabelle zeigt an, nach welcher Zeit wir welchen Wert t mod l erhalten haben.

l	t **mod** l	**Zeit in min:sec**
2	0	0.796
3	0	0.875
5	1	1.218
7	4	3.0
11	4	15.546
13	12	28.531
17	1	1 : 5.156
19	15	2 : 35.140
23	16	8 : 56.078
29	26	26 : 32.609
31	6	51 : 01.500

Nach Anwendung des chinesischen Restsatzes erhalten wir $t = 14124117396$ und können die Ordnung der Kurve $\#E_{a,b}(\mathbb{F}_p) = 99999999985875882644$ bestimmen. Die Zahlen können mit [C & P, S 336] verglichen werden.

3.4 Die Kombination von Schoof, Shanks und Mestre

Da der Schoof-Algorithmus für $l > 31$ praktisch nicht mehr anwendbar ist, empfiehlt es sich τ_l nur für $l \leq l_{ubound} \leq 31$ zu berechnen und die erhaltenen linearen Kongruenzen

$$\tau_l \equiv t \bmod l \tag{3.14}$$

in einem angehängtem Shanks - Mestre Verfahren weiter zu verarbeiten. Konkret bedeutet dies, dass die bei Shanks-Mestre generierten Listen L_1 und L_2, nur aus Elementen $(\beta, \gamma) \in I \times J$ konstruiert werden, die kombiniert als Lösung von (3.14) in Frage kommen. Die Listen in dem Kombinationsverfahren von Shanks und Mestre bezeichne ich mit L_{bs} und L_{gs}. Das lässt sich so umsetzen, indem man die Gleichungen aus (3.14) zu der Bedingung

$$t \equiv r \bmod m \text{ mit } m = \prod_{l \leq l_{ubound},\ l \in \mathbb{P}} l. \tag{3.15}$$

zusammenfasst. r ist das Ergebnis der Anwendung des chinesischen Restsatzes auf (3.14). Berücksichtigt man (3.15) bei der Erzeugung von L_{bs} und L_{gs}, so gilt für einen Punkt $P \in E_{a,b}(\mathbb{F}_p)$

$$L_{bs} = \{[u]P : u \in \{r + j * bs : j = 0, \ldots, \left\lfloor \frac{gs - r}{bs} \right\rfloor\}\},$$

$$L_{gs} = \{[v * gs]P : v \in \{0, \ldots, \left\lfloor \frac{ubound}{gs} \right\rfloor\}\} \text{ mit}$$

$$bs = m,\ ubound = 2\sqrt{p} \text{ und } gs = \sqrt{\left\lfloor \frac{ubound}{bs} \right\rfloor} * bs.$$

Mit diesen stark reduzierten Listen kann nun ein Shanks-Mestre Verfahren analog zu Unterkapitel 3.2 durchgeführt werden.

3.5 Die Erzeugung von $E_{a,b}(\mathbb{F}_p)$ mit einer bestimmten Ordnung

In diesem Unterkapitel gebe ich eine Möglichkeit an elliptische Kurven mit einer bestimmten Ordnung zu konstruieren. Die Ideen stammen dabei von A. 0. L. Atkin und F. Morain. Um den genauen theoretischen Hintergrund besser verstehen zu können, sei auf [Cohen] verwiesen. Ich werde hier nur die

Funktionsweise der Methode beschreiben. Grundlegend für unsere Betrachtungen ist der Begriff der Fundamentaldiskriminante.

Definition 3.3.4 (Fundamentaldiskriminante) Eine negative ganze Zahl D nennt man eine Fundamentaldiskriminante, wenn der größte ungerade Teiler von D kein Quadrat ist und

$$|D| \equiv d \bmod 16 \Rightarrow d \in \{3, 4, 7, 8, 11, 15\}$$

gilt.

Ausgehend von einer Primzahl p und einer Fundamentaldiskriminante D mit $(\frac{D}{p}) = 1$ kann man, vorausgesetzt es gilt

$$(3.16) \quad \exists (u, v) \in \mathbb{N} \times \mathbb{N} : 4p = u^2 + |D|v^2,$$

eine Kurve mit einer Ordnung, die aus den Parametern u, v und p auf einfache Weise kalkulierbar ist, erzeugen. Der folgende Satz gibt die möglichen Kurvenordnungen und Kurven für $D \in \{-3, -4\}$ an.

Satz 3.3.5 (Kurvenordnung für $D \in \{-3, -4\}$) Für ein $D \in \{-3, -4\}$ mit $(\frac{D}{p}) = 1$ und einen quadratischen nicht Rest $g \bmod p$, der im Fall $p \equiv 1 \bmod 3$ die Bedingung $g^{\frac{p-1}{3}} \not\equiv 1$ erfüllt (Für $D = -3$ muss man zusätzlich fordern, dass die kubische Gleichung $x^3 = g$ keine Lösung in $\mathbb{F}_p$ hat.), dann haben die elliptischen Kurven der Gestalt $E_{a,b}(\mathbb{F}_p)$ für $D = -4$ die Parameter

$$(a, b) \in \{(-g^k \bmod p, 0) : k \in \{0, 1, 2, 3\}\}$$

und für $D = -3$

$$(a, b) \in \{0, (-g^k \bmod p) : k \in \{0, 1, 2, 3, 4, 5\}\}.$$

Die möglichen Ordnungen dieser Kurven sind dann für $D = -4$ durch

$$\{p + 1 \pm u, p + 1 \pm 2v\}$$

und für $D = -3$ durch

$$\{p + 1 \pm u, p + 1 \pm (u \pm 3v)/2\}$$

gegeben. (u, v) erfüllen dabei gemeinsam mit D und p (3.16).

Findet man nun zu einem gegebenen $p \in \mathbb{P}$ Elemente g und D, die den

Bedingungen aus Satz 3.3.5 genügen, so kann man bereits Kurven mit einer bestimmten Ordnung konstruieren. Die Zuordnung der Ordnungen zu den elliptischen Kurven kann in der Praxis mit zufällig generierten Punkte ausgetestet werden. Doch was macht man, wenn die Diskriminanten -3 und -4 für das Verfahren nicht geeignet sind ? In diesem Fall kann man sich durch Tabellierung diverser Werte zur Kurvenkonstruktion helfen. Die Tabelle findet man in [C & P, S 331] und wird von mir als T bezeichnet. Sie enthält für die Diskriminanten bis zu -427 zwei Zahlen r und s. r kann dabei auch durch einen Ausdruck der Form $a+b\sqrt{c}$ für $a, b \in \mathbb{Z}$ und $c \in \mathbb{N}$ gegeben sein. s hingegen ist durchgehend durch eine natürliche Zahl gegeben. Beim Auslesen aus dieser Tabelle werden die Zahlen Modulo p reduziert und können wie folgt verwendet werden.

Satz 3.3.6 (Kurvenkonstruktion für $-4 > D \geq -427$) Sei g eine ganze Zahl mit denselben Eigenschaften wie in Satz 3.3.5. Für eine Fundamentaldiskriminante D aus dem Bereich [-427,-7] gelte für ein beliebiges $p \in \mathbb{P}$ $(\frac{D}{p}) = 1$. Dann kann man über die Tabelle T elliptische Kurven der Form $E_{a,b}(\mathbb{F}_p)$ mit Parametern

$$(a, b) \in \{(-3rs^3g^{2k}, 2rs^5g^{3k}) : k \in \{0, 1\}\}$$

erzeugen, deren Ordnungen durch

$$\{p + 1 \pm u\}$$

gegeben sind. (u, v) erfüllen dabei gemeinsam mit D und p (3.16). Die Werte r und s aus der Tabelle sind Modulo p zu interpretieren.

Die Konstruktion von Kurven nach diesem Muster wird in der Klasse `CMCurveGeneration` implementiert. Diese kann von einem Wert p ausgehend eine Kurve nach den Sätzen (3.3.5) und (3.3.6) konstruieren. Die Tabelle T ist dazu in einer XML Datei abgelegt.

Kapitel 4

ECPP

In diesem Kapitel beschäftige ich mich mit Primalitätsbeweisen, basierend auf elliptischen Kurven. Goldwasser und Kilian gelang es unter Verwendung von Pseudokurven (vgl. Definition (2.1.1)) einen Primzahltest, mit einer Komplexität von $O(ln(n)^k)$ anzugeben. n ist der Testkandidat und k eine absolute Konstante. Die dazu benötigte Theorie liefert ebenfalls ein Satz von Goldwasser und Kilian.

