Institut für Formale Methoden der Informatik

Abteilung Theoretische Informatik

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2008

Faktorisierung mit elliptischen Kurven

Jörg Oberfell

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. habil. Volker Diekert
Betreuer:	Dipl.-Inf. Manfred Kufleitner
begonnen am:	18. Mai 2005
beendet am:	8. September 2005
CR-Klassifikation:	F.2.1, I.1.2

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1

2 Grundlagen **3**
2.1 Primzahleigenschaften 3
2.2 Eigenschaften von ganzen Zahlen 4
2.3 Kongruenzen 5

3 Die $p-1$ - Methode **7**
3.1 Mathematischer Hintergrund 7
3.2 Algorithmus 8
3.3 Laufzeit 9

4 Elliptische Kurven **11**
4.1 Einleitung 11
4.2 Arithmetik elliptischer Kurven 11
4.2.1 Addition, Subtraktion 12
4.2.2 Beispiele 13
4.2.3 Multiplikation 14
4.3 Pseudokurven 14

5 Elliptic Curve Method (ECM) **17**
5.1 Algorithmus 17
5.2 Funktionsweise 18
5.2.1 Schritt 2 19
5.2.2 Vorbereitung zu Schritt 4 19
5.2.3 Schritt 4 22
5.3 Laufzeitanalyse 22
5.3.1 Gruppenordnung elliptischer Kurven 22
5.3.2 Laufzeitbestimmung 23

6 Implementierung **27**
6.1 Die Datei ecm.cc 27
6.2 Die Datei ecmethod.cc 28
6.2.1 Die Funktion findCurve 29
6.2.2 Die Funktion logP 30
6.2.3 Die Funktion primePowerMultipliers 30
6.2.4 Die Funktion completeFactorization 31
6.2.5 Die Funktionen ecm 31
6.3 Die Datei EllipticCurve.cc 32
6.3.1 Additive Funktionen 32
6.3.2 Multiplikation 33
6.4 Weitere Hilfsprozeduren 33

7 Beispiele **37**
7.1 Laufzeiten abhängig von der Suchgrenze 38
7.2 Tests mit RSA-Zahlen 38
7.3 Faktorisierung größerer Zahlen 40

8 Ausblick **43**

Literatur **45**

Literaturverzeichnis **45**

Kapitel 1

Einleitung

Diese Studienarbeit behandelt die Faktorisierung großer Zahlen. Der Schwerpunkt der hier vorliegenden Arbeit liegt auf der im Jahr 1985 von H. W. Lenstra vorgestellten Elliptic Curve Method, einer Faktorisierungsmethode, die auf elliptischen Kurven aufbaut. Sie geht auf ein älteres Faktorisierungsverfahren, die so genannte $p-1$-Methode, zurück, die um 1974 von John M. Pollard entwickelt wurde.

Die Elliptic Curve Method ermöglicht es, Zahlen nahezu beliebiger Größe zu faktorisieren, sofern deren Primfaktoren nicht wesentlich mehr als ca. 20 bis 30 Dezimalstellen besitzen. Für Zahlen dieser Art ist sie anderen bekannten Faktorisierungsalgorithmen überlegen.

Die Studienarbeit ist Teil eines Softwareprojekts in Zusammenarbeit mit zwei weiteren Studenten. Im Rahmen dieses Projekts sind zwei Faktorisierungsalgorithmen (das Quadratische Sieb und das hier vorliegende Verfahren mit elliptischen Kurven) und ein Algorithmus zum Primzahltest (Elliptic Curve Primality Proving) entstanden.

1.1 Motivation

Die Faktorisierung von großen Zahlen ist ein wichtiger Forschungsgegenstand. Von der Größe der faktorisierbaren Zahlen hängen zahlreiche Anwendungen, u. a. in der Kryptographie, ab. Als Beispiel sei das RSA-Verschlüsselungsverfahren genannt. Ergibt sich hier die Möglichkeit, den privaten, geheimen Schlüssel zu knacken, ist der Einsatz des RSA-Systems sinnlos. Daher untersucht die Wissenschaft verschiedene Möglichkeiten der Faktorisierung großer Zahlen, um damit herauszufinden, bis zu welcher Größenordnung diese Zahlen in angemessener Zeit zerlegbar, und damit für Verschlüsselungsverfahren unbrauchbar sind.

In dieser Arbeit geht es um die Implementierung eines modernen Verfahrens zur Faktorisierung, der Elliptic Curve Method. Zunächst werden mathematische Grundlagen eingeführt.

In Kapitel 3 wird die $p-1$-Methode erläutert. Kapitel 4 führt in die Theorie der elliptischen Kurven ein, soweit sie für das Verständnis dieser Ausarbeitung nötig ist. In Kapitel 5 folgt schließlich die Elliptic Curve Method. Die anschließenden Kapitel gehen auf die Implementierungsdetails und Tests der Programme ein.

Kapitel 2

Grundlagen

In diesem Kapitel werden einige mathematische Grundbegriffe eingeführt. Der im Gebiet der Zahlentheorie erfahrene Leser kann dieses Kapitel überspringen und lediglich als Nachschlagewerk verwenden. Im Verlauf der Ausarbeitung wird an den entsprechenden Stellen auf diese Definitionen und Sätze verwiesen.

2.1 Primzahleigenschaften

Zunächst soll das Legendre-Symbol[1] definiert werden. Es gibt an, ob eine Zahl a ein Quadrat modulo p ist.

Definition 2.1. *Sei $a \in \mathbb{N}$, $p > 2$ prim. Das Legendre-Symbol wird definiert durch*

$$\left(\frac{a}{p}\right) := \begin{cases} 0 & \textit{falls } a \equiv 0 \ (\textit{mod } p) \\ 1 & \textit{falls } a \textit{ ein Quadrat mod } p \textit{ ist} \\ -1 & \textit{falls } a \textit{ kein Quadrat mod } p \textit{ ist} \end{cases}$$

Für das Legendre-Symbol gilt folgende Berechnung nach Euler:

$$(a/p) \equiv a^{(p-1)/2} \pmod{p}$$

Definition 2.2. *Sei $k \in \mathbb{N}$. Dann ist*

$$\pi(k) := \#\{p \,|\, p \textit{ prim} \wedge p \leq k\}$$

die Anzahl der Primzahlen bis einschließlich k.

Das folgende Primzahltheorem (prime number theorem, PNT) nach Hadamard und de la Vallée Poussin, das von beiden unabhängig voneinander 1896 bewiesen wurde, gibt die

[1] Adrien-Marie Legendre, 1752 – 1833, französischer Mathematiker

Anzahl der Primzahlen bis zu einer vorgegebenen Schranke an. Beweise finden sich in [8] oder [1]. Der hier nicht weiter verwendete zweite Teil des Theorems geht auf Tschebyscheff zurück. Es sei angemerkt, dass noch exaktere Abschätzungen existieren.

Theorem 2.3 (Primzahltheorem). *Für $x \to \infty$ verhält sich die Anzahl $\pi(x)$ der Primzahlen kleiner als x wie $x/\ln x$. Genauer: Es gilt für $x \geq 3$ die Abschätzung $\frac{1}{2} \cdot \frac{x}{\ln x} < \pi(x) < 2 \cdot \frac{x}{\ln x}$.*

2.2 Eigenschaften von ganzen Zahlen

Definition 2.4. *Sei $a \in \mathbb{Z}$ und $b \in \mathbb{N}^+$.*

$$a \text{ heißt } b\text{-glatt} \quad \rightleftharpoons \quad a = \prod_i p_i^{a_i} \text{ mit } p_i \leq b,\ p_i \text{ prim und } a_i \geq 1$$

Man sagt auch "a ist b-smooth", oder kürzer "a hat nur 'kleine' (Prim-)Faktoren".

Definition 2.5. *Seien $x, y \in \mathbb{N}$. Dann ist $\psi(x, y) := \#\{n \mid 1 \leq n \leq x \ \wedge \ n \text{ ist } y\text{-glatt}\}$ die Anzahl der y-glatten Zahlen bis einschließlich x.*

Theorem 2.6 (Dickman, 1930). *Für jede feste reelle Zahl $u > 0$ gibt es eine reelle Zahl $\rho(u) > 0$ derart, dass*

$$\psi(x,\, x^{1/u}) \sim \rho(u) \cdot x$$

Mit diesem Theorem lässt sich die Funktion $\rho(u)$ numerisch abschätzen. Damit erhält man $\ln \rho(u) \sim -u \ln u$ und für die Anzahl der glatten Zahlen ergibt sich daher

$$\psi(x, x^{1/u}) \sim \frac{u}{e^u} \cdot x$$

Um die Laufzeit der ECM-Methode zu beweisen, ist ein weiterer Satz nötig. Ein Beweis dazu findet sich in [2]. Es gilt

$$\psi(x,\, x^{1/u}) = xu^{-u+o(u)} \text{ für } u \to \infty \text{ und } u < \frac{(1-\epsilon)\ln x}{\ln \ln x}$$

Für $\psi(x, y)$ mit $y > \ln^{1+\epsilon} x$ und x groß gilt also die Abschätzung

$$\psi(x,\, y) = xu^{-u+o(u)}$$

2.3 Kongruenzen

Lemma 2.7. *Sei $n \in \mathbb{N}$. Dann gilt:*

$$\forall x > 0,\ ggT(x,n) = 1: \quad a \not\equiv b\ (mod\ n) \Rightarrow xa \not\equiv xb\ (mod\ n)$$

Satz 2.8 (kleiner Fermat). *Sei $a \in \mathbb{Z}$, p prim. Dann gilt*

$$a^p \equiv a \pmod p$$

Ist zusätzlich $a \not\equiv 0 \pmod p$, so gilt

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

Beweis. Sei $a \not\equiv 0 \pmod p$ und $M := \{1, \ldots, p-1\}$ die Menge aller Reste modulo p ohne die Null. Es ist klar, dass je zwei Elemente aus M nicht zueinander kongruent sind. Definiere nun $N := aM := \{am \mid m \in M\}$ für ein $a \not\equiv 0 \pmod p$. Dann sind nach Lemma 2.7 auch in N je zwei Elemente zueinander nicht kongruent. Jede dieser Mengen enthält also $p-1$ Elemente, die paarweise (modulo p) nicht kongruent sind. Es gibt aber auch maximal $p-1$ Elemente in $M = \mathbb{F}_p^* = N$. Daher ist das Produkt der Elemente aus M kongruent zum Produkt der Elemente aus N, also

$$\prod_{m \in M} m \equiv \prod_{n \in N} n \pmod p$$

und ein Vergleich der Elemente der Produkte ergibt (modulo p):

$$\begin{aligned} 1 \cdot 2 \cdot \ldots \cdot (p-1) &\equiv (1 \cdot a) \cdot (2 \cdot a) \cdot \ldots \cdot ((p-1) \cdot a) \\ &= 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot a^{p-1} \end{aligned}$$

und daher

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

für $a \not\equiv 0 \pmod p$. Somit ist der zweite Teil des Satzes gezeigt. Daraus ergibt sich sofort der erste Teil des Satzes, wobei hier zusätzlich der triviale Fall $a \equiv 0$ erlaubt ist. ■

Kapitel 3

Die $p-1$ - Methode

Sei $n = \prod_{i=1}^{h} p_i^{a_i}$ mit $h \geq 2, a_i \geq 1$ und $3 \leq p_1 < \ldots < p_h$, p_i prim, eine (ungerade, zusammengesetzte) natürliche Zahl, die faktorisiert werden soll. Die $p-1$-Methode nach John M. Pollard versucht, einen nichttrivialen Faktor von n zu finden.

Der folgende Abschnitt beginnt mit der mathematischen Erläuterung der $p-1$-Methode. In Abschnitt 3.2 wird der Algorithmus aufgeführt und beschrieben. Anschließend folgt eine kurze Abschätzung der Laufzeit der $p-1$-Methode.

