Unterschrift des Betreuers

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

D I P L O M A R B E I T

Attacken auf Public-Key-Kryptosysteme und ihre Implementierung in Maple

Ausgeführt am Institut für

Diskrete Mathematik und Geometrie

der Technischen Universität Wien

unter der Anleitung von Prof. Dr. Johann Wiesenbauer

durch

Roman Sonnenschein
Höfl 10
3251 Purgstall an der Erlauf

11. November 2011

Unterschrift (Student)

Vorwort

Ich möchte mich an dieser Stelle bei all jenen bedanken, die mich im Laufe meines Studiums unterstützt haben:

Als Erstes möchte ich mich bei meiner Familie bedanken, in der ich aufwachsen durfte, allen voran bei meinen Eltern, die mich in meinem Ausbildungsweg immer unterstützt haben. Ganz besonders aber möchte ich mich bei meiner lieben Frau Elisabeth bedanken, die mich im Laufe des Studiums unterstützt hat und während der Erstellung der Diplomarbeit viele Stunden entbehren musste.

Weiters bedanken möchte ich mich bei Prof. Dr. Johann Wiesenbauer für den Vorschlag des Themas der vorliegenden Arbeit sowie deren Betreuung.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Schon seit der Antike beschäftigen sich Mathematiker mit der Frage, wie man aus einer gegebenen natürlichen Zahl die zugehörige Primfaktorzerlegung bestimmen kann. Und selbst das heutzutage bekannteste Public-Key-Kryptosystem RSA hat mit dieser jahrtausende alten Frage zu tun, sind doch die mit dieser Fragestellung einhergehenden Schwierigkeiten bis heute nicht überwunden und somit zugleich auch Grundlage für RSA.

1.1 Gliederung der Arbeit

Nachfolgend darf ich einen kurzen Überblick über den Aufbau der vorliegenden Arbeit geben:

Nach einer kurzen, allgemeinen Begriffsdefinition für Kryptographie in *Kapitel 2* werden die beiden am weitesten verbreiteten öffentlichen Verschlüsselungsverfahren samt möglicher Angriffe behandelt:

In *Kapitel 3* werden neben dem Verfahren zur Ver- und Entschlüsselung mit RSA auch die vielen möglichen Angriffe darauf vorgestellt. Ein größerer Abschnitt dieses Kapitels ist dem Zahlkörpersieb gewidmet, dem momentan schnellsten Faktorisierungsverfahren, um RSA zu brechen.

Ein Vorteil von ElGamal besteht darin, dass man dieses Kryptosystem über verschiedenen Gruppen verwenden kann, weshalb man mit ElGamal eine gute Alternative zu RSA gefunden hat. Neben einer allgemeinen Beschreibung des Verschlüsselungsverfahrens an sich werden in *Kapitel 4* auch Möglichkeiten zur Berechnung diskreter Logarithmen erklärt.

In *Kapitel 5* finden sich ein paar Bemerkungen zur Implementierung des Zahlkörpersiebes, einer Attacke auf RSA, sowie der Index-calculus-Methode, einem Angriff auf ElGamal, in der Mathematiksoftware Maple wider.

Anhang A enthält den Code meiner Implementierung des Zahlkörpersiebes und Anhang B jenen der Index-calculus-Methode.

Kapitel 2

Kryptographische Grundlagen

In der Literatur herrscht bezüglich der Bezeichnungen für die gängigsten Begriffe in der Kryptographie weitestgehend Übereinstimmung. Der Vollständigkeit halber und um Verwechslungen zu vermeiden wollen wir an dieser Stelle trotzdem die wichtigsten Begriffe definieren.

2.1 Allgemeine kryptographische Begriffsdefinitionen

Im Folgenden werden wir davon ausgehen, dass die Information, die ein Sender in verschlüsselter Form dem Empfänger zukommen lassen möchte, schon in einer Form vorliegt, die man leicht in etwas umwandeln kann, sodass die beschriebenen kryptographischen Verfahren anwendbar sind. Da wir den Vorgang des Verschlüsselns als das Anwenden einer Abbildung betrachten wollen, heißt diese Forderung, dass die Information als ein Element des Urbildraumes angesehen wird, das wir dann als *Klartext* bzw. als *Nachricht* M bezeichnen wollen, wobei zwecks sprachlicher Vielfalt mit der Bezeichnung *Nachricht* manchmal auch die dahinterliegende Information gemeint sein kann.
Das Ergebnis der Abbildung eines Klartextes M aus dem Raum aller möglichen Nachrichten $\mathcal{M}$ wollen wir als *Chiffrat*, manchmal auch als *Geheimtext* bezeichnen und liegt im Raum aller möglichen Chiffrate $\mathcal{C}$. Den Vorgang des Verschlüsselns nennen wir auch *chiffrieren* und das befugte entschlüsseln *dechiffrieren*, wohingegen wir das unbefugte entschlüsseln *entziffern* oder auch *brechen* nennen wollen.
Ein Verschlüsselungsverfahren wollen wir also als eine Abbildung $E_K : \mathcal{M} \to \mathcal{C}$ betrachten, wobei erst der *Schlüssel* K festlegt, wie das Chiffrat C eines Klartextes M konkret aussieht. Das Entschlüsseln wird folglich durch die Funktion $D_K : \mathcal{C} \to \mathcal{M}$ repräsentiert.

Damit der Empfänger einer verschlüsselten Nachricht wieder *eindeutig* die ursprüngliche Nachricht erhält muss

$$D_K\big(E_K(M)\big) \;=\; M \qquad \forall\, M \in \mathcal{M}$$

gelten, dh. die Verschlüsselungsfunktion muss *injektiv* sein. Um ein Verschlüsselungssystem darüber hinaus auch zur Erstellung von Signaturen verwenden zu können, muss zusätzlich auch die Bedingung

$$E_K\big(D_K(M)\big) \;=\; M \qquad \forall\, M \in \mathcal{M}$$

erfüllt sein.

Wie allgemein im Bereich der Kryptographie üblich so wollen auch wir *Alice* und *Bob* stellvertretend für 2 Personen verwenden, die sicher miteinander kommunizieren wollen, und zwar in der Form, dass Bob an Alice eine Nachricht senden möchte.
Um ein klein wenig die Art bzw. den Aufwand eines Angriffs auf eine verschlüsselte Übertragung wider zu spiegeln, unterscheiden wir bei den Angreifern in zwei Kategorien: *Eve* (vom

Englischen „eavesdrop“: lauschen) stellt eine Angreiferin dar, die nur die Kommunikation zwischen Alice und Bob heimlich abhört, selbst aber nicht aktiv in diese eingreift. Dahingegen steht *Mallory* (in Anlehnung an das englische Wort „malicious“: boshaft) für einen Angreifer, der nicht nur abhört, sondern zB. auch (verschlüsselte) Nachrichten abfängt, selbst Nachrichten verschickt, einzelne Bits bei der Übertragung oder am PC ändert, oder Ähnliches.

2.2 Public-Key-Kryptographie

Die wesentliche Neuerung durch Public-Key-Kryptographie war jene, dass es für zwei Teilnehmer möglich wurde, sicher miteinander zu kommunizieren, obwohl sie bislang keinen gemeinsamen, geheimen Schlüssel ausgetauscht hatten. Diese bisherige Notwendigkeit verhinderte, dass spontane Kommunikation stattfinden konnte. (zB: Jemand surft im Internet, entdeckt eine Homepage und möchte dort etwas einkaufen, wozu er seine Bankdaten bekannt geben muss.)
Während im Falle der klassischen, symmetrischen Verschlüsselung quasi nur *ein* Schlüssel existiert, muss man bei asymmetrischen Kryptosystemen genauer sein und zwischen einem *öffentlichen Schlüssel* K_E zum *ver*schlüsseln und einem *privaten Schlüssel* K_D zum *ent*schlüsseln unterscheiden. Diese unterscheiden sich bei Public-Key-Kryptosystemen wesentlich voneinander. Deshalb darf der öffentliche Schlüssel allgemein bekannt bzw. soll sogar möglichst vielen zugänglich sein, damit jeder, der möchte, dem Empfänger eine Nachricht in verschlüsselter Form zukommen lassen kann. Der private Schlüssel hingegen enthält jene Information, mit der sich die ursprüngliche Nachricht relativ leicht wiederherstellen lässt. Dh. man spricht dann von einem *Schlüsselpaar* $K = \big(K_E, K_D\big)$. Die Bedingung, welche für ein asymmetrisches Kryptosystem ein korrektes Entschlüsseln sicher stellt, lautet dann

$$D_{(K_D)}\big(E_{(K_E)}(M)\big) \;=\; M \qquad \forall\, M \in \mathcal{M}\,.$$

Und wenn zusätzlich auch

$$E_{(K_E)}\big(D_{(K_D)}(M)\big) \;=\; M \qquad \forall\, M \in \mathcal{M}$$

gilt, dann kann man mit diesem Public-Key-Kryptosystem auch Signaturen erstellen.
Prinzipiell funktionieren fast alle Verschlüsselungen nach dem Prinzip, dass das Verschlüsseln und das rechtmäßige Entschlüsseln jeweils nur überschaubaren Aufwand verursachen. Ein Angreifer jedoch muss, um eine abgehörte Nachricht zu entziffern, dafür viel mehr Aufwand treiben als die beiden eigentlichen Kommunikationsteilnehmer. Deshalb ist ein wesentliches Kriterium bei Public-Key-Kryptographie, dass sich der private Schlüssel K_D nicht leicht aus den Informationen des öffentlichen Schlüssels K_E herleiten lässt. Und nur mit der zusätzlichen, geheimen Information darf es leicht sein, die Funktion $E_{(K_E)}$ umzukehren. Dadurch kann man die Funktion zum Chiffrieren ohne Bedenken öffentlich zugänglich machen.

Kapitel 3

RSA

RSA, das schon seit längerer Zeit bekannteste und auch am weitesten verbreitete Public-Key-Verschlüsselungsverfahren wurde 1977 erstmals veröffentlicht und ist nach seinen drei Erfindern

Ronald L. **R**ivest
Adi **S**hamir
Leonard M. **A**dleman

benannt. Im Jahr 1997 veröffentlichte das britische Government Communications Headquarters (GCHQ) eine Arbeit, die schon aus dem Jahr 1973 stammt und ein Public-Key-Kryptosystem vorstellt, das im Wesentlichen RSA gleicht [Sti06, Kap. 5.1, S. 161f]. RSA wird sowohl für die Verschlüsselung von Daten als auch für die Signaturerstellung verwendet. Durch seine große Bekanntheit und häufige Verwendung in vielen verschiedenen Implementierungen ist es auch eines der am besten untersuchten Public-Key-Kryptosysteme.
20 Jahre nach der Veröffentlichung durch Rivest, Shamir und Adleman hat Dan Boneh dieses Jubiläum als Anlass genommen, um die vielen bis dahin entdeckten Attacken auf RSA zusammenzufassen [Bon98]. Trotz einer Vielzahl von Angriffen war bisher kein einziger so wirkungsvoll, um das RSA-Verfahren zu brechen. Alle bisher bekannten Methoden zeigen bloß Schwachstellen mathematischer und menschlicher Natur auf, auf die man bei einer sicheren Implementierung und Verwendung von RSA achten muss.

3.1 Zahlentheoretische Grundlagen von RSA

Im Folgenden sollen nun die mathematischen Grundlagen, auf denen RSA beruht, vorgestellt bzw. deren Schreibweise definiert werden. Als zahlentheoretische Kernelemente von RSA sind unter anderem der euklidische Algorithmus, schnelles Potenzieren (im Restklassenring $\mathbb{Z}_n$), der chinesische Restsatz und der kleine Satz von Fermat zu nennen. Die für diese Arbeit wichtigsten Sätze bzw. Definitionen möchte ich nun anführen:

3.1.1 Restklassenring $\mathbb{Z}_n$

$\mathbb{Z}_n$ sei der *Restklassenring* $\mathbb{Z}/n\mathbb{Z}$ für $n \in \mathbb{N}$. Er besitzt die beiden kanonischen Repräsentantensysteme

$$\{0, 1, \ldots, n-1\} \qquad \text{und} \qquad \{1, 2, \ldots, n\},$$

wobei wir in der Regel Ersteres verwenden werden.
Alle $a \in \mathbb{Z}_n$ mit $ggT(a, n) = 1$ sind invertierbar. Da in $\mathbb{Z}_p$ für $p \in \mathbb{P}$ alle $a \in \mathbb{Z}_p \backslash \{0\}$ invertierbar sind, ist $\mathbb{Z}_p$ ein Körper.

3.1.2 Schnelles Potenzieren - repeated square & multiply

Zum Beispiel wird beim Verschlüsseln mit RSA eine 2048-Bit-Zahl M mit einem 2048-Bit-Exponenten e potenziert. Dh. der naive Ansatz, M einfach so oft mit sich selbst zu multiplizieren wie nötig, dh. ca. 10^{616} mal, würde alleine für die Verschlüsselung schon einen nicht bewältigbaren Aufwand erzeugen. Beim Entschlüsseln muss man ebenfalls eine Rechnung in ähnlicher Größenordnung ausführen. Deshalb ist es wichtig, dass man für dieses Problem einen guten Algorithmus zur Verfügung hat, mit dem sich der Aufwand in Grenzen halten lässt. Sehen wir uns dazu ein Beispiel an:

Beispiel 3.1
Wenn man a^{151} berechnen möchte, so kann man dies auch anschreiben als:

$$\begin{aligned} a^{151} &= a^{1+2+4+16+128} = a^1 \cdot a^2 \cdot a^4 \cdot a^{16} \cdot a^{128} = \\ &= a^{2^0} \cdot a^{2^1} \cdot a^{2^2} \cdot a^{2^4} \cdot a^{2^7} = a \cdot a^2 \cdot \left(a^2\right)^2 \cdot \left(\left(\left(a^2\right)^2\right)^2\right)^2 \cdot \left(\left(\left(\left(\left(\left(a^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2 \end{aligned}$$

□

Um diese Darstellung ausnützen zu können, nimmt man das ursprüngliche a her und quadriert es. Und das Ergebnis davon quadriert man wieder. Und dessen Ergebnis wird wiederum genauso quadriert, usw. ... . Dh. so erhält man eine Folge von Zahlen $a, a^2, \left(a^2\right)^2, ...$, wobei jede Zahl das Quadrat der vorhergehenden ist. Die wesentliche Idee des schnellen Potenzierens besteht darin, dass man sich - entsprechend der Binärdarstellung des Exponenten - aus dieser Folge bestimmte Werte herausnimmt und von diesen ausgewählten Werten dann das Produkt bildet und man so zum gewünschten Ergebnis kommt.

Beispiel 3.2 (Fortsetzung von Bsp. 3.1)
In unserem Beispiel, der Berechnung von a^{151}, ist das die erste, zweite, dritte, fünfte und achte Zahl dieser Folge, was auch genau der Binärdarstellung von $151 = (10010111)_2$ von rechts nach links gelesen entspricht. Abbildung 3.1 soll diese Idee verdeutlichen und in Algorithmus 3.3 verallgemeinert werden.

```
input  : Basis der Potenz: a
         Exponent e
output : Potenz A = a^e

A := 1
while e > 0 do
    if e ≡ 1 mod 2 then
        A := A · a
        e := e − 1
    end
    a := a^2
    e := e/2
end
return A
```

Algorithmus 3.3: schnelles Potenzieren - right-to-left (implizit)

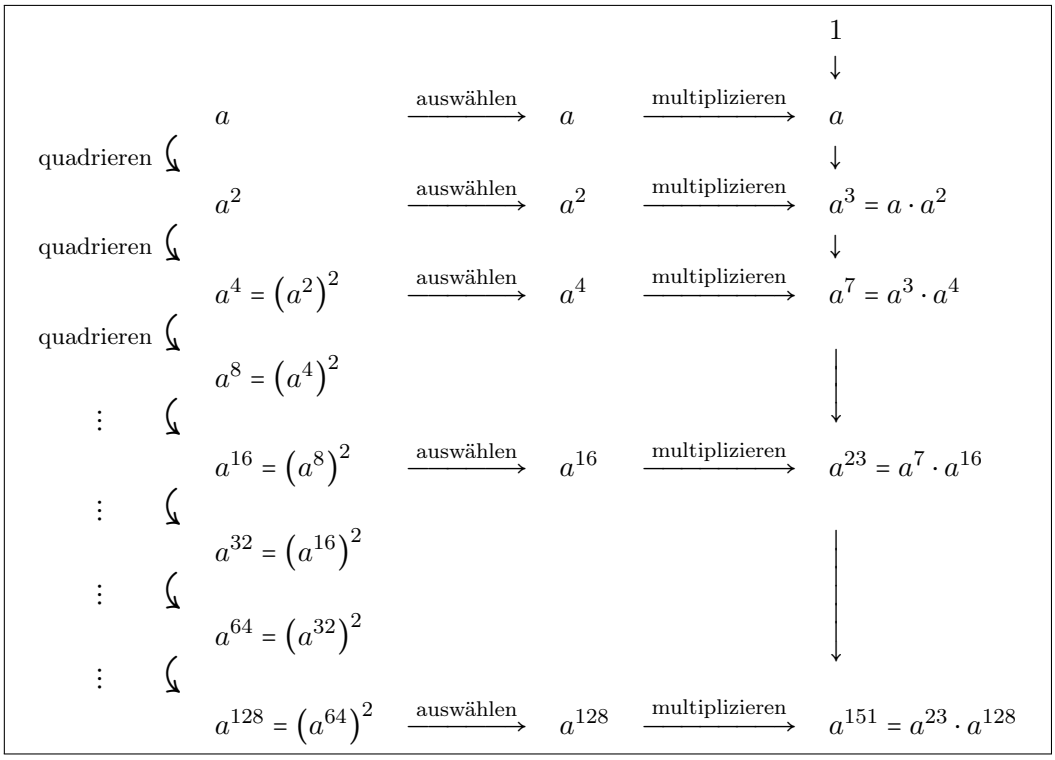


Abbildung 3.1: schnelles Potenzieren - repeated square & multiply

3.1.3 Schnelles Potenzieren im Restklassenring $\mathbb{Z}_n$

Beim schnellen Potenzieren in einem Restklassenring $\mathbb{Z}_n$ kann man aufgrund der Homomorphieeigenschaft $\varphi_n(a \cdot b) \equiv \varphi_n(a) \cdot \varphi_n(b) \mod n$ der Abbildung $\varphi_n : \mathbb{Z} \mapsto \mathbb{Z}_n$ während der Berechnung einer hohen Potenz auch zwischendurch immer wieder modulo n reduzieren. Dh. man kann nach jedem mal Quadrieren das neue Quadrat bzw. nach einer ggf. durchgeführten Multiplikation das Produkt modulo n reduzieren. Für nähere Informationen dazu siehe [Sti06, S.176ff] und [MvOV97, Kap. 2.4.4, S. 71f].

Da die Anwendung dieses Algorithmus' in kryptographischen Implementierungen in der so beschriebenen Art und Weise eine Angriffsfläche für side-channel-Attacken bietet, werden wir in Kapitel 4.1.3 Varianten davon kennen lernen, von denen manche diese Angriffe verhindern sollen.

3.1.4 Eulersche φ-Funktion

Die *eulersche φ-Funktion* ist definiert als

$$\varphi(n) \;:=\; \left|\left\{a \in \mathbb{Z}_n \;\middle|\; ggT(a,n) = 1\right\}\right| \qquad \forall n \in \mathbb{N}\,.$$

Für $n = p \in \mathbb{P}$ folgt

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

und für den bei RSA zutreffenden Fall $n = p \cdot q$ mit $p, q \in \mathbb{P}$ bedeutet dies:

$$\varphi(n) \;=\; (p-1)(q-1)\,.$$

Für weitere Eigenschaften von φ siehe zB. [Sti06, S. 9f] oder [MvOV97, S. 65].

3.1.5 Lagrange, Euler, kleiner Fermat

Satz 3.4 (Satz von Lagrange)
Sei G eine endliche Gruppe der Größe $n := |G|$. Dann ist die Ordnung jeder Untergruppe U von G ein Teiler der Gruppenordnung. Dh. es gilt

$$\forall U \trianglelefteq G : \ |U| \Big| |G| .$$

Beweis
siehe zB: [Buc10, S. 37, Theorem 3.10.9] □

Korollar 3.5 (Satz von Euler)
Sei $a \in {\mathbb{Z}_n}^* \subseteq \mathbb{Z}_n$ die Gruppe aller invertierbaren Elemente von $\mathbb{Z}_n$, dh. $ggT(a, n) = 1$ und sei weiters $n \geq 2$. Dann gilt

$$a^{\varphi(n)} \equiv 1 \bmod N .$$

Beweis
siehe zB: [Sti06, S. 170]: Die Ordnung der multiplikativen Gruppe ${\mathbb{Z}_N}^*$ beträgt $\varphi(N)$. □

Korollar 3.6 (kleiner Satz von Fermat)
Sei $a \in \mathbb{Z}_p$. Dann gilt

$$a^p \equiv a \bmod p .$$

Beweis
Falls $a \equiv 0$ ist, dann ist $0^p \equiv 0$ trivialerweise erfüllt, da in diesem Fall sogar echte Gleichheit herrscht.
Falls $a \not\equiv 0$ ist, dann sind die Voraussetzungen vom Satz von Euler (3.5) erfüllt, und wir erhalten: $a^{\varphi(p)} \equiv 1 \bmod p$. Da $p \in \mathbb{P}$ prim ist, folgt $\varphi(p) = p - 1$. Dh. wir erhalten die Kongruenz $a^{p-1} \equiv 1 \bmod p$, was wir auf beiden Seiten mit a multiplizieren und so den kleinen Satz von Fermat erhalten. □

3.1.6 Euklidischer Algorithmus

In seiner Grundversion berechnet der *euklidische Algorithmus* für zwei Eingabewerte $a, b \in \mathbb{Z}$ den größten gemeinsamen Teiler $ggT(a, b)$. Eine kurze und sehr elegante Implementierung findet sich in [MvOV97, Alg. 2.104, S. 66] und wird durch Algorithmus 3.7 bewerkstelligt.

```
input  : a, b ∈ ℕ mit a ≥ b
output : d = ggT(a, b)

while b ≠ 0 do
    r := (a mod b)
    a := b
    b := r
end
return a
```

Algorithmus 3.7: Euklidischer Algorithmus

Eine etwas ausführlichere Version, die auch die im Laufe der Berechnung des ggT auftretenden Quotienten mit zurück liefert, kann man in [Sti06, S. 164] nachschlagen.