Satz 4.1.1 (Goldwasser - Kilian ECPP Theorem) Sei $n > 1$ eine natürliche Zahl mit $ggT(6, n) = 1$. $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ bezeichnet eine Pseudokurve nach Definition (2.1.1). Weiters betrachtet man $s, m \in \mathbb{N}$ mit $s|m$. Wählen wir nun ein $P \in E_{a,b}(\mathbb{Z}/n\mathbb{Z})$, sodass die folgenden elliptischen Multiplikationen definiert sind, und die Ergebnisse

$$[m]P = \mathcal{O}$$

und

$$\forall\ q \in \mathbb{P} : q|s\ \Rightarrow\ [\frac{m}{q}]P \neq \mathcal{O}$$

liefern. Dann gilt,

$$\forall\ p \in \mathbb{P} : p|n\ \Rightarrow\ \#E_{a,b}(\mathbb{F}_p) \equiv 0 \bmod s.$$

Für $s > (n^{\frac{1}{4}} + 1)^2$ folgt sogar die Primalität von n.

Beweis p sei ein Primfaktor von n. Betrachtet man die elliptischen Multiplikationen Modulo p, so beweist das $s|\#E_{a,b}(\mathbb{F}_p)$. Wenn nun zusätzlich $s > (n^{\frac{1}{4}} + 1)^2$ gilt, dann kann man $\#E_{a,b}(\mathbb{F}_p) > (n^{\frac{1}{4}} + 1)^2$ folgern. Die Ordnung von $E_{a,b}(\mathbb{F}_p)$ ist allerdings durch den Satz von Hasse nach oben durch

$(p^{\frac{1}{2}}+1)^2$ beschränkt, weshalb

$$p^{\frac{1}{2}} > n^{\frac{1}{4}} \Leftrightarrow p > n^{\frac{1}{2}}$$

gelten muss. Das bedeutet für n, dass alle seine Primfaktoren größer als $\sqrt{n}$ sind. Da dies eine Eigenschaft ist die nur Primzahlen besitzen, folgt $n \in \mathbb{P}$.

Goldwasser und Kilian haben aufbauend auf Satz 4.1.1 einen Primzahltest entwickelt. Für eine natürliche Zahl n als Eingangsparameter liefert die Methode entweder die Aussage *n ist keine Primzahl* oder *n ist eine Primzahl, wenn q eine Primzahl ist* für ein $q < n$. Auf das Resultat q kann dann das Verfahren erneut angewendet werden. Benötigt werden in diesem Zusammenhang elliptische Kurven, deren Ordnungen genügend große Pseudoprimteiler besitzen. Ein solcher Teiler übernimmt quasi die Rolle von s in Satz 4.4.1. Zur Erklärung sind hier die Schritte des Verfahrens aufgelistet, wobei $n \in \mathbb{N}$ den Kandidaten für den Primzahltest bezeichnet.

1. *Wahl einer Pseudokurve :* Man wählt eine Pseudokurve $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$ nach Definition (2.1.1).

2. *Berechnung von $m = \#E_{a,b}(\mathbb{Z}/n\mathbb{Z})$:* m ist das Ergebnis einer Anwendung des Schoof-Algorithmus auf $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$. Falls der Schoof-Algorithmus für ein bestimmtes $l \in \mathbb{P}$ kein Ergebnis liefert oder m nicht im Hasseintervall liegt, dann gilt $n \notin \mathbb{P}$.

3. *Faktorisiere $m = k * q$:* Die Zahl $m = k * q$ muss faktorisiert werden, sodass letztendlich $q > (n^{\frac{1}{4}}+1)^2$ gilt. Um eine brauchbare Aussage erhalten zu können, sollte q eine Pseudoprimzahl sein. Erfüllt keiner der Faktoren von m diese Bedingungen, so wird zu Schritt 1 zurückgekehrt.

4. *Wahl eines Punktes auf $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$:* Zuerst wählt man zufällig ein $x \in [0, n-1]$, sodass $Q = x^3+ax+b \bmod n$ die Bedingung $(\frac{Q}{n}) \neq -1$ erfüllt. Nun kann mit deterministischen Algorithmen die Quadratwurzel von Q mod n berechnet werden. Liefern diese kein korrektes Ergebnis, so kann $n \notin \mathbb{P}$ gefolgert werden. Andernfalls kann man einen Punkt P mit $P_x = x$ definieren.

5. *Punktoperationen durchführen:* P sei der im vorigen Schritt generierte Punkt von $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$. Zu Beginn berechnet man $U = [\frac{m}{q}]P$. Wenn für das Ergebnis $U = \mathcal{O}$ gilt, so muss man einen Schritt zurückgehen und einen neuen Punkt wählen. Andernfalls wird $V = [q]U$ gebildet. Aus $V \neq \mathcal{O}$ folgt dann $n \notin \mathbb{P}$. Für $V = \mathcal{O}$ wird schließlich die Aussage $n \in \mathbb{P} \Leftarrow q \in \mathbb{P}$ retourniert.

Bevor ich auf die Details der Implementierung eingehe, werde ich noch die Verbesserung von Atkin und Morain zu diesem Algorithmus vorstellen. Grundlegende Idee ist es, anstatt der zufällig generierten Kurven im Goldwasser - Kilian - Verfahren Kurven mit bekannter Ordnung zu verwenden. Dazu greift man auf die in Kapitel 3 behandelte Kurvenkonstruktion nach Atkin und Morain zurück. Diese hat den Vorteil, dass man zu einer Auswahl von Ordnungen eine passende elliptische Kurve erzeugen kann. Dazu werden die folgenden Schritte durchgeführt.

1. *Wahl einer passenden Diskriminante :* In diesem initialen Schritt muss man eine Fundamentaldiskriminante D mit $(\frac{D}{n})$ und für die eine Lösung $(u, v) \in \mathbb{N} \times \mathbb{N}$ von $4n = u^2 + |D|v^2$ existiert, aussuchen.

2. *Bestimmung der möglichen Kurvenordnungen:* Über die Sätze 3.3.5 und 3.3.6 kann man sich für die aktuelle Fundamentaldiskriminante D, die möglichen Kurvenordnungen herleiten.

3. *Faktorisierung der Kurvenordnungen:* Ziel dieses Schritts ist es, eine der zuvor berechneten Kurvenordnungen $m = k * q$ so zu faktorisieren, sodass q eine Pseudoprimzal ist und $q > (n^{\frac{1}{4}} + 1)^2$ gilt. Dabei probiert man einfach alle zuvor berechneten Ordnungen aus. Wird man nicht fündig so muss man mit Schritt 1 weitermachen.

4. *Erzeugung der Kurve:* Hat man in dem vorherigen Schritt ein m gefunden, so kann die Kurve zu diesem m erzeugt werden.

5. *Wahl eines Punktes auf $E_{a,b}(\mathbb{Z}/n\mathbb{Z})$:* Ist analog zu Goldwasser - Kilian-Test.

6. *Punktoperationen durchführen:* Ist analog zu Goldwasser - Kilian-Test.

Jetzt stelle ich die Implementierung der Methoden vor. Da sich die Verfahren von Goldwasser - Kilian und Atkin - Morain nur durch die Strategie bei der Wahl von Kurven unterscheiden, bietet es sich an, eine gemeinsame Basisklasse zu verwenden. Diese Klasse kann den gesamten Prozess des Verfahrens abbilden und für die unterschiedlichen Schritte passende Schnittstellen anbieten. In dem `EllipticCurves` Framework gibt es dazu die abstrakte Klasse `ECPPBase`, welche die Methoden `ECPPBase->AttemptToFactor(...)`, `ECPPBase->ChoosePoint(...)` und `ECPPBase->OperateOnPoint(...)` implementiert. Zusätzlich hat `ECPPBase` auch die virtuelle Methode `ECPPBase ->PrecomputingParameter(...)`, die von etwaigen Ableitungen überschrieben werden muss. Die Methode `ECPPBase.Execute()` bildet den zu Grunde

liegenden Prozess beim ECPP ab. Um übergeordneten Schichten Nachrichten über den Status des Primalitätsbeweises senden zu können, wird das Event `OnNewStateAvailable` gefeuert und gibt eine entsprechende Statusmeldung weiter.

```
1  public BigInteger Execute() {
2
3     // get initial parameter data
4     EllipticCurve curve
5     BigInteger m, q;
6
7     // Fire event precomputing parameter
8     OnNewStateAvailableWrapper(this,
9         new OnNewStateInfoAvailableEventArgs(
10            "Precomputing  \r\n parameter \r\n data"));
11
12    // start precomputation of parameter data
13    PrecomputingParameters(out curve, out m, out q);
14
15    // Fire event operating point
16    OnNewStateAvailableWrapper(this,
17        new OnNewStateInfoAvailableEventArgs(
18            "Operation on Point."));
19
20    // choose point and operate on it
21    OperateOnPoint(curve, m, q);
22
23    // if q is prime then n is prime
24    return q;
25 }
```

