HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

MASTERARBEIT

Herr Dipl.-Inf. (FH)
Tom-Steve Watzke

Kryptoanalytische Verfahren für die Elliptische-Kurven-Kryptographie

Mittweida, 2011

Fakultät **Mathematik/Naturwissenschaften/Informatik**

MASTERARBEIT

Kryptoanalytische Verfahren für die Elliptische-Kurven-Kryptographie

Autor:
Tom-Steve Watzke

Studiengang:
Diskrete und Computerorientierte Mathematik

Seminargruppe:
ZD09w1

Erstprüfer:
Prof. Dr. rer. nat. Klaus Dohmen

Zweitprüfer:
Dipl.-Ing. Eberhard Richter

Einreichung:
Mittweida, 05.08.2011

Verteidigung/Bewertung:
Mittweida, August 2011

Bibliografische Angaben

Watzke, Tom-Steve: Kryptoanalytische Verfahren für die Elliptische-Kurven-Kryptographie, 69 Seiten, 10 Abbildungen, Hochschule Mittweida, Fakultät Mathematik/Naturwissenschaften/Informatik

Masterarbeit, 2011

Referat

In der vorliegenden Masterarbeit werden die neuesten Entwicklungen kryptoanalytischer Verfahren zur Lösung des diskreten Logarithmus in der Elliptischen-Kurven-Kryptographie zusammengefasst, um Schlussfolgerungen über die Sicherheit aktuell verwendbarer Schlüssellängen aufzustellen. Dabei werden auch derzeit eingesetzte Hardwarelösungen betrachtet, die ebenfalls die Sicherheit aktuell verwendbarer Schlüssellängen beeinflussen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbolverzeichnis

π	Kreiszahl Pi
m	Modul einer Restklasse
$\phi(m)$	Eulersche ϕ-Funktion
$\mathbb{K}$	Körper
$\mathbb{Z}/m\mathbb{Z}$	Restklassenring der ganzen Zahlen modulo m
$(\mathbb{Z}/m\mathbb{Z})^\times = \mathbb{Z}_m^\times$	Restklassengruppe der ganzen Zahlen modulo m bzgl. der Multiplikation
G	Abelsche Gruppe
g	Erzeugendes Element einer Gruppe G
$\langle g \rangle$	die durch g generierte zyklische Gruppe
E	Elliptische Kurve in einer affinen Ebene mit beliebigem a und b
$\mathbb{O}$	Punkt im Unendlichen
F	Eine beliebige homogene Funktion
f	Eine beliebige Funktion

1 Einleitung

1.1 Motivation

Das Thema der vorliegenden Masterarbeit wurde im Rahmen von Grundlagenforschungen für zukünftige Projekte der pitcom PROJECT GmbH definiert. Als Teil der pitcom Unternehmensgruppe ist sie unter anderem im Bereich Forschung und Entwicklung tätig. Aktuelle Entwicklungsfelder dieses Bereiches widmen sich der Verwendung mobiler Endgeräte und der Anpassung von Dienstleistungen zur Eignung für die mobile Umgebung.

Der Vorteil der mobilen Umgebung ist die unmittelbare Übertragbarkeit jeder Information - unabhängig von Zeit und Ort. Einsatzgebiete ergeben sich in den Anwendungen der Zeit- oder Standorterfassung und in der Einbeziehung von Sensoren oder Datenträgern, beispielsweise RFID-Chips. Die Ergebnisse lassen sich von Sicherheits- oder Wachdiensten und mobilen Ambulanzdiensten einsetzen und somit in derartigen Einsatzgebieten, in denen die mobile Daten-Akquirierung in Verbindung mit der Sicherheit oder Gesundheit von Menschen eine große Rolle spielt.

Je wichtiger die zu übertragenden Informationen sind, desto mehr Sicherheitsmechanismen sind einzubeziehen. Viele Verfahren der Kryptographie sind nicht geeignet für die Einbettung in mobile Endgeräte, die hinsichtlich Rechenleistung, Speicher und Batterielaufzeit beschränkt sind.

Insbesondere sind derartige Kryptosysteme nicht einsetzbar, welche die Verwendung großer Schlüssellängen erfordern. Aus großen Schlüssellängen folgt das Rechnen mit großen Zahlen, was wiederum eine hohe Laufzeit und viel Speicher benötigt. Für Schlüssellängen gelten allerdings auch gewisse Anforderungen, so dass Angriffe auf Kryptosysteme eine zu hohe Laufzeit aufweisen.

Kryptosysteme, die auf elliptischen Kurven und endlichen algebraischen Strukturen basieren, ermöglichen die Verwendung kürzerer Schlüssellängen als beispielsweise das RSA-Kryptosystem. Die Sicherheit Elliptischer-Kurven-Kryptosysteme beruht dabei auf der Berechnung des diskreten Logarithmus in der Gruppe einer elliptischen Kurve.

1.2 Zielsetzung

Das Ziel dieser Masterarbeit ist die Betrachtung der Entwicklung kryptoanalytischer Verfahren, die auf abelschen Gruppen elliptischer Kurven basieren. Neben alge-

braischen Verbesserungen und der Parallelisierung sollen schließlich auch Laufzeit-Auswirkungen von Hardwarelösungen zur Berechnung diskreter Logarithmen betrachtet werden.

Schließlich sind die Ergebnisse dieser Betrachtungen in Bezug auf die Sicherheit einsetzbarer Schlüssellängen mit den empfohlenen Schlüssellängen nach [ECR10] zu vergleichen. Dabei sind Aussagen zu treffen, welche Schlüssellängen für das mobile Umfeld empfohlen oder angewendet werden können.

1.3 Aufbau der Masterarbeit

Im zweiten Kapitel wird eine Einführung der Kryptographie mit Betrachtung der rechenaufwändigen Probleme und deren Laufzeit gegeben. Für Berechnungen diskreter Logarithmen werden die erforderlichen mathematischen Grundlagen über Kongruenzen dargestellt und anschließend Grundbegriffe der projektiven und affinen Ebenen und die Grundlagen der elliptischen Kurven und der darauf basierenden Kryptographie betrachtet.

Im dritten Kapitel wird die Kryptoanalyse und die Pollard ρ-Methode für abelsche Gruppen dargelegt. Dabei wird zuerst der von Pollard verwendete Algorithmus zum Zyklenfinden von Floyd und anschließend die Pollard ρ-Methoden für die Faktorisierung und diskrete Logarithmen dargestellt.

Im vierten Kapitel werden die von Edlyn Teske vorgeschlagenen Irrfahrten betrachtet, die im Mittel für bestimmte Parameter eine bessere Laufzeit im Gegensatz zu Pollards Irrfahrt aufweisen.

Im fünften Kapitel wird die parallele Kollisionssuche von Oorschot und Wiener dargestellt, die es ermöglicht die Pollard ρ-Methode derart zu parallelisieren, so dass sich ein linearer Beschleunigungsfaktor ergibt. Dabei wird auch die Anwendung auf die Pollard ρ-Methode betrachtet.

Im sechsten Kapitel wird ein Gruppenautomorphismus beschrieben, der die Menge der zu betrachtenden Gruppenelemente auf die Hälfte reduziert. Das Problem dabei sind die auftretenden Zyklen mit kurzer Periodenlänge. Dabei werden aktuelle Methoden vorgestellt, welche diese Zyklen versuchen zu vermeiden.

Im letzten Kapitel werden die Verkürzung der Laufzeit in Abhängigkeit von aktuellen Hardwarelösungen zusammengefasst. Daraus ergeben sich Schlussfolgerungen über die Sicherheit einsetzbarer Schlüssellängen für die Elliptische-Kurven-Kryptographie, die in direkter Korrelation stehen mit den empfohlenen Schlüssellängen [ECR10].

2 Grundlagen

2.1 Public-Key-Kryptographie

In der vorliegenden Masterarbeit werden grundlegende rechenaufwändige Probleme von Public-Key-Kryptosystemen untersucht. Es folgt eine kurze Einführung [HMV04, S. 2-4] in die Grundlagen der Kryptographie.

Die Kryptographie befasst sich mit der Informationssicherheit und den damit verbundenen Methoden und Verfahren, Informationen vor unbefugtem Verändern oder Lesen abzusichern. Das zugrundeliegende Kommunikationsmodell besteht aus zwei Kommunikationsteilnehmern, beispielsweise Alice und Bob, und einem unsicheren Übertragungskanal zwischen diesen.

Sicherheitsrisiken ergeben sich bei der Übertragung vertraulicher Informationen über unsichere Übertragungskanäle. Informationen können dabei durch nicht autorisierte Personen gelesen oder verändert werden, bevor diese Bob erreichen. Daraus ergeben sich die folgenden Ziele der Kryptographie [HMV04, S. 2-3]:

- **Vertraulichkeit:** Schutz der Information vor unberechtigtem Lesen,
- **Integrität:** Schutz der Informationen vor unautorisiertem Verändern,
- **Authentizität:** Eindeutige Identifizierbarkeit des Urhebers der Information,
- **Verbindlichkeit:** Nichtabstreitbarkeit des Ursprungs der Information.

Die Methoden zur Absicherung von Informationen der modernen Kryptographie sind unterteilt in symmetrische und asymmetrische Verfahren. Symmetrische Verfahren verwenden einen geheimen Schlüssel, um die Informationen zu Ver- und Entschlüsseln. Der Nachteil dabei ist die Übertragung des geheimen Schlüssels.

Asymmetrische Kryptoverfahren verwenden je Kommunikationsteilnehmer ein Schlüsselpaar: einen öffentlichen und einen privaten Schlüssel. Mit einem öffentlichen Schlüssel werden Informationen für den Besitzer des privaten Schlüssels verschlüsselt, dessen digitale Signatur überprüft oder dieser authentifiziert. Der Besitzer eines privaten Schlüssels ist dabei imstande, die mit dem öffentlichen Schlüssel verschlüsselten Informationen zu entschlüsseln, digitale Signaturen zu erzeugen oder sich zu authentifizieren.

Der Vorteil asymmetrischer Kryptoverfahren im Gegensatz zu symmetrischen ist, dass die kommunizierenden Parteien keinen gemeinsamen Schlüssel kennen müssen. Asymmetrische Kryptosysteme werden daher auch als Public-Key-Verfahren bezeichnet.

2.2 Rechenaufwändige Probleme

Algorithmen der modernen Kryptographie basieren auf Operationen in endlichen algebraischen Strukturen der diskreten Mathematik, beispielsweise endlichen Körpern oder endlichen zyklischen Gruppen. Weiterführende Literatur dazu ist beispielsweise [KM09], [KK10] und [SSW03]. Die Komplexität der Algorithmen wird durch die Landau-Notation beschrieben, siehe unter anderem [Hof09]. Dabei wird die Laufzeit der Algorithmen mit dem asymptotischen Verhalten von Funktionen verglichen.

Definition 2.1 Sei n die Bitlänge einer Eingabe in einen Algorithmus, $c \in \mathbb{R}$ und $c \geq 0$, so heißt ein Algorithmus **polynomial**, wenn er eine asymptotische Laufzeit $f_p \in \mathcal{O}(n^c)$ hat und **exponentiell**, wenn seine asymptotische Laufzeit $f_e \in \Omega(2^{cn})$ ist.

Ein asymmetrisches Kryptosystem ist RSA. Bei diesem Verfahren wird das Produkt n von zwei großen Primzahlen p und q gebildet. Dieses Produkt ist einfach zu berechnen, im Gegensatz zur Zerlegung von n in p und q, ohne p und q zu kennen.

Satz 2.2 *(Fundamentalsatz der Arithmetik) Jede natürliche Zahl $z > 1$ ist entweder selbst eine Primzahl oder lässt sich als Produkt von Primzahlpotenzen darstellen und ist bis auf die Reihenfolge dieser Faktoren eindeutig. Für jedes $z \in \mathbb{N}$ existieren $n \in \mathbb{N}$, $p_1, \ldots, p_n \in \mathbb{N}$ Primzahlen und $a_1, \ldots, a_n \in \mathbb{N}$, so dass gilt:*

$$z = \prod_{i=1}^{n} p_i^{a_i} \tag{2.1}$$

Definition 2.3 Das Produkt von Primzahlpotenzen aus (2.1) wird **kanonische Primfaktorzerlegung** oder **Faktorisierung** einer Zahl $z \in \mathbb{N}$ genannt, wenn für die Primzahlpotenzen gilt: $p_1 < \ldots < p_n$.

Bemerkung 2.4 Bis heute ist kein polynomieller Algorithmus für die Faktorisierung bekannt, weshalb die Faktorisierung als rechenaufwändiges Problem für das RSA-Krytosystem gewählt wurde.

Der diskrete Logarithmus ist ein weiteres rechenaufwändiges Problem und die Basis vieler kryptographischer Verfahren, beispielsweise des ElGamal-Verfahrens oder von Elliptischen-Kurven-Kryptosystemen. Die Berechnungen werden in endlichen algebraischen Strukturen durchgeführt. In der vorliegenden Masterarbeit werden folgende Notationen und Definitionen verwendet.

Definition 2.5 Sei $(G, \times)$ eine endliche zyklische Gruppe bezüglich der Multiplikation, die durch das Element $g \in G$ generiert wird. Die Schreibweise $\langle g \rangle$ bezeichnet die **zyklische Gruppe**, die durch g generiert wird und $|\langle g \rangle|$ oder $|G|$ bezeichnet die **Ordnung** von G, der kleinsten Zahl $a \in \mathbb{N}$, für die gilt $g^a = 1$. Ein erzeugendes Element g wird **primitiv** oder **Primitivwurzel** genannt, wenn seine Ordnung gleich $\phi(m)$ ist.

Definition 2.6 Seien $m \in \mathbb{N}$ prim, $G = \langle g \rangle$ eine abelsche Gruppe erzeugt durch $g \in G$ und $x \in \mathbb{Z}$. Das kleinste positive $\boldsymbol{x}$ wird **diskreter Logarithmus** oder **Index** genannt, wenn für $y \in G$ gilt:

- $\boldsymbol{y = g^x}$ für G bezüglich der Multiplikation
- $\boldsymbol{y = xg}$ für G bezüglich der Addition

Die Sicherheit asymmetrischer Kryptosysteme, die auf diesen Problemen basieren, ist gekennzeichnet durch eine hohe Laufzeit zur Berechnung des diskreten Logarithmus oder der Faktorisierung. Dabei ist die asymptotische Laufzeit kryptoanalytischer Verfahren zur Lösung der Probleme nicht polynomial und benötigt damit für eine gegebene Schlüssellänge eine entsprechend hohe Laufzeit.

2.3 Kongruenzen

Für Berechnung des diskreten Logarithmus x in endlichen abelschen Gruppen werden Kongruenzen gelöst, daher werden Kongruenzen im Folgenden eingeführt. Weiterführende Literatur dazu ist [Bun08] und [SSW03].

Definition 2.7 Für $s,\ t,\ d \in \mathbb{Z} \setminus \{0\}$ ist $\gcd(s,\ t) = d$ der **größte gemeinsame Teiler** (greatest common divisor) von s und t, der sich durch den euklidischen Algorithmus, dargestellt in Algorithmus 2.1, berechnen lässt.

Es gilt folgender Satz mit Beweis in [SSW03], aus dem der Algorithmus 2.2 folgt.

Satz 2.8 *Seien $s,\ t,\ d \in \mathbb{Z} \setminus \{0\}$ und $d = \gcd(s,\ t)$. So existieren $u,\ v \in \mathbb{Z}$ mit*

$$gcd(s,\ t) = d = su + tv \tag{2.2}$$

Definition 2.9 Für $s,\ t,\ d \in \mathbb{Z} \setminus \{0\}$ und $u,\ v \in \mathbb{Z}$ ist $\mathrm{egcd}(s,\ t) = (d,\ u,\ v)$ der **erweiterte größte gemeinsame Teiler**, der für die Gleichung (2.2) neben dem gemeinsamen Teiler d auch u und v ermittelt. Der erweiterte euklidische Algorithmus, dargestellt in Algorithmus 2.2, berechnet dieses Tripel $(d,\ u,\ v)$.

Algorithmus 2.1: $\gcd(s,\ t)$

Input : $s,\ t \in \mathbb{Z}$
Output : d mit $d \in \mathbb{Z}$
begin
  if $t = 0$ **then return** s;
  $d \leftarrow \gcd(t,\ s \mod t)$;
  return d
end

Algorithmus 2.2: $\mathrm{egcd}(s,\ t)$

Input : $s,\ t \in \mathbb{Z}$
Output : $(d,\ u,\ v)$ mit $d,\ u,\ v \in \mathbb{Z}$
begin
  if $t = 0$ **then return** $(s,\ 1,\ 0)$;
  $(d',\ s',\ t') \leftarrow \mathrm{egcd}(t,\ s \mod t)$;
  return $\left(d', t', s' - \left\lfloor \frac{s}{t} \right\rfloor t'\right)$
end

Definition 2.10 Seien $m,\ s,\ t \in \mathbb{Z}$ mit $m > 0$. Es heißt ***a* kongruent *b* modulo *m*** genau dann, wenn $m \mid (s-t)$ gilt, in Zeichen: $s \equiv t \pmod{m}$.

Definition 2.11 Sind $m,\ s,\ t,\ x \in \mathbb{Z}$ mit $m > 0$, so bezeichnet man

$$sx \equiv t \pmod{m} \tag{2.3}$$

als **lineare Kongruenz**. Dabei existiert eine Lösungsmenge $X = \{x \in \mathbb{Z};\ m \mid (sx-t)\}$.

Definition 2.12 Seien $n \in \mathbb{N}$ und $m_i,\ s_i,\ t_i \in \mathbb{Z}$ mit $m_i > 0$ für $i = 1,\ \ldots,\ n$. Ein System von linearen Kongruenzen der Form

$$s_1 x \equiv t_1 \pmod{m_1}, \quad \ldots, \quad s_n x \equiv t_n \pmod{m_n} \tag{2.4}$$

heißt **simultane lineare Kongruenzen**.

Im Folgenden wird die Lösbarkeit linearer Kongruenzen im Satz 2.13 betrachtet und die Form der Lösung durch den Beweis des Satzes 2.13 gegeben, der auch in [Bun08, S. 83ff.] bewiesen ist.

Satz 2.13 *Seien $d,\ s,\ m \in \mathbb{Z} \setminus \{0\}$ mit $d \geq 1$, $m \geq 1$ und $d = gcd\,(s,\ m)$. Eine lineare Kongruenz (2.3) ist genau dann lösbar, wenn $d \mid t$ gilt. Gilt $d \mid t$, dann hat (2.3) genau d Lösungen. Sind dabei s und m teilerfremd, so existiert eine eindeutige Lösung.*

Beweis: Seien $y,\ u,\ v \in \mathbb{Z}$. Laut der Definition 2.10 gilt für (2.3), dass $m \mid (sx-t)$:

$$my = sx - t \quad \Rightarrow \quad sx - my = t. \tag{2.5}$$

Für (2.5) gilt laut den Gesetzen der Teilbarkeit, dass $s \mid t$ und $m \mid t$. Mit $\mathrm{egcd}(s,\ m) = (d,\ u,\ v)$ als Lösung von

$$su + mv = d \tag{2.6}$$

gilt weiterhin, dass $s \mid d$, $d \mid m$ und $d \mid t$. Daher gibt es für den Fall $d \nmid t$ keine Lösung für (2.3). Wird (2.3) in eine reduzierte Darstellung der Form $\frac{s}{d}x \equiv \frac{t}{d} \pmod{\frac{m}{d}}$ überführt, liefert $\mathrm{egcd}\left(\frac{s}{d},\ \frac{m}{d}\right) = (d',\ u',\ v')$ mit $d' = 1$ und u' dem Inversen bezüglich der Multiplikation zu s. Mit dem Vergleich von u mit u' folgt, dass dieses Inverse von s bereits durch $\mathrm{egcd}(s,\ m)$ ermittelt wird. Damit folgt die kleinste Lösung für x:

$$x \equiv \frac{tu}{d} \pmod{\frac{m}{d}} \tag{2.7}$$

Ist $d = 1$, also s und m teilerfremd, enthält die Lösungsmenge $X = \{x\}$ die eindeutige Lösung x. Ist $d > 1$ hat die lineare Kongruenz (2.3) mehrere Lösungen und es ergibt sich aus (2.5) und (2.6) für x die Lösungsmenge

$$X = \left\{x,\ x + \frac{m}{d},\ x + \frac{2m}{d}, \ldots, x + \frac{(d-1)m}{d}\right\} \tag{2.8}$$

□

Die Methode zur Lösung von simultanen linearen Kongruenzen basiert auf dem **Chinesischen Restsatz**. Der Satz 2.14 wurde in [Bun08, S. 89ff.] bewiesen und besteht aus dem Algorithmus 2.3.

Satz 2.14 *(**Chinesischer Restsatz**) Sei $n \in \mathbb{N}$ und seien $m_1, \ldots, m_n \in \mathbb{Z}$ paarweise teilerfremd, so existiert zu beliebigen Zahlen $a_1, \ldots, a_n \in \mathbb{Z}$ ein $x \in \mathbb{Z}$ mit*

$$x \equiv a_i \pmod{m_i} \quad \forall i = 1, \ldots, n \tag{2.9}$$

Bemerkung 2.15 Sind $m_1, \ldots, m_n$ nicht paarweise teilerfremd, so ist $M = \mathtt{lcm}(m_1, m_2, \ldots, m_n)$ das kleinste gemeinsame Vielfache aller m_i für $i = 1, \ldots, n$. Andernfalls liefert $\mathtt{lcm}(m_1, m_2, \ldots, m_n)$ das Produkt aller m_i.

Algorithmus 2.3: Chinesischer Restsatz

Input : $a_i,\ m_i \quad \forall i = 1, \ldots, n$
Output : $x \in \mathbb{Z}$
begin
  $M \leftarrow \mathtt{lcm}(m_1, m_2, \ldots, m_n)$;
  $x \leftarrow 0$;
  for $i \leftarrow 1$ **to** n **do**
    $s_i \leftarrow \frac{M}{m_i}$;
    $(d_i, u_i, v_i) \leftarrow \mathtt{egcd}(s_i, m_i)$;
    $x \leftarrow x + a_i s_i u_i$;
  return $x \bmod M$
end

2.4 Affine und projektive Ebenen

Elliptische Kurven sind kubische Kurven, die in projektiven Ebenen definiert sind. Eine einführende Betrachtung projektiver Ebenen ist deshalb notwendig, da unter diesen Bedingungen elliptische Kurven abelsche Gruppen bilden. Analog zu [Was03, Kap. 2.3] folgt eine kurze Einführung über projektive Ebenen und dem Punkt im Unendlichen. Im Folgenden sei $\mathbb{K}$ ein Körper.

Definition 2.16 Seien $A,\ B \in \mathbb{K}^3 \setminus \{0\}$ mit $A = (a_0,\ a_1,\ a_2)$ und $B = (b_0,\ b_1,\ b_2)$. Die Äquivalenzrelation $\sim$ für A und B ist wie folgt definiert:

$$(a_0,\ a_1,\ a_2) \sim (b_0,\ b_1,\ b_2) \quad \Leftrightarrow \quad \exists \lambda \in \mathbb{K} \setminus \{0\} : a_i = \lambda b_i \quad \forall i \in \{0,\ 1,\ 2\} \tag{2.10}$$

Die Schreibweise $(x : y : z)$ bezeichne die Äquivalenzklasse von (x, y, z). Eine **projektive Ebene** $\mathbb{P}^2_{\mathbb{K}}$ über $\mathbb{K}$ ist die Menge der Äquivalenzklassen $\left\{(x : y : z) \in \mathbb{K}^3 \setminus \{0\}\right\} / \sim$.

Definition 2.17 Punkte $(x : y : z) \in \mathbb{P}^2_{\mathbb{K}}$ mit $z \neq 0$ heißen **endlich**, es gilt $(x : y : z) = (x/z : y/z : 1)$. Punkte $(x : y : z)$ mit $z = 0$ werden **Punkte im Unendlichen** genannt.

