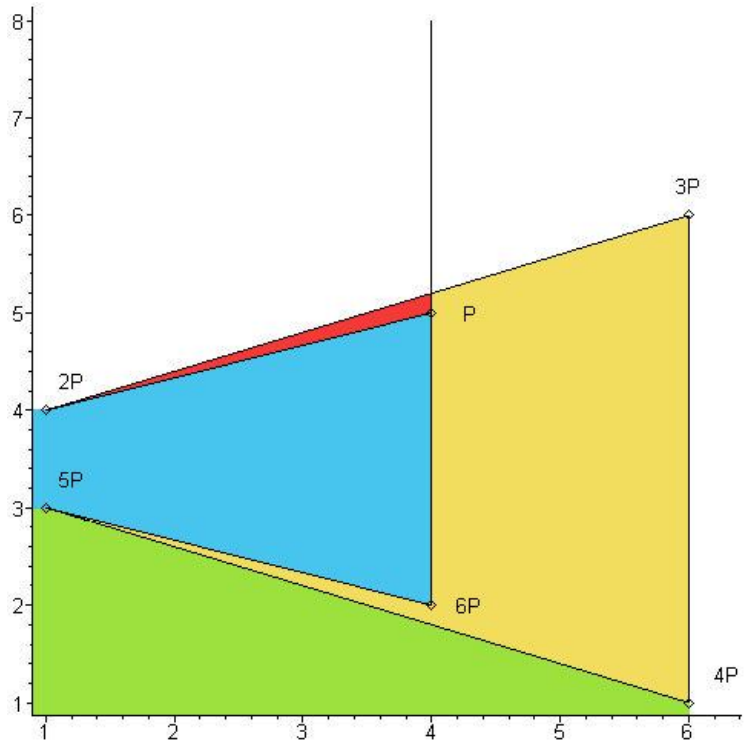
Implementierung und Evaluation einiger Algorithmen für Kryptosysteme auf elliptischen Kurven

Michael Gorski



Studienarbeit

Betreuer
Prof. Dr. Johannes Blömer
Dipl. Inf. Dipl. Math. Martin Otto

Danksagung

Ich möchte mich an dieser Stelle bei Prof. Blömer bedanken, der mir die Möglichkeit verschafft hat meine Studienarbeit in einem Thema zu schreiben, welches sehr spannend ist und die Basis für viele interessante Aspekte der Kryptographie bereitstellt. Martin Otto half mir in vielen Gesprächen, eine konkretere Sichtweise über das Thema zu bekommen. Durch die große Fülle an Literatur, die mir durch Herrn Blömer und Herrn Otto zur Verfügung gestellt wurde, war es schnell möglich mich in das Thema einzuarbeiten.

Erklärung

Hiermit versichere ich, die vorliegende Studienarbeit selbstständig verfasst und keine anderen Hilfsmittel als die angegebenen Quellen verwendet zu haben.
Paderborn, den 23.06.2004

Inhaltsverzeichnis

Einleitung

Die Kryptographie beschäftigt sich mit der Ver- und Entschlüsselung von Nachrichten. Zu diesem Zwecke entstanden zwei verschiedene Typen von Verschlüsselungsverfahren. Man unterscheidet zwischen symmetrischen und asymmetrischen Verfahren. In symmetrischen Verschlüsselungsverfahren wird zur Ver- und Entschlüsselung ein und derselbe Schlüssel verwendet. Solche Verfahren bieten den Vorteil, dass man mit ihnen schnell ver- und entschlüsseln kann. Der Schlüsselaustausch ist bei diesem Verfahren jedoch sehr aufwändig, da man ihn über einen abhörsicheren Kanal übertragen muss.

In der Mitte der 70er Jahre wurde von Diffie und Hellman als erstes ein asymmetrisches Verfahren zum Schlüsselaustausch entwickelt. Mit diesem ist möglich, dass sich zwei Gesprächspartner auf einen gemeinsamen geheimen Schlüssel einigen und dies über eine öffentlich zugänglichen Kanal durchführen können. Man spricht hierbei auch von Public-Key-Kryptographie, da es in solchen Verfahren immer einen öffentlichen und einen privaten Schlüssel gibt und alle Informationen über einen öffentlichen Kanal transportiert werden. Mit Verfahren der Public-Key-Kryptographie ist es möglich Nachrichten auszutauschen, ohne vorher einen geheimen Schlüssel über einen sicheren Kanal austauschen zu müssen. Alle asymmetrischen Verfahren basieren auf schwer lösbaren mathematischen Problemen.

Das wohl bekannteste asymmetrische Verschlüsselungsverfahren ist RSA, es basiert auf dem Faktorisierungsproblem. Dabei ist einen so genanntes RSA Modul n gegeben, welches ein Produkt aus zwei Primzahlen p und q ist. Dieses RSA Modul ist Teil des öffentlichen Schlüssels dieses Verfahrens. Wenn man die Primzahlen p und q groß genug gewählt, dann nimmt man an, dass die Zerlegung von n in seinem Primteiler sehr schwer ist. Gelingt es n zu faktorisieren, so ist dieses Kryptosystem gebrochen und damit nicht mehr sicher. Das Faktorisierungsproblem wäre damit gelöst.

Ein zweites schwieriges mathematisches Problem ist das so genannte diskrete Logarithmus Problem. Man nimmt als Basis eine endliche abelsche Gruppe G. Sei $P \in G$ und n die Ordnung der von P erzeugten zyklischen Untergruppe

$$< P > = \{kP \mid k \in \mathbb{Z}\}$$

von G. Damit ist n die kleinste natürliche Zahl mit $nP = 0$, wobei 0 das neutralelement von G ist. Sei $Q \in < P >$. Gesucht ist ein $k \in \mathbb{Z}$ mit

$$Q = kP.$$

Es wird angenommen, dass es schwer ist ein solches k zu finden. Kryptographische Ver-

fahren auf elliptischen Kurven basieren auf diesem Problem.

Elliptische Kurven wurden etwa 1985 für die Kryptographie entdeckt und werden heutzutage immer mehr als Basis für Kryptosysteme benutzt. Miller [Mil86] und Koblitz [Kob87] stellen diese als Alternative zu abgeschlossenen Körpern dar. Elliptische Kurven bieten den Vorteil, dass sie effizienter als RSA basierte Verfahren sind, da sie wesentlich kleinere Schlüssellängen benötigen, um die gleiche Sicherheit gewährleisten zu können. Nach heutigem Wissensstand ist es schwieriger den diskreten Logarithmus über elliptische Kurven zu berechnen, als beispielsweise über $\mathbb{Z}_p$ mit p prim. Die kürzeren Schlüssellängen und die erhöhte Schwierigkeit des diskreten Logarithmus sind die wichtigsten Gründe um elliptische Kurven in der Kryptographie einzusetzen. Die folgende Tabelle (aus [Mir03, S. 122]) verdeutlicht das Verhältnis der Schlüssellänge von RSA im Vergleich zu elliptischen Kurven Verfahren (ECC).

RSA Schlüssellängen (in Bit)	ECC Schlüssellängen (in Bit)	Verhältnis RSA/ECC Schlüssellängen
512	106	$4,8:1$
768	132	$5,8:1$
1024	160	$6,4:1$
2048	210	$9,8:1$

Die Sicherheit wird mit wachsender Schlüssellänge immer größer. Wir können deutlich erkennen, dass elliptische Kurven Verfahren bei steigender Sicherheit immer Vorteilhafter werden, da sie einen geringeren Anstieg der Schlüssellänge als RSA zu verzeichnen haben.

Die Arithmetik auf elliptischen Kurven ist zwar aufwändiger als die mit primen Restklassenringen, aber durch die geringere Länge der Zahlen erreicht man trotzdem einen Geschwindigkeitsvorteil gegenüber RSA Verfahren. Mit Hilfe von elliptischen Kurven ist es z. B. möglich Kryptographie auf Smart-Cards ohne Coprozessor zu implementieren, womit diese wesentlich billiger herzustellen werden.

Für die Kryptographie werden bevorzugt starke elliptische Kurven verwendet, da es sonst sehr leicht möglich seien kann erfolgreiche Angriffe gegen auf ihnen basierende Kryptosysteme durchführen zu können. Starke elliptische Kurven zeichnen sich durch folgende Merkmale aus:

- Die Gruppenordnung sollte eine Primzahl sein.
- Die Anzahl Kurvenpunkte sollte nicht gleich der Gruppenordnung sein.

Wenn man nun eine elliptische Kurve für ein Kryptosystem benutzt, welche keine prime Anzahl von Punkten hat, dann kann das System eventuell durch die Methode von Pohlig-Hellman gebrochen werden.

In der Kryptographie betrachtet man entweder elliptische Kurven über dem Grundkörper $\mathbb{F}_q$ mit $q = p^n$ und p prim oder über $\mathbb{F}_{2^n}$ mit $n \in \mathbb{N}$. Wir werden in dieser Studienarbeit ausschließlich elliptische Kurven über $\mathbb{F}_q$ mit einer Charakteristik größer als 3 betrachten. Meist gehen wir aber von der Form $\mathbb{F}_p$ mit p prim aus.

Es wird darum gehen Algorithmen für elliptische Kurven in geeigneter Weise zu implementieren. Dafür ist es notwendig die für die Darstellung erforderlichen Datenstrukturen zu entwickeln und die für elliptische Kurven bekannten Grundrechenoperationen zu implementieren. Der Hauptteil der Arbeit befasst sich mit drei Algorithmen, die dazu eingesetzt werden, um die Gruppenordnung (Anzahl der Punkte) einer elliptischen Kurve zu bestimmen. Dies ist wichtig um, wie oben angesprochen, starke elliptische Kurven finden zu können. Auch diese Algorithmen werden wir implementiert.

Im ersten Teil befassen wir uns mit den Grundlagen der Mathematik, die für die Betrachtung und die Diskussion von elliptischen Kurven relevant sind. Anschließend folgt in Kapitel 2 eine fundierte mathematische Einführung in die Thematik der elliptischen Kurven. Wir sprechen hierbei oft von elliptischen Kurven, das ist aber nicht ganz korrekt. Genauer müssten wir von der Punktmenge einer elliptischen Kurve sprechen, da eine solche Kurve mathematisch gesehen nur eine Funktionsgleichung darstellt. In der Kryptographie spricht man aber von elliptischen Kurven anstatt von Punktmengen über ihnen, deshalb behalten wir diesen sprachlichen Gebrauch bei. Wir machen deutlich, wie man elliptische Kurven in affiner und projektiver Form darstellen kann, wann wir von einer elliptischen Kurve sprechen und wie man auf ihnen rechnen kann. Dazu leiten wir graphisch die benötigten Operationen her. Das Kapitel 3 behandelt Algorithmen, mit denen man die Elementaroperationen auf elliptischen Kurven in sehr schneller Weise praktisch realisieren kann.

Der zweite Teil dieser Studienarbeit befasst sich mit der Bestimmung der Gruppenordnung von elliptischen Kurven, also mit dem Zählen der Punkte, welche auf einer betrachteten Kurve liegen. Kapitel 4 behandelt den naiven Algorithmus zum Punkte zählen. Dieser ist sehr einfach und auch sehr langsam. Anschließend diskutieren wir im Kapitel 5 den Algorithmus von Shanks und Mestre, er ist randomisiert und hat auch eine sehr schlechte Laufzeit, die aber deutlich schneller als die des in Kapitel 4 vorgestellten naiven Algorithmus ist. Den größten Teil dieser Arbeit widmen wir in Kapitel 6 dem Algorithmus von Schoof. Er ist der erste deterministische Algorithmus, der die Anzahl der Punkte in polynomieller Zeit berechnen kann. Häufig vorkommende Herleitungs- und Umformungsfehler der uns vorliegenden Literatur werden wir so beseitigen, dass eine schnelle Implementierung des Algorithmus leicht möglich wird.

Teil drei behandelt die von dieser Studienarbeit geforderte Implementierung der in Teil eins und zwei vorgestellten Algorithmen. Dieser Teil wird komplett von uns entworfen und ist in dieser Form nicht wieder zu finden. Wir haben für die Implementierung spezielle Datenstrukturen und Klassen entwickelt, welche zur Darstellung von elliptischen Kurven notwendig sind. Diese werden in Kapitel 7 ausführlich beschrieben. Als Programmiersprache verwenden wir Java. Das Kapitel 8 richtet sich mehr an den Endanwender, der auf der Basis der Implementation arbeiten möchte. Hierbei werden Ablaufbeispiele gegeben, welche den Anwender schnell in die Funktionsweisen der Klassen einführen sollen. Im Kapitel 9 geben wir eine Übersicht über den aktuellen Stand der Forschung. Wir zeigen chronologisch wie sich die Algorithmen zum bestimmen der Gruppenordnung entwickelt haben und wie schnell sie bis zum heutigen Stand sind. Das Kapitel 10 zeigt einen Geschwindigkeitsvergleich der von uns implementierten Algorithmen und gibt eine graphische Darstellungen der theoretischen Laufzeiten. Zum Schluß besprechen in Kapitel 11 wie die von uns implementierten Algorithmen noch entscheidend verbessert werden können, um die Laufzeit noch weiter zu verbessern. Dies ist allerdings nicht Aufgabe dieser Arbeit. Die Aufgabe ist es die Algorithmen überhaupt in Programme umzusetzen. Im Anhang A befindet sie der komplette von uns entworfene Quelltext der Klassen. Darauf folgt in Anhang B die ausführliche Java Dokumentation sämtlicher Klassen und Funktionen.

Teil I
Grundlagen

1 Mathematische Grundlagen

Es sollen hier zunächst grundlegende mathematische Begriffe erläutert werden, die wir immer wieder für verschiedene Algorithmen benötigen und die nicht jedem Leser geläufig sind.

Satz 1.1 ***Chinesischer Restsatz***
Seien $p_1, \ldots, p_t$ paarweise teilerfremde natürliche Zahlen und $x_1, \ldots, x_t$ beliebige ganze Zahlen. Dann gibt es eine ganze Zahl x mit

$$\begin{aligned} x &\equiv x_1 \bmod p_1, \\ x &\equiv x_2 \bmod p_2, \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ x &\equiv x_t \bmod p_t. \end{aligned}$$

Außerdem ist x modulo $p = p_1 p_2 \cdots p_t$ eindeutig bestimmt, das heißt wenn sowohl x und x' die obigen Kongruenzen erfüllen, so gilt

$$x \equiv x' \bmod p.$$

Die Zahl x lässt sich folgendermaßen berechnen: Setze

$$m_i = \frac{p}{p_i} = \prod_{\substack{j=1 \\ j \neq i}}^{t} p_j.$$

Man erhält für x

$$x = \sum_{i=1}^{t} x_i a_i m_i$$

mit $a_i m_i \equiv 1 \bmod p_i$.

1.1 Gruppen

Definition 1.2 *Eine Gruppe G ist eine Menge von Elementen zusammen mit einer Abbildung*

$$G \times G \to G$$

$$(a, b) \mapsto a \circ b$$

die folgende Eigenschaften erfüllt:

1. *$a \circ (b \circ c) = (a \circ b) \circ c$ für alle $a, b, c \in G$ (Assoziativgesetz).*
2. *Es gibt ein eindeutig bestimmtes Element $e \in G$ mit $a \circ e = e \circ a = a$ für alle $a \in G$ (neutrales Element).*
3. *Für jedes $a \in G$ existiert ein $b \in G$ mit $a \circ b = b \circ a = e$, dabei kann man auch a^{-1} für b schreiben.*

Wenn in G auch $a \circ b = b \circ a$ für alle $a, b \in G$ (Kommutativität) gilt, dann heißt G abelsche Gruppe.

Definition 1.3 *Sei G eine Gruppe, deren Verknüpfung wir additiv schreiben, mit endlich vielen Elementen. Dann nennt man die Anzahl der Elemente in G auch die Ordnung von G. Man schreibt $\#G$. Sei a ein Element der Gruppe G, dann nennt man die kleinste natürliche Zahl m mit $ma = 0$ die Ordnung von a.*

1.2 Ringe

Definition 1.4 *Ein Ring ist eine Menge R zusammen mit zwei Verknüpfungen*

$$(a, b) \longmapsto a + b \quad \textit{und} \quad (a, b) \longmapsto ab,$$

so dass folgende Axiome erfüllt sind:

1.) *$(R, +)$ ist eine abelsche Gruppe, deren neutrales Element wir mit 0 bezeichnen.*

2.) *Die Multiplikation ist assoziativ, d.h. für $a, b, c \in R$ gilt $a(bc) = (ab)c$.*

3.) *Es existiert ein neutrales Element 1 bezüglich der Multiplikation, d.h. für alle $a \in R$ ist $1a = a1 = a$.*

4.) *Es gelten die Distributivgesetze $a(b + c) = ab + ac$ und $(b + c)a = ba + ca$ für alle $a, b, c \in R$. Falls für alle $a, b \in R$ $ab = ba$ gilt, so heißt R kommutativer Ring.*

1.3 Körper

Definition 1.5 *Ein kommutativer Ring mit $1 \neq 0$ in dem jedes von Null verschiedene Element ein Inverses bezüglich der Multiplikation hat, heißt Körper.*

Definition 1.6 *Ein Körper F hat die Charakteristik 0, falls für alle natürlichen Zahlen $m \in \mathbb{N}$ mit $\mathbb{N} \neq \mathbb{N}_0$, dass Element*

$$m1 = \overbrace{1 + \cdots + 1}^{m \text{ mal}}$$

von Null verschieden ist. Falls es hingegen eine natürliche Zahl $m \in \mathbb{N}$ gibt, so daß $m1 = 0$ ist, so heißt die kleinste natürliche Zahl mit dieser Eigenschaft Charakteristik von F.

Lemma 1.7 *Sei $\mathbb{F}$ ein beliebiger Körper der Charakteristik $p > 0$. Dann gilt für $a, b \in \mathbb{F}$ und alle natürlichen Zahlen t:*

$$(a+b)^{p^t} = a^{p^t} + b^{p^t}.$$

Definition 1.8 *Ein endlicher Körper ist ein Körper mit endlich vielen Elementen. Ein Körper $\mathbb{F}$ heißt algebraisch abgeschlossen, wenn jedes Polynom mit Koeffizienten aus $\mathbb{F}$ über $\mathbb{F}$ ganz in Linearfaktoren zerfällt. Zu jedem Körper $\mathbb{F}$ existiert ein Oberkörper $\overline{\mathbb{F}}$, der algebraisch abgeschlossen ist. Alle algebraisch abgeschlossenen Oberkörper von $\mathbb{F}$ sind zueinander isomorph. Wir sprechen daher von dem algebraischen Abschluss von $\mathbb{F}$.*

Definition 1.9 *Sei $\mathbb{F}$ ein Körper. Ein Polynom über $\mathbb{F}$ in den n Variablen $x_1, \ldots, x_n$ ist ein Ausdruck der Form*

$$f(x_1, \ldots, x_n) = \sum_{k_1, \ldots, k_n \geq 0} \gamma_{k_1, \ldots, k_n} x_1^{k_1} \cdots x_n^{k_n}$$

mit Koeffizienten $\gamma_{k_1,\ldots,k_n} \in \mathbb{F}$, von denen nur endlich viele ungleich Null sind. Die Menge aller Polynome über $\mathbb{F}$ in $x_1, \ldots, x_n$ bezeichnen wir mit $\mathbb{F}[x_1, \ldots, x_n]$.

Satz 1.10 ***(Satz von Lagrange)*** *Sei x eine Element einer Gruppe G, dann teilt die Ordnung des Elementes x die Gruppenordnung $\#G$ von G.*

1.4 Das Legendre - Symbol

Der Ausdruck $\left(\frac{a}{p}\right)$ bezeichnet das Legendre - Symbol. Mit dessen Hilfe kann man überprüfen, ob eine ganze Zahl $a \in \mathbb{Z}_p$ ein Quadrat modulo einer Primzahl p ist, also ob es ein $x \in \mathbb{Z}_p$ gibt, für das $x^2 = a \bmod p$ gilt. Es können dabei die folgenden drei Werte auftreten.

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{wenn } ggT(a,p) \neq 1 \\ 1 & \text{wenn } a \text{ ein quadratischer Rest modulo } p \text{ ist, so dass } x^2 = a \bmod p \\ -1 & \text{wenn } a \text{ ein quadratischer Nichtrest modulo } p \text{ ist} \end{cases}$$

Der folgende Satz drückt wichtige Eigenschaften des Legendre - Symbols aus.

Satz 1.11 *Sei $\left(\frac{a}{p}\right)$ das Legendre - Symbol, dann gilt:*

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

2.) $\left(\frac{a\,b}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$

3.) Es gibt in $\mathbb{Z}_p^*$ genauso viele quadratische Reste wie quadratische Nichtreste und das sind jeweils $\frac{p-1}{2}$.

Beweis siehe [Blö00, S. 25].

Ein Algorithmus, der das Legendre - Symbol berechnet, ist der Algorithmus 1.1. Da wir das Legendre - Symbol eben ausführlich besprochen haben und die Ausführung des Algorithmus trivial ist, verzichten wir auf dessen Erläuterung.

Algorithmus 1.1 Legendre - Symbol

Eingabe: $a \in \mathbb{Z}_p$ mit p prim

Ausgabe: $\left(\frac{a}{p}\right) \in \{-1, 0, 1\}$

SCHRITT 1: **Falls** $a \equiv 0 \bmod p$ **Rückgabe:** 0

SCHRITT 2: **Setze** $e \leftarrow \frac{p-1}{2}$.

SCHRITT 3: **Rückgabe:** $a^e \bmod p$

Die Berechnung des Legendre - Symbols benötigt $O(\log e)$ Multiplikationen. Zum Abschuss der Betrachtung des Legendre - Symbols bleibt noch zu erwähnen, dass es dem Quadratischen Reziprozitätsgesetz genügt. Das ist eine wichtige Eigenschaft, besonders wenn man mit dem Legendre - Symbol rechnen will.

Satz 1.12 ***Quadratisches Reziprozitätsgesetz***
Für jede Primzahl $p \neq 2$ ist $\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}}$ und $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}$. Für zwei verschiedene Primzahlen p und q gilt

$$\left(\frac{p}{q}\right) = (-1)^{\frac{(p-1)(q-1)}{4}} \left(\frac{q}{p}\right).$$

Der Beweis dieses Satzes ist in fast jedem Buch über Zahlentheorie zu finden. Dieses Gesetz erlaubt es, die Rollen von p und q zu vertauschen. Man kann die Rechenregel

$$\left(\frac{q + mp}{p}\right) = \left(\frac{q}{p}\right)$$

benutzen, um zu errechnen, ob eine Zahl ein quadratischer Rest modulo p ist.

1.5 Die Quadratwurzel modulo einer Primzahl

Wenn man mit dem Legendre - Symbol herausgefunden hat, dass eine Zahl $a \in \mathbb{Z}_p$ ein Quadrat modulo einer Primzahl ist, weiß man immer noch nicht für welche Werte von $x \in \mathbb{Z}_p$ die Gleichung $x^2 = a \bmod p$ erfüllt ist. Dazu kann man den folgenden Algorithmus aus [Coh00, S. 32] benutzen.

Algorithmus 1.2 Wurzel mod p

Eingabe: $a \in \mathbb{Z}_p$, $p > 2$ prim, so dass $\left(\frac{a}{p}\right) = 1$

Ausgabe: $x \in \mathbb{Z}_p$ mit $x^2 = a \bmod p$

SCHRITT 1: **Falls** $p \equiv 3 \bmod 4$

Dann $k \leftarrow \frac{p-3}{4}$ und $x \leftarrow a^{k+1} \bmod p$ **Rückgabe:** x

SCHRITT 2: Wähle n zufällig, bis $\left(\frac{n}{p}\right) = -1$.

SCHRITT 3: **Setze** $left \leftarrow p - 1$.

SCHRITT 4: **Für** $i = 1, \ldots, \lceil \log_2(p-1) \rceil$

Falls $\frac{left}{2}$ ganzzahlig und ungerade

Dann $q \leftarrow \frac{left}{2}$, $e \leftarrow i$ und beende die Schleife.

Sonst Setze $left \leftarrow \frac{left}{2}$.

SCHRITT 5: **Setze** $y \leftarrow n^q$, $r \leftarrow e$, $x \leftarrow a^{\frac{(q-1)}{2}}$, $b \leftarrow ax^2$ und $x \leftarrow ax$.

SCHRITT 6: **Solange** $b \not\equiv 1 \bmod p$

Setze $m \leftarrow 0$.

Solange $b^{2^m} \not\equiv 1 \bmod p$

Setze $m \leftarrow m + 1$.

Setze $t \leftarrow y^{2^{r-m-1}}$, $y \leftarrow t^2$, $r \leftarrow m$, $x \leftarrow xt$ und $b \leftarrow by$.

SCHRITT 7: **Rückgabe:** x

Das Prinzip des Algorithmus 1.2 soll hier kurz beschrieben werden. Wir können zwei Zahlen $e, q \in \mathbb{Z}_p$ mit q ungerade finden, so dass die Gleichung

$$p - 1 = 2^e q$$

erfüllt ist. Die multiplikative Gruppe $\mathbb{Z}_p^*$ ist isomorph zu der additiven Gruppe $\mathbb{Z}_{p-1}$. Deshalb gibt es eine zyklische Untergruppe G der Ordnung 2^e. Angenommen man kann einen Generator $z \in G$ finden, dann sind die Quadrate der Untergruppe G Elemente der Ordnung geteilt durch 2^{e-1} und gerade Potenzen von z. Wenn a ein quadratischer Rest modulo p ist, dann ist

$$a^{\frac{(p-1)}{2}} = a^{\frac{(2^e q)}{2}} = (a^q)^{(2^{e-1})} = 1 \bmod p.$$

Sei außerdem $b = a^q \bmod p$ eine Quadrat in G. Damit existiert ein gerades k, wobei $k \in \{0, 1, \ldots, 2^e - 1\}$, so dass

$$a^q z^k = 1 \bmod G. \tag{1}$$

Wenn ein k die Gleichung (1) erfüllt, dann setzen wir

$$x = a^{\frac{(p-1)}{2}} z^{\frac{k}{2}} \bmod p.$$

Damit ist klar, dass x die gesuchte Lösung der Gleichung $x^2 = a \bmod p$ ist.

Da a gegeben ist und wir e und q leicht errechnen können, bleibt noch die Frage wie wir z und k berechnen können. Um z zu bestimmen wählen wir in Schritt 2 ein zufälliges n, welches ein quadratischer Nichtrest modulo p ist und setzen $z = n^q \bmod p$. Damit wissen wir, dass z ein Generator ist, da

$$z^{\frac{2^e}{2}} = z^{2^{e-1}} = -1 \bmod p$$

ergibt. Einen quadratischen Nichtrest für n findet man mit Wahrscheinlichkeit $\frac{1}{2}$, da es genauso viel quadratische Reste, wie quadratische Nichtreste gibt. Die Berechnung des korrekten Wertes für t können wir vollkommen deterministisch, wie in Schritt 6 ermitteln.

Der Algorithmus 1.2 ist randomisiert aufgrund der Tatsache, dass in Schritt 2 ein zufälliges n gewählt wird. Er hat insgesamt eine erwartete Laufzeit von $O(\ln^4 p)$.

2 Einführung in Elliptische Kurven

In diesem Kapitel sollen dem Leser die für elliptische Kurven relevanten Definitionen, Eigenschaften und Begriffe erläutert werden. Um elliptische Kurven zu verstehen, ist es notwendig, dass wir zuerst affine und projektive Kurven einführen. Anhand dieser die Thematik der elliptischen Kurven zu veranschaulichen. Wir werden hier nur das erläutern, was für das Verständnis dieser Kurven zwingend notwendig ist. Für eine tiefe Betrachtung verweisen wir auf das Buch [Wer02, S.11 ff], an dem wir uns in diesem Kapitel sehr stark orientieren werden.

Zunächst führen wir die affinen Kurven ein, die wir als Grundlage benötigen. Im Folgenden bezeichnen wir mit $\mathbb{F}$ einen beliebigen Körper.

Definition 2.1 *Sei die Menge* $\mathbb{A}^2(\mathbb{F}) = \{(a,b)|\ a,b \in \mathbb{F}\}$ *ein zweidimensionaler affiner Raum. Sei* f *ein Polynom mit den Variablen* $x, y \in \mathbb{F}$ *der Form*

$$f(x,y) = \sum_{i,j \geq 0} \gamma_{i,j} x^i y^j$$

mit Koeffizienten $\gamma_{i,j} \in \mathbb{F}$ *von denen nur endlich viele ungleich Null sind. Wir nehmen an, dass* $f \neq 0$. *Die Menge* $C_f(\mathbb{F}) = \{(a,b) \in \mathbb{A}^2(\mathbb{F})|\ f(a,b) = 0\}$ *heißt Nullstellenmenge von* f. *Jede Nullstellenmenge* $C_f(\mathbb{F})$ *nennt man affine ebene Kurve.*

Die Koordinaten $(a,b) \in \mathbb{A}^2(\mathbb{F})$ werden auch als affine Koordinaten bezeichnet.