Aufruf : $EEA(a,b)$
input : $a,b \in \mathbb{N}$ mit $a \geq b$
output : Trippel (d,x,y) mit $d = ggT(a,b)$ und $a\,x + b\,y = d$

if $b = 0$ **then**
  $d := a,\ \ x := 1,\ \ y := 0$
  return (d,x,y)
end
$x_2 := 1,\ \ x_1 := 0,\ \ y_2 := 0,\ \ y_1 := 1$
while $b > 0$ **do**
  $q := \left\lfloor \frac{a}{b} \right\rfloor$

$$\left.\begin{array}{rcl} r & := & a - q\,b \\ x & := & x_2 - q\,x_1 \\ y & := & y_2 - q\,y_1 \end{array}\right\}$$

oder vektoriell geschrieben: $\begin{pmatrix} r \\ x \\ y \end{pmatrix} = \begin{pmatrix} a \\ x_2 \\ y_2 \end{pmatrix} - q \cdot \begin{pmatrix} b \\ x_1 \\ y_1 \end{pmatrix}$

  $a := b,\ \ b := r$
  $x_2 := x_1,\ \ x_1 := x$
  $y_2 := y_1,\ \ y_1 := y$
end
$d := a,\ \ x := x_2,\ \ y := y_2$
return (d,x,y)

Algorithmus 3.8: Erweiterter Euklidischer Algorithmus (EEA)

3.1.7 Erweiterter Euklidischer Algorithmus (EEA)

Der *erweiterte Euklidische Algorithmus* berechnet für zwei Zahlen $a,b \in \mathbb{N}$ mit $a \geq b$ nicht nur den größten gemeinsamen Teiler $d = ggT(a,b)$ sondern auch zwei Werte $x,y \in \mathbb{Z}$ mit

$$a\,x + b\,y \;=\; d\,.$$

Dabei könnte man den Algorithmus 3.8 aus [MvOV97, Alg. 2.107, S. 67] bei passender Rechenumgebung auch in einer vektoriellen Variante ausführen, womit eine Verbesserung der Performance und der Lesbarkeit möglich werden würde.

3.1.8 Chinesischer Restsatz

Der *chinesische Restsatz* ist zwar für das Funktionieren von RSA nicht unbedingt erforderlich, aber in der Praxis kann durch dessen Anwendung beim Dechiffrieren Rechenzeit eingespart werden:

Satz 3.9 (chinesischer Restsatz)
Seien m_i für $i = 1,\ldots,r$ paarweise teilerfremde natürliche Zahlen mit $m_i \geq 2$ und seien $a_i \in \mathbb{Z}_{m_i}\ \ \forall i = 1,\ldots,r$. Dann besitzt das System der Kongruenzen

$$\begin{array}{rcll} x & \equiv & a_1 & \bmod m_1 \\ x & \equiv & a_2 & \bmod m_2 \\ \vdots & \vdots & \vdots & \\ x & \equiv & a_r & \bmod m_r \end{array}$$

für jede Auswahl $(a_1,a_2,\ldots,a_r)$ eine Lösung. Und diese Lösung ist modulo $M := \prod_{i=1}^{r} m_i$ sogar eindeutig [MvOV97, 2.120, S. 68].

Die Berechnung der eindeutigen Lösung $x \in \mathbb{Z}_M$ funktioniert mittels des Algorithmus' 3.10 - vgl. [Sti06, Kap. 5.2.2, S. 167-170] bzw. [KK10, Kapitel 7.2, S. 117-121].

input : paarweise teilerfremde $m_1, m_2, \ldots, m_r$
$(a_1, a_2, \ldots, a_r) \in \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_r}$
output : x mit $x \equiv a_i \mod m_i \quad \forall i = 1, \ldots, r$

$M := \prod_{i=1}^{r} m_i$
for $i = 1$ **to** r **do**
  $M_i := \frac{M}{m_i}$; // $ggT(M_i, m_i) = 1$
  // Berechnung eines multiplikativ Inversen M_i^* zu M_i modulo m_i:
  if $M_i > m_i$ **then**
    $(d, x, y) := E_{rweiterter} E_{uklidischer} A_{lgorithmus}\,(M_i, m_i)$
    $M_i^* := x$
  else
    $(d, x, y) := E_{rweiterter} E_{uklidischer} A_{lgorithmus}\,(m_i, M_i)$
    $M_i^* := y$
  end
end
$x := \sum_{i=1}^{r} (a_i \cdot M_i^* \cdot M_i) \mod M$
return x

Algorithmus 3.10: chinesischer Restsatz

3.2 Funktionsweise von RSA

Wir wollen nun mithilfe der soeben definierten Begriffe beschreiben, wie RSA funktioniert. Nachfolgend wollen wir vorerst ganz grob umreißen, wie RSA abläuft. *Alice* wird uns dabei als Empfängerin einer Botschaft dienen, die von *Bob* verschlüsselt an sie gesandt werden soll.

3.2.1 Kurzbeschreibung von RSA

Als Erstes wählt Alice zwei Primzahlen $p, q \in P$ und bildet das Produkt $N := p \cdot q$ und $v := kgV(p-1, q-1)$. Zusätzlich wählt sie ein $e \in \{2, \ldots, N-1\}$ mit $ggT\big(e, v\big) = 1$. Dann berechnet sie sich mittels des erweiterten euklidischen Algorithmus' ein d mit $d\,e \equiv 1 \mod v$. Wenn Bob die Nachricht $M \in \mathbb{Z}_N$ an Alice senden möchte, dann schickt er ihr $C := M^e \mod N$. Alice bildet $C^d \equiv M \mod N$ und erhält damit Bobs Nachricht M.

Diese äußerst verknappte und *unvollständige* Beschreibung von RSA soll nur die Grundidee von RSA vermitteln. Die einzelnen Schritte wollen wir im Folgenden detailierter ausführen.

3.2.2 Schlüsselerzeugung

Bei der Erzeugung des öffentlichen und des privaten Schlüssels für RSA gibt es einige Punkte, auf die man aufpassen muss, damit das Verschlüsselungsverfahren auch tatsächlich sicher ist.

Zu Beginn muss Alice die Stelligkeit der beiden Primzahlen p und q festlegen. Dabei sollten sich die beiden Zahlen um ein paar Stellen unterscheiden oder, wenn sie schon die gleiche Stellenanzahl besitzen, dann müssen sich p und q an zumindest der letzten Hälfte der Stellen plus ein paar Stellen darüber hinaus unterscheiden (siehe erste Fußnote in Kap. 3.3.8).
Zumeist wird aber nur von der Bitlänge des Moduls $N = p\,q$ gesprochen, und nicht von jener der beiden Primfaktoren.

Die deutsche Bundesnetzagentur veröffentlicht regelmäßig die als geeignet eingestuften Para-

meter für Verschlüsselungen.[1] Darunter finden sich auch die Parameter für RSA wieder. Im Moment[2] wird dort eine Schlüssellänge für N von mindestens 2048 Bit empfohlen.

Nachdem nun die Stelligkeiten feststehen, wählt Alice zufällig eine Primzahl mit passender Stellenanzahl. Sehr wesentlich dabei ist, dass die beiden Primzahlen p und q nicht zu erraten bzw. nicht nachvollziehbar sind. So würde es zum Beispiel keinen Sinn machen irgendwelche systematisch erzeugbaren Primzahlen zu verwenden (zB. Fermat- oder Mersenne-Primzahlen), denn ein Angreifer könnte ganz leicht alle in Frage kommenden Fermat- bzw. Mersenne-Primzahlen durchprobieren und würde in kürzester Zeit eine Faktorisierung in Händen halten. Genauso darf keine sonstige, schon bekannte Primzahl verwendet werden (zB. aus einer Liste der bisher größten Primzahlen, oder Ähnliches), denn solche Informationen stehen natürlich auch jedem Angreifer zur Verfügung. Diese beiden Primzahlen müssen also unbedingt selbst und zufällig, dh. nicht nachvollziehbar bzw. auch nicht erratbar erstellt werden.
Um einer Faktorisierung durch die Pollard'sche $p-1$-Methode zu entgehen, soll $p-1$ einen „großen" Primfaktor besitzen. Da in realistischen Anwendungen $p-1$ immer gerade ist, soll also $\frac{p-1}{2}$ einen möglichst großen Primfaktor besitzen. Am besten ist $q := \frac{p-1}{2}$ selbst eine Primzahl. q nennt man dann auch eine *Germain-Primzahl*. Um auch andere Attacken abwehren zu können, sollte man sog. starke Primzahlen verwenden [MvOV97, Kap. 4.2.2, S. 149 bzw. Note 8.8(iii), S. 291]:

Definition 3.11 (starke Primzahl)
Eine Primzahl p heißt eine *starke Primzahl*, wenn

- $p-1$ einen großen Primfaktor r besitzt,
- $p+1$ einen großen Primfaktor besitzt und
- $r-1$ einen großen Primfaktor besitzt.

Bemerkung 3.12 (großer Primfaktor)
Durchschnittlich können wir erwarten, dass die Stelligkeit des größten Primfaktors einer zufällig gewählten, natürlichen Zahl n ca. 63% der Stelligkeit von n beträgt. Die Formulierung, dass $p-1$ einen „großen Primfaktor" besitzen soll, meint, dass die Stelligkeit des größten Primfaktors von $p-1$ nicht wesentlich geringer als 63% sein soll.

Ein Algorithmus zur Erzeugung starker Primzahlen findet sich in [MvOV97, Alg. 4.53, S. 150].

Nachdem nun die beiden Primzahlen p und q gewählt sind, bilden wir das Produkt $N := p \cdot q$, womit wir den ersten Teil des öffentlichen Schlüssels berechnet haben. Sei v als $v := kgV(p-1, q-1)$ definiert. Dann ist der zweite Teil des öffentlichen Schlüssels, der sog. *öffentliche Exponent* eine Zahl

$$e \in \{2, \ldots, v-1\} \qquad \text{mit} \quad ggT(e, v) = 1\,.$$

Mit der Forderung $p \neq q$ muss mindestens eine der beiden Primzahlen ungerade sein. Deshalb ist $v = kgV(p-1, q-1)$ jedenfalls gerade. Somit fällt auch die Wahl $e = 2$ weg, genauso wie auch alle anderen Möglichkeiten e mit $e \equiv 0 \mod 2$ wegfallen. Auch die Wahl $e = v-1$ muss man vermeiden, da sonst wegen

$$C^e = C^{v-1} \equiv \left(m^{v-1}\right)^{v-1} = m^{(v-1)(v-1)} \equiv m^{(-1)(-1)} = m^1 = m \qquad \mod N$$

ein Angreifer auf ein abgehörtes Chiffrat C nur ein zweites Mal den öffentlichen Schlüssel anwenden braucht, um den Klartext zu erhalten.
Tatsächlich liegt also e in der Menge

$$e \in \{3, 5, 7, \ldots, v-5, v-3\}\,.$$

[1] http://www.bundesnetzagentur.de/DE/Sachgebiete/QES/Veroeffentlichungen/Algorithmen/algorithmen_node.html
[2] letzte Veröffentlichung: 20. Mai 2011

Als *öffentlichen Schlüssel* bezeichnen wir nun das Zahlenpaar

$$(n, e)\,.$$

Mittels des Erweiterten Euklidischen Algorithmus' (s. Alg. 3.8 in Kap. 3.1.7) können wir uns ein modulares Inverses von e modulo v berechnen: Der Aufruf $EEA(v, e)$ liefert als dritte Komponente, die wir mit d (für decryption exponent) bezeichnen wollen[3], die gesuchte Zahl mit

$$d\,e \;\equiv\; 1 \mod kgV(p-1, q-1)\,.$$

Als *privaten Schlüssel* bezeichnen wir das Trippel

$$(p, q, d)\,,$$

welches unbedingt geheim gehalten werden muss - nicht nur als Trippel, sondern jeder Wert einzeln für sich macht bei Bekanntwerden eine Verschlüsselung mit dem Modul N nutzlos (siehe Alg. 3.22).
Man könnte die beiden Werte p und q nach der Schlüsselerzeugung auch löschen, da man zum Entschlüsseln nur die beiden Werte (N, d) benötigt.

Bemerkung 3.13 ($(p-1)\cdot(q-1)$ vs. $kgV(p-1, q-1)$)
In den meisten Arbeiten (genauso wie in der ursprünglichen Originalarbeit [RSA78] auch) wird anstatt des $kgV(p-1, q-1)$ das Produkt $\varphi(N) = (p-1)\cdot(q-1)$ verwendet; sowohl bei der Grundmenge $\{2, \ldots, (p-1)(q-1)\}$, aus der e gewählt wird, als auch als Modul bei der Berechnung von d als modulares Inverses zu e. Das so erhaltene e kann natürlich *auch* zur Entschlüsselung hergenommen werden, es ist aber nicht das kleinst mögliche e, das verwendet werden kann [Wie99].

Für den sehr unwahrscheinlichen Fall, dass bei der Erzeugung der beiden Primfaktoren eine Zahl als prim angesehen wird, die tatsächlich aber keine Primzahl ist (*Charmichael-Zahl*), führen viele Implementierungen von RSA bei der Schlüsselerzeugung probehalber ein paar Verschlüsselungen und Entschlüsselungen durch [Pom03, Kap. 3.6, S. 52]. Sollte dabei ein Fehler auftreten, so wird dieses N verworfen und ein neues generiert.

Für die schnelle Variante des Entschlüsseln (siehe Kap. 3.2.4), bei der der chinesische Restsatz angewandt wird, kann man schon bei der Schlüsselerzeugung die Werte

$$d_p \;:=\; d \mod (p-1) \qquad \text{und} \qquad d_q \;:=\; d \mod (q-1)$$

einmalig vorausberechnen anstatt bei jeder Entschlüsselung immer wieder dieselben beiden Werte zu berechnen [KL08, S. 358].

Da es einen Angriff gibt, der eine Laufzeit von $\mathcal{O}\Big(\min\big(\sqrt{d_p}, \sqrt{d_q}\big)\Big)$ besitzt, dürfen die beiden Werte d_p und d_q nicht zu klein sein [Bon98, S. 5].

3.2.3 Verschlüsseln

Wenn Bob nun an Alice eine verschlüsselte Nachricht senden möchte, dann muss Bob wie folgt vorgehen:

- Als erstes muss Bob seine Nachricht, die er sende möchte, darstellen als eine Zahl

$$M \;\in\; \{2, \ldots,\; N-2\}\,.$$

[3] Im Zusammenhang mit RSA steht d international üblich für den *privaten Exponenten* (decryption exponent). Diese Bedeutung ist auch an dieser Stelle gemeint! Im Zusammenhang mit dem Erweiterten Euklidischen Algorithmus kann d manchmal auch für den größten gemeinsamen Teiler stehen.

(Die Restklassen $-1, 0, 1 \in \mathbb{Z}_N$ würden beim „Verschlüsseln“ mit einem ungeraden Exponenten e auf sich selbst abgebildet werden und somit wäre dann $C \equiv M$. Wenn die Nachricht so lang ist, dass ihre digitale Darstellung als Zahl größer als $N-2$ ist, dann muss Bob die Nachricht zuerst in mehrere Blöcke $M = (M_1, M_2, \ldots, M_r)$ unterteilen, sodass jeder Block M_i (für $i = 1 \ldots r$) einzeln darstellbar ist als eine Zahl zwischen 2 und $N-2$. In diesem Fall werden alle restlichen Verschlüsselungsschritte für jeden dieser r Blöcke einzeln ausgeführt und man erhält schlussendlich genauso viele Chiffratblöcke als Nachrichtenblöcke. Die so erhaltenen Chiffrate C_i werden dann zu einer Folge $C := (C_1, C_2, \ldots, C_r)$ zusammengefügt und als Ganzes an Alice geschickt.

- Um M resp. M_i so zu verschlüsseln, dass nur Alice die Nachricht wieder entschlüsseln kann, muss Bob Alices öffentlichen Schlüssel verwenden. Dh. er benötigt dazu ihre Werte (N, e).
- Nun berechnet Bob das Chiffrat $C := M^e \mod N$ resp. $C_i := {M_i}^e \mod N \ \forall\ i = 1, \ldots, r$, wozu er einen Algorithmus für schnelles Potenzieren im Restklassenring $\mathbb{Z}_N$ verwendet (siehe Kap. 3.1.3 bzw. Kap. 4.1.3).
- Abschließend sendet Bob C resp. $(C_1, C_2, \ldots, C_r)$ an Alice.

3.2.4 Entschlüsseln

Wenn Alice von Bob den Geheimtext C $\big($resp. $(C_1, C_2, \ldots, C_r)\big)$ empfangen hat, dann entschlüsselt Alice das Chiffrat durch Berechnen von $\widetilde{M} := C^d \mod N$, für das gilt: $\widetilde{M} \equiv M$, weshalb Alice Bobs Nachricht lesen kann.

Alice hat für die Berechnung von $C^d \mod N$ zwei Möglichkeiten:

1. Die einfachere Möglichkeit für Alice ist, genau denselben Algorithmus für das schnelle Potenzieren in $\mathbb{Z}_N$ zu verwenden wie Bob (siehe Kap. 3.1.3 bzw. Kap. 4.1.3). Es gibt aber eine schnellere Möglichkeit um zu genau demselben Ergebnis zu kommen:
2. Wenn man die Primfaktorzerlegung des Moduls N kennt, dann kann man die Berechnung der großen Potenz $C^d \mod N$ beschleunigen, indem man den chinesischen Restsatz (Kap. 3.1.8) anwendet.
   Dazu benötigt Alice die beiden Werte

$$d_p := d \mod (p-1) \qquad \text{und} \qquad d_q := d \mod (q-1)\,,$$

   die schon bei der Schlüsselerzeugung berechnet worden sein sollten. Sodann führt Alice die beiden Rechnungen

$$\widetilde{M}_p := C^{d_p} \mod p \qquad \text{und} \qquad \widetilde{M}_q := C^{d_q} \mod q$$

   aus und kombiniert die beiden Teilergebnisse $\widetilde{M}_p$ und $\widetilde{M}_q$ mittels des chinesischen Restsatzes (Alg. 3.10) zu einem $\widetilde{M} \in \mathbb{Z}_N$ mit

$$\begin{aligned} \widetilde{M} &\equiv \widetilde{M}_p \mod p \qquad \text{und} \\ \widetilde{M} &\equiv \widetilde{M}_q \mod q\,. \end{aligned}$$

   Für das so erhaltene $\widetilde{M}$ gilt

$$\left.\begin{aligned} \widetilde{M} &\equiv C^{d_p} \equiv C^d \mod p \\ \widetilde{M} &\equiv C^{d_q} \equiv C^d \mod q \end{aligned}\right\} \quad \Rightarrow \quad \widetilde{M} \equiv C^d \mod \underbrace{kgV(p,q)}_{=N}$$

   womit wir gezeigt haben, dass wir bei der Verwendung des chinesischen Restsatzen dasselbe Ergebnis erhalten, als ohne dessen Verwendung.

Für weitere Literatur zum chinesischen Restsatz und seiner Anwendung bei RSA siehe [KK10, Kap. 7.2, S. 117-121] bzw. [Sti06, S. 167ff].

3.2.5 Korrektheit von RSA

Dass $\widetilde{M} \equiv M$ gilt, dh. dass RSA tatsächlich wieder „richtig“ entschlüsselt, sichert uns der folgende Satz

Satz 3.14 (Korrektheit von RSA)
Seien p, q, N, e, d und v wie in Abschnitt 3.2.2 gewählt. Dann gilt:

$$\left(M^e\right)^d \equiv M \mod N \qquad \forall\ M \in \mathbb{Z}_N\,.$$

Beweis
Wegen $e\,d \equiv 1 \mod v$ gibt es ein $k \in \mathbb{Z}$ mit $e\,d + k\,v = 1$. Da v als $v = kgV(p-1, q-1)$ definiert ist, existieren geeignete $k_p, k_q \in \mathbb{Z}$ mit $e\,d + k_p \cdot (p-1) = 1$ und $e\,d + k_q \cdot (q-1) = 1$. Somit folgt aus Korollar 3.5 weiters:

$$\left.\begin{array}{rcl} \left(M^e\right)^d & = & M^{e\,d} \equiv M^{e\,d + k_p \cdot (p-1)} = M^1 = M \mod p \\ \left(M^e\right)^d & = & M^{e\,d} \equiv M^{e\,d + k_q \cdot (q-1)} = M^1 = M \mod q \end{array}\right\} \quad \Rightarrow \quad \left(M^e\right)^d \equiv M \mod \underbrace{kgV(p,q)}_{=N}$$

womit gezeigt ist, dass für eine korrekte Entschlüsselung nur die Bedingung $ggT(M, N) = 1$ erfüllt sein braucht, was in der Regel gelten wird.

□

3.3 Attacken auf RSA

Gleichzeitig mit der Möglichkeit der Verschlüsselung taucht auch die Frage auf, wie man als Angreifer auch ohne Kenntnis des (privaten) Schlüssels trotzdem zur Information M gelangen kann. Im Falle von RSA heißt das, wie man nur aus den Werten (N, e, C) auf den Klartext M schließen kann.
Die naheliegenste Möglickeit wäre, in $\mathbb{Z}_N$ die e-te Wurzel aus C zu ziehen. Dies ist zu Zeit jedoch noch immer ein nicht-triviales Problem, ansonsten wäre RSA nicht praxistauglich.
Auch die bekannteste Möglichkeit um RSA zu knacken - nämlich N zu faktorisieren, um sich analog der Schlüsselerzeugung den geheimen Schlüssel d zu berechnen - ist zwar schon ein altes, aber immer noch nicht effizient gelöstes Problem - v.a. für RSA-Moduli.

Sicherheit bei einem kryptographischen Verschlüsselungssystem heißt nicht nur, dass die Umkehrung der Verschlüsselungsfunktion - im Falls von RSA: $x \mapsto x^e \mod N$ - nicht in akzeptabler Zeit mit annehmbarem Aufwand möglich sein darf.
Darüber hinaus soll nicht nur der gesamte Klartext geheim bleiben, sondern ohne den geheimen Schlüssel sollte man überhaupt keine Informationen über den Klartext bekommen können. Diese Anforderung nennen wir *semantische Sicherheit*. So zum Beispiel kann man bei RSA ohne weiteres den Wert des Jacobi-Symbols $\left(\frac{M}{N}\right)$ berechnen. Dem kann man jedoch leicht durch Hinzufügen von Zufälligkeit entgegnen [Bon98, S. 2].

Die folgenden Attacken liefern Bedinungen, die für eine sichere Verwendung von RSA eingehalten werden müssen.