Definition 2.18 Die Punktmenge $\{(x,\ y) \in \mathbb{K} \times \mathbb{K}\} = \mathbb{A}^2_{\mathbb{K}}$ heißt **affine Ebene** über $\mathbb{K}$.

Die Definitionen der affinen und projektiven Ebene zeigen, dass eine affine Ebene eine Teilmenge einer projektiven Ebene ist, wenn eine Abbildung von $\mathbb{A}^2_{\mathbb{K}} \to \mathbb{P}^2_{\mathbb{K}}$ mit $(x,\ y) \mapsto (x : y : 1)$ eingeführt wird. Das bedeutet also, dass eine affine Ebene die endlichen Punkte einer projektiven Ebene repräsentiert.

Definition 2.19 Ein Polynom $F(x,\ y,\ z)$ heißt **homogen** vom Grad $d \in \mathbb{N}$, wenn es aus der Summe aller Terme der Form $ax^iy^jz^k$ mit $a \in \mathbb{K}$ und $i + j + k = d$ besteht.

Lemma 2.20 *Ist das Polynom F homogen vom Grad d, so gilt $F(\lambda x,\ \lambda y,\ \lambda z) = \lambda^d Fy(x,\ y,\ z)\ \ \forall \lambda \in \mathbb{K}$.*

Folgerung 2.21 *Ist F homogen vom Grad d und $(x_A,\ y_A,\ z_A) \sim (x_B,\ y_B,\ z_B)$, so gilt $F(x_A,\ y_A,\ z_A) = 0$ nur dann, wenn $F(x_B,\ y_B,\ z_B) = 0$.*

Beispiel 2.22 Sei $f(x,\ y,\ z) = 3y^2 - 2x - z$. Es gilt $f(1,\ 1,\ 1) = 0$ und scheint damit im Punkt $(1 : 1 : 1)$ aufzugehen. Jedoch ist $f(3,\ 3,\ 3) = 18$ und $(1 : 1 : 1) = (3 : 3 : 3)$, was im Widerspruch zur Folgerung 2.21 steht. Zur Vermeidung solcher Probleme werden homogene Polynome verwendet.

Durch Einfügen geeigneter Potenzen von z wird ein Polynom homogen. Für das Beispiel 2.22 folgt damit ein $F(x,\ y,\ z) = 3y^2z - 2xz^2 - z^3$. Werden die Polynome zweier paralleler Geraden der affinen Ebene in homogene Polynome überführt und der Schnittpunkt gesucht, ergibt sich ein unendlicher Punkt.

Definition 2.23 Ein unendlicher Punkt einer projektiven Ebene ist $\mathbb{O} = (0 : 1 : 0) = (0 : -1 : 0)$, der sich als Schnittpunkt affiner paralleler Geraden ergibt.

Die Betrachtungen dieser Masterarbeit werden sich auf endliche algebraische Strukturen beschränken, deren Charakteristik char($\mathbb{K}$) weder zwei noch drei ist. Für weitere allgemeine Betrachtungen elliptischer Kurven über Restklassenkörper mit Charakteristiken zwei oder drei sei beispielsweise auf [HMV04], [CF03] oder [Was03] verwiesen.

Definition 2.24 Ein Polynom heißt **nicht-singulär**, wenn es keine mehrfachen Nullstellen hat. Diese werden auch Singularitäten genannt.

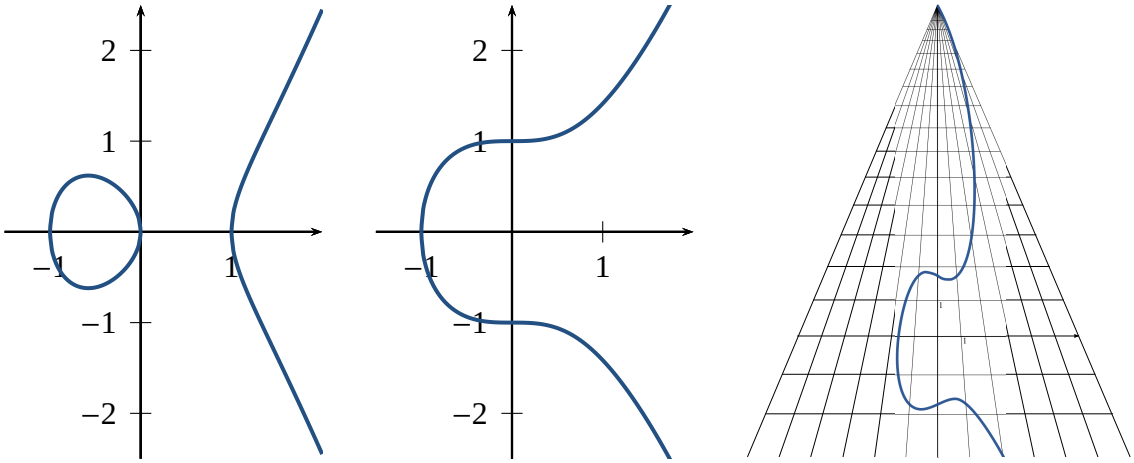
Definition 2.25 Seien $a,\ b \in \mathbb{K}$, $F(x,\ y,\ z) := y^2z - x^3 - axz^2 - bz^3 \in \mathbb{K}[x,\ y,\ z]$ ein homogenes Polynom vom Grad 3, wobei char($\mathbb{K}$) $\notin \{2,\ 3\}$. Die wie folgt definierte Nullstellenmenge $E_{\mathbb{P}}$ heißt **elliptische Kurve** über $\mathbb{P}^2_{\mathbb{K}}$, wenn diese an keiner Stelle singulär ist.

$$E_{\mathbb{P}} := \left\{(x : y : z) \in \mathbb{P}^2_{\mathbb{K}}; F(x,\ y,\ z) = 0\right\} \tag{2.11}$$

Definition 2.26 Seien $a,\ b \in \mathbb{K},\ f(x,\ y) := y^2 - x^3 - ax - b \in \mathbb{K}[x,\ y]$ die affine oder nichthomogene Gleichung der Kurve $E_{\mathbb{P}}$. Die wie folgt definierte Kurve heißt **elliptische Kurve** $E_{\mathbb{A}}$ über der affine Ebene $\mathbb{A}^2_{\mathbb{K}}$:

$$E_{\mathbb{A}} := \{(x,\ y) \in \mathbb{K} \times \mathbb{K};\ f(x,\ y) = 0\} \cup \{\mathbb{O}\} \tag{2.12}$$

Die graphische Darstellung von elliptischen Kurven ist über den meisten Körpern wenig aussagekräftig, deshalb werden die folgenden Betrachtungen der Operationen vorerst im Zahlenkörper der reellen Zahlen $\mathbb{R}$ dargestellt. Die Abbildung 2.1 zeigt drei verschiedene nicht-singuläre elliptische Kurven, wobei die Kurve (3), erzeugt mit einem JAVA-Applet in [Lau99], perspektivisch dargestellt ist und den Sachverhalt der Definition 2.26 veranschaulicht.



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

Abbildung 2.1: Beispiele elliptischer Kurven

2.5 Gruppenstruktur elliptischer Kurven

Die Punktmenge einer elliptischen Kurve $E = E_{\mathbb{A}}$ wurde im letzten Kapitel eingeführt, dabei werde für weitere Betrachtungen die elliptische Kurve $E_{\mathbb{A}}$ verwendet. Die elliptische Kurve E wird im Weiteren in der folgenden verkürzten Form angegeben:

$$E : y^2 = x^3 + ax + b \ \text{ mit } \ a,\ b \in \mathbb{K} \tag{2.13}$$

Für die Bildung einer Gruppe über einer elliptischen Kurve E ist eine innere Verknüpfung von Punkten in E notwendig. Zuerst wird dabei eine Verknüpfung $\times$ definiert.

Definition 2.27 Seien $P,\ Q \in E$ zwei Punkte einer elliptischen Kurve E. Die binäre innere Verknüpfung $\times$ bezeichnet das Produkt zweier Punkte P und Q und liefert den dritten Schnittpunkt der Geraden $\overline{PQ}$ mit E.

Für die Verknüpfung $\times$ gilt nun folgender Satz 2.28 mit Beweis in [KK10, S. 234].

Satz 2.28 *[KK10, Satz 13.7] Seien $P = (x_P, y_P)$, $Q = (x_Q, y_Q) \in E \setminus \{\mathbb{O}\}$ zwei Punkte der elliptischen Kurve $E : y^2 = x^3 + ax + b$. Dann gilt $\mathbb{O} \times \mathbb{O} = \mathbb{O}$, $\mathbb{O} \times P = (x_P, -y_P) =: -P$ und*

$$P \times Q = \begin{cases} \mathbb{O}, & \text{falls } P = -Q \\ (u, v(u - x_P) + y_P), & \text{sonst} \end{cases} \quad \text{mit}$$

$$u = v^2 - x_P - x_Q \quad \text{und} \quad v = \begin{cases} \frac{y_P - y_Q}{x_P - x_Q}, & \text{falls } P \neq Q \wedge P \neq -Q \\ \frac{3x_P^2 + a}{2y_P}, & \text{falls } P = Q \neq -P \end{cases}.$$

Definition 2.29 Die binäre innere Verknüpfung $+$, bezeichnet als **Punktaddition**, für zwei Punkte $P, Q \in E$ wird mithilfe der Verknüpfung $\times$ auf E definiert als

$$P + Q := \mathbb{O} \times (P \times Q) = -(P \times Q). \tag{2.14}$$

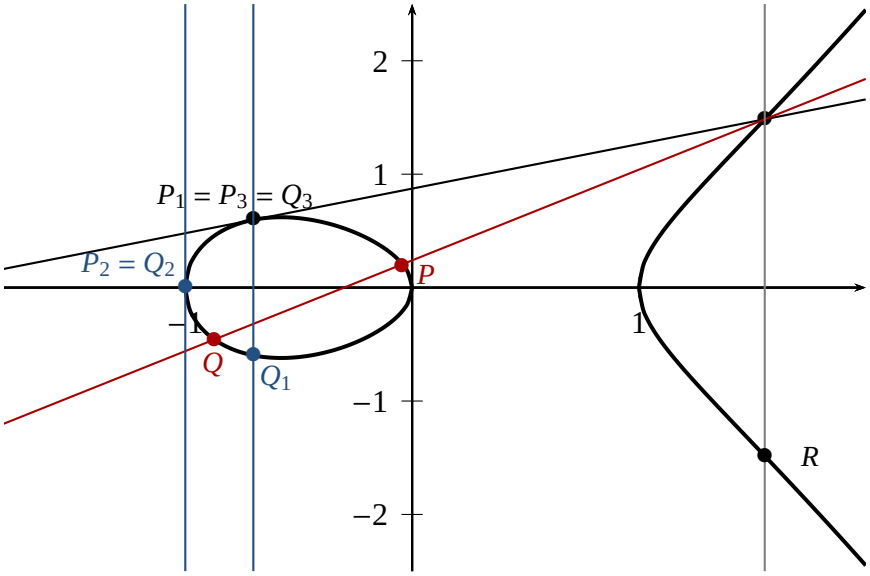


Abbildung 2.2: Punktaddition von P und Q und Sonderfälle $P_i + Q_i$ für $i = 1, 2, 3$

Bemerkung 2.30 Die Punktaddition lässt sich grafisch beschreiben: Lege durch die beiden Punkte P und Q eine Gerade. Sie schneidet die Kurve E in einem weiteren Punkt PQ. Schließlich schneidet die Gerade durch die Punkte $\mathbb{O}$ und PQ die Kurve E in einem Punkt R, welcher der resultierende Punkt ist.

Definition 2.31 Die Punktaddition eines Punktes P mit sich selbst wird auch **Punktverdopplung** genannt. Dabei wird die Schreibweise $P + P = PP = 2P$ verwendet und ergibt beispielsweise für $3P = 2P + P$.

Die Punkte einer elliptischen Kurve E bilden zusammen mit der inneren Verknüpfung *Punktaddition* eine abelsche Gruppe bezüglich der Addition. Der Satz 2.32 fasst diese Aussage zusammen und wurde auch in [KK10, S. 237] bewiesen.

Satz 2.32 *Zusammen mit der Punktaddition $+$ bildet eine elliptische Kurve E eine abelsche Gruppe $(E, +)$ mit neutralem Element $\mathbb{O}$.*

Beweis: Die Kommutativität der Punktaddition folgt aus der Kommutativität der Verknüpfung $\times$. Der Punkt $\mathbb{O}$ im Unendlichen ist ein neutrales Element, denn mit der Kommutativität gilt für alle $P \in E$ der elliptischen Kurve E, dass $\mathbb{O} + P = \mathbb{O} \times (\mathbb{O} \times P) = \mathbb{O} \times (-P) = P$. Für jeden Punkt $P \in E$ existiert ein inverses Element, denn es gilt $P + (-P) = \mathbb{O} \times (P \times (-P)) = \mathbb{O} \times \mathbb{O} = \mathbb{O}$. Der Nachweis des Assoziativgesetzes ist sehr aufwendig und wurde in [Was03, Kap. 2.4] bewiesen. □

Lemma 2.33 *Sei $n \in \mathbb{N}$. Jeder Punkt $P \in E$ erzeugt mit der Punktverdopplung eine Untergruppe $U_E = \langle P \rangle$ von E mit $n+1$ Punkten, die neue Punktmenge hat die Form*

$$U_E = \{P, 2P, \ldots, nP\} \cup \{\mathbb{O}\} \tag{2.15}$$

2.5.1 Anzahl der Punkte einer elliptischen Kurve

Die Kenntnis über die Mächtigkeit der Punktmenge einer elliptischen Kurve, auch Gruppenordnung genannt, ist für kryptologische Anwendungen ein wichtiges Indiz für die Sicherheit. Der *Algorithmus von Schoof* [KK10] ist ein Verfahren zur Bestimmung dieser Ordnung.

Hat beispielsweise die Gruppenordnung kleine Primteiler, so lässt sich diese Gruppe in Gruppen kleinerer Ordnung zerlegen und erleichtert die Berechnung des diskreten Logarithmus über dieser elliptischen Kurve. Für kryptoanalytische Algorithmen ist meist eine Abschätzung dieser Ordnung ausreichend, es gilt folgender Satz von Hasse [Sil86].

Satz 2.34 *(Hasse) Sei $q \in \mathbb{N}$ und $\mathbb{K}_q$ ein Restklassenkörper, so liegt die Anzahl $n \in \mathbb{N}$ der Punkte einer elliptischen Kurve E im Intervall: $q + 1 - 2\sqrt{q} \leq n \leq q + 1 + 2\sqrt{q}$.*

Beispiel 2.35 $q = 2^{10} + 7 = 1073741831$ ist prim, also liegt die Anzahl der Punkte jeder elliptischen Kurve über $\mathbb{Z}_q$ im Intervall $1073676296 \leq n \leq 1073807368$.

2.6 Elliptische-Kurven-Kryptographie

Für die Verwendung elliptischer Kurven in der Kryptographie werden Parameter festgelegt, auf Basis welcher Informationen signiert, ver- und entschlüsselt werden. Weiterführende Literatur sind beispielsweise [HMV04] und [CF03].

Definition 2.36 Sei W eine Menge und w ein zufällig gewähltes Element von W, wobei die Auswahl mit Zurücklegen gleichverteilt sei. Die Schreibweise für solch eine

zufällige Auswahl sei fortan $w \in_R W$.

Definition 2.37 Seien $E(\mathbb{K}_p)$ eine elliptische Kurve über einen Restklassenkörper $\mathbb{K}_p$ und $V \in E(\mathbb{K}_p)$, w eine zyklische Untergruppe $\langle V \rangle$ mit primer Ordnung $m = \langle V \rangle$ erzeugt. Die Elemente des Quadrupels $(p,\ E,\ V,\ m)$ werden als **öffentliche Domain-Parameter** oder kurz Domain-Parameter bezeichnet. Ein $x \in_R \mathbb{N},\ x < m$ wird **privater Schlüssel** und $Y \in E(\mathbb{K}_p)$ **öffentlicher Schlüssel** genannt, wenn gilt $Y = xV$. Die Bestimmung eines solchen x mit gegebenen Domain-Parametern und Q wird als das **Elliptische-Kurven-Diskrete-Logarithmen-Problem** (ECDLP) bezeichnet.

Die grundlegenden Verfahren für die Ver- und Entschlüsselung werden als nächstes kurz eingeführt [HMV04, S. 13ff.]. Für die Verschlüsselung wird ein Klartext w durch einen Punkt $W \in E(\mathbb{K}_p)$ repräsentiert und anschließend addiert zu kY mit $k \in_R \mathbb{N},\ k < m$. Der Absender schickt dem Empfänger die Punkte $Z_1 = kV$ und $Z_2 = W + kY$. Diese Verschlüsselung wird in Algorithmus 2.4 zusammengefasst.

Algorithmus 2.4: Verschlüsselung basierend auf ElGamal und elliptischen Kurven

Input : Domain-Parameter $(p,\ E,\ V,\ m)$, öffentlicher Schlüssel Y und Klartext w.
Output : Chiffrentext $(Z_1,\ Z_2)$
begin
  $W \leftarrow$ Punktrepräsentant von w;
  $k \in_R \{1,\ 2,\ \ldots,\ m-1\}$;
  $Z_1 \leftarrow kV$;
  $Z_2 \leftarrow W + kY$;
  return $(Z_1,\ Z_2)$
end

Der Empfänger berechnet mittels seines privaten Schlüssels den Punkt $W = Z_2 - xZ_1$, denn es gilt $xZ_1 = d(kV) = k(xV) = kY$. Schließlich wird aus W der Klartext w extrahiert. Diese Entschlüsselung wird in Algorithmus 2.5 zusammengefasst.

Algorithmus 2.5: Entschlüsselung basierend auf ElGamal und elliptischen Kurven

Input : Domain-Parameter $(p,\ E,\ V,\ m)$, privater Schlüssel x und Chiffrentext $(Z_1,\ Z_2)$.
Output : Klartext w
begin
  $W \leftarrow Z_2 - xZ_1$;
  $w \leftarrow$ Klartext-Extrahieren aus W;
  return w
end

In [ECR10, S. 29ff.] wurden verschiedene Mindest-Sicherheitsstufen, siehe Tabelle 2.1, für asymmetrische Kryptosysteme empfohlen, um vor entsprechende Angreifer zu schützen. Je nach verfügbarem Kapital werden verschiedene Hardwarelösungen eingesetzt: PCs, programmierbare Logikgatter-Anordnungen (FPGAs) oder anwen-

dungsspezifische integrierte Schaltkreise (ASICs). Dabei werden bekannte Schlüssellängen von symmetrischen Kryptosystemen mit asymmetrischen Kryptosystemen verglichen und in Sicherheitsstufen eingeordnet, was in Tabelle 2.2 dargestellt ist.

Angreifer	**Kapital [$]**	**Hardware**	**MS**
Hacker	0	PC	53
	400	PCs/FPGA	58
	0	Schadsoftware	73
Organisationen	10000	PCs/FPGA	64
	300000	FPGA/ASIC	68
	1000000	FPGA/ASIC	78
Geheimdienste	300000000	ASIC	84

Tabelle 2.1: Empfohlene Mindest-Sicherheitsstufe (MS) für Schlüssel laut ECRYPT2

Sicherheitsstufen (bits)	80	112	128	160	192	256
EC-Parameter m	160	224	256	320	384	512
RSA-Modul m von $\mathbb{Z}_m$	1024	2048	3072	5312	7936	15424

Tabelle 2.2: RSA und Elliptische-Kurven (EC) Schlüssellängen für gleiche Sicherheitsstufen

Die Certicom Corporation hat für die verschiedenen Sicherheitsstufen als Standard [Cer00] geeignete Domain-Parameter vorgeschlagen und hat 1997 für einige Sicherheitsstufen Herausforderungen und Übungen [Cer97] mit verschiedenen Preisen dotiert. In Tabelle 2.3 sind die Herausforderungen und Übungen für Elliptische Kurven mit primer Ordnung dargestellt.

	Bits	**Bezeichnung**	**Geschätzte Anzahl von Maschinentagen**	**Preis**	**gelöst im Jahr**
Übungen	79	ECCp-79	146	Buch	1997
	89	ECCp-89	4360	Buch	1998
	97	ECCp-97	71982	$5000	1998
HF St 1	109	ECCp-109	9000000	$10000	2002
	131	ECCp-131	23000000000	$20000	
HF St 2	163	ECCp-163	$2.3 \cdot 10^{15}$	$30000	
	192	ECCp-191	$4.8 \cdot 10^{19}$	$40000	
	239	ECCp-239	$1.4 \cdot 10^{27}$	$50000	
	359	ECCp-359	$3.7 \cdot 10^{45}$	$100000	

Tabelle 2.3: Certicom ECC Übungen und Herausforderungen (HF) in Stufen (St)

Das zuletzt gelöste ECDLP ist die in dem Standart [Cer00] definierte elliptische Kurve secp112r1 mit einer 112-Bit langen Primzahl. Das Problem wurde durch das LACAL der EPF von Lausanne `http://lacal.epfl.ch` mit mehr als 200 Playstation 3 Spielekonsolen im Zeitraum vom 13. Januar bis 8. Juli 2009 berechnet [BKK+09c].

3 Kryptoanalytische Grundverfahren

3.1 Übersicht und Methoden

Die Untersuchung kryptographischer Verfahren auf Schwachstellen und der damit verbundenen Absicherung gegen Angriffe auf verschlüsselte Informationen wird Kryptoanalyse genannt. Betrachtet wird dabei unter anderem das Diskrete-Logarithmen-Problem auf Basis von Gruppen elliptischer Kurven. Die Sicherheit umfasst dabei nicht das Problem selbst, sondern die hohe Laufzeit zur Lösung des Problems.

Beispiel 3.1 Ein RSA-Kryptosystem gilt als sicher [ECR10], wenn ein primes $m > 2^{1024}$ gewählt wird. Dahingegen gilt ein Elliptisches-Kurven-Kryptosystem als sicher, wenn ein primes $m > 2^{160}$ verwendet wird, siehe Tabelle 2.2.

In diesem und den folgenden Kapiteln werden algebraische Angriffe auf das Diskrete-Logarithmen-Problem für Gruppen elliptischer Kurven oder allgemeiner für abelsche Gruppen untersucht. Für folgende Betrachtungen sei nun $G = (\mathbb{Z}/m\mathbb{Z})^\times = \mathbb{Z}_m^\times$ die Restklassengruppe modulo m bezüglich der Multiplikation.

Die **Brute-Force-Methode** bezeichnet die Enumeration aller möglichen Lösungen, um den diskreten Logarithmus x in $y = g^x$ mit $y,\ g \in G$ und $x \in \mathbb{Z}$ zu finden. Eine Variante besteht in der fortlaufenden Multiplikation des erzeugenden Elementes g, bis x gefunden ist: $g,\ g^2,\ g^3,\ g^4,\ \ldots,\ g^x$. Die Laufzeit beträgt $O(|G|)$, weshalb m möglichst groß gewählt werden sollte, sodass die Suche zu aufwändig wird. Die Laufzeit verhält sich zwar linear zu $|G|$, ist allerdings exponentiell zur Größe der Zahlen in G.

Eine Abwandlung dieser naiven Methoden wurde 1971 von Daniel Shanks [Sha71] vorgestellt und ist bekannt als der **Baby-Step-Giant-Step-Algorithmus**. Der ausschlaggebende Grundgedanke ist die Erzeugung einer Liste mit $i\left\lceil \sqrt{|G|}\right\rceil g$ für $i = 0,\ 1,\ \ldots,\ \left\lceil \sqrt{|G|}\right\rceil$ und der anschließenden Suche des diskreten Logarithmus in dieser Liste. Dessen Laufzeit von $O(\sqrt{m})$ ist zwar kleiner als die der naiven Methoden, verbraucht jedoch zu viel Speicher für $m > 2^{160}$.

Einführend wird im Kapitel 3.3 die **Pollard ρ-Methode** zur Faktorisierung betrachtet, die John Michael Pollard im Jahre 1975 als ein heuristisches Verfahren [Pol75] zur Primfaktorzerlegung veröffentlicht hat. Darauf folgende Kapitel führen die Pollard ρ-Methode für diskrete Logarithmen [Pol78] ein.