Die Methode `OperateOnPoint(...)` führt die Schritte *Wahl eines Punktes* und *Punktoperation durchführen* aus. Der erste Schritt erfolgt über `ECPP-Base->ChoosePoint(...)`, was im Anschluß abgedruckt ist. Da man mit einer Pseudokurve arbeitet, werden die elliptischen Multiplikationen mit try and catch überwacht. Falls die Summe zweier Punkte nicht definiert ist, so wird eine Exception vom Typ `NotInvertibleException` geworfen. Dieses Konzept habe ich bereits bei der Implementierung der ECM Varianten (vgl. Kapitel 2) angewendet. Tritt in diesem Zusammenhang eine solche Exception auf, so wird sie abgefangen und an ihrer Stelle eine `IsCompositeNumber-Exception` geworfen. Das Auftreten einer `IsCompositeNumberException` während eines ECPP Durchlaufs bedeutet, dass n als zusammengesetzt erkannt wurde.

```
protected bool OperateOnPoint(EllipticCurve curve,
                              BigInteger m,
                              BigInteger q) {
    AffinePoint P, U, V;
    try {
         do {
           P = ChoosePoint(curve);
           U = curve.Mult(P, m / q);
         } while (U.IsPointAtInfinity());
    if (!(V = curve.Mult(U, q)).IsPointAtInfinity())
          throw new IsCompositeNumberException(N);
    } catch (Exception ex) {
        if(ex is NotInvertibleException)
           throw new IsCompositeNumberException(N);
        throw ex;
    }
return true;
}

protected AffinePoint ChoosePoint(EllipticCurve curve){
    BigInteger x, y = null, tmp;
    do{
        x = BigInteger.GetRandom(N - 1);
     }while((tmp = curve.Eval(x)).Jacobi(N) == -1);
     try {
         y = tmp.modSqrt(N);
     } catch (Exception ex) {
         if (ex is NoSquareRootExistsException)
             throw new IsCompositeNumberException(N);
     }
     if (y.modPow(2, N) != tmp)
         throw new IsCompositeNumberException(N);
     return new AffinePoint(x, y, 1);
}
```

Die Methode `ECPPBase->AttemptToFactor(...)` werde ich hier nicht abdrucken. Zur Faktorisierung von $m = k * q$ wird eine ECM Stage 2 Implementierung verwendet. Der Primzahlbeweis von Goldwasser - Kilian ist in der Klasse `ECPPGoldwasserKilian` umgesetzt, die wie oben bereits erwähnt wurde, `ECPPBase` als abstrakte Basisklasse hat. Hier wird die virtuelle Methode `ECPPBase->PrecomputingParameter(...)` überschrieben, sodass die Wahl der Kurve zufällig erfolgt. Für die zufällige Wahl einer Pseudokurve wird in der Klasse `EllipticCurve` die statische Methode `EllipticCurve->GetRandomCurve(...)` bereitgestellt. Anschließend wird nach dem Schoof-Algorithmus versucht, die Ordnung der Pseudokurve zu bestimmen. Falls der Schoof-Algorithmus nicht terminiert, so kann man folgern, dass n nicht prim ist. In diesem Fall wirft die Methode `ClassicalSchoof.Execute()` ei-

ne Exception vom Typ `SchoofUnterminatedException`. Tritt eine solche in der unten angeführten Methode auf, so wird sie gefangen und durch eine `IsCompositeNumberException` ersetzt. Sollte der Schoof-Algorithmus terminieren, dann muss das Ergebnis in dem Hasseintervall liegen. Ist das nicht der Fall, wird wieder eine `IsCompositeNumberException` geworfen. Die Schleife wird solange durchlaufen, bis ein passendes m gefunden wird.

```
protected override void PrecomputingParameters(
                            out EllipticCurve curve,
                            out BigInteger m,
                            out BigInteger q) {
  BigInteger hasseUBound, hasseLBound;
  ClassicalSchoof schoof;
  q = null;
  m = null;
  do {
     curve = EllipticCurve.GetRandomCurve(N);
     schoof = new ClassicalSchoof(curve);
     try {
         m = schoof.Execute();
     } catch (Exception ex) {
         if(ex is SchoofUnterminatedException)
             throw new IsCompositeNumberException(N);
     }
     curve.GetHasseIntervall(out hasseUBound, out
         hasseLBound);
     if (m < hasseLBound || m > hasseUBound)
         throw new IsCompositeNumberException(N);
  } while ((q = AttemptToFactor(m)) != null);
}
```

Die Verbesserungsvorschäge von Atkin und Morain sind in der Klasse `ECPP-AtkinMorain` realisiert worden. In einer Überschreibung von `ECPPBase->PrecomputingParameter(...)` wird zu einem zuvor gewähltem $m = q * k$ (vgl. Atkin - Morain Schritt 3) eine Pseudokurve erzeugt, die wenn n prim wäre, die Ordnung m hätte. Dazu wird wieder eine Instanz von der im vorherigen Kapitel vorgestellten Klasse `CMCurveGeneration` erzeugt und mit einer Diskriminante D und Werten $(u, v) \in \mathbb{N} \times \mathbb{N}$, die $4n = u^2 + |D|v^2$ erfüllen, initialisiert. Anschließend können über die Methoden `CMCurveGeneration->GetPossible-CurveOrders()` und `CMCurveGeneration->GetCorrespondingCurve(...)` die benötigten elliptischen Kurven generiert werden.

```
protected override void PrecomputingParameters(
                           out EllipticCurve curve,
                           out BigInteger m,
                           out BigInteger q) {
   CMCurveGeneration cmGen = new CMCurveGeneration(N);
```

```
6       int discrIndex = 0, D;
7       CornacchiaSmith.CornacchiaSmithResult csResult;
8       List<BigInteger> orders;
9       do {
10          while (!cmGen.SearchForDiscriminant(
11                      N, ref discrIndex,
12                      out D, out csResult)) ;
13          cmGen.Discrminant = D;
14          cmGen.CsResult = csResult;
15          orders = cmGen.GetPossibleCurveOrders();
16          q = null;
17          m = null;
18          foreach (BigInteger currOrder in orders)
19              if ((q = AttemptToFactor(m = currOrder)) != null
                    &&
20                   NumberTheoryUtilities.FermatTest(q, 2))
21                      break;
22      } while (q == null && discrIndex < cmGen.Discriminants.
            Length);
23      curve = cmGen.GetCorrespondingCurve(m);
24  }
```

Zum Abschluss dieses Kapitels möchte ich noch auf den Einsatz der Methode in der Praxis eingehen. Prinzipiell ist es nicht empfehlenswert ECPP für die Suche nach neuen Primzahlen heranzuziehen, da die Faktorisierung von m sehr viel Zeit in Anspruch nehmen kann. Hat man aber ein n gegeben, das mit hoher Wahrscheinlichkeit prim ist, so kann man die Primalität von n mit den hier vorgestellten Methoden beweisen. Das Produkt eines ECPP ist eine Liste von Werten, wobei ein Eintrag die Form

$$(n_i, a_i, b_i, m_i, q_i, P_i) \in \mathbb{N} \times \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z} \times \mathbb{N} \times \mathbb{N} \times E_{a,b}(\mathbb{Z}/n\mathbb{Z})$$

hat. Hier handelt es sich um die Parameter des i-ten ECPP Durchlaufs, bei einer rekursiven Anwendung des Verfahrens auf $n = n_0$. Existiert eine solche Auflistung für ein n bis zu einem q_i von dem man $q_i \in \mathbb{P}$ weiß, so kann diese quasi als Echtheitszertifikat für die Primalität von n verwendet werden.

Kapitel 5

Kryptographie mit elliptischen Kurven

In diesem Kapitel werden kryptographische Anwendungen von elliptischen Kurven behandelt. Zu Beginn gehe ich auf das Problem des diskreten Logarithmus auf $E_{a,b}(\mathbb{F}_p)$ ein und erkläre warum es als hinreichend schwer für Datenverschlüsselungssysteme angesehen wird. In diesem Zusammenhang erwähne ich auch einige Verfahren zur Bestimmung des diskreten Logarithmus. Daraus kann man dann Kriterien für die Eignung von Kurven zur Datenverschlüsselung herleiten. Danach werden, basierend auf elliptischen Kurven, die ElGamal Methoden zur asymmetrischen Verschlüsselung (ECES) und zur Digitalen Signatur (ECDSA) erklärt und implementiert. Zum Abschluss gibt es noch einen Vergleich mit dem im europäischen Raum weit verbreiteten RSA System. Aus den Vorteilen ergeben sich auch die Einsatzgebiete für ECC und ECDSA.

5.1 Das Problem des diskreten Logarithmus auf $E_{a,b}(\mathbb{F}_p)$

Die asymmetrische Verschlüsselung von Daten mit elliptischen Kurven (ECC) passiert nach dem Verfahren von ElGamal, wobei man als zu Grunde liegende Gruppe $E_{a,b}(\mathbb{F}_p)$ wählt (vgl. [Werner]). Die Sicherheit des Verfahrens stützt sich damit auf die Schwierigkeit, den diskreten Logarithmus auf $E_{a,b}(\mathbb{F}_p)$ berechnen zu können. Dieses Problem wird in der Literatur kurz als ECDLP (Elliptic Curve Discrete Logarithm Problem) bezeichnet.