3.3.1 Enumeration

Wenn man nur eine begrenzte Zahl an möglichen Klartexten erwartet, ist eine der einfachsten Ideen, als Angreifer selbst all diese Klartexte M_i mit dem allseits bekannten öffentlichen

Schlüssel (N, e) zu verschlüsseln und zu vergleichen, ob eines der erhaltenen Chiffrate $C_i := {M_i}^e \mod N$ gleich dem abgehörten C ist. Dieser Angriffsmöglichkeit liegt zugrunde, dass RSA ein *deterministisches Verfahren* ist, dh. dass ein Klartext beim Verschlüsseln mit einem festen Schlüssel immer in genau ein und denselben Geheimtext umgewandelt wird.

Diese Vorgehensweise ist zum Beispiel erfolgsversprechend, wenn eine kurze Klartext-Nachricht erwartet wird. Dann ist die Anzahl der möglichen Nachrichten in gewisser Weise beschränkt, sagen wir durch $\mathcal{L}$.
Mit dem Wissen, dass die Nachricht M als Zahl dargestellt zwischen $1 < M < \mathcal{L}$ liegt, hat dieses Vorgehen lineare Laufzeit in $\mathcal{L}$. Oder anders gesagt: wenn l die (Binär-)Stelligkeit von $\mathcal{L}$ ausdrückt, bedeutet dies exponentielle Laufzeit in l.

Als sicherheitsbedachter Absender kann man diesem Angriff entgegenwirken, indem man einen randomisierten Teil an die Nachricht anfügt, der dann für den rechtmäßigen Empfänger auch klar als solcher erkennbar ist. Dadurch werden zum einen die kurzen Nachrichten verlängert, und zum anderen wird die Menge der zu erwartenden Klartexte vergrößert.
(Die Verwendung eines Zeitstempels als Padding ist meiner Meinung nach nicht ausreichend, denn wenn ein Angreifer die Nachricht während der Übertragung abhört, weiß er auch, wie spät es zu diesem Zeitpunkt war und so kann der Angreifer den Zeitstempel ziemlich gut einschränken. Darüber hinaus gibt es auch nur eine begrenzte Anzahl an Zeitformaten für den Zeitstempel.)

Für sehr kurze Nachrichten kommt aber auch noch der nächste Angriff in Frage:

3.3.2 reelle Wurzel ziehen

Für sehr kurze Nachrichten, die nicht gepaddet wurden - dh. die nicht an ausgemachter Stelle mit einer zufälligen Zeichenkette ergänzt wurden - könnte es passieren, dass beim Verschlüsseln der Fall

$$M^e < N$$

auftritt. In diesem Fall würde bei der Berechnung von $C \equiv M^e \mod N$ keine modulare Reduktion stattfinden und es würde gelten

$$C = M^e .$$

(Man beachte: aus '≡' wurde '=' !)
Eine Lauscherin Eve könnte in diesem Fall einfach durch ziehen der e-ten Wurzel von C im Reellen den Klartext berechnen:

$$M = \sqrt[e]{C} .$$

Deshalb sollte ein Sender einer mit RSA verschlüsselten Nachricht diese padden, sodass M^e den Wert des Moduls N übersteigt und somit auch tatsächlich eine Reduktion stattfindet. Dabei sollte der Sender, meiner Meinung nach, aber darauf achten, dass er nicht nur soviele Zeichen hinzufügt, um gerade noch eine modulare Reduktion zu erreichen. Denn Eve kann auch versuchen im Reellen die e-te Wurzel

$$\sqrt[e]{C + k \cdot N}$$

für $k \in \mathbb{N}$ unterhalb einer gewissen Schranke zu ziehen, wodurch ein „minimalistisches Padden“ gefährlich wäre.

Beispiel 3.15
Angenommen ein Klartextzeichen benötigt 8 Bit zur binären Darstellung. Sei weiters N ein 4096-Bit-Modul und der öffentliche Exponent $e = 3$. Damit würden beim Verschlüsseln erst

input : öffentlicher Schlüssel (N,e),
Chiffrat C,
Schranke $\mathcal{L} > M$,
α mit $\frac{1}{2} < \alpha < 1$
output : Klartext M

$T := \lceil \mathcal{L}^{\alpha} \rceil$
for $r = 1$ **to** T **do**
  $x_r := \frac{C}{r^e} \bmod N$
end
Sortieren der Paare (r, x_r) *für* $r=1 \ldots T$ *nach deren 2. Komponente*
for $s = 1$ **to** T **do**
  if s^e *mod* $N = x_r$ *für ein passendes* r **then**
    return $r \cdot s \bmod N$
  end
end

Algorithmus 3.16: Enumeration bis zur Wurzel von M

Nachrichten ab einer Länge von 171 Zeichen zu einer modularen Reduktion führen.

□

Wenn m die Stelligkeit von M, und n die Stelligkeit von N bezeichnet, kann man allgemein sagen, dass dieser Angriff Erfolg hat, wenn

$$M < N^{\frac{1}{e}} \qquad \Leftrightarrow \qquad m < \frac{n}{e}$$

gilt. Die zweite Ungleichung gilt sowohl für Dezimalstelligkeit als auch für Binärstelligkeit.

3.3.3 Enumeration - quadratische Verbesserung

In der Praxis werden oft auch *hybride Verschlüsselungssysteme* eingesetzt. Dh. man verwendet ein asymmetrisches Verschlüsselungssystem, wie zB. RSA, um den Schlüssel für ein symmetrisches Kryptosystem zu übermittlen, da diese oftmals schneller sind. Stellen wir uns nun folgendes Szenario vor: Wir sind ein Angreifer und wissen, dass mit der von uns abgehörten Chiffre C ein Klartext M mit einer Bit-Länge (höchstens) l übertragen wird (zB. ein 128-Bit-Key für das symmentrische Verschlüsselungsverfahren AES). Dh. für den Klartext M (=Schlüssel des symm. Verfahren) gilt: $M < \mathcal{L} := 2^l$, wobei $\mathcal{L}$ eine uns bekannte, möglichst scharfe obere Schranke für den gesuchten Klartext ist und l deren Binärstelligkeit.

Der Algorithmus 3.16 [KL08, S. 360f] liefert mit hoher Wahrscheinlichkeit den Klartext M als Ergebnis.

Eine wesentliche Idee, die wir dabei ausnützen möchten, ist, dass es für eine zufällige Zahl M sehr wahrscheinlich ist, dass sie keine Primzahl, sondern zusammengesetzt ist.
Nehmen wir an, dass M *keine* Primzahl ist. Dann lässt sich M anschreiben als $M = r \cdot s$ mit $r, s < M$. Wir als Angreifer kennen zwar weder M noch die beiden Faktoren r und s, aber wir wissen, dass es sie gibt. Also wissen wir, dass gilt: $s = \frac{M}{r}$. Daraus folgt nun aber, dass $s^e = \frac{M^e}{r^e} \equiv \frac{C}{r^e} \bmod N$ ist.
In einem ersten Schritt läuft r in Bereich $[1, T]_{\mathbb{N}}$. Dabei berechnen wir uns jeweils den Wert $\frac{C}{r^e}$ und speichern das Paar $\left(\frac{C}{r^e}, r\right)$ ab. Nach diesem Durchlauf sortieren wir die gespeicherten Paare nach deren 1. Komponente. Anschließend läuft s ebenfalls in $[1, T]_{\mathbb{N}}$. Hierbei berechnen wir jeweils den Wert s^e und suchen, ob dieser Wert als 1. Komponente in der sortierten Liste vorkommt. Fall ja, dann haben wir mit s und der zweiten Komponente r zwei Werte gefunden,

die $r \cdot s = M$ erfüllen, womit wir den Klartext kennen.

Diese Vorgehensweise führt zum Erfolg, wenn *beide* Faktoren r und s von $M = r\,s$ kleiner als T sind. Je größer α ist, desto wahrscheinlicher ist es, dass es zwei Faktoren r und s mit $r\,s = M$ und $r, s \leq T$ gibt.
An dieser Stelle kommt die Tatsache zu tragen, dass die Stelligkeit des größten Primfaktors einer zufälligen Zahl im Durchschnitt ungefähr 63% beträgt. Mit dem Parameter α kann man somit die Erfolgswahrscheinlichkeit und zugleich natürlich auch die Laufzeit des Algorithmus' steuern.

Wesentliche Voraussetzung für diesen Angriff ist, dass der Klartext M dabei *keine* Primzahl ist. Dass die Wahrscheinlichkeit für eine zufällige Zahl, keine Primzahl zu sein, hoch ist, sagt uns der Primzahlsatz [CP00, Th. 1.1.4, S. 9]:

Satz 3.17 (Primzahlsatz)
Sei mit $\pi(x)$ die Anzahl der Primzahlen kleiner gleich x definiert:

$$\pi(x) \;=\; \left|\left\{p \leq x \;\middle|\; p \in \mathbb{P}\right\}\right| .$$

Dann verhält sich $\pi(x)$ wie

$$\pi(x) \;\sim\; \frac{x}{\ln x} \qquad \text{für } x \to \infty ,$$

dh. $\lim\limits_{x\to\infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1$.

Korollar 3.18 (Primzahldichte)
Aus dem Primzahlsatz 3.17 erhalten wir, dass sich die *Primzahldichte* wie

$$\frac{\pi(x)}{x} \;\sim\; \frac{1}{\ln x} \qquad \text{für } x \to \infty$$

verhält.

Dh. für unser Beispiel, dass ein zufälliger 128-Bit-Schlüssel mit ca. 98,9%-iger Wahrscheinlichkeit keine Primzahl ist.
Somit liefert also Algorithmus 3.16, abhängig von der Wahl von α, auch tatsächlich den Klartext M. Nur für den Fall, dass $M \in \mathbb{P}$ eine Primzahl ist, oder α zu klein gewählt wurde, führt dieser Angriff nicht zum gewünschten Erfolg.

Bemerkung 3.19

- Sollte beim Bilden des Inversen von r mod N (in der ersten for-Schleife) ein Fehler auftreten, weil es nicht existiert, würde dies sofort zu einer Faktorisierung von N führen und wir hielten nicht nur diesen einen Klartext M sondern sogar eine Faktoriserung von N in Händen.

- Der Aufwand des Algorithmus 3.16 setzt sich zusammen aus den beiden for-Schleife mit einer Laufzeit von jeweils $\mathcal{O}(T) = \mathcal{O}(2^{\frac{l}{2}})$ und dem Sortieren der Paare (r, x_r), was mit einer Laufzeit von $\mathcal{O}(T \cdot \log T) = \mathcal{O}(2^{\frac{l}{2}} \cdot \log 2^{\frac{l}{2}}) = \mathcal{O}(2^{\frac{l}{2}} \cdot \frac{l}{2} \cdot \log 2) = \mathcal{O}(l \cdot 2^{\frac{l}{2}})$ den dominanten Teil darstellt. Für das Suchen in der zweiten for-Schleife verwenden wir binäres Suchen.

Diese Angriffsmöglichkeit ist ein weiterer Grund dafür, dass man eine zu sendende Nachricht mit zufälligen Bits padden soll.

3.3.4 gemeinsamer Modul I - Simmons

Der folgende Angriff [Bon98, S. 3-4] stammt von Simmons:

Eine Schlüsselverteilzentrale oder auch eine größere Organisation wie zB. eine Firma könnte auf die Idee kommen, nicht für jeden Mitarbeiter ein eigenes Paar Primzahlen (p, q) zu erzeugen, sondern, um sich Aufwand zu ersparen, einfach einen einzigen Modul N zu bestimmen und jedem Mitarbeiter nur ein anderes Schlüsseltrippel (N, e_i, d_i) zuzuweisen. Diese vordergründige Arbeitsersparnis führt aber zu einem Sicherheitsrisiko, wie wir noch sehen werden. Jeder Mitarbeiter bekommt ein eigenes Exponentenpaar (e_i, d_i) zugewiesen und auf den ersten Blick sieht es so aus, als könnte kein Mitarbeiter jene Nachrichten, die an einen anderen Mitarbeiter gerichtet sind, entschlüsseln.[4] Tatsächlich aber kann man bei Kenntnis eines Paares (e, d) den Modul $N = p \cdot q$ faktorisieren und dann wie bei der normalen Schlüsselerzeugung mithilfe von p und q ein modulares Inverses zu jedem beliebigen öffentlichen Exponenten berechnen.

Wollen wir also zeigen, wie man bei gegebenen (N, e, d) die beiden Primfaktoren (p, q) mit $p \cdot q = N$ berechnen kann: Dazu bedienen wir uns eines Las-Vegas-Algorithmus', den wir - praktisch betrachtet - einfach so oft anwenden, bis wir eine Faktorisierung von N erhalten. Dieser beruht auf folgender Beobachtung [Bon98, S. 3]:

Satz 3.20
Für zwei verschiedene Primzahlen $p \neq q$ und $N := p \cdot q$ hat die Kongruenz $x^2 \equiv 1 \bmod N$ wegen des chinesischen Restsatzes genau 4 Lösungen:
Die beiden trivialen Quadratwurzeln von 1 sind: $x_{1,2} \equiv \pm 1 \pmod{N}$ und die beiden nicht-trivialen Quadratwurzeln $x_{3,4}$ erhalten wir aus den jeweils eindeutigen Lösungen der beiden folgenden Kongruenzsysteme:

$$x_3 \equiv \begin{cases} 1 \bmod p \\ -1 \bmod q \end{cases} \qquad \text{und} \qquad x_4 \equiv \begin{cases} -1 \bmod p \\ 1 \bmod q \end{cases} .$$

Bemerkung 3.21
Analog dazu, wie für die beiden tivialen Lösungen gilt, dass die eine Lösung das additiv Inverse der anderen ist

$$x_1 \equiv -x_2 \bmod N ,$$

gilt auch für die beiden nicht-trivialen Lösungen die analoge Kongruenz

$$x_3 \equiv -x_4 \bmod N.$$

Wenn man also die Wurzel w aus 1 mod N zieht und man annimmt, dass das Ergebnis, für das $w^2 \equiv 1 \pmod{N}$ gilt, zufällig aus einer der 4 Möglichkeiten gewählt wird, so hat man eine 50%-ige Chance, dass man eine nicht-triviale Lösung erhält. Mit einer solchen nicht-trivialen Lösung w hat man in Folge aber die Möglichkeit N zu faktorisieren, indem man den $\mathrm{ggT}(w \pm 1, N)$ berechnet. Sollte die Wurzel $w = 1$ sein, so kann man wiederum davon die Wurzel ziehen und hoffen, dass das Ergebnis eine nicht-triviale Wurzel ist. Sollten wir bei diesem iterierten Wurzelziehen irgendwann einmal '-1' erhalten, so brechen wir das Verfahren ab und versuchen es nocheinmal.

Kommen wir nun zur Frage, wie man von 1 eine nicht-triviale Quadratwurzel mod N ziehen kann, ohne eine Faktorisierung von N ausnützen zu können:

[4] Sollte ein Mitarbeiter A seinen privaten Exponenten als öffentlichen Exponenten eines anderen Kollegen B entdecken, so könnte A seinen öffentlichen Schlüssel e_A zum *ent*schlüsseln der an B gerichteten Nachrichten verwenden - und umgekehrt.
Sollte es zwei öffentliche Exponenten e_A und e_B mit $e_A \mid e_B$ geben, dann könnte B den Wert $d_B \cdot \frac{e_B}{e_A}$ als geheimen Exponenten zum Entschlüsseln von Nachrichten an A verwenden.

An dieser Stelle möge sich der Leser noch einmal das Kapitel 3.4 ins Bewusstsein rufen. Einerseits gilt $ed \equiv 1 \bmod v$, wobei v wiederum als $v := kgV(p-1, q-1)$ definiert ist. Daraus folgt, es gibt ein $k \in \mathbb{Z}$, sodass sich $ed-1$ schreiben lässt als

$$ed - 1 = k \cdot v.$$

Andererseits gilt wegen des Satzes von Euler (Satz 3.5)

$$\forall a \in {\mathbb{Z}_N}^* : \quad \left\{ \begin{array}{rclcl} a^v & = & \left(a^{p-1}\right)^{\frac{v}{p-1}} & \equiv & 1^{\frac{v}{p-1}} = 1 \mod p \\ a^v & = & \left(a^{q-1}\right)^{\frac{v}{q-1}} & \equiv & 1^{\frac{v}{q-1}} = 1 \mod q \end{array} \right\} \quad \Rightarrow \quad a^v \equiv 1 \mod \underbrace{kgV(p,q)}_{=N}.$$

Diese beiden Tatsachen gemeinsam ergeben also, dass

$$\forall a \in {\mathbb{Z}_n}^* : \; a^{ed-1} = \left(a^v\right)^k \equiv 1^k = 1 \bmod N$$

gilt. Wegen $p \neq q \in \mathbb{P}$ ist $v = kgV(p-1, q-1)$ jedenfalls gerade und wir können $ed-1$ darstellen als

$$ed - 1 = 2^s r \qquad \text{wobei} \qquad s := \max_t \left\{ t \in \mathbb{N} \; : \; 2^t \text{ teilt } ed-1 \right\} \qquad \text{und} \qquad s \geq 1. \tag{3.1}$$

Sei nun $a \in {\mathbb{Z}_n}^*$ beliebig aber fest gewählt. Sollte $ggT(a, N) > 1$ sein, so haben wir damit einen Primfaktor von N gefunden, was unser Ziel war. Ansonsten gilt wie soeben gezeigt

$$a^{2^s r} \equiv 1 \bmod N.$$

Die Idee besteht nun darin, aus $a^{2^s r} \equiv 1 \pmod N$ die Quadratwurzel modulo N zu ziehen, indem wir - natürlich mit dem 'square & multiply'-Algorithmus aus Kap. 3.1.3 - den Wert $a^{2^{s-1} r}$ berechnen. Sollte die Wurzel eine Zahl $w \neq \pm 1$, also eine nicht-triviale Wurzelsein, so hatten wir mit diesem Wert für a Erfolg und wir können damit nun N faktorisieren indem wir $ggT(w \pm 1, N)$ bilden. *Beide* Vorzeichenmöglichkeiten liefern als Ergebnis einen Primfaktor von N.
Sollte als Wurzel $w = 1$ herauskommen und der Exponent $2^{s-1} r$ immer noch gerade sein, so können wir (durch halbieren des Exponenten) solange die Wurzel davon ziehen, bis entweder für die Wurzel ein Wert ungleich 1 herauskommt oder bis kein Halbieren des Exponenten mehr möglich ist, da der Exponent den Wert r erreicht hat und ungerade ist. Sollten wir im Verlauf dieser Werte auf den Wert $w = -1$ stoßen, so hatten wir mit dieser Wahl für a keinen Erfolg. Im Falle, dass wir den Exponenten nicht mehr halbieren können sollten, da sogar $a^r = 1$ gilt, führt dieses a ebenfalls zu keiner Faktorisierung. In beiden Fällen müssen wir dieses konkrete a verwerfen und einen neuen Wert für a suchen, und zwar solange, bis wir ein a finden, das zu einem $w \neq \pm 1$ führt.

Dh. was wir soeben gemacht haben, war, dass wir die Folge der Restklassen

$$\underbrace{a^{2^s r}}_{\equiv 1 \bmod N}, a^{2^{s-1} r}, a^{2^{s-2} r}, \; \ldots \; , a^{2^2 r}, a^{2r}, a^r$$

betrachtet und gehofft haben, dass das erste Auftreten eines Wertes ungleich 1 eine nicht-triviale Wurzel liefert. Da r in der Praxis jedoch sehr groß ist, müsste bei diesem Vorgehen (trotz des square & multiply-Algorithmus') *jedesmal* eine große Potenz gebildet werden.

Um Rechenzeit zu sparen setzt man diese Idee in der Praxis „von der anderen Seite“ aus angehend um und betrachtet die Restklassenfolge

$$a^r, \underbrace{a^{2r}}_{\equiv (a^r)^2}, \underbrace{a^{2^2 r}}_{\equiv (a^{2r})^2}, \; \ldots \; \underbrace{a^{2^{s-1} r}}_{\equiv \left(a^{2^{s-2} r}\right)^2}, \underbrace{a^{2^s r}}_{\equiv \left(a^{2^{s-1} r}\right)^2} \equiv 1 \pmod N.$$

input : Modul N, öffentlicher Exponent e, privater Exponent d
output : (nicht-triviale) Faktorisierung (p, q) von N

$FaktorGefunden := False$
while $FaktorGefunden = False$ **do** // probieren bis man Erfolg hat
  wähle $a \in \mathbb{Z}_n{}^*$ zufällig
  if $ggT(a, N) > 1$ **then**
    // nicht-trivialer Faktor von N gefunden
    $FaktorGefunden := True$
    return $\left(a, \frac{N}{a}\right)$
  end
  $s := \max\left\{t \in \mathbb{N} : 2^t \text{ teilt } ed-1\right\}$; // Vielfachheit von 2 in $ed-1$
  $r := \frac{ed-1}{2^s}$; // ungerader Anteil von $ed-1$
  // es gilt: $ed - 1 = 2^s r$
  $z_1 := a^r$; // mittels 'sq&mult'-Alg.
  if $z_1 \equiv 1 \ (mod\ N)$ **then**
    // Faktorisierung von N mit *diesem* a nicht möglich
    next ; // Sprung zum Beginn der while-Schleife
  end
  $z_2 := (z_1)^2 \ (mod\ N)$
  while $z_2 \not\equiv 1 \ (mod\ N)$ **do** // solange quadrieren bis $z_2 \equiv 1$ ist
    $z_1 := z_2$
    $z_2 := (z_1)^2 \ (mod\ N)$
  end
  // es gilt: $z_2 \equiv 1 \ (mod\ N)$
  if $z_1 \not\equiv -1 \ (mod\ N)$ **then** // Faktorisierung von N möglich
    $p := ggT(z_1 + 1, N)$
    $q := \frac{N}{p}$
    $FaktorGefunden := True$
    return (p, q)
  else
    // Faktorisierung von N mit *diesem* a nicht möglich
  end
end

Algorithmus 3.22: Faktorisieren von N mittels (e, d)

Hierbei braucht nur für die Berechnung des ersten Elements a^r eine große Potenz ausgewertet werden. Jedes weitere Element lässt sich ganz leicht durch Quadrieren (und Reduktion modulo N) seines Vorgängers berechnen. In dieser Folge suchen wir nach jener Restklasse w, deren Nachfolger die Restklasse 1 ist. Dh. für unser gesuchtes w gilt:

$$w \not\equiv 1 \mod N \qquad \text{aber} \qquad w^2 \equiv 1 \mod N$$

Von dieser Seite aus betrachtet unterscheiden wir folgende Fälle:
Ist $w \equiv -1 \pmod N$, dann war dieses konkrete a ein Misserfolg und wir wiederholen diesen Vorgang nochmals, bloß für ein anderes a. Ebenso, wenn schon $a^r = 1 \mod N$ gilt.
Und im Fall, dass $w \not\equiv \pm 1 \pmod N$, liefert uns sowohl $ggT(w + 1, N)$ als auch $ggT(w - 1, N)$ die beiden Primfaktoren von N.