S. C. Pohlig und M. E. Hellmann veröffentlichten 1978 einen Algorithmus [PH78], der das Problem des diskreten Logarithmus in G derart in Teilprobleme zerlegt, so dass Gruppen mit primer Ordnung p betrachtet werden, welche die Ordnung von G teilen.

Auf die kleineren Gruppen wird beispielsweise die Pollard ρ-Methode angewendet und ergibt auf diese Weise eine kürzere Laufzeit. Die Teilergebnisse werden mittels des Chinesischen Restsatzes 2.14 auf G zurückgeführt, wodurch sich der diskrete Logarithmus in G ergibt.

3.2 Floyds Algorithmus zum Zyklenfinden

Robert W. Floyd stellte 1967 ein Algorithmus [Flo67] vor, der Schleifen in gerichteten Graphen findet. Der Algorithmus ist auch bekannt als Hase-Igel-Algorithmus. Die Grundidee Algorithmus von Floyd ist die Verwendung von zwei gleichzeitigen Irrfahrten durch einen gerichteten Graphen: die eine wird Igel und die andere, welche die doppelte Anzahl von Schritten durchführt, wird Hase genannt.

Definition 3.2 Seien $n \in \mathbb{N}$ und $f : G \to G$ eine beliebige Funktion, die eine endliche Folge (x_i) durch $x_{i+1} = f(x_i)$ für $i = 0,\ 1,\ 2,\ \ldots,\ n$ mit einem Startwert x_0 erzeugt. Die so erzeugte Folge (x_i) wird **Irrfahrt** genannt. Eine solche Funktion f wird als **Iterationsfunktion** bezeichnet.

Knuth wendete Floyds Algorithmus auf Restklassengruppen modulo m unter Verwendung von einer Iterationsfunktion f an und hat gezeigt, dass die durch f erzeugte Irrfahrt kollidiert [Knu81, S. 517].

Satz 3.3 *Sei f eine Iterationsfunktion und (x_i) die durch f erzeugte Irrfahrt. So existieren ein eindeutiges $i \in \mathbb{N}$ und $\lambda,\ \mu \in \mathbb{N}$, so dass gilt $x_i = x_{2i}$ mit $\mu \leq i \leq \mu + \lambda$.*

Beweis: Es existieren $\lambda,\ \mu \in \mathbb{N}$, so dass die Glieder $x_0,\ x_1,\ \ldots,\ x_\mu,\ \ldots,\ x_{\mu+\lambda-1}$ der Irrfahrt unterschiedlich sind, aber $x_{\mu+\lambda} = x_\mu$. Dabei ist λ die Periodenlänge eines Zyklus und μ die Vorperiode eines Zyklus. Bewegt sich die Irrfahrt im Zyklus gilt für alle $k \in \mathbb{N}$, dass $x_{\mu+k\lambda} = x_\mu$. Damit folgt, dass für alle $j \in \mathbb{N},\ j \geq \mu$ gilt $x_{j+k\lambda} = x_j$. Seien nun $i,\ d \in \mathbb{N}$ und $i \geq \mu$ so gewählt, dass $\lambda \mid i$, etwa $i = d\lambda$, dann gilt:

$$x_{2i} = x_{i+i} = x_{i+d\lambda} = x_i. \tag{3.1}$$

Damit ist ein passendes i und das Kollisionsglied x_i gefunden. □

Algorithmus 3.1: Floyds Algorithmus

Input : Funktion f und Startwert x_0
Output : Treffpunkt x_i mit Index i
begin
    $(x_i,\ x_j,\ i) \leftarrow (f(x_0),\ f(f(x_0)),\ 1)$; `// (Igel, Hase, Index)`
    while $x_i \neq x_j$ **do** $(x_i,\ x_j,\ i) \leftarrow (f(x_i),\ f(f(x_j)),\ i+1)$;
    return $(x_i,\ i)$
end

Treffen sich beide Zufallsbewegungen Igel und Hase, ist eine Kollision gefunden und damit ein Zyklus. Dabei ist $x_i = x_j = x_{2i}$ für $i,\ j \in \mathbb{N},\ i = 0,\ 1,\ 2,\ \ldots$ und $j = 0,\ 1,\ 2,\ \ldots$. Die Laufzeit des Algorithmus 3.1 von Floyd beträgt $O(\mu + \lambda)$.

Eine mögliche Erweiterung des Algorithmus 3.1 umfasst die Berechnung von λ und μ. Dabei wird ausgegangen von x_i und x_j, nachdem eine Kollision eingetreten ist. In diesem Fall ist der Anfang der Schleife gleich weit entfernt von der aktuellen Position des Igels x_i und dem Startwert x_0 [Knu81, S. 517: 6.(c)].

Die Vorperiode μ und die Periodenlänge λ eines Zyklus lassen sich demnach durch den wie folgt erweiterten Algorithmus 3.2 auf Basis Floyds Algorithmus 3.1 bestimmen.

Algorithmus 3.2: Erweiterter Floyds Algorithmus

Input : Funktion f und Startwert x_0
Output : Treffpunkt x_i mit Index i, Vorperiode μ und Periodenlänge λ
begin
  $(x_i,\ x_j,\ i) \leftarrow (f(x_0),\ f(f(x_0)),\ 1)$; `// (Igel, Hase, Index)`
  while $x_i \neq x_j$ **do** $(x_i,\ x_j,\ i) \leftarrow (f(x_i),\ f(f(x_j)),\ i+1)$;
  `// Bestimmung von` μ
  $\mu \leftarrow 0;\ x_i = 0$;
  while $x_i \neq x_j$ **do** $(x_i,\ x_j,\ \mu) \leftarrow (f(x_i),\ f(x_j),\ \mu+1)$;
  `// Bestimmung von` λ
  $\lambda \leftarrow 1;\ x_i = f(x_j)$;
  while $x_i \neq x_j$ **do** $(x_i,\ \lambda) \leftarrow (f(x_i),\ \lambda+1)$;
  return $(x_i,\ i,\ \mu,\ \lambda)$
end

3.3 Pollards ρ-Methode zur Faktorisierung

Pollard veröffentlichte [Pol75] eine Methode zur Faktorisierung einer Zahl $m \in \mathbb{N}$, die auf Floyds Algorithmus 3.1 basiert.

Definition 3.4 Die von Pollard vorgeschlagene Iterationsfunktion f für die ρ-Methode zur Faktorisierung sei wie folgt definiert:

$$f : \mathbb{Z}_m \to \mathbb{Z}_m,\ x \mapsto (x^2 + c) \quad \text{mit} \quad c \in \mathbb{Z}_m \setminus \{0, 2\} \tag{3.2}$$

Pollard modifizierte Floyds Algorithmus durch eine andere Vergleichsmethode: der Prüfung der Teilbarkeit durch $\gcd(x_i - x_j,\ m)$ und durch eine zufällige Wahl von x_0 und c. Auf diese Weise lassen sich schrittweise alle Teiler der Zahl m finden. Die Methode ist in Algorithmus 3.3 dargestellt und gibt `None` zurück, falls kein Teiler gefunden werden konnte. In diesem Fall ist entweder p prim oder es muss mit anderen

Startwerten c und x_0 neu gestartet werden.

Algorithmus 3.3: Pollards ρ-Methode zur Faktorisierung

Input : Zu faktorisierende Zahl m
Output : Primfaktor p
begin
  $x_0 \in_R \mathbb{Z}_{m-1}$; $c \in_R \mathbb{Z}_{m-3} \setminus \{0\}$; $p \leftarrow 1$;
  $(x_i,\ x_j) \leftarrow (x_0,\ x_0)$; `// (Igel, Hase)`
  $f \leftarrow$ lambda $x : (x^2 + c) \bmod m$;
  while $p = 1$ **do**
    $(x_i,\ x_j) \leftarrow (f(x_i),\ f(f(x_j)))$;
    $p \leftarrow \gcd(x_i - x_j,\ m)$;
  return p **if** $1 < p < m$ **else** *None*
end

Dabei wird die Anzahl der Schritte bis zum Auffinden einer Kollisionen durch den Satz 3.5 beschrieben. Die Laufzeit des Algorithmus 3.3 beträgt damit $O(\sqrt{|Z_m|})$.

Satz 3.5 *Seien f gegeben durch Gleichung (3.2) und X die Anzahl der Iterationen bis das erste doppelte Glied der Irrfahrt (x_i), die durch f erzeugt wird, gefunden wurde. Dann gilt $E(X) \approx \sqrt{\frac{\pi m}{2}}$.*

Beweis: [OW96, Kap. A] Seien $m,\ k \in \mathbb{N}$ mit $1 \leq k \leq m$ und X eine Zufallsvariable. Es werden k Elemente der Gruppe gewählt, so dass die Auswahlen unabhängig voneinander sind. Die Wahrscheinlichkeit $Pr(X > k)$, dass nach k-maligem Ziehen mit Zurücklegen kein Element doppelt gezogen wird, für ein großes m und ein k sehr viel kleiner als m ist

$$Pr(X > k) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{m}\right) \approx e^{-\frac{k(k-1)}{2m}} \tag{3.3}$$

Für den Erwartungswert $E(X)$ gilt laut [OW96, Kap. A]

$$E(X) = \sum_{k=1}^{\infty} k \cdot Pr(X = k) = \sum_{k=1}^{\infty} k \cdot (Pr(X > k-1) - Pr(X > k)) = \sum_{k=1}^{\infty} Pr(X > k) \tag{3.4}$$

Eine Näherung für $E(X)$ ist wie folgt. Der dabei entstehende Fehler ist maximal 1, da die Funktion monoton fallend ist und 1 niemals übersteigt.

$$E(X) \approx \sum_{k=0}^{\infty} e^{-\frac{k(k-1)}{2m}} \approx \int_0^{\infty} e^{-\frac{x(x-1)}{2m}} dx = \sqrt{\frac{\pi m}{2}} \tag{3.5}$$

□

Beispiel 3.6 Die Abbildung 3.1 stellt die Ausführung des Algorithmus 3.3 dar. Dabei wird die Zahl 253 faktorisiert mit den Startwerten $x_0 = 4$ und $c = 17$.

i, j	x_i	x_j	p
0	4	4	1
1	33	94	1
2	94	21	1
3	251	44	23
4	21	**251**	23
5	205	205	253
6	44	182	23
7	182	21	23
8	**251**	44	23

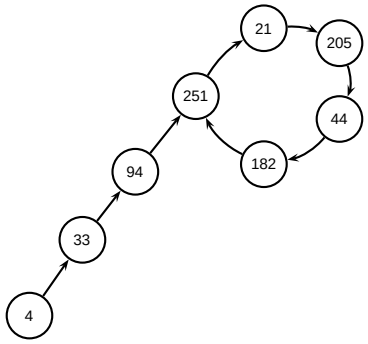


Abbildung 3.1: Faktorisierung mit dem Pollard ρ-Algorithmus

Zur Verdeutlichung der ρ-Form wurde der Algorithmus fortgeführt, denn dieser hätte bereits im Durchlauf $i = 3$ gestoppt und den Primfaktor 23 gefunden. Demnach ergeben sich die Primfaktoren $253 = 23 \cdot 11$. Die Tabellen 3.1 und 3.2 stellen die Durchläufe des Algorithmus mit entsprechenden Parametern dar und bestätigen die Primfaktoren 11 und 23.

i, j	x_i	x_j	p
0	14	14	1
1	10	6	1
2	6	4	1
3	11	10	1
4	4	11	1
5	14	14	23

$m = 23$, $x_0 = 14$ und $c = 21$

i, j	x_i	x_j	p
0	22	22	1
1	14	2	1
2	2	3	1
3	17	14	1
4	3	17	1
5	22	22	23

$m = 23$, $x_0 = 22$ und $c = 13$

Tabelle 3.1: Test des Primfaktors 23 mit dem Pollard ρ-Algorithmus

i, j	x_i	x_j	p
0	7	7	1
1	8	1	1
2	1	8	1
3	4	4	11

$m = 11$, $x_0 = 7$ und $c = 3$

i, j	x_i	x_j	p
0	5	5	1
1	10	8	1
2	8	10	1
3	5	5	11

$m = 11$, $x_0 = 5$ und $c = 7$

Tabelle 3.2: Test des Primfaktors 11 mit dem Pollard ρ-Algorithmus

Aufbauend auf den Betrachtungen der Pollard ρ-Methode zur Faktorisierung wird im Kapitel 3.4 die von Pollard veröffentlichte ρ-Methode eingeführt, die diskrete Logarithmen berechnet.

3.4 Pollards ρ-Methode für diskrete Logarithmen

Im Juli 1978 hat Pollard ein verbessertes Verfahren [Pol78] vorgeschlagen, welches auf Shanks Idee, der Zerlegung von $G = \mathbb{Z}_m^\times$, und Floyds Algorithmus 3.1 basiert. Pollards verallgemeinerte Methode ist auf jede abelsche Gruppe anwendbar.

Definition 3.7 Sei $m \in \mathbb{N}$ prim, g eine Primitivwurzel modulo m, $y \in G$ mit $G = \langle g \rangle = \mathbb{Z}_m^\times$ und $x_0 = 1$. Die von Pollard vorgeschlagene Funktion f für die ρ-Methode für diskrete Logarithmen sei wie folgt definiert:

$$f : G \to G,\ x \mapsto \begin{cases} yx, & \text{falls } 0 \le x < \frac{1}{3}m \\ x^2, & \text{falls } \frac{1}{3}m \le x < \frac{2}{3}m \\ gx, & \text{falls } \frac{2}{3}m \le x < m \end{cases} \tag{3.6}$$

Satz 3.8 *Seien $m \in \mathbb{N}$, f die Iterationsfunktion (3.6) und (x_i) die durch f erzeugte Irrfahrt. Ist eine Kollision von $x_i,\ x_j \in \mathbb{Z}_m$ mit $x_i = x_j = x_{2i}$ laut Satz 3.3 gefunden, dann existieren $s,\ t \in \mathbb{Z}_{m-1}$, so dass die Lösungsmenge der lineare Kongruenz $sx \equiv t \pmod{m-1}$ den diskreten Logarithmus x enthält.*

Beweis: Sei f die Iterationsfunktion (3.6) und (x_i) die durch f erzeugte Irrfahrt. Dann ergeben sich die Glieder x_i der Irrfahrt durch $x_i = y^{a_i} g^{b_i}$ mit $a_i,\ b_i \in \mathbb{Z}_{m-1}\ \forall i = 0, 1, 2, \ldots, |G|$. Sind $x_i,\ x_j \in \mathbb{Z}_m$ mit $x_i = x_{2i} = x_j$ gefunden, die laut Satz 3.3 kollidieren, dann gilt:

$$x_i = x_j \ \Rightarrow\ y^{a_i} g^{b_i} = y^{a_j} g^{b_j} \ \Leftrightarrow\ y^{a_i - a_j} = g^{b_j - b_i} \tag{3.7}$$

Laut der Definition 2.6 des diskreten Logarithmus gilt $y = g^x$, damit folgt:

$$y^{a_i - a_j} = g^{(a_i - a_j)x} = g^{b_j - b_i} \ \Rightarrow\ sx \equiv t \pmod{m-1} \ \text{ mit } \ s = a_i - a_j,\ t = b_j - b_i \tag{3.8}$$

Die lineare Kongruenz $sx \equiv t \pmod{m-1}$ ergibt laut Satz 2.13 eine Lösungsmenge X, in der ein x zu finden ist, für das $y = g^x$ gilt. Damit ist ein diskreter Logarithmus x gefunden. □

Folgerung 3.9 *Aus Satz 3.8 und der Definition 3.7 der Funktion f folgen Funktionen α und β, welche die Exponentenfolgen (a_i) und (b_i) erzeugen mit $a_0 = 0$ und $b_0 = 0$:*

$$\alpha : G \times \mathbb{Z}_{m-1} \to \mathbb{Z}_{m-1},\ (x,\ a) \mapsto \begin{cases} a+1, & \textit{falls } 0 \le x < \frac{1}{3}m \\ 2a, & \textit{falls } \frac{1}{3}m \le x < \frac{2}{3}m \\ a, & \textit{falls } \frac{2}{3}m \le x < m \end{cases} \tag{3.9}$$

$$\beta : G \times \mathbb{Z}_{m-1} \to \mathbb{Z}_{m-1},\ (x,\ b) \mapsto \begin{cases} b, & \textit{falls } 0 \le x < \frac{1}{3}m \\ 2b, & \textit{falls } \frac{1}{3}m \le x < \frac{2}{3}m \\ b+1, & \textit{falls } \frac{2}{3}m \le x < m \end{cases} \tag{3.10}$$

Mit den bisherigen Modifikationen Floyds Algorithmus 3.1 lassen sich zwei Tripel

$(x_i,\ a_i,\ b_i)$ und $(x_j,\ a_j,\ b_j)$ finden, so dass gilt $x_i = x_j$. Laut Satz 3.8 ergibt sich somit der gesuchte diskrete Logarithmus x. Alle bisherigen Betrachtungen der Pollard ρ-Methode für diskrete Logarithmen sind in Algorithmus 3.4 zusammengefasst.

Algorithmus 3.4: Pollards ρ-Algorithmus für diskrete Logarithmen

Input : g, y und m
Output : Exponent x
begin
  // Schritt 1 - Finden einer Kollision
  $i \leftarrow 0$;
  $(x_i,\ a_i,\ b_i) \leftarrow (1,\ 0,\ 0)$; // Igel
  $(x_j,\ a_j,\ b_j) \leftarrow (1,\ 0,\ 0)$; // Hase
  // Funktionen f, α und β siehe Gleichungen (3.6), (3.9) und (3.10)
  while $i \neq 0$ *and* $x_i \neq x_j$ **do**
    $(x_i,\ a_i,\ b_i) \leftarrow (f(x_i),\ \alpha(x_i,\ a_i),\ \beta(x_i,\ b_i))$
    $(x_j,\ a_j,\ b_j) \leftarrow (f(f(x_j)),\ \alpha(f(x_j),\ \alpha(x_j,\ a_j)),\ \beta(f(x_j),\ \beta(x_j,\ b_j)))$;
    $i \leftarrow i+1$;
  if $b_i = b_j$ **then return** None;
  $(s,\ t) \leftarrow (a_i - a_j,\ b_j - b_i)$;

  // Schritt 2 - Finden von x
  $X \leftarrow$LCE$(s,\ t,\ m-1)$;
  for $x \in X$ **do**
    $y_t \leftarrow g^x$;
    if $y_t = y$ **then return** x
end

Dabei sei LCE$(s,\ t,\ m)$ ein Algorithmus, welcher zur linearen Kongruenz der gegebenen Parameter eine Lösungsmenge liefert, eine Sage-Implementierung von LCE$(s,\ t,\ m)$ stellt Listing A.1 dar. Findet der Algorithmus zum diskreten Logarithmus $x \in \mathbb{Z}_m$ mit $x = \log_g y$ keine Lösung wird None zurückgegeben, sonst x. Die Eingabe des Algorithmus besteht aus allen bekannten Werten $g,\ y \in \mathbb{Z}_m$ und $m \in \mathbb{N}$. Eine Sage-Implementierung des Algorithmus 3.4 ist in Listing A.2 zu finden.

Beispiel 3.10 Es ist der diskrete Logarithmus $x = \log_g(y)$ zu lösen. Dabei sei $G = \mathbb{Z}_{53}^{\times} = \langle 2 \rangle$ mit Ordnung $|\mathbb{Z}_{53}| = 52$ und $y = 33$.

1. Für die zwei Irrfahrten ergeben sich folgende Tripel:

i, j	0	1	2	3	4	5	6	7	8	9	**10**
x_i	1	33	29	46	39	25	42	31	7	19	**43**
a_i	0	1	2	4	4	4	8	8	16	17	34
b_i	0	0	0	0	1	2	4	5	10	10	20
x_j	1	29	39	42	7	43	29	39	42	7	**43**
a_j	0	2	4	8	16	34	16	32	12	24	50
b_j	0	0	1	4	10	20	42	33	16	34	16

   Mit dem Index $i = 10$ wurde eine Kollision gefunden. Es ergibt sich die Kongru-

enz $36x \equiv 48 \pmod{52}$, denn $t \equiv a_i - a_j \equiv 34 - 50 \equiv 48 \pmod{52}$ und $s \equiv b_j - b_i \equiv 16 - 20 \equiv 36 \pmod{52}$.

2. Zu lösen ist die Kongruenz $36x \equiv 48 \pmod{52}$. Da $\gcd(36, 52) = 4$ existieren vier Lösungen $X = \{10, 23, 36, 49\}$. Es ist nun zu prüfen, welches $x \in X$ Lösung von $y = g^x$ ist. Dazu wird die Menge X durchsucht, bis die Lösung x gefunden wurde:

$$2^{10} = 17 \qquad \mathbf{2^{23} = 33} \qquad 2^{36} = 36 \qquad 2^{49} = 20$$

Aus $y = g^x = 2^{23} = 33$ folgt, dass $\boldsymbol{x = 23}$ der gesuchte diskrete Logarithmus ist. Eine graphische Darstellung der beiden Irrfahrten zeigt wie in Abbildung 3.2 die ρ-Struktur, dabei ist die Irrfahrt (x_i) aus diesem Beispiel **rot** markiert.

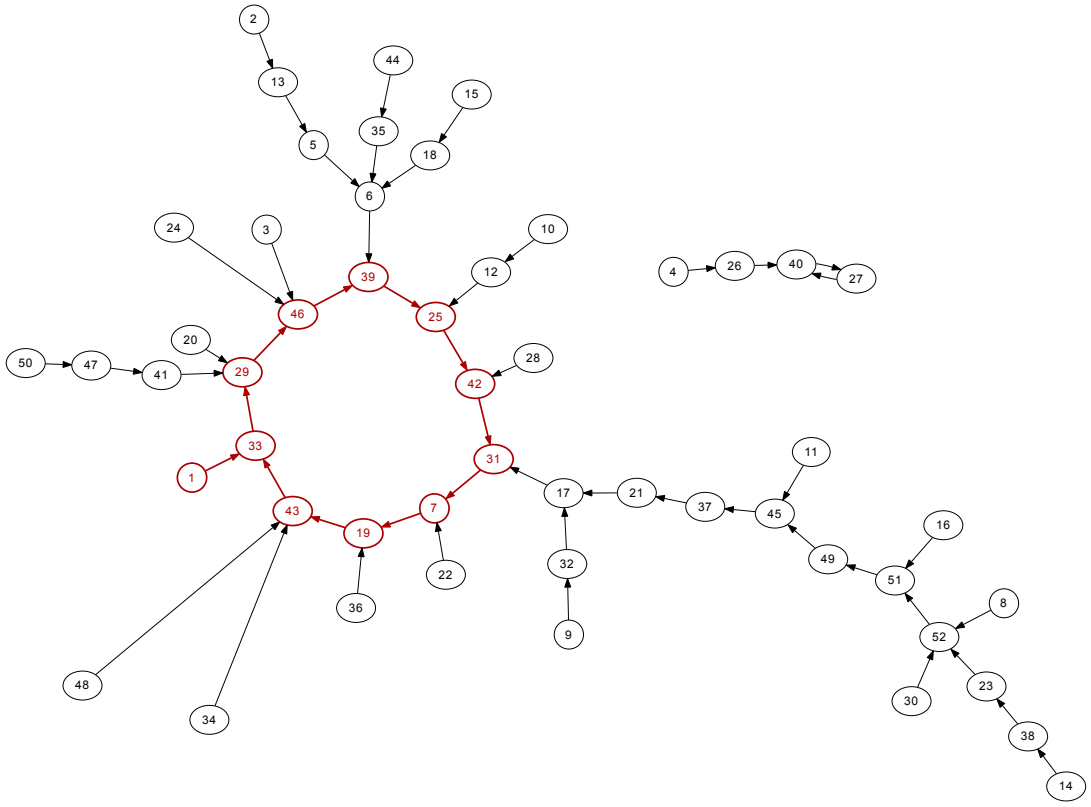


Abbildung 3.2: Durchläufe der Pollard ρ-Methode im Beispiel 3.10 mit allen Startpunkten

Die Abbildung 3.2 wurde mit dem Programm `neato` des Pakets GraphViz erstellt, wobei die Beschreibungsstruktur durch Listing A.3 als Teil eines Sage-Notebooks generiert wurde.