Eine für elliptische Kurven wichtige Eigenschaft ist die Singularität, mit deren Hilfe man überprüfen kann, ob eine bestimmte Kurve eine elliptische Kurve ist.

Definition 2.2 *Eine affine ebene Kurve* $C_f(\mathbb{F})$ *heißt singulär im Punkt* $(a,b) \in C_f(\mathbb{F})$, *wenn die beiden partiellen Ableitungen von* f *verschwinden, so dass*

$$\frac{\partial f}{\partial x}(a,b) = 0 \ \ und \ \ \frac{\partial f}{\partial y}(a,b) = 0$$

ist. Wenn $C_f(\overline{\mathbb{F}})$ *in keinem Punkt singulär ist, dann heißt* $C_f(\mathbb{F})$ *nicht singulär. In diesem Fall werden in keinem Punkt gleichzeitig beide partielle Ableitungen von* f *zu Null.*

Mit diesem Wissen können wir die affine Form der elliptischen Kurven beschreiben.

Definition 2.3 *Sei* f *ein Polynom mit zwei Variablen* $x, y \in \mathbb{F}$ *der Form*

$$f(x,y) = y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6,$$

wobei $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}$ *sind. Eine elliptische Kurve* E *in affiner Form über dem Körper* $\mathbb{F}$ *ist eine Vereinigung der Menge von Punkten einer nicht singulären affinen ebenen Kurve*

$C_f(\mathbb{F})$ über dem Polynom f zusammen mit einem speziellen Punkt $\mathcal{O}$, der auch als Punkt im Unendlichen bezeichnet wird. Wir schreiben formal

$$E(f) := C_f(\mathbb{F}) \cup \{\mathcal{O}\}.$$

Mit $E(f)$ ist die Menge der elliptischen Kurven über einem Polynom f gemeint. Wir werden im Folgenden die Schreibweise $E(\mathbb{F})$ verwenden. Sie bezeichnet eine spezielle elliptische Kurve über dem Körper $\mathbb{F}$. Die Gleichung der affinen ebenen Kurve $C_f(\mathbb{F})$ lässt sich durch

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

schreiben. Diese Gleichung wird in der Literatur auch als allgemeine Weierstraßgleichung bezeichnet. Genauer gesagt als affine Form der allgemeinen Weierstraßgleichung. Wenn wir elliptische Kurven in der Kryptographie betrachten, meinen wir mit dem Begriff elliptische Kurve immer die Gruppe der elliptische Kurve. Diese Gruppe beinhaltet, wie zuvor definiert, neben den Punkten der Weierstraßgleichung zusätzlich den Punkt im Unendlichem $\mathcal{O}$.

Da wir uns uns hier aber nur mit elliptischen Kurven über dem Primkörper $\mathbb{F}_q$, $q = p^n$ und p prim mit einer Charakteristik größer als drei beschäftigen, fallen die Koeffizienten a_1, a_2 und a_3 der allgemeinen Weierstraßgleichung weg. Die Koeffizienten a_4 und a_6 nennen wir ab jetzt a und b. Daraus entsteht die vereinfachte Form der allgemeine Weierstraßgleichung

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

Die in der Definition 2.3 beschriebene Darstellung einer elliptischen Kurve vereinfacht sich dadurch. Wir können somit eine elliptische Kurve $E(\mathbb{F}_q)$ über dem Körper $\mathbb{F}_q$ mit einer Charakteristik größer als drei durch

$$E(\mathbb{F}_q) : y^2 = x^3 + ax + b \tag{2}$$

mit $a, b \in \mathbb{F}_q$ darstellen. Hinzu kommt, wie erwähnt, noch der Punkt im Unendlichen $\mathcal{O}$. Die Gleichung (2) bezeichnen wir als spezielle Form der Weierstraßgleichung. Nach der Definition 2.3 muss die Gleichung (2) eine nicht singuläre affine ebene Kurve sein. Dies können wir überprüfen, indem wir die Determinante der Gleichung (2) berechnen, sie ergibt sich durch $4a^3 + 27b^2$. Wenn diese Determinante ungleich 0 modulo p ist, so handelt es sich im vorliegenden Fall um eine elliptische Kurve.

Das nun folgende Beispiel soll die vorangegangenen Definitionen noch etwas verständlicher machen.

Beispiel 2.4 *Wir nehmen an, dass das Polynom f die Form $f = y^2 - x^3 - 2x - 4$ hat. Sei $p = 5$ und der zugrunde liegende Körper $\mathbb{F}_5$. Dann hat die Menge der Nullstellen die folgende Gestalt: $C_f(\mathbb{F}_5) = \{(a, b) \in \mathbb{F}_p) | \, 0 = b^2 - a^3 - 2a - 4\}$. Um die Elemente der Menge $C_f(\mathbb{F}_5)$ zu finden, müssen wir für $a, b \in \mathbb{F}_5$ die Lösungen der Gleichung*

$$y^2 = x^3 + 2x + 4 \tag{3}$$

finden. Dazu setzen wir der Reihe nach alle Möglichkeiten von a in die rechte Seite der Gleichung (3) ein und überprüfen, ob das Legendre-Symbol das Ergebnis 1 liefert, d.h. es muss getestet werden, ob die rechte Seite ein Quadrat modulo 5 ist.

a	$a^3 + 2a + 4$	Quadrat modulo 5	b
0	4	ja	$2, 3$
1	0	nein	-
2	1	ja	$1, 4$
3	2	nein	-
4	1	ja	$1, 4$

Aus der Tabelle kann man die Werte für $C_f(\mathbb{F}_p)$ leicht ablesen. Damit ergibt sich $C_f(\mathbb{F}_5) = \{(0, 2), (0, 3), (2, 1), (2, 4), (4, 1), (4, 4)\}$. Jetzt bleibt noch die affine ebene Kurve $C_f(\mathbb{F}_5)$ auf Singularität hin zu überprüfen. Wir testen also, ob es Punkte aus $C_f(\mathbb{F}_5)$ gibt, für die die beiden partiellen Ableitungen von f zu Null werden. Diese ergeben

$$\frac{\partial f}{\partial x}(a, b) = -3x^2 - 2, \ \frac{\partial f}{\partial y}(a, b) = 2y.$$

Da die beiden Ableitungen $\frac{\partial f}{\partial x}(a, b)$ und $\frac{\partial f}{\partial y}(a, b)$ in keinem Punkt zu Null werden, können wir durch Einsetzen erkennen, dass die Kurve $C_f(\mathbb{F}_5)$ nicht singulär ist. Auch über die Determinante $4a^3 + 27b^2$ der Gleichung (3) können wir testen, ob die Kurve $C_f(\mathbb{F}_5)$ singulär ist. Da der Wert der Determinante $4 \bmod 5$ ergibt wissen wir, dass die Kurve $C_f(\mathbb{F}_5)$ nicht singulär ist. Die Punkte der Gleichung (3) können wir damit zusammen mit dem Punkt $\mathcal{O}$ als elliptische Kurve bezeichnen.

Nachdem wir nun kurz die affinen Kurven eingeführt haben, können wir jetzt zu den projektiven Kurven übergehen. Sie werden, wenn bestimmte Bedingungen gelten, auch als elliptische Kurven bezeichnet. Wann projektive Kurven elliptische Kurven sind, werden wir im Folgenden diskutieren. Projektive Kurven bieten den Vorteil, dass man sämtliche Operationen, ohne invertieren zu müssen, durchführen kann.

Definition 2.5 *Sei die Menge $\mathbb{P}^2(\mathbb{F}) = \{(a, b, c) | \, a, b, c \in \mathbb{F}\,, (a, b, c) \neq (0, 0, 0)\}$ ein zweidimensionaler projektiver Raum. Ein Polynom g mit drei Variablen x, y, z, wobei $x, y, z \in$*

$\mathbb{F}$ *der Form*

$$g(x, y, z) = \sum_{i+j+k=d} \gamma_{i,j,k}\, x^i y^j z^k$$

mit Koeffizienten $\gamma_{i,j,k}$*, von denen nur endlich viele von Null verschieden sind und für die gilt* $i + j + k = d$ *wenn* $\gamma_{i,j,k}$ *nicht verschwindet, heißt homogen vom Grad* d*.*

Definition 2.6 *Sei* g *ein homogenes Polynom in* $\mathbb{F}$ *vom Grad* d*, dann ist die Menge der Nullstellen von* g *wie folgt definiert.*

$$C_g(\mathbb{F}) = \left\{(a, b, c) \in \mathbb{P}^2(\mathbb{F}) |\ g(a, b, c) = 0\right\}.$$

Jede dieser Nullstellenmengen wird auch als projektive ebene Kurve bezeichnet.

Die Koordinaten $(a, b, c) \in \mathbb{P}^2(\mathbb{F})$ werden als projektive Koordinaten bezeichnet. Elliptische Kurven können sowohl in affinen also auch in projektiven Koordinaten dargestellt werden.

Auch für die Nullstellenmenge $C_g(\mathbb{F})$ des Polynoms $g(x, y, z)$ können wir die Eigenschaft der Singularität wie folgt definieren.

Definition 2.7 *Eine projektive ebene Kurve* $C_g(\mathbb{F})$ *eines homogenen Polynoms* g *vom Grad* d *heißt singulär im Punkt* $(a, b, c) \in \mathbb{P}^2(\mathbb{F})$*, wenn alle partiellen Ableitungen von* g *im Punkt* (a, b, c) *verschwinden. Es gilt*

$$\frac{\partial g}{\partial x} = 0,\ \frac{\partial g}{\partial y} = 0,\ \frac{\partial g}{\partial z} = 0.$$

Die projektive ebene Kurve $C_g(\mathbb{F})$ *heißt nicht singulär, falls* $C_g(\overline{\mathbb{F}})$ *in keinem Punkt singulär ist.*

Darüber hinaus liegt nicht nur der Punkt (a, b, c) im projektiven Raum, sondern auch der Punkt (ta, tb, tc) mit $t \neq 0$. Das bedeutet, dass ein Punkt auf einer projektiven Kurve über $\mathbb{F}$ genau p viele Repräsentanten hat, die alle demselben Punkt einer affinen ebenen Kurve entsprechen. Im Gegensatz dazu gibt es für alle Punkte, ausgenommen dem Punkt $\mathcal{O}$, auf einer affinen Kurve nur genau eine Ausprägung. Der Punkt $\mathcal{O}$ ist auf einer affinen ebenen Kurve nicht darstellbar. Wir werden später sehen, dass diese Tatsache, einen Punkt als (ta, tb, tc) darstellen zu können, uns bei projektiven Operationen auf elliptischen Kurven einige Vorteile verschafft. Es ist dadurch möglich z. B. zwei Punkte zu Addieren, ohne ein Inverses einer Koordinate der beteiligten Punkte bilden zu müssen. Dies gilt, da wir davon ausgehen, dass Investieren teurer ist als Addieren oder Multipliziert. Im Gegensatz dazu gibt es für alle Punkte, ausgenommen dem Punkt $\mathcal{O}$, auf einer affinen Kurve nur genau eine Ausprägung. Der Punkt $\mathcal{O}$ ist auf einer affinen ebenen Kurve nicht darstellbar. Wir

werden später sehen, dass diese Tatsache, einen Punkt als (ta, tb, tc) darstellen zu können, uns bei projektiven Operationen auf elliptischen Kurven einige Vorteile verschafft. Es ist dadurch möglich z. B. zwei Punkte zu Addieren, ohne ein Inverses einer Koordinate der beteiligten Punkte bilden zu müssen. Dies gilt, da wir davon ausgehen, dass Investieren teurer ist als Addieren oder Multipliziert.

Es ist möglich eine affine Kurve in eine projektive Kurve zu überführen. Die Punkte der affinen Kurve können dabei, wie eben erwähnt, im projektiven Raum auch mehrere Repräsentanten haben. An dieser Stelle wollen wir mit einem Beispiel zeigen, wie man eine affine Kurve in eine projektive Kurve umwandelt.

Beispiel 2.8 *Es soll uns die Kurve $f(x, y) = y^2 - x^3 - 2x - 4$ aus Beispiel 2.4 als Ausgangspunkt dienen. Die dazugehörige Nullstellenmenge $C_f(\mathbb{F}_5)$ können wir leicht bestimmen indem wir $f(x, y) = 0$ setzen. Da jede dieser Nullstellenmengen eine affine Kurve darstellt, erhalten wir durch Umformung eine affine Kurve der Form $b^2 = a^3 + 2a + 4$. Sei c beliebig und ungleich 0 und sei $a' = ac$, $b' = bc$, dann können wir dies in die affine Kurve einsetzen und erhalten*

$$\left(\frac{b'}{c}\right)^2 = \left(\frac{a'}{c}\right)^3 + 2\left(\frac{a'}{c}\right) + 4.$$

Um die Brüche zu eliminieren kann man das ganze noch mit c^3 multiplizieren und erhält

$$b'^2 c = a'^3 + 2a'c^2 + 4c^3. \tag{4}$$

Die Gleichung (4) ist sicherlich eine projektive Kurve und damit eine Lösung des Nullstellenpolynoms $C_g(\mathbb{F}_5)$ eines homogenen Polynoms g. Wir erhalten somit $g(x, y, z) = yz - x^3 - 2xz^2 - 4z^3$ als Gleichung für ein homogenes Polynom.

Auf diese Art kann man also eine affine Kurve in eine projektive Kurve umwandeln. Nun wäre es doch interessant zu wissen, wie man affine Koordinaten der Punkte einer affinen Kurve in projektive Koordinaten einer projektiven Kurve umrechnen kann. Dieses wird durch die folgende Abbildung möglich.

Definition 2.9 *Sei s die Abbildung*

$$s(a, b) = (a, b, 1)$$

eines Punktes $(a, b) \in \mathbb{A}^2(\mathbb{F})$ im affinen Raum nach einem Punkt $(a, b, c) \in \mathbb{P}^2(\mathbb{F})$ des projektiven Raums. Sei s^{-1} die Umkehrabbildung

$$s^{-1}(a, b, c) = \left(\frac{a}{c}, \frac{b}{c}\right)$$

für $c \neq 0$.

Die Abbildung s ist leicht nachvollziehbar, denn setzt man in einer projektive Kurve $z = 1$, ergibt sich daraus die dazugehörige affine Kurve auf der der einzusetzende Punkt ja auch liegt. Eine wichtige Eigenschaft dieser Abbildung ist die Injektivität, das heißt, dass jedes Bild aus $\mathbb{P}^2(\mathbb{F})$ höchstens ein Urbild aus $\mathbb{A}^2(\mathbb{F})$ besitzt. Mit anderen Worten: es gibt zu jedem Punkt im projektiven Raum einen oder keinen entsprechenden Punkt im affinen Raum.

Nach Definition 2.9 ist die Abbildung von s^{-1} für alle Elemente des projektiven Raums definiert mit $c \neq 0$. Wir wollen die Gleichung für die projektive Ebene

$$b'^{\,2}c = a'^{\,3} + 2a'c^2 + 4c^{\,3} \tag{5}$$

aus Beispiel 2.8 noch etwas näher betrachten. Wenn der Fall eintritt, dass $c = 0$, dann erhalten wir $0 = a'^{\,3}$ und damit muss auch $a' = 0$ sein. Für b' können wir hierbei also beliebige Werte aus $\mathbb{F}$ einsetzen und erhalten somit Punkte, die zwar auf der Gleichung (5) für die projektive Ebene, nicht aber auf der Gleichung $b'^{\,2} = a'^{\,3} + 2a' + 4$ für die affine Ebene liegen. Wir können also sagen, dass die projektive Ebene alle Punkte der affinen Ebene umfasst und darüber hinaus noch einen weiteren Punkt $(0, 1, 0)$, der auch durch (0,t,0) dargestellt werden kann.

Wenn man alle Lösungen von $C_g(\mathbb{F})$ durch die Funktion $s\,(C_f(\mathbb{F}))$ darstellen kann, dann fehlt, wie eben erwähnt, noch die einzige Lösung, nämlich $(0, 1, 0)$, die durch diese Abbildung nicht dargestellt werden kann. Der Punkt $(0, 1, 0)$ wird auch als *Punkt im Unendlichen* bezeichnet. Dieser Punkt bildet das Neutralelement einer auf elliptischen Kurven definierten Gruppe.

Bis hier hin haben wir affine und projektive Kurven kennen gelernt. Wir haben die Zusammenhänge der Punktedarstellung erläutert und die Eigenschaft der Singularität eingeführt. Für unsere Zwecke soll dies erst einmal reichen, um die nun folgenden elliptischen Kurven beschreiben zu können. Ausführlichere Darstellungen der Hintergründe sind in [Wer02] zu finden.

Wir haben bereits in der Definition 2.3 elliptische Kurven in affiner Form eingeführt. Nun können wir elliptische Kurven aber auch in projektiver Form darstellen.

Definition 2.10 *Sei g ein homogenes Polynom vom Grad 3 der Form*

$$g(x, y, z) = y^2z + a_1xyz + a_3yz^2 - x^3 - a_2x^2z - a_4xz^2 - a_6z^3,$$

wobei $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}$ sind. Eine elliptische Kurve E über einem Körper $\mathbb{F}$ in projektiver Form ist die Menge der Punkte einer nicht singuläre projektiven ebenen Kurve $C_g(\mathbb{F})$

eines homogenes Polynom g. Wir schreiben

$$E(g) := C_g(\mathbb{F}).$$

Mit $E(g)$ ist die Menge der elliptischen Kurven über einem homogenen Polynom g gemeint. Die Punkte einer elliptischen Kurve erfüllen damit die Gleichung

$$y^2z + a_1xyz + a_3yz^2 = x^3 + a_2x^2z + a_4xz^2 + a_6z^3. \tag{6}$$

Auch hier schreiben wie ähnlich wie bei elliptischen Kurven in affiner Form $E(\mathbb{F})$ und meinen damit eine spezielle elliptische Kurve über dem Körper $\mathbb{F}$. Für Körper $\mathbb{F}_q$ mit einer Charakteristik größer als drei können wir die Gleichung (6) erneut, ähnlich wie im affinen Fall, zu

$$E(\mathbb{F}_p) : y^2z = x^3 + axz^2 + bz^3 \tag{7}$$

vereinfachen. Auch hier kann man überprüfen ob es sich im vorliegenden Fall um eine elliptische Kurve handelt, indem man testet, ob die Determinante $4a^3 + 27b^2 \neq 0 \bmod p$ ist. Der Punkt im Unendlichen $\mathcal{O}$ kann durch die Gleichung (7) auch ausgedrückt werden. Er wird durch dir Koordinaten $(0, 1, 0)$ repräsentiert.

Im Folgenden Kapitel soll es speziell um die Operationen auf elliptischen Kurven gehen, mit deren Hilfe eine Gruppenstruktur definiert wird.

3 Elementare Operationen

Nachdem wir im vorherigen Kapitel eine kurze Einführung in die mathematischen Hintergründe der elliptischen Kurven gegeben haben, werden wir nun Gruppen auf elliptischen Kurven definieren, um überhaupt mit ihnen arbeiten zu können. Gruppen haben wir bereits in Definition 1.2 kennen gelernt.

Wir benötigen für die Definition einer Gruppe zuerst eine Abbildung, die zwei Elemente auf ein weiteres Element innerhalb der Gruppe projiziert. Diese Gruppenoperation wurde in der Literatur als Addition definiert. Eine Gruppenstruktur kann man auf einer elliptischen Kurve über einem beliebigen Körper (siehe Definition 1.5) definieren. Für unsere Zwecke nehmen wir den Körper $\mathbb{F}_p$ mit p prim. Die Addition auf einer elliptischen Kurve wollen wir im Folgenden graphisch herleiten.

Wir beginnen mit der affinen Form der Weierstraßgleichung $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$. Wie schon erwähnt ist $\mathcal{O}$ der einzige Punkt der nicht auf dieser Kurve liegt, sondern nur im projektiven Raum. Er lässt sich deshalb in unserer affinen Betrachtung nicht korrekt darstellen, wir können ihn uns aber als an der Spitze der y-Achse liegend vorstellen. Für die geometrischen Abbildungen verwenden wir die Kurve $E : y^2 = x^3 - 4x + 5$ und als Grundkörper die reellen Zahlen $\mathbb{R}$, um eine anschauliche Darstellung der Kurve zu erhalten. Später stellen wir diese Kurve auch noch über einem Körper der ganzen Zahlen dar. Dabei erhalten wir allerdings nur eine Menge von Punkten. Über den reellen Zahlen lassen sich elliptische Kurven daher zu mindestens für die graphische Betrachtung besser darstellen. Für die Kryptographie kann man die reellen Zahlen nicht als Grundkörper nehmen, dass sie keine endliche Menge an Elementen besitzen. Die Abbildung 1 zeigt die Addition zweier Punkte $P, Q \in E(\mathbb{R})$. Dabei sind P und Q nicht gleich und P ist auch nicht der Inverse Punkt von Q. Mit anderen Worten ist P nicht der an der x-Achse gespiegelte Punkt Q. Um die Punkte P und Q zu addieren, legt man eine Gerade durch beide Punkte. Diese schneidet die Kurve in einem dritten Punkt, den wir als $-R$ bezeichnen. Da jede elliptische Kurve symmetrisch zur x-Achse ist, erhalten wir den Punkt R wenn wir $-R$ an der x-Achse spiegeln. Es ergibt sich $P + Q = R$ unter der eben genannten Bedingung. Damit die Definition der Addition komplett ist, müssen wir es auch erlauben, dass sich ein Punkt verdoppeln lässt. Auch für diesen Fall gibt es eine geometrische Lösung die wir im Abbildung 2 dargestellt haben. Wenn wir $P + P$ beschreiben wollen, dann wird an die Kurve im Punkt P eine Tangente gelegt, welche die Kurve in einem Punkt $-R$ schneidet. Nach der Spiegelung von $-R$ an der x-Achse erhalten wir R als Ergebnis und damit $P + P = 2P = R$.

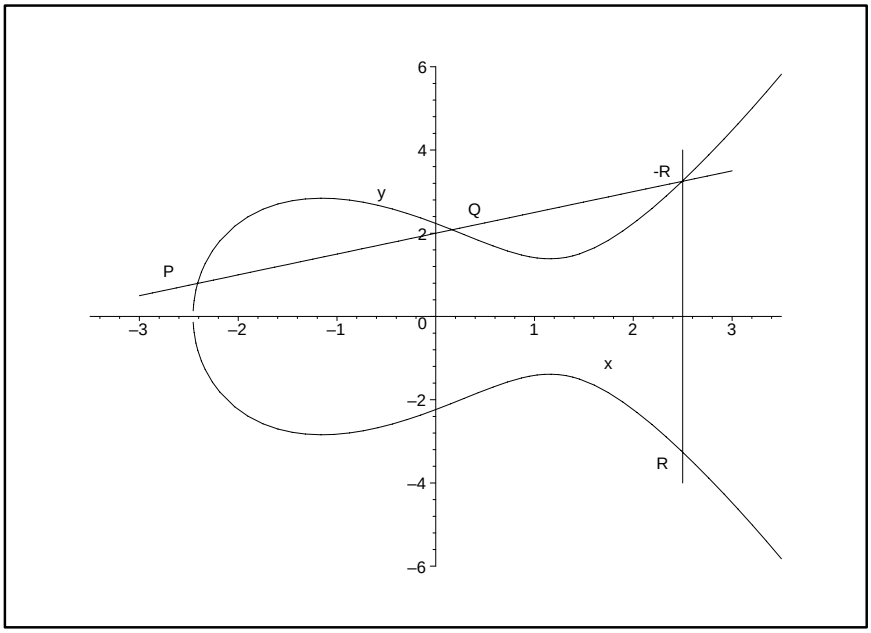


Abbildung 1: Addition: $P + Q = R$, mit $P \neq \pm Q$

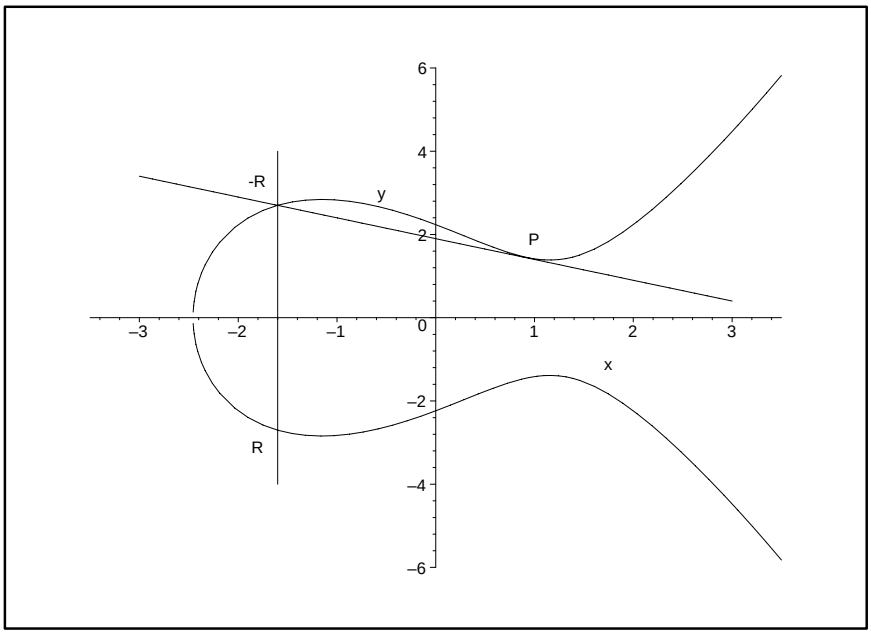


Abbildung 2: Addition: $P + P = 2P = R$

Implizit haben wir auch schon geometrisch gezeigt wie man einen Punkt P negiert.

Das geschieht durch die Spiegelung des Punktes an der x-Achse, hierbei ändert sich nur das Vorzeichen der y-Koordinate.

Als letzte geometrische Operation zeigt Abbildung 3 das Ergebnis von $P - P = \mathcal{O}$. Dabei wird wieder eine Gerade durch P und $-P$ gelegt, die in diesem Fall nicht die affine Kurve schneidet, sondern ihren Schnittpunkt in $\mathcal{O}$ hat. Dieser Punkt liegt jedoch im projektiven Raum und kann dadurch hier nur durch einen senkrechten Strich, der die y-Achse an der Spitze schneiden soll, veranschaulicht werden.