Diese Idee ist im Algorithmus 3.22 zusammengefasst.

Nachdem wir nun einen Algorithmus zur Verfügung haben, der abhängig von einem zufällig gewählten a eine Faktorisierung von N ermöglicht, müssen wir uns noch überlegen, unter

welchen Umständen ein zufällig gewähltes a zu einem Erfolg, sprich zu einer Faktorisierung führt oder nicht [Buc10, S. 143f]:

Satz 3.23
Sei $a \in \mathbb{Z}_N{}^*$, dh. $ggT(a, N) = 1$ und seien e, d, s und r wie in (3.1) gewählt.
Wenn die Ordnung von a^r mod p ungleich der Ordnung von a^r mod q ist, dann gilt für ein passendes $t \in \{0, 1, \ldots, s-1\}$, dass $1 < ggT(a^{2^t r} \pm 1, N) < N$ ist. Somit erhalten wir einen nicht-trivialen Faktor von N.

Beweis
Wegen $(a^r)^{2^s} \equiv 1 \bmod N$ und aufgrund des Satzes von Lagrange (3.4) ist die Ordnung von a^r mod N ein Teiler von 2^s und liegt somit in der Menge

$$ord\big(a^r \bmod N\big) \in \left\{2^i \;\middle|\; 0 \le i \le s\right\}.$$

Da die Ordnungen von a^r modulo p und modulo q unterschiedlich sind, sei o.B.d.A. t so gewählt, dass

$$2^t = ord\big(a^r \bmod q\big) < ord\big(a^r \bmod p\big).$$

Trivialerweise gilt dann $a^{2^t r} \equiv 1 \bmod q$ aber $a^{2^t r} \not\equiv 1 \bmod p$. Oder anders gesagt: $a^{2^t r} - 1$ ist zwar ein Vielfaches von q aber nicht von p. Daraus folgt:

$$ggT\big(a^{2^t r} - 1, N\big) = q.$$

□

Jetzt wissen wir, welche Eigenschaft a haben muss, damit Algorithmus 3.22 eine Faktorisierung von N liefert. Dass diese Wahrscheinlichkeit größer als $\frac{1}{2}$ ist, sagt uns der folgende Satz:

Satz 3.24 (Faktorisierungswahrscheinlichkeit für ein a)
Die Anzahl jener $a \in \mathbb{Z}_n{}^*$ mit $ord(a^r \bmod p) \neq ord(a^r \bmod q)$ beträgt mindestens $\frac{(p-1)(q-1)}{2}$.
Da $|\mathbb{Z}_N{}^*| = (p-1)(q-1)$ ist, gilt für die Erfolgswahrscheinlichkeit $\mathbf{P}$

$$\mathbf{P} \ge \frac{1}{2}.$$

Beweis
Siehe [Buc10, S. 144, Theorem 9.3.8] oder [Sti06, S. 204ff].

□

Dh. für ein einziges a ist die Wahrscheinlichkeit, zu einer Faktorisierung von N zu gelangen, mindestens 50%. Wenn man Algorithmus 3.22 mit m verschiedenen Werten für a durchlaufen lässt, so sinkt die Wahrscheinlichkeit, doch kein passendes a gefunden zu haben, auf

$$P_{\text{Misserfolg}} \le \left(\frac{1}{2}\right)^m.$$

Für hinreichend viele Versuche lässt sich somit die Erfolgswahrscheinlichkeit beliebig groß machen, was den Algorithmus 3.22 für die Praxis tauglich macht.

Und welche Auswirkungen hat nun dieser Algorithmus, der aus (N, e, d) die beiden Faktoren (p, q) liefern kann?

- Zum einen kann dadurch jeder RSA-Teilnehmer alle verschlüsselten Nachrichten, die für einen anderen Empfänger mit demselbem Modul N bestimmt waren, auch mitlesen. Da ein ernsthafter Einsatz von RSA dies zu unterbinden weiß, ist dieses Ergebnis wohl eher theoretischer Natur.

- Für die Praxis viel wichtiger ist die Schlussfolgerung, dass, wenn der geheime Schlüssel von Alice einmal bekannt geworden ist, gleichsam die Faktorisierung von N bekannt geworden ist. Deshalb reicht es nicht, dass sie sich bloß ein neues Exponentenpaar $(e_{\text{neu}}, d_{\text{neu}})$ zulegt, sondern Alice muss sich auch einen neuen Modul, dh. zwei neue Primfaktoren (p, q) berechnen.

Weiters bedeutet dies, dass die beiden Probleme

1. Gegeben (N, e). Finde den privaten Exponenten d.
2. Gegeben N. Finde die beiden Faktoren p, q von N.

in gewisser Weise gleich schwierig sind.

Weitere Informationen zu diesem Angriff sind in [Buc10, Kap. 9.3.4, S. 142-145], [Sti06, Kap. 5.7.2, S. 202-206] und [KK10, Kap. 7.3.2, S. 122-125] zu finden.

3.3.5 gemeinsamer Modul II

Im vorigen Abschnitt haben wir gesehen, dass 2 Personen mit demselben Modul voneinander alle Geheimtexte entziffern können. Aber es gilt sogar noch mehr:
Wenn jemand eine Nachricht M verschlüsselt an zwei Empfänger mit demselben Modul verschickt, kann ein lauschender Angreifer den Klartext zumeist rekonstruieren:

Seien (N, e_1) und (N, e_2) die beiden öffentlichen Schlüssel mit $e_1 \neq e_2$. Weiters nehmen wir an, dass $ggT(e_1, e_2) = 1$ sei, was keine wesentliche Einschränkung ist [Pom03, S. 36]. Eve hört die beiden Chiffrate

$$C_1 \equiv M^{e_1} \bmod N \qquad \text{und} \qquad C_2 \equiv M^{e_2} \bmod N$$

ab. Weiters wendet sie den Erweiterten Euklidischen Algorithmus 3.8 auf e_1 und e_2 an. Weil $ggT(e_1, e_2) = 1$ ist, liefert der Alg. ihr zwei Zahlen X, Y mit

$$Xe_1 + Ye_2 = 1 \ .$$

Nun brauch Eve nur noch $C_1^X \cdot C_2^Y \bmod N$ zu bilden und kann sich dadurch

$$C_1^X \cdot C_2^Y \ \equiv \ (M^{e_1})^X \cdot (M^{e_2})^Y \ = \ M^{e_1 X + e_2 Y} \ = \ M^1 \ = \ M \qquad \bmod N$$

berechnen, was ihr Zugang zum Klartext verschafft.

Insgesamt wäre es also besser gewesen, dass die beiden RSA-Teilnehmer mit demselben Modul auch denselben Exponenten verwendet hätten [KL08, S. 361f]. Oder sie müssten den Modul N geheim halten, was dem Prinzip eines öffentlichen Schlüssels widerspricht. Daher darf, um die Sicherheit von RSA nicht zu gefährden, jeder Modul nur einer einzigen Person zugeteilt werden.

3.3.6 kleiner öffentlicher Exponent

Für manche Anwendungen kann es auf den ersten Blick sinnvoll erscheinen, einen kleinen öffentlichen Exponenten e zu wählen. Zum Beispiel hat eine Smart-Card nur begrenzte Rechenkapazität. Wenn man mit ihr eine Implementierung von RSA umsetzen möchte, könnte man mit dem Argument der Rechengeschwindigkeit einen kleinen Wert für e fordern, damit die Berechnung auf der Smart-Card nicht allzu lange dauert.

Ein anderer Grund könnte sein, dass ich das Verschlüsseln bzw. das Verifizieren meiner Signatur all jenen, die mir eine chiffrierte Nachricht zukommen lassen möchten, schnell bzw. mit wenig Rechenleistung ermöglichen möchte.

Wie die folgenden Abschnitte aber zeigen werden, kann eine kleine Wahl für e ein Sicherheitsrisiko darstellen:

3.3.6.1 Håstad's Broadcast Attacke - Einführung

Zum besseren Verständnis betrachten wir den nun folgenden Angriff vorerst in seiner aller einfachsten Möglichkeit ([Pom03, Kapitel 2.8, S. 37] bzw. [KK10, Kapitel 7.4.2, S. 126-128]). Im nächsten Abschnitt 3.3.6.2 werden wir ihn dann in seiner allgemeinsten Form kennen lernen.

Nehmen wir an, Bob sendet dieselbe Botschaft an *drei* verschiedene RSA-Teilnehmer mit den öffentlichen Schlüsseln $(N_1, e), (N_2, e)$ und (N_3, e) und es gelte $e = 3$. Dh. die öffentlichen Exponenten der drei Empfänger sind alle *gleich*. Wir können davon ausgehen, dass die drei Moduli N_1, N_2 und N_3 teilerfremd sind, denn andernfalls könnte ein Angreifer $ggT(N_i, N_j)\ \forall i \neq j$ bilden und erhielte dadurch einen nicht-trivialen Faktor von zumindest zwei der Moduli und könnte die Botschaft erst recht lesen. Weiters nehmen wir an, dass der zu verschlüsselnde Klartext $M < N_i\ \ \forall i = 1, 2, 3$ ist.
Bob berechnet für die Verschlüsselung

$$C_1 \equiv M^3 \bmod N_1\ , \qquad C_2 \equiv M^3 \bmod N_2 \qquad \text{und} \qquad C_3 \equiv M^3 \bmod N_3 \tag{3.2}$$

und sendet die Chiffrate über seine Leitung hinaus. Wenn Eve Bobs Leitung belauscht, kann sie wie folgt den Klartext M berechnen:
Eve kennt vorerst nur C_1, C_2 und C_3 und berechnet sich das Produkt der Moduli $N := N_1 N_2 N_3$. Da die N_i teilerfremd sind, kann Eve den chinesischen Restsatz (Alg. 3.10) anwenden um sich das eindeutige $C < N$ zu konstruieren, das kongruent $C_i \bmod N_i$ ist. Dh. für $C < N$ gilt

$$C \equiv C_1 \bmod N_1\ , \qquad C \equiv C_2 \bmod N_2 \qquad \text{und} \qquad C \equiv C_3 \bmod N_3\ .$$

Eve kennt also den Wert $C < N$, für den gilt:

$$C \equiv C_i \bmod N_i \quad \forall i = 1, 2, 3\ .$$

M^3 hat wegen (3.2) ebenfalls die Eigenschaft

$$M^3 \equiv C_i \bmod N_i \quad \forall i = 1, 2, 3\ .$$

Aber wegen $M < N_i\ \forall i = 1, 2, 3$ folgt, dass auch $M^3 < N_1 N_2 N_3 = N$ ist. Dh. wir können zusammenfassen:

$$\left.\begin{array}{rcl} C & \equiv & M^3 \bmod N \\ C & < & N \\ M^3 & < & N \end{array}\right\} \quad \Rightarrow \quad C = M^3\ . \qquad \text{(man beachte: aus '}\equiv\text{' wurde '}=\text{' !)}$$

Somit gilt: $M = \sqrt[3]{C}$ und Eve kann den Klartext M durch numerisches Ziehen der 3. Wurzel in $\mathbb{R}$ (!) berechnen, was in polynomieller Laufzeit möglich ist.

Diesen Angriff kann man natürlich leicht auch auf ein $e > 3$ erweitern. Wenn e der gemeinsame öffentliche Exponent von k Empfängern ist, so muss nur $k \geq e$ sein um mit dieser Attacke Erfolg zu haben. Im Falle $k \ngeq e$, wo echt mehr Chiffrate als notwendig vorliegen, reicht es, sich e Chiffrate davon auszusuchen, was auch Rechenersparnis zur Folge hat.

Die allgemeine Form dieses Angriffs ist in Algorithmus 3.25 zusammengefasst.

input : öffentliche Schlüssel $(N_1, e), (N_2, e), \ldots, (N_k, e)$
Chiffrate $C_1, C_2, \ldots, C_k$
mit $k \geq e$ und $M \leq N_i \;\; \forall i = 1, \ldots, k$
output : Klartext M
(oder eine Faktorisierung (N_i, p_i, q_i) eines Moduls $N_i = p_i\, q_i$)

$N := N_1 \cdot N_2 \cdot \ \ldots \ \cdot N_k$
for *i = 1 , . . . , k* **do**
  $\widetilde{N}_i := \frac{N}{N_i}$
  if $\widetilde{N}_i > N_i$ **then**
    $(t, x, y) := E_{rweiterter} E_{uklidischer} A_{lgorithmus}\left(\widetilde{N}_i, N_i\right)$; // $\widetilde{N}_i\, x + N_i\, y = t$
    $\mathbb{N}_i^* := x$
  else
    $(t, x, y) := E_{rweiterter} E_{uklidischer} A_{lgorithmus}\left(N_i, \widetilde{N}_i\right)$; // $N_i\, x + \widetilde{N}_i\, y = t$
    $\mathbb{N}_i^* := y$
  end
  // *EEA* `siehe Kap. 3.1.7`
  if $t > 1$ **then**
    `// nicht-trivialen Faktor von` N_i `gefunden`
    return $(N_i, t, \frac{N_i}{t})$
  end
end
$C := \sum_{i=1}^{k} \left(C_i \cdot N_i^* \, \widetilde{N}_i\right) \mod N$
$M := \sqrt[e]{C}$
return M

Algorithmus 3.25: Håstad's Broadcast Attacke

Wenn man die konkrete Wahl $e = 3$ vermeiden aber trotzdem den Zeit- bzw. Rechenaufwand fürs Verschlüsseln möglichst gering halten möchte, dann ist ein oft gewählter Wert $e = 2^{16} + 1 = 65537$. Bei dieser Wahl für e kommen beim Algorithmus für's schnelle Potenzieren nämlich nur 2 der optionalen Multiplikationen vor (entsprechend den beiden Einsern in der Binärdarstellung von $2^{16} + 1 = (10000000000000001)_2$.

Für diesen Angriff hatten wir ein paar einschränkende Annahmen getroffen, die nur dem leichteren Verständnis der Grundidee dienten. Aus Sicht der Praxis können wir uns zurecht zwei Fragen stellen:

1. Wie wirkt es sich aus, wenn jemand, um der soeben beschriebenen Attacke zu entgehen, nicht allen dieselbe Nachricht M schickt, sondern eine leicht abgeänderte Version von M?
   Angenommen „vorne" wird zum gleich bleibenden Teil M der Nachricht zB. noch eine Zahl hinzufügt: der Reihe nach bekommt der i-te Empfänger den Klartext

   $$M_i = [\text{ Binärdarstellung von } i \parallel \text{Binärdarstellung von } M\ ]$$

   verschlüsselt übermittelt. Dh. wenn m die Binärstelligkeit von M ist, dann gilt $M_i := i2^m + M$ und die soeben beschriebene Methode kann offensichtlich nicht mehr angewandt werden.

2. Was passiert, wenn die öffentlichen Exponenten nicht alle gleich (demselben e) sondern verschieden sind? Dh. was ist, wenn für die öffentlichen Schlüssel $(N_1, e_1), (N_2, e_2), \ldots, (N_k, e_k)$ gilt, dass auch $e_i \neq e_j$ für $i \neq j$ erlaubt ist?

Dies sind zwei realistische Einwände um dem soeben geführten Angriff aber nur *scheinbar* zu entkommen. Wir werden im nächsten Abschnitt sehen, warum selbst eine Kombination dieser beiden Abwehrversuche durch eine allgemeinere Variante dieser Attacke angreifbar bleibt.

3.3.6.2 Håstad's Broadcast Attacke - allgemein

Der Angriff aus dem letzten Kapitel 3.3.6.1 ist ein Spezialfall von Håstads Broadcast Attacke [Hås88]. Wollen wir deshalb nochmals kurz die Situation von vorhin zusammenfassen, um sie dann in weiterer Folge zu verallgemeinern:
Beim Angriff im letzten Abschnitt war wesentlich, dass die öffentlichen Exponenten alle gleich waren und dass alle k Empfänger denselben Klartext M übermittelt bekamen. Dh. es galt: $C_i = M^e \bmod N_i \quad \forall i = 1, ..., k$.

Die erste Idee zur Vermeidung des Angriffs war, lineares Padding einzuführen. Dh. alle Empfänger bekommen einen leicht unterschiedlichen Klartext M_i übermittelt: $M_i := i \cdot 2^m + M$ - vereinfacht gesprochen wird „vorne einfach durchnummeriert". Die Abbildung $f_i : M \mapsto M_i := M + i \cdot 2^m$ ist somit linear (in M).
Die zweite Idee war, unterschiedliche öffentliche Exponenten e_i der Empfänger zuzulassen.

Insgesamt also werden im allgemeinen Fall die Chiffrate $C_i := \big(f_i(M)\big)^{e_i} \bmod N_i$ übermittelt. Håstad hat gezeigt: Wenn genügend viele Chiffrate C_i samt den zugehörigen Polynomen f_i bekannt sind, kann ein lauschender Angreifer den eigentlichen Klartext M errechnen. Dabei braucht f_i nicht linear zu sein, sondern kann ein beliebiges, bekanntes Polynom sein.

Um Håstads Ergebnis zu zeigen, benötigen wir ein Ergebnis von Coppersmith [Cop97], das wir hier zitieren wollen:

Satz 3.26 (Coppersmith)
Sei $N \in \mathbb{Z}$ und $f \in \mathbb{Z}_N[x]$ ein normiertes Polynom vom Grad d. Weiters sei $X := N^{\frac{1}{d}-\varepsilon}$ für ein

$\varepsilon \geq 0$. Dann kann man aus N und f effizient alle Nullstellen x_j berechnen mit

$$f(x_j) \equiv 0 \mod N \quad \text{und} \quad |x_j| < X \ .$$

Die Laufzeit wird dabei von der Laufzeit des LLL-Algorithmus dominiert, der auf ein Gitter der Dimension $\mathcal{O}(w)$ mit $w := \min\left(\frac{1}{\varepsilon}, \log_2 N\right)$ angewandt wird.

Beweis
siehe zB: [Bon98, Kap. 4.1, S. 6-8]

□

Je größer ε bzw. je kleiner somit X, jene Schranke, bis zu der die Nullstellen gefunden werden sollen, ist, umso schneller ist der Algorithmus. Vor allem (betragsmäßig) ganz kleine Nullstellen von f mod N können damit schnell gefunden werden.
Wir werden auf diesen Satz noch zurückgreifen. Kommen wir nun also zur Kernaussage dieses Abschnittes [Bon98, Th. 6, S. 9]:

Satz 3.27 (Håstad)
Seien $N_1, ..., N_k \in \mathbb{Z}$ paarweise teilerfremd, sei $N_{min} := \min_{i=1,...,k}(N_i)$ und seien weiters die k Polynome $g_i \in \mathbb{Z}_{N_i}[x] \ \forall\, i = 1, \ldots, k$ alle vom Grad höchsten d und sei $k \geq d$.
Wenn es nun ein $M < N_{min}$ gibt, das *simultan* Nullstelle für alle g_i mod N_i ist, dh. wenn gilt

$$g_i(M) \equiv 0 \mod N_i \qquad \forall i = 1, \ldots, k \ ,$$

dann kann man M effizient berechnen.

Bemerkung 3.28
In der praktischen Anwendung dieses Satzes zum Brechen von RSA werden wir für die im Satz 3.27 angeführten Polynomen g_i die Polynome $g_i = f_i^{e_i} - C_i$ verwenden. Des weiteren muss man nicht nur zwischen den Polynomen selbst sondern auch zwischen den Polynomgraden von f_i und jenen von g_i gut unterscheiden:

$$\text{grad}(g_i) = e_i \cdot grad(f_i) \qquad \forall i = 1, \ldots, k \ .$$

Beweis (von Satz 3.27 - Håstad)
Sei $N := N_1 \cdot N_2 \cdot \ldots \cdot N_k$. O.B.d.A. können wir die Polynome g_i als normiert annehmen. (Sollten die Polynome nicht normiert sein, so können wir sie normieren, in dem wir jeweils mit dem modularen Inversen des entsprechenden führenden Koeffizienten von g_i multiplizieren. Das Normieren verändert ja die Nullstellen nicht. Sollte bei der Inversenberechnung modulo N_i ein Fehler auftreten, dh. sollte das Inverse nicht existieren, so kann dies nur deshalb passieren, weil der größte gemeinsame Teiler von N_i und dem führenden Koeffizienten von $g_i(x)$ ein nicht-trivialer Teiler von N_i ist.)
Weiters können wir durch Multiplikation mit einer entsprechenden Potenz von x erreichen, dass alle Polynome g_i denselben Grad $d = \max_{i=1,...,k}\left\{grad(g_i)\right\}$ haben.
Um den Satz von Coppersmith (Satz 3.26) anwenden zu können, streben wir ein einziges, normiertes Polynom in $\mathbb{Z}_N{}^*[x]$ an, welches wir uns aus unseren gegebenen $g_i(x)$ als eine passende Linearkombination errechnen werden: Dazu berechnen[5] wir uns die modularen Inversen N_i^* von $\widetilde{N}_i := \frac{N}{N_i}$ modulo N_i mit

$$N_i^* \widetilde{N}_i \equiv 1 \mod N_i \qquad \forall i = 1, \ldots, k \ .$$

Somit gilt für die Koeffizienten $\lambda_i := N_i^* \widetilde{N}_i \quad \forall i = 1, \ldots, k$

$$\lambda_i \equiv \begin{cases} 1 \mod N_j, & j = i \\ 0 \mod N_j, & j \neq i \end{cases} \qquad \text{oder kurz:} \qquad \lambda_i \equiv \delta_{i,j} \mod N_j \ .$$

[5] ... ganz genauso wie beim Beweis des Chinsischen Restsatzes

Damit stellen wir das Polynom

$$g(x) := \sum_{i=1}^{k} \lambda_i \, g_i(x)$$

auf, welches wir in $\mathbb{Z}_N[x]$ betrachten und für welches gilt:

1. Die Polynome $g_i(x)$ sind alle normiert und vom selben Grad d. Deshalb ist der führende Koeffizient von $g(x)$ gleich $\sum_{i=1}^{k} \lambda_i \equiv 1 \bmod N$.