Die Python-Klasse des Listings A.3 ist dabei derart allgemein implementiert, so dass jede mögliche Beschreibungsstruktur für die Pollard ρ-Methode erstellt werden kann. Die Beschreibungsstruktur ist dabei abhängig den Parametern g, y und m.

3.5 Verallgemeinerung der Pollard ρ-Methode für diskrete Logarithmen

Die Pollard ρ-Methoden für Gruppen über elliptische Kurven und für abelsche Gruppen folgen aus den Betrachtungen des Kapitels 3.4. Für eine abelsche Gruppe werden folgende Bedingungen in Definition 3.11 [Tes00, S. 810] angenommen.

Definition 3.11 Zur Anwendung einer Kollisionssuche wie die Pollard ρ-Methode werden von einer abelschen Gruppe folgende Bedingungen für zwei Gruppenelemente g und h oder für G angenommen:

1. Es lässt sich das Produkt (oder die Summe) von $g \times h$ berechnen.
2. Es lässt sich prüfen, ob $g = h$.
3. Sei $r \in \mathbb{N}$ klein. Es gibt eine Methode zur Partitionierung von G in paarweise disjunkte $G_1, G_2, \ldots, G_r$ mit annähernd gleicher Mächtigkeit.

Folgerung 3.12 *Für Gruppen über elliptische Kurven ist für die Gruppe G die Gruppe $G_E = E(\mathbb{K}_m)$ zu wählen. Die Gruppe G_E ist dabei die in Satz 2.32 betrachte Gruppe bezüglich der Addition über einer elliptischen Kurve zusammen mit der Punktaddition. Laut der Definition 2.6 des diskreten Logarithmus gilt damit $y = xg$.*

Definition 3.13 Seien $V \in G$, $G_V = \langle V \rangle$ eine abelsche Gruppe einer elliptischen Kurve und $Y \in G_V$. Sei $(G1, G2, G3)$ eine Partition von G_V und $X_0 = a_0 V$ mit $a_0 \in_R \mathbb{Z}_{m-1}$. Die verallgemeinerte Iterationsfunktion f von Pollard für die ρ-Methode für diskrete Logarithmen sei wie folgt definiert:

$$f : G_V \to G_V,\ X \mapsto \begin{cases} Y + X, & \text{falls } X \in G_1 \\ 2X, & \text{falls } X \in G_2 \\ V + X, & \text{falls } X \in G_3 \end{cases} \tag{3.11}$$

Die beiden Funktionen α und β aus den Gleichungen (3.9) und (3.10) werden auf dieselbe Weise verallgemeinert, wie die Iterationsfunktion f aus Definition 3.13.

Für die Betrachtung des diskreten Logarithmus über elliptischen Kurven gilt nun zusammenfassend folgender Satz, welcher aus den bisherigen Betrachtungen folgt.

Satz 3.14 *Seien $P, Q \in G_V$ Punkte der abelschen Gruppe G_V einer elliptischen Kurve E. Dann gibt es zur Lösung des diskreten Logarithmus x mit $Q = xP$ zwei verschiedene Paare $(a_i, b_i), (a_j, b_j) \in \mathbb{Z}_{m-1} \times \mathbb{Z}_{m-1}$, sodass gilt:*

$$\begin{aligned} a_i P + b_i Q &= a_j P + b_j Q \\ (a_i - a_j)P &= (b_j - b_i)Q \\ (b_j - b_i)x &\equiv (a_i - a_j) \pmod{m-1} \end{aligned} \tag{3.12}$$

4 Additive und gemischte Irrfahrten

4.1 Analyse von Pollards Iterationsfunktion

In den wissenschaftlichen Arbeiten [Tes98] und [Tes00] hat Edlyn Teske die Struktur und statistischen Eigenschaften der Iterationsfunktionen von Pollard veröffentlicht. Dabei hat er festgestellt, dass sich die Erwartungswerte der experimentellen Anwendungen von Pollards Iterationsfunktionen (3.6) und (3.11) nicht zufällig genug verhalten.

Teske hat eine andere Methode zum Zyklen-Finden gewählt, die auf Schnorr und Lenstras Algorithmus [SL84] basiert. Diese Methode verwendet nur eine Irrfahrt, um eine geeignete Kollision laut Satz 3.3 zu finden. Der notwendige Speicherbedarf wird dabei auf 8 Speicherplätze beschränkt und wird wie folgt verwaltet [Tes98, S. 545]:

- Es werden 8 Tripel $(y_{\sigma_d},\ a_{\sigma_d},\ b_{\sigma_d})$ mit $d = 1,\ \ldots,\ 8$ gespeichert.
- Die Initialisierung sei $\sigma_d = 0\ \forall d$ und damit $(y_0,\ a_0,\ b_0)$ in jeder Zelle.
- Nach jeder Berechnung von $(y_i,\ a_i,\ b_i)$ wird geprüft, ob $\exists d$ mit $y_i = y_{\sigma_d}$.
  - Ist ein y_{σ_d} gefunden, wird die entsprechende lineare Kongruenz gelöst.
  - Wenn $i > 3\sigma_1$, dann verschiebe alle Zellen nach links, sodass $\sigma_d = \sigma_{d+1}\ \forall d = 1,\ \ldots,\ 7$, setze $\sigma_8 = i$ und speichere $(y_i,\ a_i,\ b_i)$ in der letzten Zelle.

Eine Sage-Implementierung ist im Listing A.5 zu finden. Auf Basis dieser Methode hat Teske Experimente [Tes98, S. 549] und [Tes00, S. 813ff.] durchgeführt und ein Vergleichsmerkmal definiert.

Definition 4.1 Sei $E(l(\lambda,\mu))$ der Erwartungswert der Anzahl der Schritte bis eine Kollision gefunden wurde. Das Verhältnis $\frac{E(l(\lambda,\mu))}{\lambda+\mu}$ wird als der **erwartete Verzögerungsfaktor** δ bezeichnet.

Teskes Experimente mit obiger Methode zum Zyklen-Finden ergaben $\delta \approx 1.13$. Dabei stellte er fest, dass für eine zufällige Irrfahrt eine Kollision nach

$$1.13\sqrt{\frac{\pi|G|}{2}} \approx 1.416\sqrt{|G|} = L_0\sqrt{|G|} \tag{4.1}$$

Schritten eintritt.

Definition 4.2 Im Fall einer zufälligen Irrfahrt bezeichne $\boldsymbol{L_0\sqrt{|G|}}$ das arithmetische Mittel der Schritte bis eine Kollision gefunden wurde. Zum Vergleich einer Irrfahrt mit

der zufälligen Irrfahrt bezeichne L das folgende Verhältnis

$$L := \frac{l_\Delta(\lambda,\mu)}{\sqrt{|G|}} \tag{4.2}$$

Dabei sei $l_\Delta(\lambda,\mu)$ das arithmetische Mittel der Schrittanzahl bis eine Kollision gefunden wurde.

Mit diesen Vorbetrachtungen hat Teske Experimente für die Iterationsfunktionen f der Definitionen 3.7 und 3.13 durchgeführt und kam zu folgender Erkenntnis.

Beobachtung 4.3 *Sei $G = \langle g \rangle$ mit $g \in G$. Ist g eine Primitivwurzel modulo m, so wird im Durchschnitt ein kleinstmögliches L unter allen erzeugenden Elementen g erreicht.*

Die folgende Abbildung 4.1 fasst Teskes experimentelle Untersuchungen zusammen und verdeutlich den Inhalt der Beobachtung 4.3. Dabei wurden für Irrfahrten mit primitiven Elementen und nicht primitiven Elementen 3·55980 Diskrete-Logarithmen-Berechnungen durchgeführt.

Die **blaue** Irrfahrt stellt die originale Iterationsfunktion von Pollard mit primitiven Generatoren dar, die **rote** die verallgemeinerte Iterationsfunktion über Untergruppen primer Ordnung und die **grüne** die verallgemeinerte Iterationsfunktion über Untergruppen über elliptischer Kurven mit primer Ordnung. Verglichen werden dabei die (Unter-)Gruppenordnung zwischen 10^{n-1} und 10^n für $3 \leq n \leq 13$ in Bezug zum durchschnittlichen L aus Gleichung (4.2).

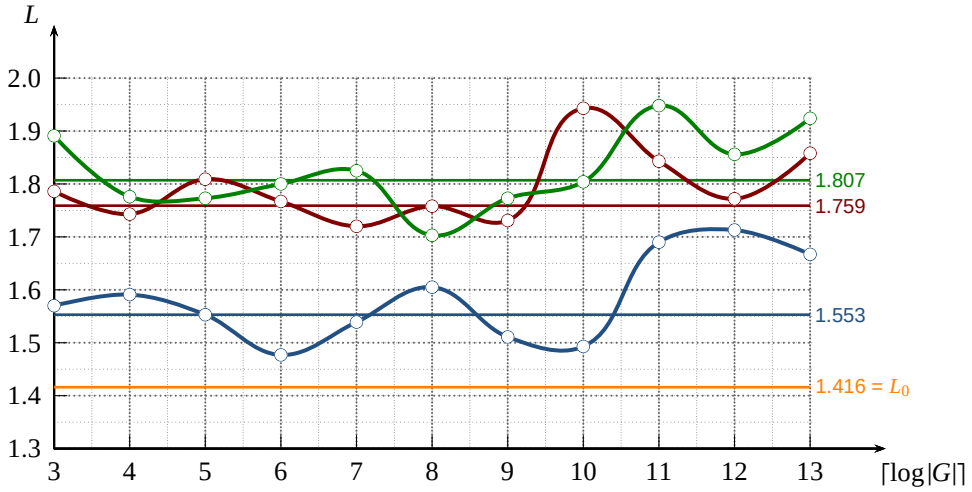


Abbildung 4.1: Experimenteller Vergleich der Generatoren für Pollards Iterationsfunktion

Im Vergleich zur zufälligen Irrfahrt, deren durchschnittliches L_0 in Abbildung 4.1 **orange** dargestellt ist, ist Pollards originale Irrfahrt näher an der zufälligen Irrfahrt, im Gegensatz zu Irrfahrten über Untergruppen primer Ordnung, welche sogar eine 1.25-fach schlechtere Laufzeit besitzen.

Der Unterschied beider Typen von Irrfahrten lässt sich auch durch die von den zugrundeliegenden Iterationsfunktionen beschriebene Struktur verdeutlichen. Dazu seien folgende Beispiele 4.4 und 4.5 gegeben mit den entsprechenden Abbildungen für die Teilfunktionen f_1, f_2 und f_3 der Iterationsfunktionen f aus den Definitionen 3.7 und 3.13.

Beispiel 4.4 Sei $G = (\mathbb{Z}/11\mathbb{Z})^\times = \langle g \rangle$ mit $g = 2$ und $y = 5$. Die Abbildung 4.2 zeigt die Struktur der erzeugten Folge der Iterationsfunktion aus Definition 3.7.

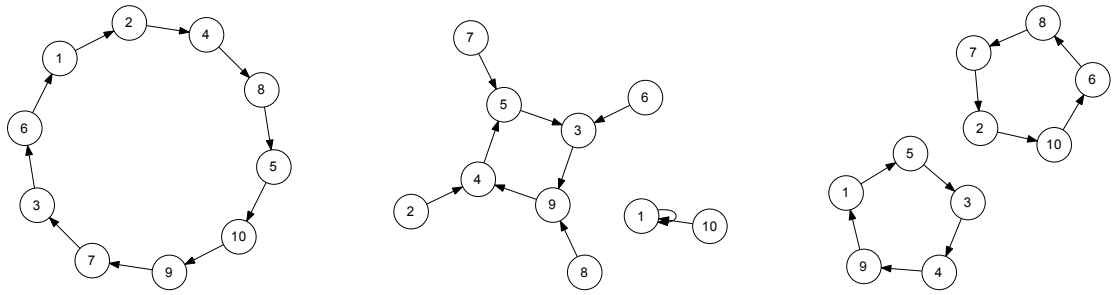


Abbildung 4.2: Struktur der erzeugten Folge der Iterationsfunktion aus Definition 3.7.

Beispiel 4.5 Sei $G = (\mathbb{Z}/23\mathbb{Z})^\times = \langle g \rangle$ mit $g = 2$ und $y = 8$, wobei $|G| = 11$. Die Abbildung 4.3 zeigt die Struktur der erzeugten Folge der Iterationsfunktion aus Definition 3.13.

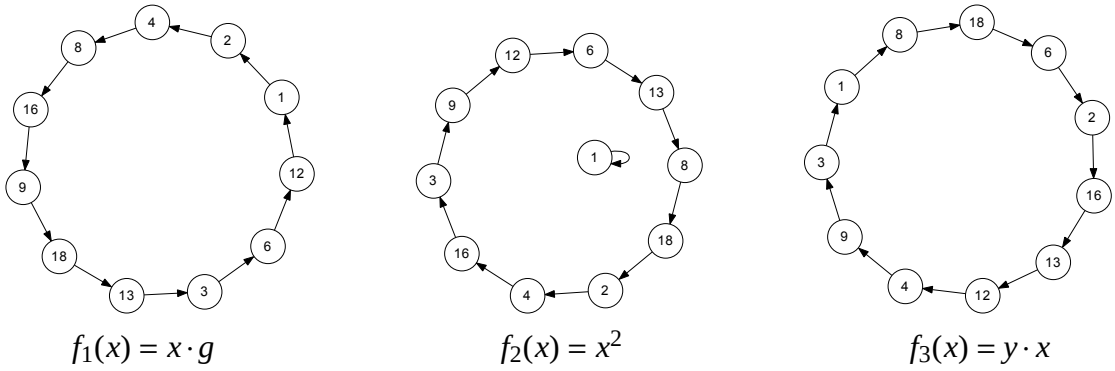


Abbildung 4.3: Struktur der erzeugten Folge der Iterationsfunktion aus Definition 3.13.

Beide Abbildungen 4.2 und 4.3 wurden mit dem Programm `neato` erzeugt. Die dafür notwendige Beschreibungsstruktur wurde durch die Python-Klasse des Listings A.6 generiert.

Der entscheidende Unterschied ist erkennbar in der Darstellung der Funktion f_2 in beiden Abbildungen 4.2 und 4.3. Dabei ist f_2 der Abbildung 4.2 nicht-bijektiv im Gegensatz zu f_2 der Abbildung 4.3, die bijektiv ist. Dieser Unterschied wirkt sich auf die Laufzeit und die durchschnittlichen L der beiden Irrfahrten aus.

Nach einer Analyse der Iterationsfunktionen f sind ihm zwei wesentliche Merkmale aufgefallen: In zwei Fällen die Multiplikation mit einer Konstanten und in einem Fall die Multiplikation mit der Iterationsvariable x selbst. Es folgen die Erkenntnisse von Teske, die im Wesentlichen auf der Anzahl der Partitionen und der Art der Irrfahrten basieren, um eine Beschleunigung der Pollard ρ-Methode zu erreichen.

4.2 Teskes Irrfahrten

Teske schlug neue Irrfahrten [Tes98] vor, welche die Gruppe derart aufteilen und darauf Teilfunktionen f_i anwenden, so dass die resultierende Irrfahrt näher an der zufälligen Irrfahrt liegt als Pollards Irrfahrt. Zur Partitionierung der Gruppe G in r paarweise disjunkte Blöcke $G_1, G_2, \ldots, G_r$ hat Teske eine Hashfunktion h vorgeschlagen, so dass eine Iterationsfunktion f mit r verschiedenen Regeln entsteht.

Definition 4.6 Sei $r \in \mathbb{N}$ mit kleinem r und $h : G \to \{0, \ldots, r-1\}$ eine Hashfunktion, so dass die h-induzierte r-Partition $\langle g \rangle = \bigcup_{i=0}^{r-1} G_i$ mit $G_i = \{y : y \in \langle g \rangle,\ h(y) = i\}$ die Blöcke G_i ergibt, welche eine etwa gleichgroße Mächtigkeit aufweisen.

Bemerkung 4.7 Teske verwendete eine multiplikative Hashfunktion, die aus dem erzeugenden Element $g \in G$ einen gleichverteilten Zufallswert b berechnet, um einen Block G_b auszuwählen. “Für praktische Anwendung können auch einfacher zu berechnende Hashfunktionen verwendet werden” [Tes98, S. 813], beispielsweise $h : G \to \{0, \ldots, r-1\}$ mit $y \mapsto y \bmod r$.

4.2.1 Additive Irrfahrten

Teske hat sogenannte r-additive Irrfahrten [Tes98, S. 818] wie folgt definiert.

Definition 4.8 Seien $r \in \mathbb{N}$, $C_0, \ldots, C_{r-1} \in_R G$ und $h : G \to \{0, \ldots, r-1\}$ eine Hashfunktion. Eine Irrfahrt in einer endlichen abelschen Gruppe G wird ***r*-additiv** genannt, wenn für eine Iterationsfunktion $f : G \to G$, welche eine Irrfahrt (x_i) mit $i = 0, 1, 2, \ldots$ erzeugt, gilt:

$$x \mapsto x \cdot C_{h(x)}. \tag{4.3}$$

Die Bezeichnung r-additive Irrfahrten wird dabei durch den folgenden Satz 4.9 deutlich, in welchem die Exponentenfolgen (a_i) und (b_i) betrachtet werden.

Satz 4.9 *Sei eine additive Irrfahrt (x_i) mit $i = 0, 1, 2, \ldots$ erzeugt durch eine Iterationsfunktion f wie in Definition 4.8. Dann existieren Exponentenfolgen (a_i) und (b_i) der folgenden Form:*

$$a_{i+1} = a_i + c_{h(x_i)} \quad \textit{und} \quad b_{i+1} = b_i + d_{h(x_i)} \ \textit{mit}\ c_j, d_j \in_R \{1, \ldots, |G|\}. \tag{4.4}$$

Beweis: Seien $g,\ y \in G$ gegeben, wobei $x = \log_g y$ gesucht ist. Seien weiterhin C_j mit $j = h(x_i)$ für eine Hashfunktion h laut Definition 4.6 gegeben durch $C_j = g^{c_j} \cdot y^{d_j}$ mit $c_j,\ d_j \in_R \{1, \ldots, |G|\}$. Dann ergeben sich die Komponenten x_i der Irrfahrt (x_i) durch $x_i = g^{a_i} \cdot y^{b_i}$ mit den Startwerten $x_0 = g^{a_0}$, $a_0 \in_R \{1, \ldots, |G|\}$ und $b_0 = 0$. Es folgen zwei Exponentenfolgen (a_i) und (b_i) der Form:

$$a_{i+1} = a_i + c_j \quad \text{und} \quad b_{i+1} = b_i + d_j \ \text{ mit } \ j = h(x_i). \tag{4.5}$$

□

Nachdem Teske 11000 Berechnungen für verschiedene r durchführen ließ, zeigte sich die Leistung der jeweiligen r-additiven Irrfahrten. Die Experimente wurden in Gruppen über elliptischen Kurven mit primer Gruppenordnung durchgeführt.

Abbildung 4.4 fasst seine Ergebnisse [Tes98, S. 819] zusammen. Dabei sind vergleichsweise die durchschnittlichen L der vorherigen Irrfahrten gekennzeichnet und die additiven Irrfahrten durch die **purpurne** Kennline. Erkennbar ist dabei die deutliche Annäherung der r-additiven Irrfahrten an die zufällige Irrfahrt für ein $r > 17$. Mit größeren r steigt allerdings auch der Speicherbedarf für die Verwaltung zur Kollisionssuche laut Satz 4.9.

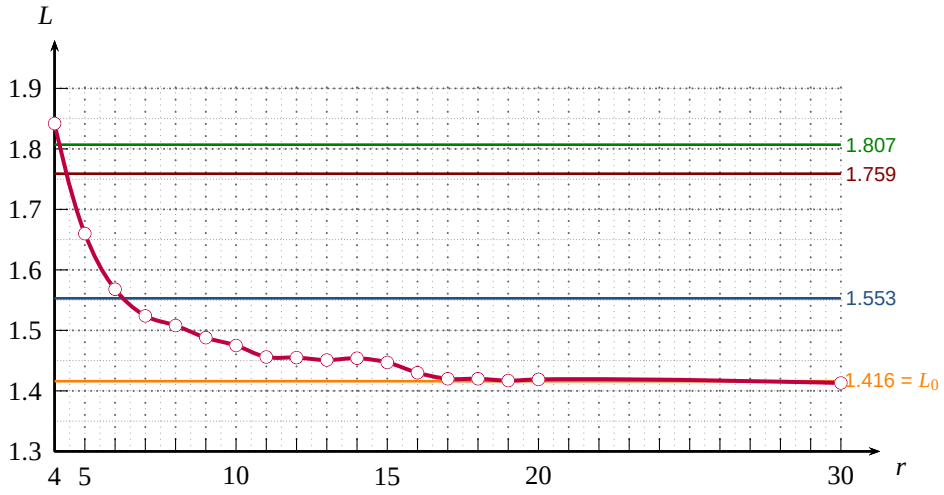


Abbildung 4.4: Experimenteller Vergleich der r-additiven Irrfahrten

Während der Experimente kam Teske zu folgenden Erkenntnissen.

Beobachtung 4.10 *[Tes98, S. 818ff.] Für $r \geq 4$ ist der durchschnittliche Wert für L unabhängig von der Gruppenordnung. Für $r = 3$ gibt es keinen konstanten durchschnittlichen Wert für L, welcher abhängig ist von der Gruppenordnung.*

Die Beobachtung 4.10 enthält erneut einen Hinweis darauf, weshalb Pollard für einen Fall seiner Iterationsfunktion eine Quadrierung der Iterationsvariablen gewählt hat, was die Leistung und Laufzeit deutlich verbessert. Aufbauend auf den additiven Irrfahrten hat Teske daher auch gemischte Irrfahrten betrachtet: Irrfahrten mit Additionen r und Quadrierungen s.

4.2.2 Gemischte Irrfahrten

Definition 4.11 Seien $r,\ s \in \mathbb{N}$, $C_0,\ \ldots,\ C_{r-1} \in_R G$ und $h : G \to \{0,\ \ldots,\ r-1+s\}$ eine Hashfunktion. Eine Irrfahrt in einer endlichen abelschen Gruppe G wird $\boldsymbol{r+s}$**-gemischt** genannt, wenn für eine Iterationsfunktion $f : G \to G$, welche eine Irrfahrt (x_i) mit $i = 0,\ 1,\ 2,\ \ldots$ erzeugt, gilt:

$$x \mapsto \begin{cases} x \cdot C_{h(x)} & , \text{falls } h(x) \in \{0,\ \ldots,\ r-1\} \\ x^2 & , \text{falls } h(x) \in \{r,\ \ldots,\ r-1+s\} \end{cases} . \tag{4.6}$$

Beispiel 4.12 Pollards Irrfahrt generiert durch die Iterationsfunktion aus Definition 3.7 ist eine $2+1$-gemischte Irrfahrt.

Schließlich hat Teske auch für gemischte Irrfahrten Experimente [Tes98, S. 820ff.] durchgeführt. Dabei wurden 200 verschiedene Durchgänge in Untergruppen über elliptischen Kurven mit primer Gruppenordnung berechnet. Abbildung 4.5 fasst seine Ergebnisse zusammen, wobei die x-Achse das Verhältnis von Verdopplung s zu Additionen r darstellt.