3.1 Mathematischer Hintergrund

Sei p ein (gesuchter) Primfaktor von n, und M vorläufig eine natürliche Zahl, die später genauer definiert werden wird. Im Folgenden treffen wir zwei Annahmen:

- Angenommen, $p-1$ teilt M, d. h. es ist $M = a \cdot (p-1)$ für ein $a > 1$, dann gilt nach dem kleinen Satz von Fermat (2.8):

$$2^M = 2^{a(p-1)} = (2^a)^{p-1} \equiv 1 \pmod{p}$$

- Angenommen, es ist zusätzlich $2^M \not\equiv 1 \pmod{n}$, dann gilt für den größten gemeinsamen Teiler:

$$1 < \left(2^M - 1, n\right) < n$$

  Beweis. Zunächst zur linken Ungleichung: Nach der ersten Annahme ist $2^M - 1 \equiv 0 \pmod{p}$, also ist p ein Teiler von $2^M - 1$. Außerdem ist p nach Voraussetzung ein (noch unbekannter) Teiler von n. Somit ist der ggT $\geq p$ und insbesondere größer als Eins. Da der $\mathrm{ggT}(a, b) \leq \min\{a, b\}$ ist und da $2^M - 1$ nach der zweiten Annahme nicht von n geteilt wird, ist auch die rechte Ungleichung erfüllt. ■

Pollard hatte 1974 die Idee, M so zu wählen, dass M aus möglichst vielen kleinen (d. h. k-glatten) Teilern besteht, so dass große Hoffnung besteht, dass auch $p-1$ von diesen Teilern

zerlegt wird. Dadurch, dass M sehr viele Teiler enthält, werden alle $q-1 \in \{\prod_{a \in N} a \mid \emptyset \neq N \subseteq M\}$ gleichzeitig untersucht, wobei die verschiedenen q potentielle Teiler von n sind. Definiere daher für eine geeignete Grenze $k \in \mathbb{N}$

$$M := M(k) := \mathrm{kgV}(1, \ldots, k)$$

3.2 Algorithmus

Der unten abgedruckte $p-1$-Algorithmus von Pollard versucht einen (nicht unbedingt primen) Faktor g von n zu finden. Bei zu kleiner Wahl der Grenze k kann es vorkommen, dass der Algorithmus nur triviale Faktoren von n findet. Daher sollte der Algorithmus wiederholt mit wachsender Grenze k aufgerufen werden, bis ein nichttrivialer Faktor von n entdeckt wird.

Algorithmus 3.1 (Pollard $p-1$ Faktorisierungsmethode).

Eingabe: *eine ungerade, positive Zahl n, die faktorisiert werden soll, sowie eine "geeignete" Suchgrenze k.*

Ausgabe: *ein evtl. zusammengesetzter Faktor g von n, der auch trivial sein kann.*

1. [Primpotenzbasis $M(k)$ erzeugen]
   Bestimme alle Primzahlen $p_1 < ... < p_m \leq k$;
   Bestimme zu jedem p_i ein a_i mit $a_i = \max\{a \in \mathbb{N} \mid p_i^a \leq k\}$;

2. [Berechne $c^{M(k)}$ mod n]
   c := 2; //oder ein zufälliger Wert c ∈ [2, n – 2]
   for (i = 1 .. m)
   for (j = 1 .. a_i)
   c := c^{p_i} mod n;

3. [ggT berechnen]
   g := ggT(c – 1, n);
   return g;

Die Berechnung des kgV's kann unterschiedlich erfolgen. Es gilt

$$\begin{aligned} M(k) &= \mathrm{kgV}(1, \ldots, k) \\ &= 2^{\lfloor \log_2 k \rfloor} \cdot 3^{\lfloor \log_3 k \rfloor} \cdot 5^{\lfloor \log_5 k \rfloor} \cdot \ldots \cdot q^{\lfloor \log_q k \rfloor} \end{aligned}$$

mit q prim, $q \leq k$ und $\nexists\, q_2$ prim: $q < q_2 \leq k$. Eine Möglichkeit besteht im Auffinden aller Primzahlen im Intervall $[2, k]$ und der jeweils passenden maximalen Potenzen a_i, so dass $\forall i: \; p_i^{a_i} \leq k$ gilt. Dieses Verfahren benutzt der obige Algorithmus.

Eine andere (rekursive) Methode schlägt [4] vor: Angenommen $M(k)$ existiert bereits, so lässt sich $M(k+1)$ folgendermaßen berechnen:

$$M(k+1) = \begin{cases} p \cdot M(k) & \text{falls } \exists p \text{ prim}, a \geq 1, \text{ so dass gilt} : k+1 = p^a \\ M(k) & \text{sonst} \end{cases}$$

Probleme: Der Algorithmus kann fehlschlagen, d. h. nur triviale Faktoren finden. In der Praxis tritt vor allem der triviale Faktor 1 auf, vgl. [4]. Hier kann eine Erweiterung der Primpotenzbasis, also die Erhöhung der Suchgrenze k, zu besseren Ergebnissen führen. Bei Auftreten des anderen trivialen Faktors n ist dagegen eine andere Wahl des Parameters c erfolgsversprechend.

Beispiel (nach [4]): Für $n = 2047 = 2^{11} - 1 = 23 \cdot 89$ ist

$$\text{ggT}(2^M - 1, n) = \begin{cases} n & \text{falls } 11|M \\ 1 & \text{sonst} \end{cases}$$

d. h. für $k \leq 10$ wird der triviale Faktor 1 errechnet, und eine beliebige Erhöhung von k ab dem Wert 11 führt immer auf den zweiten trivialen Faktor n.
Eine ggf. unendliche Erhöhung der Suchgrenze k bringt also in diesem Beispiel rein gar nichts. Die Variation des Parameters c führt hier zwar auch nicht sofort zum Erfolg, da für $c = 2, \ldots, 11$ keine Faktoren gefunden werden, aber eine zufällige Wahl von c kann hier weiterhelfen – beispielsweise führt $c = 12$ zum Faktor $\text{ggT}(12^{M(8)} - 1, n) = 89$.

In der Literatur wird ein zweiter Suchschritt vorgeschlagen, der nach einigen fehlgeschlagenen Durchläufen gestartet werden kann. Darauf soll hier aber nicht weiter eingegangen werden.

3.3 Laufzeit

Zur Berechnung von $c^{M(k)}$ [Schritt 2] kann die schnelle Exponentiation verwendet werden. Die Laufzeit zur Berechnung einer Potenz a^b mittels der schnellen Exponentiation beträgt $O(\log b)$. Unter Verwendung des Primzahltheorems (Theorem 2.3) kann nun gezeigt werden, dass die Laufzeit von [Schritt 2] unter Verwendung der schnellen Exponentiation mit Exponent $b = M(k)$ ungefähr k Schritte benötigt.

Behauptung 3.1. *Die Berechnung von $c^{M(k)}$ kann mit $O(k) = O(\log M(k))$ Rechenschritten durchgeführt werden.*

Beweis. Es reicht zu zeigen: $k \sim \log M(k)$.

$$\begin{aligned}
\log M(k) &= \log(p_1^{a_1} \cdot \ldots \cdot p_x^{a_x}) \\
&= \log p_1^{a_1} + \ldots + \log p_x^{a_x} \\
&\leq \underbrace{\ln k + \ldots + \ln k}_{\pi(k)\text{-mal}} \\
&= \pi(k) \cdot \ln k \\
&\sim \tfrac{k}{\ln k} \cdot \ln k \qquad \text{nach Theorem 2.3} \\
&= k
\end{aligned}$$

■

Kapitel 4

Elliptische Kurven

4.1 Einleitung

Definition 4.1. *Sei K ein Körper der Charakteristik $\neq$ 2 und 3, und seien $a, b \in K$ mit $4a^3 + 27b^2 \neq 0$. Dann heißt die Menge*

$$E := E_{a,b}(K) := \{(x, y) \in K^2 \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

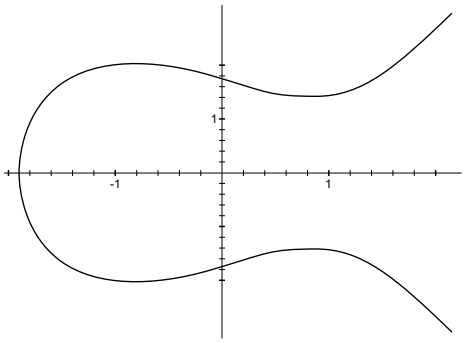
elliptische Kurve über K, wobei $\mathcal{O}$ der Punkt im Unendlichen ist.

Eine elliptische Kurve ist nach dieser Definition nicht-singulär, d. h. das Polynom $x^3 + ax + b$ hat keine mehrfachen Nullstellen. Dafür sorgt die Einschränkung $4a^3 + 27b^2 \neq 0$. Die Eigenschaft 'nicht-singulär' bei einer Gleichung dritten Grades führt dazu, dass jede Gerade durch zwei Punkte der Kurve $E(K)$ diese auch in einem dritten (nicht unbedingt verschiedenen) Punkt schneidet (ein tangentialer Punkt zählt doppelt). Diese Eigenschaft soll im nächsten Abschnitt zur Definition einer Gruppenoperation verwendet werden.

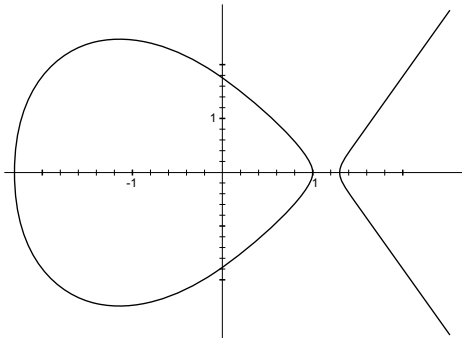
Die Gleichung $y^2 = x^3 + ax + b$ der Kurve wurde in affiner Form notiert. Generell kann man elliptische Kurven auch in projektiver Schreibweise darstellen, die affine Notation ist hier jedoch ausreichend. Abbildung 4.1 zeigt einige Beispiele von elliptischen Kurven über den Körpern $\mathbb{R}$ und $\mathbb{F}_7$.

4.2 Arithmetik elliptischer Kurven

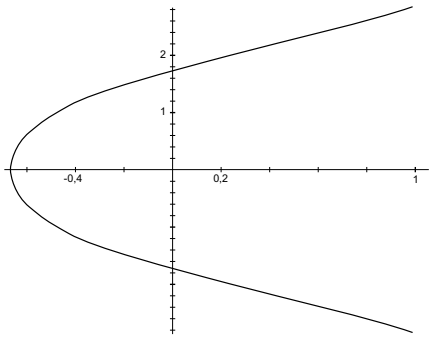
In Definition 4.1 wurden elliptische Kurven lediglich als Mengen von Punkten (x, y) über einem Körper K eingeführt. Durch Definition einer geeigneten Operation soll E nun zu einer Gruppe erweitert werden. Dabei dient der Punkt $\mathcal{O}$ als Neutralelement.



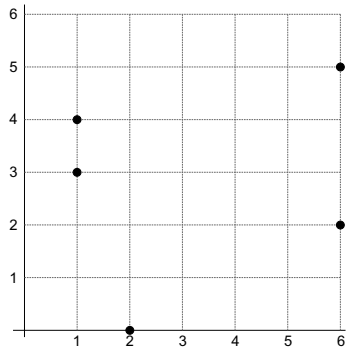
$y^2 = x^3 - 2x + 3$ über $\mathbb{R}$



$y^2 = x^3 - 4x + 3$ über $\mathbb{R}$



$y^2 = x^3 + 4x + 3$ über $\mathbb{R}$



$y^2 = x^3 - 2x + 3$ über $\mathbb{F}_7$

Abbildung 4.1: Schaubilder einiger elliptischer Kurven über $\mathbb{R}$ bzw. $\mathbb{F}_7$

4.2.1 Addition, Subtraktion

Wie oben beschrieben, schneidet eine Gerade die Kurve in genau drei Punkten, die P, Q und R genannt werden sollen. Der Punkt R wird an der x-Achse gespiegelt und S genannt. Die Addition zweier Punkte soll nun als Ergebnis diesen gespiegelten Punkt der Geraden liefern, d. h. $P + Q = -R = S$. Genauere Details liefert die folgende Definition.