2. Es gilt:

$$\begin{aligned} & g(x) \equiv g_i(x) && \bmod N_i && \forall i = 1, \ldots, k && \Rightarrow \\ \Rightarrow \quad & g(M) \equiv g_i(M) \equiv 0 && \bmod N_i && \forall i = 1, \ldots, k && \Rightarrow \\ \Rightarrow \quad & g(M) \equiv 0 && \bmod N\,. \end{aligned}$$

Um den Satz von Coppersmith (Satz 3.26) anwenden zu können, bleibt nur noch zu zeigen, dass $M < N^{\frac{1}{d}}$ ist, was wegen

$$\left.\begin{array}{rcl} d \le k & \overset{M \ge 1}{\Longrightarrow} & M^d \le M^k \\ M < N_i \quad \forall i = 1, \ldots k & \Rightarrow & M^k < N_1 \cdot N_2 \cdot \ldots \cdot N_k = N \end{array}\right\} \Rightarrow \; M^d < N \; \Rightarrow \; M < N^{\frac{1}{d}}$$

auch erfüllt ist. Wir haben also gezeigt, dass wir Satz 3.26 anwenden und somit die Nullstelle M von $g(x)$ mod N effizient berechnen können.

□

Wie wird Satz 3.27 (Satz von Håstad) nun aber in der Praxis angewandt?
Angenommen Bob möchte eine Nachricht M an k RSA-Empfänger versenden. Um dem Angriff wie im Kapitel 3.3.6.1 beschrieben zu entgehen, wendet er auf den Klartext M für jeden Empfänger extra eine Polynomfunktion f_i an, um damit den Klartext zu „padden“ und verschlüsselt das so erhaltene Zwischenergebnis $f_i(M)$ mit dem öffentlichen Schlüssel (N_i, e_i), wobei die öffentlichen Exponenten e_i in der allgemeinen Form des Angriffs auch unterschiedlich sein dürfen.
Håstads Satz besagt nun, dass ein Angreifer bei Kenntnis von genügend vielen Chiffraten C_i samt zugehörigen N_i, e_i und f_i die Nachricht M trotzdem leicht berechnen kann. Als Polynome $g_i(x)$ verwendet man dabei (wie auch schon in Bemerkung 3.28 angekündigt)

$$g_i(x) := \big(f_i(x)\big)^{e_i} - C_i \qquad \forall i = 1, \ldots, k$$

mit dem Grad

$$d_i := \operatorname{grad}\big(g_i(x)\big) = e_i \cdot \operatorname{grad}\big(f_i(x)\big)\,.$$

Dh. als Polynomgrad d im Satz 3.27 verwenden wir $d := \max\limits_{i=1,\ldots,k}\big(d_i\big)$. Somit ist dieser Angriff im Falle von $k > d = \max\limits_{i}\big\{e_i \cdot grad(f_i)\big\}$ erfolgreich. Insbesondere im Fall lauter gleicher öffentlicher Exponenten $e_i := e$ und *linearem* Padden folgt - in Übereinstimmung mit dem Ergebnis aus Kap. 3.3.6.1 - dass der Angriff erfolgreich ist, wenn $k \ge e$ ist.

Für weitere Details zu Håstad's Broadcast Attacke sei auf[Bon98, Kap. 4.1 u. 4.2, S. 6-9] und [MvOV97, Kap. 8.8, §8.2, S. 313f] verwiesen.

3.3.6.3 Abhängige Klartexte Attacke - Franklin/Reiter

Franklin und Reiter [CFPR96] haben einen Angriff entdeckt, der dann zum Tragen kommen kann, wenn Bob verschiedene Nachrichten, die miteinander in *bekannter* Beziehung zueinander

stehen, mit Alices öffentlichen Schlüssel (N, e) verschlüsselt, und an sie sendet:
Nehmen wir an, Bob möchte die beiden verschiedenen Botschaften $M_1 \neq M_2$ verschlüsselt an Alice senden und M_1 und M_2 stehen miteinander in einer affinen Beziehung[6]

$$M_1 \equiv f(M_2) \mod N$$

für ein bekanntes, lineares Polynom $f(x) := ax + b \in \mathbb{Z}_N[x]$ mit $b \not\equiv 0$. Wegen

$$C_1 \equiv \big(f(M_2)\big)^e \mod N$$

ist M_2 Nullstelle des Polynoms

$$g_1(x) := \big(f(x)\big)^e - C_1 \in \mathbb{Z}_N[x].$$

Klarerweise ist M_2 aber wegen $C_2 \equiv (M_2)^e \bmod N$ auch Nullstelle des Polynoms

$$g_2(x) := x^e - C_2 \mod N \in \mathbb{Z}_N[x].$$

Somit kommt der Term $x - M_2$ als Faktor in *beiden* Polynomen $g_1(x)$ und $g_2(x)$ vor. Dh. ein Angreifer kann bei Kenntnis von $\big(N, e, f(x), C_1, C_2\big)$ die beiden Polynome $g_1(x)$ und $g_2(x)$ aufstellen und darauf den Euklidischen Algorithmus anwenden[7], um den $\mathrm{ggT}\big(g_1(x), g_2(x)\big)$ zu berechnen. Wenn das Ergebnis linear ist, so erhalten wir als Ergebnis $ggT\big(g_1, g_2\big) = x - M_2$ und haben somit M_2 berechnen können.

Im Fall $e = 3$ kann man zeigen, dass der ggT linear sein *muss* [Bon98, Kap. 4.3, S. 9f].

Für $e > 3$ ist der ggT in den meisten Fällen ebenfalls linear. Nur bei ganz seltenen Konstellationen für f, M_1 und M_2 führt dieser Angriff nicht zum gewünschten Ziel, da der ggT dann nicht mehr linear ist.

3.3.6.4 partial key exposure Attacke - Boneh, Durfee u. Frankel

Dieser Attacke von Boneh, Durfee und Frankel [BDF98] liegt folgender Satz von Coppersmith zu Grunde [Bon98, Th. 10, S. 11]:

Satz 3.29 (Coppersmith)
Wenn man von einem der beiden Primfaktoren p oder q eines RSA-Moduls $N = pq$ mit n Binärstellen die vordersten $\lceil \frac{n}{4} \rceil$ oder die letzten $\lceil \frac{n}{4} \rceil$ Binärstellen kennt, kann man N effizient faktorisieren.

Die Grundlage dieses Angriffs liefert der folgende Satz [Bon98, Th. 9, S. 11]:

Satz 3.30 (partial key exposure)
Sei N ein RSA-Modul mit n Binärstellen und sei (e, d) ein dazugehöriges Schlüsselpaar. Wenn man von d die $\lceil \frac{n}{4} \rceil$ letzten Binärstellen kennt, kann man N mit einer Laufzeit von $\mathcal{O}(e \log_2 e)$ faktorisieren.

Beweis
Der Beweis [Bon98, S. 11] verwendet, dass d modulo $\varphi(N) = (p-1)(q-1)$ berechnet wurde. Sei $k \in \mathbb{N}$ jene natürliche Zahl mit

$$1 = e\,d - k\,\varphi(N)\,.$$

[6] Wie Coppersmith gezeicht hat, kann in der Praxis solch ein affiner Zusammenhang von M_1 und M_2 durch unterschiedliches Padden derselben Nachricht auftreten [Pom03, S. 40].

[7] Eigentlich funktioniert der Euklidische Algorithmus nur in Euklidischen Ringen. Für unsere Absichten ist aber der Euklidische Algorithmus auch auf Polynome aus $\mathbb{Z}_N[x]$ anwendbar: Denn sollte bei der Berechnung des größten gemeinsamen Teilers ein Fehler auftreten, so würden wir dadurch eine Faktorisierung von N erhalten.

Klarerweise muss dann wegen $d \leq \varphi(N)$ gelten, dass $k \leq e$ ist. Dann gilt:

$$\begin{aligned} 1 &= e\,d - k\,(N-p-q+1) && \Big| \cdot p \\ \Rightarrow \quad p &= (ed)\,p - k\,p\,(N-p-q+1) \quad = \\ &= (ed)\,p - k\,p\,(N-p+1) + k\,N \quad = \\ &= k\,p^2 + \Big((ed) - k(N+1)\Big)\,p + k\,N \end{aligned}$$

Somit gilt diese Gleichung auch modulo jeder beliebigen Zahl, also insbesondere gilt

$$p \;\equiv\; k \cdot p^2 + \Big((ed) - k(N+1)\Big) \cdot p + k\,N \qquad \mod 2^{\frac{n}{4}}\,. \tag{3.3}$$

Da die letzten $\left\lceil \frac{n}{4} \right\rceil$ Bits von d bekannt sind, ist auch der Wert von $ed \mod 2^{\frac{n}{4}}$ bekannt. Für jedes $\widetilde{k} \in \mathbb{N}$ mit $0 < \widetilde{k} \leq e$ ist somit die rechte Seite der Kongruenz (3.3) ein quadratisches Polynom in p, dessen Nullstellen mod $2^{\frac{n}{4}}$ wir mit $R_{\widetilde{k}}$ bezeichnen wollen. Man kann zeigen, dass die Anzahl der möglichen Nullstellen modulo $2^{\frac{n}{4}}$ durch

$$\left| \bigcup_{\widetilde{k}=1}^{e} R_{\widetilde{k}} \right| \;\leq\; e \log_2 e$$

beschränkt ist. Für jede der in Frage kommenden Nullstellen können wir den Satz 3.29 anwenden und da k durch e beschränkt ist, müssen wir bei dieser Vorgehensweise auf das „richtige“ $\widetilde{k}$ mit $\widetilde{k} = k$ stoßen, womit wir eine Faktorisierung von N erhalten.

□

Bemerkung 3.31
Für den Fall der Berechnung von d als Inverses zu e modulo $v = kgV(p-1, q-1)$ (und nicht modulo $\varphi(N) = (p-1)(q-1)$) lässt sich der erste Teil des Beweises meiner Meinung nach mit einer zusätzlichen Voraussetzung wie folgt adaptieren:

Beweis
Sei $k \in \mathbb{N}$ jene natürliche Zahl mit

$$1 \;=\; e\,d - k\,v$$

und sei weiters $\Delta := ggT(p-1, q-1)$. Dann gilt:

$$\begin{aligned} 1 &= e\,d - k\,v && \Big| \cdot \Delta \\ \Rightarrow \quad \Delta &= e\,d\,\Delta - k\,v\,\Delta \;=\; e\,d\,\Delta - k\,(p-1)(q-1) \;= \\ &= e\,d\,\Delta - k\,(N-p-q+1) && \Big| \cdot p \\ \Delta\,p &= (e\,d\,\Delta)\,p - k\,p\,(N-p+1) + k\,N \;= \\ &= k \cdot p^2 + \Big(e\,d\,\Delta - k(N+1)\Big) \cdot p + k\,N \end{aligned}$$

Weiters gelte die zusätzliche Voraussetzung $\Delta = ggT(p-1, q-1) = 2$. Dies ist zum Beispiel der Fall, wenn nicht nur $p \neq q \in \mathbb{P}$ sondern auch $\frac{p-1}{2}, \frac{q-1}{2}$ prim, also Germain-Primzahlen sind, was durchaus realistisch ist (s. Kap. 3.2.2). Dann gilt

$$\begin{aligned} 2\,p &= k \cdot p^2 + \Big(2\,e\,d - k(N+1)\Big) \cdot p + k\,N \\ \Rightarrow \quad p &= \frac{k}{2} \cdot p^2 + \left(e\,d - k\,\frac{N+1}{2}\right) \cdot p + \frac{k}{2}\,N \quad \in \mathbb{Z} \end{aligned}$$

□

Ein expliziter Algorithmus für den soeben beschriebenen Angriff ist in [SDO06, S.668] beschrieben.

3.3.6.5 Coppersmiths Short Pad Attacke

Coppersmith hat gezeigt, dass es einen Angriff auf RSA gibt, wenn eine Nachricht am Ende immer mit derselben (hinreichend geringen) Anzahl an zufälligen Bits aufgefüllt wird.

Sei n die Binärstelligkeit von N und sei $m := \left\lfloor \frac{n}{e^2} \right\rfloor$ als die Anzahl der hinzugefügten, zufälligen Bits definiert. Eine zu verschlüsselnde Nachricht kann somit höchstens $n - m$ Binärstellen haben.
Angenommen Bob möchte Alice die Nachricht $M_1 := 2^m M + r_1$ mit der zufälligen Zeichenkette $0 < r_1 < 2^m$ schicken und ein aktiver Angreifer Mallory fängt das Chiffrat $C_1 := {M_1}^e \mod N$ ab und verhindert seine Weiterleitung an Alice. Irgendwann wird Bob versuchen die Nachricht nocheinmal zu senden. Dazu verschlüsselt er diesmal den Klartext $M_2 := 2^m M + r_2$ und sendet $C_2 = {M_2}^e \mod N$ an Alice. Realistischerweise können wir davon ausgehen, dass das zweite Padden ein anderes Ergebnis liefert als das erste, weshalb wir $r_1 \neq r_2$ annehmen können. Nachdem Mallory nun auch C_2 abgehört hat, kann er sich trotz der Unkenntnis von r_1 und r_2 den Klartext M rekonstruieren [Bon98, Kap. 4.4, S. 10f].

Für $e = 3$ dürfte somit der zufällige Teil nur $\frac{1}{9}$ des Klartextes ausmachen. Um bei der beliebten Verwendung von $e = 65537$ auch nur ein einziges hinzugefügtes Bit zu knacken, würde dieser Angriff somit erst bei einem Modul N mit über 4 Milliarden Binärstellen relevant werden.

3.3.7 kleiner privater Exponent

Motivation für einen kleinen privaten Exponenten d könnte - ganz analog zum vorigen Abschnitt - sein, dass man den Rechenaufwand fürs Entschlüsseln bzw. Signieren gering halten möchte. Dabei würde man bei der Schlüsselerzeugung - abweichend von der in Abschnitt 3.2.2 beschriebenen Vorgehensweise - zuerst d wählen und danach erst ein passendes e dazu berechnen [MvOV97, S. 288].

3.3.7.1 Iterationsangriffe - cycling attacks

Der (einfache) *Iterationsangriff* - auch *cycling attack* genannt - verwendet die Tatsache, dass das Verschlüsseln mit RSA eine Permutation der Menge $\mathbb{Z}_N$ darstellt und dass jede Permutation, oft genug angewendet, wieder die Identität ergibt. Dh. wir wenden auf das abgehörte Chiffrat C nochmals den öffentlichen Schlüssel an, und auf dessen Ergebnis wiederum, usw ... [MvOV97, S. 289f]. Dh. wir betrachten die Folge der Restklassen

$$C^e \bmod N, \quad (C^e)^e = C^{e^2} \bmod N, \quad \left(C^{e^2}\right)^e = C^{e^3} \bmod N, \quad \left(C^{e^3}\right)^e = C^{e^4} \bmod N, \quad \ldots$$

und für irgendein $k \in \mathbb{N}$ muss dann gelten

$$C^{e^k} \equiv C \mod N .$$

Folglich muss dann

$$C^{e^{k-1}} \equiv M \mod N$$

sein, womit wir bei Kenntnis von k den Klartext M effizient berechnen können.

Als *allgemeinen Iterationsangriff (generalized cycling attack)* wollen wir jene Erweiterung dieses Angriffs bezeichnen, welche nach dem kleinsten $u \in \mathbb{N}$ mit

$$ggT\left(C^{e^u} - C, N\right) > 1$$

sucht. O.B.d.A. sei dieses k minimal. Klarerweise gilt dann $u \leq k$.

1. Sollte tatsächlich $u = k$ sein, so entspricht dies genau dem Fall des soeben beschriebenen (einfachen) Iterationsangriffs und wir können (nur dieses eine) M berechnen.

2. Für den Fall $u < k$ wissen wir, dass dann entweder

$$C^{e^u} \equiv C \mod p \qquad \text{und} \qquad C^{e^u} \not\equiv C \mod q$$

oder

$$C^{e^u} \not\equiv C \mod p \qquad \text{und} \qquad C^{e^u} \equiv C \mod q$$

gelten muss, da k minimal gewählt war. Jedenfalls gilt dann

$$1 \;<\; ggT\left(C^{e^u} - C, N\right) \;<\; N\,,$$

womit wir N faktorisiert haben und nicht nur den Klartext M berechnen sondern auch alle anderen bisher und künftig abgehörten Chiffrate entziffern können.

[Pom03, Kap. 2.5, S. 33f] betrachtet diesen Angriff aus dem Blickwinkel der Gruppentheorie in der Gruppe aller Permutationen einer N-elementigen Menge und in [Bau97, Kap. 10.4.2, S. 189-191] finden sich ausführliche Beispiele wieder. [Pom03, Kap. 2.6, S. 35] betrachtet dieselbe Idee aus der Sicht von Bahnen, Stabilisatoren und Bahnlängen.

3.3.7.2 Angriff von M. Wiener - $d < N^{0,25}$

1990 hat Michael J. Wiener in [Wie90] eine Möglichkeit vorgestellt, wie man im Falle eines kleinen privaten Exponenten d mithilfe von Kettenbrüchen den geheimen Schlüssel effizient berechnen kann: Wenn d klein genug ist, kann man zeigen, dass der Bruch $\frac{k}{d}$ in der Kettenbruchentwicklung von $\frac{e}{N}$ vorkommen muss.
Für die Theorie der Kettenbrüche sei verwiesen auf zB. [KK10, Kap. 7.5, S. 128-136]. Für unsere Zwecke reicht es aus, sich auf folgende Inhalte zu beschränken:

Definition 3.32 (Kettenbruch)
Sei $q_0 \in \mathbb{Z}$ und seien $q_1, q_2, \ldots, q_m \in \mathbb{N}$. Dann heißt

$$[q_0, q_1, \ldots, q_m] \;:=\; q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots_{q_{m-1} + \frac{1}{q_m}}}}} \in \mathbb{Q}$$

ein (endlicher) *Kettenbruch* mit Wert $x := [q_0, q_1, \ldots, q_m] \in \mathbb{Q}$.

Satz 3.33 (Eindeutigkeit der Kettenbruchentwicklung)
Für jede Zahl $x \in \mathbb{R}$ existiert im Wesentlichen[8] genau eine Darstellung als Kettenbruch.

Für $x := \frac{a}{b} \in \mathbb{Q}$ sind die q_i der Kettenbruchentwicklung von x genau die Quotienten, die bei der Berechnung von $ggT(a, b)$ mittels des Euklidischen Algorithmus' auftreten.

[8] Für $x \in \mathbb{R}\backslash\mathbb{Q}$ ist die Kettenbruchentwicklung eindeutig (und unendlich).
Für $x = \frac{a}{b} \in \mathbb{Q}$ terminiert der Euklidische Algorithmus immer, dh. bei der Berechnung des $ggT(a, b)$ treten nur endlich viele Quotienten auf und somit ist auch die Kettenbruchentwicklung von x endlich. Wegen

$$[q_0, q_1, \ldots, q_m, 1] = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots_{q_m + \frac{1}{1}}}}} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ddots_{q_m + 1}}}} = [q_0, q_1, \ldots, q_m + 1]$$

kann man mit der Bedingung $q_m \geq 2$ Eindeutigkeit erzwingen.
Für $x \in \mathbb{Z}$ lautet die Kettenbruchentwicklung $[x]$ und ist eindeutig.

Definition 3.34 (Näherungsbruch)
Sei $[q_0, q_1, \ldots, q_m]$ ein Kettenbruch. Für $0 \leq k \leq m$ heißt dann

$$[q_0, q_1, \ldots, q_k] \in \mathbb{Q}$$

der k-te Näherungsbruch von $[q_0, q_1, \ldots, q_m]$.

Bemerkung 3.35 (Berechnung der Näherungsbrüche)
Sei $[q_0, q_1, \ldots, q_m]$ ein Kettenbruch. Für $i = 0, \ldots, m$ lassen sich dann der Zähler r_i und der Nenner s_i des i-ten Näherungsbruches $\frac{r_i}{s_i}$ des Kettenbruches $[q_0, q_1, \ldots, q_m]$ durch folgende Rekursionen berechnen:

$$\begin{array}{l|l|ll} r_0 := q_0 & r_1 := q_1 q_0 + 1 & r_k := q_k r_{k-1} + r_{k-2} & \text{für } k \geq 2 \\ s_0 := 1 & s_1 := q_1 & s_k := q_k s_{k-1} + s_{k-2} & \text{für } k \geq 2 \end{array}$$

Satz 3.36 (Näherungsbruch)
Sei $x \in \mathbb{R}$ und sei $\frac{r}{s} \in \mathbb{Q}$ ein Bruch in gekürzter Form, sodass

$$\left| x - \frac{r}{s} \right| < \frac{1}{2s^2}$$

gilt. Dann ist $\frac{r}{s}$ ein Näherungsbruch von x, dh. $\frac{r}{s}$ kommt in der Kettenbruchentwicklung von x vor.

Kommen wir nun zur Kernaussage von Wieners Angriff [Bon98, Kap. 3, S. 4f]:

Satz 3.37 (Satz von Wiener)
Sei $N = pq$ ein RSA-Modul und sei (e, d) ein Exponentenpaar mit $1 < d, e < \varphi(N)$ und $ed \equiv 1 \bmod \varphi(N)$.
Wenn $d < \frac{1}{3} N^{\frac{1}{4}}$ und $q < p < 2q$ ist, dann kann Eve nur aus der Kenntnis der Werte für (N, e) effizient den geheimen Exponenten d berechnen.

Beweis
Laut Voraussetung gilt ja

$$\begin{aligned} ed \equiv 1 \bmod \varphi(N) &\Leftrightarrow \exists k \in \mathbb{Z} : \; ed - k\varphi(N) = 1 \\ &\Leftrightarrow \exists k \in \mathbb{Z} : \; \frac{e}{\varphi(N)} - \frac{k}{d} = \frac{1}{d\varphi(N)} \end{aligned}$$

Dh. der Bruch $\frac{k}{d}$ ist eine gute Näherung für $\frac{e}{\varphi(N)}$. Nun kennt Eve zwar $\varphi(N)$ nicht, aber da $\varphi(N)$ in etwa so groß ist wie N, könnte sie im Bruch $\frac{e}{\varphi(N)}$ den Wert N statt $\varphi(N)$ verwenden und sich fragen, was sie über $\frac{e}{N} - \frac{k}{d}$ aussagen kann. Vielleicht ist $\frac{k}{d}$ sogar auch für den neuen Ausdruck $\frac{e}{N}$ eine gute Näherung, den sie kennt bzw. bestimmen kann.
Unser Ziel ist es also $\left|\frac{e}{N} - \frac{k}{d}\right|$ abzuschätzen, in der Hoffnung, dass die beiden Brüche nahe genug beieinander liegen, sodass wegen Satz (3.36) der Bruch $\frac{k}{d}$ in der Kettenbruchentwicklung von $\frac{e}{N}$ vorkommen muss.