Definition 5.1.1 (ECDLP) Gegeben sei eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$ und ein Punkt $P \in E_{a,b}(\mathbb{F}_p)$ mit $Ord_{E_{a,b}(\mathbb{F}_p)}(P) = n$. P erzeugt somit eine zyklische Untergruppe $<P>$ von $E_{a,b}(\mathbb{F}_p)$. Hat man nun ein $Q = [l]P$ aus $<P>$ mit $l < n$ gegeben, dann ist das ECDLP durch die Bestimmung der natürlichen Zahl l gegeben.

Nun kennt man bestimmte Verfahren zur Lösung des ECDLP auf elliptischen Kurven, die alle bei geeigneter Wahl von P und $E_{a,b}(\mathbb{F}_p)$ praktisch nicht in absehbarer Zeit determinieren können. Die Methoden kann man in zwei Kategorien einteilen. Zum einen gibt es die Methoden, die quasi für jede abelsche Gruppe einsetzbar sind und zum anderen kennt man Algorithmen, die zielgerichtet auf elliptische Kurven implementiert sind. Diese Verfahren werden jetzt vorgestellt, wobei und $P \in E_{a,b}(\mathbb{F}_p)$ mit $Ord_{E_{a,b}(\mathbb{F}_p)}(P) = n$ gilt. Ich beginne mit den allgemein gültigen Verfahren.

1. *Bildung von Vielfachen :* Hier werden einfach die Vielfachen $[k]P$ für $0 \leq k < n$ gebildet. Da man wirklich alle k ausprobieren muss, kann diese Methode nur für kleine n eingesetzt werden.

2. *Babystep - Giantstep - Algorithmus :* Diese Methode habe ich in Kapitel 3 bei dem Hybridverfahren von Shanks, Mestre und Schoof angewendet. Eine allgemeine Beschreibung des Verfahrens für eine beliebige zyklische Gruppe G kann in [C & P, S 200 - 202] nachgelesen werden. Die benötigten Listen erfordern einen Speicherplatz von $O(\sqrt{m})$, wobei m die Ordnung von G bezeichnet. Bildung des Durchschnitts und Sortierung der Listen ergeben einen Aufwand von $O(\sqrt{m} * ln(m))$ Operationen.

3. *Pholig - Hellman - Verfahren* In diesem Verfahren wird das Problem des diskreten Logarithmus auf Untergruppen der zu Grunde liegenden Gruppe reduziert. l wird dazu Modulo jedes Primteilers von n berechnet und kann dann mit dem chinesischen Restsatz effektiv berechnet werden. Diese Methode ist nur dann praktikabel wenn n in kleine Primteiler zerfällt (vgl. [Werner, S 77 - 79]).

4. *Die Pollardsche - ρ -Methode* Dieser Algorithmus hat eine Komplexität von $O(\sqrt{n})$ Gruppenoperationen. Er benötigt wenig Speicherplatz und lässt sich leicht und effektiv parallelisieren.

5. *Die Pollardsche - λ - Methode* Das Verfahren ist prinzipiell langsamer als die ρ Methode. Kann man allerdings den diskreten Logarithmus auf ein Intervall einschränken, terminiert sie rascher. Gute Parallelisierbarkeit zählt auch zu den Vorteilen dieser Methode.

Die nächste Auflistung zeigt die speziell für elliptische Kurven ausgerichteten Algorithmen.

1. *Der MOV - Algorithmus* Bei diesem Verfahren kann das ECDLP in $E_{a,b}(\mathbb{F}_p)$ auf das DL-Problem in $\mathbb{F}^x_{q^l}$ reduziert werden. Eine Gefahr für die Sicherheit besteht nur, wenn es möglich ist, l hinreichend klein zu halten. Speziell für supersinguläre Kurven kann der MOV-Algorithmus das ECDLP in probabilistisch subexponentieller Zeit lösen. Eine Kurve $E_{a,b}(\mathbb{F}_p)$ bezeichnet man als supersingulär, wenn p die Spur des Frobenius $tr(\phi) = p + 1 - \#E_{a,b}(\mathbb{F}_p)$ teilt.

2. *Der SSSA - Algorithmus* Diese Methode kann das ECDLP für anomale elliptische Kurven in polynomialer Laufzeit lösen. Eine Kurve heißt anomal, wenn $\#E_{a,b}(\mathbb{F}_p) = p$ gilt.

Aus den optimalen Rahmenbedingungen dieser Methoden kann man die Eigenschaften **starker elliptischer Kurven** ableiten. Unter einer starken elliptischen Kurve versteht man eine Kurve, die sich für kryptographische Anwendungen eignet. Die soeben angeführten Methoden dürfen bei diesen Kurven praktisch keine Erfolgschancen haben.

Definition 5.1.2(starke elliptische Kurve) Gegeben sei eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$ mit den zusätzlichen Eigenschaften

1. $n = \#E_{a,b}(\mathbb{F}_p)$ sollte idealerweise prim sein oder zumindest einen großen Primteiler enthalten.

2. Aufgrund des SSSA-Verfahrens muss $\#E_{a,b}(\mathbb{F}_p) \neq p$ gelten.

3. $Min(\{l \in \mathbb{N} : p^l \equiv 1 \bmod n\}) \geq 20$ (vgl. [Mirbach, S 111]).

Der Punkt 1 sorgt für die Ineffizienz des Pholig - Hellman - Verfahrens und 2 schließt eine Anwendung des SSSA-Algorithmus aus. Aufgrund von Punkt 3 ist der MOV-Algorithmus nicht mehr praktikabel.

Möchte man bei der Wahl einer Kurve kein Risiko eingehen, so hat man die Möglichkeit, aus 15 von NIST (U.S. National Institute of Standards and Technology) vorgeschlagenen Kurven zu wählen. Diese wurden aufwendig getestet und zählen mit an Sicherheit grenzender Wahrscheinlichkeit zu den starken elliptischen Kurven.

5.2 Die asymmetrische Verschlüsselung

Die asymmetrische Verschlüsselung funktioniert, wie ich oben bereits erwähnt habe, nach dem Verfahren von ElGamal. Angenommen, die zwei Personen, Agnes und Bruno wollen miteinander verschlüsselte Nachrichten austauschen. Jeder der beiden verfügt dazu über einen privaten und öffentlichen Schlüssel. Weiters wird auch ein Tupel von Systemparametern

$$(5.1) \quad (p, a, b, x, y, n)\ \mathbb{P} \times \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{P}$$

benötigt. p ist die Charakteristik eines endlichen Primkörpers $\mathbb{F}_p$. Über a und b ist dann die Kurve $E_{a,b}(\mathbb{F}_p)$ definiert. Weiters existiert ein $P \in E_{a,b}(\mathbb{F}_p)$ mit $P = (x, y)$. n steht für die Ordnung von P. Wenn man sich nicht dazu entscheidet, für n eine Primzahl zu wählen, dann wird ein Tupel der Form

$$(5.2) \quad (p, a, b, x, y, n, c)\ \mathbb{P} \times \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{N} \times \mathbb{N}$$

herangezogen. (5.2) enthält, verglichen mit (5.1), ein weiteres Element $c \in \mathbb{N}$, das man auch als Kofaktor bezeichnet. Dieses ist gegeben durch $c = \frac{\#E_{a,b}(\mathbb{F}_p)}{n}$ und gibt das Verhältnis der Gesamtgruppenordnung zu $Ord_{E_{a,b}(\mathbb{F}_p)}(P)$ an. Wenn $E_{a,b}(\mathbb{F}_p)$ eine Primzahlordnung hat, dann ist der Kofaktor gleich 1.

Die Systemparameter werden in der `EllipticCurves` Library in Instanzen von `ECDSSystemParameter` abgelegt. Es bestehen die Möglichkeiten, Instanzierungen über die bereits vorgestellten NIST - Kurven vorzunehmen. Diese Kurvenparameter habe ich in XML persistiert und in der Enumeration `NistCurveId` ihre Namen zusammengefasst. Man kann die ID einer NIST - Kurve über diese Enumeration laden und dem Konstruktor von `ECDSSystemParameter` übergeben. Man erhält somit in einer Codezeile ein nach NIST - Empfehlung erzeugtes Systemtupel.

Agnes und Bruno können sich aufbauend auf den Systemparametern (5.1) ihren privaten und öffentlichen Schlüssel erzeugen. Angenommen, Agnes generiert sich ein Schlüsselpaar, dann sind die dazu notwendigen Schritte in der folgenden Auflistung angeführt.

1. Agnes wählt zufällig eine Zahl $d \in [1, n-1]$. Dabei handelt es sich um ihren privaten Schlüssel, der geheim gehalten werden muss.
2. Sie berechnet im Anschluss den Punkt $Q = [d]P$.
3. Q bildet den zu dem privaten Schlüssel d korrespondierenden öffentlichen Schlüssel.