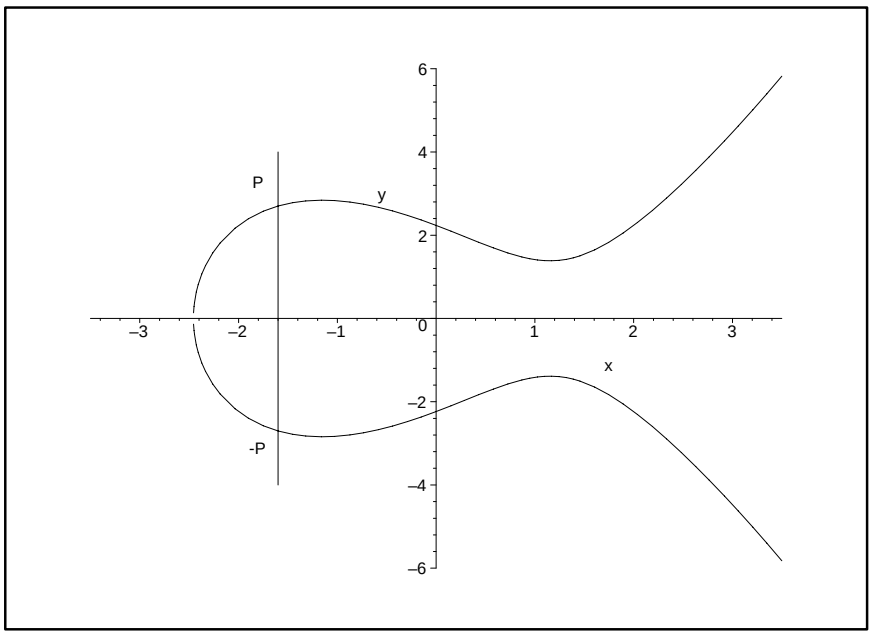


Abbildung 3: Addition: $P - P = \mathcal{O}$

In Abbildung 4 sehen wir die Kurve $E(\mathbb{Z}_7) : y^2 = x^3 - 4x + 5$ über dem Primkörper $\mathbb{Z}_7$. Hierbei erhält man natürlich nur eine Punktemenge, bei der die Koordinaten der Punkte nur ganzzahlige Werte aus $\mathbb{Z}_7$ annehmen können. Die Punkte sind in der folgenden Wertetabelle aufgelistet.

x	$x^3 - 4x + 5$	Quadrat modulo 7	y
0	5	nein	-
1	2	ja	$3, 4$
2	5	nein	-
3	6	nein	-
4	4	ja	$2, 5$
5	5	nein	-
6	1	ja	$1, 6$

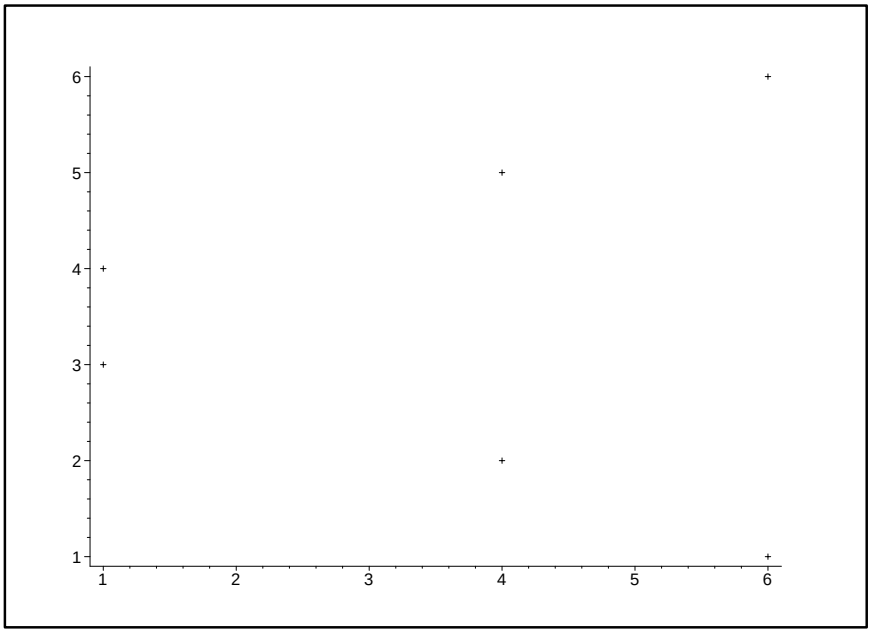


Abbildung 4: $E(\mathbb{Z}_7) : y^2 = x^3 - 4x + 5$

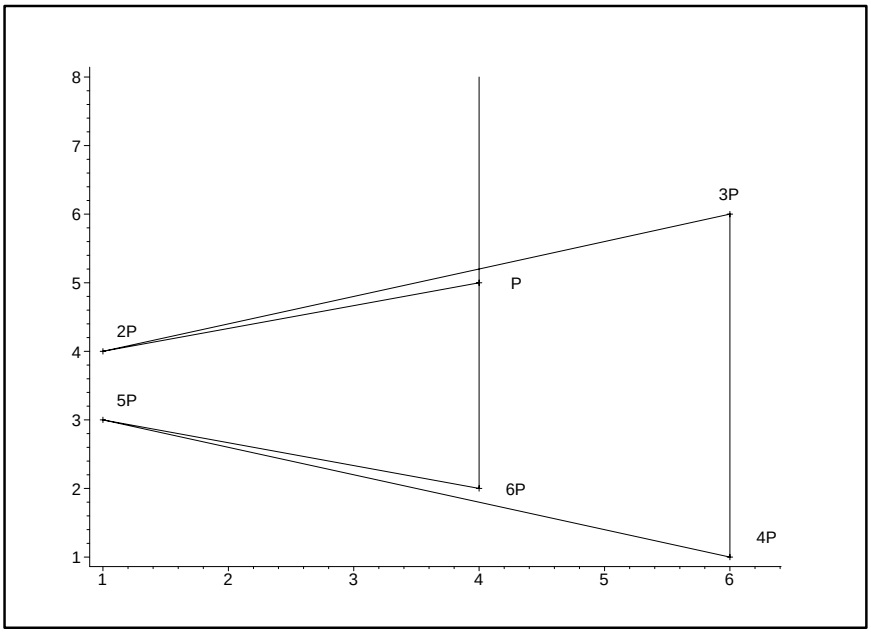


Abbildung 5: grafische Bestimmung der Ordnung des Punktes $P = (4, 5) \in E(\mathbb{Z}_7) : y^2 = x^3 - 4x + 5$

Wir wollen jetzt zum Abschluss der geometrischen Betrachtung die Vielfachen des Punktes $P = (4, 5)$ der Kurve $E(\mathbb{Z}_7) : y^2 = x^3 - 4x + 5$ geometrisch darstellen. Dieses

zeigt Abbildung 5. Addiert man zu $6P$ erneut P hinzu, so erhält man den Punkt im unendlichen $\mathcal{O}$. Er soll hier nur durch einen senkrechten Strich dargestellt werden, der die y-Achse an der Spitze schneidet. Wenn man nun $\mathcal{O} + P$ berechnen will, dann landet man wieder bei P. Dieses entspricht auch $(1 + 7k)P$ für $k \geq 0$.

Nach der geometrischen Veranschaulichung werden wir uns nun den konkreten mathematischen Berechnungen widmen. Als Ausgangspunkt dient uns dabei der folgende Satz.

Satz 3.1 *Sei $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ eine elliptische Kurve mit $p > 3$ prim. Seien außerdem zwei Punkte P und Q auf der elliptischen Kurve $E(\mathbb{F}_p)$ mit $P = (x_1, y_1)$ und $Q = (x_2, y_2)$ gegeben, dann gilt:*

1. *Die Negation von $P = (x_1, y_1)$ ergibt $-P = (x_1, -y_1)$,*
2. *Wenn $P = \mathcal{O}$ dann ist die Negation $-P = \mathcal{O}$,*
3. *$P + Q = \mathcal{O}$, wenn $P = -Q$,*
4. *$P + \mathcal{O} = P$,*
5. *$P + Q = R$ mit $R = (x_3, y_3)$, wenn $P \neq -Q$, dabei ergeben sich x_3 und y_3 wie folgt:*

$$x_3 = \lambda^2 - x_1 - x_2, \qquad y_3 = \lambda(x_1 - x_3) - y_1$$

mit

$$\lambda = \begin{cases} \left(\frac{y_1 - y_2}{x_1 - x_2}\right) & \text{wenn } P \neq Q \\ \frac{3x_1^2 + a}{2y_1} & \text{wenn } P = Q. \end{cases}$$

Der Beweis dieses Satzes ist in [Wer02, S.47ff] zu finden.

Wir werden uns jetzt mit den Grundoperationen für projektive Koordinaten beschäftigen. Die Arithmetik mit projektiven Koordinaten lässt sich schneller durchführen, als die mit affinen Koordinaten, da man diese ohne Invertieren von Koordinaten durchführen kann. In [CMO98] sind Verfahren dazu in knapper Form dargestellt. Die nun folgenden Algorithmen sollen auch später bei der Implementierung umgesetzt werden.

Wenn wir annehmen, dass das Invertieren von Koordinaten teurer als das Multiplizieren ist, dann ist es effizienter mit projektiven Koordinaten zu rechnen, um somit die Inversion einsparen zu können. Für projektive Koordinaten verwenden wir die projektive Form

$$E(\mathbb{F}_p) : y^2 z = x^3 + axz^2 + bz^3 \tag{8}$$

einer elliptischen Kurve. Ein Punkt (x, y, z) in projektiven Koordinaten entspricht für $z \neq 0$ den affinen Koordinaten $(\frac{x}{z}, \frac{y}{z})$. Den projektiven Punkt $(0, 1, 0)$ stellen wir durch $\mathcal{O}$ dar.

3.1 Die projektive Addition

Der nun folgende Algorithmus beschreibt die Addition zweier Punkte P und Q, bei dem die Punkte der eben genannten Gleichung (8) genügen sollen. Es ist zu beachten, dass es sich hierbei um eine reine Addition, keine Verdopplung handelt, deshalb gilt der folgende Algorithmus nur für $P \neq \pm Q$ und $P, Q \neq \mathcal{O}$.

Algorithmus 3.1 Addition

Eingabe: $P, Q \in \mathrm{E}(\mathbb{F}_p)$ mit $P = (x_1, y_1, z_1)$, $Q = (x_2, y_2, z_2)$ und $P \neq \pm Q$, $P, Q \neq \mathcal{O}$

Ausgabe: $R \in \mathrm{E}(\mathbb{F}_p)$ mit $R = P + Q$

Schritt 1: $u = y_2 z_1 - y_1 z_2$

Schritt 2: $v = x_2 z_1 - x_1 z_2$

Schritt 3: $A = u^2 z_1 z_2 - v^3 - 2v^2 x_1 z_2$

Schritt 4: $x_3 = vA$

Schritt 5: $y_3 = u(v^2 x_1 z_2 - A) - v^3 y_1 z_2$

Schritt 6: $z_3 = v^3 z_1 z_2$

Schritt 7: **Rückgabe:** $R = (x_3, y_3, z_3)$

Wenn wir nun die Addition eines Punktes mit sich selbst als eine Verdopplung darstellen wollen, dann benötigen wir eine andere Vorgehensweise, als die eben beschriebene. Es gilt $P = Q$ und $P, Q \neq \mathcal{O}$. Die Berechnung funktioniert nach folgendem Verfahren.

Algorithmus 3.2 Addition (Verdoppelung)

Eingabe: $P, Q \in \mathrm{E}(\mathbb{F}_p)$ mit $P = (x_1, y_1, z_1)$, $Q = (x_2, y_2, z_2)$ und $P = \pm Q$, $P, Q \neq \mathcal{O}$

Ausgabe: $R \in \mathrm{E}(\mathbb{F}_p)$ mit $R = P + Q$

Schritt 1: $w = az_1^2 + 3x_1^2$

Schritt 2: $s = y_1 z_1$

Schritt 3: $B = x_1 y_1 s$

Schritt 4: $h = w^2 - 8B$

Schritt 5: $x_3 = 2hs$

Schritt 6: $y_3 = w(4B - h) - 8y_1^2 s^2$

Schritt 7: $z_3 = 8s^3$

Schritt 8: **Rückgabe:** $R = (x_3, y_3, z_3)$

Die Verdopplung benötigt dabei 13 Multiplikationen, im Vergleich dazu braucht die eben vorgestellte Addition 14 Multiplikationen.

Den Fall, dass einer der beiden Punkte dem Punkt im Unendlichen entspricht, brauchen wir hier nicht mehr extra zu betrachten. Dieser Fall wird genauso wie bei affinen Koordinaten behandelt, nur eben mit projektiven Koordinaten. Dies gilt auch wenn beide Punkte $\mathcal{O}$ sind.

Die nachfolgende Tabelle (aus [CMO98, S. 53]) soll einen Überblick über die Geschwindigkeiten der affinen und projektiven Addition vermitteln und dabei den Vorteil der projektiven Operationen verdeutlichen. Da Addition und Subtraktion in $\mathbb{F}_p$ vergleichsweise sehr schnelle Operationen sind, lassen wir diese hier außen vor. Das Quadrieren in $\mathbb{F}_p$ zählen wir dabei auch mit zu den Multiplikationsoperationen in $\mathbb{F}_p$.

Art der Operation	Inversionen	Multiplikationen
affine Addition	1	3
affine Verdopplung	1	4
projektive Addition	-	14
projektive Verdopplung	-	13

Tabelle 1: Geschwindigkeitsvergleich zwischen affiner und projektiver Addition

Es wird deutlich, dass die projektiven Operationen immer vorzuziehen sind, solange das Invertieren um etwa Faktor 11 langsamer als Multiplizieren ist, was auch meistens der Fall ist.

3.2 Die projektive skalare Multiplikation

Die skalare Multiplikation auf einer elliptischen Kurve wird als $kP = R$ dargestellt. Dabei ist k eine positive ganze Zahl größer gleich 0 und P ein beliebiger Punkt auf einer elliptischen Kurve. Wir wollen nun ein paar Algorithmen zur Punktmultiplikation vorstellen. Dabei beginnen wir mit einer etwas „langsameren“ Variante der binären Multiplikationsmethode und stellen im Anschluss eine wesentlich schnellere Variante vor.

Algorithmus 3.3 Multiplikation (binär)

Eingabe: $P \in \mathrm{E}(\mathbb{F}_p)$, ganze Zahl k mit $k = \sum_{j=0}^{l-1} k_j 2^j$, $k_j \in \{0, 1\}$

Ausgabe: $R \in \mathrm{E}(\mathbb{F}_p)$ mit $R = kP$

SCHRITT 1: $R \leftarrow \mathcal{O}$

SCHRITT 2: **Für** $j = l - 1$ bis 0

Setze $R \leftarrow 2R$

Falls $k_j = 1$ **Setze** $R \leftarrow R + P$

SCHRITT 3: **Rückgabe:** R

Wenn n die binäre Länge der Zahl k ist, dann hat der Algorithmus 3.3 eine Laufzeit von $O(\log n)$. Das folgende Beispiel veranschaulicht die Funktionsweise dieser einfachen Variante der skalaren Multiplikation.

Beispiel 3.2 *Sei P ein beliebiger Punkt auf einer beliebigen Kurve. Wir wollen nun mit der binären Multiplikation $R = 9P$ berechnen. Es lassen sich dabei alle Operationen auf die in Kapitel 3.1 vorgestellten Additionen zurückführen. Wie wir sehen ergibt sich R als*

$$(2P + 2P) + (2P + 2P) + P = R.$$

Es gibt aber noch wesentlich schnellere Varianten zur Berechnung der skalaren Multiplikation, dafür benötigt man allerdings noch eine spezielle Darstellung.

3.2.1 Die NAF - Darstellung

Die Bezeichnung NAF steht für non-adjacent form. Es gibt keine zwei aufeinander folgenden Elemente, die nicht Null sind. Mit anderen Worten: von zwei aufeinander folgenden Zahlen ist mindestens eine der beiden eine Null. Die NAF hat den Vorteil, dass man im Vergleich zur Binärdarstellung mehr Nullen erhält, was dadurch zu weniger Operationen führt. Bei der NAF handelt es sich um eine spezielle Darstellung des Faktors k. Dieser wird dabei nicht mehr in binärer Form dargestellt, sondern als $k = \sum_{l=0}^{n} c_l 2^l,\ c_l \in \{-1, 0, 1\}$. Man benötigt bei der Multiplikation mit der NAF damit weniger Additionsoperationen als ohne sie, was zu einem deutlichen Geschwindigkeitsvorteil führt. Die NAF ist durch den Algorithmus 3.4 leicht zu errechnen. Dabei ist rechts das niederwertigste Bit und links das höchstwertigste Bit angeordnet. Die NAF wird hierbei von rechts nach links mit Werten gefüllt. Wir verwenden den folgenden Algorithmus aus [KDJ04, S. 28].

Algorithmus 3.4 NAF

Eingabe: positive ganze Zahl k

Ausgabe: $c = (c_n, \ldots, c_0)$

SCHRITT 1: $i \leftarrow k,\ l \leftarrow 0$

SCHRITT 2: **Solange** $i > 0$

Falls $(i \equiv 1 \bmod 2)$

$c_l \leftarrow 2 - (i \bmod 4)$

$i \leftarrow i - c_l$

Sonst $c_l \leftarrow 0$

$i \leftarrow \frac{i}{2}$

$l \leftarrow l + 1$

SCHRITT 3: **Rückgabe:** c

Diese Darstellung soll aber auch noch mal an einem Beispiel verdeutlicht werden. Wir vergleichen dabei auch die binäre Darstellung der betrachteten Zahl k. Es ist noch anzumerken, dass die NAF mindestens so lang ist wie die binäre Form, aber höchstens ein Bit

länger. Sei n die Bitlänge einer gegebenen Dezimalzahl k, dann hat der Algorithmus 3.4 eine Laufzeit von $O(n)$.

Beispiel 3.3 *Für unser Beispiel wählen wir $k = 119$. Die Binärdarstellung von k ergibt sich durch $k_{bin} = \sum_{l=0}^{n-1} c_l 2^l,\ c_l \in \{0,1\}$. Damit erhalten wir $k_{bin} = 1110111$, was in diesem Fall 7 Bit lang ist. Nun wollen wir die NAF k_{NAF} für k bestimmen. Dieses veranschaulichen wir in der folgenden Tabelle.*

Runde	l	c_l	i	$c = (c_n, \ldots, c_0)$
1	0	-1	119	-1
2	1	0	60	$0, -1$
3	2	0	30	$0, 0, -1$
4	3	-1	15	$-1, 0, 0, -1$
5	4	0	8	$0, -1, 0, 0, -1$
6	5	0	4	$0, 0, -1, 0, 0, -1$
7	6	0	2	$0, 0, 0, -1, 0, 0, -1$
8	7	1	1	$1, 0, 0, 0, -1, 0, 0, -1$

Wir erhalten also mit $k_{NAF} = (1, 0, 0, 0, -1, 0, 0, -1)$ eine 8 Bit lange NAF - Darstellung von k. Es wird deutlich, dass die NAF wesentlich mehr Nullen enthält, was in der Berechnung für jede Null eine Addition erspart. Während die NAF nur drei Elemente besitzt, die verschieden von Null sind, hat die Binärdarstellung sechs Elemente verschieden von Null.

3.2.2 Die DOUBLE-AND-ADD-OR-SUBTRACT Multiplikation

Eine Methode um „schnell“ die skalare Multiplikation durchführen zu können, ist die DOUBLE-AND-ADD-OR-SUBTRACT Methode, welche mit der Zahl k in NAF rechnet.

Algorithmus 3.5 Multiplikation (DOUBLE-AND-ADD-OR-SUBTRACT)

Eingabe: $P \in \mathrm{E}(\mathbb{F}_p)$, ganze Zahl k in NAF mit $k = \sum_{l=0}^{n} c_l 2^l,\ c_l \in \{-1, 0, 1\}$

Ausgabe: $R \in \mathrm{E}(\mathbb{F}_p)$ mit $R = kP$

SCHRITT 1: $P \leftarrow \mathcal{O}$

SCHRITT 2: **Für** $l = n$ bis 0

$R \leftarrow 2R$

Falls $c_l = 1$ **Setze** $R \leftarrow R + P$

Sonst Falls $c_l = -1$ **Setze** $R \leftarrow R - P$

SCHRITT 3: **Rückgabe:** R

Sei n die Länge der Zahl k in NAF, dann hat der Algorithmus eine Laufzeit von $O(n)$.

Teil II
Algorithmen zum Punkte zählen

Damit elliptische Kurven über $\mathbb{F}_p$ für die Kryptographie interessant sind, ist es wichtig, dass ihre Gruppenordnung (Anzahl der Punkte) prim ist, damit der diskrete Logarithmus „schwer“ zu berechnen ist. Um die Gruppenordnung zu erhalten, werden wir drei Algorithmen angeben, die diese bestimmen.

Weiterhin bezeichnen wir mit $\#E(\mathbb{Z}_p)$ die Gruppenordnung von E. Aus den vorangegangenen Kapiteln wissen wir bereits, dass man auf der projektiven Weierstraßgleichung einen Punkt mehr darstellen kann, als auf der affinen Weierstraßgleichung. Es handelt sich dabei um den Punkt $\mathcal{O}$, der immer auf einer elliptischen Kurve liegt. Nun reicht es also aus die affine Weierstraßgleichung

$$E(\mathbb{Z}_p) : y^2 = x^3 + ax + b \tag{9}$$

zu betrachten, um die restlichen Punkte der Kurve zu berechnen. Damit wir alle Lösungen der Gleichung (9) für $E(\mathbb{Z}_p)$ berechnen können, setzen wir alle möglichen Werte von $x \in \mathbb{Z}_p$ in die rechte Seite der Gleichung (9) ein und testen, ob sich daraus ein Quadrat modulo p ergibt. Ob es sich im konkreten Fall um ein Quadrat modulo p handelt, können wir mit dem Legendre - Symbol aus Kapitel 1.4 bestimmen. Für die Lösungen der Gleichung (9) können wir damit die folgenden drei Fälle unterscheiden:

$$\left(\frac{x^3+ax+b}{p}\right) = \begin{cases} -1 & \text{wenn es kein } y \text{ mit } y^2 = x^3+ax+b \text{ gibt} \\ 0 & \text{wenn es genau ein } y \text{ mit } y^2 = x^3+ax+b \text{ gibt} \\ 1 & \text{wenn es genau zwei } y \text{ mit } y^2 = x^3+ax+b \text{ gibt} \end{cases}$$

Wenn uns das Legendre - Symbol als Ergebnis -1 zurück liefert, dann bedeutet dies, dass es keinen Wert für $y \in \mathbb{Z}_p$ gibt, für den $y^2 = x^3 + ax + b$ ist. Erhalten wir 0 als Ergebnis, hat die elliptische Kurve an der Stelle x eine Nullstelle und wenn wir 1 erhalten, gibt es genau zwei $y \in \mathbb{Z}_p$, für die die Gleichung $y^2 = x^3 + ax + b$ erfüllt ist. Aufgrund der Symmetrie einer elliptischen Kurve zur x-Achse bekommen wir die zwei Lösungen y und $-y$. Halten wir diese Ergebnisse noch einmal fest, dann ergibt sich für $\left(\frac{x^3+ax+b}{p}\right) = -1$ keine Lösung, für $\left(\frac{x^3+ax+b}{p}\right) = 0$ genau eine und für $\left(\frac{x^3+ax+b}{p}\right) = 1$ genau zwei Lösungen der Kurvengleichung (9). Zusammen ergibt dies für alle $x \in \mathbb{Z}_p$

$$\sum_{x\in\mathbb{Z}_p} \left(\left(\frac{x^3+ax+b}{p}\right) + 1 \right).$$

Da ja auch noch der Punkt $\mathcal{O}$ auf einer elliptischen Kurve liegt, erhalten wir für die Anzahl

der Punkte auf einer elliptischen Kurve:

$$\begin{aligned} \#E(\mathbb{Z}_p) &= 1 + \sum_{x \in \mathbb{Z}_p} \left(\left(\frac{x^3 + ax + b}{p} \right) + 1 \right) \\ &= 1 + \sum_{x \in \mathbb{Z}_p} \left(\frac{x^3 + ax + b}{p} \right) + \sum_{x \in \mathbb{Z}_p} 1 \\ &= 1 + p + \sum_{x \in \mathbb{Z}_p} \left(\frac{x^3 + ax + b}{p} \right). \end{aligned} \tag{10}$$

Die Gleichung (10) ist die Basis für den sehr einfachen naiven Algorithmus, der die schlechteste Laufzeit von allen Punktezählalgorithmen besitzt. Im Folgenden Kapitel wird dieser kurz beschrieben.

Nun können wir auch Abschätzen, wie viele Punkte höchstens auf einer elliptischen Kurve liegen. Man betrachte den Fall, dass wir für jedes $x \in \mathbb{Z}_p$ nach Einsetzen in die rechte Seite der Gleichung (9) den Wert 1 des Legendre - Symbols erhalten. Es ergibt sich also aus

$$\begin{aligned} \#E(\mathbb{Z}_p) &\leq 1 + p + \sum_{x \in \mathbb{Z}_p} 1 \\ &\leq 1 + p + p \\ &\leq 1 + 2p. \end{aligned}$$

Dadurch haben wir mit

$$\#E(\mathbb{Z}_p) \leq 1 + 2p$$

eine obere Schranke für die maximal mögliche Anzahl von Punkten auf einer elliptischen Kurve gefunden. Eine Abschätzung der minimalen und maximalen Anzahl von Punkten wird durch den folgenden Satz von Hasse gegeben.

Satz 3.4 ***von Hasse*** *Sei $E(\mathbb{F}_p)$ eine elliptische Kurve über $\mathbb{F}_p$. Dann gilt für die Anzahl der Punkte von $E(\mathbb{F}_p)$:*

$$|\#E(\mathbb{Z}_p) - p - 1| \leq 2\sqrt{p}.$$

Es ergibt sich:

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{Z}_p) \leq p + 1 + 2\sqrt{p}. \tag{11}$$

Wir können die Anzahl der Punkte $\#E(\mathbb{Z}_p)$ auf der elliptischen Kurve $E(\mathbb{Z}_p)$ durch

$$\#E(\mathbb{Z}_p) = p + 1 - t,$$

wobei $|t| \leq 2\sqrt{p}$ ist, darstellen. Der Wert für t befindet sich also in dem Intervall $-2\sqrt{p} \leq t \leq 2\sqrt{p}$, dem so genannten Hasse Intervall.

Auf den Beweis dieses Satzes wollen wir an dieser Stelle verzichten, er ist in [Sil86, S.131] zu finden.

4 Der naive Algorithmus

Der hier beschriebene Algorithmus zur Bestimmung von $\#E(\mathbb{Z}_p)$ ist sehr einfach zu implementieren, hat aber für praktische Zwecke aufgrund der zu hohen Laufzeit von $O(p \log p)$ keine Bedeutung. Er folgt unmittelbar aus der Gleichung (10).

Algorithmus 4.1 naiver Algorithmus

Eingabe: $E(\mathbb{Z}_p) : y^2 = x^3 + ax + b$

Ausgabe: $\#E(\mathbb{Z}_p)$

SCHRITT 1: **Setze** $\#E(\mathbb{Z}_p) \leftarrow 1 + p$

SCHRITT 2: **Für** $x = 0$ bis $p - 1$

Berechne das Legendre - Symbol von $LS \leftarrow \left(\frac{x^3+ax+b}{p}\right)$

Setze $\#E(\mathbb{Z}_p) \leftarrow \#E(\mathbb{Z}_p) + LS$

SCHRITT 3: **Rückgabe:** $\#E(\mathbb{Z}_p)$

5 Der Shanks-Mestre Algorithmus

Der Algorithmus von Shanks und Mestre basiert zum einen auf dem Satz von Lagrange 1.10 und zum anderen auf dem so genannten Babystep-Giantstep Algorithmus, der 1970 von D. Shanks entwickelt wurde. Mit diesem Algorithmus kann man die Ordnung von beliebigen Gruppen bestimmen. Er spielt deshalb eine zentrale Rolle im Shanks-Mestre Algorithmus. Durch den Satz von Hasse 3.4 wissen wir, dass die Gruppenordnung einer elliptischen Kurve $\#E(\mathbb{Z}_p)$ in dem Intervall aus Gleichung (11) liegen muss. Nach dem Satz von Lagrange 1.10 muss die Gruppenordnung $\#E(\mathbb{Z}_p)$ ein Vielfaches der Ordnung eines Kurvenpunktes sein. Wenn wir nun einen Punkt finden, dessen Ordnung groß genug ist, so dass genau ein Vielfaches der Punktordnung im Hasse Intervall aus Satz 3.4 liegt, dann ist dadurch die Gruppenordnung eindeutig bestimmt. Hat der betrachtete Punkt aber eine zu kleine Ordnung, also kleiner als die Länge des Hasse Intervalls $4\sqrt{p}$, dann können wir nicht sicher sein, ob uns der Algorithmus den korrekten Wert für die Gruppenordnung liefert.

Im Folgenden geben wir eine Variante des Algorithmus an, der aber unter den eben genannten Bedingungen keine Lösung findet. Zuerst einmal benötigen wir einen Algorithmus, der einen zufälligen Punkt auf einer elliptischen Kurve findet.

Algorithmus 5.1 Zufallspunkt

Eingabe: $E : y^2 = x^3 + a\,x + b$ in $\mathbb{F}_p$

Ausgabe: $(x, y) \in E(\mathbb{F}_p)$ zufällig

SCHRITT 1: Bestimme $x \in \mathbb{F}_p$ zufällig.

SCHRITT 2: **Solange** $\left(\frac{x^3+ax+b}{p}\right) = -1$
Bestimme $x \in \mathbb{F}_p$ zufällig.

SCHRITT 3: **Falls** $\left(\frac{x^3+ax+b}{p}\right) = 0$
Setze $y \leftarrow 0$
Sonst Berechne y als Quadratwurzel von $(x^3 + ax + b) \bmod p$ mit Algorithmus 1.2

SCHRITT 4: Wähle zufällig eine Zahl $w \in \{0, 1\}$.

SCHRITT 5: **Falls** $w = 0$
Rückgabe: (x, y)
Sonst Rückgabe: $(x, -y)$

Der Algorithmus 5.1 kann zufällig jeden Punkt der Kurve $E : y^2 = x^3 + a\,x + b$, außer den Punkt $\mathcal{O}$ finden. Der Punkt $\mathcal{O}$ würde für die Berechnung auch keinen Sinn machen, da seine Ordnung in jedem Fall zu klein ist. Ergibt das Legendre - Symbol in Schritt 2 den Wert 0, so wird in Schritt 3 y auf 0 gesetzt und in Schritt 5 der Punkt $(x, 0)$ zurückgegeben. Liefert das Legendre - Symbol den Wert 1, dann wird y als Quadratwurzel von $(x^3 + ax + b) \bmod p$ berechnet. Mit einer Wahrscheinlichkeit von $\frac{1}{2}$ wird anschließend in Schritt 4 w auf 1 oder 0 gesetzt. In Schritt 5 wird je nach Wahl von w der Punkt (x, y) oder $(x, -y)$ zurückgegeben.

Da in Schritt 1 ein x zufällig bestimmt wird, handelt es sich hierbei um einen randomisierten Algorithmus, wie der Name auch schon vermuten lässt. Die Wahrscheinlichkeit ein x zu finden, welches in Schritt 2 den Test besteht, beträgt $\frac{1}{2}$, die es genauso viele quadratische Reste wie quadratische Nichtreste gibt. Im Erwartungswert bracht der Algorithmus als maximal 2 Versuche, um Schritt 2 beenden zu können.