Bevor wir zur eigentlichen Abschätzung von $\left|\frac{e}{N} - \frac{k}{d}\right|$ kommen, legen wir uns zuerst ein paar einfachere Ergebnisse zurecht, auf die wir dann im späteren Verlauf der Abschätzung zurückgreifen werden:

- Wenn wir den Fehler, der beim Verwenden von N statt $\varphi(N)$ entsteht, abschätzen wollen, so erhalten wir:

$$\begin{aligned} N - \varphi(N) &= p \cdot q - (p-1)(q-1) = p + q - 1 < && \text{(wegen } p < 2q\text{)} \\ &< 2q + q - 1 < 3q < && \text{(wegen } q < \sqrt{N} < p\text{)} \\ &< 3\sqrt{N} \, . \end{aligned}$$

Insgesamt gilt für den Fehler also:

$$N - \varphi(N) \;<\; 3\sqrt{N}\;. \tag{3.4}$$

- Eine weitere Ungleichung erhalten wir durch:

$$\left.\begin{array}{rcl} k\,\varphi(N) \;=\; ed-1 \;<\; ed \;<\; \varphi(N)\,d & \overset{\cdot\frac{1}{\varphi(N)}}{\Longrightarrow} & k < d \\ \text{laut Voraussetzung gilt:} & & d < \frac{1}{3}N^{\frac{1}{4}} \end{array}\right\} \quad\Rightarrow\quad k \;<\; \frac{1}{3}N^{\frac{1}{4}} \tag{3.5}$$

- Schlussendlich benötigen wir noch:

$$d \;<\; \frac{1}{3}N^{\frac{1}{4}} \quad\overset{\cdot 2}{\Longrightarrow}\quad 2d \;<\; \frac{2}{3}N^{\frac{1}{4}} \;<\; N^{\frac{1}{4}} \quad\Rightarrow\quad \frac{1}{N^{\frac{1}{4}}} \;<\; \frac{1}{2d} \tag{3.6}$$

Nun haben wir alle Ungleichungen zur Verfügung, um $\left|\frac{e}{N} - \frac{k}{d}\right|$ abzuschätzen:

$$\frac{e}{N} - \frac{k}{d} \;=\; \frac{e\,d - k\,N}{N\,d} \;=\; \qquad \big(\text{Zähler erweitern um } \pm k\,\varphi(N)\big)$$

$$\;=\; \frac{\overbrace{ed - k\varphi(N)}^{=1 \text{ lt. Vs.}} - kN + k\varphi(N)}{Nd} \;=\; \frac{1 - k\big(N - \varphi(N)\big)}{Nd}$$

Durch Einführen von Betragsstrichen erhalten wir

$$\left|\frac{e}{N} - \frac{k}{d}\right| \;=\; \left|\frac{1 - k\big(N-\varphi(N)\big)}{Nd}\right| \;=\; \left|\frac{k\big(N-\varphi(N)\big) - 1}{Nd}\right| \;<\; \qquad (\text{da } k>0 \text{ und } N-\varphi(N)>0)$$

$$\;=\; \left|\frac{k\big(N-\varphi(N)\big)}{Nd}\right| \overset{(3.4)}{<} \left|\frac{k\cdot 3\sqrt{N}}{Nd}\right| \overset{(3.5)}{<} \frac{\left(\frac{1}{3}N^{\frac{1}{4}}\right)3\sqrt{N}}{Nd} \;=$$

$$\;=\; \frac{1}{N^{\frac{1}{4}}d} \overset{(3.6)}{<} \frac{1}{2d^2}\;.$$

Insgesamt erhalten wir also die Ungleichung $\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{1}{2d^2}$. Um Satz 3.36 $\big($mit $x{=}\frac{e}{N}$ und $\frac{r}{s}{=}\frac{k}{d}\big)$ anwenden zu können, brauchen wir nur noch zu zeigen, dass $\frac{k}{d}$ ein Bruch in gekürzter Form ist, dh. dass $ggT(k,d) = 1$ ist. Dazu verwenden wir das folgende Lemma:

Lemma 3.38 (Teilbarkeit)
Für alle $a, b, c \in \mathbb{Z}$ gilt:

$$c \,|\, a \quad\wedge\quad c \,|\, b \quad\Rightarrow\quad c \,|\, aX + bY \qquad \forall X, Y \in \mathbb{Z}\;.$$

Dieses Lemma 3.38, angewandt auf die Gleichung $ed - k\,\varphi(N) = 1$, sagt uns, dass jede Zahl, die gleichzeitig k und d teilt, zugleich auch Teiler von 1 sein muss. Dies muss somit auch für den *größten* gemeinsamen Teiler gelten und 1 hat als Teiler nur sich selbst. Daraus folgt $ggT(k,d) = 1$.
Somit sind die Voraussetzungen von Satz 3.36 erfüllt und wir können schließen, dass der Bruch $\frac{k}{d}$ in der Kettenbruchentwicklung von $\frac{e}{N}$ vorkommen *muss*.

Eve braucht somit nur alle Näherungsbrüche $\frac{r_i}{s_i}$ der Kettenbruchentwicklung von $\frac{e}{N}$ der Reihe nach durchzuprobieren. Einer davon muss der „richtige“ sein und Eve braucht nur zu schauen, für *welchen* Näherungsbruch gilt: $k = r_i$ und $d = s_i$.

Wie stellt Eve nun also fest, ob ein Näherungsbruch $\frac{r_i}{s_i}$ der gesuchte Bruch $\frac{k}{d}$ ist, mit dem sich eine abgehörte Nachricht entziffern lässt?

In der Gleichung $ed - k\varphi(N) = 1$ ersetzen wir k durch r_i und d wird ersetzt durch s_i. Dann erhalten wir die Gleichung

$$e\,s_i - r_i\,\varphi(N) = 1\,,$$

in der dann nur noch $\varphi(N)$ eine Unbekannte ist. Auflösen dieser Gleichung nach $\varphi(N)$ liefert uns

$$\varphi(N) = \frac{e\,s_i - 1}{r_i}\ .$$

Mit dem Wert N und dem (möglicherweise richtigen) Wert für $\varphi(N)$ stellen wir nun ein quadratisches Polynom auf, das p und q als Nullstellen besitzt, sofern der richtige Näherungsbruch verwendet wurde. Wir wollen also das Polynom $(X-p)(X-q)$ ausdrücken durch die Werte N und $\varphi(N)$:

$$(X-p)(X-q) \;=\; X^2 + (-p-q)\cdot X + pq \;=\; X^2 + \Big(\varphi(N) - N - 1\Big)\cdot X + N\,. \tag{3.7}$$

Dh. wir bestimmen einfach die beiden Nullstellen $\hat{p}$ und $\hat{q}$ des Polynoms

$$X^2 + \Big(\varphi(N) - N - 1\Big)\cdot X + N$$

und schauen, ob $\hat{p}\cdot\hat{q} = N$ erfüllt ist. Falls $\hat{p}\cdot\hat{q} = N$, dann haben wir eine Faktorisierung von N gefunden.
Falls $\hat{p}\cdot\hat{q} \neq N$, dann probieren wir das gleiche Verfahren für den nächsten Näherungsbruch $\frac{r_{i+1}}{s_{i+1}}$ von $\frac{e}{N}$, solange, bis wir eine Faktorisierung von N erhalten.

□

Der Wiener-Angriff ist deshalb effizient, weil die Anzahl der möglichen Näherungsbrüche für einen RSA-Modul N mit n Binärstellen (dh. $n = \lfloor \log_2 N \rfloor + 1$) nur lineares Wachstum in n besitzt.
Wenn N zum Beispiel 1024/2048/4096 Bits hat, dann muss d mindestens 255/511/1023 Bits haben um dem Angriff von Wiener zu entgehen.

Manchmal lässt sich Wieners Angriff selbst dann durchführen, wenn nicht alle Voraussetzungen von Satz 3.37 erfüllt sind. Ein Beispiel dafür ist in [KK10, Aufgabe 7.9, S. 139f] angeführt: $N = 1966981193543797,\ e = 323815174542919$.
Sollte das Verfahren keine Faktorisierung von N liefern, so wissen wir, dass eine der Voraussetzungen von Satz 3.37 (Satz von Wiener) nicht erfüllt ist.

Wiener selbst hat auch gleich Vorschläge gemacht um seinem Angriff zu entgehen:

Zwar greift bei der Idee, d groß, aber so wählen, dass d_p und d_q zum Entschlüsseln klein sind, die Attacke von Wiener nicht, aber wie schon im Abschnitt 3.2.2 erwähnt lt. [Bon98, S. 5] gibt es einen Angriff um N in $\mathcal{O}\big(\min(\sqrt{d_p}, \sqrt{d_q})\big)$ zu faktorisieren.

Anstatt des öffentlichen Schlüssels e kann man natürlich auch den Exponenten $e' := e + t\cdot v$ für große $t \in \mathbb{N}$ zum Verschlüsseln verwenden. Wiener selbst hat schon gezeigt, dass, wenn $e' > N^{1,5}$ ist, seine Attacke wirkungslos bleibt.

Diese Schranke, die Wiener aufgestellt hat und besagt, dass für ein d mit $d < N^{0.25}$ dieses effizient aus (N, e) bestimmt werden kann, ist offensichtlich keine scharfe Schranke, wie uns der nächste Abschnitt zeigen wird. Boneh und Durfee haben in [BD97] die Schranke auf $d < N^{0.292}$ angehoben. Zugleich vermutet Boneh in [Bon98, S. 5], dass die richtige Schranke bei $d < N^{0,5}$ liegt [Bon98, S. 5].

Eine Beschreibung der Wiener-Attacke findet man auch in [Geh07, Kapitel 4.3.1, S. 61f] bzw. zusätzlich mit einem Beispiel versehen in [Sti06, S. 207-210].

3.3.7.3 Angriffe von Boneh & Durfee - $d < N^{0,292}$

Im letzten Abschnitt 3.3.7.2 (Angriff von M. Wiener) haben wir gesehen, dass wir, wenn $d < \frac{1}{3}N^{0,25}$ und $q < p < 2q$ ist, effizient eine Faktorisierung erhalten können.

Boneh und Durfee haben in [BD97] diese Schranke auf $d < N^{\left(1-\frac{1}{\sqrt{2}}\right)} \approx N^{0,292}$ erweitern können und *glauben*, dass sich die tatsächliche Schranke bei $d < N^{0,5}$ befindet. Um ihren Angriff für $d < N^{0,292}$ durchzuführen, verwenden die beiden Autoren den LLL-Gitterbasisreduktionsalgorithmus (siehe zB. [MvOV97, Kap. 3.10.1, S. 118-121]). In Anlehnung an Wieners Vorschläge, seiner Attacke zu entgehen[Bon98, S. 5], haben Boneh und Durfee gezeigt, dass ihr Angriff für $e < N^{\frac{15}{8}}$ funktioniert, dh. dass man $e > N^{\frac{15}{8}}$ wählen muss, um Boneh und Durfees Angriff zu verhindern.

Hier eine kurze Beschreibung des Angriffs von Boneh und Durfee:

In ihrer Argumentation nehmen die beiden Autoren der Einfachheit halber an, dass $ggT(p-1, q-1) = 2$ ist und dass für die beiden Exponenten e, d gilt:

$$e\,d \equiv 1 \mod \frac{\varphi(N)}{2} \qquad \Leftrightarrow \qquad \exists k \in \mathbb{Z}: \quad e\,d + k\frac{\varphi(N)}{2} = 1\ . \tag{3.8}$$

Mit $\varphi(N) = N - p - q + 1$ lässt sich diese Kongruenz umschreiben auf

$$\exists k \in \mathbb{Z}: \quad e\,d + k\left(\frac{N+1}{2} - \frac{p+q}{2}\right) = 1\ .$$

Mit den Bezeichnungen $s := -\frac{p+q}{2}$ und $A := \frac{N+1}{2}$ gilt dann

$$k(A+s) \equiv 1 \mod e\ .$$

Weiters sei α so definiert, dass für e, den öffentlichen Exponenten, $e = N^{\alpha}$ gilt. Üblicherweise ist e von derselben Größenordnung wie N und somit wird α sehr nahe bei 1 sein. (Sollte α wesentlich kleiner als 1 sein, so würde dies den Angriff sogar noch verstärken.) Für den privaten Exponenten d gelte $d < N^{\delta}$. Dass für $\delta = 0,25$ der geheime Exponent d leicht gefunden werden kann, haben wir schon im vorigen Abschnitt 3.3.7.2 (Angriff von M. Wiener) gesehen.

Aus der Gleichung (3.8) folgt $k = (1-ed)\frac{2}{\varphi(N)}$ und daraus wiederum

$$|k| \;=\; \left|\frac{(1-ed)\cdot 2}{\varphi(N)}\right| \;=\; \frac{(ed-1)\cdot 2}{\varphi(N)} \;<\; \frac{2ed}{\varphi(N)}\ .$$

Da für realistische Anwendungen von RSA für $p, q \in \mathbb{P}$

$$3(p+q) \le N+3 \quad \Leftrightarrow \quad 0 \le N-3p-3q+3 \quad \Leftrightarrow \quad 2N \le 3\big(N-p-q+1\big) = 3\varphi(N) \quad \Leftrightarrow \quad \frac{2}{\varphi(N)} \le \frac{3}{N}$$

gilt, folgt

$$|k| \;<\; \frac{2ed}{\varphi(N)} \;\le\; \frac{3ed}{N}\ .$$

Wegen $N = e^{\frac{1}{\alpha}}$ folgt[9]

$$d < N^{\delta} \quad \Leftrightarrow \quad \frac{d}{N} < N^{\delta-1} = \left(e^{\frac{1}{\alpha}}\right)^{\delta-1} = e^{\frac{\delta-1}{\alpha}} \quad \Leftrightarrow \quad (3e)\cdot\frac{d}{N} < (3e)\cdot e^{\frac{\delta-1}{\alpha}} \quad \Leftrightarrow \quad \frac{3ed}{N} < 3\,e^{1+\frac{\delta-1}{\alpha}}$$

und somit folgt weiters

$$|k| \;<\; \frac{3ed}{N} \;<\; 3\,e^{1+\frac{\delta-1}{\alpha}}\ .$$

[9] Bis zum Ende des aktuellen Abschnittes 3.3.7.3 (Angriffe von Boneh & Durfee) ist mit e^x die x-te Potenz des öffentlichen Exponenten gemeint, und nicht die Exponentialfunktion.

Wegen $q < \sqrt{N} < \frac{p+q}{2} < p < 2q$ gilt[10]

$$|s| = \frac{p+q}{2} < 2q < 2\sqrt{N} = 2N^{\frac{1}{2}} = 2\left(e^{\frac{1}{\alpha}}\right)^{\frac{1}{2}} = 2e^{\frac{1}{2\alpha}} .$$

Mit der Annahme $\alpha = 1$ und dem Weglassen von Konstanten stellt sich uns nun also folgende Aufgabe:
Seien $A, e \in \mathbb{N}$ gegeben und finde dazu passende $k, s \in \mathbb{Z}$, sodass gilt

$$k(A+s) \equiv 1 \bmod e \qquad \text{mit} \qquad |s| < e^{0,5} \quad \text{und} \quad |k| < e^{\delta} .$$

Diese Aufgabenstellung kann man so betrachten: Sei $A \in \mathbb{Z}_e$ gegeben. Finde ein Element 'nahe' bei A, dessen Inverses 'klein' ist mod e. Diese Aufgabenstellung wollen wir also *kleines-Inverses-Problem* nennen. Wenn man für ein gegebenes δ effizient alle Lösungen (k, s) des kleinen-Inversen-Problems aufzählen kann, dann ist RSA mit einem privaten Exponenten $d < N^{\delta}$ unsicher (da man mithilfe von $s = \frac{p+q}{2}$ sofort N faktorisieren kann - siehe (3.7) auf Seite 34). Boneh und Durfee haben eben in [BD97] gezeigt, dass dies für $\delta < 1 - \frac{1}{\sqrt{2}} \approx 0,292$ mithilfe der LLL-Gitterbasisreduktion möglich ist:

In einem ersten Schritt zeigen die beiden Autoren, dass der Angriff erfolgreich ist, wenn gilt

$$\delta < \frac{7}{6} - \frac{1}{3}(1+6\alpha)^{\frac{1}{2}} .$$

Für $\alpha = 1$ erhalten wir die Schranke $d < N^{\frac{7}{6} - \frac{\sqrt{7}}{3}} \approx N^{0,284}$.
Für $\alpha < 1$ wird die Aussage stärker: Ist zum Beispiel $\alpha \approx \frac{2}{3}$, dh. hat e einen Wert von $e \approx N^{\frac{2}{3}}$, dann ist der Angriff erfolgreich für $\delta < \frac{7}{6} - \frac{\sqrt{5}}{3} \approx 0,421$, wobei wegen $e \approx N^{\frac{2}{3}}$ für d zusätzlich noch $d > N^{\frac{1}{3}}$ gelten muss.
Für $\alpha = \frac{15}{8}$ ergibt die Formel $\delta = 0$, weshalb dieser Angriff somit für $e > N^{\frac{15}{8}} = N^{1,875}$ wirkungslos wird, was eine Verbesserung gegenüber dem Wiener-Angriff darstellt, der schon bei $e > N^{1,5}$ nicht mehr greift.
Erst in einem zweiten Schritt zeigen Boneh und Durfee, dass sie für ein beliebiges e die Schranke $d < N^{\left(1 - \frac{1}{\sqrt{2}}\right)} \approx N^{0,292}$ beweisen können.

3.3.7.4 weitere Verbesserungen

Wie in [BD97] angeführt haben Verheul und Tilborg in [VvT97] gezeigt, dass es für $d < N^{0,5}$ einen schnelleren Weg als die vollständige Suche gibt. Jedoch hat dieser Algorithmus für $d > N^{0,25}$ trotzdem exponentielle Laufzeit.

Weiters gibt es einen Angriff, der auf dem LLL-Gitterbasisreduktionsalgorithmus basiert, wenn $N^{0.258} \leq e \leq N^{0.854}$ und $d > e$ ist [LZWD09].

3.3.8 Primfaktoren nahe beieinander

Benne de Weger hat in [dW02] ein paar Verbesserungen schon bekannter Ergebnisse gezeigt, die im Falle zu geringen Abstands Δ zwischen den beiden Primfaktoren des RSA-Moduls $N = pq$ auftreten. Seien β und δ jene Werte, für die

$$\Delta := |p - q| = N^{\beta} \qquad \text{und} \qquad d = N^{\delta}$$

[10] In ihrer Arbeit [BD97] übernehmen die beiden Autoren stillschweigend die Voraussetzung $q < p < 2q$ des Wiener-Angriffs, ohne dies zu erwähnen.

gilt, dann ist mit zu geringem Abstand für p und q gemeint, dass

$$\Delta \leq \sqrt{N} \qquad \Leftrightarrow \qquad \beta \leq \frac{1}{2}$$

ist. Für $\beta \leq \frac{1}{4}$ liegen die beiden Primfaktoren so nahe beieinander, dass die Faktorisierungmethode nach Fermat effizient ist.[11] Dh. in Folge werden uns für β nur noch Werte im Bereich von $\beta \in \left[\frac{1}{4}, \frac{1}{2}\right]$ interessieren.
Ein dabei immer wieder verwendetes Lemma wollen wir an den Anfang dieses Abschnitts stellen:

Lemma 3.39
Sei $N = p\,q$ und sei die Differenz der beiden Primfaktoren $\Delta := |p - q|$. Dann gilt

$$0 \;<\; p + q - 2\sqrt{N} \;<\; \frac{\Delta^2}{4\sqrt{N}}\,. \tag{3.9}$$

Beweis
O.B.d.A. sei $p < q$. Dann folgt aus der immer geltenden Ungleichungskette

$$p \;<\; \sqrt{N} \;<\; \frac{p+q}{2} \;<\; q$$

insbesondere, dass

$$p + q \;>\; 2\sqrt{N} \tag{3.10}$$

ist, was somit die erste Ungleichung $0 < p + q - 2\sqrt{N}$ beweist.[12]
Für die zweite Ungleichung betrachten wir

$$\Delta^2 \;=\; (p-q)^2 \;=\; (p+q)^2 - 4N \;=\; \left((p+q) + 2\sqrt{N}\right)\left((p+q) - 2\sqrt{N}\right)$$

woraus wir

$$p + q - 2\sqrt{N} \;=\; \frac{\Delta^2}{p + q + 2\sqrt{N}} \overset{(3.10)}{<} \frac{\Delta^2}{4\sqrt{N}}$$

schließen können, was gerade die zweite Ungleichung darstellt.

□

Das soeben bewiesene Lemma wird in den nun folgenden Abschnitten immer wieder angewandt werden.

3.3.8.1 Verbesserung des Wiener-Angriffs - B. de Weger

Im Abschnitt 3.3.7.2 haben wir schon den Angriff von M. Wiener kennen gelernt, der eine Faktorisierung von N ermöglicht, sofern $d < \frac{1}{3}N^{\frac{1}{4}}$ ist. B. de Weger hat diese Schranke von im Wesentlichen $\delta < \frac{1}{4}$ auf $\delta < \frac{3}{4} - \beta$ verallgemeinern können [dW02, Kapitel 4, S. 21f]:

Satz 3.40
Sei $N = p \cdot q$ ein RSA-Modul mit $\Delta := |p - q|$, sei $d = N^{\delta}$ der öffentliche Exponent, sei β jener Wert mit $\Delta = N^{\beta}$ und gelte weiters

$$\varphi(N) > \frac{3}{4}N \qquad \text{und} \qquad N > 8d\,.$$

[11] Genauer gesagt zeigt Benne de Weger in [dW02, Kap. 3, S. 20f] für die Fermat-Faktorisierung eine Laufzeit von $\mathcal{O}\left(\frac{\Delta^2}{N^{\frac{1}{2}}}\right)$, was für $\Delta < cN^{\frac{1}{4}}$ bedeutet, dass für eine Zerlegung mittels Fermats Faktorisierungsmethode höchstens $\frac{c^2}{4}$ Versuche benötigt werden. Das heißt, dass für eine kleine (und von N unabhängige) Konstante c die Fermat-Faktorisierung auch noch vernünftig anwendbar ist. Somit kann man in der heutigen Praxis die Grenze auf etwa $\Delta < 1000\,N^{\frac{1}{4}}$ hinausschieben.

[12] Benne de Weger zeigt diese Ungleichung auf anderem Wege.

Dann gilt: Wenn

$$\delta \;<\; \frac{3}{4} - \beta$$

ist, dann kann ein Angreifer N effizient faktorisieren.