Aus dem letzten Schritt kann man entnehmen, dass das miteinander assoziierte Schlüsselpaar die Gestalt

$$(Q, d) \in E_{a,b}(\mathbb{F}_p) \times I \text{ mit } I = \{0, 1, \ldots, n-1\}$$

hat. Natürlich muss Agnes ihren öffentlichen Schlüssel gemeinsam mit den, bei der Generierung verwendeten Systemparametern, die ich kurz mit T bezeichne, veröffentlichen.

Sämtliche Teilnehmer an einem ECC-System werden auf Objekte vom Typ `ECDSParticipant` abgebildet. Diese Klasse verfügt über die Methode `ECDSParticipant->GenerateKeyPair(...)`, die mehrfach überladen ist. Der geheime und der öffentliche Schlüssel werden in den zu `ECDSParticipant` gehörigen Membervariablen `PrivateKey` vom Typ `BigInteger` und `PublicKey` vom Typ `ECDSPublicKey` abgelegt. `ECDSPublicKey` Objekte referenzieren immer auf die Instanzen von `ECDSSystemparameter`, die zu ihrer Erzeugung herangezogen wurden. Weiters enthalten sie auch die Methode `ECDSPublic-Key->ToString()`, über die sie in einen äquivalenten String konvertiert werden können. Damit ist es leicht möglich diese Objekte zu serialisieren, beziehungsweise in einem Verzeichnis darzustellen, was wesentliche Punkte für den Einsatz in einem Public-Key-Kryptosystem sind. Für die Praxis ebenfalls interessant ist die Persistierung des Punktes Q. Es ist nämlich ausreichend die X-Koordinate und das least significant Bit der Y-Koordinate zu speichern. Wenn für die X-Koordinate 2 Werte als zugehörige Y-Koordinate vorkommen, so unterscheiden sich diese in dem least significant Bit, da deren Summe 0 (Modulo p) ergeben muss. Zu diesem Zweck gibt es die Klasse `CompAffinePoint`, welche die X-Koordinate und das least significant Bit der korrespondierenden Y-Koordinate eines affinen Punktes verwaltet. Es werden passende Konstruktoren für `CompAffinePoint` angeboten, die als Eingagsparameter vollwertige affine Punkte (d.h.: Objekte vom Typ `AffinePoint`) übernehmen und einen komprimierten affinen Punkt erzeugen. Das folgende Codebeispiel zeigt zwei Überladungen für die Methode `ECDSParticipant->GenerateKeyPair(...)`. Im ersten Beispiel wird die Erzeugung der Systemparamter unter Verwendung von NIST Kurven praktiziert. Das nachfolgende Codesample zeigt die Konstruktion von T mit einer beliebigen Kurve und zugehörigem Startpunkt.

```
public void GenerateKeyPair(NistCurveId requiredNistCurve) {

    // get system parameter data
    ECDSSystemParameter sp =
                 new ECDSSystemParameter(requiredNistCurve);

    // get private key randomly
    privateKey = BigInteger.GetRandom(sp.CurveOrder);

    // calculate public key point
    publicKey = new ECDSPublicKey(sp, privateKey);
}
```

```
public void GenerateKeyPair(EllipticCurve cmCurve,
                            AffinePoint P) {

    // generate System parameter
    ECDSSystemParameter sp =
       new ECDSSystemParameter(cmCurve, P, cmCurve.Order);

    // get private key randomly
    privateKey = BigInteger.GetRandom(sp.CurveOrder);

    // calculate public key point
    publicKey = new ECDSPublicKey(sp, privateKey);
}
```

Bruno möchte nun Agnes eine verschlüsselte Nachricht zukommen lassen. Dazu muss er die folgenden Schritte durchlaufen.

1. Bruno liest den öffentlichen Schlüssel Q und die Systemparameter T von Agnes aus einem Verzeichnis ein.

2. Die Nachricht wird auf Elemente $m_i \in \mathbb{F}_p$ mit $0 \leq i \leq B(m)$, wobei B von der Länge des Klartextes abhängt, abgebildet. Diese Abbildung wird weiter unten beschrieben.

3. Er wählt gewisse Werte $k_i \in [1, n-1]$ mit $0 \leq i \leq B(m)$.

4. Für jedes k_i aus dem vorangegangenem Schritt werden $[k_i]Q = (x_{i_2}, y_{i_2})$ und $[k_i]P = (x_{i_1}, y_{i_1})$ berechnet.

5. Dann bildet Bruno $m_i x_{i_2} \equiv c_i$ Modulo p.

6. Für jedes in dem vorigen Schritt erzeugtes c_i sendet Bruno ein Tupel (x_{i_1}, y_{i_1}, c_i) an Agnes.

Zuerst möchte ich auf die Abbildung der Nachricht auf Elemente in $\mathbb{F}_p$ eingehen. Dazu gibt es in `EllipticCurve` die Klasse `ECDSInternalMessage`. Diese hat den string `PlainText` und das `BigInteger` Array `SplittedPlaintext` als Member. Eine Instanz von `ECDSInternalMessage` kann über einen Konstruktor, der einen den Klartext repräsentierenden string übernimmt, erzeugt werden. Dabei wird der Member `PlainText` auf den Klartext gesetzt. Anschließend ist es möglich, über die Methode `ECDSInternalMessage->Encode-Message(...)` den Klartext in ein `BigInteger` Array zu konvertieren, das in `SplittedPlainText` abgelegt wird. Als Eingansparameter erwartet sich `ECDSInternalMessage->EncodeMessage(...)` ein Objekt vom Typ `Big-Integer`, das bei dieser Anwendung gleich der Charakteristik von $\mathbb{F}_p$ ist. Dadurch ist die Blocklänge der Nachricht festgelegt. Anschließend wird der string `PlainText` in eine `byte`-Liste kopiert. Damit die Länge dieser `byte`-Liste ein Vielfaches der Blocklänge ist, werden gegebenenfalls am Ende `byte.MinValue` Größen eingefügt. Zum Schluß werden die `BigInteger` Ele-mente aus `SplittedPlainText` erzeugt. Die `BigInteger` Klasse implementiert dazu die Methode `BigInteger->FromArray(...)`, die eben aus einem `byte`-Array eine `BigInteger`-Instanz erzeugt. Die statische Klasse `ECDSExe-cution` beinhaltet alle Algorithmen zur Ver- und Entschlüsselung. Verschlüsselt an sich wird über die Methode `ECDSExecution->Encrypt(...)`. Diese funktioniert nach dem oben beschriebenen Muster und retourniert ein Objekt vom Typ `ECDSCipher`, welches das entstandene Schiffrat darstellt. Für den zu übertragenden Punkt (x_1, y_1) ist es ausreichend nur x_1 und das least significant Bit von y_1 zu senden. Zu diesem Zweck enthält `ECDSCipher` einen Member vom Typ `CompAffinePoint` damit nicht die komplette Y-Koordinate übertragen werden muss.

```
public static ECDSCipher Encrypt(ECDSPublicKey pk,
                                 ECDSInternalMessage msg,
                                 BigInteger[] kValues) {
   ECDSSystemParameter sp = pk.ComTupel;
   AffinePoint P = sp.InitPoint.GetAffineRepresentation(sp.
       Curve),
               Q = pk.Q.GetAffineRepresentation(sp.Curve),
               R, S;
   msg.EncodeMessage(sp.CurveOrder);
   ECDSCipher cip = new ECDSCipher(sp.Curve.N);
   BigInteger c1;
   for (int i = 0; i < msg.SplittedPlainText.Length; i++)  {
        R = sp.Curve.Mult(P, kValues[i]);
        S = sp.Curve.Mult(Q, kValues[i]);
        c1 = (S.XCoordinate * msg.SplittedPlainText[i])
                                          % sp.Curve.N;
        cip.Blocks.Add(new ECDSCipher.CipherBlock(
                                  R, c1, sp.Curve.N));
   }
   return cip;
}
```

Agnes erhält Brunos Nachricht und möchte diese entschlüsseln. Dazu geht sie wie folgt vor.