Definition 4.2. *Sei K ein Körper mit $2 \neq \mathrm{char}\, K \neq 3$ und sei $E(K)$ eine elliptische Kurve, d. h. $E(K) = \{(x, y) \in K \times K \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$ für $a, b \in K$ mit $4a^3 + 27b^2 \neq 0$. Seien außerdem $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$ zwei (evtl. identische) Punkte der Kurve. Durch*

$$\begin{aligned}
-\mathcal{O} &:= \mathcal{O} \\
-P_1 &:= (x_1, -y_1) \\
P_1 + \mathcal{O} &:= P_1 \\
P_1 + P_2 &:= \begin{cases} \mathcal{O} & \textit{falls } P_2 = -P_1 \\ (x_3, y_3) & \textit{sonst} \end{cases}
\end{aligned}$$

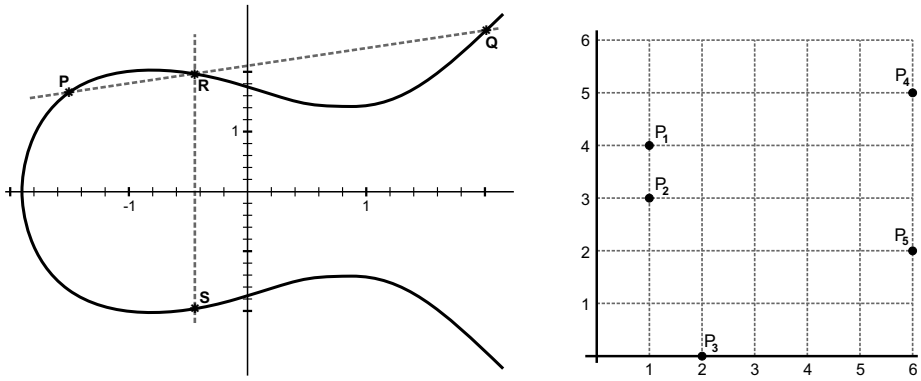
wird eine kommutative Addition '+' mit inverser Operation '−' definiert, wobei gilt:

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \\ m &= \begin{cases} \dfrac{3x_1^2 + a}{2y_1} & \textit{falls } x_1 = x_2 \\ \dfrac{y_2 - y_1}{x_2 - x_1} & \textit{sonst} \end{cases} \end{aligned}$$

Beweis. Ein Beweis der Gruppeneingenschaft findet sich u. a. in [9]. ■

4.2.2 Beispiele

Bei elliptischen Kurven über dem Körper $\mathbb{R}$ läuft die Addition zweier Punkte P_1 und P_2 anschaulich folgendermaßen ab: Die Gerade durch die beiden Punkte P und Q schneidet die Kurve in einem (nicht unbedingt verschiedenen) Punkt R. Durch Spiegelung dieses Punktes an der x-Achse erhält man einen weiteren Punkt S, der ebenfalls auf der Kurve liegt. Dieser ist das Ergebnis der Addition. Abbildung 4.2 zeigt dies an einem Beispiel. Der Sinn dieser Konstruktion ist, dass sich die drei Punkte zum Neutralelement aufaddieren, d. h. es gilt $P + Q + R = \mathcal{O}$ mit $R = -S$.

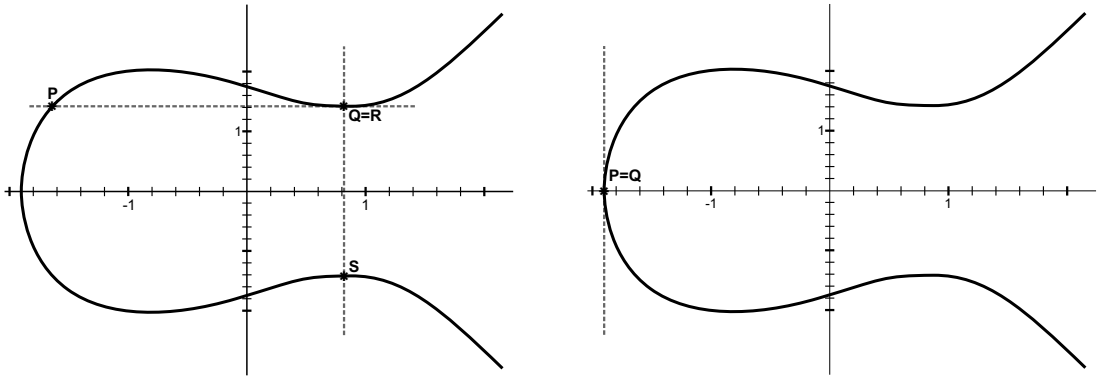


$y^2 = x^3 - 2x + 3$ über $\mathbb{R}$ $\qquad$ $y^2 = x^3 + 2x + 3$ über $\mathbb{F}_7$

Abbildung 4.2: Addition zweier Punkte auf einer elliptischen Kurve

Im linken Bild in Abb. 4.2 ist die Addition zweier beliebiger Punkte P und Q dargestellt. Das Ergebnis der Berechnung ist der Punkt $S = -R = P + Q$. Das rechte Bild zeigt weniger anschaulich die Addition in einem endlichen Körper. Hier gilt u. a.

$$\begin{aligned} P_1 &= -P_2 & \qquad P_3 &= -P_3 \\ P_1 + P_2 &= \mathcal{O} & \qquad P_1 + P_1 &= P_5 + P_5 = P_4 \\ P_1 + P_3 &= P_5 & \qquad P_2 + P_4 &= P_1 \end{aligned}$$



$y^2 = x^3 - 2x + 3$ über $\mathbb{R}$

Abbildung 4.3: Spezialfälle der Addition zweier Punkte auf einer elliptischen Kurve

Die Beispiele in Abb. 4.3 sind Spezialfälle. Hier fallen jeweils zwei Punkte zusammen. In beiden Fällen soll jeweils $S = P + Q$ berechnet werden. Im linken Bild ist $P = (-\sqrt{8/3}, y)$ und $Q = (\sqrt{2/3}, y)$ mit gleichen y-Werten. Da beide Punkte gleiche y-Werte haben, ist $m = 0$. Daher ist $x_3 = -\sqrt{2/3} + \sqrt{8/3} = \sqrt{2/3}$ und $y_3 = -y$ und somit $S = (\sqrt{2/3}, -y) = -Q$.

Das Beispiel im rechten Bild zeigt den Spezialfall $P = Q = (x, 0)$. Hier ist also $P = -Q$ und damit $P + Q = P + P = \mathcal{O}$.

4.2.3 Multiplikation

Basierend auf der Addition zweier Punkte kann man eine Multiplikation als k-fache Addition eines Punktes P definieren. Dabei ist K weiterhin ein Körper der Charakteristik $\neq 2, 3$.

Definition 4.3. *Sei $k \in \mathbb{Z}$, $P \in E_{a,b}(K)$. Dann ist*

$$[k]P := \begin{cases} [-k] \cdot (-P) & \textit{falls } k < 0 \\ \mathcal{O} & \textit{falls } k = 0 \\ \underbrace{P + \ldots + P}_{k\text{-mal}} & \textit{falls } k > 0 \end{cases}$$

die Multiplikation eines elliptischen Punktes P mit einem Skalar k.

4.3 Pseudokurven

Im nächsten Kapitel wird die Elliptic Curve Method zur Faktorisierung von Zahlen beschrieben. Die Berechnungen erfolgen dann nicht mehr in einem Körper der Form $\mathbb{F}_p$,

sondern im Ring $\mathbb{Z}_n$. Dennoch werden elliptische Kurven verwendet. Da die Operationen aber in einem Ring stattfinden, existieren nicht für alle Elemente a auch die Inversen a^{-1}. Daher soll nun Definition 4.1 erweitert werden.

Definition 4.4. *Sei $n \in \mathbb{N}$ mit $ggT(n, 6) = 1$, d. h. $2, 3 \nmid n$. Seien $a, b \in \mathbb{Z}_n$ mit $ggT(4a^3 + 27b^2, n) = 1$. Dann heißt die Menge*

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

elliptische **Pseudokurve**, *wobei $\mathcal{O}$ der Punkt im Unendlichen ist.*

Die Definition schließt den Fall n prim mit ein; eine elliptische Kurve ist also auch eine elliptische Pseudokurve. Auch auf dieser Menge wird wiederum eine Addition definiert, die Pseudokurve bildet jedoch keine Gruppe, falls n zusammengesetzt ist.

Definition 4.5. *Aufbauend auf Def. 4.2 ergeben sich bei der Addition mit einer Pseudokurve folgende Änderungen:*

$$P_1 + P_2 \quad := \quad \begin{cases} \mathcal{O} & \text{falls } P_2 = -P_1 \\ (x_3, y_3) & \text{falls } m \text{ definiert ist} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

wobei für m kleinere Einschränkungen nötig sind, die Formeln aber unverändert bleiben:

$$m \quad = \quad \begin{cases} \dfrac{3x_1^2 + a}{2y_1} & \text{falls } P_1 = P_2 \text{ und } ggT(2y_1, n) = 1 \\ \dfrac{y_2 - y_1}{x_2 - x_1} & \text{falls } x_1 \neq x_2 \text{ und } ggT(x_2 - x_1,\ n) = 1 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Die Berechnung der x_3- bzw. y_3-Koordinaten bleibt unverändert wie in Def. 4.2 erhalten.

Kapitel 5

Elliptic Curve Method (ECM)

In diesem Kapitel soll die Elliptic Curve Method zur Faktorisierung zusammengesetzter, natürlicher Zahlen vorgestellt werden. Sie basiert auf der $p-1$ - Methode von Pollard, die in Kapitel 3 beschrieben wurde.

Pollards Verfahren beruht darauf, dass die zu den Primteilern p_i von n passenden Zahlen $p_i - 1$ aus kleinen Faktoren bestehen, die in einer Faktorbasis M enthalten sind. Dann ist der $\mathrm{ggT}(c^M - 1, n)$ ein Faktor von n.

In Pollards Methode wird nur die Gruppe $\mathbb{Z}_p^*$ benutzt; die Methode funktioniert, falls die Gruppenordnung $p-1$ kleine Teiler besitzt. In Lenstras ECM hat man dagegen eine riesige Auswahl an Gruppen bzw. Gruppenordnungen, die man durchtesten kann.

Im folgenden Abschnitt wird zunächst der Algorithmus vorgestellt. Kapitel 5.2 beschreibt die Funktionsweise des Algorithmus und zeigt, warum mit ihm nichttriviale Faktoren von n gefunden werden können. In Kapitel 5.3 wird anschließend die Laufzeit der Elliptic Curve Method analysiert.

5.1 Algorithmus

Der Algorithmus orientiert sich im Wesentlichen an dem Algorithmus von Pollard, vgl. S. 8. Zunächst wird eine obere Suchgrenze B festgelegt. Sie hat direkten Einfluss auf die Größe der Primpotenzbasis M und sollte am Anfang nicht zu hoch gesetzt werden, es sei denn, es ist bekannt, dass die zu faktorisierende Zahl aus großen Primfaktoren besteht. Eine Übersicht über mögliche Werte von B ist in Tabelle 7.2 aufgelistet.

Schritt 2 wählt eine zufällige Pseudokurve und einen Punkt, auf dem gerechnet werden soll. Im dritten Schritt wird die Primpotenzbasis M berechnet. Dieser Schritt dient lediglich der Anschauung und der Vergleichsmöglichkeit mit der $p-1$ - Methode und kann bei der

Implementierung mit Schritt 4 vermischt werden. Schritt 4 berechnet nun $[M]\,P$. Falls kein Faktor gefunden wird, wählt Schritt 5 nach einer bestimmten Anzahl von Fehlschlägen eine höhere Suchgrenze B.

Algorithmus 5.1 (Elliptic Curve Method (ECM)). *Eingabe: eine Zahl n, die weder durch 2 noch durch 3 teilbar ist, und die sowohl keine Potenz, als auch nicht prim ist. Ausgabe: ein (nicht unbedingt primer) Faktor von n.*

1. [Suchgrenze festlegen]
   B := 10 000;

2. [Zufällige Pseudokurve E := $E_{a,b}(\mathbb{Z}_n)$ sowie einen Punkt P(x, y) $\in$ E wählen]
   Wähle x, y, a zufällig aus [0, n – 1];
   b := ($y^2 - x^3 - ax$) mod n;
   g := ggT($4a^3 + 27b^2$, n);
   if g = n then goto [2];
   elsif g > 1 then return g; // nichttrivialer Faktor gefunden

3. [Bestimme Primpotenzbasis M]
   M := kgV(1, ..., B); // so dass M = $p_1^{a_1} \cdot p_2^{a_2} \cdots p_h^{a_h}$

4. [Berechne [M]P]
   for i = 1 to h // mit h = π(B), vgl. Def. 2.2
      for j = 1 to a_i
         P := [p_i] P;
         Falls bei der Berechnung eine Invertierung d^{-1} fehlschlägt
            g := ggT(d, n); // nichttrivialer Faktor gefunden
            return g;

5. [Kein Faktor gefunden]
   ggf. Suchgrenze B erhöhen;
   goto [2];

5.2 Funktionsweise

Hier soll beschrieben werden, wann der Algorithmus 5.1 Faktoren von n finden kann. Im Algorithmus gibt es zwei Stellen, an denen die Faktorisierung erfolgreich sein kann – in

Schritt 2 und Schritt 4. Diese sollen nun untersucht werden. Abschnitt 5.2.1 geht auf den zweiten Schritt ein. In Abschnitt 5.2.2 werden einige Definitionen und Sätze eingeführt, die für Schritt 4 des Algorithmus nötig sind. Anschließend wird Schritt 4 erläutert.

5.2.1 Schritt 2

Sei $n \in \mathbb{N}$ eine ungerade, zusammengesetzte Zahl, die faktorisiert werden soll, und p ein noch unbekannter Primfaktor von n mit $p > 3$.