Bemerkung 3.41
Dh. je näher die beiden Primfaktoren p und q beieinander liegen, desto mehr Stellen darf d haben um noch immer mit (einer adaptierten Version) der Kettenbruchmethode von Wiener Erfolg zu haben. Ein kleiner, aber entscheidender Unterschied besteht darin, dass im schon aus Abschnitt 3.3.7.2 bekannten Ausdruck

$$\frac{e}{\varphi(N)} - \frac{k}{d} \;=\; \frac{1}{\varphi(N)\,d} \tag{3.11}$$

als Näherung für den Nenner $\varphi(N)$ nicht der Wert N genommen wird, sondern der etwas genauere Wert $\varphi(N) = (p-1)(q-1) = p\,q+1-(p+q) \approx N+1-2\sqrt{N}$. Dh. die Neuerung besteht darin, nicht vom Bruch $\frac{e}{N}$ sondern von $\frac{e}{N+1-2\sqrt{N}}$ die Kettenbruchentwicklung zu betrachten. Unser Ziel wird es somit sein, zu überlegen, wann $\left|\frac{e}{N+1-2\sqrt{N}} - \frac{k}{d}\right| < \frac{1}{2d^2}$ erfüllt ist, denn dann wissen wir, dass der gesuchte Bruch $\frac{k}{d}$ in der Kettenbruchentwicklung von $\frac{e}{N+1-2\sqrt{N}}$ vorkommen muss.

Beweis (von Satz 3.40)
Überlegen wir uns kurz, dass wegen $\varphi(N) = (p-1)(q-1) = N+1-(p+q) \overset{(3.10)}{<} N+1-2\sqrt{N}$ die Ungleichung

$$0 \;<\; \left(N+1-2\sqrt{N}\right) - \varphi(N)\;. \tag{3.12}$$

gilt. Andererseits ist $N+1-\varphi(N) = p+q$. Daraus folgt $N+1-2\sqrt{N}-\varphi(N) \;=\; p+q-2\sqrt{N}$ und durch Anwenden von (3.9) erhalten wir

$$\left(N+1-2\sqrt{N}\right) - \varphi(N) \;<\; \frac{\Delta^2}{4\sqrt{N}}\;. \tag{3.13}$$

Überlegen wir uns nun eine hinreichende Bedingung für $\left|\frac{e}{N+1-2\sqrt{N}} - \frac{k}{d}\right| < \frac{1}{2d^2}$:

$$\begin{aligned}
\left|\frac{e}{N+1-2\sqrt{N}} - \frac{k}{d}\right| \;&=\; \left|\frac{e}{N+1-2\sqrt{N}} - \frac{e}{\varphi(N)} + \frac{e}{\varphi(N)} - \frac{k}{d}\right| \;< && \text{(Dreiecksungleichung)}\\
&<\; e\cdot\left|\frac{1}{N+1-2\sqrt{N}} - \frac{1}{\varphi(N)}\right| + \left|\frac{e}{\varphi(N)} - \frac{k}{d}\right| \;= && \text{(3.11)}\\
&=\; e\cdot\frac{\left|\left(N+1-2\sqrt{N}\right)-\varphi(N)\right|}{\left(N+1-2\sqrt{N}\right)\cdot\varphi(N)} + \frac{1}{\varphi(N)\,d} \;< && \text{(3.12) \& (3.13) \& } e<\varphi(N)\\
&<\; \varphi(N)\cdot\frac{1}{\left(N+1-2\sqrt{N}\right)\cdot\varphi(N)}\cdot\frac{\Delta^2}{4\sqrt{N}} + \frac{1}{\varphi(N)\,d} \;= &&\\
&=\; \frac{1}{\varphi(N)}\left(\underbrace{\frac{\varphi(N)}{\left(N+1-2\sqrt{N}\right)}}_{<1}\cdot\frac{\Delta^2}{4\sqrt{N}} + \frac{1}{d}\right) \;< && \text{wegen } \varphi(N) = N+1-(p+q) < N+1-2\sqrt{N}\\
&<\; \frac{1}{\varphi(N)}\left(\frac{\Delta^2}{4\sqrt{N}} + \frac{1}{d}\right) &&
\end{aligned}$$

Mit den schon bekannten Bezeichnungen $\Delta = N^\beta$ und $d = N^\delta$ und den beiden Voraussetzungen[13]

$$\varphi(N) > \frac{3}{4}N \qquad \text{sowie} \qquad N > 8d$$

gilt dann

$$\begin{aligned}
\left|\frac{e}{N+1-2\sqrt{N}} - \frac{k}{d}\right| &< \frac{1}{\varphi(N)}\left(\frac{\Delta^2}{4\sqrt{N}} + \frac{1}{d}\right) < && \text{wegen } \varphi(N) > \tfrac{3}{4}N \\
&< \frac{4}{3N}\left(\frac{\Delta^2}{4\sqrt{N}} + \frac{1}{d}\right) = \frac{\Delta^2}{3N^{\frac{3}{2}}} + \frac{4}{3Nd} = \frac{N^{2\beta}}{3N^{\frac{3}{2}}} + \frac{4}{3Nd} < && \text{wegen } N > 8d \\
&< \frac{N^{2\beta}}{3N^{\frac{3}{2}}} + \frac{4}{3\cdot 8d\cdot d} = \frac{N^{2\beta-\frac{3}{2}}}{3} + \frac{1}{6d^2} = \frac{N^{2\beta-\frac{3}{2}}}{3} + \frac{1}{6N^{2\delta}}\,.
\end{aligned}$$

Wenn wir nun über den Exponenten $2\beta - \frac{3}{2}$ die zusätzliche Voraussetzung

$$2\beta - \frac{3}{2} < -2\delta \qquad \Leftrightarrow \qquad \delta < \frac{3}{4} - \beta$$

annehmen, dann können wir folgern, dass $N^{2\beta-\frac{3}{2}} < N^{-2\delta}$ und insgesamt erhalten wir damit

$$\begin{aligned}
\left|\frac{e}{N+1-2\sqrt{N}} - \frac{k}{d}\right| &< \frac{N^{2\beta-\frac{3}{2}}}{3} + \frac{1}{6N^{2\delta}} < \\
&< \frac{N^{-2\delta}}{3} + \frac{1}{6}N^{-2\delta} = \frac{1}{2}N^{-2\delta} = \frac{1}{2d^2}\,.
\end{aligned}$$

Dh. unter der Voraussetzung $\delta < \frac{3}{4} - \beta$ wissen wir, dass wegen Satz 3.36 auf Seite 32 der gesuchte Bruch $\frac{k}{d}$ als Näherungsbruch der Kettenbruchentwicklung von $\left|\frac{e}{N+1-2\sqrt{N}} - \frac{k}{d}\right|$ vorkommen muss. Und im Beweis von Satz 3.37 (Satz von Wiener) haben wir schon gesehen, wie wir die Näherungsbrüche berechnen und testen können, ob ein Näherungsbruch jener gesuchte Näherungsbruch ist, mit dem sich N faktorisieren lässt.

□

3.3.8.2 Verbesserung der Boneh/Durfee-Angriffe - B. de Weger

Wiederum gelten im Folgenden auch hier die Gleichungen $\Delta := |p-q| = N^\beta$ und $d = N^\delta$.

Benne de Weger hat auch für die beiden Ergebnisse von Boneh und Durfee (siehe Kapitel 3.3.7.3 auf Seite 36) Verbesserungen bzw. Verallgemeinerungen beschrieben:

Das erste Ergebnis von Boneh/Durfee $\left(d < N^{0{,}284\ldots}\right)$ konnte de Weger auf

$$\delta < \frac{1}{6}(4\beta+5) - \frac{1}{3}\sqrt{(4\beta+5)(4\beta-1)}$$

[13] Die erste der beiden Bedingungen ist quasi immer erfüllt. Sonst müsste der größere der beiden Primfaktoren in der Größenordnung von N sein und dann wäre der kleinere der beiden Primfaktoren somit durch Probedivision leicht zu bestimmen: Denn angenommen es gälte

$$\begin{aligned}
\frac{3}{4}N &> \varphi(N) = pq - p - q + 1 = N - (p+q) + 1 \\
\Leftrightarrow \qquad p+q-1 &> \frac{1}{4}N
\end{aligned}$$

und o.B.d.A sei $p > q$. Da $p+q$ in der Größenordnung von p ist, müsste somit p in der Größenordnung von N sein, was wiederum heißen würde, dass q durch Probedivision leicht zu finden wäre.

verbessern, was für $\beta = \frac{1}{2}$ die schon bekannte Schranke $\left(\delta < \frac{7}{6} - \frac{\sqrt{7}}{3} \approx 0,284...\right)$ von Boneh/Durfee darstellt.

Die zweite Schranke von Boneh/Durfee $\left(d < N^{0,292...}\right)$ konnte de Weger auf

$$\delta \ < \ 1 - \sqrt{2\beta - \frac{1}{2}}$$

erweitern, wobei wir auch hier wieder für $\beta = \frac{1}{2}$ das ursprüngliche Ergebnis von $\delta < 1 - \frac{1}{\sqrt{2}} \approx 0,292...$ erhalten.

Für nähere Informationen sei auf [dW02, Kapitel 5 u. 6, S. 22-25] verwiesen.

3.3.9 Signaturfalle

Der Angriff *Signaturfalle*, oder auch *blinding* genannt, ist nicht so sehr ein Angriff auf das mathematische Grundprinzip hinter RSA, sondern vielmehr auf die menschliche Komponente beim Verschlüsseln. Da das Signieren mittels RSA genau die inverse Operation zum Verschlüsseln mit RSA darstellt, kann Mallory als Angreifer versuchen Alice dazu zu bringen, ein an sie geschicktes und von Mallory mitgehörtes Chiffrat C zu signieren, wodurch Alice C dechiffrieren würde. Da das Signieren in der Regel eine „sinnlose“ Zeichenkette ergibt, würde es Alice sofort auffallen, wenn nach dem Erstellen einer Signatur auf eine von Mallory vorgelegte Zeichenfolge das Ergebnis ein sinnvoller Text (den sie früher schon einmal empfangen hat) wäre. Deshalb kann Mallory versuchen das Chiffrat C zu verschleiern in der Hoffnung, dass Alice die Signatur nicht verdächtig vorkommt. Dazu kann Mallory wie folgt vorgehen:
Mallory wählt ein beliebiges $r \in \mathbb{Z}_N \backslash \{0\}$ und multipliziert die e-te Potenz von r mit dem abgehörten Chiffrat C und erhält so eine neue Restklasse $\widetilde{C}$ mit

$$\widetilde{C} \ := \ r^e \cdot C \bmod N\,.$$

Aufgrund der Homomorphieeigenschaft der Modulorechnung gilt

$$\widetilde{C} \ \equiv \ r^e \cdot C \ \equiv \ r^e \cdot M^e \ = \ (r \cdot M)^e \mod N\,.$$

Wenn Mallory also nun Alice $\widetilde{C}$ zum Signieren vorlegt, dann erhält Alice als Signatur S

$$S \ := \ \widetilde{C}^d \ \equiv \ \left((rM)^e\right)^d \ = \ (rM)^{ed} \ \equiv \ rM \mod N\,.$$

Und da r zufällig gewählt war, kann man davon ausgehen, dass die Signatur $S = rM$ zufällig und somit auch unauffällig aussieht. Sollte Alice diese Signatur also freigeben, so kann sich Mallory den Klartext M wie folgt berechnen.
Dazu versucht Mallory sich zu r das modulare Inverse (mod N) auszurechnen. Für den unwahrscheinlichen Fall, dass das Inverse nicht existieren sollte, kann dies nur deshalb passieren, weil $ggT(r, N) > 1$ ist, was zu einer Faktorisierung von N führen würde. Im Regelfall $ggT(r, N) = 1$ gibt es also so ein modulares Inverses r^* mit $r^* r \equiv 1 \bmod N$. Somit braucht Mallory nur noch die Signatur S mit dem Inversen r^* multiplizieren und kann sich aufgrund

$$r^* \cdot S \ \equiv \ r^* \cdot (rM) = \underbrace{r^* r}_{\equiv 1 \bmod N} \cdot M \ \equiv \ M \mod N$$

den Klartext M berechnen.
Die Bezeichnung *blinding* kommt daher, dass ein Angreifer dem Signaturersteller nicht jenen Text vorlegt, von dem er eigentlich die Signatur (=Entschlüsselung) haben möchte, nämlich C, sondern den eigentlichen Zweck durch Multiplikation mit r^e verschleiert und so den RSA-Teilnehmer „blendet“.

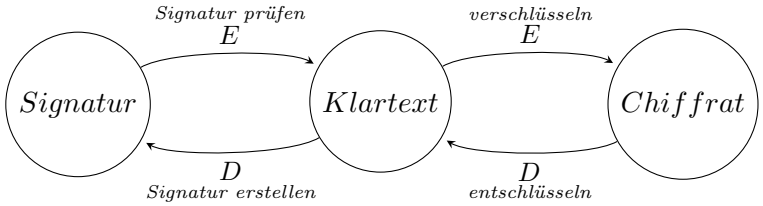


Abbildung 3.2: verschiedene Bezeichnungen für die Funktionen E und D

Dh. ein Angreifer kann versuchen, sich mit dieser Methode eine abgehörte Nachricht vom rechtmäßigen Empfänger selbst entschlüsseln zu lassen, ohne dass sich dieser dessen bewusst ist.

Bemerkung 3.42
Das Anwenden der Operation $D: x \mapsto x^d \bmod N$ hat in der Praxis abhängig vom Input x unterschiedliche Bezeichnungen: Wenn wir D auf ein Chiffrat C anwenden, so heißt dieser Vorgang natürlich *entschlüsseln*. Wenn D aber auf einen Klartext M angewandt wird, so nennen wir dies eine *Signatur erstellen* (vgl. Abbildung 3.2).

Auf genau demselben mathematischen Hintergrund beruhen auch die beiden folgenden Abwandlungen dieses Angriffs, die sich nur in der praktischen Durchführung, nicht aber im mathematischen Hintergrund unterscheiden:

Angenommen Mallory hat ein Chiffrat C, das an Alice gesandt wurde, mitgehört und möchte wissen, was die Botschaft dahinter ist. Anstatt Alice um eine Signatur zu bitten, kann Mallory auch einfach die Verschleierung $\widetilde{C} := r^e \cdot C$ an Alice senden, so als wäre es eine (mit ihrem öffentlichen Schlüssel chiffrierte) Nachricht an sie. Alice versucht natürlich die Botschaft zu entschlüsseln, erhält aber nur eine sinnlose Zeichenkette, nämlich $r \cdot M$. Mallory kann nun Alice bitten, ihm zur Analyse eines scheinbar aufgetretenenen Übertragungsfehlers die erhaltene Zeichenkette zurück zu senden. Wenn sie das tut, so kann Mallory M berechnen.

In der zweiten Variante löscht Alice nach dem fehlgeschlagenen Entschlüsseln das (für sie sinnlose) Ergebnis $r \cdot M$. Wenn sich Mallory nun Zugang zu ihrem Computer verschaffen kann, so kann er vermutlich auch Zugriff auf das Ergebnis der fehlgeschlagenen Entschlüsselung erlangen[14] und mit genau denselben mathematischen Mitteln den Klartext M entziffern.

Um manchen dieser Angriffe zu entgehen, sollte man deshalb zwei unterschiedliche Moduli (je einen Modul zum Entschlüsseln und einen zum Signatur erstellen) verwenden.

Je nach praktischen Umständen (zur Signatur vorlegen; als scheinbar normale Nachricht schicken oder sich Zugriff auf den Papierkorb verschaffen) bzw. abhängig davon, was man als Angreifer erreichen möchte (ein Chiffrat entschlüsseln (lassen) oder eine Signatur berechnen (lassen)) ist diese mathematische Vorgehensweise benannt als *Signaturfalle* [Pom03, Kapitel 2.9, S. 38], *blinding* [Bon98, S. 4], *Davidas Attacke* oder auch *verbesserter Angriff von Judy Moore* [BB05, Kapitel 2.4.2, S. 10f], [SDO06, S. 669].

3.3.10 Angriffe auf Implementierungen von RSA

Im Folgenden wollen wir uns den sog. *side-channel-Attacken* widmen, die im Gegensatz zu den bisher genannten Angriffen nicht die Mathematik hinter RSA bzw. auf die Auswahl der

[14] Die gängigsten Betriebssysteme der heutigen Zeit löschen eine Datei nicht sofort, sondern verschieben die Datei nur in einen speziellen Ordner (zB. Papierkorb), um dem User die Möglichkeit einer Wiederherstellung zu bieten. Selbst bei einem sofortigen Löschen besteht noch immer die Chance mit einem Recovery-Tool eine gelöschte Datei wiederherzustellen.

Parameter abzielt, sondern die die Umsetzung in die Praxis angreifen.

3.3.10.1 Timing-Attacke - Kocher

Stellen wir uns vor, dass ein Angreifer für gewisse Zeit Zugriff auf eine Smart-Card hat, die die Entschlüsselung eines RSA-Kryptosystems vornimmt, wobei man aber aus der Konstruktion der Smart-Card selbst weder auf den geheimen Exponenten noch auf die beiden Primfaktoren schließen kann. Um auch künftige Nachrichten mitlesen zu können hat Kocher 1996 die sog. *Timing-Attacke* vorgestellt, die Bit für Bit den privaten Exponenten d berechnet. Dazu muss der Angreifer in der Lage sein, beliebig viele Entschlüsselungen durchführen zu können und deren Zeitverbrauch zu messen.

Wollen wir also mit $C_1, C_2, \ldots, C_S \in \mathbb{Z}_N$ eine riesige Anzahl an Werten bezeichnen, auf die der Entschlüsselungsalgorithmus der Smart-Card angewandt werden soll, sowie mit T_i die tatsächlich von der Smart-Card benötigten Zeiten für die Berechnung von ${C_i}^d \mod N$ (für $i = 1, \ldots, S$), welche wir uns als Erstes berechnen wollen. Weiters sei $d = (d_n\, d_{n-1} \ldots d_1\, d_0)_2$ die Binärdarstellung des privaten Exponenten, dh. es gelte $d = \sum_{i=0}^{n} d_i\, 2^i$.

Der Angriff macht sich zu Nutzen, dass RSA erst durch Anwenden eines Algorithmus' zum schnellen Potenzieren (siehe Kap. 3.1.3 bzw. 4.1.3) praktikabel wird. Da d ungerade ist, wissen wir, dass $d_0 = 1$ sein muss. Also wollen wir uns als Angreifer als nächstes Bit das Bit d_1 berechnen. Wir wissen zwar nicht, ob $d_1 = 1$ ist und somit im square & multiply-Schritt die optionale Multiplikation auslöst, aber wir wissen, dass im Falle $d_1 = 1$ diese zusätzliche Multiplikation

$$C_i \cdot {C_i}^2 \mod N \tag{3.14}$$

lauten muss. Also lassen wir uns (abhängig von den physikalischen Spezifikationen der Smart-Card) für all unsere C_i (von einem anderen System als der Smart-Card) diese Werte berechnen und bezeichnen die Zeit, die benötigt wurde um (3.14) zu berechnen, mit $t_{i,(1)}$ für $i = 1, \ldots, S$.

Da die Zeit zur Berechnung einer modularen Multiplikation von den beiden Faktoren abhängt, sind die Werte $t_{i,(1)}$ unterschiedlich groß. Wenn nun $d_1 = 1$ ist, so hat Kocher beobachtet, dass zwischen den Werten $\{T_i\}$ und $\{t_{i,(1)}\}$ eine Korrelation besteht [Bon98, S. 12]. Im Falle $d_1 = 0$ verhalten sich die beiden Wertesampel unabhängig. Somit konnten wir nun d_1 bestimmen.

Als nächsten Schritt wollen wir uns d_2 berechnen. Dazu betrachten wir für all unsere C_i jeweils wieder die optionale Multiplikation

$$\underbrace{\left({C_i}^{d_0} \cdot \left({C_i}^2\right)^{d_1}\right)}_{\text{vorausgewertet}} \cdot \underbrace{\left(\left({C_i}^2\right)^2\right)}_{\text{vorausgewertet}} \mod N\,, \tag{3.15}$$

die beim nächsten Durchlauf der square&multiply-Schleife auftreten kann, wobei die beiden Faktoren - abhängig von der Implementierung auf der Smart-Card eventuell - schon als Zahl in $\mathbb{Z}_N$ vorausgewertet vorliegen sollen. Die so erhaltenen Zeiten $\left\{t_{i,(2)} \;\middle|\; i = 1, \ldots, S\right\}$ werden wiederum auf ihre Korrelation mit $\{T_i\}$ getestet, woraus wir auf das Bit d_2 schließen können.

Dieses Verfahren führen wir für $k \leq n$ sukzessive fort, indem wir das k-te Bit d_k von d durch Berechnung der Korrelation zwischen den Zeiten $\{t_{i,(k)} \mid i = 1, \ldots, S\}$ für die optionalen Multiplikationen $\left(\prod_{j=0}^{k-1} {C_i}^{2^j \cdot d_j}\right) \cdot \left({C_i}^{2^k}\right)$ und den Zeiten $\{T_i\}$ bestimmen. So können wir uns Bit für Bit die Binärdarstellung von d berechnen.

Wie Boneh in [Bon98, S. 13] erwähnt, kann man sich im Falle eines kleinen öffentlichen Exponenten e den Satz 3.30 zunutze machen und braucht dann nur $\left\lceil \frac{S}{4} \right\rceil$ der Bits von d mittels des soeben beschriebenen Verfahrens berechnen.

Dieser Angriff wird sowohl in [Bon98, Kap. 5.1, S. 12f] als auch in [SDO06, S. 670] dargestellt.

3.3.10.2 Power-consumption-Attacke - Kocher

Bei der *Power-Attacke* nehmen wir an, dass der Angreifer Zugang zur Hardware hat, mit der entschlüsselt wird. Wie im vorigen Abschnitt können wir uns vorstellen, dass der Angreifer zum Beispiel im Besitz einer Smart-Card ist, auf der ein privater RSA-Schlüssel implementiert ist. Auch bei diesem Angriff ist es das Ziel, die Binärdarstellung des privaten Exponenten d herauszufinden. Wiederum nützen wir die unterschiedlich langen Zeiten für einen Schleifendurchlauf des square & multiply-Algorithmus' aus, die von den Bits d_i abhängig sind. Im Vergleich zum Abschnitt 3.3.10.1 messen wir diesmal den variablen Stromverbrauch während einer Entschlüsselung bzw. Signaturerstellung. Durch regelmäßige Muster im zeitlich abhängigen Energieverbrauch erkennt man die Schleifendurchläufe wieder und aufgrund eines höheren Stromverbrauchs im Falle einer zusätzlichen Multiplikation bzw. den unterschiedlich langen Runden kann man Rückschlüsse auf die Bitfolge des Exponenten d ziehen. [Pom03, S. 40]

3.3.10.3 Zufällige Fehler

Der folgende Angriff verwendet die Tatsache, dass man beim Erstellen einer Signatur (genauso wie auch beim Entschlüsseln) häufig aus Zeitersparnisgründen den chinesischen Restsatz anwendet, wozu die beiden (geheimen) Primfaktoren p, q benötigt werden. Im Falle eines Rechenfehlers führt die Veröffentlichung der falschen Signatur zu einer Faktorisierung des Moduls $N = p\,q$.