1. Zuerst berechnet Agnes über ihren privaten Schlüssel d die Punkte

$$[d](x_{i_1}, y_{i_1}) = [k_i]Q = (x_{i_2}, y_{i_2})$$

2. Zu jedem erhaltenen x_{i_2} bildet sie das inverse Element $x_{i_2}^{-1} \in \mathbb{F}_p$.

3. Agnes kann über $c_i x_{i_2}^{-1} \equiv m_i$ Modulo p die Blöcke m_i ausrechnen.

4. Durch Zusammenfügen aller m_i kann Agnes die gesamte von Bruno gesendete Nachricht bilden.

Für die Entschlüsselung einer Nachricht implementiert die Klasse `ECDSParticipant` die Methode `ECDSParticipant->Decrypt(...)`. Diese ruft die statische Methode `ECDSExecution->Decrypt(...)` auf. Als Eingangsparameter werden hier ein `ECDSCipher` Objekt, der private Schlüssel und die korrespondierenden Systemparameter erwartet. Ähnlich wie beim Verschlüsseln wird wieder eine Instanz von `ECDSInternalMessage` angelegt. Diesmal übergibt man dem Konstruktor die Anzahl der Nachrichtenblöcke, damit das Array `SplittedPlainText` initialisiert werden kann. Anschließend kann nach dem oben beschriebenen Verfahren entschlüsselt werden.

```
public static ECDSInternalMessage Decrypt(ECDSCipher cip,
                                          BigInteger privateKey,
                                          ECDSSystemParameter sp) {
    AffinePoint A;
    ECDSInternalMessage msg =
                new ECDSInternalMessage(cip.Blocks.Count);
    BigInteger invX;
    int i = 0;
    foreach (ECDSCipher.CipherBlock currBlock in cip.Blocks)
         {
        A = sp.Curve.Mult(currBlock.R.
            GetAffineRepresentation(
                          sp.Curve),
                          privateKey);
        invX = A.XCoordinate.nummodInverse(sp.Curve.N);
        msg.SplittedPlainText[i++] =
                  (currBlock.CipherImage1 * invX) % sp.Curve.
                      N;
    }
    return msg;
}
```

Für Bruno ist es wichtig, dass er immer unterschiedliche Werte für k_i wählt. Angenommen er nimmt darauf keine Rücksicht und sendet zwei Tupel,

$$(x_1, y_1, c_1),\ (x_2, y_2, c_2) \text{ mit } (x_1, y_1) = (x_2, y_2)$$

die mit demselben k konstruiert wurden. Es gilt dann

$$c_1 = m_1\overline{x},\ c_2 = m_2\overline{x},\ [k]Q = (\overline{x}, \overline{y}).$$

Durch Bildung von $m_1\overline{x} * (m_2\overline{x})^{-1} = m_1 * m_2^{-1}$ kann jemand der m_1 kennt m_2 ausrechnen.

5.3 Die digitale Signatur (ECDSA)

ECDSA (Elliptic Curve Digital Signature Algorithm) bezeichnet ein Verfahren basierend auf elliptischen Kurven, zur digitalen Signatur von Nachrichten. Ich möchte den Prozess wieder anhand eines Szenarios mit den Akteuren Agnes und Bruno durchspielen. Agnes hat also das Bedürfnis Bruno eine digital signierte Nachricht m zu schicken. T und Q bezeichnen analog zu vorhin die Systemparameter und den öffentlichen Schlüssel von Agnes. Der Ablauf der digitalen Signatur erfolgt in den folgenden Schritten.

1. Agnes wählt eine Zufallszahl $k \in \mathbb{N} \cap [1, n-1]$.

2. Anschließend berechnet sie den Punkt $[k]P = (x_1, y_1)$ und bildet $r \equiv x_1 \bmod n$. Falls $r \equiv 0 \bmod n$ gilt, dann muss zu Schritt 1 zurückgekehrt und ein neues k ausgesucht werden.

3. Jetzt wird der Wert von $k^{-1} \bmod n$ gebildet. Gilt $ggT(k, n) \neq 1$ so existiert k^{-1} nicht und es muss wieder mit Schritt 1 neu begonnen werden.

4. Agnes bestimmt nun den Hashwert der Nachricht $h(m)$. Als Hashfunktion h wird meistens der SHA-1 Algorithmus verwendet. Der erhaltene Hashwert wird in eine natürliche Zahl e konvertiert.

5. Sie berechnet die Zahl $s = k^{-1}(e + dr) \bmod n$. Wenn $s = 0$ oder $s \in \mathbb{Z}/n\mathbb{Z}$ nicht invertierbar ist, dann ist wieder mit Schritt 1 weiterzumachen.

6. Die digitale Signatur der Nachricht m ist dann durch das Paar (r, s) gegeben.

Digitale Signaturen werden in `EllipticCurve` durch die Klasse `ECDSASignature` repräsentiert. Diese enthält zwei `BigInteger` Member r und s, welche die digitale Signatur bilden. Der Algorithmus zur digitalen Signatur ist in der statischen Klasse `ECDSAExecution` in der Methode `ECDSAExecution->Signate(...)` implementiert. Als Eingangsparameter werden die Nachricht m, die Systemparameter T und der private Schlüssel des Senders erwartet. Aufgerufen wird diese Methode nur indirekt von Instanzierungen von `ECDSParticipant` über die Methode `ECDSParticipant->Signate(...)`. Wie das folgende Codebeispiel zeigt, muss $n \in \mathbb{P}$ nicht notwendigerweise erfüllt sein.

```
/// Creates a signature of the input plaintext
/// by using the additional input parameter data
public static ECDSASignature Signate(string plainText,
                                     ECDSSystemParameter sp,
                                     BigInteger privateKey) {
   BigInteger r, s, k, invk = null;
   // get hashvalue of plaintext
   BigInteger hashValue = new SHA1().GetHashValue(plainText);

   AffinePoint kP;
   do {
     // get appropriate k
     do {
        // get random k out of [0,n-1]
        k = BigInteger.GetRandom(sp.CurveOrder);
        // get multiple of init point P
        kP = sp.Curve.Mult(sp.InitPoint.
            GetAffineRepresentation(
                          sp.Curve), k);
        r = kP.XCoordinate % sp.CurveOrder;
     } while (r == BigInteger.Zero ||
              !TryInvert(k, sp.CurveOrder,
              out invk)); // check if k is invertible

     // calculate s = invk(hashValue + privateKey*r) mod n
     s = (hashValue + privateKey * r) % sp.CurveOrder;
     s = (s * invk) % sp.CurveOrder;
    } while (! TryInvert(s, sp.CurveOrder));
           // if s is not invertible one has to take another
               k

    // the signature was created successfully
    return new ECDSASignature(r, s);
}
```

Bruno erhält die Nachricht m gemeinsam mit der Signatur (r, s). Aus m kann er entnehmen, dass Agnes der Absender ist. Um die Richtigkeit seiner Vermutung zu bestätigen, geht er folgendermaßen vor.

1. Bruno überprüft ob r und s aus dem Intervall $[1, n-1]$ stammen. Ist das nicht der Fall dann wird die Signatur nicht akzeptiert.

2. Analog zu Agnes berechnet er $h(m)$ und die Zahl e.

3. Weitere durchzuführende Berechnungen sind

$$w = s^{-1} \bmod n,$$

$$u_1 \equiv ew \bmod n \text{ und } u_2 \equiv rw \bmod n$$

4. Bruno kalkuliert den Punkt $R = u_1P + u_2Q = (x_0, y_0)$. Falls $R = \mathcal{O}$ gilt, dann akzeptiert Bruno die Signatur nicht. Andernfalls berechnet er $v \equiv x_0 \bmod n$. Die Signatur wird nur akzeptiert, wenn $v = r$ gilt.

Die Logik dieses Verfahrens ist wieder in der statischen Klasse `ECDSAExecution` in der Methode `ECDSAExecution->VerifySignature(...)` umgesetzt. `ECDSParticipant` Instanzen greifen über `ECDSParticipant->VerifySignature(...)` auf `ECDSAExecution->VerifySignature(...)` zu. Die Eingansparameter zur Verifikation sind ein Objekt vom Typ `ECDSASignature`, der Klartext der empfangenen Nachricht und der öffentliche Schlüssel des vermuteten Senders.

```
1  public static bool VerifySiganture(ECDSASignature signature,
2                                     string plainText,
3                                     ECDSPublicKey pukOfSender)
                                          {
4      ECDSSystemParameter sp = pukOfSender.ComTupel;
5      if (!signature.VerifySize(sp.CurveOrder)) return false;
6      BigInteger hashValue = new SHA1().GetHashValue(plainText)
           ;
7      BigInteger w, u1, u2;
8      if(!TryInvert(signature.S, sp.CurveOrder, out w)) return
           false;
9      u1 = (hashValue * w) % sp.CurveOrder;
10     u2 = (signature.R * w) % sp.CurveOrder;
11     AffinePoint R = sp.Curve.Mult(
12                     sp.InitPoint.GetAffineRepresentation(
13                     sp.Curve), u1);
14     R = sp.Curve.Add(R, sp.Curve.Mult(
15                     pukOfSender.Q.GetAffineRepresentation(
16                     sp.Curve), u2));
17     if (R.IsPointAtInfinity()) return false;
18     return (R.XCoordinate % sp.CurveOrder) == signature.R;
19 }
```

Für die Fälschung einer digitalen Signatur benötigt man den privaten Schlüssel des Absenders, der allerdings geheim ist. Zur Berechnung desselben ist man dann gezwungen ein ECDLP zu lösen, womit die Sicherheit des Verfahrens gewährleistet ist. Auch bei der digitalen Signatur muss darauf geachtet werden, dass immer unterschiedliche k gewählt werden. Angenommen Agnes signiert zweimal mit demselben k, dann gilt

$$ks_1 \equiv e_1 + dr \bmod n \tag{5.3}$$

und

$$ks_2 \equiv e_2 + dr \bmod n. \tag{5.4}$$

Werden die beiden Kongruenzen voneinander subtrahiert, so erhält man

$$k(s_1 - s_2) \equiv e_1 - e_2 \bmod n,$$

woraus man den Wert von k berechnen kann, wenn man die Hashwerte e_1 und e_2 kennt. Aus k erhält man dann r, womit man (5.4) oder (5.3) nach d auflösen kann. Damit ist eine theoretische Möglichkeit gegeben den privaten Schlüssel von Agnes zu berechnen.