In Schritt 2 soll nun eine Kurve der Form $y^2 = x^3 + ax + b$ und ein Punkt $P = (x, y)$ auf der Kurve berechnet werden, $a, b, x, y \in \mathbb{Z}_n$. Dazu wählen wir zufällig $a, x, y \in \mathbb{Z}_n$ und bestimmen $b := y^2 - x^3 - ax$. Durch Berechnung von $g := \mathrm{ggT}(4a^3 + 27b^2, n)$ wird getestet, ob die Kurve doppelte Nullstellen hat. Es gibt drei mögliche Fälle: (1) Ist $g = n$, d. h. $n \mid 4a^3 + 27b^2$, so müssen a, b, x, y neu bestimmt werden. (2) Ist $1 < g < n$, dann ist in g ein nichttrivialer Faktor von n gefunden worden und wir sind vorläufig fertig[1]. (3) Im letzten Fall $g = 1$ wurde die Kurve und der Punkt erfolgreich bestimmt, und wir können mit der eigentlichen Faktorisierung fortfahren. Sei also im Folgenden $\mathrm{ggT}(4a^3 + 27b^2, n) = 1$. Da $4a^3 + 27b^2$ keine gemeinsamen Teiler mit n hat, gibt es keine mehrfachen Nullstellen modulo p für alle Teiler p von n.

5.2.2 Vorbereitung zu Schritt 4

Definition 5.1. *Sei $m \in \mathbb{Z}$ und seien $x, y \in \mathbb{Q}$ derart, dass ihre Nenner keine gemeinsamen Teiler mit m haben (d. h. sie sind prim zu m). Sei außerdem $d := x - y$ vollständig gekürzt mit $d = \frac{z}{n}$. Definiere*

$$x \equiv y \ (mod\ m) \quad \rightleftharpoons \quad m \mid z$$

Zwei Elemente aus $\mathbb{Q}$ sind also äquivalent, wenn die (vollständig gekürzte) Differenz ihrer Zähler durch m teilbar ist.

Satz 5.2. *Für jedes $x \in \mathbb{Q}$ mit Nenner prim zu m gibt es ein eindeutig bestimmtes $y \in \mathbb{Z}_m$, so dass $x \equiv y \ mod\ m$. Dabei wird y auch als der kleinste nichtnegative Rest bezeichnet.*

Im Folgenden soll nun ein wichtiger Satz aufgeführt und bewiesen werden.

Satz 5.3. *Sei E eine elliptische Kurve mit $a, b \in \mathbb{Z}$ und $(4a^3 + 27b^2,\ n) = 1$, und seien*

[1] die beiden Ergebnisse g und n/g der Faktorisierung sind möglicherweise noch nicht prim und müssen weiter faktorisiert werden.

$P_1 \neq -P_2$ Punkte, deren Nenner[2] prim zu n sind. Dann gilt:

$$P_1 + P_2 \in E \text{ hat Nenner prim zu } n$$
$$\Longleftrightarrow$$
$$\nexists p \,|\, n \text{ so dass für } (P_1 \bmod p) \text{ und } (P_2 \bmod p) \text{ auf } (E \bmod p) \text{ gilt:}$$
$$(P_1 \bmod p) + (P_2 \bmod p) = (\mathcal{O} \bmod p) \in (E \bmod p)$$

Dabei ist E mod p diejenige elliptische Kurve über $\mathbb{F}_p$, die man durch Reduktion modulo p der Koeffizienten von $y^2 = x^3 + ax + b$ erhält.

Beweis. Sei $P_1 = (x_1,\ y_1)$, $P_2 = (x_2,\ y_2)$ und $P_3 := P_1 + P_2 \in E$ und seien alle Nenner prim zu n. Sei außerdem p ein Teiler von n.

Zu zeigen "$\Rightarrow$": P_1 mod $p + P_2$ mod $p \neq \mathcal{O}$ mod p. Wir unterscheiden folgende Fälle:

1. $x_1 \not\equiv x_2 \bmod p$: Nach Definition 4.5 der Addition ist $P_1 \bmod p + P_2 \bmod p \neq \mathcal{O} \bmod p$.

2. $x_1 \equiv x_2 \bmod p$. Dies führt zur Unterscheidung von zwei Teilfällen:

2a. $P_1 = P_2 =: P(x, y)$. Sowohl die Berechnung '$[2]P$' als auch '$[2]P \bmod p$' erfolgen über die Additionsformel mit $m = \frac{3x^2+a}{2y}$ (vgl. Def. 4.5), wobei im zweiten Fall jeder Term modulo p reduziert wird. Es bleibt zu zeigen, dass $2y$ nicht durch p teilbar ist. Das übernimmt der folgende Widerspruchsbeweis: Angenommen $p \,|\, 2y$. Der Nenner von x des Punktes $[2]P(x, y)$ ist *nicht* durch p teilbar. Daher ist der Zähler $3x^2 + a$ der Steigung m ebenfalls (wie der Nenner $2y$) durch p teilbar. Das bedeutet aber, dass x eine Nullstelle (mod p) sowohl des Polynoms $x^3 + ax + b$ als auch von dessen Ableitung ist. Das widerspricht der Annahme, dass es keine mehrfachen Nullstellen modulo p gibt.

2b. $P_1 \neq P_2$. In diesem Fall gelten die zwei Bedingungen $x_1 \equiv x_2$ und $x_1 \neq x_2$. Daher ist $x_2 = x_1 + p^r\ell$ mit $r \geq 1$ und ℓ derart, dass weder Zähler noch Nenner durch p teilbar sind. Die Nenner von $P_1 + P_2$ sind nach Voraussetzung nicht durch p teilbar. Unter Verwendung der Additionsformel mit $m = \frac{y_2-y_1}{x_2-x_1}$, $x_3 = m^2 - x_1 - x_2$ und $y_3 = m(x_1 - x_3) - y_1$ folgt, dass auch y_2 in der Form $y_1 + p^r m$ darstellbar ist. Es gilt aber gleichzeitig auch

$$\begin{aligned}
y_2^2 &= x_2^3 + ax_2 + b \\
&= (x_1 + p^r\ell)^3 + a(x_1 + p^r\ell) + b \\
&= x_1^3 + 3x_1^2p^r\ell + 3x_1p^{2r}\ell^2 + p^{3r}\ell^3 + ax_1 + p^r\ell a + b \\
&= x_1^3 + ax_1 + b + p^r\ell(3x_1^2 + a) + p^{2r}\ell^2(3x_1 + p^r\ell) \\
&\equiv x_1^3 + ax_1 + b + p^r\ell(3x_1^2 + a) \pmod{p^{r+1}} \\
&= y_1^2 + p^r\ell(3x_1^2 + a)
\end{aligned} \tag{5.1}$$

[2]Wenn hier von Nennern gesprochen wird, dann sind damit die Nenner n_1, n_2 der Koordinaten von Punkten $(\frac{z_1}{n_1}, \frac{z_2}{n_2})$ gemeint.

Somit ist neben $x_1 \equiv x_2$ auch $y_1 \equiv y_2 \bmod p$, und daher $P_1 \bmod p = P_2 \bmod p$ und schließlich $(P_1 \bmod p) + (P_2 \bmod p) = [2]P_1 \bmod p$. Dieses Ergebnis ist genau dann $\mathcal{O} \bmod p$, wenn $y_1 \equiv -y_2 \bmod p$, also wenn $y_1 \equiv y_2 \equiv 0 \bmod p$. In diesem Fall wäre dann $y_2^2 - y_1^2 = (y_2 - y_1)(y_2 + y_1)$ durch p^{r+1} teilbar und damit auch der Zähler dieses Terms. Somit folgt aus (5.1), dass $3x_1^2 + a \equiv 0 \bmod p$. Das ist aber nicht möglich, da $x^3 + ax + b$ nach Voraussetzung keine mehrfachen Nullstellen hat (und x_1 deshalb nicht die Nullstelle des Polynoms und dessen Ableitung sein kann).

Damit ist die erste Richtung gezeigt. Nun zum zweiten Teil.
Der rechte Teil der Äquivalenz des zu beweisenden Satzes lässt sich umschreiben zu $\forall p \mid n$: $(P_1 \bmod p) + (P_2 \bmod p) \neq (\mathcal{O} \bmod p)$. Unter dieser Voraussetzung soll gezeigt werden, dass die Koordinaten von $P_1 + P_2$ zu n teilerfremde Nenner haben, die Nenner also nicht durch p teilbar sind, und zwar für *alle* Primfaktoren p von n.
Sei p im Folgenden ein fester Primteiler von n.

1. $x_1 \not\equiv x_2 \bmod p$. Mit $m = \frac{y_2 - y_1}{x_2 - x_1}$ aus Def. 4.5 sieht man, dass keiner der Nenner durch p teilbar ist. Sei also

2. $x_1 \equiv x_2 \bmod p$. Für y_2 kann es nur zwei Möglichkeiten geben: $y_2 \equiv \pm y_1 \bmod p$, und die Voraussetzung $P_1 \bmod p + P_2 \bmod p \neq \mathcal{O} \bmod p$ schließt den Fall $y_1 \equiv y_2 \equiv 0$ aus.

2a. $P_1 = P_2$. Mit der Variante $m = \frac{3x_1^2 + a}{2y_1}$ der Addition (Def. 4.5) und der Bedingung $y_1 \not\equiv 0$ folgt sofort, dass die Nenner in $[2]P_1$ prim zu n sind.

2b. $P_1 \neq P_2$. Wiederum kann x_2 als $x_1 + p^r \ell$ geschrieben werden, wobei ℓ nicht durch p teilbar ist. Unter Verwendung von (5.1) erhält man

$$\frac{y_2^2 - y_1^2}{x_2 - x_1} \equiv 3x_1^2 + a \pmod p$$

Da p nicht $y_1 + y_2 \equiv 2y_1$ teilt, ist im Nenner von

$$\frac{y_2^2 - y_1^2}{(y_2 + y_1)(x_2 - x_1)} = \frac{y_2 - y_1}{x_2 - x_1}$$

kein Teiler p vorhanden, und nach der Formel $m = \frac{y_2 - y_1}{x_2 - x_1}$ für die Addition gibt es kein p im Nenner von $P_1 + P_2$.

■

5.2.3 Schritt 4

In Schritt 4 wird der gewählte Punkt P mit der Primpotenzbasis multipliziert, und zwar schrittweise zuerst mit den kleinen Primzahlen. In jedem Schritt wird modulo n reduziert. Diese Rechung funktioniert (ist aber noch nutzlos), bis der glückliche Fall eintritt, dass in einer der internen Additionen ein $d := x_2 - x_1$ bzw. $d := 2y_1$ (vgl. Def. 4.5) nicht invertierbar ist. Das kann nur eintreten, wenn der $\mathrm{ggT}(d,\ n) \neq 1$ ist.

Nach Satz 5.3 passiert dies, wenn $[k]P$ berechnet werden soll – wobei $[k]P$ ein Teilschritt der Berechnung $[M]P$ ist – und für ein $p \,|\, n$ gilt $k(P \bmod p) = \mathcal{O} \bmod p$, der Punkt also Ordnung m hat, mit m Teiler von k.

Die Verwendung des erweiterten euklidischen Algorithmus zur Berechnung des Inversen von d mod n liefert den $\mathrm{ggT}(d,\ n) =: g$, falls die Inversion fehlschlägt. Für g gilt die Beziehung $1 < g \leq n$, d. h. es kann u. U. auch der triviale Faktor n herauskommen. Das tritt nach Satz 5.3 aber nur dann ein, wenn $[k]P \bmod p = \mathcal{O} \bmod p$ für *alle* Primteiler p von n an genau diesem Zwischenschritt k der Multiplikation von $[M]P$ gilt, wobei k die ersten Faktoren von M enthält. Dieser Fall ist sehr unwahrscheinlich, falls n zwei oder mehr sehr große Primteiler enthält. Daher ist es nahezu sicher, einen nichttrivialen Faktor von n zu erhalten.

Hier sei noch einmal der Vorteil gegenüber Pollards $p-1$-Methode erwähnt: Anstelle der einen Gruppe $\mathbb{Z}_p^* = (\mathbb{Z}/p\mathbb{Z})^*$ werden hier die Gruppen E mod p verwendet. Falls sich die Wahl als ungünstig herausstellt, wird einfach eine neue Kurve E und ein neuer Punkt $P \in E$ gewählt, und das Ganze wiederholt.

5.3 Laufzeitanalyse

Die Laufzeit hängt von der glücklichen (zufälligen) Wahl der Kurven ab und kann deshalb nur heuristisch abgeschätzt werden. Zunächst sollen daher Theoreme von H. Hasse und Lenstra zur Berechnung der Gruppenordnung aufgeführt werden. Die Ordnung $\#E(K)$ einer elliptischen Kurve $E_{a,b}(K)$ ist dabei die Anzahl der Punkte der Kurve einschließlich des Punkts $\mathcal{O}$ im Unendlichen. Anschließend soll die Laufzeit der Elliptic Curve Method bestimmt werden.