Nachdem wir nun wissen wie wir einen zufälligen Punkt auf einer elliptischen Kurve bestimmen können, widmen wir uns jetzt dem eigentlichen Algorithmus von Schanks-Mestre. Dieser funktioniert nach dem schon erwähnten Babystep-Giantstep Algorithmus. Es werden hierbei zwei Listen mit Kurvenpunkten berechnet. Die Liste $L1$ berechnet dabei die Babysteps. Dies nennt man so, weil man sich dabei nur in sehr kleinen Schritten vorwärts bewegt. Die Liste $L2$ berechnet anschließend die Giantsteps, die sich dabei in größeren Schritten als die Babysteps vorwärts bewegen. Das soll heißen, dass man in der Liste $L2$ mehr Additionen auf dem gewählten Punkt anwendet und sich dadurch in größeren Schritten bewegt. Da wir davon ausgehen, dass nur ein Vielfaches der Punktordnung

im Intervall aus Gleichung (11) liegt, kann es auch nur ein Paar geben für dass $L1_i = L2_j$ gilt. Die Indizes i und j bezeichnen die Positionen innerhalb der entsprechenden Liste. Wenn wir eine solche Übereinstimmung gefunden haben, müssen wir in Schritt 5 nicht alle Elemente der Liste $L2$ berechnen und können die Schleife frühzeitig beenden. Gibt es erst beim letzten Listenelement von $L2$ eine Übereinstimmung, müssen wir natürlich alle Elemente berechnen lassen.

Algorithmus 5.2 Shanks-Mestre

Eingabe: $E : y^2 = x^3 + a\,x + b$ in $\mathbb{F}_p$

Ausgabe: $\#E(\mathbb{F}_p)$

SCHRITT 1: Wähle $Q \in E(\mathbb{F}_p)$ mit Algorithmus 5.1, wobei angenommen wird, dass die Ordnung von Q größer als $4\sqrt{p}$ ist.

SCHRITT 2: Sei $D = (p+1)\,Q$ und $W = D + (\lfloor 2\sqrt{p} \rfloor)\,Q$.

SCHRITT 3: Sei $m = \lceil 2\,p^{\frac{1}{4}} \rceil$ und $R = mQ$.

SCHRITT 4: **Für** $i = 0, \ldots, m-1$

Berechne $L1_i = i\,Q$.

SCHRITT 5: **Für** $j = 0, \ldots, m-1$

Berechne $L2_j = W - jR$.

Für $i = 0, \ldots, m-1$

Falls $L1_i = L2_j$ **Setze** $t = jm + i - \lfloor 2\sqrt{p} \rfloor$ und beende die Schleife.

SCHRITT 6: **Setze** $\#E(\mathbb{F}_p) = p + 1 - t$.

Der nun vorliegende randomisierte Algorithmus hat eine erwartete Gesamtlaufzeit von $O(p^{\frac{1}{4}+\epsilon})$ und ist damit in der Praxis viel zu langsam.

6 Der Schoof Algorithmus

6.1 Grundlagen

Ziel dieses Kapitels ist es, den Algorithmus von Schoof aus [Sch85] näher zu erläutern, dieser soll später auch implementiert werden. Bevor wir uns aber dem Algorithmus im Einzelnen zuwenden können, ist es notwendig im Vorfeld ein paar Begriffe, die speziell für das Verständnis dieses Algorithmus notwendig sind, zu erläutern.

Im Schoof Algorithmus sind die Gruppen der l-ten Torsionspunkte ein grundlegender Bestandteil. Sie sind wie folgt definiert:

Definition 6.1 *Sei $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ über $\mathbb{F}_p$. Dann ist*

$$E[l] = \left\{ P \in E(\overline{\mathbb{F}}_p) \mid lP = \mathcal{O} \right\}$$

die Menge der l-Torsionspunkte. Es gilt $E[l] \subseteq E(\overline{\mathbb{F}}_p)$.

Die Menge der l-Torsionspunkte bezeichnet man auch als l-Torsionsgruppe, da sie eine Untergruppe von $E(\overline{\mathbb{F}}_p)$ bildet. Es liegen nach der Definition 6.1 alle Punkte aus $E(\overline{\mathbb{F}}_p)$ in der l-Torsionsgruppe, die die Ordnung l haben.

Um die Punkte der l-Torsionspunkte effizienter berechnen zu können, benötigen wir spezielle Polynome, die so genannten Divisionspolynome.

Definition 6.2 *Sei $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ eine elliptische Kurve über $\mathbb{F}_p$. Dann ist für $n \geq -1$ das n-te Divisionspolynom $\psi_n \in \mathbb{F}_p[x, y]$ durch*

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

rekursiv definiert, wobei a und b die Konstanten der Kurve $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ sind.

Wir verwenden im Folgenden die Schreibweise ψ_n, an Stelle von $\psi_n(x, y)$. Der Zusammenhang zwischen den l-Torsionspunkten und den Divisionspolynomen besteht darin, dass die

Nullstellen der l-ten Divisionspolynome genau die l-Torsionspunkte sind. Man kann die Divisionspolynome über die l-Torsionsgruppen herleiten und erhält dadurch die rekursive Definition 6.2 der Divisionspolynome. Ein Beispiel für die Herleitung des 3. Divisionspolynoms wird in [Mir03, S.33] ausführlich erläutert. Es gilt der folgende Satz:

Satz 6.3 *Sei $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ eine elliptische Kurve über $\mathbb{F}_p$ und sei P ein Punkt auf der Kurve $E(\mathbb{F}_p)$. Dann gilt:*

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

Den Beweis dieses Satzes werden wir hier nicht durchführen, er folgt aus der Definition der Divisionspolynome.

Für Berechnungen mit den Divisionspolynomen ist es wichtig, sie zu vereinfachen. Eine erste Vereinfachung entsteht, wenn die rechte Seite der Kurvengleichung $x^3 + ax + b$ an den Stellen in die Divisionspolynome eingesetzt wird, wo y^2 vorkommt. Man reduziert damit die Anzahl der auftretenden y Faktoren. Der folgende Satz zeigt den sich daraus ergebenden Zusammenhang.

Satz 6.4 *Sei ψ_n das n-te Divisionspolynom, dann kann man alle Vorkommen von y^2 durch die Kurvengleichung $x^3 + ax + b$ ersetzen und erhält*

$$\psi'_n \in \begin{cases} y\,\mathbb{F}_p[x] & \textit{wenn } n \textit{ gerade ist.} \\ \mathbb{F}_p[x] & \textit{wenn } n \textit{ ungerade ist.} \end{cases}$$

Durch Induktion über n kann man mit Hilfe der rekursiven Definition der Divisionspolynome die Korrektheit dieses Satzes zeigen. Auf diesen Beweis verzichten wir an dieser Stelle.

Wie wir gesehen haben, sind die Divisionspolynome abhängig von zwei Variablen x und y. Um die Berechnung so zu vereinfachen, dass die Divisionspolynome nur noch von einer Variable abhängig sind, verwendet man reduzierte Divisionspolynome $f_n(x) \in \mathbb{F}[x]$, welche aus den Divisionspolynomen ψ'_n aus Satz 6.4, durch folgende Definition hervorgehen:

Definition 6.5

$$f_n(x) = \begin{cases} \psi'_n & \textit{wenn } n \textit{ ungerade ist.} \\ \frac{\psi'_n}{2y} & \textit{wenn } n \textit{ gerade ist.} \end{cases}$$

Auch hierbei schreiben wir oft verkürzt f_n anstatt $f_n(x)$. Aus Satz 6.3 wissen wir, dass ein Punkt $P = (x, y)$ genau dann in $E[l]$ liegt, wenn $\psi_l(x, y) = 0$ gilt. Das ist äquivalent zu $f_l(x) = 0$, das heißt also, dass der Punkt P in $E[l]$ liegt, wenn $f_l(x) = 0$ ist. Wir können nun die reduzierten Divisionspolynome durch folgende rekursive Definition aus [BSS00, S.41] auch ohne Zuhilfenahme der nicht reduzierten Divisionspolynome ψ_n berechnen:

Definition 6.6 *Sei $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ über dem Körper $\mathbb{F}_p$ mit $p > 3$ prim. Sei außerdem $F(x) = 4\,(x^3 + ax + b)$. Wir schreiben auch F an Stelle von $F(x)$. Dann können die reduzierten Divisionspolynome f_n durch*

$$f_{-1} = -1, \quad f_0 = 0, \quad f_1 = 1, \quad f_2 = 1, \quad f_3 = \psi_3, \quad f_4 = \frac{\psi_4}{\psi_2},$$

$$f_n = \begin{cases} f_{2m+1} = & \begin{cases} f_{m+2} f_m^3 - F^2 f_{m-1} f_{m+1}^3 & \text{wenn } m \text{ ungerade ist und } m \geq 3, \\ F^2 f_{m+2} f_m^3 - f_{m-1} f_{m+1}^3 & \text{wenn } m \text{ gerade ist und } m \geq 2, \end{cases} \\ f_{2m} = & \left(f_{m+2} f_{m-1}^2 - f_{m-2} f_{m+1}^2\right) f_m \quad \text{wenn } m > 2 \end{cases}$$

beschrieben werden.

Mithilfe der Divisionspolynome kann man auch die skalare Multiplikation eines Punktes durch folgenden Satz errechnen:

Satz 6.7 *Sei $P = (x, y) \in E(\overline{\mathbb{F}}_p)$ und $l \in \mathbb{Z}_{\geq 1}$ mit $nP \neq \mathcal{O}$, dann gilt*

$$nP = \left(x - \frac{\psi_{n-1}\psi_{n+1}}{\psi_n^2}, \frac{\psi_{n+2}\psi_{n-1}^2 - \psi_{n-2}\psi_{n+1}^2}{4y\psi_n^3} \right)$$

Beweis siehe [Lan78].
Diesen Satz werden wir im Verlauf der Betrachtung des Schoof Algorithmus noch häufiger gebrauchen.

Eine sehr wichtige uns häufig benutzte Abbildung ist der Frobenius-Endomorphismus.

Lemma 6.8 *Sei $E(\mathbb{F}_p)$ eine elliptische Kurve über dem Primkörper $\mathbb{F}_p$. Dann ist*

$$\phi : E(\overline{\mathbb{F}}_p) \longrightarrow E(\overline{\mathbb{F}}_p), \quad (x, y) \longmapsto (x^p, y^p)$$

ein Endomorphismus der Gruppe $E(\overline{\mathbb{F}}_p)$. Dieser Endomorphismus wird auch als Frobenius-Endomorphismus bezeichnet.

Beweiß siehe [Mül91, S. 61].
Der folgende Satz stellt die Grundlage des Schoof Algorithmus dar.

Satz 6.9 *Der Frobenius-Endomorphismus ϕ erfüllt im Endomorphismenring $End(E(\overline{\mathbb{F}}_p))$ die Relation*

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

mit $|t| \leq 2\sqrt{p}$. Es gilt für alle Punkte $P \in E(\overline{\mathbb{F}}_p)$

$$\left(\phi^2 - t\phi + p\right)(P) = \mathcal{O}.$$

Dabei sei $i : E(\overline{\mathbb{F}}_p) \to E(\overline{\mathbb{F}}_p)$ die Abbildung, die jedem Punkt $P \in E(\overline{\mathbb{F}}_p)$ den Punkt iP zuordnet.

Den Beweis dieses Satzes kann man in [CR88, S.71] finden.

Das im Satz verwendete t wird in der Literatur auch als *Spur des Frobenius* bezeichnet. Das folgende Lemma soll nun zeigen, dass das hier verwendete t mit dem t aus dem Satz von Hasse 3.4 übereinstimmt.

Lemma 6.10 *Die Zahl t aus Lemma 6.9 und die Zahl t aus dem Satz von Hasse 3.4 mit $\#E(\mathbb{F}_p) = p + 1 - t$ stimmen überein.*

Beweis siehe [Mül91, S. 62].
Um nun die Anzahl der Punkte einer elliptischen Kurve bestimmen zu können, müsste man für alle Punkte einer elliptischen Kurve $E(\overline{\mathbb{F}}_p)$ ein t finden, für das die Gleichung $(\phi^2 - t\phi + p)\,(P) = \mathcal{O}$ erfüllt ist. Da sich die möglichen Werte für t aber in der Größenordnung $O(\sqrt{p})$ befinden, wäre dieses Verfahren zu aufwändig. Man berechnet stattdessen den Wert von t für eine bestimmte Anzahl der ersten aufeinander folgenden Primzahlen. Dabei betrachtet man nicht mehr alle Punkte der Gruppe, sondern nur noch die, die sich in den entsprechenden Torsionsgruppen befinden. Dazu schränken wir den Frobenius ϕ aus Lemma 6.8 und die Relation aus Satz 6.9 auf die l-Torsionsgruppe ein. Wir wissen, dass die l-Torsionsgruppen Untergruppen von $E(\overline{\mathbb{F}}_p)$ sind, deshalb müssen wir nicht alle Punkte der Kurve betrachten, sondern nur die, die auch in der betrachteten l-Torsionsgruppe liegen. Aus einer hinreichend großen Anzahl von betrachteten l-Torsionsgruppen lässt sich die Ordnung von $E(\mathbb{F}_p)$ eindeutig bestimmen.

Definition 6.11 *Sei ϕ der Frobenius und sei $E[l]$ die l-Torsionsgruppe, dann wird ϕ auf die Elemente aus $E[l]$ durch*

$$\phi_l : E[l] \longrightarrow E[l], \quad (x, y) \longmapsto (x^p, y^p)$$

eingeschränkt.

Satz 6.12 *Sei $E[l]$ die l-Torsionsgruppe und ϕ_l der auf $E[l]$ eingeschränkte Frobenius-Endomorphismus. Dann erfüllt auch der eingeschränkte Frobenius-Endomorphismus ϕ_l für alle $P \in E[l]$ die Gleichung*

$$(\phi_l^2 - t_l\phi_l + p)(P) = \mathcal{O}.$$

Hierbei ist $t_l \equiv t \bmod l$.

Beweis siehe [Mir03, S.38].

6.2 Überblick

Nach der Klärung der Grundbegriffe wollen wir die Funktionsweise des Schoof Algorithmus in knapper Form vorstellen, um dem Leser schon mal einen groben Überblick verschaffen zu können. Im Anschluss gehen wir auf die einzelnen Punkte im Detail ein. Zum Zählen der Punkte einer elliptischen Kurve reicht es aus, die Punkte in ihrer affinen Form $y^2 = x^3 + ax + b$ zu zählen. Wir wissen aber, dass es zusätzlich noch den Punkt $\mathcal{O}$ gibt, der nicht durch die affine Darstellung repräsentiert werden kann. Nach dem Satz von Hasse 3.4 wissen wir, dass die Anzahl der Punkte einer elliptischen Kurve durch die Gleichung

$$\#E(\mathbb{F}_p) = p + 1 - t$$

berechnet werden kann. Dabei ist p durch den Körper $\mathbb{F}_p$ gegeben. Die einzige Unbekannte ist t, welche sich aber im Hasse Intervall

$$-2\sqrt{p} \leq t \leq 2\sqrt{p}$$

befinden muss. Um den korrekten Wert für t zu bekommen, wählt man eine bestimmte Anzahl an Untergruppen, den l-Torsionsgruppen, aus und bestimmt für diese den Wert t_l. Dieses t_l ist nichts anderes, als das t der gesamten Gruppe bezogen auf die l-Torsionsgruppe. Es gilt:

$$t_l \equiv t \bmod l$$

Der Wert für l ist dabei immer eine Primzahl. Wenn wir für mehrere l-Torsionsgruppen den Wert t_l bestimmt haben, dann können wir mit dem Chinesischen Restsatz 1.1 das somit entstandene Gleichungssystem lösen. Damit wir eine eindeutige Lösung für t erhalten, darf nur genau eine Lösung des Gleichungssystems

$$\begin{aligned} t &\equiv t_2 \bmod 2, \\ t &\equiv t_3 \bmod 3, \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ t &\equiv t_l \bmod l_n. \end{aligned}$$

im Hasse Intervall $-2\sqrt{p} \leq t \leq 2\sqrt{p}$ liegen. Liegen mehrere Lösungen in dem Intervall, dann ist t nicht mehr eindeutig. Damit nur genau eine Lösung in diesem Intervall liegt, muss folglich das Produkt der Primzahlen $l = 2, 3, \ldots, l_n$ größer als die Länge des Hasse Intervalls sein. Im ersten Schritt berechnen wir die Anzahl der benötigten Primzahlen. Anschließend betrachten wir jede l-Torsionsgruppe und berechnen für sie den Wert t_l. Der Fall $l = 2$ ist gesondert von den anderen zu betrachten, er ist wesentlich einfacher.

Die Fälle in denen $l \geq 3$ ist, verlaufen alle nach demselben Schema. Als Basis dient uns der Satz 6.9. Wir betrachten die Gleichung

$$\left(\phi^2 - t\phi + p\right)(P) = \mathcal{O},$$

die wir in

$$\phi^2(P) + p(P) = t\phi(P)$$

umformen können und überprüfen anschließend, welcher der drei Fälle

$$\begin{aligned} \phi^2(P) &= -p(P) \quad \textbf{Fall A1} \\ \phi^2(P) &= +p(P) \quad \textbf{Fall A2} \\ \phi^2(P) + p(P) &= t\phi(P) \quad \textbf{Fall B} \end{aligned}$$

vorliegt. Eine genauere Betrachtung werden wir an späterer Stelle durchführen, hier soll es uns erst einmal darum gehen, einen Überblick über die Abläufe zu bekommen. Damit man bestimmen kann, welcher Fall vorliegt, benutzt man spezielle Polynomgleichungen, welche aus reduzierten Divisionspolynomen bestehen. Mit diesen und dem für die jeweils betrachtete l-Torsionsgruppe charakteristischen Divisionspolynom, kann man über den größten gemeinsamen Teiler erkennen, welcher Fall vorliegt. Wenn am Ende des Algorithmus die t_l bekannt sind, dann werden wir sie im letzten Schritt mit dem Chinesischen Restsatz zusammen führen und erhalten damit einen Wert für t, der die Gruppenordnung einer elliptischen Kurve eindeutig durch $\#E(\mathbb{F}_p) = p+1-t$ bestimmt. Im Folgenden stellen wir die eben kurz angesprochenen Punkte des Schoof Algorithmus einmal in knapper Form dar.

Algorithmen Skizze

Schritt 1: Berechnung der benötigten Primzahlen $l = 2, 3, 5, \ldots, l_n$. (in Kapitel 6.3.1)
Schritt 2: Berechnung von $t_2 = t \bmod 2$ für den Fall $l = 2$. (in Kapitel 6.3.2)
Schritt 3: Berechnung von $t_l = t \bmod l$ für den Fall $l \geq 3$. (in Kapitel 6.3.3)
Schritt 4: Berechnung von t und $\#E(\mathbb{F}_p) = p + 1 - t$. (in Kapitel 6.3.4)

Aufbauend auf diesen Algorithmus werden wir die einzelnen Schritte Stück für Stück beschreiben und somit zu einer genaueren Betrachtung des Algorithmus kommen.

6.3 Der Schoof Algorithmus im Detail

Wir betrachten eine elliptische Kurve $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$ in affiner Form der Weierstraßgleichung über dem endlichen Körper $\mathbb{F}_p$ mit $p > 3$ prim. Nun wollen wir die Gruppenordnung (Anzahl der Punkte) $\#E(\mathbb{F}_p)$ der Kurve $E(\mathbb{F}_p)$ bestimmen. Dazu dient uns als Ausgangspunkt die Gleichung aus dem Satz von Hasse 3.4 $\#E(\mathbb{F}_p) = p + 1 - t$ mit $|t| \leq 2\sqrt{p}$. Durch diesen Satz wissen wir bereits, dass sich t in dem Intervall

$$-2\sqrt{p} \leq t \leq 2\sqrt{p}$$

befinden muss.

6.3.1 Berechnung der benötigten Primzahlen $l = 2, 3, 5, \ldots, l_n$.

Wir bestimmen als erstes die Menge der Primzahlen, die wir benötigen um am Ende des Algorithmus den Wert für t eindeutig bestimmen zu können. Die größte für die Berechnung relevante Primzahl wird mit l_n bezeichnet.

$$l_n = \min\left\{ l' \;\middle|\; \prod_{\substack{l=2 \\ l \text{ prim}}}^{l'} l > 4\sqrt{p} \right\}$$

Für unsere Berechnungen brauchen wir die Primzahlen $l = 2, 3, 5, \ldots, l_n$. Wir wollen jetzt $t_l \equiv t \bmod l$ für alle $l = 3, 5, \ldots, l_n$ berechnen, dazu müssen wir überprüfen für welches t_l die Gleichung

$$(\phi_l^2 - t_l\phi_l + p)(P) = \mathcal{O}$$

erfüllt ist. Mit $l = 2$ haben wir einen Spezialfall, der gesondert betrachtet werden muss. Das kommt dadurch, dass 2 die einzige gerade Primzahl ist.

6.3.2 Der Fall $l = 2$, Berechnung von $t_2 \equiv t \bmod 2$

Die Bestimmung von $t_2 \equiv t \bmod 2$ ist äquivalent zu $t_2 \equiv \#E(\mathbb{F}_p) \bmod 2$, denn der Satz von Cauchy [Hun74, S. 93] sagt uns, dass die Gruppenordnung genau dann gerade ist, wenn es einen Punkt der Ordnung 2 gibt. Für die weitere Argumentation benötigen wir noch das folgende Lemma.

Lemma 6.13 *Sei $P = (x, y) \in E(\mathbb{F}_p)$ mit $p > 2$. Dann ist genau dann $2P = \mathcal{O}$, wenn $x^3 + ax + b = 0$ Nullstellen in $\mathbb{F}_p$ hat.*

Beweis Den Beweis dieses Lemmas kann man leicht durchführen, dazu nehmen wir an, dass $P = (x, y)$ und der negierte Punkt $-P = (x, -y)$ ist. Es gilt

$$2P = \mathcal{O} \Leftrightarrow P = -P \Leftrightarrow (x, y) = (x, -y) \Leftrightarrow y = -y \Leftrightarrow 2y = 0 \Leftrightarrow y = 0. \qquad \square$$

Es gibt also genau dann einen Punkt der Ordnung 2, wenn $x^3 + ax + b = 0$ eine Nullstelle besitzt. Um zu testen, ob die Gleichung $y^2 = x^3 + ax + b$ eine Nullstelle aufweist, berechnen wir den größten gemeinsamen Teiler von $x^3 + ax + b$ und dem charakteristischen Polynom $x^p - x$ des Körpers $\mathbb{F}_p$. Wenn der Grad des größten gemeinsamen Teilers der beiden Polynome größer Null ist, dann existiert mindestens ein Punkt der Ordnung 2 und damit ist $\#E(\mathbb{F}_p)$ gerade also $t_2 \equiv \#E(\mathbb{F}_p) \equiv 0 \bmod 2$. Wenn der Grad des größten gemeinsamen Teilers aber gleich 0 ist, dann gibt es keine Punkte der Ordnung 2 auf $E(\mathbb{F}_p)$ und damit ist $\#E(\mathbb{F}_p)$ ungerade also $t_2 \equiv \#E(\mathbb{F}_p) \equiv 1 \bmod 2$. Da die Kurve $E(\mathbb{F}_p) : y^2 + x^3 + ax + b$ symmetrisch zur x-Achse liegt und keine Nullstellen hat, gibt es zu jedem Punkt oberhalb der x-Achse einen negativen Punkt unterhalb der x-Achse, mit unterschiedlichen y-Koordinaten. Hinzu kommt noch der Punkt $\mathcal{O}$, der ja bekanntlich nicht auf der Kurve $E(\mathbb{F}_p) : y^2 + x^3 + ax + b$ liegt. In der Summe ergibt sich somit eine ungerade Anzahl von Punkten.

Hier ist noch mal das Wichtigste kurz zusammengefasst.

Für den Fall, dass $l = 2$ ist gilt:

$$t_2 \equiv \begin{cases} 0 \bmod 2 & \text{wenn der Grad von } ggT(x^3 + ax + b, x^p - x) > 0 \text{ ist} \\ 1 \bmod 2 & \text{sonst} \end{cases}$$

Die Laufzeit dieses Schritts beträgt nach [Sch95, S.233] $O(\log^3 p)$.

6.3.3 Der Fall $l \geq 3$, Berechnung von $t_l \equiv t \bmod l$

Nachdem wir den Fall für die einzige gerade Primzahl $l = 2$ abgeschlossen haben, wollen wir nun $t_l \equiv t \bmod l$ für $l \geq 3$ berechnen.

Unsere Aufgabe ist es nun für $l \geq 3$ herauszufinden, für welchen Wert von $t_l \equiv t \bmod l$ die Gleichung

$$(\phi_l^2 - t_l \, \phi_l + p)(P) = \mathcal{O} \tag{12}$$

für alle Punkte P aus der Torsionsgruppe $E[l]$ erfüllt ist. Die Werte für t_l liegen im Bereich $\{0, \ldots, l-1\}$. Ab jetzt reduzieren wir p modulo l und erhalten $p_l \equiv p \bmod l$ mit $p_l \in \{0, \ldots, l-1\}$. Wir schließen hierbei $P = \mathcal{O}$ aus, da die Gleichung (12) in diesem Fall für alle Werte von t_l erfüllt ist. Damit macht die Betrachtung von $P = \mathcal{O}$ keinen Sinn und wird hier von uns vernachlässigt. Wir betrachten somit nur noch die Punkte aus der l-Torsionsgruppe $E[l]^\times = E[l] \setminus \{\mathcal{O}\}$. Die Gleichung (12) kann nun schrittweise aufgelöst

werden.

$$\begin{aligned} & \phi_l^2(P) - t_l\,\phi_l(P) + p_l(P) = \mathcal{O} \\ \Rightarrow \quad & \phi_l^2(P) + p_l(P) = t_l\,\phi_l(P) \end{aligned}$$

Mit $\phi_l(P) = (x^p, y^p)$ und $\phi_l^2(P) = \phi_l^2(x,y) = \phi_l(x^p, y^p) = (x^{p^2}, y^{p^2})$ folgt

$$(x^{p^2}, y^{p^2}) + p_l\,(x,y) = t_l\,(x^p, y^p). \tag{13}$$

Aus Satz 6.7 wissen wir bereits, dass wir eine Punktmultiplikation nP mithilfe der Divisionspolynome errechnen können. Der Term $t_l\,(x^p, y^p)$ aus Gleichung (13) wie folgt ersetzen:

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

Durch das Lemma 1.7 ist es möglich, die Exponenten, die über einer Summe stehen, in die einzelnen Terme hinein zu ziehen, allerdings nur wenn die Exponenten Primzahlpotenzen von p des Körpers $\mathbb{F}_p$ darstellen. Damit erhalten wir

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

Mit diesem Wissen lässt sich die komplette Gleichung (13) zu

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

$$= \begin{cases} \mathcal{O} & \text{wenn } t_l = 0 \text{ ist,} \\ \left(x^p - \left(\frac{\psi_{t_l-1}\psi_{t_l+1}}{\psi_{t_l}^2} \right)^p, \left(\frac{\psi_{t_l+2}\psi_{t_l-1}^2 - \psi_{t_l-2}\psi_{t_l+1}^2}{4y\psi_{t_l}^3} \right)^p \right) = t_l\phi_l(P) & \text{sonst} \end{cases}$$

umformen. Dabei haben wir $p_l(P)$ auch nach Satz 6.7 ersetzt. Wie wir sehen, ergeben sich hierbei zwei mögliche Fälle, die wir überprüfen müssen.

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

Fall B: $\phi^2(P) \neq \pm\, p_l(P)$

Sei im Folgenden $R = x^3 + ax + b$. Um nun zu überprüfen, welcher der beiden Fälle vorliegt, betrachten wir folgendes: Wenn wir uns im Fall A befinden würden, dann hätten $\phi^2(P)$ und $p_l(P)$ dieselbe x-Koordinate. Es müsste also gelten:

$$x^{p^2} = x - \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2} \tag{14}$$

Um besser mit der Gleichung (14) arbeiten zu können, stellen wir sie so um, dass die

rechte Seite zu Null wird.

$$\begin{aligned} & x^{p^2} = x - \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2} \\ \Rightarrow \quad & \left(x^{p^2} - x\right)\psi_{p_l}^2 = -\psi_{p_l-1}\psi_{p_l+1} \\ \Rightarrow \quad & \left(x^{p^2} - x\right)\psi_{p_l}^2 + \psi_{p_l-1}\psi_{p_l+1} = 0 \end{aligned}$$

Aus Definition 6.5 wissen wir, dass wir die Divisionspolynome ψ_n durch die reduzierten Divisionspolynome f_n ausdrücken können. Wir wenden

$$\psi_n = \begin{cases} 2yf_n & \text{wenn } n \text{ gerade ist,} \\ f_n & \text{wenn } n \text{ ungerade ist} \end{cases}$$

als Ersetzungsregel an. Durch die eben vorgenommene Fallunterscheidung der reduzierten Divisionspolynome müssen wir die beiden möglichen Fälle für gerade und ungerade p_l einzeln betrachten. Wir erhalten:

$$\left(x^{p^2} - x\right) 4Rf_{p_l}^2 + f_{p_l-1}f_{p_l+1} = 0 \quad \text{falls } p_l \text{ gerade ist,} \tag{15}$$

$$\left(x^{p^2} - x\right) f_{p_l}^2 + 4Rf_{p_l-1}f_{p_l+1} = 0 \quad \text{falls } p_l \text{ ungerade ist.} \tag{16}$$

Je nachdem, ob p_l gerade oder ungerade ist, benutzen wir entweder die Gleichung (15) oder (16). Das sich daraus ergebende Polynom nennen wir $g(x)$. Wir haben oben bereits festgelegt, dass $R = x^3 + ax + b$ die rechte Seite der Kurvengleichung $y^2 = x^3 + ax + b$ ist. Nun entstehen beim Einsetzen der reduzierten Divisionspolynome aber bei f_n mit n gerade zusätzliche y, diese fallen aber durch das Einsetzen der Gleichung R weg, da sich hierbei nur gerade Potenzen von y ergeben. Die entstandenen Polynome sind somit genau wie die reduzierten Divisionspolynome nur noch von einer Variablen abhängig.