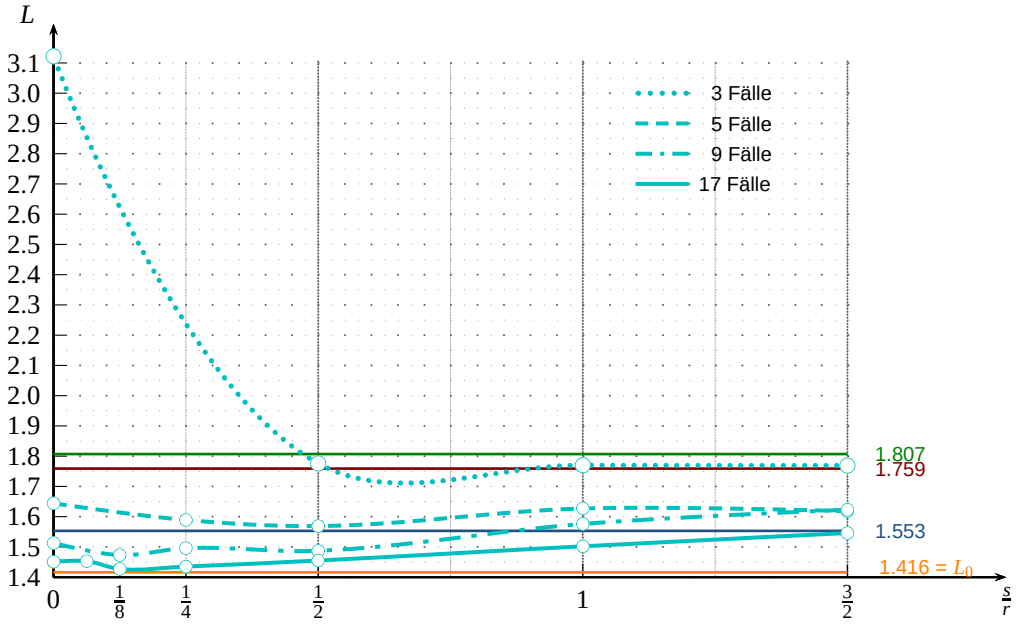


Abbildung 4.5: Experimenteller Vergleich der gemischten Irrfahrten

Aus der Abbildung 4.5 folgt das optimale Verhältnis für gemischte Irrfahrten: zwischen $\frac{1}{4}$ und $\frac{1}{2}$. Im Gegensatz dazu verschlechtert sich die Leistung und Laufzeit, wenn ein Verhältnis größer als 1 gewählt wird. Aus den Betrachtungen folgt weiterhin, dass aus der Verwendung von Verdopplungs-Fällen keine deutliche Verbesserung folgt.

4.3 Nachweisbarkeit guter Irrfahrten

Neben experimentellen Untersuchungen hat Teske seine Erkenntnisse nachgewiesen [Tes98, S. 822ff.]. In diesem Kapitel sollen die wichtigsten Erkenntnisse zur Beweisführung dargestellt werden.

Lemma 4.13 *Seien $m \in \mathbb{N}$ prim, $G = \langle g \rangle$ eine endliche zyklische Gruppe und f wie in Definition 4.8, für die der Satz 4.9 gilt. So gibt es eine bijektive Abbildung $\mathbb{Z}/m\mathbb{Z} \ni z \mapsto g^z \in G$, die r-additive Irrfahrten in Irrfahrten in Restklassen modulo m überführt.*

Beweis: Seien $g,\ y \in G$ gegeben, wobei $x = \log_g y$ gesucht ist. Der Startwert x_0 sei $x_0 = g^{a_0}$ mit $a_0 \in_R \{1, \ldots, |G|\}$. Seien weiterhin C_j mit $j = h(x_i)$ für eine Hashfunktion h laut Definition 4.6 gegeben durch $C_j = g^{c_j} \cdot y^{d_j}$ mit $c_j,\ d_j \in_R \{1, \ldots, |G|\}$ und die Iterationsfunktion f aus Definition 4.8 mit der rekursiven Form $x_{i+1} = x_i \cdot C_j$. Dann ergeben sich die Komponenten x_i der Irrfahrt (x_i) mit $b_0 = 0$ und $i = 0,\ 1,\ 2,\ \ldots$ durch

$$x_{i+1} = x_i \cdot C_j = x_i \cdot g^{c_j} \cdot y^{d_j}. \tag{4.7}$$

Mit $y = g^x$ folgt

$$x_{i+1} = x_i \cdot g^{c_j} \cdot y^{d_j} = x_i \cdot g^{c_j} \cdot (g^x)^{d_j} = x_i \cdot g^{t_j} \ \text{ mit } \ t_j = c_j + x d_j. \tag{4.8}$$

Die Exponentenfolgen (a_i) und (b_i) werden wie in Satz 4.9 erzeugt durch die Iterationsfunktionen

$$a_{i+1} = a_i + c_j \qquad \text{und} \qquad b_{i+1} = b_i + d_j. \tag{4.9}$$

Aus den Rekursionsgleichungen (4.8) und (4.9) folgt

$$x_{i+1} = x_i \cdot g^{z_j} \ \text{ mit } \ z_j = a_j + x b_j. \tag{4.10}$$

Es folgt also, dass sich jedes Glied der Irrfahrt (x_i) darstellen lässt als $x_i = g^{z_j}$, so dass für eine Irrfahrt (z_i) gilt

$$z_{i+1} = z_i + t_j \ \text{ mit } \ j = h(x_i) \tag{4.11}$$

Somit ergeben sich zwei Irrfahrten: (x_i) in den Elementen von G und (z_j) in den Restklassen modulo m. □

Auf Basis des Lemmas 4.13 hat Teske seine Betrachtung über Restklassenkörper modulo m fortgeführt und eine spezielle Wahrscheinlichkeitsverteilung P_{M_r} eingeführt, um diese mit der Gleichverteilung U zu vergleichen.

Definition 4.14 [Tes00, S. 823] Seien P_G und Q_G Wahrscheinlichkeitsverteilungen über einer Gruppe G. Der **Variationsabstand** von P_G und Q_G ist

$$\|P_G - Q_G\| := \frac{1}{2} \sum_{a \in G} |P_G(a) - Q_G(a)| = \max_{A \subseteq G} |P_G(A) - Q_G(A)| \tag{4.12}$$

Lemma 4.15 *Sei eine Irrfahrt (v_i) in den Restklassen modulo m gegeben durch die rekursive Form*

$$v_{i+1} = v_i + v_i \quad \textit{mit} \quad t_i \in_R \mathbb{Z}_m,\ v_0 = 0 \quad \textit{für} \quad i = 0,\ 1,\ 2,\ \ldots \tag{4.13}$$

So existiert eine Wahrscheinlichkeitsverteilung P_{M_r}, auf der die Irrfahrt (v_i) basiert und die für $r = m$ der Gleichverteilung entspricht.

Beweis: [Tes00, S. 823] Seien eine Irrfahrt (v_i) gegeben durch (4.13) und $r \in \mathbb{N}$. Seien weiterhin $M_r = \{c_0,\ c_1,\ \ldots,\ c_{r-1}\} \subseteq Z_m$ eine Menge von Restklassen modulo m mit den Wahrscheinlichkeiten $p_0,\ p_1,\ \ldots,\ p_{r-1}$, für die gilt $p_j > 0$ für $j = 0,\ 1,\ \ldots,\ r-1$ und $\sum_{j=0}^{r-1} p_j = 1$. Für ein solches M_r sei die Wahrscheinlichkeitsverteilung P_{M_r} definiert durch:

$$P_{M_r} : \mathbb{Z}_m \to [0,1],\ \ P_{M_r}(c) = \begin{cases} p_j & \text{,falls } c = c_j \text{ für } c_j \in M_r \\ 0 & \text{,sonst} \end{cases} \tag{4.14}$$

Eine Irrfahrt (z_i) generiert durch (4.11) würde dieser Definition genügen, wenn $j \in_R \{0,\ 1,\ \ldots,\ r-1\}$ gemäß einer Verteilung P_{M_r}, anstatt von der Hashfunktion h abzuhängen. Für den Spezialfall $r = m$ und $p_j = \frac{1}{r}$ entspricht $P_{M_r} = U$ der Gleichverteilung U. Eine darauf basierende Irrfahrt (v_i) erzeugt durch eine Iterationsfunktion $f : \mathbb{Z}_m \to \mathbb{Z}_m$ würde damit einer zufälligen Irrfahrt entsprechen. □

Basierend auf Erkenntnissen von M. V. Hildebrand folgt der Satz 4.16.

Satz 4.16 *[Tes00, Theorem 5.1] Seien $r \in \mathbb{N},\ r \geq 2$ und P_{M_r} laut (4.14) mit $M_r = \{c_0,\ c_1,\ \ldots,\ c_{r-1}\} \subseteq Z_m$ und $p_0,\ p_1,\ \ldots,\ p_{r-1}$, für die gilt $p_j > 0$ für $j = 0,\ 1,\ \ldots,\ r-1$ und $\sum_{j=0}^{r-1} p_j = 1$. Sei $\varepsilon > 0$, so existiert für ein genügend großes m ein derartig konstantes $\gamma > 0$, welches gegebenfalls von r und den p_j abhängt, jedoch nicht von m, so dass für ein $n = \left\lfloor \gamma m^{\frac{2}{r-1}} \right\rfloor$ gilt $E\left(\|P_{M_r}^{(n)} - U\|\right) < \varepsilon$. Dabei wird der Erwartungswert über eine gleichverteilte Auswahl jedes möglichen M_r berechnet.*

Folgerung 4.17 *[Tes98, S. 824] Mit steigender Gruppenordnung wird der Teil der Irrfahrt, der sich nicht zufällig verhält, vernachlässigbar klein im Verhältnis zum arithmetischen Mittel der Länge der Irrfahrt, bis eine Kollision gefunden wurde.*

Zusammenfassend gilt folgender Satzes 4.18.

Satz 4.18 *[Tes98, S. 824ff.] Sei G eine endliche abelsche Gruppe mit primer Gruppenordnung. Angenommen sei die Verwendung der ρ-Methode zur Berechnung des diskreten Logarithmus in G. Sei $r \geq 6$ und sei L definiert wie in (4.2). Wird eine r-additive Irrfahrt in G in Verbindung mit einer unabhängigen Hashfunktion h berechnet, so wird der durchschnittliche Wert für L nicht größer als 1.45 mit wachsender Gruppenordnung.*

5 Parallele Irrfahrten

Die parallele Ausführung von kritischen Rechenprozessen ist ein wichtiges Mittel zur Reduzierung der Laufzeit. Oorschot und Wiener haben neue Methoden [OW96] zur Parallelisierung von Algorithmen, welche Kollisionen von Irrfahrten finden, vorgestellt. Diese sind derart allgemein beschrieben, dass sie sich auf alle geeigneten kryptoanalytischen Probleme anwenden lassen.

5.1 Direkte Parallelisierung

Brent hatte bereits 1990 die Parallelisierung der ρ-Methode zur Faktorisierung untersucht und hat festgestellt, dass “Leider die parallelisierte Implementation der “Rho”-Methode keine lineare Verbesserung ergibt.” [Bre90, S. 29]. Die Auswirkungen auf die Pollard ρ-Methode für diskrete Logarithmen wird im Folgenden dargestellt.

Satz 5.1 *Seien f gegeben durch $x_{i+1} = f(x_i)$ mit $i = 0, 1, 2, \ldots$ und X die Anzahl der Iterationen bis das erste doppelte Glied der Irrfahrt (x_i), die durch f erzeugt wird, gefunden wurde. Erfolgt die Berechnung der Pollard ρ-Methode durch u Prozessoren parallel, dann gilt $E(X) \approx \sqrt{\frac{\pi m}{2u}}$.*

Beweis: Sei f eine Iterationsfunktion, die eine Irrfahrt (x_i) für $i = 1, 2, \ldots, n$ erzeugt. Die Wahrscheinlichkeit, dass keine Kollision nach k-maligem Auswählen mit Zurücklegen gefunden wird, ist

$$Pr(X > k) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{m}\right) \approx e^{-\frac{k(k-1)}{2m}} \tag{5.1}$$

Wie bereits in Satz 3.5 angegeben, sind $\sqrt{\frac{\pi m}{2}}$ Iterationen notwendig, bevor eine Kollision gefunden wird. Erfolgt die Berechnung der Pollard ρ-Methode von u Prozessoren parallel, dann berechnet jeder Prozessor für sich einen diskreten Logarithmus. Damit ist die Wahrscheinlichkeit der u parallel rechnenden Prozessoren

$$[P(X > k)]^u = \left[\prod_{i=0}^{k-1} \left(1 - \frac{i}{m}\right)\right]^u \approx e^{-\frac{k(k-1)u}{2m}} \tag{5.2}$$

Analog zum Beweis von Satzes 3.5 folgt $E(X) \approx \sqrt{\frac{\pi m}{2u}}$. □

Oorschot und Wiener haben hieraus gefolgert, dass “Der zu erwartende Beschleunigungsfaktor lediglich $\sqrt{u}$ beträgt und es eine schlechte Form der Parallelisierung ist, da es $\sqrt{u}$ länger dauert als die serielle Implementierung.” [OW96, S. 4].

5.2 Parallele Kollisionssuche

5.2.1 Verfahren zum Finden einer kleinen Anzahl von Kollisionen

In [OW96] wurden mehrere Verfahren beschrieben, um Kollisionen zu finden, je nachdem wie viele Kollisionen gefunden werden müssen. Für die Pollard ρ-Methode ist das Verfahren zum Finden einer kleinen Anzahl von Kollisionen ausreichend, da für diese nur das Auffinden einer Kollision eine Rolle spielt.

Definition 5.2 Gegeben sei eine Iterationsfunktion $f : S \to S$ über einer Menge S. Oorschot und Wiener bezeichnen die Irrfahrt (x_i), erzeugt durch $x_i = f(x_{i-1})$, $i = 1, 2, \ldots, n$, als **Pfad** (engl. trail) von Punkten x_i und definieren eine Teilmenge S_d von S als die **Menge der unterscheidbaren Punkte** x_d. Unterschieden werden diese Punkte von den anderen durch eine einfach zu bestimmende Eigenschaft, beispielsweise der Anzahl von führenden Nullen oder dem Hamminggewicht von x_i.

Jeder Prozessor vollzieht den Algorithmus 5.1, dabei seien $x_i \in_R S$ die zufällige Auswahl eines Punktes $x_i \in S$, $S_D \subseteq S_d$ die Menge der bereits getroffenen unterscheidbaren Punkte x_d, $S_C \subset S$ die Menge der gefundenen Kollisionen und `solveCollision`(x_i) ein Algorithmus, der den diskreten Logarithmus c oder $\emptyset$ zurückliefert.

Algorithmus 5.1: Paralleles Finden einer kleinen Anzahl von Kollisionen

```
Input : Funktion f
begin
    x_0 ∈_R S;
    x_i ← f(x_i);
    while x_i ∉ S_d do
        x_i ← f(x_i);
    if x_i ∈ S_D then
        c ←solveCollision(x_i);
        if c ≠ ∅ then
            S_C ← S_C ∪ {c};
    else
        S_D ← S_D ∪ {x_i};
end
```

Bemerkung 5.3 Die Kollision eines Pfades mit dem Startpunkt eines anderen Pfades ergibt keine Kollision in f und wird “Robin Hood” genannt [OW96, S. 6].

Beobachtung 5.4 *Sei $\theta = \frac{|S_d|}{|S|}$. Die Längen der Pfade sind geometrisch verteilt mit Erwartungswert $\frac{1}{\theta}$. Ist θ klein und damit die Pfade lang, dann sind Robin Hoods selten.*

Bemerkung 5.5 Es besteht auch die Möglichkeit, dass ein Pfad einen Zyklus ent-

hält. In diesem Fall kann die maximale Pfadlänge beschränkt werden, beispielsweise durch $l_t = \frac{l_m}{\theta}$ mit $l_m = 20$.

Folgerung 5.6 *[OW96, S. 7] Der Anteil der Pfade, der l_t übersteigt ist $(1-\theta)^{l_t} \approx e^{-l_m}$. Jeder dieser verlassenen Pfade ist etwa l_m mal länger als der Durchschnitt, so dass der Anteil der ersparten Laufzeit ungefähr $l_m e^{-l_m}$ beträgt, beispielsweise ist das für $l_m = 20$ ungefähr $20e^{-20} < 5 \cdot 10^{-8}$.*

5.2.2 Laufzeit-Analyse

Im Gegensatz zur direkten Parallelisierung findet bei der parallelen Kollisionssuche jeder Prozessor entweder einen unterscheidbaren Punkt oder eine Kollision und erhöht damit die Wahrscheinlichkeit, eine geeignete Kollision zu finden. Folgende Sätze fassen die Erkenntnisse aus [OW96, S. 7] zusammen.

Satz 5.7 *Seien $n = |S|$, u die Anzahl der Prozessoren und $q \in (0, 1]$ die Wahrscheinlichkeit, eine geeignete Kollision zu finden. Der Erwartungswert der Anzahl der Schritte, bis jeder Prozessor eine geeignete Kollision findet, beträgt $\frac{1}{u}\sqrt{\frac{\pi n}{2q}}$.*

Beweis: Aus Satz 5.1 folgt die Wahrscheinlichkeit, dass keine geeignete Kollision nach k Schritten unter allen Prozessoren u gefunden wurde:

$$\prod_{i=0}^{k-1}\left(1 - \frac{iq}{m}\right) \approx e^{-\frac{k(k-1)q}{2m}} \tag{5.3}$$

Analog zum Beweis des Satzes 3.5 wird fortgefahren. Damit folgt, dass der zusätzliche Aufwand lediglich $\frac{1}{\sqrt{q}}$ beträgt im Vergleich zum Fall, in dem alle Kollisionen geeignet sind. Der Grund ist die steigende Anzahl gefundener Kollisionen mit dem Quadrat der verbrauchten Zeit. □

Folgerung 5.8 *Sei t die benötigte Zeit für eine Iteration $x_{i+1} = f(x_i)$. Die Laufzeit, bis die erste geeignete Kollision gefunden wird, ist im Mittel*

$$T = t\left(\frac{1}{u}\sqrt{\frac{\pi m}{2q}} + \frac{1}{\theta}\right) \tag{5.4}$$

Bemerkung 5.9 Ist eine geeignete Kollision gefunden, so benötigt der jeweilige Prozessor im Durchschnitt noch $\frac{1}{\theta}$ Schritte, bis ein unterscheidbarer Punkt erreicht ist.

5.3 Anwendung der parallelen Kollisionssuche

Für die Anwendung der parallelen Kollisionssuche auf die Pollard ρ-Methode ergeben sich einige Anpassungen: Wahl des Startpunktes x_0 und des Algorithmus `solveCollision`(x_i). Oorschot und Wiener haben vorgeschlagen, dieselbe Iterationsfunktion $f : G \to G$ wie Pollard zu verwenden.

Jeder Prozessor vollzieht den Algorithmus 5.2, dabei seien

- $a_0,\ b_0 \in_R \mathbb{Z}_{m-1}$ die zufällige Auswahl der Startexponenten a_0 und b_0,
- $G_D \subseteq G_d$ die Menge der bereits getroffenen unterscheidbaren Punkte x_d als Teilmenge der Menge G_d der unterscheidbaren Punkte,
- $G_C \subset G$ die Menge der gefundenen Kollisionen und
- LCE($x_i,\ a_i,\ b_i$) ein Algorithmus, welcher zur linearen Kongruenz mit gegebenen Parametern den diskreten Logarithmus x oder $\emptyset$ liefert.

Algorithmus 5.2: Parallele Kollisionssuche mit Pollard ρ-Methode

Input : Funktion f
begin
  // Funktionen f, α und β siehe Gleichungen (3.6), (3.9) und (3.10)
  $a_0,\ b_0 \in_R G$;
  $x_0 \leftarrow g^{a_0} y^{b_0}$;
  $(x_i,\ a_i,\ b_i) \leftarrow (f(x_i),\ \alpha(x_i,\ a_i),\ \beta(x_i,\ b_i))$;
  while $x_i \notin G_d$ **do**
    $(x_i,\ a_i,\ b_i) \leftarrow (f(x_i),\ \alpha(x_i,\ a_i),\ \beta(x_i,\ b_i))$
  if $(x_j,\ a_j,\ b_j) \in G_D$ *mit* $x_i = x_j$ **then**
    $x \leftarrow$LCE$(x_i,\ a_i - a_j,\ b_j - b_i)$;
    if $x \neq \emptyset$ **then**
      $G_C \leftarrow G_C \cup \{x\}$;
  else
    $G_D \leftarrow G_D \cup \{(x_i,\ a_i,\ b_i)\}$;
end

Eine Sage-Implementierung ist im Listing A.4 zu finden. Dabei werden die Pfade der parallelen Kollisionssuche nacheinander anstatt parallel ausgeführt, um Pfade zu protokollieren, die eine geeignete Kollision ergeben. Das Beispiel 5.10 zeigt derartig erzeugte Pfade.

Beispiel 5.10 Analog zu Beispiel 3.10 ist der diskrete Logarithmus $x = \log_g y$ zu lösen. Dabei seien wieder $G = \mathbb{Z}_{53}^{\times} = \langle 2 \rangle$ mit Ordnung $|\mathbb{Z}_{53}| = 52$ und $y = 33$. Folgende zwei Pfade t_1 und t_2 kollidieren in einem unterscheidbaren Punkt und resultieren in einer geeigneten Kollision.

		0	1	2	3	4	5	6	7	8
t_1	x_i	36	19	43	**33**					
	a_i	4	4	8	8					
	b_i	24	25	50	51					
t_2	x_j	6	39	25	42	31	7	19	43	**33**
	a_j	16	17	17	34	34	16	17	34	34
	b_j	34	34	35	18	19	38	38	24	25

Es ergibt sich die Kongruenz $26x \equiv 26 \pmod{52}$, denn $t \equiv a_i - a_j \equiv 8 - 34 \equiv 26 \pmod{52}$ und $s \equiv b_j - b_i \equiv 25 - 51 \equiv 26 \pmod{52}$. Es ergibt sich $x = 23$, allerdings ist die Mächtigkeit der Lösungsmenge derart groß, dass sich andere Kollisionen besser eignen, beispielsweise die sich aus folgenden Pfaden t_3 und t_4 ergebende Kollision.

		0	1	2	3	4	5	6	7	8	9
t_3	x_i	14	38	23	52	51	49	45	37	21	**17**
	a_i	15	16	16	32	32	32	32	32	32	12
	b_i	52	0	1	2	3	4	5	6	7	14
t_4	x_j	11	45	37	21	**17**					
	a_j	3	4	4	4	8					
	b_j	25	25	26	27	2					

Es ergibt sich die Kongruenz $48x \equiv 12 \pmod{52}$, daraus die kleinere Lösungsmenge $X = \{10,\ 23,\ 36,\ 49\}$ und schließlich $x = 23$. In Abbildung 5.1 sind alle vier Pfade und die getroffenen unterscheidbaren Punkte dargestellt.

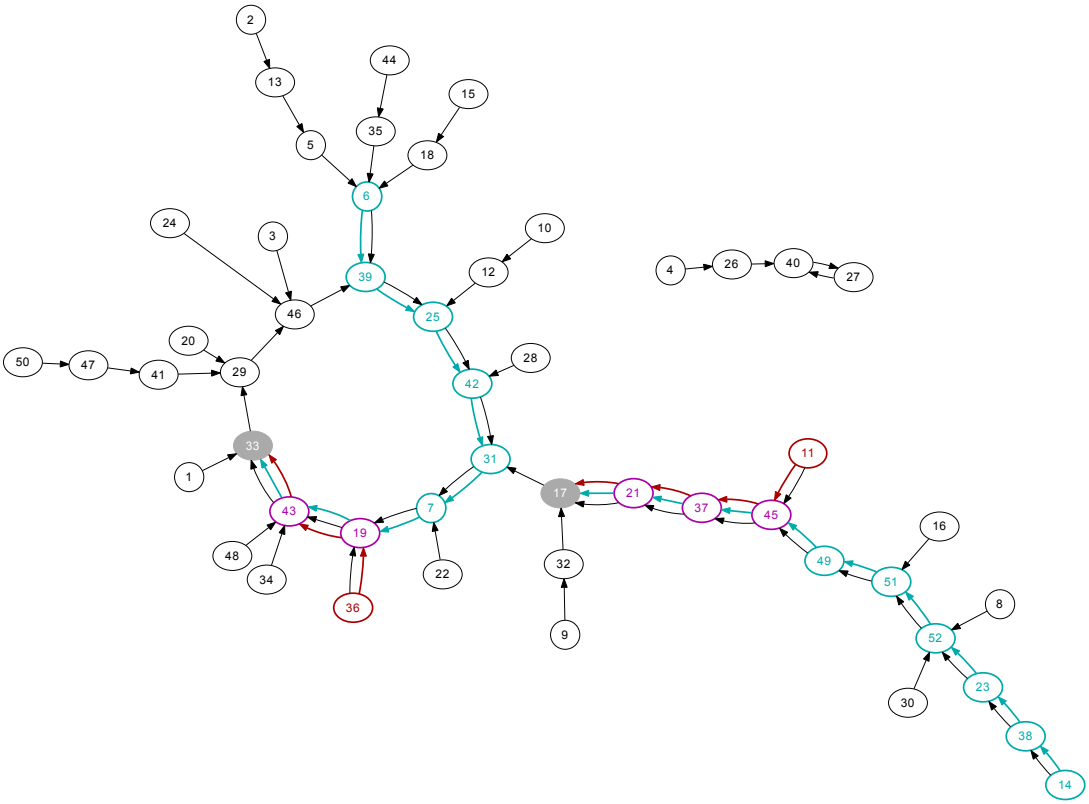


Abbildung 5.1: Darstellung der Pfade der parallelen Pollard ρ-Methode im Beispiel 5.10

Die Abbildung 5.1 wurde mit dem Programm `neato` des Pakets GraphViz erstellt, wobei die Beschreibungsstruktur durch Listing A.3 als Teil eines Sage-Notebooks generiert und mit den Pfaden der parallelen Kollisionssuche ergänzt wurde.