5.3.1 Gruppenordnung elliptischer Kurven

Theorem 5.4 (Hasse). *Sei p prim, $k \geq 1$ und $a, b \in \mathbb{F}_{p^k}$. Für die Ordnung $\#E$ der Kurve $E := E_{a,b}(\mathbb{F}_{p^k})$ gilt:*

$$| \#E - (p^k + 1) | \quad \leq \quad 2\sqrt{p^k} \tag{5.2}$$

Für Körper der Form $\mathbb{F}_p$ gibt es aber auch eine direkte Formel. Unter Verwendung des Legendre-Symbols gilt:

$$\#E(\mathbb{F}_p) = p + 1 + \sum_{x \in \mathbb{F}_p} \left(\frac{x^3 + ax + b}{p} \right) \tag{5.3}$$

Das Hasse-Theorem 5.4 mit $k = 1$ lässt dabei die maximale Abweichung der Summe von dem Wert $p + 1$ abschätzen, denn umgeformt und in Intervallschreibweise erhält man für (5.2) mit $k = 1$:

$$\#E(\mathbb{F}_p) \quad \in \quad p + 1 + [-2\sqrt{p},\ +2\sqrt{p}] \tag{5.4}$$

Definition 5.5. *Sei $p > 3$ prim, und sei $S \subseteq \mathbb{N} \cap (p + 1 + [-2\sqrt{p},\ 2\sqrt{p}])$ mit $|S| \geq 3$ eine Menge*[3] *ganzer Zahlen im Hasse-Intervall. Definiere*

$$N_1(S) := \#\{\, (a, b) \in \mathbb{F}_p \times \mathbb{F}_p \mid 4a^3 + 27b^2 \neq 0 \ \wedge\ \#E_{a,b}(\mathbb{F}_p) \in S\}$$

als die Anzahl der Paare (a, b) in $\mathbb{F}_p^2$ mit $4a^3 + 27b^2 \neq 0$, deren Gruppenordnung $\#E_{a,b}(\mathbb{F}_p)$ eine der Zahlen in S ist, und analog

$$N_2(S) := \#\{\, (a, x, y) \in \mathbb{F}_p^3 \mid b := y^2 - x^3 - ax,\ 4a^3 + 27b^2 \neq 0 \text{ und } \#E_{a,b}(\mathbb{F}_p) \in S\}$$

Eine Abschätzung dieser Zahlen liefert das folgende Theorem von Lenstra.

Theorem 5.6 (Lenstra). *Gegeben seien die gleichen Voraussetzungen wie in Def. 5.5. Dann gibt es ein $c > 0$ mit*

$$N_1(S) \ > \ \frac{c \ \cdot \ |S| \ \cdot \ p^{3/2}}{\ln p}$$

$$N_2(S) \ > \ \frac{c \ \cdot \ |S| \ \cdot \ p^{5/2}}{\ln p}$$

Beweis. Ein Beweis findet sich in [6]. ■

5.3.2 Laufzeitbestimmung

Sei n zusammengesetzt, ko-prim zu 6, und keine Potenz. Sei außerdem p der kleinste, und q ein weiterer Primfaktor von n. In Algorithmus 5.1, Schritt [2] werden zunächst a, b, P gewählt. Der Algorithmus ist erfolgreich, falls für irgendein M der Form

$$M = p_k^{\beta} \prod_{i<k} p_i^{a_i}$$

[3] Die Menge S könnte z. B. alle Primzahlen oder alle glatten Zahlen (d. h. mit "kleinen" Primfaktoren) im Hasse-Intervall enthalten.

mit $k \leq \pi(B)$, B eine Suchgrenze aus dem Algorithmus, und $\beta \leq a_k$ die beiden Ereignisse

$$[M]\,P = 0 \quad \text{in } E_{a,b}(\mathbb{F}_p) \tag{5.5}$$

$$[M]\,P \neq 0 \quad \text{in } E_{a,b}(\mathbb{F}_q) \tag{5.6}$$

eintreten. Da die Wahrscheinlichkeit dieser beiden Ereignisse wesentlich von der Wahrscheinlichkeit von (5.5) abhängt, kann (5.6) im Folgenden vernachlässigt werden.

Der Fall (5.5) tritt ein, wenn $\#E_{a,b}(\mathbb{F}_p)$ für das gewählte B, siehe Schritt 1 oder 5 im Algorithmus, B-glatt ist. Nach Theorem 5.6 gilt für die Erfolgswahrscheinlichkeit $P(B)$ unter Verwendung der Notation in Def. 2.5

$$PB > c \cdot \frac{\psi(p+1+2\sqrt{p},\ B) - \psi(p+1-2\sqrt{p},\ B)}{\sqrt{p}\ \ln p}$$

Die Berechnungen in Schritt [4] des Algorithmus benötigen ungefähr B Rechenschritte. Daher soll B so gewählt werden, dass B/PB minimal wird. Das ist der Fall, wenn B möglichst klein, die Erfolgswahrscheinlichkeit PB dagegen groß ist. Die folgende Umformung erleichtert unsere Berechnung:

$$PB \approx c \cdot \frac{\psi\left(\frac{3}{2}p,\ B\right) - \psi\left(\frac{1}{2}p,\ B\right)}{p \ln p}$$

Unter Verwendung der Abschätzungen in Kapitel 2.2 sieht man, dass das Minimum eintritt, falls

$$B = \exp\left[\left(\frac{\sqrt{2}}{2} + o(1)\right)\sqrt{\ln p \ln\ln p}\right]$$

und für dieses B wird die Komplexität von B/PB abgeschätzt durch

$$\exp\left[\left(\sqrt{2} + o(1)\right)\sqrt{\ln p \ln\ln p}\right]$$

Da p nicht bekannt ist, muss für B ein "guter" Wert gefunden werden. Im Algorithmus wird zunächst mit B = 10 000 begonnen, und dieser Wert jeweils nach einer gewissen Anzahl an Misserfolgen (im Programm zu dieser Studienarbeit nach jeweils 500 Fehlschlägen) verdoppelt.

Die Komplexität von ECM beträgt also insgesamt

$$\left(e^{\sqrt{\ln p \ln\ln p}}\right)^{\sqrt{2}+o(1)}$$

falls p der kleinste Primfaktor der zu faktorisierenden Zahl n ist. Dabei geht $o(1)$ gegen 0 für $p \to \infty$.

Der worst-case tritt bei ECM ein, falls n das Produkt zweier ähnlich großer Primzahlen ist. Dann beträgt die Komplexität, diesmal in Abhängigkeit von n ausgedrückt, $\left(e^{\sqrt{\ln n \ln \ln n}}\right)^{1+o(1)}$. Dies ist die gleiche (heuristische) Komplexität wie beim Quadratischen Sieb (QS) oder beim Zahlkörpersieb (Number Field Sieve, NFS). Allerdings sind die Berechnungen bei ECM etwas aufwändiger, so dass bei worst-case-Zahlen eher die Verfahren QS oder NFS zum Einsatz kommen.

Ab einer bestimmten Größe[4] von n kommen dagegen QS und NFS nicht mehr in Frage. Dann bleibt der Versuch mit ECM die einzige Möglichkeit, die Zahl zu faktorisieren.

[4]abhängig von der Implementierung und der eingesetzten Hardware.

Kapitel 6

Implementierung

In diesem Kapitel werden die wichtigsten Funktionen und das Zusammenspiel der einzelnen Programmdateien erläutert. Abbildung 6.1 zeigt die Ablauffolge der einzelnen Funktionen in den Dateien *ecm.cc* und *ecmethod.cc* als Schaubild, um einen Überblick zu bekommen. Als Programmiersprache wurde C++ verwendet. Die Implementierung benutzt das frei verwendbare Programmpaket GMP (GNU Multiple Precision Arithmetic Library), das auf http://www.swox.com erhältlich ist. Es ermöglicht das Rechnen mit großen Zahlen. In den folgenden Abschnitten werden nun einige Funktionen erklärt.

6.1 Die Datei ecm.cc

Die Datei *ecm.cc* dient als Startpunkt der ECM-Methode. Sie enthält die main-Funktion, sowie Hilfsfunktionen zur Behandlung von Fehlern bzw. von falschen Benutzereingaben. Die main-Funktion musste in einer eigenen Datei untergebracht werden, da der ECM-Algorithmus auch aus *ecpp.cc* in der Studienarbeit 'Primzahltest auf Elliptischen Kurven' aufgerufen wird, und zwei main-Funktionen innerhalb eines Programmes nicht erlaubt sind. Daher ist der eigentliche ECM-Algorithmus in der Datei *ecmethod.cc* zu finden.

Die main-Methode wird vom Benutzer von der Konsole aus gestartet und erhält als Parameter die zu faktorisierende Zahl n. Zusätzlich kann als optionaler Parameter die Suchgrenze angegeben werden. Die Parameter -v, -f bzw. -r dienen einer erweiterten Ausgabe (verbose), der Ausgabe ohne Konsolenanweisungen (file-optimized) bzw. dazu, dass immer die gleichen Zufallszahlen generiert werden (no random). Die Reihenfolge der Parameter ist unerheblich. Der Programmaufruf hat also die Syntax

./ecm [-b <suchgrenze>] [-v] [-f] [-r] <n>

Hat der Benutzer keine Parameter eingegeben oder enthalten die eingegebenen Zahlen unzulässige Zeichen, so wird die Funktion *usage* aufgerufen, die dem Benutzer die Syntax erklärt. Anschließend wird das Programm beendet.

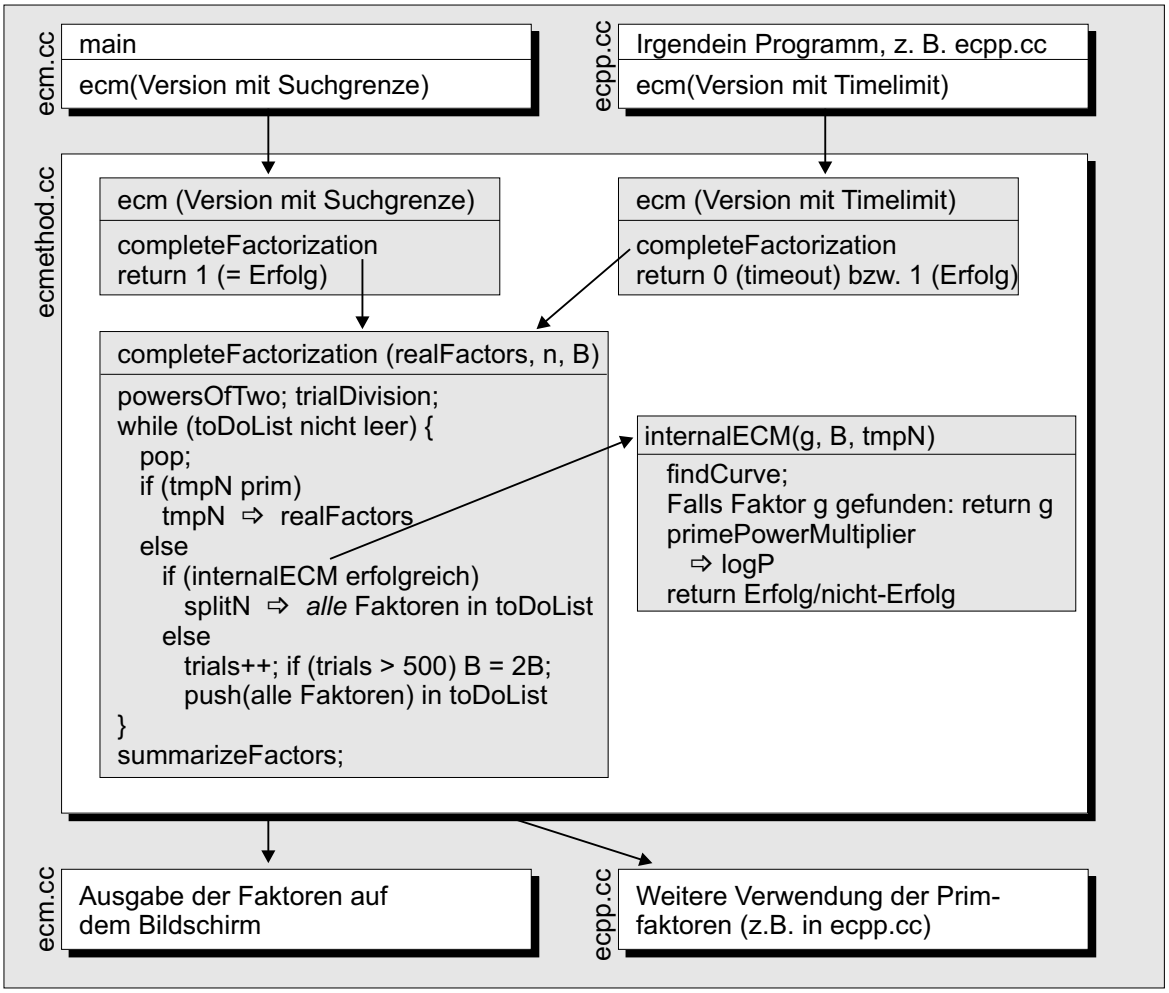


Abbildung 6.1: Ablauf der Funktionsaufrufe

Nach dem Auswerten der Parameter wird ein Objekt der Time-Klasse, die in *tools.h* definiert ist, erzeugt, das beim Aufruf des Konstruktors die Startzeit des Programms speichert. Anschließend wird die Funktion *ecm* aufgerufen, die in Abschnitt 6.2.5 beschrieben ist. Nach der Berechnung aller Faktoren der eingegebenen Zahl n in dieser Funktion werden die Ergebnisse sowie die Laufzeit auf dem Bildschirm ausgegeben, und das Programm endet.