Damit wir entscheiden können, ob wir uns im Fall A oder im Fall B befinden, berechnen wir den größten gemeinsamen Teiler von $g(x)$ und dem l-ten Divisionspolynom, welches die l-Torsionspunkte charakterisiert. Da l eine ungerade Primzahl ist, können wir das l-te Divisionspolynom durch das l-te reduzierte Divisionspolynom ersetzen, ohne zusätzliche y zu erhalten. Wir haben auch hier ein Polynom, welchen nur von einer Variablen abhängt. Ist nun der Grad des größten gemeinsamen Teilers von $g(x)$ und f_l größer 0, so befinden wir uns im Fall A, ansonsten im Fall B. Auch an dieser Stelle wollen wir der Übersichtlichkeit halber das Wichtigste noch einmal kompakt zusammenfassen.

Wir berechnen zuerst das Polynom $g(x)$, welches wie folgt dargestellt wird:

$$g(x) = \begin{cases} \left(x^{p^2} - x\right) 4Rf_{p_l}^2 + f_{p_l-1}f_{p_l+1} = 0 & \text{falls } p_l \text{ gerade ist,} \\ \left(x^{p^2} - x\right) f_{p_l}^2 + 4Rf_{p_l-1}f_{p_l+1} = 0 & \text{falls } p_l \text{ ungerade ist.} \end{cases}$$

Außerdem wird der Grad des $ggT\left(g(x), f_l\right)$ berechnet.

$$\text{Grad}\left(ggT\left(g(x), f_l\right)\right) \begin{cases} > 0 & \text{dann befinden wir uns im } \textbf{Fall A}\text{: } \phi^2(P) = \pm\, p_l(P) \\ = 0 & \text{dann befinden wir uns im } \textbf{Fall B}\text{: } \phi^2(P) \neq \pm\, p_l(P) \end{cases}$$

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

Nehmen wir einmal an, dass $\phi^2(P) = \pm\, p_l(P)$ gilt, dann müssen wir weiterhin entscheiden, welcher der beiden Unterfälle

$$\textbf{Fall A1: } \phi^2(P) = +\, p_l(P),$$
$$\textbf{Fall A2: } \phi^2(P) = -\, p_l(P)$$

vorliegt. Dazu nehmen wir uns noch einmal unsere Ausgangsgleichung

$$\phi_l^2(P) + p_l(P) = t_l\, \phi_l(P). \tag{17}$$

Angenommen $\phi^2(P) = p_l(P)$, dann kann die die Gleichung (17) wie folgt umgeformt werden:

$$\begin{aligned} & \phi_l^2(P) + p_l(P) = t_l\, \phi_l(P) \\ \Rightarrow\quad & p_l(P) + p_l(P) = t_l\, \phi_l(P), \qquad \text{da } \phi^2(P) = p_l(P) \\ \Rightarrow\quad & 2p_l(P) = t_l\, \phi_l(P) \\ \Rightarrow\quad & 2\frac{p_l}{t_l}(P) = \phi_l(P) \end{aligned} \tag{18}$$

Da $p_l \neq 0$ und $P \neq \mathcal{O}$ muss auch $t_l \neq 0$ sein, da sich sonst ein Widerspruch ergeben würde. Wenn wir die eben erhaltene Gleichung (18) nun Quadrieren erhalten wir:

$$\begin{aligned} & \left(2\frac{p_l}{t_l}(P)\right)^2 = \phi_l^2(P) \\ \Rightarrow\quad & 4\frac{p_l^2}{t_l^2}(P) = \phi_l^2(P). \end{aligned} \tag{19}$$

Es kann erneut $\phi^2(P) = p_l(P)$ in die Gleichung (19) eingesetzt werden. Daraus ergibt sich:

$$\begin{aligned} & 4\frac{p_l^2}{t_l^2}(P) = p_l(P) \\ \Rightarrow \quad & 4\frac{p_l}{t_l^2}\, p_l(P) = p_l(P) \\ \Rightarrow \quad & 4\frac{p_l}{t_l^2} = 1 \\ \Rightarrow \quad & t_l^2 = 4\, p_l \bmod l. \end{aligned} \tag{20}$$

Wenn der Fall A1 vorliegt, muss nach der Gleichung (20) $4\,p_l$ ein Quadrat modulo l sein und es somit zwei Werte für t_l geben für die die Gleichung $t_l^2 = 4\,p_l \bmod l$ erfüllt ist. Wenn $4\,p_l$ aber kein Quadrat modulo l ist, befinden wir uns im Fall A2. Mit dem Legendre - Symbol aus Kapitel 1.4 können wir testen, ob in der vorliegenden Betrachtung ein Quadrat vorliegt, oder nicht. Ergibt das Legendre - Symbol

$$\left(\frac{4\,p_l}{l}\right) = 1,$$

dann ist $4\,p_l$ also ein Quadrat und Fall A1 tritt ein. Mit Hilfe der Umformungsregeln für das Legendre - Symbol ergibt sich

$$\left(\frac{4\,p_l}{l}\right) = \left(\frac{4}{l}\right)\left(\frac{p_l}{l}\right) = \left(\frac{2}{l}\right)\left(\frac{2}{l}\right)\left(\frac{p_l}{l}\right).$$

Da $\left(\frac{2}{l}\right) = 1$ oder $\left(\frac{2}{l}\right) = -1$ folgt $\left(\frac{4}{l}\right) = 1$ in jedem Fall. Es bleibt also nur noch die Frage zu klären, ob p_l ein Quadrat modulo l ist. Wenn dies der Fall ist, gibt es auch ein t_l, welches die Gleichung $t_l^2 = 4\,p_l \bmod l$ erfüllt und damit gilt dann auch $\phi^2(P) = +\,p_l(P)$ und wir befinden uns im Fall A1, andernfalls im Fall A2. Zusammengefasst ergibt sich für die Fallunterscheidung zwischen A1 und A2:

Entscheidung im **Fall A** ($\phi^2(P) = \pm\, p_l(P)$) über die Unterfälle **A1** und **A2**:

$$\left(\frac{p_l}{l}\right) = \begin{cases} 1 & \text{dann befinden wir uns im } \textbf{Fall A1}\text{: } \phi^2(P) = +\,p_l(P), \\ -1 & \text{dann befinden wir uns im } \textbf{Fall A2}\text{: } \phi^2(P) = -\,p_l(P). \end{cases}$$

Fall A1: $\phi^2(P) = +\,p_l(P)$

Im Fall A1 wissen wir bereits, dass die Gleichung

$$t_l^2 = 4\,p_l \bmod l$$

eine Lösung für t_l hat, da p_l ein Quadrat modulo l ist. Wir müssen nun eine Quadratwurzel $w \in \{0, \ldots, l-1\}$ von p_l modulo l mit $w^2 = p_l \bmod l$ finden. Es gilt damit:

$$\begin{aligned} & \phi^2(P) = p_l\,(P) \\ \Rightarrow \quad & \phi^2(P) = w^2\,(P) \end{aligned} \tag{21}$$

Die Gleichung (21) ist äquivalent zu

$$\phi(P) = \pm\, w\,(P). \tag{22}$$

Es muss nun überprüft werden, ob ein Punkt $P \in E[l]^\times$ existiert, für den Gleichung (22) erfüllt ist. Dazu vergleichen wir die x-Koordinaten von $\phi(P)$ und $w\,(P)$, denn wenn diese nicht gleich sind, dann kann es auch einen solchen Punkt nicht geben. Dabei verwenden wir wieder den Satz 6.7, der es uns erlaubt die Multiplikation mit Hilfe von Divisionspolynomen darzustellen. Da wir dieses Verfahren weiter oben im Schoof Algorithmus schon einmal angewendet haben, werden wir hier nicht noch einmal die ganze Herleitung beschreiben.

$$\begin{aligned} x^p &= x - \frac{\psi_{w-1}\psi_{w+1}}{\psi_w^2} \\ (x^p - x)\,\psi_w^2 + \psi_{w-1}\psi_{w+1} &= 0 \end{aligned}$$

Auch hierbei arbeiten wir mit den reduzierten Divisionspolynomen. Es ergeben sich erneut zwei Fallunterscheidungen in Abhängigkeit von w:

$$(x^p - x)\,4Rf_w^2 + f_{w-1}f_{w+1} = 0 \quad \text{falls } w \text{ gerade ist,} \tag{23}$$

$$(x^p - x)\,f_w^2 + 4Rf_{w-1}f_{w+1} = 0 \quad \text{falls } w \text{ ungerade ist.} \tag{24}$$

Ist w gerade benutzen wie das Polynom aus der Gleichung (23) ansonsten das aus Gleichung (24). Wir nennen das jeweils benutzte Polynom $h(x)$. Wenn nun der Grad des größten gemeinsamen Teilers von $h(x)$ und dem für die l-Torsionspunkte charakteristischen reduzierten Divisionspolynom f_l gleich Null ist, dann gibt es keinen Punkt $P \in E[l]^\times$, für den die Gleichung (22) erfüllt ist. Damit kommen wir in den Sonderfall A12. Für diesen gilt $\phi^2(P) = -\,p_l(P)$ er ist damit genauso wie der Fall A2 zu behandeln. Wenn aber der Grad des größten gemeinsamen Teilers von $h(x)$ und f_l größer als Null ist, dann gibt es mindestens einen Punkt für den $\phi(P) = \pm\, w\,(P)$ gilt. Ob jetzt $\phi(P) = +\,w\,(P)$ oder $\phi(P) = -\,w\,(P)$ gilt, können wir durch Vergleich der y-Koordinaten von $\phi(P)$ und $w\,(P)$

feststellen.

$$\begin{aligned} y^p &= \frac{\psi_{w+2}\psi_{w-1}^2 - \psi_{w-2}\psi_{w+1}^2}{4y\psi_w^3} \\ 4y^{p+1}\psi_w^3 &= \psi_{w+2}\psi_{w-1}^2 - \psi_{w-2}\psi_{w+1}^2 \\ 4y^{p+1}\psi_w^3 - \psi_{w+2}\psi_{w-1}^2 + \psi_{w-2}\psi_{w+1}^2 &= 0 \\ 4\left(y^2\right)^{\frac{p+1}{2}}\psi_w^3 - \psi_{w+2}\psi_{w-1}^2 + \psi_{w-2}\psi_{w+1}^2 &= 0 \\ 4R^{\frac{p+1}{2}}\psi_w^3 - \psi_{w+2}\psi_{w-1}^2 + \psi_{w-2}\psi_{w+1}^2 &= 0 \end{aligned}$$

Dabei haben wir bereits die y mit geraden Potenzen durch die Kurvengleichung R ersetzt. Nachdem wir die reduzierten Divisionspolynome eingesetzt haben, unterscheiden wir zwei Fälle für w gerade und w ungerade. Für w gerade erhalten wir

$$\begin{aligned} & 4R^{\frac{p+1}{2}}(2yf_w)^3 - 2yf_{w+2}f_{w-1}^2 + 2yf_{w-2}f_{w+1}^2 = 0 \\ \Rightarrow\ & 2y\left(4R^{\frac{p+1}{2}}4Rf_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0 \\ \Rightarrow\ & 2y\left(16R^{\frac{p+1}{2}+1}f_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0 \\ \Rightarrow\ & 2y\left(16R^{\frac{p+3}{2}}f_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0 \end{aligned} \tag{25}$$

und für w ungerade ergibt sich

$$\begin{aligned} & 4R^{\frac{p+1}{2}}f_w^3 - f_{w+2}(2yf_{w-1})^2 + f_{w-2}(2yf_{w+1})^2 = 0 \\ \Rightarrow\ & 4R^{\frac{p+1}{2}}f_w^3 - 4Rf_{w+2}f_{w-1}^2 + 4Rf_{w-2}f_{w+1}^2 = 0 \\ \Rightarrow\ & 4R\left(R^{\frac{p+1}{2}-1}f_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0 \\ \Rightarrow\ & 4R\left(R^{\frac{p-1}{2}}f_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0. \end{aligned} \tag{26}$$

Für unsere Berechnung benutzen wir, abhängig von w entweder die Gleichung (25) oder die Gleichung (26). Wir das verwendete Polynom kurz k, welches entweder ein Polynom aus $y\mathbb{F}_p[x]$ oder aus $\mathbb{F}_p[x]$ ist.

Wenn nun der Grad des größten gemeinsamen Teilers von k und f_l größer als Null ist, wissen wir, dass $\phi(P) = +w(P)$ gilt. Ist der Grad aber gleich Null so gilt $\phi(P) = -w(P)$. Da $w^2 = p_l \bmod l$ ergibt, können wir die Quadratwurzel w von p_l bestimmen. Den Wert für t_l können wir durch die eben schon besprochene Gleichung (20) berechnen. Wenn der Grad des größten gemeinsamen Teilers von k und f_l größer als Null ist, nehmen wir den

positiven Wert $+w$ und damit ist $t_l = +2\,w \bmod l$. Ist aber der Grad gleich Null, so nehmen wir den negativen Wert $-w$ und erhalten $t_l = -2\,w \bmod l$.

Im **Fall A1** ($\phi^2(P) = +\,p_l(P)$) gilt: Es wird das Polynom

$$h(x) = \begin{cases} (x^{\,p} - x)\,4Rf_w^2 + f_{w-1}f_{w+1} = 0 & \text{falls } w \text{ gerade ist,} \\ (x^{\,p} - x)\,f_w^2 + 4Rf_{w-1}f_{w+1} = 0 & \text{falls } w \text{ ungerade ist} \end{cases}$$

berechnet.

$$\text{Grad}\,(ggT\,(h(x), f_l)) \begin{cases} > 0 & \text{dann existiert ein Punkt } P \in E[l]^\times \text{ für den gilt: } \phi^2(P) = +\,p_l(P) \\ = 0 & \text{dann befinden wir uns im } \textbf{Fall A12}\text{: } \phi^2(P) = -\,p_l(P) \end{cases}$$

Wenn $\text{Grad}\,(ggT\,(h(x), f_l)) > 0$, dann wird das Polynom k berechnet

$$k = \begin{cases} 2y\left(16R^{\frac{p+3}{2}}f_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0 & \text{falls } w \text{ gerade ist,} \\ 4R\left(R^{\frac{p-1}{2}}f_w^3 - f_{w+2}f_{w-1}^2 + f_{w-2}f_{w+1}^2\right) = 0 & \text{falls } w \text{ ungerade ist} \end{cases}$$

Es gilt $w^2 \equiv p_l \bmod l$.

$$\text{Grad}\,(ggT\,(k, f_l)) \begin{cases} > 0 & \text{dann ist } t_l \equiv +2\,w \bmod p, \\ = 0 & \text{dann ist } t_l \equiv -2\,w \bmod p \end{cases}$$

Fall A2: $\phi^2(P) = -\,p_l(P)$

Die Betrachtung dieses Falles ist recht leicht und lässt sich daher ohne großen Rechenaufwand abschließen. Aus $\phi^2(P) = -\,p_l(P)$ ergibt sich:

$$\phi_l^2(P) + p_l(P) = \mathcal{O} = t_l\,\phi_l(P).$$

Da $\phi_l(P) \neq \mathcal{O}$ ist, gilt die Gleichung $t_l\,\phi_l(P) = \mathcal{O}$ nur für $t_l = 0 \bmod l$ und damit ist der Fall abgeschloßen.

Im **Fall A2** ($\phi^2(P) = -\,p_l(P)$) gilt:

$$t_l \equiv 0 \bmod l.$$

Fall B: $\phi^2(P) \neq \pm\,p_l(P)$

Im Fall B wissen wir bereits, dass $t_l \neq 0 \bmod l$ ist, denn sonst wären wir im Fall A. Damit kann t_l nur noch aus der Menge $\{1, \ldots, l-1\}$ sein. Nun müssen wir die gesamte Gleichung

$$\phi_l^2(P) + p_l(P) = t_l\,\phi_l(P) \tag{27}$$

betrachten. Wir haben bereits weiter oben gezeigt, wie man die Gleichung (27) durch die Divisionspolynome ausdrücken kann. Unser Ziel ist es die Gleichung (27) mit den eingesetzten Divisionspolynomen so vereinfachen, dass wir gut mit ihr arbeiten können. Zunächst berechnen wir die linke Seite der Gleichung (27). Nachdem wir die Divisionspolynome eingesetzt haben ergibt sich

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

Wir werden nun die uns vorliegende Addition mit Hilfe des Satzes 3.1 durchführen, wobei uns bekannt ist, dass $\phi^2(P) \neq \pm p_l(P)$ ist. Es ergibt sich

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

$$= \left(\lambda^2 - x^{p^2} - x + \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2}, \lambda\left(2x^{p^2} - \lambda^2 + x - \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2}\right) - y^{p^2}\right) \tag{28}$$

mit

$$\begin{aligned}
\lambda &= \frac{y^{p^2} - \frac{\psi_{p_l+2}\psi_{p_l-1}^2 + \psi_{p_l-2}\psi_{p_l+1}^2}{4y\psi_{p_l}^3}}{x^{p^2} - x + \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2}} \\
&= \frac{4y^{p^2+1}\psi_{p_l}^3 - \psi_{p_l+2}\psi_{p_l-1}^2 + \psi_{p_l-2}\psi_{p_l+1}^2}{4y\psi_{p_l}^3\left(\frac{\left(x^{p^2}-x\right)\psi_{p_l}^2 + \psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2}\right)} \\
&= \frac{4y^{p^2+1}\psi_{p_l}^3 - \psi_{p_l+2}\psi_{p_l-1}^2 + \psi_{p_l-2}\psi_{p_l+1}^2}{4y\psi_{p_l}\left(\left(x^{p^2} - x\right)\psi_{p_l}^2 + \psi_{p_l-1}\psi_{p_l+1}\right)} \\
&= \frac{\psi_{p_l+2}\psi_{p_l-1}^2 - \psi_{p_l-2}\psi_{p_l+1}^2 - 4y^{p^2+1}\psi_{p_l}^3}{4y\psi_{p_l}\left(\left(x - x^{p^2}\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right)}.
\end{aligned} \tag{29}$$

Die rechte Seite der Gleichung (27) lässt sich durch

$$t_l\phi_l(P) = \left(x^p - \left(\frac{\psi_{t_l-1}\psi_{t_l+1}}{\psi_{t_l}^2}\right)^p, \left(\frac{\psi_{t_l+2}\psi_{t_l-1}^2 - \psi_{t_l-2}\psi_{t_l+1}^2}{4y\psi_{t_l}^3}\right)^p\right) \tag{30}$$

berechnen. Damit wir testen können für welches t_l die Gleichung (27) erfüllt ist, müssen wir einen konkreten Wert für t_l in die eben errechnete linke (28) und rechte Seite (30) der Gleichung (27) einsetzen und anschließend die x-Koordinaten vergleichen. Der Ausgangspunkt dieses Vergleiches stellt sich durch

$$\lambda^2 - x^{p^2} - x + \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2} = x^p - \frac{\psi_{t_l-1}^p\psi_{t_l+1}^p}{\psi_{t_l}^{2p}} \tag{31}$$

dar. Wie wollen nun die Gleichung (31) so umstellen, dass auf einer Seite eine Null steht und alle Brüche verschwinden. Wir setzen zunächst $\lambda = \frac{\alpha}{\beta}$. Dabei entspricht α dem Zähler und β dem Nenner des eben errechneten λ aus Gleichung (29). Durch Umformung erhalten wir

$$
\begin{aligned}
& -\frac{\alpha^2}{\beta^2} + x^{p^2} + x^p + x - \frac{\psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2} = \frac{\psi_{t_l-1}^p \psi_{t_l+1}^p}{\psi_{t_l}^{2p}} \\
\Rightarrow \quad & -\frac{\alpha^2}{\beta^2} + \frac{\left(x^{p^2} + x^p + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2} = \frac{\psi_{t_l-1}^p \psi_{t_l+1}^p}{\psi_{t_l}^{2p}} \\
\Rightarrow \quad & \frac{-\alpha^2\psi_{p_l}^2 + \left(\left(x^{p^2} + x^p + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right)\beta^2}{\psi_{p_l}^2\beta^2} = \frac{\psi_{t_l-1}^p \psi_{t_l+1}^p}{\psi_{t_l}^{2p}} \\
\Rightarrow \quad & \left(\left(\left(x^{p^2} + x^p + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right)\beta^2 - \alpha^2\psi_{p_l}^2\right)\psi_{t_l}^{2p} = \psi_{t_l-1}^p \psi_{t_l+1}^p \beta^2 \psi_{p_l}^2 \\
\Rightarrow \quad & \left(\left(-\psi_{p_l-1}\psi_{p_l+1} + \psi_{p_l}^2\left(x^{p^2} + x^p + x\right)\right)\beta^2 - \psi_{p_l}^2\alpha^2\right)\psi_{t_l}^{2p} - \psi_{t_l-1}^p \psi_{t_l+1}^p \psi_{p_l}^2 \beta^2 = 0 \\
\Rightarrow \quad & \left(\left(\psi_{p_l-1}\psi_{p_l+1} - \psi_{p_l}^2\left(x^{p^2} + x^p + x\right)\right)\beta^2 + \psi_{p_l}^2\alpha^2\right)\psi_{t_l}^{2p} + \psi_{t_l-1}^p \psi_{t_l+1}^p \psi_{p_l}^2 \beta^2 = 0
\end{aligned}
$$

mit

$$
\begin{aligned}
\alpha &= \psi_{p_l+2}\psi_{p_l-1}^2 - \psi_{p_l-2}\psi_{p_l+1}^2 - 4y^{p^2+1}\psi_{p_l}^3 \quad \text{und} \\
\beta &= 4y\psi_{p_l}\left(\left(x - x^{p^2}\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right).
\end{aligned}
$$

Nebenbei bemerkt wurden in diesem Schritt die meisten Umformungsfehler in der Literatur gemacht, diese haben wir hier selbstverständlich behoben. Auch hier ersetzen wir wieder die Divisionspolynome durch die reduzierten Divisionspolynome. Dies führt uns zu vier Fallunterscheidungen, die sich aus geraden und ungeraden p_l und geraden und ungeraden t_l ergeben. Wir beginnen mit den Fall, dass p_l und t_l gerade sind:

$$
\begin{aligned}
\Big(\Big(f_{p_l-1}f_{p_l+1} - 4Rf_{p_l}^2\left(x^{p^2} + x^p + x\right)\Big)\beta^2 + 16R^2 f_{p_l}^2\alpha^2\Big)4^p R^p f_{t_l}^{2p} & \\
+4Rf_{t_l-1}^p f_{t_l+1}^p f_{p_l}^2\beta^2 = 0 & \qquad (32)
\end{aligned}
$$

mit

$$
\alpha = f_{p_l+2}f_{p_l-1}^2 - f_{p_l-2}f_{p_l+1}^2 - 16R^{\frac{p^2+3}{2}}f_{p_l}^3
$$

und

$$
\beta = 8Rf_{p_l}\left(\left(x - x^{p^2}\right)4Rf_{p_l}^2 - f_{p_l-1}f_{p_l+1}\right).
$$

In diesem Fall entsteht nach dem Ersetzen der Divisionspolynome durch die reduzierten Divisionspolynome in α ein $2y$ Faktor. Da aber in Gleichung (32) mit α^2 gerechnet wird, ziehen wir den $2y$ Faktor aus α heraus und ergänzen die Gleichung (32) an der entsprechenden Stelle um $(2y)^2 = 4R$. Damit erhalten wir eine Formel ohne y.

Im Fall p_l gerade und t_l ungerade berechnen sich α und β wie im vorherigen Fall, da sie nur von p_l abhängig sind und wir dieses hier nicht verändern. Der restliche Teil der Formel berechnet sich wie folgt:

$$\left(\left(f_{p_l-1}f_{p_l+1} - 4Rf_{p_l}^2\left(x^{p^2} + x^p + x\right)\right)\beta^2 + 16R^2 f_{p_l}^2\alpha^2\right) f_{t_l}^{2p} + 4^{p+1}R^{p+1}f_{t_l-1}^p f_{t_l+1}^p f_{p_l}^2\beta^2 = 0 \tag{33}$$

Genau wie im vorherigen Fall wird auch hier der $2y$ Faktor aus α in die Gleichung (33) gezogen.

Wenden wir uns nun dem Fall p_l ungerade und t_l gerade zu.

$$\left(\left(4Rf_{p_l-1}f_{p_l+1} - f_{p_l}^2\left(x^{p^2} + x^p + x\right)\right)16R\beta^2 + f_{p_l}^2\alpha^2\right) 4^p R^p f_{t_l}^{2p} + 16Rf_{t_l-1}^p f_{t_l+1}^p f_{p_l}^2\beta^2 = 0 \tag{34}$$

mit

$$\alpha = 4R\left(f_{p_l+2}f_{p_l-1}^2 - f_{p_l-2}f_{p_l+1}^2 - R^{\frac{p^2-1}{2}}f_{p_l}^3\right)$$

und

$$\beta = f_{p_l}\left(\left(x - x^{p^2}\right)f_{p_l}^2 - 4Rf_{p_l-1}f_{p_l+1}\right).$$

Hierbei bleibt nach den Ersetzen der Divisionspolynome durch die reduzierten Divisionspolynome bei β ein $4y$ Faktor bestehen. Hier ziehen wir $4y$ in die Gleichung (34). Da wir dort mit β^2 rechnen, wird aus $(4y)^2 = 16R$.

Als letztes untersuchen wir den Fall p_l und t_l ungerade. Dabei bleiben, wie im vorherigen Fall für p_l ungerade und t_l gerade α und β gleich, da sie, wie schon erwähnt, nur von p_l abhängig sind. Der Unterschied zum vorherigen Fall besteht darin, dass t_l, bei unverändertem p_l, ungerade ist.

$$\left(\left(4Rf_{p_l-1}f_{p_l+1} - f_{p_l}^2\left(x^{p^2} + x^p + x\right)\right)16R\beta^2 + f_{p_l}^2\alpha^2\right) f_{t_l}^{2p} + 4^{p+2}R^{p+1}f_{t_l-1}^p f_{t_l+1}^p f_{p_l}^2\beta^2 = 0 \tag{35}$$

Auch hierbei wurde der $4y$ Faktor aus β in die Gleichung (35) gezogen.

Je nach Fall benutzen wir entweder die Gleichungen (32), (33), (34) oder (35). Die jeweils benutzte Gleichung bezeichnen wir mit D_x. Damit wir überprüfen können, ob die

Gleichung (27) für einen konkreten Wert von t_l erfüllt ist, setzen wir einen Wert für t_l aus dem Intervall $\{1, \ldots, l-1\}$ in die jeweilige Gleichung (32), (33), (34) oder (35) ein. Das Intervall sollte dabei von vorne nach hinten durchlaufen werden. Man beginnt also mit $t_l = 1$ und inkrementiert den Wert jeweils um eins bis das passende t_l gefunden ist.

Wir berechnen den Grad des größten gemeinsamen Teilers von D_x und f_l. Ist der Grad gleich Null, wissen wir, dass unsere Wahl für t_l falsch war und wir beginnen den Test erneut mit einem anderen noch nicht gewählten Wert für t_l. Ergibt sich für den Grad des größten gemeinsamen Teilers von D_x und f_l ein Wert größer als Null, müssen wir auch noch die y-Koordinate der linken und rechten Seite von Gleichung (27) vergleichen. Für λ ergibt sich

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

was wir zuvor schon berechnet haben. Nun setzen wir y der linken (28) und rechten Seite (30) der Gleichung (27) gleich und formen den Ausdruck so um, dass wir wieder auf einer Seite eine Null stehen haben und der Term keine Brüche enthält. Dabei ist auch hier $\lambda = \frac{\alpha}{\beta}$.