6 Automorphismen über Gruppen elliptischer Kurven

Die jüngsten Laufzeit-Verbesserungen der Pollard ρ-Methode basieren auf der Anwendung von Automorphismen auf Gruppen. Ein Automorphismus ist eine wie folgt definierte Abbildung. Weiterführende Literatur dazu ist [KM09].

Definition 6.1 Seien $(G, \diamond)$ und $(H, \circ)$ Gruppen. Eine Abbildung $\eta : G \to H$ heißt **Homomorphismus** von $(G, \diamond)$ in $(H, \circ)$, wenn für alle $x,\ y \in G$ gilt $\eta(x \diamond y) = \eta(x) \circ \eta(y)$. Eine Abbildung η wird **Automorphismus** genannt, wenn diese ein bijektiver Homomorphismus ist. Dabei gilt $\eta : G \to G$.

Für die nun folgenden Betrachtungen werden ausschließlich Gruppen G elliptischer Kurven E über $\mathbb{Z}_m$ betrachtet. Gesucht ist ein diskreter Logarithmus x, so dass gilt $Q = xP$ für $P,\ Q \in G$. Es sei $G_V = \langle V \rangle$ die durch den Punkt $V \in G$ erzeugte Gruppe.

Durch die Definition einer Äquivalenzrelation auf G_V ist es möglich, die zu durchsuchende Menge zu verkleinern. Es sei nun $\sim_\eta$ wie folgt definiert.

Definition 6.2 Sei $G_V = \langle V \rangle$ die durch den Punkt $V \in G$ erzeugte Gruppe. Für die Äquivalenzrelation $\boldsymbol{\sim_\eta}$ gilt $P \sim_\eta Q \Leftrightarrow P = \eta(Q)$ mit $P,\ Q \in G_V$ und $\eta : G_V \to G_V$.

6.1 Inversabbildung mit ergebnislosen Zyklen

Wiener und Zuccherato haben in [WZ98] vorgeschlagen, eine Gruppeneigenschaft elliptischer Kurven auszunutzen, um die zu betrachtende Gruppe zu verkleinern.

Definition 6.3 Sei $G_V = \langle V \rangle$ die durch den Punkt $V \in G$ erzeugte Gruppe. Der Automorphismus η wird **Inversabbildung** genannt, wenn für alle $P \in G_V$ gilt $\eta : G_V \to G_V,\ P \mapsto -P$.

Folgerung 6.4 *Für die Äquivalenzrelation $\sim_\eta$ folgt damit $P \sim_\eta Q \Leftrightarrow P = \eta(Q) = -P$.*

Eine Iterationsfunktion f zur Verwendung in der Pollard ρ-Methode wird derart angepasst, sodass f eine Irrfahrt auf der Menge $G_V / \sim_\eta$ der Äquivalenzklassen erzeugt.

Definition 6.5 Sei $X_i = (x_i,\ y_i) \in E(\mathbb{Z}_m)$ mit $i = 0,\ 1,\ 2,\ \ldots$ ein Punkt einer elliptischen Kurve. Für einen Repräsentant $\boldsymbol{\sim_\eta (x_i,\ y_i)}$ einer Äquivalenzklasse aus $G_V / \sim_\eta$ gilt $\sim_\eta (x_i,\ y_i) = (x_i,\ y_i + (y_i \bmod 2)(m - 2y_i))$ [BLS11, S. 134].

“Theoretisch ist es möglich, auf diese Weise die durchschnittliche Laufzeit der Pollard ρ-Methode um einen Faktor von $\sqrt{2}$ zu reduzieren.” [WZ98] Jedoch entstehen dadurch “ergebnislose Zyklen” (engl. fruitless cycles).

Die dabei entstehenden $2z$-Zyklen mit $z \in \mathbb{N}$ werden als nächstes dargestellt. Es sei $X_i \in G_V$ mit $i = 0,\ 1,\ 2,\ \ldots$ und f Teskes Iterationsfunktion aus Definition 4.8. Für elliptische Kurven E und $X,\ Y,\ V \in G_V$ sei f gegeben durch Gleichung (6.1):

$$f : G_V \to G_V,\ X \mapsto X + C_j \quad \text{mit} \quad C_j = c_j V + d_j Y \tag{6.1}$$

Sind nun X_i und X_{i+1} Glieder der durch f erzeugten Irrfahrt (X_i), die sich in der gleichen Teilgruppe G_{V_j} befinden. Dann ergibt sich in beiden Fällen der inverse Punkt:

$$\begin{aligned} X_{i+1} &= f(X_i) = -(X_i + C_j) \\ &\downarrow \\ X_{i+2} &= f(X_{i+1}) = -(X_{i+1} + C_j) = -(-(X_i + C_j) + C_j) = X_i \end{aligned} \tag{6.2}$$

Damit ist die Irrfahrt in einem **2-Zyklus** gefangen. Statistisch gesehen ist die Wahrscheinlichkeit für eine Auswahl des kleinsten Repräsentanten einer Äquivalenzklasse $\frac{1}{2}$ und für die Auswahl aus r Teilmengen $\frac{1}{r}$. Damit ergibt sich die Wahrscheinlichkeit $\frac{1}{2r}$, dass ein 2-Zyklus eintritt. Ähnlich verhält sich dieser Sachverhalt mit $2z$-Zyklen mit $z \in \mathbb{N}$ mit kleiner werdenden Wahrscheinlichkeiten. Ein 4-Zyklus kann also die folgende Form haben:

$$X_i \ \to\ X_i + C_j \ \to\ -(X_i + C_j + C_k) \ \to\ -(X_i + C_k) \ \to\ X_i \tag{6.3}$$

Wang und Zhang veröffentlichten in [WZ11] ein Satz, der die Wahrscheinlichkeit von $2z$-Zyklen beschreibt und in [WZ11, S. 7ff.] bewiesen wurde.

Satz 6.6 *[WZ11, S. 7] Seien $z,\ r \in \mathbb{N}$ und $c \in \mathbb{R}$ mit $c \in \left[\frac{1}{2},\ 1\right]$. Die Wahrscheinlichkeit p_{2z}, dass ein Punkt einer r-additiven Irrfahrt in einen $2z$-Zyklus verfällt, ist*

$$p_{2z} = \frac{cz!(r-1)!}{2^z r^{2z-1}(r-z)!} \tag{6.4}$$

Nachdem eine additive Irrfahrt in einem $2z$-Zyklus gefangen ist, ist es ebenfalls möglich, dass weitere kleinere Zyklen auftreten, was im Satz 6.6 berücksichtigt wurde.

In [WZ11, S. 8] haben Wang und Zhang für $r \in \{20,\ 128,\ 1024\}$ errechnete Wahrscheinlichkeiten dargestellt, die für $z = 1,\ 2,\ \ldots,\ 6$ in Tabelle 6.1 abgebildet sind. Zu erkennen ist in dieser, dass die Wahrscheinlichkeit, in einen Zyklus zu geraten, sinkt, je mehr die Gruppe aufgeteilt wird, also je größer r.

z	c	p_{2z}	$r = 20$	$r = 128$	$r = 1024$
1	1	$\frac{1}{2r}$	$2^{-5.3}$	$2^{-8.0}$	$2^{-11.0}$
2	$\frac{1}{2}$	$\frac{r-1}{4r^2}$	$2^{-10.7}$	$2^{-16.0}$	$2^{-22.0}$
3	$\frac{1}{2}$	$\frac{3(r-1)(r-2)}{8r^5}$	$2^{-14.6}$	$2^{-22.4}$	$2^{-31.4}$
4	$\frac{13}{24}$	$\frac{39(r-1)(r-2)(r-3)}{48r^7}$	$2^{-18.0}$	$2^{-28.4}$	$2^{-40.3}$
5	$\frac{71}{120}$	$\frac{71(r-1)(r-2)(r-3)(r-4)}{32r^9}$	$2^{-21.2}$	$2^{-34.0}$	$2^{-48.9}$
6	$\frac{461}{720}$	$\frac{461(r-1)(r-2)(r-3)(r-4)(r-5)}{64r^11}$	$2^{-24.3}$	$2^{-39.3}$	$2^{-57.2}$

Tabelle 6.1: Wahrscheinlichkeiten, dass eine Irrfahrt in einen $2z$-Zyklus verfällt

“Allerdings steigt mit größer werdendem r auch der Berechnungsaufwand.” [BKK+10, S. 72] Dabei sind beispielsweise für $r = 20$ noch circa $4 \cdot 10^6$ Schritte pro Sekunde möglich, wohingegen für $r = 1024$ noch $3.5 \cdot 10^6$ und für $r = 65536$ nur noch $5 \cdot 10^5$ Schritte pro Sekunde möglich sind.

6.2 Vermeidung von ergebnislosen Zyklen

Seitdem Wiener und Zuccherato die Inversabbildung [WZ98] vorgeschlagen haben, wurden die wissenschaftlichen Arbeiten [GLVC98], [DGM99], [BKK+10], [BLS11] und [WZ11] veröffentlicht, welche die Inversabbildung in Verbindung mit der Pollard ρ-Methode untersucht haben.

In [WZ98] wurde die Verwendung einer **vorausschauenden Methode** vorgeschlagen, um das Auftauchen von 2-Zyklen zu erkennen und zu vermeiden. Eine vorausschauende Methode wurde für den Folgepunkt X_{i+1} wie folgt definiert [BKK+11b]:

$$X_{i+1} = \begin{cases} X_r \in_R G_V & \text{, falls } j = h(\sim_\eta (X_i + C_j)) \text{ für } 0 \leq j < r \\ \sim_\eta (X_i + C_k) & \text{, mit } k \geq h(X_i) \text{ minimal,} \\ & \text{so dass } h(\sim_\eta (X_i + C_k)) \neq i \bmod r \end{cases} \tag{6.5}$$

Diese Methode umgeht zwar 2-Zyklen, allerdings keine $2z$-Zyklen mit größerer Periodenlänge mit $z > 1$. Des Weiteren lassen sich dadurch die 2-Zyklen nicht verhindern [WZ98], weshalb die Verwendung einer **größeren Anzahl von Blöcken r** für eine additive Irrfahrt vorgeschlagen wurde, so dass die Wahrscheinlichkeit des Auftauchens von 2-Zyklen vermindert wird.

Bos et al. haben in [BKK+10] dargestellt, dass 2-Zyklen weitere 2-Zyklen und 4-Zyklen verursachen und haben daher die folgenden **Änderungen der Iterations-**

funktion vorgeschlagen, um wiederkehrende Zyklen zu vermeiden.

$$X_{i+1} = \begin{cases} \sim_\eta (X_i + C_j) & \text{, falls } X_{i+1} \notin G_{V_j} \\ \sim_\eta (2X_i) & \text{, sonst} \end{cases} \tag{6.6}$$

Auch diese Methode verhindert keine $2z$-Zyklen mit einer größeren Periodenlänge und erfordert eine allgemeinere Behandlung von $2z$-Zyklen, da diese unvermeidbar sind und die Vermeidung weiterer Zyklen würde noch mehr Verwaltungs- und Implementierungsaufwand verursachen.

Gallant et al. haben in [GLVC98] eine allgemeine Methode vorgestellt, welche die Zyklen erkennt und aus diesen aussteigt. In [BKK+10], [BLS11] und [WZ11] wird diese Methode als $\alpha\beta$-**Zyklen-Erkennungsmethode** bezeichnet. Dabei werden nach α Schritten β aufeinanderfolgende Punkte gespeichert und die nachfolgenden Punkte mit diesen β Punkten verglichen.

Das Aussteigen aus dem Zyklus wird dabei durch einen neu berechneten Punkt C_e ermöglicht, so dass gilt $X_{i+1} = \sim_\eta (X_i + C_e)$ und C_e ist unabhängig von X_i. Eine andere Möglichkeit wird beschrieben, indem r neue Werte C_j berechnet werden. Bos et al. haben diese Ausstiegsmethoden in [BKK+10] analysiert und sich aufgrund der Häufigkeit der 2-Zyklen für die Verdopplung, wie in (6.6), entschieden.

Wang und Zhang und Bernstein et al. haben schließlich in [WZ11] und [BLS11] vorgeschlagen, die veränderte Iterationsfunktion und die $\alpha\beta$-Zyklen-Erkennungsmethode zu kombinieren, wodurch die Laufzeit um einen Faktor von circa $\sqrt{2}$ verringert wird.

6.3 Anwendung der Inversabbildung

In [WZ11] und [BLS11] wurde ein Verfahren vorgeschlagen, welches die $\alpha\beta$-Zyklen-Erkennungsmethode und eine Verdopplung zum Zyklen-Ausstieg kombiniert. Die Anwendung der Inversabbildung auf eine additive Irrfahrt ergibt eine wie folgt beschriebene Irrfahrt.

Die Irrfahrt beginnt mit einem Startpunkt $X_0 = \sim_\eta (a_0 Y)$ mit $Q \in_R G_V,\ a_0 \in_R \mathbb{Z}_{m-1}$. Die Iterationsfunktion f aus (6.1) erzeugt die Folgeglieder der Irrfahrt: $X_{i+1} = \sim_\eta (X_i + C_{h(X_i)})$. Die Hashfunktion h bildet dabei die Punkte X_i auf die Menge $\{0,\ 1,\ \ldots,\ r-1\}$ ab. Für die Punkte C_j gilt $C_j = c_j V + d_j Y$ mit $c_j,\ d_j \in_R \mathbb{Z}_{m-1}$ für $j \in \{0,\ 1,\ \ldots,\ r-1\}$.

Bereits in [WZ98] wurde eine vorausschauende Methode vorgeschlagen, die bereits berechnete Punkte X_i auf einen 2-Zyklus überprüft: $X_{i-1} = X_{i-3}$. Wenn $X_{i-1} \neq X_{i-3}$ gilt, wird fortgefahren mit $X_i = X_{i-1}$, andernfalls wird, wie in [BLS11] und [WZ11] vorgeschlagen, der kleinere der beiden Punkte verdoppelt: $X_i = \sim_\eta (2\min(X_{i-1}, X_{i-3}))$.

Definition 6.7 Sei f Teskes Iterationsfunktion (6.1) und (X_i) die durch f erzeugte Irrfahrt mit $i = 0,\ 1,\ 2,\ \ldots$. Ein Punkt X_i wird **Ausstiegspunkt** genannt, wenn dieser in einer Menge von β zuvor gespeicherten Punkten gefunden wird. Damit ergibt sich folgende neue Iterationsfunktion:

$$X_{i+1} = \begin{cases} \sim_\eta (2X_i) & \text{, falls } X_i \text{ ein Ausstiegspunkt ist} \\ \sim_\eta (X_i + C_j) & \text{, sonst} \end{cases} \tag{6.7}$$

Mit der Iterationsfunktion (6.7) ist ein Ausstieg aus einem Zyklus möglich. Bos et al. haben in [BKK+10, S. 77] die folgende Heuristik aufgestellt und bewiesen.

Heuristik 6.8 *[BKK+10, S. 77] Ein Zyklus mit mindestens einer Verdopplung ist sicherlich nicht ergebnislos.*

Die Aussage der Heuristik 6.8 ist bei genauerer Betrachtung in der Iterationsfunktion (6.7) zu erkennen. Denn wie bereits in den Formen der 2-Zyklen (6.2) und 4-Zyklen (6.3) dargestellt, führt die Punktaddition mit C_j in Verbindung mit der Inversabbildung zwangsweise zu einem Zyklus. Daher unterbricht die Punktverdopplung jeden möglichen Zyklus, da dieser keine Punktverdopplung mit C_j enthält.

Durch die eben betrachtete Irrfahrt scheinen Verzweigungen zu entstehen. Eine Verzweigung durch die Inversabbildung wurde bereits durch die Wahl eines Repräsentanten in Definition 6.5 umgangen. Auf gleiche Weise lässt sich das Verzweigen der Zyklenerkennung durch die Wahl eines allgemeinen Ausstiegspunkts vermeiden.

In [BLS11] wird für jeden möglichen $2z$-Zyklus mit $z \in \mathbb{N}$ und entsprechender Häufigkeit der Punkt X_{i-1} mit dem Punkt X_{i-2z} verglichen. Ist ein Zyklus gefunden, wird das Minimum aller Punkte $\{X_{i-1},\ \ldots,\ X_{i-2z}\}$ gesucht und wie oben verdoppelt.

In [WZ11] wird dafür ein auf der $\alpha\beta$-Zyklenerkennungsmethode basierendes Verfahren verwendet und hängt nicht von der Periodenlänge $2z$ eines Zyklus ab. Dabei werden folgende Variablen verwendet:

N	...	Mächtigkeit der Menge X_N, bestehend aus aufeinanderfolgenden zu speichernden Punkten
X_{min}	...	bisher gefundener kleinster Punkt
X_{neu}	...	neu berechneter kleinster Punkt der Menge X_N
X_e	...	Ausstiegspunkt

Das in [WZ11] beschriebene Verfahren lässt zunächst N aufeinanderfolgende Punkte X_i der Irrfahrt durch f aus (6.7) erzeugen, wobei für jeden Punkt laut der parallelen Kollisionssuche geprüft wird, ob ein unterscheidbarer Punkt vorliegt. Anschließend wird das X_{neu} aus dieser Menge X_N berechnet und geprüft, ob $X_{min} = X_{neu}$. Wenn dieser Fall eingetreten ist, wurde ein Zyklus gefunden.

Laut [WZ11] ist dieses bisherige Verfahren ungeeignet für häufig auftretende 2- und 4-Zyklen. Unter der Bedingung, dass $N \mid 4$ wurde das Verfahren dahingehend modifiziert, dass stets vier aufeinanderfolgende X_i überprüft werden, aus welchen das Minimum gebildet wird, beispielsweise $\{X_1,\ X_2,\ X_3,\ X_4\}$. Dabei wird getestet, ob $\min(X_1,\ X_2) = \min(X_3,\ X_4)$, um das Auftreten kurzer Zyklen besser zu erkennen.

Das Verfahren wurde in [WZ11, S. 11] als Algorithmus 1 zusammengefasst. Allerdings beinhaltet dieser Algorithmus einen Fehler in der Zyklenerkennung (Zeile 27), der durch den Autor dieser Masterarbeit bemerkt wurde. Ist laut diesem Algorithmus ein 2-Zyklus gefunden worden, wird die äußere `for`-Schleife abgebrochen. Anschließend findet der Algorithmus in diesem Fall nicht zugewiesene Variablen vor und kann daher nicht fortgesetzt werden.

Eine weitere Verbesserung ist möglich, wenn das in Algorithmus 1 verwendete Verfahren zum Ermitteln des kleinsten Punktes durch ein fortlaufendes Verfahren ersetzt wird, wie es in [BLS11] vorgeschlagen wurde. Dabei wird der kleinste Punkt X_{neu} innerhalb einer Liste von gespeicherten Punkten $[X_0,\ X_1,\ \ldots,\ X_N]$ ermittelt, indem mit $X_{neu} = \min(X_0,\ X_1)$ gestartet und schrittweise $X_{neu} = \min(X_{neu},\ X_i)$ für $i \in \{2,\ \ldots,\ N\}$ berechnet wird.

Der Algorithmus 1 in [WZ11, S. 11] ist zusammen mit der Verbesserung in Algorithmus 6.1 dargestellt, dabei wird zusätzlich noch auf kurze Zyklen geprüft.

Algorithmus 6.1: Parallele Pollard ρ-Methode mit Inversabbildung

Input : Startwert X_0
Output : Unterscheidbarer Punkt $X_d \in G_d$
begin
  $(X_{min},\ X_{neu},\ X_e) \leftarrow$ (None, None, None);
  while *True* **do**
    if $X_e \neq \mathbb{O}$ **then** $(X_0,\ X_e) \leftarrow (2X_e,$ None$)$;
    if $X_0 \in G_d$ **then return** X_0;
    for $i \in \{1,\ 2,\ \ldots,\ N\}$ **do**
      $X_i \leftarrow f(X_{i-1})$;
      if $X_i \in G_d$ **then return** X_i;
    $X_{neu} \leftarrow \min(X_0, X_1)$;
    for $i \in \{1,\ 2,\ \ldots,\ N\}$ **do**
      if $X_{neu} = X_i$ **then**
        $X_e \leftarrow X_{neu}$;
        break
      else $X_{neu} \leftarrow \min(X_{neu},\ X_i)$;
    if $X_{neu} = X_{min}$ **then** $X_e \leftarrow X_{min}$;
    else $X_0 \leftarrow X_N$;
end

6.4 Experimente und Auswirkungen

Bos et al. haben in [BKK+10] für Teskes Irrfahrten mit und ohne Inversabbildung Wahrscheinlichkeiten für den Fall ermittelt, dass eine Kollision nach der $n+1$-ten Iteration eintritt. Nachdem eine Million Irrfahrten experimentell in zufälligen Gruppen elliptischer Kurven von 31bit-primer Ordnung durchgeführt wurden, wurden die berechneten Mittelwerte ins Verhältnis zur Anzahl der Schritte gesetzt, bis eine Kollision gefunden wird. Die Ergebnisse sind in Tabelle 6.2 [BKK+10, S. 70] dargestellt.

Irrfahrt	**ohne $\sim_\eta$**	**mit $\sim_\eta$**
8-additiv	1.083	1.039
16-additiv	1.037	1.017
32-additiv	1.018	1.009
16+4-gemischt	1.043	1.038
16+8-gemischt	1.078	1.077

Tabelle 6.2: Mittelwerte der benötigten Laufzeit mit und ohne Inversabbildung

Aus Tabelle 6.2 folgt, dass sich alle betrachteten Irrfahrten schlechter verhalten, als die zufällige Irrfahrt. Irrfahrten auf Basis der Inversabbildung verhalten sich zwar im Mittel besser als Irrfahrten ohne diese, haben jedoch auch einen großen Verwaltungsaufwand zur Vermeidung von Zyklen.

Wang und Zhang haben in [WZ11] Experimente präsentiert, die in einer Gruppe einer elliptischen Kurve von 131bit-primer Ordnung durchgeführt wurde. Dabei wurden eine Million unterscheidbare Punkte gefunden und der Mittelwert von nötigen Iterationen in einer Tabelle dargestellt, die in Tabelle 6.3 abgebildet ist.

r	**Iterationen ohne $\sim_\eta$**	**Iterationen mit $\sim_\eta$**
20	951.17	951.66
128	949.94	952.37
1024	951.02	949.53

Tabelle 6.3: Experimentelle Ergebnisse von Wang und Zhang

Experimente mit der in Listing A.7 implementierten Python-Klasse zeigen stets wiederkehrende Zyklen, trotz dass eine Verdopplung des kleinsten Punktes im Zyklus durchgeführt wurde. Beobachtet wurde dabei auch, dass Verdopplungen zu Punkten führt, welche sich in der Irrfahrt vor dem Zyklus befinden. Dieser Fall trat bei vielen Gruppen elliptischer Kurven mit kleiner primer Ordnung auf, konnte jedoch aufgrund fehlender Ressourcen nicht für große Ordnungen betrachtet werden.