6.2 Die Datei ecmethod.cc

Die Datei *ecmethod.cc* enthält die wesentlichen Teile der Elliptic Curve Method. Funktionen, die auf den elliptischen Kurven basieren, sind in der Datei *EllipticCurve.cc* vorhanden, da sie auch beim Elliptic Curve Primality Proving (ECPP) verwendet werden.

Weitere allgemeinere Funktionen sind in der Datei `tools.cc` zu finden. Hier sollen nun die wichtigsten Funktionen von `ecmethod.cc` beschrieben werden.

6.2.1 Die Funktion findCurve

Diese Funktion versucht, eine Kurve $E_{a,b}(\mathbb{Z}_n)$ mit zufälligen Parametern a, b zu finden. Die Kurve muss nicht-singulär sein, d. h. es muss $4a^3 + 27b^2 \not\equiv 0 \pmod{n}$ gelten. Auf dieser Kurve wird zusätzlich ein Punkt P(x,y) bestimmt und ebenfalls zurückgegeben. Die Signatur der Funktion lautet

```
bool findCurve(mpz_t g, mpz_t a, mpz_t b, point &p, mpz_t n)
```

Die Parameter haben abhängig vom Rückgabewert der Funktion (true bzw. false) die folgende Bedeutung[1]:

g	(out):	undefiniert (true) bzw. ein nichttrivialer Faktor von n (false)
a	(out):	einer der Kurvenparameter (true) bzw. undefiniert (false)
b	(out):	einer der Kurvenparameter (true) bzw. undefiniert (false)
p	(out):	ein gefundener Punkt auf der Kurve (true) or undefiniert (false)
n	(in):	die Zahl, die faktorisiert werden soll
Returnwert:		true, falls eine Kurve und ein Punkt gefunden wurden, bzw. false, falls ein nichttrivialer Faktor von n entdeckt wurde.

Ablauf der Funktion: Zunächst werden zufällige Werte für den Punkt (x, y) und den Kurvenparameter a im Intervall $[0, n-1]$ ausgewählt und dann $b := y^2 - x^3 - ax \pmod{n}$ berechnet. Danach wird $g := \mathrm{ggT}(4a^3 + 27b^2,\ n)$ geprüft. Es gibt drei mögliche Fälle:

- $g = 1$ $\rightarrow$ Kurvenparameter erfolgreich gewählt.
- $1 < g < n$ $\rightarrow$ g ist ein nichttrivialer Faktor von n $\rightarrow$ Erfolg!
- $g = n$ $\rightarrow$ Zufällige Wahl der Kurvenparameter und des Punktes wiederholen.

In den ersten beiden Fällen wird die Funktion mit true bzw. false verlassen und bei true beginnt die eigentliche Faktorisierung mit der Funktion `primePowerMultipliers`, die in Abschnitt 6.2.3 beschrieben wird.

[1] Ein in-Parameter bezeichne im Folgenden eine Variable, die in der betreffenden Funktion *nicht* verändert wird. Ein out-Parameter wird von der Funktion mit Werten gefüllt; sein Inhalt wird im Allgemeinen vorher gelöscht. Die Variable muss auf jeden Fall initialisiert sein. Schließlich weist ein in/out-Parameter darauf hin, dass Werte von außerhalb der Funktion übergeben werden, und dann von der Funktion geändert oder um weitere Werte ergänzt werden.

6.2.2 Die Funktion logP

Zur Berechnung des Logarithmus $\log_p arg$ einer Zahl arg zur Basis p wird die Funktion

```
unsigned long logP(mpz_t arg, mpz_t p)
```

verwendet. Aus Effizienzgründen wurde auf die exakte Berechnung des Logarithmus mittels ld(arg) / ld(p) unter Verwendung des mpfr-Pakets[2] verzichtet und die Berechnung folgendermaßen durchgeführt:

```
result := 0;
tmp := p;
while (tmp ≤ arg)
    tmp *= p;
    result++;
```

Dies erwies sich als erheblich schneller als die Berechnung mit der mpfr-Bibliothek. Bei Tests dieser Funktion waren Geschwindigkeitsunterschiede im Bereich 1 : 90 zu beobachten. Die gesamte Elliptic Curve Method wurde bei einem Testlauf mit einer ursprünglichen Laufzeit von 75 min. um 14 % beschleunigt.

6.2.3 Die Funktion primePowerMultipliers

Diese Funktion führt die Multiplikation des zuvor in ***findCurve*** gewählten Punktes P mit der Primpotenzbasis M durch, d. h. es wird $[M]\,P$ berechnet. Dabei ist $M = \prod_i (p_i^{a_i})$ für alle p_i und $a_i \geq 1$ mit $p_i^{a_i} \leq B$. Die Berechnung wurde in der Form implementiert, dass zunächst die kleinen Primfaktoren multipliziert werden und dann die großen. Der Algorithmus berechnet also $[M]\,P = [\cdots p_3 \cdot p_2 \cdots p_2 \cdot p_1 \cdots p_1] \cdot P$ in der Reihenfolge von *rechts* nach *links*! Dabei wird bei jeder Multiplikation geprüft, ob eine der Inversionen im Ring $\mathbb{Z}_n$ fehlgeschlagen ist, und in diesem Fall ein Faktor von n herausgegeben.

```
bool primePowerMultipliers(mpz_t g, point &p, mpz_t B, mpz_t a, mpz_t n)
```

Die Parameter haben abhängig vom Rückgabewert der Funktion (true bzw. false) die folgende Bedeutung:

g	(out):	der gefundene Faktor von n (true), oder undefiniert (false)
p	(in):	Punkt der elliptischen Pseudokurve
B	(in):	Suchgrenze
a	(in):	Parameter der elliptischen Pseudokurve $E_{a,b}(\mathbb{Z}_n)$
n	(in):	die zu faktorisierende Zahl
Returnwert:		true, falls ein Faktor gefunden wurde, sonst false.

[2] Multiple Precision Floating-Point Reliable Library, Teil der gmp-Bibliothek

6.2.4 Die Funktion completeFactorization

Die Funktion

```
int completeFactorization(vector<powerStruct*>  &realFactors, mpz_t n, mpz_t B)
```

kümmert sich um die komplette Faktorisierung der eingegebenen Zahl n: Nach dem Entdecken eines Faktors f teilt diese Funktion die Zahl n durch den gefundenen Faktor, prüft, ob n/f und f schon prim sind, und startet ggf. die Faktorisierung dieser neuen Zahlen durch Aufruf von internalECM, s. Abb. 6.1.

6.2.5 Die Funktionen ecm

Die Elliptic Curve Method kann auf zwei verschiedene Arten aufgerufen werden. In beiden Fällen dient dazu die überladene Funktion *ecm* mit den zwei möglichen Parameterlisten

```
int ecm(vector<powerStruct*> &realFactors, mpz_t n, int limit)
```

bzw.

```
int ecm(vector<powerStruct*> &realFactors, mpz_t n, mpz_t B)
```

Die erste Variable ist dabei ein out-Parameter, die beiden anderen werden nicht verändert. Beide Funktionen bekommen als ersten Parameter einen Vektor, der aus Zeigern auf powerStructs besteht. Diese sind in *tools.h* folgendermaßen definiert:

```
typedef struct powerStruct {
    char *base;
    unsigned long exp;
}
```

Der Vektor enthält nach Beendigung der Funktion *ecm* die komplette Faktorisierung von n als Folge von Paaren $base^{exp}$, wobei *base* mit einer hohen Wahrscheinlichkeit[3] prim ist.

Der zweite Parameter enthält in beiden Versionen die zu faktorisierende Zahl. Bei der ersten Funktion wird zusätzlich als dritter Parameter ein Zeitlimit in Sekunden verlangt. Das Programm bricht nach dieser Zeit die Faktorisierung ab, und liefert als Rückgabewert die 0. Diese Variante wird vom Programm ecpp aufgerufen (vgl. Abschnitt 6.1). Bei der zweiten Variante kann als alternativer dritter Parameter ein Startwert für die Suchgrenze angegeben werden. Diese Variante wird aufgerufen, wenn vom Benutzer das Programm *./ecm* gestartet wird.

In beiden Fällen wird bei erfolgreicher Faktorisierung der Returnwert 1 zurückgegeben.

[3] unter Verwendung des Miller-Rabin-Tests

6.3 Die Datei EllipticCurve.cc

In der Datei *EllipticCurve.cc* sind Funktionen enthalten, die die Berechnungen mit Punkten auf elliptischen Kurven ermöglichen. Die fünf Funktionen gehen direkt auf die in Kapitel 4.2 und 4.3 beschriebene Arithmetik mit elliptischen Kurven zurück und sollen daher nur kurz angerissen werden.

6.3.1 Additive Funktionen

Die Funktion

```
void negate(point &p, point &q, mpz_t n)
```

gibt für einen gegebenen Punkt (x, y, z) den neuen Punkt (x, –y, z) zurück, wobei das additiv Inverse modulo n berechnet wird.
Um eine schnelle Multiplikation zu ermöglichen, ist die Funktion

```
bool doublePoint(point &p, mpz_t d, point &q, mpz_t a, mpz_t n)
```

nötig. Sie berechnet für einen Punkt Q das Doppelte $P := [2]Q = Q+Q$ unter Verwendung einer leicht modifizierten Kopie der Funktion

```
bool add(point &p, mpz_t d, point &p1, point &p2, mpz_t a, mpz_t n)
```

Auf dieser Addition wird außerdem noch eine Subtraktion

```
bool sub(point &p, mpz_t d, point &p1, point &p2, mpz_t a, mpz_t n)
```

aufgebaut, die für zwei Punkte P_1 und P_2 unter Verwendung der Addition und der Negation den Wert $P_1 + (-P_2)$ berechnet.

Für die vier Funktionen gilt, falls zutreffend, die folgende Parameterliste. Die einzelnen Parameter haben Rückgabewerte, die vom Returnwert abhängig sein können.

p	(out):	der zu berechnende Punkt (true) bzw. undefiniert (false)
d	(out):	undefiniert (true) bzw. ein gefundener Faktor von n (false)
q, p_1, p_2	(in):	Punkte, die addiert, subtrahiert oder verdoppelt werden sollen
a	(in):	ein Kurvenparameter von $E_{a,b}(\mathbb{Z}_n)$
n	(in):	die zu faktorisierende Zahl und gleichzeitig ein Kurvenparameter von $E_{a,b}(\mathbb{Z}_n)$
Returnwert:		true, falls der Punkt berechnet werden konnte, bzw. false, falls eine Inversion fehlschlug und somit ein nichttrivialer Faktor d von n entdeckt wurde.

6.3.2 Multiplikation

Die Multiplikation eines Skalars $k \in \mathbb{Z}^+$ mit einem Punkt P wurde als k-fache Addition $[k]\,P := P + \ldots + P$ dieses Punktes definiert. Die Funktion

```
bool mul(point &q, mpz_t d, mpz_t k, point &p, mpz_t a, mpz_t n)
```

berechnet diese Multiplikation auf eine schnellere Art. Der Pseudocode dazu ist in Algorithmus 6.1 abgedruckt.

q	(out):	der zu berechnende Punkt (true) bzw. undefiniert (false)
d	(out):	undefiniert (true) bzw. ein gefundener Faktor von n (false)
p	(in):	Punkt
a	(in):	ein Kurvenparameter von $E_{a,b}(\mathbb{Z}_n)$
n	(in):	die zu faktorisierende Zahl und gleichzeitig ein Kurvenparameter von $E_{a,b}(\mathbb{Z}_n)$
Returnwert:		true, falls die Multiplikation berechnet werden konnte, bzw. false, falls eine Inversion fehlschlug und ein Faktor entdeckt wurde.