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

$$\Rightarrow \quad \alpha\left(-\frac{\alpha^2}{\beta^2} + \frac{\left(2x^{p^2} + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}}{\psi_{p_l}^2}\right) - \beta y^{p^2} = \frac{\beta\left(\psi_{t_l+2}\psi_{t_l-1}^2 - \psi_{t_l-2}\psi_{t_l+1}^2\right)^p}{4^p y^p \psi_{t_l}^{3p}}$$

$$\Rightarrow \quad -\frac{\alpha^3}{\beta^2} + \frac{\alpha\left(\left(2x^{p^2} + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right)}{\psi_{p_l}^2} - \beta y^{p^2} = \frac{\beta\left(\psi_{t_l+2}\psi_{t_l-1}^2 - \psi_{t_l-2}\psi_{t_l+1}^2\right)^p}{4^p y^p \psi_{t_l}^{3p}}$$

$$\Rightarrow \quad \alpha\beta^2\left(\left(2x^{p^2} + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right) - \beta^3 y^{p^2}\psi_{p_l}^2 - \alpha^3\psi_{p_l}^2 = \frac{\beta^3\psi_{p_l}^2\left(\psi_{t_l+2}\psi_{t_l-1}^2 - \psi_{t_l-2}\psi_{t_l+1}^2\right)^p}{4^p y^p \psi_{t_l}^{3p}}$$

$$\Rightarrow \quad 4^p y^p \psi_{t_l}^{3p}\left(\alpha\beta^2\left(\left(2x^{p^2} + x\right)\psi_{p_l}^2 - \psi_{p_l-1}\psi_{p_l+1}\right) - \beta^3 y^{p^2}\psi_{p_l}^2 - \alpha^3\psi_{p_l}^2\right) - \beta^3\psi_{p_l}^2\left(\psi_{t_l+2}\psi_{t_l-1}^2 - \psi_{t_l-2}\psi_{t_l+1}^2\right)^p = 0 \tag{36}$$

Die erhaltene Gleichung (36) wandeln wir erneut durch einsetzen der reduzierten Divisionspolynome um und erhalten vier mögliche Fälle. Hierbei können wir α und β aus den vier zuvor behandelten Fällen übernehmen. Wir müssen aber beachten, dass wir im Fall p_l gerade den entstandenen $2y$ Faktor aus α und im Fall p_l ungerade, den $4y$ Faktor aus β

herausgezogen haben. Wir dürfen diese Terme also bei unseren folgenden Berechnungen nicht außer acht lassen. Beginnen wir erneut mit dem Fall p_l und t_l gerade, dann ergibt sich:

$$\begin{aligned}
& 4^p y^p 2^{3p} y^{3p} f_{t_l}^{3p} \left(2y\alpha\beta^2 \left(\left(2x^{p^2}+x\right) 4Rf_{p_l}^2 - f_{p_l-1}f_{p_l+1}\right) - 4R\beta^3 y^{p^2} f_{p_l}^2 - 8Ry\alpha^3 4Rf_{p_l}^2\right) \\
& -\beta^3 4Rf_{p_l}^2 \left(2yf_{t_l+2}f_{t_l-1}^2 - 2yf_{t_l-2}f_{t_l+1}^2\right)^p = 0 \\
\Rightarrow\ & 2^{5p} y^{4p} f_{t_l}^{3p} \left(2y\alpha\beta^2 \left(\left(2x^{p^2}+x\right) 4Rf_{p_l}^2 - f_{p_l-1}f_{p_l+1}\right) - 4R\beta^3 y^{p^2} f_{p_l}^2 - 32R^2 y\alpha^3 f_{p_l}^2\right) \\
& -2^{p+2} R\beta^3 f_{p_l}^2 y^p \left(f_{t_l+2}f_{t_l-1}^2 - f_{t_l-2}f_{t_l+1}^2\right)^p = 0 \\
\Rightarrow\ & y\Bigg(2^{5p+1} R^{2p} f_{t_l}^{3p} \left(\alpha\beta^2 \left(\left(2x^{p^2}+x\right) 4Rf_{p_l}^2 - f_{p_l-1}f_{p_l+1}\right) - 2R^{\frac{p^2+1}{2}} \beta^3 f_{p_l}^2 - 16R^2\alpha^3 f_{p_l}^2\right) \\
& -2^{p+2} R^{\frac{p+1}{2}} \beta^3 f_{p_l}^2 \left(f_{t_l+2}f_{t_l-1}^2 - f_{t_l-2}f_{t_l+1}^2\right)^p\Bigg) = 0 \qquad (37)
\end{aligned}$$

Wir sehen, dass die entstandene Gleichung (37) nur noch ein y beinhaltet und damit aus $y\mathbb{F}_p[x]$ ist. In zweiten Fall lassen wir p_l erneut gerade sein und nehmen an, t_l sei ungerade.

$$\begin{aligned}
& 4^p y^p f_{t_l}^{3p} \left(2y\alpha\beta^2 \left(\left(2x^{p^2}+x\right) 4Rf_{p_l}^2 - f_{p_l-1}f_{p_l+1}\right) - \beta^3 y^{p^2} 4Rf_{p_l}^2 - 8Ry\alpha^3 4Rf_{p_l}^2\right) \\
& -4R\beta^3 f_{p_l}^2 \left(4Rf_{t_l+2}f_{t_l-1}^2 - 4Rf_{t_l-2}f_{t_l+1}^2\right)^p = 0 \\
\Rightarrow\ & 2^{2p+1} R^{\frac{p+1}{2}} f_{t_l}^{3p} \left(\alpha\beta^2 \left(\left(2x^{p^2}+x\right) 4Rf_{p_l}^2 - f_{p_l-1}f_{p_l+1}\right) - 2R^{\frac{p^2+1}{2}} \beta^3 f_{p_l}^2 - 16R^2\alpha^3 f_{p_l}^2\right) \\
& -4^{p+1} R^{p+1} \beta^3 f_{p_l}^2 \left(f_{t_l+2}f_{t_l-1}^2 - f_{t_l-2}f_{t_l+1}^2\right)^p = 0 \qquad (38)
\end{aligned}$$

Wir erhalten die Gleichung (38), welche wirklich nur noch von einer Variablen x abhängt. Als dritten Fall nehmen wir an, dass p_l ungerade und t_l gerade ist. Hier müssen wir daran denken, an β den Faktor $4y$ heran zu multiplizieren.

$$\begin{aligned}
& 4^p y^p 2^{3p} y^{3p} f_{t_l}^{3p} \left(4^2 R\alpha\beta^2 \left(\left(2x^{p^2}+x\right) f_{p_l}^2 - 4Rf_{p_l-1}f_{p_l+1}\right) - 4^3 Ry\beta^3 y^{p^2} f_{p_l}^2 - \alpha^3 f_{p_l}^2\right) \\
& -4^3 Ry\beta^3 f_{p_l}^2 \left(2yf_{t_l+2}f_{t_l-1}^2 - 2yf_{t_l-2}f_{t_l+1}^2\right)^p = 0 \\
\Rightarrow\ & 2^{5p} R^{2p} f_{t_l}^{3p} \left(16R\alpha\beta^2 \left(\left(2x^{p^2}+x\right) f_{p_l}^2 - 4Rf_{p_l-1}f_{p_l+1}\right) - 64R^{\frac{p^2+3}{2}} \beta^3 f_{p_l}^2 - \alpha^3 f_{p_l}^2\right) \\
& -2^{p+6} R^{\frac{p+3}{2}} \beta^3 f_{p_l}^2 \left(f_{t_l+2}f_{t_l-1}^2 - f_{t_l-2}f_{t_l+1}^2\right)^p = 0 \qquad (39)
\end{aligned}$$

Es ergibt sich auch hier eine Gleichung (39) in der nur eine Variable x vorkommt. Als

letzten Fall lassen wir p_l ungerade sein und nehmen diesmal an, dass auch t_l ungerade ist.

$$
\begin{aligned}
& 4^p y^p f_{t_l}^{3p} \left(4^2 R\alpha\beta^2 \left(\left(2x^{p^2}+x\right) f_{p_l}^2 - 4R f_{p_l-1} f_{p_l+1}\right) - 4^3 R y \beta^3 y^{p^2} f_{p_l}^2 - \alpha^3 f_{p_l}^2\right) \\
& -4^3 R y \beta^3 f_{p_l}^2 \left(4R f_{t_l+2} f_{t_l-1}^2 - 4R f_{t_l-2} f_{t_l+1}^2\right)^p = 0 \\
\Rightarrow\ & 4^p y^p f_{t_l}^{3p} \left(16 R\alpha\beta^2 \left(\left(2x^{p^2}+x\right) f_{p_l}^2 - 4R f_{p_l-1} f_{p_l+1}\right) - 64 R^{\frac{p^2+3}{2}} \beta^3 f_{p_l}^2 - \alpha^3 f_{p_l}^2\right) \\
& -4^{p+3} R^{p+1} y \beta^3 f_{p_l}^2 \left(f_{t_l+2} f_{t_l-1}^2 - f_{t_l-2} f_{t_l+1}^2\right)^p = 0 \\
\Rightarrow\ & y\Bigg(4^p R^{\frac{p-1}{2}} f_{t_l}^{3p} \left(16 R\alpha\beta^2 \left(\left(2x^{p^2}+x\right) f_{p_l}^2 - 4R f_{p_l-1} f_{p_l+1}\right) - 64 R^{\frac{p^2+3}{2}} \beta^3 f_{p_l}^2 - \alpha^3 f_{p_l}^2\right) \\
& -4^{p+3} R^{p+1} \beta^3 f_{p_l}^2 \left(f_{t_l+2} f_{t_l-1}^2 - f_{t_l-2} f_{t_l+1}^2\right)^p\Bigg) = 0 \qquad (40)
\end{aligned}
$$

Wir erhalten eine Gleichung (40), bei der y erhalten bleibt. Abhängig von p_l und t_l rechnen wir stets nur mit einer der Gleichungen (37), (38), (39) oder (40). Hierbei bezeichnen wir die benutzte Gleichung mit D_y.

Nach den langen Berechnungen können wir nun endlich testen, ob unser gewähltes t_l, welches den vorangegangenen Test der x-Koordinaten schon bestanden hat, auch den Test der y-Koordinaten besteht und damit unserem gesuchten Wert für t_l entspricht. Wir berechnen den Grad des größten gemeinsamen Teilers von D_y, also einer der vier Gleichungen (37), (38), (39) oder (40) und f_l. Ist der Grad gleich Null, wissen wir, dass die Gleichung (27) für unser gewähltes t_l nicht erfüllt ist. In diesem Fall müssen wir einen anderen Wert für t_l wählen. Dafür erhöhen wir den Wert von t_l einfach um eins und beginnen erneut mit dem Test für die x-Koordinaten. Sollte der Grad des größten gemeinsamen Teilers von D_y und f_l größer als Null sein, dann haben wir den korrekten Wert für t_l gefunden.

Im **Fall B** ($\phi^2(P) \neq \pm p_l(P)$) gilt:

Setze $t_l = 1$ und berechne das Polynom D_x.

$$
\text{Grad}\left(ggT\left(D_x, f_l\right)\right) \begin{cases} = 0 & \text{dann gilt } \phi_l^2(P) + p_l(P) \neq t_l \phi_l(P), \\ & \text{wähle } t_l = t_l + 1 \text{ und starte erneut bei der Berechnung von } D_x, \\ > 0 & \text{dann muss mit } D_y \text{ weiter getestet werden} \end{cases}
$$

Berechne das Polynom D_y.

$$
\text{Grad}\left(ggT\left(D_y, f_l\right)\right) \begin{cases} = 0 & \text{dann gilt } \phi_l^2(P) + p_l(P) \neq t_l \phi_l(P), \\ & \text{wähle } t_l = t_l + 1 \text{ und starte erneut bei der Berechnung von } D_x, \\ > 0 & \text{korrekten Wert für } t_l \text{ gefunden, es gilt } \phi_l^2(P) + p_l(P) = t_l \phi_l(P). \end{cases}
$$

6.3.4 Berechnung von t und $\#E(\mathbb{F}_p) = p + 1 - t$

Nachdem wir nun für alle $l = 2, 3, \ldots, l_n$ den Wert $t_l \equiv t \bmod l$ berechnet haben, erhalten wir ein Kongruenzsystem der Form

$$\begin{aligned} t &\equiv t_2 \bmod 2, \\ t &\equiv t_3 \bmod 3, \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ t &\equiv t_{l_n} \bmod l_n. \end{aligned}$$

Dieses können wir mit Hilfe des Chinesischen Restsatzes 1.1 lösen und damit den konkreten Wert für t berechnen. Zu beachten ist hierbei, dass t sich in dem Hasse Intervall

$$-2\sqrt{p} \leq t \leq \sqrt{p}$$

befinden muss. Ist t kleiner als $-2\sqrt{p}$, dann muss das Produkt der Primzahlen $l = 2, 3, \ldots, l_n$ aus dem eben besprochenen Kongruenzsystem solange hinzu addiert werden, bis t im Hasse Intervall liegt. Ist t aber größer als $2\sqrt{p}$, dann muss das eben erwähnte Produkt solange von t abgezogen werden, bis t sich im Hasse Intervall befindet. Jetzt brauchen wir das erhaltene t nur noch in die Gleichung

$$\#E(\mathbb{F}_p) = p + 1 - t$$

einzusetzen und erhalten die Anzahl der Punkte $\#E(\mathbb{F}_p)$ der Kurve $E(\mathbb{F}_p) : y^2 = x^3 + ax + b$.

6.4 Der Schoof Algorithmus kurz zusammengefasst

Damit wir einen Gesamtüberblick über den Schoof Algorithmus bekommen, werden wir hier noch einmal kurz den Ablauf in Form des Algorithmus 6.1 angeben. Wir bezeichnen mit $deg(f)$ den Grad eines Polynoms f und mit $nextprime(l)$ die nächst größere Primzahl von l. Das heißt, wenn $l = 3$ ist, dann ergibt $nextprime(l)$ die Primzahl 5. Hat l den Wert 1, dann erhält man $nextprime(l) = 2$.

Algorithmus 6.1 Schoof Algorithmus

Eingabe: $E : y^2 = x^3 + a\,x + b$ in $\mathbb{F}_p$

Ausgabe: $\#E(\mathbb{F}_p)$

SCHRITT 1: Berechne Primzahlen $l = 2, 3, 5, \ldots, l_n$, so dass das Primzahlenprodukt $\prod_{l=2}^{l_n} l \geq 4\sqrt{p}$ ist.

SCHRITT 2: **Setze** $l \leftarrow 2$.

SCHRITT 3: **Falls** $deg\left(ggT(x^3 + ax + b, x^p - x)\right) > 0$

Setze $t_l \leftarrow 0$.

Sonst Setze $t_l \leftarrow 1$.

SCHRITT 4: **Falls** $l < l_n$ **Setze** $l \leftarrow nextprime(l)$.

Sonst Gehe zu Schritt: 7

SCHRITT 5: **Falls** $deg\,(ggT(g(x), f_l)) > 0$ *(Fall A)*

Falls $\left(\frac{p_l}{l}\right) = 1$ *(Fall A1)*

Berechne w als Quadratwurzel von p_l modulo l

Falls $deg\,(ggT(h(x), f_l)) > 0$

Falls $deg\,(ggT(k, f_l)) > 0$

Setze $t_l \leftarrow +2w$ und gehe zu Schritt: 4.

Sonst Setze $t_l \leftarrow -2w$ und gehe zu Schritt: 4.

Sonst Setze $t_l \leftarrow 0$ und gehe zu Schritt: 4. *(Fall A12)*

Sonst Setze $t_l \leftarrow 0$ und gehe zu Schritt: 4. *(Fall A2)*

Sonst Setze $t_l \leftarrow 0$. *(Fall B)*

SCHRITT 6: **Setze** $t_l \leftarrow t_l + 1$.

Falls $deg\,(ggT(D_x, f_l)) > 0$

Falls $deg\,(ggT(D_y, f_l)) > 0$

Korrekten Wert für t_l gefunden, gehe zu Schritt: 4

Sonst Gehe zu Schritt: 6

Sonst Gehe zu Schritt: 6

SCHRITT 7: Berechne t aus den erhaltenen Werten für t_l mit dem Chinesischen Restsatz.

SCHRITT 8: **Setze** $\#E(\mathbb{F}_p) = p + 1 - t$.

SCHRITT 9: **Rückgabe:** $\#E(\mathbb{F}_p)$.

6.5 Laufzeit des Schoof Algorithmus

Der Algorithmus von Schoof ist der erste Algorithmus, der in polynomieller Zeit die Anzahl der Punkte auf einer elliptischen Kurve berechnet. Nach [Sch85] und [Sch95] ergibt sich die Laufzeit wie folgt. In Schritt 1 bestimmen wir die Anzahl der für den Algorithmus benötigten Primzahlen. Dies lässt sich in Zeit $O(\log p)$ durchführen. Die Berechnung der reduzierten Divisionspolynome stellt einen wesentlichen Bestandteil des Schoof Algorithmus dar. Damit verbunden ist die Berechnung der größten gemeinsamen Teiler, welche in $O(\log^3 p)$ berechnet werden können. Für die Berechnung von t_l ergibt sich ein Rechenaufwand von $O(\log^7 p)$. Da wir dies für $O(\log p)$ Primzahlen durchführen müssen ergibt dies eine Gesamtlaufzeit von $O(\log^8 p)$. Damit ist die Laufzeit des Schoof Algorithmus polynomiell.

6.6 Anwendungsbeispiel

Damit deutlich wird, wie der Schoof Algorithmus funktioniert und welche Ausmaße er in Bezug auf die Länge der Divisionspolynome hat, wollen wir ein komplettes Beispiel durchführen. Als elliptische Kurve soll uns $E(\mathbb{F}_7) : y^2 = x^3 + x + 1$ dienen. Dabei ergibt sich $a = 1$ und $b = 1$. Wir führen nun den Schoof Algorithmus Schritt für Schritt, wie er in Algorithmus 6.1 beschrieben ist, durch.

Schritt 1: Zuerst beginnen wir, die für unsere Berechnung benötigten Primzahlen zu berechnen. Hierbei muss das Produkt der ersten Primzahlen größer als die Länge des Hasse Intervalls $4\sqrt{p}$ sein. Es ergibt sich für die Länge des Hasse Intervalls $4\sqrt{7} \approx 10,58$. Das Produkt $2 \cdot 3 \cdot 5 = 30$ erfüllt damit unsere Bedingung. Für die Berechnung benötigen wir damit die Primzahlen $2, 3, 5$, mit $l_n = l_3 = 5$.

Schritt 2: Nun können wir mit der Berechnung von t_2 für $l = 2$ beginnen.

Schritt 3: Der größte gemeinsame Teiler von $x^3 + x + 1$ und $x^7 - x$ ist Null, damit sind die beiden Polynome teilerfremd. Der Grad dieses größten gemeinsamen Teilers ist damit auch gleich Null. Auf Grund dieser Tatsache können wir

$$t_2 = 1 \bmod 2$$

setzen.

Schritt 4: Da $l = 2$ kleiner $l_3 = 5$ ist, setzen wir l auf die nächst größere Primzahl $l = 3$.

Schritt 5: Zuerst berechnen wir $p_3 = 1 \bmod 3$ und das Polynom $g(x)$. Dieses ergibt sich durch

$$g(x) = x^{49} + 6x.$$

Für das reduzierte Divisionspolynom f_3 erhalten wir

$$f_3 = 3x^4 + 6x^2 + 5x + 6.$$

Der größte gemeinsame Teiler von $g(x)$ und f_3 ergibt das Polynom f_3. Dieses hat bekanntlich einen Grad größer als Null. Damit wissen wir, dass wir uns in *Fall A* befinden. Da das Legendre - Symbol von $\left(\frac{1}{3}\right) = 1$ ist, befinden wir uns im *Fall A1*. Wir berechnen als nächstes die Quadratwurzel w von p_3 modulo 3.

$$\begin{aligned} w^2 &= p_3 \bmod 3 \\ w^2 &= 1 \bmod 3 \\ w &= \pm 1 \end{aligned}$$

Wir berechnen das Polynom $h(x)$ und anschließend den $ggT(h(x), f_3)$. Wir erhalten

$$h(x) = x^7 + 6x$$

und

$$ggT(h(x), f_3) = ggT(x^7 + 6x, 3x^4 + 6x^2 + 5x + 6) = 0.$$

Da die beiden Polynome $h(x)$ und f_3 teilerfremd sind, d. h. der $ggT(h(x), f_3)$ Null ist und damit auch dessen Grad, wissen wir, dass wir uns im Fall A12 befinden. Der Wert von t_3 ist somit durch

$$t_3 = 0 \bmod 3$$

gegeben. Wir haben damit den Fall $l = 3$ abgeschlossen und gehen über zu Schritt 4.

Schritt 4: Für $l = 3$ haben wir eine Primzahl, die kleiner $l_n = l_3 = 5$ ist. Somit gibt es eine weitere Torsionsgruppe, die wir untersuchen müssen. Wir setzen unser $l = 5$ und fahren mit Schritt 5 fort.

Schritt 5: Hier müssen wir $p_5 = p \bmod 5$, $g(x)$ und f_5 berechnen. Wir erhalten $p_5 = 2 \bmod 5$,

$$g(x) = 4x^{52} + 4x^{50} + 4x^{49} + 6x^4 + 2x^2 + x + 6$$

und

$$f_5 = 5x^{12} + 6x^{10} + 2x^9 + 2x^7 + 6x^6 + 4x^5 + 6x^4 + 4x^2 + 2x.$$

Für den $ggT(g(x), f_5)$ erhalten wir den Wert 6, was einem Grad von 0 entspricht. Aufgrund dieser Feststellung befinden wir uns in Fall B. Es ist jetzt klar, dass t_5 nun nicht mehr Null sein kann, denn sonst wären wir im Fall A geblieben. Wir setzen $t_5 = 1$ und

berechnen das Polynom D_x für $p_5 = 2$ und $t_5 = 1$. Da der Grad dieses Polynoms schon weit über 100 reicht, haben wir es bei der Berechnung, um mit kleineren Werten rechnen zu können, modulo f_5 genommen. Das können wir machen, da wir uns nur mit dem größten gemeinsamen Teiler von D_x und f_5 beschäftigen. Nebenbei bemerkt sollte man die Reduzierung um das entsprechende reduzierte Divisionspolynom so früh wie möglich bei der Berechnung der Polynome $g(x)$, $h(x)$, D_x und D_y durchführen, um den Grad der Polynome möglichst klein zu halten. Das Polynom D_x ergibt sich durch

$$D_x = 4x^{11} + 6x^{10} + 4x^9 + 6x^7 + 4x^5 + 5x^4 + 3x^3 + x^2 + 2x + 1.$$

Testen wir nun, ob der Grad des $ggT(D_x, f_5)$ größer als Null ist. Es ergibt sich $ggT(D_x, f_5) = 6$ und damit ein Grad von Null. Dadurch kommt der Wert $t_5 = 1$ nicht in Frage. Wir setzt $t_5 = 2$ und berechnen D_x für $p_5 = 2$ und $t_5 = 2$. Es ergibt sich $D_x = 0$, nachdem wir es modulo f_5 genommen haben. Damit wissen wir, dass der $ggT(D_x, f_5) = f_5$ und der Grad damit größer als Null ist. Um aber sicher sein zu können, dass $t_5 = 2$ der richtige Wert ist, fehlt noch die Überprüfung des Grades des $ggT(D_y, f_5)$. Dazu berechnen wir zuerst D_y und reduzieren es bei der Berechnung modulo f_5. Wir erhalten

$$D_y = 3x^{11} + x^{10} + x^9 + 6x^8 + 2x^7 + 3x^6 + 3x^5 + x^4 + 4x^3 + x^2 + 5x + 1.$$

Der Grad von $ggT(D_y, f_5)$ ergibt in diesem Fall einen Wert von Null, was $t_5 = 2$ damit nicht zu unserer gesuchten Lösung macht. Halten wir noch einmal fest, dass weder 1 noch 2 für t_5 in Frage kommt. Es muss damit entweder 3 oder 4 der richtige Wert für t_5 sein. Fahren wir mit den Tests für $p_5 = 3$ fort. Wir erhalten $D_x = 0$ modulo f_5. Folglich ergibt sich für $ggT(D_x, f_5) = f_5$. Wieder ist der $ggT(D_y, f_5)$ von uns zu berechnen. Diesmal erhalten wir für $D_y = 0$ modulo f_5. Dadurch ist leicht ersichtlich, dass $ggT(D_y, f_5) = f_5$ ist und damit der Grad des $ggT(D_y, f_5)$ größer also Null ist. Damit haben wir auch unser

$$t_5 = 3 \bmod 5$$

gefunden. Schritt 4 sagt uns, dass wir kein weiteres t_l mehr berechnen müssen, deshalb gehen wir zu Schritt 7 über.

Schritt 7: Die vorangegangenen Berechnungen liefern uns

$$\begin{aligned} t &\equiv t_2 \equiv 1 \bmod 2, \\ t &\equiv t_3 \equiv 0 \bmod 3, \\ t &\equiv t_5 \equiv 3 \bmod 5. \end{aligned}$$

Dieses Gleichungssystem können wir leicht mit dem Chinesischen Restsatz lösen. Es ergibt sich

$$t \equiv 3 \bmod 30.$$

Setzen wir dieses in die Gleichung

$$\#E(\mathbb{F}_7) = 7 + 1 - t$$

ein, erhalten wir

$$\#E(\mathbb{F}_7) = 5.$$

Damit haben wir die Anzahl der Punkte der Kurve $E(\mathbb{F}_7) : y^2 = x^3+x+1$ mit $\#E(\mathbb{F}_7) = 5$ gefunden. Wir können die Punkte auch zählen, indem wir für jeden Wert $x \in \mathbb{F}_7$ prüfen, ob x^3+x+1 ein quadratischer Rest modulo 7 ist. Dies soll folgende Tabelle verdeutlichen.

x	x^3+x+1	Quadrat modulo 7	y
0	1	ja	$1, 6$
1	3	nein	-
2	4	ja	$2, 5$
3	3	nein	-
4	6	nein	-
5	5	nein	-
6	6	nein	-

Es wird deutlich das 1 und 4 die einzigen Quadrate modulo 7 sind. Damit ist die Punktemenge der Kurve $y^2 = x^3 + x + 1$ durch

$$\{(0, 1), (0, 6), (2, 2), (2, 5)\}$$

gegeben. Hinzu kommt noch der Punkt $\mathcal{O}$, was ist zu Anzahl der Punkte $\#E(\mathbb{F}_7) = 4 + 1 = 5$ führt. Damit hat uns der Schoof Algorithmus das korrekte Ergebnis für die Gruppenordnung der elliptischen Kurven $E(\mathbb{F}_7) : y^2 = x^3 + x + 1$ geliefert.

Teil III
Implementierung

7 Die Umsetzung der Algorithmen für elliptische Kurven in Programme

In diesem Kapitel werden wir uns mit der konkreten Umsetzung der oben besprochenen Algorithmen befassen. Wir beginnen mit einem Überblick über die verwendeten Datenstrukturen und Klassen. Diese Klassen stellen wir anschließend im Einzelnen vor. Hierbei soll deutlich werden, inwieweit die einzelnen Klassen und Funktionen zusammenwirken. Die Implementierung der Algorithmen wurde in Java mit der Version 1.5.0 beta durchgeführt. Ältere Versionen von Java sind für die Ausführung der Algorithmen nicht zu empfehlen, da sie einige wichtige Funktionen, die es erst ab Version 1.5 gibt, nicht beinhalten.

Für die Kryptographie ist es wichtig, dass man in den Algorithmen „große“ Zahlen (über 100 Bit Länge) verwenden kann. Nun sind die Standardbereiche für ganze Zahlen in Java bei integer nur 32 Bit und bei long 64 Bit lang. Wir benötigen also einen Zahlenbereich, mit dem wir auch sehr große Zahlen beliebiger Größe darstellen können. Java bietet uns dafür zwei Klassen an. Zum einem ist es die BigInteger Klasse für die Darstellung beliebig langer ganzzahliger Werte und zum anderen die BigDecimal Klasse um beliebig lange gebrochen rationale Zahlen darstellen zu können. Die beiden benötigten Klassen sind zwar auch schon in früheren Java-Versionen vorhanden, doch wurde speziell die BigDecimal Klasse um einige für uns relevante Methoden erweitert.

Bei der Implementierung der Algorithmen und Datenstrukturen ist besonders darauf zu achten, dass weitestgehend mehrfache Berechnungen vermieden werden. Diese wirken sich unter Umständen deutlich auf die tatsächliche Laufzeit der Algorithmen aus. Wie wir im vorangegangenen Beispiel kurz erwähnt haben, ist es wichtig, dass die Polynome $g(x)$, $h(x)$, D_x und D_y schon bei der Berechnung möglichst klein gehalten werden. Das erreichen wir, indem die einzelnen Bestandteile dieser Polynome mit dem jeweils relevanten Divisionspolynom f_l reduzieren.