Schlussfolgerungen in [BKK+10], [WZ11] und [BKK+11b] ergeben schließlich einen Beschleunigungsfaktor der Laufzeit von bis etwa $\sqrt{2}$. Allerdings treffen diese Schlussfolgerungen nur für übliche Server und Desktop-Rechner zu. Zur Parallelisierung wer-

den jedoch Hardwarelösungen eingesetzt, die auf schnelle Berechnungen ausgelegt sind, beispielsweise Grafikkarten oder Playstation 3 (PS3).

Für diese Systeme begünstigt die Verwendung von additiven Irrfahrten die nötige Laufzeit. Werden allerdings zum Ausstieg aus Zyklen Punktverdopplungen notwendig, so folgt laut [BKK+11b, S. 6], dass diese nebenläufigen Prozesse aufeinander warten müssen und die Verarbeitung neu strukturiert werden muss, wenn eine Verdopplung durchgeführt wird.

Bernstein et al. haben in [BLS11] eine erfolgreiche Anwendung der Inversabbildung für Berechnungen auf der PS3 veröffentlicht. Die Experimente wurden dabei auf Basis der standardisierten elliptischen Kurve secp112r1 durchgeführt. Im Gegensatz zu der von Bos et al. erreichten Leistung von 456 Takten pro Iteration [BKK+11b], haben Bernstein et al. 365 Takte pro Iteration erreicht und damit ein Beschleunigung der Laufzeit von rund 1.25.

6.5 Weitere Methoden zur Laufzeitverbesserung

Eine in [BKK+09a] vorgeschlagene Methode ist die **Vermeidung, die Koeffizientenfolgen (a_i) und (b_i) zu berechnen**. In den üblichen Berechnungen werden neben der Erzeugung der Irrfahrt (X_i) eben auch diese Koeffizienten mitberechnet, um sie, im Fall der parallelen Kollisionssuche, dem zentralen Server zu übermitteln und im Fall einer geeigneten Kollision direkt den diskreten Logarithmus zu berechnen.

Allerdings sind nicht alle beigetragenen unterscheidbaren Punkte nützlich zur Berechnung des diskreten Logarithmus, mit anderen Worten sind im Idealfall lediglich zwei unterscheidbare Punkte zur Berechnung des diskreten Logarithmus nötig. Daher reicht es aus, lediglich die Koeffizienten, durch welche der Startpunkt erzeugt wurde, zusammen mit dem unterscheidbaren Punkt an den zentralen Server zu übertragen. Wurde eine Kollision gefunden, so lassen sich die resultierenden Koeffizienten durch die explizite Erzeugung der Koeffizientenfolgen (a_i) und (b_i) berechnen.

In [BKK+11b] wurde auf eine weitere Methode **Tag-Tracing** hingewiesen. Ziel dieser Methode ist die Berechnung der y-Koordinate nur dann, wenn die x-Koordinate zu einem unterscheidbaren Punkt gehört. Beschrieben wurde dabei die Verallgemeinerung für das ECDLP, welches einen Beschleunigungsfaktor von $\frac{6}{5}$ aufweist. Jedoch ist für die Anwendung der Inversabbildung die y-Koordinate zur Bestimmung des Repräsentanten der Äquivalenzklasse erforderlich, weshalb das Tag-Tracing mit der Anwendung der Inversabbildung nicht kombinierbar ist.

Die neuesten wissenschaftlichen Arbeiten [BKK+11b] und [BLS11] beschreiben **Verbesserungen der verwendeten Arithmetik** zur Beschleunigung der Berechnungen

auf einer Playstation 3. Entscheidende Beschleunigungen ergeben die Verbesserungen der Operationen **Multiplikation und Inversion** bezüglich der Multiplikation. In [BLS11] wird für die erreichten 362 Takte pro Iteration aufgelistet, welche Operationen die meisten arithmetischen Befehle benötigen:

- 5 Multiplikationen: je 43.75 arithmetische Befehle und damit insgesamt 218.75
- 1 Quadrierung: 31.75 arithmetische Befehle
- 1 Kanonisierung (der y-Koordinate): 24.75 arithmetische Befehle
- 6 Substraktionen: 10.00 arithmetische Befehle
- $\frac{1}{224}$ Inverisonen minus $\frac{3}{224}$ Multiplikationen: $\approx$ 20.83 arithmetische Befehle

Wie im letzten Punkt der Auflistung und in [BLS11] dargestellt, haben Bernstein et al. die besonders aufwendigen Operationen Inversion bezüglich der Multiplikation durch die modulare Montgomery-Inversion ersetzt. Dabei werden n Inversionen durch $3(n-1)$ Multiplikationen gebündelt berechnet. Zukünftige Verbesserungen werden also besonders auf der zugrundeliegenden Arithmetik basieren.

In den nun folgenden Schlussfolgerungen wird der Einfluss der eingesetzten Hardwarelösungen dargestellt und alle bisherigen Betrachtungen zusammengefasst, unter Einbeziehung der Auswirkungen auf die einsetzbaren Schlüssellängen.

7 Schlussfolgerungen

7.1 Auswirkungen von Hardwarelösungen auf die Laufzeit

Die zum Zeitpunkt der Masterarbeit stattfindenden Berechnungen [BKK+09a] beziehen sich auf die Certicom-Herausforderung ECC2K-130 [Cer97]. Dabei finden die Berechnungen in einer Gruppe einer elliptischen Kurve über einen Binärkörper $\mathbb{Z}_{2^{130}}$ statt, wobei die parallele Kollisionssuche, additive Irrfahrten und Automorphismen angewendet werden. In [BKK+11a] werden dazu die aktuellen Informationen dargestellt.

Aufbauend auf den seit 2009 stattfindenden Berechnungen wurde in [BKK+09b] unter anderem über die Sicherheit von 160-Bit Schlüssellängen für die Elliptische-Kurven-Kryptographie berichtet. Dabei wurden die eingesetzten Hardwarelösungen in Bezug auf die mögliche Leistung betrachtet. Dargestellte Hardwarelösungen sind Desktop- und Server-Rechner, Sony Playstation 3 Spielekonsolen (PS3), Graphikkarten, FPGAs und ASICs, die im Folgenden kurz eingeführt werden.

Definition 7.1 Seien T_t die Taktfrequenz einer Recheneinheit einer Maschine, T_c die Anzahl von Takten pro Iteration und T_m die Anzahl der Recheneinheiten in einer Maschine. Für die **Iterationen pro Sekunde T_e** gilt

$$T_e = \frac{T_t}{T_c} T_m \tag{7.1}$$

Seien r die Anzahl der Blöcke einer Partition der durch den Punkt $V \in E(\mathbb{K}_p)$ erzeugten Gruppe und $q = |\langle V \rangle|$. Ein **Maschinenjahr T_y**, bis eine Maschine ein ECDLP gelöst hat, ist

$$T_y = \frac{\sqrt{\frac{\pi q}{2}}}{T_e \cdot (3600 \cdot 24 \cdot 365.25)\sqrt{1 - \frac{1}{r}}} \tag{7.2}$$

Über Benchmarks für einige **Desktop- und Server-CPUs** wurde in [BKK+09b] und [BKK+09a] berichtet und in [BKK+11a] werden aktuelle Testergebnisse veröffentlicht. Die Informationen beider Quellen sind in Tabelle 7.1 zu finden. Dabei ist erkennbar, dass die Intel 4-Kern-Prozessoren in etwa dieselbe Anzahl an Takten pro Iteration benötigen. Angenommen, es ergeben sich für einen Intel Core i7-2600K ebenfalls 533 Takte pro Iteration, dann wären $25.516 \cdot 10^6$ Iterationen pro Sekunde möglich.

Prozessor	T_m	T_t [GHz]	T_c	T_e
AMD Opteron 875	2	2.210	1058	$4.178 \cdot 10^6$
AMD Phenom II X4 905e	4	2.508	596	$16.832 \cdot 10^6$
Intel Core 2 Quad Q6600	4	2.394	533	$17.966 \cdot 10^6$
Intel Core 2 Extreme Q6850	4	2.997	534	$22.449 \cdot 10^6$
Intel Core i7-2600K *	4	3.400	533	$25.516 \cdot 10^6$

Tabelle 7.1: Laufzeitbenchmarks für Desktop- und Server-CPUs bezüglich ECC2K-130. * eigene Schätzung.

Eine **Sony Playstation 3** besteht laut [BKK+09b] aus einer Multikern-Architektur mit einem zentralen PowerPC-Kern und acht zusätzlichen Kernen, die Synergistic Processor Elements (SPEs) genannt werden. Nach der Einrichtung eines Linux-Betriebssystems und der MPM-Bibliothek von IBM lassen sich sechs dieser SPEs direkt für parallele Berechnungen verwenden.

Die Besonderheit der SPEs ist die schnelle Ausführung gleichartiger Rechenoperationen auf mehrere gleichzeitig vorliegende Datenströme. Dadurch entstehen ebenso Nachteile, beispielsweise bei der Anwendung der Inversabbildung, für die Verzweigungen notwendig sind. In Tabelle 7.2 sind die Verbesserungen der arithmetischen Operationen und die dadurch entstehende Leistung der Iterationen pro Sekunde ersichtlich. Eine wesentliche Beschleunigung von 1.25-fach lässt sich durch die Anwendung der neuesten Erkenntnisse aus [BLS11] erreichen, wodurch sich mehr als 30 Millionen Iterationen pro Sekunde ergeben könnten.

Quelle	**ECDLP**	T_m	T_t [GHz]	T_c	T_e
[BKK+11b]	secp112r1	6	3.192	456	$42.000 \cdot 10^6$
[BLS11]	secp112r1	6	3.192	362	$52.906 \cdot 10^6$
[BKK+11a]	ECC2K-130	6	3.192	749	$25.570 \cdot 10^6$

Tabelle 7.2: Laufzeitbenchmarks für die PS3 (SPE)

NVIDIA Grafikkarten werden immer mehr für parallele Berechnungen verwendet. Durch NVIDIAs CUDA-Bibliothek werden die Berechnungen auf den Grafikprozessoren ermöglicht. Derzeit [BKK+11a] berechnen Grafikkarten vom Typ NVIDIA GTX 295 unterscheidbare Punkte. Dabei werden 60 Kerne der zwei GT200b GPUs verwendet. Im März 2011 wurden neue Algorithmen eingesetzt, welche die Anzahl der Takte pro Iteration auf 1379 verbessern und damit **54.03 Millionen** Iterationen pro Sekunde ermöglichen. Seit April 2011 werden 214 solcher Grafikkarten eingesetzt.

2006 wurden in [GPP06] erstmals auch programmierbare Logikgatter-Anordnungen, kurz **FPGAs**, zur Berechnung vorgeschlagen. Dabei wurde ein System namens COPACOBANA (Cost-Optimized Parallel Code Breaker) entworfen, welches 20 Module mit jeweils 6 FPGAs enthält. Seitdem wurde COPACOBANA mehrfach verbessert

und neue FPGAs eingesetzt. In [FBB+10] wurde schließlich eine darauf basierende Architektur vorgestellt, deren 128 FPGAs jeweils eine Taktfrequenz von 111 MHz und jeweils eine Leistung von **111 Millionen** Iterationen pro Sekunde erreichen.

Die teuerste unter allen Hardwarelösungen stellen **ASICs** dar, anwendungsspezifische integrierte Schaltkreise. Schätzungen in [BKK+09a] deuten eine Leistung von **800 Millionen** Iterationen pro Sekunde je ASIC an. Allerdings werden in der gleichen Quelle die Kosten auf unter 60000 Euro geschätzt.

Trotz all dieser leistungsstarken Hardwarelösungen liegen die Möglichkeiten, ECCp-131 zu lösen, in weiter Zukunft. Bernstein et al. erreichen mit aktuellen Verbesserungen [BLS11] für secp112r1 35.6 PS3-Jahre pro SPE. Extrapoliert man dies laut [BKK+09b, S. 10], ergeben sich für die 163-Bit Schlüssellänge, verwendet in Herausforderung ECCp-163, unter Berücksichtigung einer 20% schlechteren Laufzeit

$$\left(\frac{163}{112}\right)^2 \cdot 2^{\frac{163-112}{2}} \cdot 1.2 \cdot 35.6 \approx 4.294 \cdot 10^9 \text{ PS3-Jahre pro SPE.} \tag{7.3}$$

Es werden also mehr als 715 Millionen Playstation 3 benötigt, um ein darauf basierendes ECDLP in einem Jahr zu lösen. Fünf COPACOBANAs, bestehend aus 5·128 FPGAs, könnten laut [FBB+10] die Herausforderung ECC2K-130 in etwa einem Jahr lösen. Pro COPACOBANA ergeben sich $128 \cdot (111 \cdot 2^{20} \cdot 3600 \cdot 24 \cdot 365.25) \approx 2^{58.7}$ Iterationen pro Jahr. Damit wäre die Herausforderung ECC2K-130 mit etwa $2^{60.9}$ Iterationen durch fünf COPACOBANAs in 1.088 Jahren lösbar.

Die Ordnung der in der Herausforderung ECC2K-130 erzeugten Gruppe hat 129 Bits. Führt man die selben Betrachtungen für einen in [FBB+10] vorgestellten COPACOBANA durch, ergeben sich für die 163-Bit Schlüssellänge, verwendet in Herausforderung ECCp-163, unter Berücksichtigung einer 20% schlechteren Laufzeit

$$\left(\frac{163}{129}\right)^2 \cdot 2^{\frac{163-129}{2}} \cdot 1.2 \cdot 1.088 \cdot 5 \approx 1.366 \cdot 10^6 \text{ COPACOBANA-Jahre.} \tag{7.4}$$

Demnach werden also mehr als eine Million COPACOBANAs benötigt, um das ECCp-163 Problem in einem Jahr zu berechnen. Die Betrachtungen zeigen, dass mit den aktuellen Algorithmen und aktueller Technik ein ECDLP mit einem 163-Bit langen Schlüssel derzeit unmöglich zu lösen ist. In [BKK+11b] wurde auch geschlussfolgert, dass “160-Bit prime ECDLP derzeit außer Reichweite sind”.

Schließlich wurde schon in [BKK+09b, S. 11] zusammengefasst, dass unter diesen Voraussetzungen selbst 2020 noch ein Angriff auf eine 160-Bit Elliptische-Kurven-Kryptographie viel Aufwand erfordert, hohe Kosten verursacht und nahezu unmöglich ist.

7.2 Zusammenfassung

Trotz der heutigen Technik und Algorithmen ist es nicht möglich, einen Angriff auf eine 160-Bit Elliptische-Kurven-Kryptographie durchzuführen. Es folgt also aus den Betrachtungen in dieser Masterarbeit, dass das minimal empfohlene Sicherheitslevel von 80-Bit noch bis mindestens 2013 bestand hat.

Die folgende Tabelle fasst die aktuellen Empfehlungen von ECRYPT II [ECR10], der französischen Behörde für Netzwerk- und Informationssicherheit (FNISA) aus [FNI10], der Bundesnetzagentur (BNA) aus [Bun11] und des US-amerikanischen National Institute of Standards and Technology (NIST) aus [BBB+11] zusammen. Dabei werden von der Bundesnetzagentur nur Schlüssellängen für Signaturalgorithmen vorgeschlagen.

<table>
<tr><th>Jahre</th><th>ECRYPT II</th><th>FNISA</th><th>BNA</th><th>NIST</th></tr>
<tr><td>2009-2012</td><td rowspan="2">160</td><td rowspan="3">≥ 200</td><td rowspan="2">224</td><td>160</td></tr>
<tr><td>2011-2015</td><td rowspan="3">224</td></tr>
<tr><td>2013-2020</td><td>192</td><td rowspan="3">256</td></tr>
<tr><td>2021-2030</td><td>224</td><td rowspan="3">≥ 256</td></tr>
<tr><td>2031-2040</td><td>256</td><td>256</td></tr>
<tr><td>>2040</td><td>512</td><td>384</td></tr>
</table>

Tabelle 7.3: Empfohlene Schlüssellängen für die Elliptische-Kurven-Kryptographie

Für mobile Endgeräte ist die Wahl eines geeigneten Kryptosystems entscheidend. Aufgrund des enormen Ressourcenverbrauchs wurden Kryptosysteme mit großen Schlüssellängen über Umwege in mobile Endgeräte integriert. Ein Beispiel dafür sind microSD-Karten von certgate, auf denen ein Mikroprozessor für Schlüsselberechnungen auf Basis des RSA-Kryptosystems und 2048-Bit langen Schlüsseln enthalten ist. Nachteilig ist solch eine Lösung allerdings dann, wenn kein microSD-Kartenleser vorhanden ist, beispielsweise bei Sensorknoten, oder wenn aufgrund des Energieverbrauchs kein Kartenleser eingesetzt werden soll.

Elliptische-Kurven-Kryptosysteme bilden damit eine ideale Grundlage zur Einbettung in mobile Endgeräte und Sensorknoten, wobei eine geeignete Implementierung der Arithmetik und die Wahl einer geeigneten Schlüssellänge für die Laufzeit ausschlaggebend ist und damit auch für die Benutzerfreundlichkeit sicherheitsrelevanter Abläufe. Zusammenfassend lassen sich folgende Aussagen über derzeit sichere Schlüssellängen treffen.

Je nach Einsatzgebiet werden gewisse Schlüssellängen vorgeschrieben oder empfohlen. Für eine sichere Kommunikation zwischen Behörden, Geheimdiensten oder Regierungen werden Schlüssellängen von mindestens 224 Bit vorgeschrieben, wohingegen für kleine oder größere Organisationen Schlüssellängen von mindestens 160 Bit laut den bisherigen Betrachtungen noch als sicher eingestuft werden.

Zur Verwertung der Ergebnisse dieser Masterarbeit für zukünftige Projekten der pitcom PROJECT GmbH ist es demnach bezüglich der Elliptischen-Kurven-Kryptographie ratsam, die Schlüssellängen 160 Bit und 224 Bit gleichermaßen vorzusehen und zu betrachten. Auf diese Weise werden alle denkbaren Einsatzgebiete beachtet.

Für weniger leistungsfähige mobile Geräte lassen sich damit noch Elliptische-Kurven-Kryptosysteme mit 160 Bit Schlüssellängen einsetzen und bilden so die Basis für eine sichere Kommunikation für kleine Organisationen. Für leistungsfähigere mobile Geräte ist es empfehlenswert, Elliptische-Kurven-Kryptosysteme mit 224 Bit Schlüssellängen einsetzen, so dass diese dem empfohlenen Sicherheitslevel für größere Organisationen entsprechen.

Weitere wichtige Hinweise zur Verwertung ergeben sich aus den Anforderungen der Elliptischen-Kurven-Kryptographie und der von mobilen Endgeräten zur Verfügung gestellten Algorithmen. Dabei sind geeignete Algorithmen für die zugrundeliegende Arithmetik des verwendeten Restklassenkörpers und der verwendeten elliptischen Kurve derart zu wählen, dass diese dem aktuellen Stand der Wissenschaft entsprechen, möglichst Seitenkanalangriffe verhindern und schließlich eine schnellstmögliche Berechnung des Chiffren-Textes ermöglichen.

7.3 Ausblick

Für kryptographische Anwendungen der elliptischen Kurven sind auch andere **Angriffsmethoden** und -möglichkeiten zu beachten, welche Schlussfolgerungen auf verwendete Schlüssel zulassen. Zwei mögliche Schwachpunkte sind die Erzeugung von Zufallszahlen und mögliche Seitenkanalangriffe. So sollten beispielsweise keine Werte für die Erzeugung von Zufallszahlen verwendet werden, die als leicht rekonstruierbare Startwerte die Berechnung solcher Zufallszahlen ermöglichen.

Des Weiteren ergeben sich aus rechenintensiven Operationen, beispielsweise Multiplikation oder Inversion bezüglich der Multiplikation, oder deren Ressourcenverbrauch, wie Laufzeit oder Speicherverbrauch, ebenfalls Rückschlüsse auf verwendete Schlüssel. Brumley und Tuveri haben solche Seitenkanalangriffe in [BT11] dargestellt.

Es existieren auch elliptische Kurven über Körper mit der Charakteristik zwei, die auch **Koblitz-Kurven** genannt werden. Für die dabei verwendete Arithmetik für endliche Binärkörper $\mathbb{Z}_{2^m}$ gibt es zwar optimierte Algorithmen, allerdings sind Koblitz-Kurven auch anfälliger gegen Angriffe. Beispielsweise existiert ein spezieller Homomorphismus, der sogenannte Frobeniusendomorphismus, welcher die Laufzeit kryptoanalytischer Algorithmen um einen Faktor von $\sqrt{m}$ beschleunigt.

Es existieren auch Kryptosysteme über anderen Kurven, beispielsweise **hyperelliptische Kurven**. Diese ermöglichen eine beschleunigte Berechnung der Gruppenelemente, eine größere Auswahl von Kurven und insbesondere kleinere Schlüssellängen, als elliptische Kurven. Allerdings ergeben sich auch einige Nachteile, beispielsweise des benötigten Speichers für Vorberechnung und der für hyperelliptischen Kurven langsameren Arithmetik im Vergleich zu elliptischen Kurven. Literatur hierzu ist [CF03].

Die **Kryptographie basierend auf hyperelliptischen Kurven** bildet also ein weiteres betrachtenswertes Thema. Dabei besonders interessant ist die Fragestellung, ob die Berechnung von Gruppenelementen und die Verkleinerung der Schlüssellängen eine Verkürzung der Berechnungszeit ergeben oder nicht. Folglich stellt sich die Frage, ob Hyperelliptische-Kurven-Kryptosysteme Elliptische-Kurven-Krytosysteme ablösen können.

Anhang A: Sage-Implementierungen

Die folgenden Listings beinhalten Implementierungen vom Autor dieser Masterarbeit. Das zugrundeliegende Computer-Algebra-System ist Sage, was auf der Programmiersprache Python basiert.