Mit sogenannten *Index-Calculus-Methoden* kann man in subexponentieller Zeit ein allgemeines DL-Problem lösen. Eine Anwendung dieser Methode auf ECDLP ist nicht möglich. Auch für das Faktorisierungsproblem, das die Sicherheit beim RSA gewährleistet, kennt man subexponentielle Methoden. Wegen diesen Tatsachen wird das ECDLP sicherer als das Faktorisierungsproblem und das allgemeine DL-Problem betrachtet. Für die Praxis bedeutet dies, dass mit einer kleineren Schlüssellänge als beim RSA Verfahren die gleiche Sicherheit geboten werden kann. Beispielsweise ist ein RSA Verfahren mit einer Schlüssellänge von 512 Bits mit einem ECC mit 106 Bits gleichzusetzen. Aufgrund der niedrigen Schlüssellänge ist ECC prädestiniert für den Einsatz auf Chipkarten.

Kapitel 6

Überlange Zahlen in C#

Die in den vorangehenden Kapiteln implementierten Algorithmen für elliptische Kurven verwenden durchgehend den Datentyp `BigInteger`. Dieser Abschnitt soll die Klassen der `BigInteger`-API skizzieren und deren Funktionalität dokumentieren. Die Klassenbibliothek implementiert einen Ganzzahlentyp, dessen maximale Stellenanzahl softwareseitig unbeschränkt ist. Zusätzlich werden arithmetische Operationen und zahlentheoretische Funktionen für diesen Typ zur Verfügung gestellt. Bei der Entwicklung standen Performance, Übersichtlichkeit und Wiederverwendbarkeit in Zusammenhang mit algebraischen Strukturen im Vordergrund. Als Vorlage für die Klasse `BigInteger`, die gewissermaßen das Kernstück der API darstellt, diente mir eine aus dem Mono-Code-Projekt übernommene Klasse. Ich habe darin einige Fehler ausgebessert und die Funktionalität erweitert, sodass beispielsweise ein Rechnen mit negativen Zahlen möglich ist. Die Klasse `Kernel`, sowie die Idee, diese verschachtelt in `BigInteger` einzusetzen, stammt von den Mono-Code-Projekt Entwicklern Ben Maurer, Chew Keong Tan und Sebastien Pouliot.

6.1 Die Datenstruktur BigInteger

Der ganzzahlige Datentyp wird durch die Klasse `BigInteger` repräsentiert. Diese beinhaltet als Member ein `uint` - Array. Eine Instanz von `BigInteger` kann folglich als eine Liste von natürlichen Zahlen aufgefasst werden, die jeweils aus dem Intervall $[0, 2^{32} - 1]$ stammen. Mit dieser Datenstruktur lässt sich jedes (etwaige Einschränkungen durch die beschränkte Speicherkapazität werden vernachlässigt) $n \in N$ eindeutig darstellen, indem das `Uint32` - Array als Darstellung von n bezüglich der Basis 2^{32} interpretiert wird. Für eine

Instanz *number* von BigInteger, welche der natürlichen Zahl n entspricht, gilt dann

$$n = \sum_{i=0}^{l-1} number[i] * b^i \quad \text{mit}$$

$$b = Symbol.Uint32.MaxValue = 2^{32} - 1,$$
$$number[i] \in \{u \in N | 0 \leq u < b\} \text{ und}$$
$$l = min(k \in N | b^k > n).$$

Die Wahl der Zahlendarstellung bezüglich der Basis 2^{32}, ermöglicht es, bei der Implementierung von arithmetischen Funktionen für `BigInteger` (z.B.: ganzzahlige Addition) auf die bereits vorimplementierten elementaren Operationen von `Symbol.Uint32` zurückzugreifen. Weiters erweist sich die Konvertierung einer solchen Zahl in ein byte-Array als unkompliziert. Man muss nur die verwendeten `Symbol.Uint32` Größen als 4-byte Blöcke auffassen und in der richtigen Reihenfolge zu einem byte-Array zusammenfügen. Von der Methode `BigInteger->GetBits()` wird eine Instanz von `System.Collections.BitArray`, welche die Bitfolge der aktuellen Zahl beinhaltet, retourniert. Diese kann in Zusammenhang mit Divisionsalgorithmen und Potenzfunktionen für überlange Zahlen verwendet werden. Um auch negative Zahlen darstellen zu können, wird der Klasse die bool Eigenschaft `Signum` zugeteilt, welche durch ihren Wert das Vorzeichen der Zahl festlegt. Bevor ich auf das Design und die Richtlinien für eine performante Implementierung eingehe, wird eine strukturelle Beschreibung der API angeführt.

6.2 Die Komponenten der BigInteger API

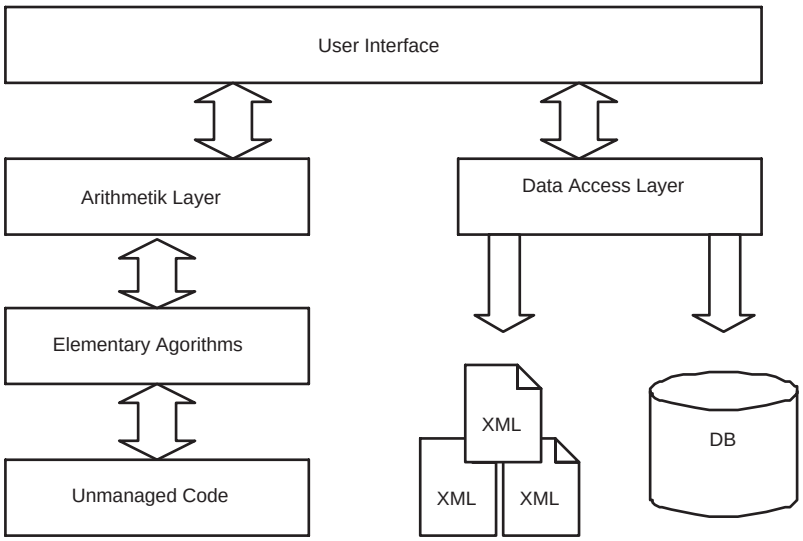
Die `BigInteger`-API ist eine auf dem .NET Framework aufgesetzte Klassenbibliothek und verwedet keine externen Softwarekomponenten. Um in gewissen Anwendungsfällen eine gesteigerte Performance zu erzielen, kann die Klassenbibliothek mit einer Datenquelle (z.B.: XML-Files, Datenbank) betrieben werden. In diesen Datenquellen werden bekannte Primzahlen abgelegt.

Die *User Interface* Komponente enthält alle Klassendefinitionen, die für das

Arbeiten mit `BigInteger` Instanzen notwedig sind. Somit können ganze Zahlen beliebiger Länge abgebildet und zu arithmetischen Operationen herangezogen werden. An dieser Schnittstelle werden auch noch einige zahlentheoretische Funktionen und Anwendungen zur Verfügung gestellt.

Beim Call einer elmentaren arithmetischen Operation, dies kann direkt über einen dafür vorgesehenen Operator oder indirekt über eine zahlentheoretische Funktion geschehen, übernimmt der *ArithmetikLayer* die Kontrolle des Programmflusses. Dieser bildet eine Schicht zwischen dem User Interface und den *Elementary Algorithm*, welche sämtliche elementaren Operationen implementieren. Das Ergebnis einer arithmetischen Operation ergibt sich aus dem Zusammenwirken von *ArithmetikLayer* und *Elementary Algorithm*.

Beim Aufruf von Funktionen aus dem *User Interface*, die Datenquellen verwenden, wird die Anfrage der Daten an den DataAccessLayer weitergegeben. Dieser verwendet für den Datenzugriff abstrakte Klassen, welche die notwendigen Eigenschaften und Funktionalitäten von Datenquellen definieren. Somit können als Datenquelle Datenbanken oder strukturierte Dateien (z.B.: XML) verwendet werden.