7.1 Vorstellung der Datenstrukturen

Die wichtigsten Basisdatenstrukturen um elementare Zahlen darstellen zu können sind, wie eben erwähnt, die BigInteger und die BigDecimal Klasse. Beide dienen uns dazu, komplexere Datenstrukturen aufbauen zu können. Oft benötigen wir Listen um z. B. Punkte (wie im Algorithmus 5.2 von Shanks-Mestre) abspeichern zu können. Dieses lässt sich

durch eine spezielle Form der Array Datenstruktur verwirklichen. Da die Indizierung eines beliebigen Java Arrays auf integer und damit auf eine Länge von 32 Bit beschränkt ist, wären Berechnungen ab einer bestimmten Größe mit dieser Datenstruktur logischerweise nicht mehr möglich. Damit wir dieser Beschränkung nicht unterliegen, haben wir eine BigArray Datenstruktur entwickelt.

7.1.1 BigArrays

Die BigArrays sind Kombinationen aus Ringlisten und Java Arrays, in denen man beliebig viele Elemente desselben Typs speichern kann. Die Grundstruktur besteht aus einer Ringlisten. In jedem Listenknoten befindet sich ein Java Array. Da die Anzahl der Ringlistenknoten nicht begrenzt ist, können auf diese Weise beliebig viele Elemente aubespeichert werden. Wir können aus ein BigArray auch als eine Aneinanderreihung von vielen Java Arrays vorstellen. Durch die Ringstruktur ist der Zugriff auf die einzelnen Elemente sehr schnell. Der Positionszeiger wird hierbei immer nur in eine Richtung bewegt und kann jeden Listenknoten in linearer Zeit erreichen. Die Abbildung 6 verdeutlicht die Struktur des BigArrays noch einmal grafisch, dabei stellen die ovalen grauen Kreise die Ringlistenknoten dar und die weißen Kästchen im Inneren bilden die Java Arrays eines Knotens. Die Pfeile zeigen jeweils auf den Nachfolgeknoten.

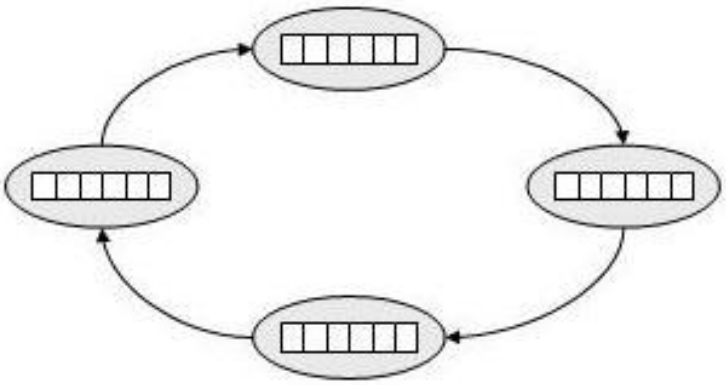


Abbildung 6: Aufbau der BigArray Datenstruktur

Bei der Erzeugung eines solchen BigArrays gibt man die Anzahl der Elemente, die maximal in der Datenstruktur gespeichert werden sollen, an. Danach wird errechnet, wie viele Knoten und damit verbundene Java Arrays zur Speicherung der Elemente benötigt werden. Die Knoten werden anschließend mit Java Arrays initialisiert und zu einer Ringsstruktur verknüpft.

Wir haben drei verschiedene Formen der BigArray Datenstruktur implementiert. Diese sind dazu da, um die BigArrays speziell auf die Elemente, die in ihnen gespeichert

werden, anzupassen. Die zwei einfachsten Varianten sind die BigIntegerArray Klasse und die BigProjectivePointArray Klasse, welche dazu da sind Elemente vom Typ BigInteger beziehungsweise ProjectivePoint in einer BigArray Datenstruktur speichern zu können. Die dritte Form der BigArray Datenstruktur ist die Polynom Klasse. Diese ist um einiges umfangreicher als die beiden anderen eben erwähnten Klassen. Wir werden sie deshalb in einem eigenen Unterabschnitt vorstellen.

7.1.2 Darstellung von Polynomen

Die Polynom Klasse dient dazu, Polynome beliebiger Länge abzuspeichern. Auch hierbei kommt erneut die BigArray Datenstruktur zum Tragen. Ein Polynom können wir formal durch die Gleichung

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

darstellen. Da wir uns nur mit Polynomen mit einer Veränderlichen befassen, reicht es aus, die Koeffizienten $a_n, a_{n-1}, \ldots, a_1, a_0$ in unserer eben vorgestellten BigArray Datenstruktur zu speichern. Wir bezeichnen mit $B[i]$ das i-te Element eines BigArrays. Das erste Element in diesem Array befindet sich an Position $B[0]$ und stellt den Koeffizienten a_n von x^n dar. Das zweite Element also $B[1]$ entspricht folglich dem Koeffizienten a_{n-1}. Dieses geht bis zum letzten Element $B[n-1]$ des Arrays und entspricht dem Koeffizienten a_0. Zusammengefasst wird dies folgendermaßen dargestellt.

BigArray Position	$B[0]$	$B[1]$	$\ldots$	$B[n-2]$	$B[n-1]$
Koeffizienten	a_n	a_{n-1}	$\ldots$	a_1	a_0

Die Polynom Klasse wird später bei der Vorstellung der Klassen noch einmal aufgegriffen, dabei werden speziell ihre Funktionen betrachtet.

7.1.3 Darstellung von Kurvenpunkten

Damit wir Punkte einer elliptischen Kurve darstellen können, benötigen wir eine Datenstruktur, die die Koordinaten der Punkte einer solchen Kurve speichert. Wir haben dazu die ProjectivePoint Klasse entwickelt. Sie speichert Punktkoordinaten und kann diese sowohl in projektiver wie auch in affiner Form übergeben bekommen. Intern werden dabei affine Koordinaten in projektive Koordinaten umgewandelt und sämtliche Koordinaten in projektiver Form gespeichert.

7.2 Der Aufbau der Klassen

Nachdem wir in den vorangegangenen Abschnitten die von uns verwendeten Datenstrukturen näher beschrieben haben, wenden wir uns nun den implementierten Klassen und

deren Funktionsweise zu. Zu Beginn wollen wir mit der Abbildung 7 die Abhängigkeiten zwischen den Hauptklassen zeigen. Wir sehen dabei, dass die EllipticCurve Klasse eine zentrale Rolle einnimmt. Sie beinhaltet die Basisoperationen Addition, Subtraktion und Multiplikation für elliptische Kurven, sowie die Algorithmen zur Bestimmung der Gruppenordnung einer elliptischen Kurve.

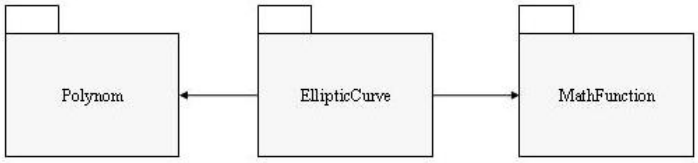


Abbildung 7: Zusammenhänge zwischen den Hauptklassen

Die Polynom Klasse stellt nicht nur eine Datenstruktur wie im Kapitel 7.1.2 beschrieben dar, sondern beinhaltet darüber hinaus Funktionen, die es erlauben mit Polynomen vom Typ Polynom zu rechnen. In ihr sind die Funktionen für die Addition, die Subtraktion, die Multiplikation und die Funktion für den größten gemeinsamen Teiler zweier Polynome gegeben.

Auf der anderen Seite der Abbildung steht die MathFunction Klasse, welche elementare mathematische Operationen bereitstellt. Dazu zählen unter anderem die Berechnung des Legendre - Symbols oder die Wurzel modulo einer Primzahl.

7.2.1 Die EllipticCurve-Klasse

In diesem Unterabschnitt soll es um die zentrale Steuerungsklasse, die wir EllipticCurve genannt haben, gehen.

Wenn man mit elliptischen Kurven rechnen will, so muss man zuerst eine Kurve, welche den Basiskörper darstellt, erzeugen. Dies geschieht durch das Anlegen eines neuen Objektes der Klasse EllipticCurve. Dabei gibt es mehrere Möglichkeiten, Parameter der Kurve zu übergeben. Die folgende Tabelle zeigt alle möglichen Eingabeformen für die Parameter der Kurve.

Parameter
(a, b, p)
(a, x, y, p)
(b, x, y, p)

Die Eingabe kann entweder über die erste Möglichkeit die beiden Parameter a und b der Kurve $y^2 = x^3 + ax + b$ mit der Ordnung p des Körpers $\mathbb{F}_p$ oder durch eine der letzten zwei Möglichkeiten aus der Tabelle erfolgen. Dabei wird ein festes p des Körpers

$\mathbb{F}_p$ mitgegeben, sowie die x und y Koordinate eines affinen Punktes, der auf der Kurve liegen soll und außerdem entweder der Kurvenparameter a oder b.

Wir werden im Anschluss an die Erklärung der Klasse ein kurzes Ablaufbeispiel geben, welches dem Benutzer die zu verwendende Syntax deutlich machen soll. Doch zunächst fahren wir mit unserer Erklärung fort. Wir haben also ein Objekt vom Typ EllipticCurve erzeugt und dabei die Parameter für eine elliptische Kurve mit übergeben. Mit diesem Objekt, nennen wir es kurz *curve*, können wir nun Rechenoperationen zwischen Punkten dieser erzeugten Kurve durchführen oder mit einem der drei implementierten Punktezählalgorithmen die Gruppenordnung unserer erzeugten Kurve *curve* bestimmen. Da die Operationen in der EllipticCurve Klasse implementiert sind, können wir sie mit den jeweiligen Parametern durch den Punktoperator aufrufen. Zum Beispiel *curve.add(A,B)* um zwei Kurvenpunkte zu addieren. Wir geben jetzt eine Übersicht über die vorhandenen Operationen. Im Anhang B befinden sich dazu noch detaillierte Beschreibung in Form einer Java Dokumentation.

isEqual: Überprüft, ob zwei Punkte gleich sind und gibt anschließend den entsprechenden Wahrheitswert zurück.

add: Berechnet die Summe zweier Punkte und gibt den resultierenden Punkt zurück.

negate: Negiert einen Punkt.

sub: Berechnet die Summe eines Punktes mit der Negation eines zweiten Punktes und gibt den resultierenden Punkt zurück.

mul: Berechnet die skalare Multiplikation eines Punktes mit einem Skalar und gibt den resultierenden Punkt zurück.

isCurvePoint: Überprüft, ob ein Punkt auf der betrachteten Kurve liegt und gibt anschließend den Wahrheitswert zurück.

getRandomPoint: Bestimmt einen zufälligen Punkt, der auf der Kurve liegt mit dem Algorithmus 5.1.

getOrder_ByNaive: Berechnet die Anzahl der Punkte auf der betrachteten Kurve mit dem naiven Algorithmus 4.1.

getOrder_ByShanks: Berechnet die Anzahl der Punkte auf der betrachteten Kurve mit dem Shanks-Mestre Algorithmus 5.2.

getOrder_BySchoof: Berechnet die Anzahl der Punkte auf der betrachteten Kurve mit dem Schoof Algorithmus 6.1.

8 Anwendung der Klassen

Nach der Erläuterung der Datenstrukturen und Klassen kommen wir nun zu der praktischen Betrachtung der Klassen. Wir zeigen im Folgenden, wie man mit ihnen arbeitet. Beginnen wollen wir mit der Erzeugung einer elliptischen Kurve.

```
BigInteger a = new BigInteger("1"); // Parameter für a
BigInteger b = new BigInteger("1"); // Parameter für b
BigInteger p = new BigInteger("7"); // Parameter für p

EllipticCurve E = new EllipticCurve(a,b,p); // erzeugt eine neue elliptische Kurve
```

Nachdem wir ein Objekt E vom Typ EllipticCurve erzeugt haben, können wir anschließend alle weiteren Operationen auf E durchführen. Da uns interessiert, wie viele Punkte sich auf unserer elliptischen Kurve E befinden (welche Ordnung die Kurve hat), zählen wir diese zunächst. Dies können wir mit einem der drei implementierten Punktezählalgorithmen durchführen. Die Syntax sieht dabei wie folgt aus.

```
// zählt die Punkte mit dem naiven Algorithmus
BigInteger Ordnung1 = E.getOrder_ByNaive();

// zählt die Punkte mit dem Shanks-Mestre Algorithmus
BigInteger Ordnung2 = E.getOrder_ByShanks();

// zählt die Punkte mit dem Schoof Algorithmus
BigInteger Ordnung3 = E.getOrder_BySchoof();
```

Natürlich ist es auch möglich, Punktoperationen auf unserer Kurve E auszuführen. Dazu müssen wir zuerst Punkte erzeugen, mit denen wir im Anschluss rechnen können.

```
BigInteger x = new BigInteger("0"); // Parameter für x
BigInteger y = new BigInteger("1"); // Parameter für y
BigInteger z = new BigInteger("2"); // Parameter für z

// erzeugt einen Punkt aus projektiven Koordinaten
ProjectivePoint P = new ProjectivePoint(x, y, z);

// erzeugt einen Punkt aus affinen Koordinaten
ProjectivePoint Q = new ProjectivePoint(x, y);
```

Jetzt stellt sich aber die Frage, ob unsere erzeugten Punkte P und Q sich auch auf der elliptischen Kurve E befinden.

```
boolean P_auf_der_Kurve = E.isCurvePoint(P); // testet, ob P auf E liegt
boolean Q_auf_der_Kurve = E.isCurvePoint(Q); // testet, ob Q auf E liegt
```

Wir bekommen als Ergebnis, dass Q auf der Kurve E liegt, P aber nicht. Da Q also auf E liegt, können wir mit ihm weiterhin rechnen. Folgende Operationen sind dabei denkbar.

```
// berechnet die skalare Multiplikation von 3Q auf der Kurve E
ProjectivePoint R = E.mul( new BigInteger("3"), Q);

// addiert zwei Punkte R und Q auf auf der Kurve E
ProjectivePoint W = E.add(R, Q);

// subtrahiert zwei Punkte R und Q auf auf der Kurve E
ProjectivePoint N = E.sub(R, Q);
```

Wir wollen darüber hinaus sehen, welche affinen oder projektiven Koordinaten unsere berechneten Punkte haben.

```
// gibt den Punkt R in projektiven Koordinaten aus
System.out.println("R = " + R);

// gibt den Punkt R in affinen Koordinaten aus
System.out.println("R = " + E.toAffineString(R));
```

Zum Abschluss unserer Betrachtung wollen wir noch überprüfen, ob die berechneten Punkte R und N gleich sind.

```
boolean sind_gleich = E.isEqual(R, N); // testet, ob zwei Punkte gleich sind
```

9 Übersicht über die derzeit bekannten Punktezählalgorithmen

Damit wir die Geschwindigkeiten der oben vorgestellten Punktezählalgorithmen mit den derzeit bekannten Algorithmen vergleichen können, geben wir nun einen kurzen Überblick, über die derzeit bekannten Punktezählalgorithmen und deren Speicherplatzbedarf (aus [LL03]) an.

Zeit $O(np^{\frac{n}{4}})$

- D. Shanks. *Class number, a theory of factorization, and genera.* In Proc. Symp. Pure Math., 20, 1971.
- J.M. Pollard. *Monte Corlo methods for index computation (mod p).* Math. Comp., 32, 1978.

Zeit $O(n^{5+\epsilon})$ und **Speicherplatz** $O(n^2)$

- R. Schoof. *Elliptic curves over finite fields and the computation of square roots mod p.* Math. Comp., 44, 1985.
- A.O.L. Atkin. *The number of points on an elliptic curve modulo a prime.* Number Theory Mailing List., 1988.
- A.J. Menezes, S.A. Vanstone, R.J. Zuccherato. *Counting points on elliptic curves over* $\mathbb{F}_{2^m}$. Math. Comp., 60, 1993.

Zeit $O(n^{4+\epsilon})$ und **Speicherplatz** $O(n^2)$

- N.D. Elkies. *Explicit isogenies.* Draft, 1991.
- A.O.L. Atkin. *The number of points on an elliptic curve modulo a prime.* Number Theory Mailing List, 1991.
- R. Schoof. *Counting points on elliptic curves over finite fields.* J.Théor. Nombres Bordeaux I, 1995.
- J.M. Couveignes. *Quelques calculs en théorie des nombres.* thèse, Université de Bordeaux I, 1994.
- R. Lercier. *Computing isogenies in* $\mathbb{F}_{2^n}$. ANTS-II, LNCS, 1996.
- J.M. Couveignes. *Computing l-isogenies with the p-torsion.* ANTS-II, LNCS, 1996.

Zeit $O(n^{3+\epsilon})$ und **Speicherplatz** $O(n^3)$

- T. Satoh. *The canonical lift of an ordinary elliptic curve over a finite field and its point counting.* J. Ramanujan Math. Soc., 15, 2000.
- M. Fouquet, P. Gaudry, R.J. Harley *An extension of Satoh's algorithm and its implementation.* J. Ramanujan Math. Soc., 2000.
- B. Skjernaa, *Satoh's algorithm in characteristic 2.* Math. Comp., 2000.

Zeit $O(n^{3+\epsilon})$ und **Speicherplatz** $O(n^2)$

- F. Vercauteren, B. Preneel, J. Vandewalle. *A Momory Efficient Version of Sotoh's Algorithm.* EUROCRYPT 2001, LNCS.
- J.F. Mestre. *AGM pour le genre 1 et 2.* Lettre à Gaudry et Harley, 2000.

Zeit $O(n^{2.5+\epsilon})$ und **Speicherplatz** $O(n^2)$

- T. Satoh, B. Skjernaa, Y. Taguchi. *Fast computation of canonical lifts of elliptic curves and its application to point counting.* 2001.
- T. Satoh. *On p-adic point counting algorithms for elliptic curves over finite fields.* ANTS-V, 2002.
- H.Y. Kim, J.Y. Park, J.H. Cheon, J.H. Park, J.H. Kim, S.G. Hahn. *Fast elliptic curve point counting using gaussian normal basis.* ANTS-V, Juli, 2002.
- P. Gaudry. *A comparison and a combination of SST and AGM algorithms for counting points of elliptic curves in characteristic 2.* ASIACRYPT, 2002.

Zeit $O(n^{2+\epsilon})$ und **Speicherplatz** $O(n^2)$

- R. Lercier, D. Lubicz. *Counting points on elliptic curves over finite fields in quadratic time.* noch nicht veröffentlicht, 2002.
- R.J. Harley. *Algorithmes avancés pour lt'arithmétique des courbes.* 2003.

10 Zeitvergleich der implementierten Punktzählalgorithmen

Um einen Eindruck über die Geschwindigkeit der von uns implementierten Algorithmen bekommen zu können, haben wir verschieden Zeitmessungen durchgeführt. Als Rechner nehmen wir einen AMD Athlon mit 1,2 Ghz und 256 MB RAM. Zunächst wollen wir aber die theoretische Laufzeit der Punktezählalgorithmen in Abbildung 8 graphisch gegenüber stellen. Man kann deutlich erkennen, dass der naive Algorithmus bei kleinen Primzahlen bis etwa 15 Bit schneller ist, als die beiden anderen Algorithmen. Der Shanks-Mestre Algorithmus ist im mittleren Bereich zwischen 15 und etwa 25 Bit der Schnellste. Deutlich erkennbar ist aber auch, dass die Kurve des Schoof Algorithmus eine deutlich geringere Steigung besitzt. Damit bietet der Schoof Algorithmus für große Zahlen einen enormen Geschwindigkeitsvorteil. Ab einer bestimmten Größe ist er aber auch nicht mehr gut anwendbar.

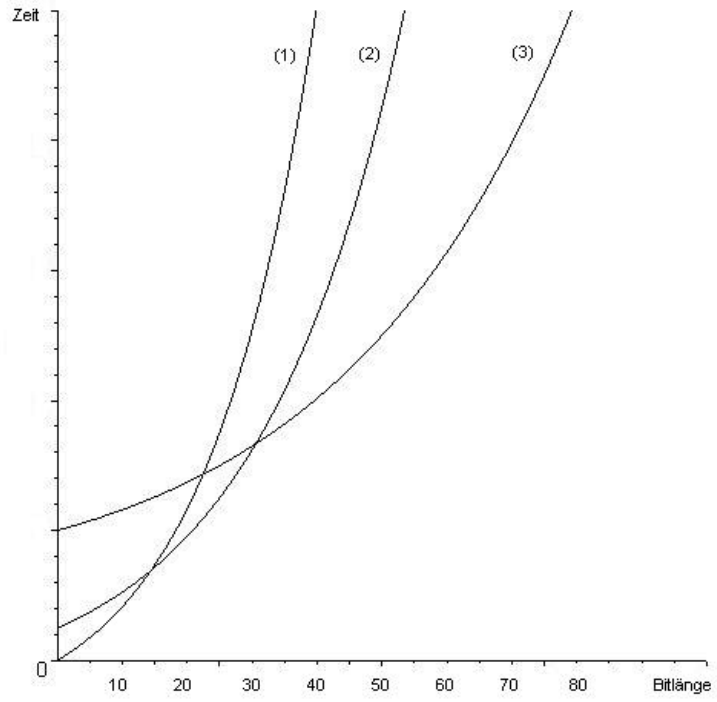


Abbildung 8: Zeitvergleich (theoretisch): (1) naiver Algorithmus (2) Shanks-Mestre (3) Schoof

Damit wir diese theoretischen Werte überprüfen können, haben wir einige Testkurven mit verschiedenen Bitlängen durch die von uns implementierten Algorithmen berechnen lassen. Als Testkurve nahmen wir bei jedem Durchlauf die Kurve $y^2 = x^3 + x + 1$. Wir benutzten für p zufällige Primzahlen der Bitlängen 5, 10, 20, 30 und 40. Die folgende Tabelle spiegelt die erhaltenen Laufzeiten wieder.

Algorithmus	Laufzeit für Bitlänge				
	5	10	20	30	40
naiver Algorithmus	10 ms	130 ms	20 s	6 h 53 m	-
Shanks Mestre Algorithmus	200 ms	260 ms	1 s	4 s	2 m
Schoof Algorithmus	6 m	-	-	-	-

Für die nicht ausgefüllten Felder haben wir keine Zeitmessung durchgeführt, da diese Werte eine enorm lange Berechnung mit den von uns implementierten Algorithmen benötigen. Wir sehen deutlich, dass die tatsächliche Laufzeit des Schoof Algorithmus erheblich langsamer, als die Laufzeit der anderen Algorithmen ist. Da es uns bei der Implementierung vorrangig darum ging lauffähige Algorithmen zu schreiben, wurde auf die Leistungsoptimierung keinen Wert gelegt. Dies sollte auch nicht Teil dieser Studienarbeit sein.

11 Ausblicke

Die von mir vorgestellten elliptischen Kurven stellen ein Grundgerüst dar, auf dem man kryptographische Verfahren aufbauen kann. Sie bieten dabei die am Anfang erwähnten Vorteile, dass sie mit kleineren Zahlen vergleichsweise höhere Sicherheit erreichen können, als beispielsweise RSA basierte Verfahren. Viele public key Verfahren lassen sich ausgehend vom Körper der ganzen Zahlen auf elliptische Zahlen übertragen.

Wir haben in dieser Studienarbeit Basisoperationen für das Rechnen mit elliptischen Kurven und ein paar wichtige Algorithmen zur Bestimmung der Gruppenordnung von elliptischen Kurven implementiert und theoretisch dargestellt. Besonderen Wert haben wir auf den Algorithmus von Schoof gelegt. Es wurden dabei alle Fehler aus den wichtigen Polynomgleichungen korrigiert. Die Korrektheit der Implementierung wurde mit Hilfe des vorgestellten naiven Algorithmus verifiziert. Dies gibt uns die Sicherheit, dass wir in dieser Arbeit eine durchaus fehlerfreie Abhandlung des Schoof Algorithmus entwickelt haben. Es wurde dabei stets darauf Wert gelegt, tiefgründige Erläuterungen der doch sehr komplexen Algorithmenstruktur zu geben. Dadurch ist eine umfassende, vollständige und vor allem korrekte Beschreibung entstanden, die sich in der Literatur auf diese Weise nicht wieder finden lässt.

Der von uns implementierte Schoof Algorithmus wurde als eine lauffähige, nicht vollständig optimierte Version entwickelt. Das Ziel bestand nur darin den Algorithmus überhaupt in einem Programm zu realisieren. Durch schnelle Arithmetik, wie z. B. schnelle Multiplikation, ist es möglich, ihn auch in der Praxis an seine theoretische Laufzeit heran zu bekommen. In Kapitel 9 sind einige Verbesserungen angeführt, mit deren Hilfe man die Laufzeit des Schoof Algorithmus noch erheblich verbessern kann.

Literatur

[Blö00] J. Blömer. *Algorithmen in der Zahlentheorie.* http://webserv.upb.de/cs/ag-bloemer/lehre/azt_WS2000/, 2000.

[BSS00] I.F. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 2000.

[Buc03] J. Buchmann. *Einführung in die Kryptographie, 3. Auflage.* Springer-Verlag Berlin Heidelberg, 2003.

[CMO98] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. *Advances in Cryptology – ASIACRYPT'98*, Lecture Notes in Computer Science, volume 1541, 1998.

[Coh00] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag Berlin Heidelberg, 2000.

[CR88] L. Charlab and D. Robbins. An elementary introduction to elliptic curves. *CRD Expository Report, Institute for Defense Analysis Princeton*, 31, 1988.

[GG99] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[Hun74] Th.W. Hungerford. *Algebra.* Graduate Texts in Mathematics 73. Springer-Verlag, 1974.

[KDJ04] V. Kumar, S. Doraiswamy, and Z. Jainullabudeen. *Design and Analysis of Algorithms: Elliptic Curve Cryptography.* http://www.eas.asu.uzi/˜ cse450sp/projects/final_P113.doc, 2004.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48, number 177, 1987.

[Lan78] S. Lang. *Elliptic curves: Diophantine analysis.* Springer-Verlag, 1978.

[LL03] R. Lercier and D. Lubicz. *Algorithmic aspects of Mestre's p-adic point counting ideas.* http://www.cacr.math.uwaterloo.ca/conferences/2003/ecc2003/lercier.pdf, 2003.

[Mil86] V.S. Miller. Use of elliptice curves in cryptography. *Advances in Cryptology – CRYPTO '85*, Lecture Notes in Computer Science, volume 218, 1986.

[Mir03] A. Mirbach. *Elliptische Kurven: Die Bestimmung ihrer Punktzahl und Anwendung in der Kryptographie*. Verlagshaus Monsenstein und Vannerdat, 2003.

[Mül91] V. Müller. *Die Berechnung der Punktzahl von elliptischen Kurven über endlichen Primkörpern*. Diplom, Universität des Saarlandes, 1991.

[Sch85] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation*, 44, number 170, 1985.

[Sch95] R. Schoof. Counting points on elliptic curves over finite fields. *Journal de Théorie des Nombres*, 7, 1995.

[Sil86] J.H. Silverman. *The arithmetic of elliptic curves*. Graduate Texts in Mathematics 106. Springer-Verlag, 1986.

[Sti02] D. R. Stinson. *Cryptography: Theory and Practice, 2. Edition*. Chapman & Hall/CRC, 2002.

[Wer02] A. Werner. *Elliptische Kurven in der Kryptographie*. Springer-Verlag Berlin Heidelberg, 2002.

A Programm CD

B Java Dokumentation

Im Folgenden befindet sich die komplette Java Dokumentation der implementierten Klassen.

All Classes
BigIntegerArray
BigProjectivePointArray
EllipticCurve
IOFunction
MathFunction
Polynom
ProjectivePoint
Timer
test

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class BigIntegerArray

```
java.lang.Object
  └ BigIntegerArray
```

public class **BigIntegerArray**
extends java.lang.Object

Klasse zum speichern beliebig vieler Elemente vom Typ BigInteger in einer BigArray Datenstruktur.

Constructor Summary

BigIntegerArray(java.math.BigInteger l) Konstruktor um ein BigIntegerArray für maximal l Elemente vom Typ BigInteger zu erzeugen.

Method Summary

java.math.BigInteger	**get**(java.math.BigInteger pos) Gibt aus dem BigIntegerArray das an Position pos liegende BigInteger zurück.
java.math.BigInteger	**getArrayLength**() Gibt die Größe des BigIntegerArrays zurück.
void	**set**(java.math.BigInteger pos, java.math.BigInteger elem) Fügt ein BigInteger in das BigIntegerArray an Position pos ein.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

BigIntegerArray

```
public BigIntegerArray(java.math.BigInteger l)
```

Konstruktor um ein BigIntegerArray für maximal l Elemente vom Typ BigInteger zu erzeugen.

Method Detail

set

```
public void set(java.math.BigInteger pos,
                java.math.BigInteger elem)
```

Fügt ein BigInteger in das BigIntegerArray an Position pos ein.

---

get

```
public java.math.BigInteger get(java.math.
BigInteger pos)
```

Gibt aus dem BigIntegerArray das an Position pos liegende BigInteger zurück.

---

getArrayLength

```
public java.math.BigInteger getArrayLength()
```

Gibt die Größe des BigIntegerArrays zurück.