A.1 Notebook: Pollard-Rho-Methods

Das Sage-Notebook “Pollard-Rho-Methods” mit den Listings A.1 bis A.4 wurde mit der URL `https://sage.hs-mittweida.de/home/pub/3/` erstmals am 15.06.2011 veröffentlicht.

```
# linear congruence eqation ... aX == b (mod m)
#     check function: lambda x, y: x == y
def LCE(a, b, m, check = None, debug = False):
    var('x')
    if debug: print "%s*X == %s (mod %s)" % (a, b, m)
    X = solve_mod([a*x == b], m, solution_dict=True)
    if debug: print "X = %s" % sorted([d[x] for d in X])
    if check == None: return X
    else:
        for dictX in X:
            if check(dictX[x], y):
                return dictX[x]
        return None
```

Listing A.1: Funktion zum Lösen von linearen Kongruenzen $ax \equiv b \pmod{m}$

Eine Sage-Implementierung des Algorithmus 3.4 ist in Listing A.2 zu finden, welches zusätzlich die Funktion `LCE` aus Listing A.1 benötigt. Das Beispiel 3.6 wurde mittels des Listings A.2 erstellt.

```
class PollardRhoDLPFloyd:
    """Pollards rho method to solve the DLP y = g^x (mod m)

        input: int y, g, m for Z_g = <g>

        output: x
    """
    def __init__(self, m, g, y, a_0 = None, debug = False):
        self.debug = debug
        self.m = m
        self.G = IntegerModRing(m)
        self.H = IntegerModRing(m-1)
        self.g = self.G(g)
        self.y = self.G(y)
        self.a_0 = a_0

        # storage for values
        self.C = []
        self.Ci = []
        self.Cn = 8
```

```
        # f(x)
        self.f = lambda x: (self.y*x  if 0 < x <= self.m//3 else
                            x**2 if self.m//3 < x <= self.m*2//3 else
                            self.g*x)
        # alpha(x)
        self.a = lambda x, n: (n+1 if 0 < x <= self.m//3 else
                               n*2 if self.m//3 < x <= self.m*2//3 else
                               n)
        # beta(x)
        self.b = lambda x, n: (n   if 0 < x <= self.m//3 else
                               n*2 if self.m//3 < x <= self.m*2//3 else
                               n+1)

    def search(self):
        if self.a_0 != None:
            a_0 = self.H(self.a_0)
        else:
            a_0 = self.H.random_element()

        (x_i, a_i, b_i) = (self.g**a_0, a_0, self.H(0))
        (x_2i, a_2i, b_2i) = (x_i, a_i, b_i)

        if self.debug:
            print "STEP 2:"
            print "(x_i, a_i, b_i) (x_2i, a_2i, b_2i)"

        while True:
            (x_i, a_i, b_i) = (self.f(x_i), self.a(x_i, a_i), self.b(x_i, b_i))
            (x_2i, a_2i, b_2i) = (self.f(self.f(x_2i)),
                                  self.a(self.f(x_2i), self.a(x_2i, a_2i)),
                                  self.b(self.f(x_2i), self.b(x_2i, b_2i)))

            if self.debug:
                print "(%s, %s, %s) (%s, %s, %s)" % (x_i, a_i, b_i, x_2i, a_2i, b_2i)

            if x_i == x_2i: break

        # step 2 ... finding the exponent
        if GCD(b_2i - b_i, self.m) == 1 and a_2i != a_i:
            (s, t) = (a_i - a_2i, b_2i - b_i)
            x = LCE(s, t, m-1, lambda x, y: mod(self.g, self.m)^x == self.y,
                    debug = self.debug)
        else: x = None

        if x == None:
            if self.debug:
                print "==== no useful values, restarting ===="
            return self.search()

        return x

y = 33; g = 2; m = 53
pr = PollardRhoDLPFloyd(m, g, y, a_0 = 0, debug = True)
time x = pr.search()
print "Solution: x = log_g(y) = %s\n" % x
```

Listing A.2: Pollards ρ-Methode für diskrete Logarithmen basierend auf Floyds Methode

Die Abbildung 3.2 wurde mittels des Programms `neato` erstellt, wobei die dafür notwendige Beschreibungsstruktur durch das Listing A.3 generiert wurde.

```
class NeatoPollardRhoStructure:
    def __init__(self, m, g, y, f_ls = None):
        self.m = m
        self.y = y
        self.G = IntegerModRing(m)
        self.g = self.G(g)

         # f(x)
        self.f = lambda x: (self.y*x  if 0 < x <= self.m//3 else
                            x**2 if self.m//3 < x <= self.m*2//3 else
                            self.g*x)

        self.dots = {}
        self.lists = []

    def generate(self):
        # generating all random walks over all starting points
        for s in range(1, m-1):
            ls = []; x = self.G(s)
            while x not in ls:
                tmp = self.f(x)
                ls.append(x)
                xs = "%s" % x
                if xs not in self.dots.keys():
                    self.dots[xs] = set()
                self.dots[xs].add("%s" % tmp)
                x = tmp
            self.lists.append(ls)
        return self

    def out(self):
        # neato print out
        sorted_keys = sorted(self.dots.keys(), key=lambda d: int(d))

        print "digraph G {"
        print "\tnode [width=0.3, height=0.3] { %s }" % " ".join(sorted_keys)
        for i in sorted_keys:
            print "\t%s -> { %s }" % (i, " ".join(self.dots[i]))
        print "\toverlap=false;\n}"

m = 53; g = 2; y = 33
nea = NeatoPollardRhoStructure(m, g, y)
nea.generate().out()
```

Listing A.3: Erzeugung der Beschreibungsstruktur für `neato` zur Darstellung der Irrfahrten von allen Startwerten

Das Listing A.4 beinhaltet die Parallele Kollisionssuche, angewendet auf die Pollard ρ-Methode aus Listing A.2. Dabei gelten folgende Einschränkungen:

- Keine Berücksichtigung von “Robin-Hoods”
- Nacheinander-Ausführung der Pfade zur vereinfachten Verwaltung der erfassten unterscheidbaren Punkte und der Protokollierung für ein Beispiel

```
from sage.crypto.util import least_significant_bits
def get_hamming_weight(n):
    n = Integer(n)
    b = n.binary()
    # binary representation
    n_b = least_significant_bits(n, len(b))
```

```
    h_n = hamming_weight(vector(GF(2), n_b))
    return h_n

class PollRhoParallel:
    def __init__(self, m, y, h):
        self.m = m; self.y = y
        self.G = IntegerModRing(m)
        self.g = self.G.unit_gens()[0]
        self.starts = []
        self.dpoints = {}
        self.collisions = []
        self.hamm = h

    def info(self):
        print "m = %s, g = %s, y = %s" % (self.m, self.g, self.y)
        print "Trails: ", len(self.dpoints)
        for i in self.dpoints.keys():
            (a, b) = self.dpoints[i]
            print "(x, a, b) =", (i, a, b)
        print "Collisions: ", len(self.collisions), self.collisions

    def f(self, x):
        return (self.y*x  if 0 < x <= self.m//3 else
                x**2 if self.m//3 < x <= self.m*2//3 else
                self.g*x) % self.m

    def a(self, x, n):
        return mod((n+1 if 0 < x <= self.m//3 else
                n*2 if self.m//3 < x <= self.m*2//3 else
                n), (self.m - 1))

    def b(self, x, n):
        return mod((n   if 0 < x <= self.m//3 else
                n*2 if self.m//3 < x <= self.m*2//3 else
                n+1), (self.m - 1))

    def genStart(self):
        a_0 = self.G.random_element()
        b_0 = self.G.random_element()
        x_0 = self.g^a_0 * self.y^b_0
        return (x_0, a_0, b_0)

    def distinguish(self, x):
        return get_hamming_weight(x) == self.hamm

    def runTrail(self):
        (x, a, b) = (x_0, a_0, b_0) = self.genStart()
        print (x, a, b)

        while True:
            (x, a, b) = (self.f(x), self.a(x, a), self.b(x, b))
            print (x, a, b)
            if self.distinguish(x): break;

        self.checkPoint((x, a, b), start=(x_0, a_0, b_0))
        print "------"

    def checkPoint(self, tripel, start=None):
        (x_0, a_0, b_0) = start
        (x_i, a_i, b_i) = tripel
        if x_i not in self.dpoints.keys():
            self.dpoints[x_i] = (a_i, b_i)
        elif x_i not in self.starts:
            (a_j, b_j) = self.dpoints[x_i]
            if mod(a_i, self.m) != mod(a_j, self.m):
```

```
                s = a_i - a_j
                t = b_j - b_i
                x = LCE(s, t, self.m-1, lambda x, y: mod(self.g, self.m)^x == self.y)
                if x:
                    self.collisions.append(x)
                    print "x_i: ", (x_i, a_i, b_i), "<-", (x_0, a_0, b_0)

    def searchDL(self):
        i = 0
        while not len(self.collisions):
            self.runTrail()
            i += 1
        return i
        #return self.collisions[0]

    def getCollision(self):
        return self.collision[0]

m = 53; g = 2; y = 33; h = 2
P = PollRhoParallel(m, y, h)
P.searchDL()
P.info()
```

Listing A.4: Parallele Kollisionssuche angewendet auf die Pollard ρ-Methode

Das Listing A.5 beinhaltet das von Edlyn Teske verwendete Verfahren zum Finden eines Zyklus, was in Kapitel 4.1 beschrieben wurde.

```
class PollardRhoDLP:
    def __init__(self, m, g, y, a_0 = None, debug = False):
        self.m = m
        self.G = IntegerModRing(m)
        self.H = IntegerModRing(m-1)
        self.g = self.G(g)
        self.y = self.G(y)
        self.a_0 = a_0
        self.debug = debug

        # storage for values
        self.C = []
        self.Ci = []
        self.Cn = 8

        # f(x)
        self.f = lambda x: (self.y*x  if 0 < x <= self.m//3 else
                            x**2 if self.m//3 < x <= self.m*2//3 else
                            self.g*x)
        # alpha(x)
        self.a = lambda x, n: (n+1 if 0 < x <= self.m//3 else
                               n*2 if self.m//3 < x <= self.m*2//3 else
                               n)
        # beta(x)
        self.b = lambda x, n: (n   if 0 < x <= self.m//3 else
                               n*2 if self.m//3 < x <= self.m*2//3 else
                               n+1)

    def getEntry(self, x):
        entry = None
        ls = filter(lambda c: c[0] == x, self.C)
        if len(ls) > 0:
            entry = ls[0]
        return entry
```

```
    def storeEntry(self, i, x, a, b):
        if len(self.C) == self.Cn:
            s1 = self.Ci[0]
            if i < 3*s1: return

        self.C.append([x, a, b])
        self.Ci.append(i)

        if len(self.C) > self.Cn:
            self.C.pop(0)
            self.Ci.pop(0)

    def search(self):
        if self.a_0 != None:
            a_0 = self.H(self.a_0)
        else:
            a_0 = self.H.random_element()
        (x_i, a_i, b_i) = (self.g**a_0, a_0, self.H(0))
        i = 0
        self.storeEntry(i, x_i, a_i, b_i)
        check = None

        if self.debug:
            print "STEP 2:"
            print "(x_i, a_i, b_i)"

        while True:
            if self.debug:
                print "(%s, %s, %s)" % (x_i, a_i, b_i)

            (x_i, a_i, b_i) = (self.f(x_i), self.a(x_i, a_i), self.b(x_i, b_i))
            i += 1
            check = self.getEntry(x_i)
            if check != None: break
            else: self.storeEntry(i, x_i, a_i, b_i)

        (x_j, a_j, b_j) = check
        if GCD(b_j - b_i, self.m) == 1 and a_j != a_i:
            (s, t) = (a_i - a_j, b_j - b_i)
            x = LCE(s, t, m-1, lambda x, y: mod(self.g, self.m)^x == self.y,
                    debug = self.debug)
        else: x = None

        if x == None:
            return self.search()

        return x

y = 33; g = 2; m = 53
pr = PollardRhoDLP(m, g, y, a_0 = 0, debug = True)
time x = pr.search()
print "Solution: x = log_g(y) = %s\n" % x
```

Listing A.5: Pollards ρ-Methode für diskrete Logarithmen basierend auf Schnorr und Lenstras Methode

Beide Abbildungen 4.2 und 4.3 wurden mit dem Programm `neato` erzeugt. Die dafür notwendige Beschreibungsstruktur wurde durch die Python-Klasse des Listings A.6 generiert.

```
class NeatoWalkStructure:
    def __init__(self, m, g, y, f_ls = None):
        self.m = m
        self.y = y
        self.G = IntegerModRing(m)
        self.g = self.G(g)
        self.generateGroup()

        if f_ls == None:
            self.f = [lambda x: x*self.g, lambda x: x*x, lambda x: x*self.y]
        else: self.f = f_ls
        self.F = []

    def generateGroup(self, debug=False):
        self.GM = []
        for i in range(0, self.m):
            x = self.g ** i
            if x not in self.GM: self.GM.append(x)
            else: break
        if debug: print self.g, sorted(self.GM), len(self.GM)

    def testGroup(self):
        for i in self.GM:
            self.g = self.G(i)
            self.generateGroup()

    def generate(self):
        for i, f in enumerate(self.f):
            self.F.append({})
            for x in self.GM:
                a = "%s" % x
                if a not in self.F[i].keys():
                    self.F[i][a] = set()
                self.F[i][a].add("%s" % f(self.G(x)))
        return self

    def out(self):
        # neato print out
        for W in self.F:
            sorted_keys = sorted(W.keys(), key=lambda d: int(d))

            print "digraph G {"
            print "\tnode [shape=circle, fixedsize=true, fontname=Arial] { %s }" % (
                  " ".join(sorted_keys))
            for i in sorted_keys:
                print "\t%s -> { %s }" % (i, " ".join(W[i]))
            print "\toverlap=false;\n}"

m = 23; y = 8; g = 2
nea = NeatoWalkStructure(m, g, y)
nea.generate().out()
```

Listing A.6: Erzeugung der Beschreibungsstruktur für `neato` zur Darstellung der Abfolge generiert durch die Iterationsfunktion

A.2 Notebook: Negation Map

Das Sage-Notebook “Negation Map” mit den Listings A.7 und A.8 wurde mit der URL `https://sage.hs-mittweida.de/home/pub/4/` erstmals am 12.07.2011 veröffentlicht.

Für die Python-Klasse des Listings A.7 gelten folgende Einschränkungen:

- Berechnung der Pfade bis zum Auffinden eines entscheidbaren Punktes.
- Keine Berechnung eines diskreten Logarithmus oder Speicherung der gefundenen unterscheidbaren Punkte.

Ziel der Implementierung war die Untersuchung der auftretenden $2z$-Zyklen und den damit verbundenen nötigem Verwaltungsaufwand zum Erkennen dieser Zyklen und zum Ausstieg aus diesen. Die Grundlage dazu bildete der fehlerhafte Algorithmus 1 aus [WZ11, S. 11].

```
from sage.crypto.util import least_significant_bits
def get_hamming_weight(n):
    n = Integer(n)
    b = n.binary()
    # binary representation
    n_b = least_significant_bits(n, len(b))
    h_n = hamming_weight(vector(GF(2), n_b))
    return h_n

class PRhoNegMap:
    def __init__(self, a, b, m, r, h, n, V, Y):
        self.a = a; self.b = b; self.m = m; self.r = r; self.hamm = h; self.n = n
        self.exp = m/binomial(ceil(log(m*1.0)/log(2.0)), 2)*20.0
        self.Z_m = IntegerModRing(m)
        self.E = EllipticCurve(self.Z_m, [a,b])
        self.V = self.E(V); self.Y = self.E(Y)
        self.X_0 = self.E.random_element()
        self.c = []; self.d = []; self.C = []
        self.h = lambda X_i: mod(X_i[0], self.r)
        self.eta = lambda X_i: self.E([X_i[0],
                     (self.Z_m(1) - self.Z_m(2)*self.Z_m(mod(X_i[1], 2))) * X_i[1]])
        self.f = lambda X_i: X_i + self.C[self.h(X_i)]
        self.f_eta = lambda X_i: self.eta(self.f(X_i))

        self.__generateC()

    def __generateC(self):
        for j in range(0, r):
            inserted = False
            while not inserted:
                c_j = self.Z_m.random_element()
                d_j = self.Z_m.random_element()
                C_j = int(c_j)*self.V + int(d_j)*self.Y
                if C_j not in self.C and C_j[0] != 0:
                    self.c.append(c_j)
                    self.d.append(d_j)
                    self.C.append(C_j)
                    inserted = True

    def min(self, P, Q):
        return P if P<Q else Q

    def distinguish(self, P):
        return get_hamming_weight(P[0]) == self.hamm and P[0] < 40

    def generateStart(self):
        X_0 = None
        a_0 = 0
        while a_0 == 0:
```

```
            a_0 = int(Z_m.random_element())
        X_0 = self.V * a_0
        return X_0

    def runTrailWithoutNegMap(self, X_0):
        X_i = X_0; i = 0

        while True:
            i += 1
            if self.distinguish(X_i): break;
            X_i = self.f(X_i)
            if i > self.exp:
                return None, None

        return X_i, i

    def runTrailWithNegMap(self, X_0):
        X_neu = None; X_e = None; X_min = None
        a = 0; c = 0
        X = [None for i in range(self.n + 1)]
        X[0] = X_0

        while True:
            a += 1
            if a > self.exp: return None, a, c
            if X_e != None:
                c += 1
                X[0] = 2*X_e
                X_e = None
                X_neu = None

            for j in range(1, self.n + 1):
                try:
                    X[j] = self.f_eta(X[j-1])
                except TypeError:
                    return None, a, c
                if self.distinguish(X[j]): return X[j], a, c

            #print "X = ", X
            X_neu = self.min(X[0], X[1])
            for i in range(2, self.n + 1):
                if X_neu == X[i] and not mod(i - X.index(X_neu), 2):
                    X_e = X_neu
                    #print "found cycle: ", i - X.index(X_neu)
                    break
                else:
                    X_neu = self.min(X_neu, X[i])
            #print "X_neu = ", X_neu

            if X_neu == X_min and not X_min == None:
                X_e = X_neu
            else:
                X[0] = X[-1]

    def test(self, n, useMap = False, showSteps = False):
        print "--- Starting Test", "with" if useMap else "without", "negative map"
        lens = []
        fails = 0
        runTrail = self.runTrailWithNegMap if useMap else self.runTrailWithoutNegMap
        for i in range(n):
            while True:
                X_0 = self.generateStart()
                t = runTrail(X_0)
                if t[0] != None: break
                else: fails += 1 #; print t
```

```
            if showSteps: print "- %s/%s: %s" % (i+1, n, t)
            lens.append(t[1])
        print "= avg:  ", sum(lens)*1.0/n
        print "= fails:", fails
```

Listing A.7: Anwendung der Inversabbildung auf Pollards ρ-Methode für diskrete Logarithmen basierend auf parallelen additiven Irrfahrten

Für die Python-Klasse des Listings A.7 wurden einige Tests durchgeführt, die auf dem folgenden Listing A.8 basieren.

```
m = 2^10 + 7
print "m:", m, is_prime(m)
Z_m = GF(m)
print "g:", Z_m.multiplicative_generator()
print "y:", Z_m.random_element()
a = Z_m.random_element(); b = Z_m.random_element()
E = EllipticCurve(Z_m, [a,b])
V = E.gen(0)
print "V:", V, V.order()
Y = E.random_point()
print "Y:", E.random_point()

r = 16; h = 2; n = 16; tests = 1
V = [V[0],V[1]]; Y = [Y[0],Y[1]]

nm = PRhoNegMap(a, b, m, r, h, n, V, Y)
nm.test(tests, useMap = True,  showSteps = True)
nm.test(tests, useMap = False, showSteps = True)
```

Listing A.8: Test der Anwendung der Inversabbildung auf Pollards ρ-Methode für diskrete Logarithmen basierend auf parallelen additiven Irrfahrten

Literaturverzeichnis

[BBB+11] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for Key Management – Part 1: General. URL: <http://csrc.nist.gov/groups/ST/toolkit/documents/SP800-57Part1-Revision3_May2011.pdf>, 2011. NIST Special Publication 800-57.

[BKK+09a] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, et al. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. http://eprint.iacr.org/.

[BKK+09b] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, et al. On the Security of 1024-bit RSA and 160-bit Elliptic Curve Cryptography. Cryptology ePrint Archive, Report 2009/389, 2009. http://eprint.iacr.org/.

[BKK+09c] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, et al. PlayStation 3 computing breaks 2^{60} barrier; 112-bit prime ECDLP solved. URL: <http://lacal.epfl.ch/112bit_prime>, 2009.

[BKK+10] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, et al. On the Use of the Negation Map in the Pollard Rho Method. In *Algorithmic Number Theory*, volume 6197 of *Lecture Notes in Computer Science*, pages 66–82. Springer Berlin / Heidelberg, 2010.

[BKK+11a] Joppe W. Bos, Thorsten Kleinjung, Marcelo E. Kaihara, et al. Breaking ECC2K-130. URL: <http://ecc-challenge.info/>, 2011. zuletzt verfügbar am 14.07.2011.

[BKK+11b] Joppe W. Bos, Thorsten Kleinjung, Marcelo E. Kaihara, et al. Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction. To appear in The International Journal of Applied Cryptography, 2011.

[BLS11] Daniel Bernstein, Tanja Lange, and Peter Schwabe. On the Correct Use of the Negation Map in the Pollard rho Method. In *Public Key Cryptography – PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 128–146. Springer Berlin / Heidelberg, 2011.

[Bre90] R. P. Brent. Parallel algorithms for integer factorization. *London Mathematical Society Lecture Note Series vol. 154, Number Theory and Cryptography, J.H. Loxton (ed.)*, pages 26–37, 1990.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. Cryptology ePrint Archive, Report 2011/232, 2011. http://eprint.iacr.org/.

[Bun08] Peter Bundschuh. *Einführung in die Zahlentheorie*. Springer, 2008.

[Bun11] Bundesnetzagentur. Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung. *Bundesanzeiger*, 85:2034–2048, 2011. URL: <http://www.bundesnetzagentur.de/cln_1911/DE/Sachgebiete/QES/Veroeffentlichungen/Algorithmen/algorithmen_node.html>.

[Cer97] Certicom Research. Certicom ECC Challenge. URL: <http://www.certicom.com/images/pdfs/challenge-2009.pdf>, 1997. Last update Nov. 2009.

[Cer00] Certicom Research. SEC 2: Recommended elliptic curve domain parameters. URL: <http://www.secg.org/download/aid-784/sec2-v2.pdf>, 2000.

[CF03] Henri Cohen and Gerhard Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/Crc, 2003.

[DGM99] I. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. pages 103–121, 1999.

[ECR10] ECRYPT NoE. ECRYPT II Yearly Report on Algorithms and Key Lengths. URL: <http://www.ecrypt.eu.org/documents/D.SPA.13.pdf>, 2010. ICT-2007-216676.

[FBB+10] J. Fan, D. V. Bailey, L. Batina, T. Güneysu, C. Paar, and I. Verbauwhede. Breaking Elliptic Curve Cryptosystems using Reconfigurable Hardware. *IEEE Field Programmable Logic and Applications (FPL)*, pages 133–138, 2010.

[Flo67] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14:636–644, 1967.

[FNI10] FNISA. Référentiel Général de Sécurité, Annexe B1. URL: <http://www.ssi.gouv.fr/IMG/pdf/RGS_B_1.pdf>, 2010. Version 1.20 du 26 janvier.

[GLVC98] Robert Gallant, Robert Lambert, Scott Vanstone, and Certicom Corp.

Improving the parallelized pollard lambda search on binary anomalous curves. *Mathematics of Computation*, 69:1699–1705, 1998.

[GPP06] T. Güneysu, C. Paar, and J. Pelzl. On the Security of Elliptic Curve Cryptosystems against Attacks with Special-Purpose Hardware. In Workshop on Special-purpose Hardware for Attacking Cryptograhic Systems - SHARCS, 2006.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[Hof09] Dirk W. Hoffmann. *Theoretische Informatik*. Hanser, 2009.

[KK10] Christian Karpfinger and Hubert Kiechle. *Kryptologie: Algebraische Methoden und Algorithmen*. Vieweg+Teubner, 2010.

[KM09] Christian Karpfinger and Kurt Meyberg. *Algebra*. Spektrum, 2009.

[Knu81] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1981.

[Lau99] Thomas Laubrock. Krypto-Verfahren basierend auf elliptischen Kurven - HTML-Tutorial mit Java™-Applet-, 1999. URL: <http://www.elliptische-kurven.de>, verfügbar am 12.05.2011.

[OW96] Paul C. Van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12:1–28, 1996.

[PH78] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE-Transactions on Information Theory*, 24:106–110, 1978.

[Pol75] J. M. Pollard. A monte carlo method for factorization. *BIT*, 15:331–334, 1975.

[Pol78] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Math. Comp.*, 32:918–924, 1978.

[Sha71] D. Shanks. Class number, a theory of factorization and genera. in proc. symp. pure math. *AMS, Providence, R.I.*, 20:415–440, 1971.

[Sil86] J. H. Silverman. *The arithmetic of elliptic curves*. Springer-Verlag, 1986.

[SL84] C. P. Schnorr and H. W. Lenstra. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43:289–311, 1984.

[SSW03] Jr. Samuel S. Wagstaff. *Cryptanalysis of Number Theoretic Ciphers*. Chapma & Hall/CRC, 2003.

[Tes98] Edlyn Teske. Speeding up Pollard's Rho Method for Computing Discrete Logarithms. pages 541–554, 1998.

[Tes00] Edlyn Teske. On Random Walks For Pollard's Rho Method. *Mathematics of Computation*, 70:809–825, 2000.

[Was03] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/Crc, 2003.

[WZ98] Michael J. Wiener and Robert J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. pages 190–200, 1998.

[WZ11] Ping Wang and Fangguo Zhang. Computing Elliptic Curve Discrete Logarithms with the Negation Map. Cryptology ePrint Archive, Report 2011/008, 2011. `http://eprint.iacr.org/`.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 05.08.2011