Aus Performancegründen ist es nicht empfehlenswert die Schichten *User Interface*, *Arithmetik Layer* und *Elementary Algorithm* auch softwaremäßig zu trennen. Die Klasse `BigInteger` bietet beispielsweise die Modulo Operation über den Operator % an. Bei dem Aufruf von

```
1 BigInteger a;
2 BigInteger b = new BigInteger(7);
3 BigInteger c = new BigInteger(5);
4 a = b % c;
```

wird die statische Klasse `BigInteger.Kernel`, die verschachtelt in `BigInteger` implementiert ist, benötigt. Diese berechnet über die Methode `BigInteger.Kernel->multiByteDivide(...)` das Ergebnis der modularen Reduktion. Diese Vorgangsweise gilt für nahezu alle elementaren arithmetischen Operationen. Daraus erkennt man aber, dass sich das Design der Schichten im Code nicht notwendigerweise widerspiegelt. Bei einigen Methoden von `BigInteger.Kernel` wird unsafe Code verwendet. Da man dann nicht typsicher arbeiten muss ist eine performante Umsetzung der Algorithmen möglich [vgl. Mono Code Project: BigInteger.cs].

Für die zahlentheoretischen Algorithmen existieren die hier aufgelisteten Klassen.

1. **CRTSolver** Damit kann ein System linearer Kongruenzen nach dem Prinzip des chinesischen Restsatzes aufgelöst werden.

2. **CornacchiaSmith** Diese Klasse dient zur Berechnung von (u, v) mit

$$4p = u^2 + v^2|D|$$

   wobei $p \in \mathbb{P}$ gilt und D eine Fundamentaldiskriminante ist.

3. **NumbertheoryUtilities** Hierbei handelt es sich um eine Ansammlung benötigter Algorithmen. Vorwiegend enthält diese Klasse verschiedene Primzahltests (Fermat-Test, Rabin-Miller-Test, Solovay-Strassen-Test).

Der Zugriff auf den Primzahldatenspeicher erfolgt über Instanzierungen von `PrimenumberInformationCenter`, die mit verschiedenen Datenspeichern umgehen können. Im Rahmen dieser Diplomarbeit wurden die Primzahlen in einer XML-Dateistruktur abgelegt. Möchte jemand die Primzahlen aus einer Datenbank einlesen, so hat er für den Zugriff die abstrakte Klasse `PrimenumberDataSource` zu implementieren. Anschließend muss nur noch in der statischen Klasse `DAL` (Data Access Layer) die Datenquelle registriert werden. Über die Methode `DAL->SetPrimeNumberDataSource(...)` kann dieselbe aktiviert werden.

In der `BigInteger`-API habe ich noch Klassen zur Bereitstellung von Polynomarithmetik in $\mathbb{F}_p[x]$ und $\mathbb{F}_p[x][y]$ implementiert. Prinzipiell habe ich zwei Ansätze zur Darstellung von Polynomen verfolgt.

Zum einen bilden die folgenden Klassen eine objektorientierte Lösung.

1. **(Monom)** Instanzen dieser Klasse stellen Ausdrücke der Form $ax^n \in \mathbb{F}_p[x]$ dar.

2. **(Dim2Monom)** Diese Klasse erweitert `Monom` zu Ausdrücken der Gestalt $ax^iy^j \in \mathbb{F}_p[x][y]$.

3. **(PolynomBase)** `PolynomBase` ist die gemeinsame Basisklasse von `Polynom` und `Dim2Polynom`. Sie stellt Methoden zur Verwaltung (z.B.: `PolynomBase->ToArray()`, `PolynomBase->FromArray(...)`) und für grundlegende arithmetische Hilfsoperationen (z.B.: `PolynomBase->RemoveZeroMonoms()`, `PolynomBase->ReduceMod(...)`) zur Verfügung. Wichtigster Member ist eine generische Liste, die Elemente des Typs `Monom` enthält.

4. **(Polynom)** Hierbei handelt es sich um eine Erweiterung von `PolynomBase`. Objekte vom Typ `Polynom` stellen Polynome in einer Veränderlichen aus $\mathbb{F}_p[x]$ dar. Sie können beispielsweise über `Polynom.Diff()` differenziert werden und durch Anwendung des Horner-Schemas evaluiert werden.

5. **(Dim2Polynom)** `Dim2Polynom` erbt ebenfalls von `PolynomBase`. Die Funktionalität dieser Klasse ist ähnlich derer von `Polynom`. Alle Methoden sind nur auf die Verarbeitung von Polynomen aus $\mathbb{F}_p[x][y]$ ausgelegt.

6. **(PolynomKernel)** Diese statische Klasse stellt die Algorithmen zur Multiplikation, Addition und Subtraktion in $\mathbb{F}_p[x]$ und in $\mathbb{F}_p[x][y]$ bereit. Eingangsparameter dieser Methoden sind Objekte vom Typ `Polynom` beziehungsweise `Dim2Polynom`. Die Methode `PolynomKernel->GCD(...)` berechnet den größten gemeinsamen Teiler von zwei durch `Polynom` repräsentierte Polynome.

Der Vorteil dieser objektorientierten Implementierung ist der sparsame Umgang mit dem Speicherplatz. Da das Polynom aus einer Liste von Monomen besteht, allokiert man nur so viel Speicher wie auch tatsächlich benötigt wird.

Der zweite Lösungsansatz besteht einfach darin die Koeffizienten in einem `BigInteger`-Array abzulegen. Über die Indizes eines Arrayelementes werden die Koeffizienten den Potenzen zugeordnet. Das `BigInteger`-Objekt an der Stelle i ist der Koeffizient von x^i. Allerdings müssen hier Koeffizienten mit dem Wert $0 \in \mathbb{F}_p$ ebenfalls abgespeichert werden. Zur Durchführung arithmetischer Operationen mit dieser Datenstruktur gibt es die statische Klasse `PolynomArrayArithmetic`. Diese implementiert Methoden für die gängigen Rechenoperationen in $\mathbb{F}_p[x]$. Zusätzlich gibt es auch die Methoden `PolynomArrayArithmetic->MultiplyKaratsuba(...)`, die zwei Polynome nach der Methode von Karatsuba multipliziert, und `PolynomArrayArithmetic->GetOutOfRoots(...)`, die ein Polynom über seine Nullstellenmenge erzeugt. Diese Verfahren werden in Verbindung mit der Montgomery-Continuation eingesetzt [vgl.: Kapitel 2].

Ein Vergleich der beiden Ansätze wurde schon in Kapitel 3 vorgenommen, wo die Klassen zur Darstellung von Punkten auf elliptischen Kurven verwendet wurden.

Damit wurde der Aufbau und die Funktionsweise der `BigInteger`-API umrissen. Zum Abschluß dieser Diplomarbeit werde ich den Einsatz der API in der Praxis demonstrieren.

In der Klasse `NumbertheoryUtilities` wird die Methode `Numbertheory-Utilities->NextPrime()` angeboten. Diese erwartet sich als Eingangsparameter eine durch `BigInteger` repräsentierte natürliche Zahl und berechnet die nächste darauffolgende Primzahl. Bei dem Verfahren werden die nicht primen Zahlen durch Probedivision und Rabin Miller Test zur Basis 2 aussortiert. Es werden allerdings von Grund auf nur jene Zahlen betrachtet, die in einer der Restklassen Modulo 30 mit den Repräsentanten $\{1, 7, 11, 13, 17, 19, 23, 29\}$ liegen. So kann die auf 10^{1000} folgende Primzahl $10^{1000} + 453$ in 40.640 Sekunden gefunden werden. Die nächste Primzahl $10^{1000} + 1357$ kann von $10^{1000} + 453$ ausgehend in 1 Minuten und 13.875 Sekunden berechnet werden. Ein Rabin-Miller-Test angewendet auf die Zahl $10^{1000} + 1357$ bezüglich den Basen 2,3,5 benötigt 4.781 Sekunden bevor er die Zahl als relativ prim erkennt.

Kapitel 7

Literaturverzeichnis

[Berger] D. BERGER: ECM - Faktorisieren mit elliptischen Kurven; Diplomarbeit (1993).

[Brent] R. BRENT: Some Integer Factorization Algorithms using Elliptic Curves; Artikel (1985).

[Cohen] H. Cohen: A Course in Computational Algebraic Number Theory; Springer-Verlag 2000.

[C & P] R. CANDALL, C. POMERANCE: Prime Numbers - A computational Perspective; Springer-Verlag New York (2001).

[Knuth] D. E. KNUTH: The Art of Computer Programming - Volume 2 / Seminumerical Algorithms; Addison-Wesley (1969).

[Mirbach] A. MIRBACH: Elliptische Kurven - Die Bestimmung ihrer Punktezahl und Anwendungen in der Krptographie; Verlagshaus Monsenstein und Vannerdat (2003).

[Mont] P. L. MONTGOMERY: An FFT Extension of the Elliptic Curve Method of Factorization; Dissertation (1992).

[Werner] A. WERNER: Elliptische Kurven in der Kryptographie; Springer-Verlag, Berlin Heidelberg New York (2002).

[Zulfaj] H. ZULFAJ: Arithmetik in endlichen Körpern und ihre Anwendung in der Kryptographie; Diplomarbeit.