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class BigProjectivePointArray

```
java.lang.Object
  └ BigProjectivePointArray
```

public class **BigProjectivePointArray**
extends java.lang.Object

Klasse zum speichern beliebig vieler Elemente vom Typ ProjectivePoint in einer BigProjectivePointArray Datenstruktur.

Constructor Summary

BigProjectivePointArray(java.math.BigInteger l) Konstruktor um ein BigProjectivePointArray für maximal l Elemente vom Typ ProjectivePoint zu erzeugen.

Method Summary

ProjectivePoint	**get**(java.math.BigInteger pos) Gibt aus dem BigProjectivePointArray den an Position pos liegenden ProjectivePoint zurück.
java.math.BigInteger	**getArrayLength**() Gibt die Größe des BigProjectivePointArrays zurück.
void	**set**(java.math.BigInteger pos, ProjectivePoint elem) Fügt ein ProjectivePoint in das BigProjectivePointArray an Position pos ein.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

BigProjectivePointArray

```
public BigProjectivePointArray(java.math.BigInteger l)
```

Konstruktor um ein BigProjectivePointArray für maximal l Elemente vom Typ ProjectivePoint zu erzeugen.

Method Detail

set

```
public void set(java.math.BigInteger pos,
                ProjectivePoint elem)
```

Fügt ein ProjectivePoint in das BigProjectivePointArray an Position pos ein.

---

get

```
public ProjectivePoint get(java.math.BigInteger pos)
```

Gibt aus dem BigProjectivePointArray den an Position pos liegenden ProjectivePoint zurück.

---

getArrayLength

```
public java.math.BigInteger getArrayLength()
```

Gibt die Größe des BigProjectivePointArrays zurück.

Class EllipticCurve

```
java.lang.Object
  └EllipticCurve
```

public class **EllipticCurve**
extends java.lang.Object

Klasse, um elliptischen Kurven darstellen und mit ihnen rechnen zu können.

Constructor Summary

EllipticCurve(java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p)	Konstruktor, um eine elliptische Kurve aus den Kurvenparametern a, b, p zu erzeugen.
EllipticCurve(java.lang.String typ, java.math.BigInteger param, java.math.BigInteger x, java.math.BigInteger y, java.math.BigInteger p)	Konstruktor, um eine elliptische Kurve aus a,x,y,p oder aus b,x,y,p zu erzeugen.
EllipticCurve(java.lang.String a, java.lang.String b, java.lang.String p)	Konstruktor, um eine elliptische Kurve aus den Kurvenparametern a, b, p zu erzeugen.

Method Summary

ProjectivePoint	**add**(ProjectivePoint P, ProjectivePoint Q) Addiert zwei projektive Punkte und liefert den resultierenden Punkt zurück.
java.math.BigInteger	**getOrder_ByNaive**() Zählt die Anzahl der Punkte auf der betrachteten elliptischen Kurve mit dem naiven Algorithmus.

java.math.BigInteger	**getOrder_BySchoof**() Zählt die Anzahl der Punkte auf der betrachteten elliptischen Kurve mit dem Schoof Algorithmus.
java.math.BigInteger	**getOrder_ByShanks**() Zählt die Anzahl der Punkte auf der betrachteten elliptischen Kurve mit dem Shanks Mestre Algorithmus.
ProjectivePoint	**getRandomPoint**() Bestimmt zufällig einen Punkt der betrachteten elliptischen Kurve.
boolean	**isCurvePoint**(ProjectivePoint P) Überprüft, ob ein Punkt auf der vorliegenden elliptischen Kurve liegt.
boolean	**isEqual**(ProjectivePoint P, ProjectivePoint Q) Überprüft, ob zwei projektive Punkte dem gleichen affinen Punkt entsprechen.
ProjectivePoint	**mul**(java.math.BigInteger k, ProjectivePoint P) Berechnet die skalare Multiplikation eines Punktes mit einem Skalar.
ProjectivePoint	**mul**(int k, ProjectivePoint P) Berechnet die skalare Multiplikation eines Punktes mit einem Skalar.
ProjectivePoint	**mul**(java.lang.String k, ProjectivePoint P) Berechnet die skalare Multiplikation eines Punktes mit einem Skalar.
ProjectivePoint	**negate**(ProjectivePoint P) Negiert den übergebenen proejektiven Punkt und gibt ihn zurück.
ProjectivePoint	**sub**(ProjectivePoint P, ProjectivePoint Q) Berechnet die Subtraktion zweier projektiver Punkte und liefert den resultierenden Punkt zurück.
java.lang.String	**toAffineString**(ProjectivePoint P) Gibt einen projektiven Punkt in affiner Form als String zurück.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

EllipticCurve

```
public EllipticCurve(java.math.BigInteger a,
```

```
                    java.math.BigInteger b,
                    java.math.BigInteger p)
```

Konstruktor, um eine elliptische Kurve aus den Kurvenparametern a, b, p zu erzeugen. Die Parameter a und b sind hierbei die Kurvenparameter von y^2 = x^3 + ax + b. Der Parameter p gibt die Anzahl der Elemente des betrachteten Körpers F_p an. Alle Parameter werden hierbei als BigInteger Werte übergeben.

EllipticCurve

```
public EllipticCurve(java.lang.String a,
                     java.lang.String b,
                     java.lang.String p)
```

Konstruktor, um eine elliptische Kurve aus den Kurvenparametern a, b, p zu erzeugen. Die Parameter a und b sind hierbei die Kurvenparameter von y^2 = x^3 + ax + b.Der Parameter p gibt die Anzahl der Elemente des betrachteten Körpers F_p an. Alle Parameter werden hierbei als String Werte übergeben.

EllipticCurve

```
public EllipticCurve(java.lang.String typ,
                     java.math.BigInteger param,
                     java.math.BigInteger x,
                     java.math.BigInteger y,
                     java.math.BigInteger p)
```

Konstruktor, um eine elliptische Kurve aus a,x,y,p oder aus b,x,y,p zu erzeugen. Der Parameter typ kann entweder "axyp" oder "bxyp" sein. Er gibt an, ob zusätzlich zu einem affinen Kurvenpunkt (x,y) der Kurvenparameter a oder b in param übergeben wird. Der Parameter param ist also entweder der Parameter a oder b der Kurve y^2 = x^3 + ax + b. Der Parameter p gibt die Anzahl der Elemente des betrachteten Körpers F_p an.

Method Detail

isEqual

```
public boolean isEqual(ProjectivePoint P,
                       ProjectivePoint Q)
```

Überprüft, ob zwei projektive Punkte dem gleichen affinen Punkt entsprechen.

---

add

```
public ProjectivePoint add(ProjectivePoint P,
                           ProjectivePoint Q)
```

Addiert zwei projektive Punkte und liefert den resultierenden Punkt zurück.

---

toAffineString

```
public java.lang.String toAffineString(ProjectivePoint P)
```

Gibt einen projektiven Punkt in affiner Form als String zurück.

---

negate

```
public ProjectivePoint negate(ProjectivePoint P)
```

Negiert den übergebenen proejektiven Punkt und gibt ihn zurück.

---

sub

```
public ProjectivePoint sub(ProjectivePoint P,
                           ProjectivePoint Q)
```

Berechnet die Subtraktion zweier projektiver Punkte und liefert den resultierenden Punkt zurück.

---

mul

```
public ProjectivePoint mul(int k,
                           ProjectivePoint P)
```

Berechnet die skalare Multiplikation eines Punktes mit einem Skalar. Das Skalar k wird hierbei als integer übergeben.

---

mul

```
public ProjectivePoint mul(java.lang.String k,
                           ProjectivePoint P)
```

Berechnet die skalare Multiplikation eines Punktes mit einem Skalar. Das Skalar k wird hierbei als String übergeben.

---

mul

```
public ProjectivePoint mul(java.math.BigInteger k,
                           ProjectivePoint P)
```

Berechnet die skalare Multiplikation eines Punktes mit einem Skalar. Das Skalar k wird hierbei als BigInteger übergeben.

---

isCurvePoint

```
public boolean isCurvePoint(ProjectivePoint P)
```

Überprüft, ob ein Punkt auf der vorliegenden elliptischen Kurve liegt.

---

getRandomPoint

```
public ProjectivePoint getRandomPoint()
```

Bestimmt zufällig einen Punkt der betrachteten elliptischen Kurve. Es können alle Punkte außer der Punkt im Unendlichen zurück gegeben werden.

---

getOrder_ByNaive

```
public java.math.BigInteger getOrder_ByNaive()
```

Zählt die Anzahl der Punkte auf der betrachteten elliptischen Kurve mit dem naiven Algorithmus.

---

getOrder_ByShanks

```
public java.math.BigInteger getOrder_ByShanks()
```

Zählt die Anzahl der Punkte auf der betrachteten elliptischen Kurve mit dem Shanks Mestre Algorithmus.

---

getOrder_BySchoof

```
public java.math.BigInteger getOrder_BySchoof()
```

Zählt die Anzahl der Punkte auf der betrachteten elliptischen Kurve mit dem Schoof Algorithmus.

Class IOFunction

```
java.lang.Object
  └ IOFunction
```

public class **IOFunction**
extends java.lang.Object

Klasse zur Speicherung von Ergebnisdaten in einer Datei.

Constructor Summary

IOFunction()

Method Summary

static void	**clear**(java.lang.String datei) Löscht den Inhalt einer Datei.
static java.lang.String	**read**(java.lang.String datei) Liest den Inhalt einer Datei und gibt ihn als String zurück.
static void	**write**(java.lang.String s, java.lang.String datei) Speichert einen String in einer Datei ohne Zeilenumbruch.
static void	**writeln**(java.lang.String s, java.lang.String datei) Speichert einen String in einer Datei mit Zeilenumbruch.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

IOFunction

```
public IOFunction()
```

Method Detail

write

```
public static void write(java.lang.String s,
                         java.lang.String datei)
```

Speichert einen String in einer Datei ohne Zeilenumbruch.

---

writeln

```
public static void writeln(java.lang.String s,
                           java.lang.String datei)
```

Speichert einen String in einer Datei mit Zeilenumbruch.

---

read

```
public static java.lang.String read(java.lang.String datei)
```

Liest den Inhalt einer Datei und gibt ihn als String zurück.

---

clear

```
public static void clear(java.lang.String datei)
```

Löscht den Inhalt einer Datei.

Class MathFunction

```
java.lang.Object
  └ MathFunction
```

public class **MathFunction**
extends java.lang.Object

Klasse für mathemathische Hilfsfunktion.

Field Summary

static boolean	**canPrint** Variable, um die Ausgaben der print Funktion ein und aus schalten zu können.

Constructor Summary

MathFunction()

Method Summary

static java.math.BigInteger	**ceil**(java.math.BigDecimal n) Rundet eine gebrochene Zahl auf die nächste ganze Zahl auf und liefert diese zurück.
static java.math.BigInteger	**floor**(java.math.BigDecimal n) Rundet eine gebrochene Zahl auf die nächste ganze Zahl ab und liefert diese zurück.

static java.math.BigInteger	**getCRS**(BigIntegerArray r, java.math.BigInteger numberOfPrimes, java.math.BigInteger m) Berechnet die Lösung eines Kongruenzsystems mit dem Chinesischen Restsatz.
static java.math.BigInteger[]	**getGCDEX**(java.math.BigInteger a, java.math.BigInteger b) Berechnet den erweiterten Euklidischen Algorithmus von zwei ganzen Zahlen vom Typ BigInteger.
static int	**getLegendre**(java.math.BigInteger a, java.math.BigInteger p) Liefert den Wert des Legendre - Symbols von (a/p).
static java.math.BigInteger	**getRandomNumber**(java.math.BigInteger p) Liefert eine zufällige Zahl kleiner als p zurück.
static java.math.BigInteger	**getRandomNumberOdd**(java.math.BigInteger p) Liefert eine zufällige ungeraden Zahl kleiner als p zurück.
static java.math.BigInteger	**getSqrt**(java.math.BigInteger a, java.math.BigInteger p) Liefert die Quadratwurzel von a modulo p.
static java.math.BigDecimal	**nSqrt**(int n, java.math.BigInteger a) Liefert die n-te Einheitswurzel von y mittels Newton Verfahren zurück.
static void	**print**(java.lang.String s) Erzeugt eine Konsolenausgabe wie System.out.println, diese kann aber durch die Variable canPrint ein und ausgeschaltet werden.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

canPrint

public static boolean **canPrint**

Variable, um die Ausgaben der print Funktion ein und aus schalten zu können.

Constructor Detail

MathFunction

```
public MathFunction()
```

Method Detail

getRandomNumberOdd

```
public static java.math.BigInteger getRandomNumberOdd(java.math.
BigInteger p)
```

Liefert eine zufällige ungeraden Zahl kleiner als p zurück.

---

getRandomNumber

```
public static java.math.BigInteger getRandomNumber(java.math.
BigInteger p)
```

Liefert eine zufällige Zahl kleiner als p zurück.

---

getLegendre

```
public static int getLegendre(java.math.BigInteger a,
                              java.math.BigInteger p)
```

Liefert den Wert des Legendre - Symbols von (a/p).

---

getSqrt

```
public static java.math.BigInteger getSqrt(java.math.BigInteger a,
                                           java.math.BigInteger p)
```

Liefert die Quadratwurzel von a modulo p.

---

nSqrt

```
public static java.math.BigDecimal nSqrt(int n,
                                         java.math.BigInteger a)
```

Liefert die n-te Einheitswurzel von y mittels Newton Verfahren zurück.

---

ceil

```
public static java.math.BigInteger ceil(java.math.BigDecimal n)
```

Rundet eine gebrochene Zahl auf die nächste ganze Zahl auf und liefert diese zurück. Es erfolgt dabei eine Typkomvertierung von BigDecimal auf BigInteger.

---

floor

```
public static java.math.BigInteger floor(java.math.BigDecimal n)
```

Rundet eine gebrochene Zahl auf die nächste ganze Zahl ab und liefert diese zurück. Es erfolgt dabei eine Typkomvertierung von BigDecimal auf BigInteger.

---

print

```
public static void print(java.lang.String s)
```

Erzeugt eine Konsolenausgabe wie System.out.println, diese kann aber durch die Variable canPrint ein und ausgeschaltet werden.

---

getCRS

```
public static java.math.BigInteger getCRS(BigIntegerArray r,
                                          java.math.
```

```
BigInteger numberOfPrimes,
                                    java.math.BigInteger m)
```

Berechnet die Lösung eines Kongruenzsystems mit dem Chinesischen Restsatz. Von einem Kongruenzsystems der Form x = x_i mod l werden die x_i Werte durch das BigIntegerArray r übergeben. Die Anzahl der Gleichungen wird durch numberOfPrimes festgelegt. Mit m wird das Produkt der Primzahlen übergeben. Die Funktion geht davon aus, dass in dem System die Werte für l die ersten numberOfPrimes Primzahlen sind.

---

getGCDEX

```
public static java.math.BigInteger[] getGCDEX(java.math.
BigInteger a,
                                               java.math.
BigInteger b)
```

Berechnet den erweiterten Euklidischen Algorithmus von zwei ganzen Zahlen vom Typ BigInteger.

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Polynom

```
java.lang.Object
  └ Polynom
```

public class **Polynom**
extends java.lang.Object

Klasse, um Polynome der Form f(x) = a_n X^n + a_n-1 X^n-1 + ... + a_1 X + a_0 darzustellen und mit ihnen rechnen zu können.

Constructor Summary

Polynom(java.math.BigInteger deg) Konstruktor, zur Erzeugung eines Polynoms vom Grad deg.	

Method Summary

static Polynom	**add**(Polynom a, Polynom b, java.math.BigInteger p) Addiert zwei Polynome und liefert das resultierende Polynom zurück.
static Polynom []	**div**(Polynom a, Polynom b, java.math.BigInteger p) Berechnet die euklidische Division von zwei Polynomen (a/b) und liefert das Ergebnis und den Rest in einem Array vom Typ Polynom zurück.
java.math.BigInteger	**get**(java.math.BigInteger pos) Gibt den Koeffizienten des Polynoms an Position pos zurück.
static Polynom	**getAlpha**(java.math.BigInteger pl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet alpha.

static Polynom	**getBeta**(java.math.BigInteger pl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet beta.
java.math.BigInteger	**getDeg**() Gibt den Grad des Polynoms zurück.
static Polynom	**getDivisionPolynom**(java.math.BigInteger i, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p) Berechnet das i-te reduzierte Divisionspolynom f_i.
static Polynom	**getDx**(Polynom Dx1, java.math.BigInteger pl, java.math.BigInteger tl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet das Polynom Dx.
static Polynom	**getDx1**(java.math.BigInteger pl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet das Polynom Dx1.
static Polynom	**getDy**(Polynom Dy1, java.math.BigInteger pl, java.math.BigInteger tl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l)
static Polynom	**getDy1**(java.math.BigInteger pl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet das Polynom Dy1.
static Polynom	**getGxHx**(java.lang.String type, java.math.BigInteger pl, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet das Polynom g(x) oder das Polynom h(x).
static Polynom	**getK**(java.math.BigInteger w, java.math.BigInteger a, java.math.BigInteger b, java.math.BigInteger p, java.math.BigInteger l) Berechnet das Polynom k.
static Polynom	**ggT**(Polynom a, Polynom b, java.math.BigInteger p) Berechnet den ggT von zwei Polynomen a und b modulo einer Primzahl p und liefert das resultierende Polynom zurück.

static Polynom	**mod**(Polynom a, java.math.BigInteger p) Berechnet den Modulowert eines Polynoms a mit einer Primzahl p.
static Polynom	**mod**(Polynom poly1, Polynom poly2, java.math.BigInteger p) Berechnet Polynom poly1 modulo Polynom poly2.
static Polynom	**mul**(java.math.BigInteger n, Polynom x, java.math.BigInteger p) Berechnet das Produkt eines Skalars n und eines Polynoms x modulo einer Primzahl p.
static Polynom	**mul**(Polynom a, Polynom b, java.math.BigInteger p) Multipliziert zwei Polynome miteinander und liefert das resultierende Polynom zurück.
static Polynom	**negate**(Polynom x, java.math.BigInteger p) Negiert ein Polynom und liefert dieses zurück.
static Polynom	**pow**(Polynom x, java.math.BigInteger a, java.math.BigInteger p) Berechnet ein Polynom x hoch einem Skalar a modulo einer Primzahl p.
static Polynom	**prove**(Polynom x) Formatiert das Polynom, so dass am Anfang und am Ende keine Koeffizienten stehen, die 0 sind.
void	**set**(java.math.BigInteger pos, java.math.BigInteger elem) Ändert einen bestimmten Koeffizienten in dem Polynom.
static Polynom	**sub**(Polynom a, Polynom b, java.math.BigInteger p) Subtrahiert zwei Polynome voneinander und liefert das resultierende Polynom zurück.
java.lang.String	**toString**() Gibt das Polynom als formatierten String zurück.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Polynom

```
public Polynom(java.math.BigInteger deg)
```

Konstruktor, zur Erzeugung eines Polynoms vom Grad deg.

Method Detail

set

```
public void set(java.math.BigInteger pos,
                java.math.BigInteger elem)
```

Ändert einen bestimmten Koeffizienten in dem Polynom. Mit pos wird dabei die Position im Polynom angegeben und mit elem der Wert von a_pos.

---

get

```
public java.math.BigInteger get(java.math.BigInteger pos)
```

Gibt den Koeffizienten des Polynoms an Position pos zurück.

---

toString

```
public java.lang.String toString()
```

Gibt das Polynom als formatierten String zurück.

Overrides:
`toString` in class `java.lang.Object`

---

getDeg

```
public java.math.BigInteger getDeg()
```

Gibt den Grad des Polynoms zurück.

---

prove

```
public static Polynom prove(Polynom x)
```

Formatiert das Polynom, so dass am Anfang und am Ende keine Koeffizienten stehen, die 0 sind.

---

add

```
public static Polynom add(Polynom a,
                          Polynom b,
                          java.math.BigInteger p)
```

Addiert zwei Polynome und liefert das resultierende Polynom zurück.

---

negate

```
public static Polynom negate(Polynom x,
                             java.math.BigInteger p)
```

Negiert ein Polynom und liefert dieses zurück.

---

sub

```
public static Polynom sub(Polynom a,
                          Polynom b,
                          java.math.BigInteger p)
```

Subtrahiert zwei Polynome voneinander und liefert das resultierende Polynom zurück.

---

mul

```
public static Polynom mul(Polynom a,
                          Polynom b,
```

```
                                java.math.BigInteger p)
```

Multipliziert zwei Polynome miteinander und liefert das resultierende Polynom zurück.

---

mod

```
public static Polynom mod(Polynom a,
                          java.math.BigInteger p)
```

Berechnet den Modulowert eines Polynoms a mit einer Primzahl p. Dabei werden di Koeffizienten a_i modulo p berechnet.

---

div

```
public static Polynom[] div(Polynom a,
                            Polynom b,
                            java.math.BigInteger p)
```

Berechnet die euklidische Division von zwei Polynomen (a/b) und liefert das Ergebnis und den Rest in einem Array vom Typ Polynom zurück. An Position 0 steht das Ergebnis und an Position 1 der Rest.

---

ggT

```
public static Polynom ggT(Polynom a,
                          Polynom b,
                          java.math.BigInteger p)
```

Berechnet den ggT von zwei Polynomen a und b modulo einer Primzahl p und liefert das resultierende Polynom zurück.

---

getDivisionPolynom

```
public static Polynom getDivisionPolynom(java.math.BigInteger i,
                                         java.math.BigInteger a,
```

```
                                        java.math.BigInteger b,
                                        java.math.BigInteger p)
```

Berechnet das i-te reduzierte Divisionspolynom f_i.

---

pow

```
public static Polynom pow(Polynom x,
                          java.math.BigInteger a,
                          java.math.BigInteger p)
```

Berechnet ein Polynom x hoch einem Skalar a modulo einer Primzahl p.

---

mul

```
public static Polynom mul(java.math.BigInteger n,
                          Polynom x,
                          java.math.BigInteger p)
```

Berechnet das Produkt eines Skalars n und eines Polynoms x modulo einer Primzahl p.

---

mod

```
public static Polynom mod(Polynom poly1,
                          Polynom poly2,
                          java.math.BigInteger p)
```

Berechnet Polynom poly1 modulo Polynom poly2.

---

getGxHx

```
public static Polynom getGxHx(java.lang.String type,
                              java.math.BigInteger pl,
                              java.math.BigInteger a,
                              java.math.BigInteger b,
```

```
                                java.math.BigInteger p,
                                java.math.BigInteger l)
```

Berechnet das Polynom g(x) oder das Polynom h(x). Der Wert für type kann entweder "gx" oder "hx" sein, daraus wird das entsprechende Polynom berechnet.

---

getK

```
public static Polynom getK(java.math.BigInteger w,
                           java.math.BigInteger a,
                           java.math.BigInteger b,
                           java.math.BigInteger p,
                           java.math.BigInteger l)
```

Berechnet das Polynom k.

---

getAlpha

```
public static Polynom getAlpha(java.math.BigInteger pl,
                               java.math.BigInteger a,
                               java.math.BigInteger b,
                               java.math.BigInteger p,
                               java.math.BigInteger l)
```

Berechnet alpha.

---

getBeta

```
public static Polynom getBeta(java.math.BigInteger pl,
                              java.math.BigInteger a,
                              java.math.BigInteger b,
                              java.math.BigInteger p,
                              java.math.BigInteger l)
```

Berechnet beta.

---

getDx1

```
public static Polynom getDx1(java.math.BigInteger pl,
                             java.math.BigInteger a,
                             java.math.BigInteger b,
                             java.math.BigInteger p,
                             java.math.BigInteger l)
```

Berechnet das Polynom Dx1.

---

getDx

```
public static Polynom getDx(Polynom Dx1,
                            java.math.BigInteger pl,
                            java.math.BigInteger tl,
                            java.math.BigInteger a,
                            java.math.BigInteger b,
                            java.math.BigInteger p,
                            java.math.BigInteger l)
```

Berechnet das Polynom Dx.

---

getDy1

```
public static Polynom getDy1(java.math.BigInteger pl,
                             java.math.BigInteger a,
                             java.math.BigInteger b,
                             java.math.BigInteger p,
                             java.math.BigInteger l)
```

Berechnet das Polynom Dy1.

---

getDy

```
public static Polynom getDy(Polynom Dy1,
                            java.math.BigInteger pl,
                            java.math.BigInteger tl,
```

```
                                java.math.BigInteger a,
                                java.math.BigInteger b,
                                java.math.BigInteger p,
                                java.math.BigInteger l)
```

Class ProjectivePoint

```
java.lang.Object
  └ProjectivePoint
```

public class **ProjectivePoint**
extends java.lang.Object

Klasse, um projektive Punkte einer elliptischen Kurve darstellen zu können.

Constructor Summary

ProjectivePoint()	Konstruktor für den Punkt im Unendlichen O.
ProjectivePoint(java.math.BigInteger x, java.math.BigInteger y)	Konstruktor für affine Koordinaten vom Typ BigInteger.
ProjectivePoint(java.math.BigInteger x, java.math.BigInteger y, java.math.BigInteger z)	Konstruktor für projektive Koordinaten vom Typ BigInteger.
ProjectivePoint(java.lang.String x, java.lang.String y)	Konstruktor für affine Koordinaten vom Typ String.
ProjectivePoint(java.lang.String x, java.lang.String y, java.lang.String z)	Konstruktor für projektive Koordinaten vom Typ String.

Method Summary

java.math.BigInteger	**getX**() Liefert den Wert der x Koordinate des projektiven Punktes zurück.

java.math.BigInteger	**getY**() Liefert den Wert der y Koordinate des projektiven Punktes zurück.
java.math.BigInteger	**getZ**() Liefert den Wert der z Koordinate des projektiven Punktes zurück.
void	**setX**(java.math.BigInteger x) Setzt die x Koordinate des projektiven Punktes.
void	**setY**(java.math.BigInteger y) Setzt die y Koordinate des projektiven Punktes.
void	**setZ**(java.math.BigInteger z) Setzt die z Koordinate des projektiven Punktes.
java.lang.String	**toString**() Gibt den projektiven Punkt als String zurück.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

ProjectivePoint

```
public ProjectivePoint()
```

Konstruktor für den Punkt im Unendlichen O.

ProjectivePoint

```
public ProjectivePoint(java.math.BigInteger x,
                       java.math.BigInteger y,
                       java.math.BigInteger z)
```

Konstruktor für projektive Koordinaten vom Typ BigInteger.

ProjectivePoint

```
public ProjectivePoint(java.lang.String x,
                       java.lang.String y,
                       java.lang.String z)
```

Konstruktor für projektive Koordinaten vom Typ String.

ProjectivePoint

```
public ProjectivePoint(java.math.BigInteger x,
                       java.math.BigInteger y)
```

Konstruktor für affine Koordinaten vom Typ BigInteger.

ProjectivePoint

```
public ProjectivePoint(java.lang.String x,
                       java.lang.String y)
```

Konstruktor für affine Koordinaten vom Typ String.

Method Detail

setX

```
public void setX(java.math.BigInteger x)
```

Setzt die x Koordinate des projektiven Punktes.

---

setY

```
public void setY(java.math.BigInteger y)
```

Setzt die y Koordinate des projektiven Punktes.

---

setZ

```
public void setZ(java.math.BigInteger z)
```

Setzt die z Koordinate des projektiven Punktes.

---

getX

```
public java.math.BigInteger getX()
```

Liefert den Wert der x Koordinate des projektiven Punktes zurück.

---

getY

```
public java.math.BigInteger getY()
```

Liefert den Wert der y Koordinate des projektiven Punktes zurück.

---

getZ

```
public java.math.BigInteger getZ()
```

Liefert den Wert der z Koordinate des projektiven Punktes zurück.

---

toString

```
public java.lang.String toString()
```

Gibt den projektiven Punkt als String zurück.

Overrides:

`toString` in class `java.lang.Object`

Class Timer

```
java.lang.Object
  └ Timer
```

public class **Timer**
extends java.lang.Object

Klasse zur Zeitmessung von Algorithmen.

Constructor Summary

Timer()

Method Summary

java.lang.String	**getDuration**() Berechnet die Zeit, die zwischen dem starten und dem stoppen des Timers vergangen ist und liefert diese als formatierten String zurück.
void	**start**() Startet den Timer.
void	**stop**() Stoppt den Timer.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Timer

```
public Timer()
```

Method Detail

start

```
public void start()
```

Startet den Timer.

---

stop

```
public void stop()
```

Stoppt den Timer.

---

getDuration

```
public java.lang.String getDuration()
```

Berechnet die Zeit, die zwischen dem starten und dem stoppen des Timers vergangen ist und liefert diese als formatierten String zurück.

Class test

```
java.lang.Object
  └ test
```

public class **test**
extends java.lang.Object

Testklasse, um verschieden Funktion überprüfen zu können.

Constructor Summary

`test()`

Method Summary

`static void`	`main(java.lang.String[] args)`
`static void`	`testAlgorithm()`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

test

test

```
public test()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

---

testAlgorithm

```
public static void testAlgorithm()
```