Algorithmus 6.1 (Multiplikation eines Punktes). *Eingabe: eine elliptische Pseudokurve $E := E_{a,b}(\mathbb{Z}_n)$, ein (nichtnegativer) Faktor $k \in \mathbb{N}_0$ und ein Punkt $P \in E$. Ausgabe: $Q := [k]P$.*

1. [Initialisierung]
   if k = 0 return $\mathcal{O}$
   m := 3k;
   b := #Bits von m; // derart, dass m = $m_{b-1}...m_0$
   Q := P

2. [Bits von k und 3k vergleichen]
   for j = b − 2 downto 1
   Q := [2]Q // per double(Q)
   if m_j = 1 and k_j = 0 then Q := Q + P; // per add(..)
   elsif m_j = 0 and k_j = 1 then Q := Q − P; // per sub(..)
   return Q;

6.4 Weitere Hilfsprozeduren

In den Dateien `tools/tools.cc` und `PrimeIO/PrimeIO.cc` finden sich einige weitere Funktionen, die in der Elliptic Curve Method verwendet werden. Die erste Datei liefert die folgenden Funktionen:

- *void deleteLastLine()* enthält Konsolenanweisungen zum Löschen der letzten Zeile.
- *void setColor(int fg)* und *void setColor(int fg, int bg)* erlauben die Änderung der Vorder- bzw. der Vorder- und Hintergrundfarbe in der Konsole.
- *void testN(powerStruct &myPower, mpz_t n)* prüft, ob die gegebene Zahl n eine Potenz der Form $\mathsf{basis}^{\mathsf{exp}}$ ist. Der Wert von n enthält anschließend die Basis der Zahl, die zusätzlich auch in «myPower» gespeichert wird. Dort wird auch der passende Exponent zurückgegeben.
- *void trialDivision(vector<powerStruct*> &realFactors, mpz_t tmpN, int trialBound)* führt eine Probedivision durch, d. h. testet, ob «tmpN» durch Primzahlen im offenen Intervall (2, «trialBound») teilbar ist. Die gefundenen Teiler werden zusammen mit den passenden Exponenten in «realFactors» angehängt und zusammen mit dem evtl. verkleinerten «tmpN» zurückgegeben.
- *void powersOfTwo (vector<powerStruct*> &realFactors, mpz_t newN, mpz_t n)* spaltet die 2er-Potenzen von «n» ab.
- *void splitN(mpz_t tmpN,*
  vector<powerStruct> &toDoList,*
  vector<char> &factors,*
  unsigned long oldExponent)
  teilt «tmpN» durch die in «factors» enthaltenen Teiler p_i, sofern die Division aufgeht. Sei a_i die entsprechende Potenz, mit der p_i in «tmpN» auftritt. Dann werden alle passenden Paare (p_i, $a_i *$ «oldExponent») sowie das verkleinerte «tmpN» mit «oldExponent» als Exponent in die «toDoList» geschrieben.
- *unsigned long pop(mpz_t tmpN,*
  powerStruct &myPower,
  vector<powerStruct> &toDoList)*
  entfernt ein Element aus der «toDoList» und gibt es in «myPower» zurück. Außerdem wird «tmpN» auf die Basis von «myPower» gesetzt. Der Returnwert der Funktion ist der Exponent dieser Struktur.
- *powerStruct push(vector<powerStruct*> &toDoList, mpz_t base, unsigned long exp)* fügt das Paar («base», «exp») in die «toDoList» ein.
- *void summarizeFactors(vector<powerStruct*> &realFactors)* sortiert die Faktoren aufsteigend und fasst Zahlen a^x, a^y mit der gleichen Basis zu a^{x+y} zusammen.
- *Time::Time()* Konstruktor für eine Klasse zur Laufzeitmessung der Programme. Der Konstruktor speichert die bei seinem Aufruf aktuelle Zeit.

- *void Time::setNewTime()* überschreibt die in der Klasse gespeicherte Uhrzeit.
- *void Time::printTimePassed()* gibt die Zeit auf dem Bildschirm aus, die seit dem letzten Aufruf des Konstruktors bzw. von `setNewTime` vergangen ist.

In der Datei `PrimeIO/PrimeIO.cc` befinden sich Funktionen, die Primzahlen aus einer Datei einlesen, bzw. falls nicht genügend Primzahlen vorhanden sind, weitere Primzahlen erzeugen und an die Datei anhängen.

- *int PrimeIO::eof()* prüft, ob das Dateiende erreicht ist.
- *unsigned long long int PrimeIO::readFile(mpz_t number)* liest eine Primzahl ein und setzt den internen Zeiger auf die nächste Zahl. Der Rückgabewert enthält die Byte-Position, an der die eingelesene Zahl in der Datei beginnt.
- *void PrimeIO::writeFile(mpz_t number)* schreibt die gegebene Zahl an das Dateiende (interne Funktion, wird von `establishPrimes` aufgerufen).
- *int PrimeIO::jumpPosition(unsigned long long int line)* springt an die Byteposition in der Datei. Sollte nur mit Werten aufgerufen werden, die von `readFile` zurückgegeben wurde – es findet keine Prüfung statt.
- *void PrimeIO::establishPrimes(mpz_t start, mpz_t bound)* erzeugt neue Primzahlen im Intervall [start, bound].
- *PrimeIO::PrimeIO(mpz_t maxPrime)* Konstruktor. Öffnet die Primzahldatei und prüft, ob «maxPrime» Primzahlen vorhanden sind; andernfalls werden Primzahlen bis zu dieser Grenze erzeugt (per Aufruf von `establishPrimes`).

Kapitel 7

Beispiele

In diesem Kapitel sollen einige Testläufe aufgeführt werden. Die meisten Tests liefen auf einem Rechner mit zwei Intel^{R} Xeon^{TM}-Prozessoren mit jeweils 3,06 GHz. Da gleichzeitig auch andere Benutzer auf dem Server eingeloggt waren, unterlagen die Ergebnisse gewissen Schwankungen. So waren die Prozesse nachts erheblich schneller fertig als zu den Hauptarbeitszeiten tagsüber. Alle Programmabläufe auf diesem Rechner wurden mit niedrigster Priorität durchgeführt, um andere Benutzer nicht zu stark zu blockieren.

Einige der Tests wurden auf einem AMD Athlon^{TM} XP 2200+ mit 1817 MHz durchgeführt. Die Ergebnisse sind entsprechend gekennzeichnet. Da auf diesem Rechner weniger Traffic vorhanden war, liefen die Tests trotz der geringeren Geschwindigkeit des Rechners erheblich schneller.

Mehrere Vergleichstests ergaben Laufzeitdauern des Intel- bzw. des AMD-Rechners im Verhältnis von 4:1 bis 6:1. Im Schnitt hatte der Athlon ein Ergebnis also in einem Fünftel der Zeit, wobei bei diesen Vergleichen immer dieselben Kurvenparameter und Zahlen zu Grunde gelegt wurden. Es ging aber auch schneller: in Tab. 7.4 war nur das 1,7-fache nötig.

	Intel-Prozessor		AMD-Prozessor	
B	10 Kurven	500 Kurven	10 Kurven	500 Kurven
10 000	22 sek	18,3 min	4 sek	3,3 min
20 000	43 sek	35,8 min	8 sek	6,7 min
40 000	81 sek	67,5 min	17 sek	14,2 min
100 000	3:25 min	2,8 h	0:42 min	0,6 h
200 000	8:27 min	7,0 h	1:25 min	1,2 h
400 000	13:56 min	11,6 h	2:49 min	2,3 h
1 000 000	0:44:12 h	1,5 d	7:21 min	6,1 h
2 000 000	1:49:40 h	3,8 d	14:21 min	12,0 h
4 000 000	2:57:07 h	6,1 d	28:20 min	23,6 h

Tabelle 7.1: Laufzeiten abhängig vom Startwert B auf dem Intel- bzw. AMD-Prozessor

7.1 Laufzeiten abhängig von der Suchgrenze

Dieser Abschnitt stellt die Laufzeiten der Elliptic Curve Method der Höhe der Suchgrenze B gegenüber. Die Laufzeiten wurden anhand von 10 Kurven gemessen und zum besseren Vergleich mit späteren Tests auf 500 Kurven hochgerechnet. Tabelle 7.1 zeigt die Laufzeiten auf den beiden Testrechnern. Dabei wird deutlich, dass eine Verdopplung des Startwerts auch eine Verdopplung der Laufzeit bedeutet. In der Zeile für $B = 200\,000$ sieht man, welchen Schwankungen die Laufzeiten unterworfen sind, wenn der Rechner stark ausgelastet ist. Ein späterer Test für denselben Wert von B dauerte statt 8:27 nur 6:52 Minuten.

Wenn bekannt ist, in welcher Größenordnung die einzelnen Faktoren einer gegebenen Zahl n liegen, kann die Suchgrenze B manuell angepasst werden. Einige Richtwerte können Tabelle 7.2 entnommen werden.

#Ziffern zweitgrößter Faktor	B
1 .. 20	10 000
21 .. 23	50 000
24 .. 26	100 000
27 .. 29	500 000
30 .. 33	1 000 000
$>$ 33	5 000 000

Tabelle 7.2: Mögliche Werte für die Suchgrenze B

7.2 Tests mit RSA-Zahlen

Dieser Test zeigt, wie lange die Faktorisierung von RSA-Zahlen[1] dauert. Die Ergebnisse sind in Tabelle 7.3 dargestellt. Dabei wurden die Zahlen durch Multiplikation zweier Primzahlen konstruiert. Die erste Spalte zeigt die Größe der einzelnen Faktoren. Die drei anderen großen Spalten enthalten die Ergebnisse von Berechnungen, bei denen mit den Startwerten 10 000, 100 000 bzw. 1 Mio. begonnen wurde. Angegeben ist für jeden Durchlauf die Dauer, die Anzahl der Kurven, sowie der Wert von B, bei dem die Faktorisierung erfolgreich war. B wurde dabei jeweils nach 500 getesteten Kurven verdoppelt. Die Nachkommastelle bei der Anzahl an Kurven gibt an, welcher Anteil an Schritten innerhalb der letzten Kurve zur Berechnung des Faktors verwendet wurde. Die mit * gekennzeichneten Zeiten wurden auf dem Athlon ermittelt. Wichtig ist hier vor allem die Anzahl der benötigten Kurven. Um die Laufzeit auf dem Intel zu erhalten, kann näherungsweise Tabelle 7.1 herangezogen werden.

[1] Als RSA-Zahlen werden hier Zahlen bezeichnet, die aus genau zwei Primfaktoren ungefähr derselben Größe bestehen. Eine RSA-Zahl mit 40 Stellen besteht also aus zwei Faktoren mit ungefähr 20 Stellen.

#Ziffern (Faktor)	B = 10 000			B = 100 000			B = 1 000 000		
	Zeit	#K	B	Zeit	#K	B	Zeit	#K	B
20	12,8 min	718,3	20 000	2,2 min	9,5	–	10,2 min	9,0	–
	2,1 min	113,8	–	23,7 min	113,1	–	7,4 min	5,9	–
	3,2 min	202,9	–	0,3 min	0,8	–	0,1 min	0,1	–
25	5,4 h	2645,5	320 000	2,3 h	624,4	200 000	6,4 h	246,5	–
	3,1 h	2237,3	160 000	2,7 h	839,8	200 000	8,3 h	338,9	–
	2,4 h	2058,4	160 000	0,8 h	377,3	–	2,7 h	128,2	–
30	17,9 h *	4044,4	2,6 Mio	23,1 h	2437,9	1,6 Mio	0,3 h	12,8	–
	10,2 h *	3689,2	1,3 Mio	1,5 h *	889	200 000	0,5 h *	41,4	–
	5,1 h *	3233,9	640 000	0,6 h	504	200 000	31,4 h *	1306,6	4 Mio

Tabelle 7.3: Laufzeiten einiger RSA-Zahlen (40 bis 60 Stellen)

Nach der erfolgreichen und schnellen Faktorisierung der RSA-Zahlen bis 60 Dezimalstellen wurden auch zwei Tests mit einer 70-stelligen RSA-Zahl gestartet. Beginnend mit $B = 10\,000$ dauerte der Versuch auf dem Intel-Rechner insgesamt 13,3 Tage, wobei B schließlich auf 20,48 Mio. erhöht wurde. Die gleiche Zahl wurde auch auf dem AMD-Prozessor getestet. Mit B zwischen 100 000 und 51,2 Mio. dauerte die Suche nach dem 35-stelligen Primfaktor hier 18 Tage. Tabelle 7.4 zeigt die Laufzeiten auf den Testrechnern. Die Berechnung rechts hätte auch auf dem Xeon-Prozessor gestartet werden können. Eine Hochrechnung ergibt für dieselben Startwerte und Zufallszahlen dann aber eine Laufzeit von rund 30 Tagen (Faktor 1,7).

	Intel Xeon			AMD Athlon		
	Zeit	#K	B	Zeit	#K	B
	0:07:16	500	10 000	00:40:00	500	100 000
	0:13:41	500	20 000	01:23:15	500	200 000
	0:28:45	500	40 000	02:36:32	500	400 000
	0:55:41	500	80 000	05:08:59	500	800 000
	1:24:46	500	160 000	10:28:00	500	1 600 000
	3:17:25	500	320 000	20:28:59	500	3 200 000
	6:26:34	500	640 000	40:14:27	500	6 400 000
	14:32:53	500	1 280 000	82:39:15	500	12 800 000
	28:38:38	500	2 560 000	163:06:51	500	25 600 000
	52:49:49	500	5 120 000	106:35:37	162	51 200 000
	110:36:48	500	10 240 000			
	99:17:17	215	20 480 000			
Gesamt	318:49:33	5715	20 480 000	433:21:55	4662	51 200 000

Tabelle 7.4: 70-stellige RSA-Zahl auf zwei Testrechnern, Zeitangaben: [hh:mm:ss]

7.3 Faktorisierung größerer Zahlen

In diesem Abschnitt werden Beispiele vorgestellt, in denen Zahlen mit deutlich mehr als zwei Faktoren zerlegt wurden. Alle Rechnungen in diesem Abschnitt wurden auf dem AMD-Prozessor durchgeführt. Tabelle 7.5 zeigt die Ergebnisse von vier Zahlen, die jeweils aus sechs 15- bzw. 20-stelligen Faktoren bestehen und eine Länge von knapp 90 bzw. 120 Ziffern haben.

#Ziffern	1. Faktor		2. Faktor		3. Faktor		4. Faktor		5. Faktor		Gesamt	
(Faktoren)	Zeit	#K	Zeit	#K	Zeit	#K	Zeit	#K	Zeit	#K	Zeit	#K
15	0:10	14	0:21	35	0:06	13	0:27	80	0:04	16	1:08	158
	0:03	5	0:09	17	0:01	3	0:10	29	0:00	1	0:23	55
20	2:27	185	0:17	27	0:04	7	0:48	114	5:44	801	9:20	1134
	11:19	626	3:13	138	0:42	40	6:29	462	3:51	375	25:34	1641

Tabelle 7.5: Zahlen aus sechs Primfaktoren à 15 bzw. 20 Ziffern, Zeitangaben: [mm:ss]

Die anderen beiden Beispiele zeigen die Faktorisierung der beiden 292-stelligen Zahlen

1 124 106 614 617 448 278 637 139 583 097 696 743 416 442 821 211 147 906 598 975 821 283 554 864 175 301 703 305 489 201 505 212 266 927 562 735 470 168 894 824 536 045 108 563 179 833 971 015 700 167 334 240 545 054 580 094 355 375 874 283 524 490 292 725 179 871 184 152 427 950 630 574 684 662 781 485 142 824 812 559 024 146 847 961 473 103 742 335 544 602 425 632 977 708 952 713 789 = 83 * 443 * 4861 * 29009 * 563743 * 8997503 * 67884941 * 350213711 * 1406489069 * 22377406319 * 804286143433 * 5952007138057 * 54897851874223 * 479539991138519 * 1714444953078313 * 81442784742871309 * 833987772076357511 * 3020049971228473159 * 36007256833005409937 * 580577108934993439007 * 1083767671323850111751 * 7241398060865677079859 7 * 7855949842467333396519 37

und

2 946 042 335 064 874 581 846 180 740 813 148 483 810 034 929 145 663 303 038 841 217 000 922 752 046 771 352 055 669 274 640 120 813 195 275 477 085 815 087 905 188 131 067 435 447 518 126 664 793 907 519 937 503 203 880 371 809 317 062 862 239 180 102 683 865 428 097 865 793 261 909 120 043 671 813 133 115 743 064 352 768 709 708 286 211 706 194 428 780 495 962 685 688 920 202 404 019 = 53 * 701 * 3571 * 37253 * 447079 * 2397601 * 83088437 * 247936181 * 7301765593 * 10305483971 * 495597603461 * 2385182674951 * 97566747484499 * 628996350847463 * 6426283207258529 * 73856552678354921 * 253656708200460049 * 3718624529050085467 * 30134875224954411463 * 91590207321927620717 3 * 8354996217465587248781 * 65100999519622945791109 * 7357415943966441270210 71

Beide Zahlen bestehen aus jeweils 23 Primfaktoren der Längen 2, 3, ..., 24. Die ersten vier Faktoren werden durch eine Probedivision (bis 100 000) ermittelt, die restlichen Faktoren sind in den beiden Tabellen 7.6 und 7.7 dargestellt. Startwert für B war der Standardwert 10 000. Die Laufzeiten sind im Format Min:Sek angegeben.

Schließlich wurden noch größere Zahlen faktorisiert. Als Beispiel soll eine 471-stellige Zahl genannt werden, die aus einem 400-stelligen und fünf 15-stelligen Primfaktoren bestand, und in weniger als 25 Minuten zerlegt wurde. Eine Zahl dieser Größe ist weder mit dem Quadratischen Sieb noch dem Zahlkörpersieb in angemessener Zeit faktorisierbar.

Faktor	#Ziffern	Dauer	#Kurven	B erhöht auf
563 743	6	0:00	0,0	–
8 997 503	7	0:00	0,0	–
67 884 941	8	0:03	1,1	–
1 406 489 069	10	0:00	0,0	–
22 377 406 319	11	0:02	0,9	–
350 213 711	9	0:00	0,1	–
5 952 007 138 057	13	0:11	6,5	–
804 286 143 433	12	1:11	41,7	–
54 897 851 874 223	14	0:15	9,2	–
479 539 991 138 519	15	0:48	33,6	–
1 714 444 953 078 313	16	0:15	10,8	–
3 020 049 971 228 473 159	19	3:30	173,6	–
833 987 772 076 357 511	18	3:22	169,5	–
580 577 108 934 993 439 007	21	5:01	305,6	–
81 442 784 742 871 309	17	4:30	353,8	–
36 007 256 833 005 409 937	20	4:45	475,9	–
785 594 984 246 733 339 651 937	24	27:30	1486,4	40 000
1 083 767 671 323 850 111 751	22	1:04	51,2	–
72 413 980 608 656 770 798 597	23	(letzter Faktor)		
Summe		52:27	3130	40 000

Tabelle 7.6: 291-stellige Zahl aus Faktoren der Längen 2, ..., 24 (Test 1), Zeit = [mm:ss]

Faktor	#Ziffern	Dauer	#Kurven	B erhöht auf
83 088 437	8	0:01	0,2	–
447 079	6	0:00	0,2	–
247 936 181	9	0:00	0,1	–
2 397 601	7	0:00	0,0	–
495 597 603 461	12	0:08	4,3	–
7 301 765 593	10	0:05	2,6	–
97 566 747 484 499	14	0:01	0,7	–
2 385 182 674 951	13	0:13	6,4	–
10 305 483 971	11	0:36	17,8	–
628 996 350 847 463	15	2:15	93,3	–
30 134 875 224 954 411 463	20	1:53	82,6	–
73 856 552 678 354 921	17	0:48	40,6	–
6 426 283 207 258 529	16	0:46	48,9	–
253 656 708 200 460 049	18	4:44	332,7	–
3 718 624 529 050 085 467	19	17:01	921,6	20 000
65 100 999 519 622 945 791 109	23	17:04	657,8	40 000
8 354 996 217 465 587 248 781	22	21:04	591,5	80 000
915 902 073 219 276 207 173	21	2:44	54,6	–
735 741 594 396 644 127 021 071	24	(letzter Faktor)		
Summe		69:23	2865	80 000

Tabelle 7.7: 291-stellige Zahl aus Faktoren der Längen 2, ..., 24 (Test 2), Zeit = [mm:ss]

Kapitel 8

Ausblick

Mit dem vorliegenden Programm lassen sich Zahlen, deren Primfaktoren aus bis zu ungefähr 20 Dezimalstellen bestehen, in wenigen Minuten zerlegen – ein einziger Faktor darf natürlich eine beliebige Größe haben, es reicht, wenn alle übrigen Faktoren bis auf den letzten gefunden werden. Auch Faktoren in der Größenordnung von 30 Dezimalstellen können innerhalb eines Tages mit hoher Wahrscheinlichkeit entdeckt werden, im Glücksfall sogar in wenigen Stunden oder gar Minuten.

In der Wissenschaft gibt es Wettbewerbe wie die RSA-Challenge[1]. Bei diesen Wettbewerben geht es um ganz andere Dimensionen. Aktuell kann man bei erfolgreicher Faktorisierung von RSA-640, einer Zahl, die in Binärdarstellung[2] 640 Ziffern lang ist, 20 000 US-Dollar gewinnen. Weitere Zahlen bis zu RSA-2048 warten auf ihre Zerlegung, wobei die Faktorisierung von RSA-2048 mit 200 000 $ belohnt wird. Die kleinste dieser Zahlen, RSA-640, besteht umgerechnet aus 193 Dezimalstellen; ihre beiden Primfaktoren besitzen ungefähr die Hälfte der Stellen. Daher scheint die Faktorisierung mit der hier vorgestellen Elliptic Curve Method momentan nicht sinnvoll zu sein – es sei denn, man hat Glück, und der Pseudozufallszahlengenerator wählt die passenden Kurvenparameter.

Um Zahlen mit deutlich höheren Primfaktoren als 30 Dezimalstellen faktorisieren zu können, sind einige Verbesserungen denkbar. Neben der Möglichkeit, wichtige Teile der Software, beispielsweise die Addition oder die Multiplikation auf Punkten, in **Assembler** zu schreiben, sollen hier zwei mögliche Erweiterungen bzw. Veränderungen des Programms angesprochen werden.

In unserer Implementierung wird zunächst eine zufällige Kurve sowie ein zufälliger Punkt der Kurve gewählt. Anschließend wird, falls noch kein Faktor gefunden wurde, auf diesem

[1] www.rsasecurity.com/rsalabs/ → challenges

[2] das Format von RSA-xx ist in der Literatur uneinheitlich: je nach Autor ist xx entweder binär oder dezimal. Selbst rsasecurity.com verwendet auf ihren Webseiten *beide* Schreibweisen gleichzeitig.

Punkt gerechnet. Tritt auch hier kein Faktor auf, werden neue Kurven / Punkte bestimmt. Dieser Vorgang kann auch parallel durchgeführt werden. Im einfachsten Fall können dazu Kopien des Programms auf verschiedenen Prozessoren oder Rechnern gestartet werden. Für eine **parallele Implementierung** fehlt im Wesentlichen ein Koordinator, der erkennt, dass einer der Teilprozesse einen Faktor gefunden hat, und dann unter Umständen alle Programme anhält und mit der kleineren Zahl neu startet. Je nach Einsatzzweck ist bei der Parallelisierung des Programms sowohl die Implementierung auf einem Mehrprozessorsystem denkbar, als auch der Einsatz einer Client/Server-Struktur für unabhängige Rechner, so dass beispielsweise zahlreiche Rechner in einem lokalen Netzwerk nach Primfaktoren suchen können, oder sich sogar weitere Computer über das Internet an der Suche nach Faktoren beteiligen können.

Eine andere Beschleunigungsmöglichkeit ist die Verwendung von **projektiven Koordinaten**. Durch deren Einsatz entfällt die Berechnung der Inversion, so dass hier ein deutlicher Geschwindigkeitszuwachs möglich ist. Weitere Details hierzu sind z. B. in [4] aufgeführt.

Literaturverzeichnis

[1] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory,* volume 1, section 8.8, page 233. MIT Press, first edition, 1996.

[2] E. Canfield, P. Erdős, and C. Pomerance. *On a problem of Oppenheim concerning "factorisatio numerorum".* J. Number Theory 17:1–28, 1983.

[3] Leonard S. Charlap and David P. Robbins. *An Elementary Introduction to Elliptic Curves.* CRD Expository Report 31, Princeton, 1988.

[4] Richard Crandall and Carl Pomerance. *Prime Numbers — A Computational Perspective.* Springer-Verlag, second edition, 2001.

[5] Neal Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, second edition, 1994.

[6] H. Lenstra. *Factoring integers with elliptic curves.* Annals of Mathematics, 2:649–673, 1987.

[7] Carl Pomerance. *Factoring.* Proceedings of Symposia in Applied Mathematics 42, American Mathematical Society (AMS), Providence, Rhode Island (US), 1990.

[8] A. Selberg. *An elementary proof of the prime-number theorem.* Annals of Mathematics 50 No. 2, pp. 305-313, 1949.

[9] Joseph H. Silverman. *The Arithmetic of Elliptic Curves.* Springer Verlag, New York, 1986.

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, den 8. September 2005