Eigentlich müsste Alice bei der Signaturerstellung den Wert

$$S := M^d \mod N$$

tatsächlich modulo N berechnen. Um dies zu beschleunigen, kann sie, da sie p und q kennt, auch folgende Rechnung anstellen:

$$d_p := d \mod (p-1) \qquad \text{und} \qquad d_q := d \mod (q-1)$$

$$S_p := M^{d_p} \mod p \qquad \text{und} \qquad S_q := M^{d_q} \mod q\,.$$

Mittels des chinesischen Restsatzes berechnet sich Alice nun die beiden (eindeutigen) Werte $T_1, T_2 \in \mathbb{Z}_N$ mit

$$T_1 \equiv \begin{cases} 1 \mod p \\ 0 \mod q \end{cases} \qquad \text{und} \qquad T_2 \equiv \begin{cases} 0 \mod p \\ 1 \mod q \end{cases}$$

und erhält dann die Signatur S durch

$$S \;:=\; T_1\,S_p + T_2\,S_q \mod N\,.$$

Durch diese Vorgehensweise kann man die Berechnung der Signatur insgesamt etwa um den Faktor 4 beschleunigen [Bon98, S. 13].

Wenn man den Chinesischen Restsatz verwendet und sich aus den dabei verwendeten (teilerfremden) Moduli den geheimen Schlüssel berechnen kann (was bei RSA zutrifft), dann besteht die Gefahr, dass mit einem einzigen Rechenfehler durch Veröffentlichung der falschen Signatur indirekt der geheime Schlüssel Preis gegeben wird [Bon98, s. [3]].

Schauen wir uns an, wie so ein Angriff konkret aussieht:

Angenommen der Fehler passiert bei der Berechnung von genau einem der beiden Werte S_p oder S_q. Sei also o.B.d.A. S_p richtig berechnet worden und sei das falsche Ergebnis

$$\widetilde{S}_q \not\equiv S_q \mod q\,.$$

Anstatt S zu berechnen, wird als Signatur der falsche Wert

$$\widetilde{S} \;:=\; T_1 S_p + T_2 \widetilde{S}_q \mod N$$

weitergegeben bzw. veröffentlicht. Mit der zu signierenden Nachricht M und der falschen Signatur $\widetilde{S}$ kann ein Angreifer durch Anwenden des öffentlichen Schlüssels (N, e) sofort feststellen, dass ein Rechenfehler passiert sein muss, denn er erhält dann

$$\left(\widetilde{S}\right)^e = \left(T_1 S_p + T_2 \widetilde{S}_q\right)^e = (T_1 S_p)^e + \sum_{i=1}^{e-1} \underbrace{(T_1 S_p)^{e-i} \left(T_2 \widetilde{S}_q\right)^i}_{\substack{\text{jeder dieser Summanden ist ein} \\ \text{Vielfaches von } T_1 T_2}} + \left(T_2 \widetilde{S}_q\right)^e .$$

Da T_1 ein Vielfaches von q und T_2 ein Vielfaches von p ist, folgt, dass das Produkt $T_1 T_2$ ein Vielfaches von N ist. Jeder der obigen Summanden, die ein Vielfaches von $T_1 T_2$ sind, verschwindet also, wenn man modulo N rechnet. Insgesamt erhalten wir somit:

$$\left(\widetilde{S}\right)^e \equiv (T_1 S_p)^e + \left(T_2 \widetilde{S}_q\right)^e \mod N\,.$$

Diese Kongruenz gilt dann natürlich auch (mod p) und (mod q). Zusätzlich gilt aber auch: $T_1 \equiv 0 \pmod q$ und $T_2 \equiv 0 \pmod p$. Somit erhalten wir:

$$\left(\widetilde{S}\right)^e \equiv \underbrace{T_1^e}_{\equiv 1 \bmod p} S_p^e + \underbrace{T_2^e}_{\equiv 0 \bmod p} \left(\widetilde{S}_q\right)^e \equiv S_p^e = \left(M^{d_p}\right)^e \mod p$$

und

$$\left(\widetilde{S}\right)^e \equiv \underbrace{T_1^e}_{\equiv 0 \bmod q} S_p^e + \underbrace{T_2^e}_{\equiv 1 \bmod q} \left(\widetilde{S}_q\right)^e \equiv \left(\widetilde{S}_q\right)^e \mod q\,.$$

Weiters gilt: $M^{ed} \equiv M \mod N \;\Rightarrow\; M^{ed} \equiv M \mod p$ und wegen $d \equiv d_p \mod p-1$ folgt

$$M^{e \cdot d_p} \equiv M \mod p\,.$$

Kurz gesagt gilt $\left(\widetilde{S}\right)^e \equiv M \mod p$, dh. $\left(\widetilde{S}\right)^e - M$ ist ein Vielfaches von p.
Da q eine ungerade Primzahl ist, folgt aus $\widetilde{S} \equiv \widetilde{S}_q \not\equiv S_q \equiv M^{d_q} \pmod q$, dass auch $\left(\widetilde{S}\right)^e \not\equiv \left(M^{d_q}\right)^e \equiv M \pmod q$ ist, dh. dass $\left(\widetilde{S}\right)^e - M$ kein Vielfaches von q ist.

Insgesamt hat daraufhin ein Angreifer einen Wert $\left(\widetilde{S}\right)^e - M$ zur Verfügung, der ein Vielfaches von p aber kein Vielfaches von q ist. Somit liefert

$$ggT\Big(\left(\widetilde{S}\right)^e - M, N\Big) \;=\; p$$

einem Angreifer den nicht-trivialen Teiler p von N [Bon98, Kap. 5.2, S. 13f].

Um dieser Angriffsmöglichkeit Einhalt zu gebieten, kann Alice einfach nach Berechnung der Signatur kontrollieren, ob für ihr Ergebnis $\widetilde{S}$ die Kongruenz $\left(\widetilde{S}\right)^e \equiv M \pmod N$ erfüllt ist.

Für den soeben beschriebenen Angriff muss ein Angreifer die komplette Nachricht M kennen. Eine andere Möglichkeit, um dieser Attacke vorzubeugen, besteht deshalb darin, dass Alice die Nachricht M um ein paar zufällige Zeichen ergänzt (random padding), die sie geheim hält. Sollte bei der Signaturerstellung kein Fehler passieren, so besteht auch keine Gefahr der Faktorisierung, da $\left(\widetilde{S}\right)^e - M$ ein Vielfaches von N ist und somit der ggT mit N keinen Primfaktor Preis gibt. Sollte aber bei der Signaturerstellung doch ein Fehler passiert sein, so kennt der Angreifer aufgrund des Paddings nicht den fehlerhaft signierten Klartext und kann nicht die zur Berechnung notwendige Differenz bilden.

3.3.11 Bleichenbachers Attacke auf PKCS 1

Bei einer alten Version des „Public Key Cryptography Standard 1“ (PKCS 1) von „RSA Laboratories“ [RL02, Kapitel 7.2.1, S. 23] musste die Nachricht - mit dem Hintergedanken der Sicherheit - an den führenden Stellen gepaddet, dh. mit zufälligen Zeichen aufgefüllt werden. Damit der Empfänger der Nachricht nach dem Entschlüsseln weiß, welche Stellen nicht zur eigentlichen Nachricht gehören und welcher Teil des Klartextes die Botschaft enthält, wurde der Beginn des Blocks der zufälligen Zeichen mit den alpha-numerischen Zeichen „02“ gekennzeichnet und das Ende dieses Blocks mit „0“ (und in der zufälligen Zeichenfolge durfte natürlich das 0-Byte nicht vorkommen). Bei Verwendung der Byte-Schreibweise heißt das, dass die 16 führenden Bit genau `00000000 00000010` sein mussten - vgl. Abbildung 3.3.

$$M \;:=\; \big(00000000\,||\,00000010\,||\,\text{random}\,||\,00000000\,||\,\text{eigentliche Nachricht}\big)_2$$

Abbildung 3.3: RSA - Klartext im PKCS 1-Format

Nehmen wir an, das Empfangen und Entschlüsseln erledigt eine Maschine automatisch für Alice (zB. ein Mailserver). Wenn nun eine Nachricht nach dem Entschlüsseln nicht dem PKCS 1-Format entspricht, senden manche Anwendungen eine Fehlermeldung zurück, dass kein gültiger Geheimtext empfangen wurde.
Bleichenbacher [Ble98] hat gezeigt, dass diese Fehlermeldung die Achillesferse darstellt, mit der man RSA brechen kann.

Mallory kann das auf folgende Art und Weise ausnutzen [Bon98, Kap. 5.3, S. 14]:
Er kann eine abgehörte Nachricht C, die für Alice bestimmt war und die er entziffern möchte, mit einem von ihm beliebig gewählten r multiplizieren und das so erhaltenen

$$C' := r\,C \mod N$$

an Alices RSA-Empfangs- und Dechiffriermaschine schicken [KL08, S. 363]. Eine Anwendung auf ihrer Maschine empfängt und dechiffriert die Nachricht. Nun können 2 Fälle auftreten:

1. Fall: Das Programm auf Alices Maschine sendet eine Fehlermeldung zurück, weil nach der Entschlüsselung von C' die beiden führenden Stellen ungleich „02“ waren.
2. Fall: Sollte C' zufälligerweise korrekt formatiert sein, so sendet Alices Maschine keine Fehlermeldung aus.

Je nachdem, ob die Maschine eine Fehlermeldung zurücksendet oder nicht, kann Mallory also darauf schließen, ob die führenden 16 Bit der Entschlüsselung von C' aufgefasst als ASCII-codierte Zeichenfolge genau „02“ entsprechen oder nicht. Somit hat Mallory ein Orakel für alle von ihm beliebig gewählten r zur Verfügung. Bleichenbacher hat gezeigt, dass man mit solch einem Orakel die Nachricht C entziffern kann.

3.3.12 Message concealing

Für jeden öffentlichen RSA-Schlüssel (N, e) gibt es Klartexte $M \in \mathbb{Z}_N$, die beim „verschlüsseln“, also beim Abbilden auf $M^e \mod N$ auf sich selbst abgebildet werden. Ingesamt gibt es

$$\big(1 + ggT(e-1, p-1)\big) \cdot \big(1 + ggT(e-1, q-1)\big)$$

solche Nachrichten M, für die

$$M^e \equiv M \mod N$$

gilt [MvOV97, S. 290]. Die Nachrichten

$$M \equiv 1 \mod N, \qquad M \equiv 0 \mod N \qquad \text{und} \qquad M \equiv -1 \mod N$$

sind für jeden Schlüssel (N, e) *Fixpunkte* der Verschlüsselung(sabbildung). Bei realistischer und zufälliger Wahl der Primzahlen $p, q \in \mathbb{P}$ sowie von e ist der Anteil dieser „nicht verschlüsselbaren“ Nachrichten zu vernachlässigen.

3.3.13 Faktorisieren von N

Da das Faktorisieren des Moduls N der bekannteste und zugleich auch ein *immer* einsetzbarer Angriff auf RSA ist, wollen wir diese Angriffsmöglichkeit etwas ausführlicher betrachten.

Hat ein Angreifer die beiden Primfaktoren p und q einmal herausgefunden, so kann er sich, genauso wie Alice, $v := kgV(p-1, q-1)$ berechnen und anschließend unter Zuhilfenahme des Erweiterten Euklidischen Algorithmus' (Alg. 3.8) ein modulares Inverses zu $e \pmod v$ bestimmen und somit den geheimen Schlüssel berechnen.
Damit kann der Angreifer dann nicht nur eine einzige Nachricht entziffern, sondern auch *alle* schon bisher mitgehörten und zukünftigen Nachrichten, die mit diesem Modul N verschlüsselt wurden bzw. werden. (Hat ein Angreifer einmal den Modul N faktorisiert, so wäre ein bloßes Wechseln des Schlüsselpaares (e, d) sinnlos, da sich der Angreifer dann zu einem neuen $\tilde{e}$ genauso leicht wieder das zugehörige $\tilde{d}$ mit $\tilde{e}\,\tilde{d} \equiv 1 \mod v$ berechnen könnte.)
Obwohl auf dem Gebiet der Faktorisierung natürlicher Zahlen schon seit der Antike geforscht wird, ist bis heute noch kein Polynomialzeitalgorithmus zur Bestimmung einer nicht-trivialen Zerlegung einer beliebigen Zahl bekannt. Vielmehr noch: wir wissen bis heute noch nicht einmal, ob ein Polynomialzeitalgorithmus überhaupt existiert. Die Entdeckung eines polynomiellen Faktorisierungsalgorithmus' hätte weitreichende und schwerwiegende Konsequenzen für die moderne Kryptographie, da mit ihr das bekannteste und wichtigste Public-Key-Kryptosystem (RSA) nicht mehr sicher wäre. Andererseits ist aber auch noch unklar, ob man, um RSA zu brechen, einen zur Faktorisierung des Moduls N äquivalenten Aufwand treiben muss, oder ob es einen einfacheren Weg gibt, der bisher bloß noch nicht entdeckt wurde. Und selbst wenn wir irgendwann einmal zeigen können sollten, dass das RSA-Problem äquivalent zum Faktorisierungsproblem ist, was allgemein vermutet wird [MvOV97, S. 99], dann wüssten wir noch immer nicht, ob es für die Aufgabe des Faktorisierens (und somit auch für das RSA-Problem) nicht doch einen Polynomialzeitalgorithmus gibt. Beide Beweise sind im Moment noch ausständig. Dies sind also 2 *Annahmen*, auf denen die Sicherheit von RSA beruht.

Es wurden schon viele Arbeiten geschrieben, die ausschließlich Faktorisierungsmöglichkeiten zum Thema haben. Im Nachfolgenden werde und muss ich mich deshalb auf die Speerspitze der Faktorisierungsmethoden beschränken:

3.3.13.1 Zahlkörpersieb

Da das *(allgemeine) Zahlkörpersieb (engl.: (general) number field sieve - NFS)* momentan das asymtotisch schnellste Verfahren zur Faktorisierung eines RSA-Moduls N darstellt, möchte ich darauf etwas genauer eingehen:

Faktorisieren mit quadratischen Kongruenzen
Das Zahlkörpersieb gehört zur Gruppe jener Faktorisierungsmethoden, die sich quadratischer Kongruenzen bedienen. Die Grundidee hinter dieser Gruppe von Faktorisierungsalgorithmen ist, zwei ganze Zahlen $x, y \in \mathbb{Z}_N$ zu finden, deren Quadrate modulo der zu faktorisierenden Zahl N in derselben Restklasse liegen, dh. die

$$x^2 \equiv y^2 \mod N \tag{3.16}$$

erfüllen und für die zugleich auch noch

$$x \not\equiv \pm y \mod N \tag{3.17}$$

gilt. Aus der ersten Bedingung (3.16) folgt, dass die Differenz der Quadrate ein Vielfaches von N sein muss:

$$x^2 \equiv y^2 \mod N \quad \Leftrightarrow \quad x^2 - y^2 \equiv 0 \mod N \quad \Leftrightarrow \quad \exists k \in \mathbb{Z} : \; x^2 - y^2 = kN \quad \Leftrightarrow \quad N \,|\, x^2 - y^2 \; .$$

Mit Hilfe der Gleichung $x^2 - y^2 = (x-y)(x+y)$ können wir das Vielfache von N als Produkt zweier Faktoren darstellen und erhalten

$$N \,|\, (x-y)(x+y) \; . \tag{3.18}$$

Die Idee, um zu einer Faktorisierung zu kommen, ist, zu verhindern, dass die beiden Primfaktoren p und q von $N = p \cdot q$ beide im selben Faktor von $(x-y)(x+y)$ zusammenfallen. Also darf N kein Teiler von $(x-y)$ sein, dh. es muss

$$N \nmid (x-y) \quad \Leftrightarrow \quad x - y \not\equiv 0 \mod N \quad \Leftrightarrow \quad x \not\equiv y \mod N \,,$$

gelten und analog darf N kein Teiler von $(x+y)$ sein, was

$$N \nmid (x+y) \quad \Leftrightarrow \quad x + y \not\equiv 0 \mod N \quad \Leftrightarrow \quad x \not\equiv -y \mod N$$

bedeutet. Das wird aber schon durch die zweite Bedingung (3.17) garantiert.
Somit sind die beiden Primfaktoren p und q auf die beiden Faktoren $(x-y)$ und $(x+y)$ aufgeteilt. Also liefert uns sowohl $ggT(x-y, N)$ als auch $ggT(x+y, N)$ jeweils einen nichttrivialen Faktor von N.

Der Unterschied zwischen den einzelnen Faktorisierungsmethoden, welche alle quadratische Kongruenzen verwenden, besteht in der Art und Weise, *wie* man auf die beiden gesuchten Zahlen x und y kommt.
Dazu wollen wir ein paar Begriffe definieren [SS02, Kap. 1.2, S. 6f]:

Glattheit, Faktorbasis

Definition 3.43
Eine Zahl n heiße *b-glatt*, wenn alle Primfaktoren p_i von $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$ kleiner als b sind, dh. wenn gilt:

$$p_i \;\leq\; b \quad \forall i = 1 \ldots r \; .$$

Definition 3.44
Die Menge B aller Primfaktoren unterhalb einer gewissen Schranke b nennen wir eine *Faktorbasis*:

$$B \;:=\; \{p \in \mathbb{P} \mid p \leq b\}$$

Wenn eine Zahl $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$ nur Primfaktoren besitzt, die in einer Faktorbasis B vorkommen, dann sagen wir: n *„zerfällt“* über der Faktorbasis B.

Beispiel 3.45
Die Zahl $n = 2200$ besitzt die Primfaktorzerlegung $2200 = 2^3 \cdot 5^2 \cdot 11$. Dh. 2200 ist 11-glatt, und somit ist 2200 automatisch auch glatt bezüglich jeder Zahl b, die größer als 11 ist.
Weiters zerfällt 2200 über allen Faktorbasen, die zumindest die Primfaktoren 2, 5, und 11 enthalten; 2200 zerfällt aber zB. nicht über $B = \{2, 3, 5, 7\}$.

Konstruktion von Quadraten mod N mithilfe von Faktorbasen
Wesentlich bei der Konstruktion der beiden Quadrate ist, dass sie nur modulo N gleich sein müssen. Um eine Ahnung davon zu bekommen, *wie* man mithilfe von Faktorbasen Quadrate modulo N konstruieren kann, schauen wir uns ein einfaches Bespiel eines Vorläufers des Zahlkörpersiebes an:

Beispiel 3.46 (Dixon's Random-Squares-Methode)
Sei $N = 2501$ jene Zahl, die wir faktorisieren möchten. Dann wählen wir zufällige Zahlen $x_i \in \{1, \dots, n-1\}$, quadrieren diese und betrachten das Ergebnis modulo N; diesen Rest wollen wir mit r_i bezeichnen, zB:

$$x_1 = 142: \qquad 142^2 \;=\; 20164 \;\equiv\; 156 \;=: r_1 \qquad \bmod 2501$$

Es gilt also die Kongruenz: ${x_1}^2 \equiv r_1 \bmod N$. Nun ist zwar die linke Seite $\left({x_1}^2 = 142^2\right)$ der Kongruenz als Quadrat von 142 a priori schon ein Quadrat, aber die rechte Seite ($r_1 = 156 = 2^2 \cdot 3 \cdot 13$) ist leider keines. Wir können aber versuchen mehrere solcher Kongruenzen miteinander zu kombinieren (dh. miteinander zu multiplizieren), sodass sich die rechten Seiten r_i zu einem Quadrat ergänzen. Dazu suchen wir noch weitere Kongruenzen, zB:

$$\begin{array}{lrcrcrcll}
x_2 = 144: & 144^2 & = & 20736 & \equiv & 728 & = & 2^3 \cdot 7 \cdot 13 & \bmod 2501 \\
x_3 = 152: & 152^2 & = & 23104 & \equiv & 595 & = & 5 \cdot 7 \cdot 17 & \bmod 2501 \\
x_4 = 195: & 195^2 & = & 38025 & \equiv & 510 & = & 2 \cdot 3 \cdot 5 \cdot 17 & \bmod 2501
\end{array}$$

Wenn wir nun die rechten Seiten r_i genauer betrachten, ...

			2	3	5	7	13	17
r_1	=	156	2^2	$\cdot 3$			$\cdot 13$	
r_2	=	728	2^3			$\cdot 7$	$\cdot 13$	
r_3	=	595			$\cdot 5$	$\cdot 7$		$\cdot 17$
r_4	=	510	2	$\cdot 3$	$\cdot 5$			$\cdot 17$
$r_1 \cdot r_2 \cdot r_3 \cdot r_4$	=		2^6	$\cdot 3^2$	$\cdot 5^2$	$\cdot 7^2$	$\cdot 13^2$	$\cdot 17^2$

...so erkennen wir, dass im Produkt $\prod_{i=1}^{4} r_i$ die vorkommenden Primfaktoren jeweils in gerader Vielfachheit vorkommen, womit wir nun auch auf der rechten Seite ein Quadrat konstruiert haben:

$$\prod_{i=1}^{4} r_i \;=\; 156 \cdot 728 \cdot 595 \cdot 510 \;=\; 2^6 \cdot 3^2 \cdot 5^2 \cdot 7^2 \cdot 13^2 \cdot 17^2 \;=\; \left(2^3 \cdot 3 \cdot 5 \cdot 7 \cdot 13 \cdot 17\right)^2$$

Dh. mit den Bezeichnungen $x := \prod_{i=1}^{4} x_i$ und $y := 2^3 \cdot 3 \cdot 5 \cdot 7 \cdot 13 \cdot 17$ gilt dann:

$$x^2 \;=\; \left(142 \cdot 144 \cdot 152 \cdot 195\right)^2 \;=\; 142^2 \cdot 144^2 \cdot 152^2 \cdot 195^2 \;\equiv\; 156 \cdot 728 \cdot 595 \cdot 510 \;=\; y^2 \qquad \bmod N$$

Der tatsächliche Wert von $x^2 \equiv y^2 \bmod N$ ist für uns uninteressiert. (Wir wissen, dass mit unserer Konstruktionsweise die beiden Quadrate x^2 und y^2 in derselben Restklasse liegen *müssen.*) Uns interessieren nur die Werte der Restklassen $x \equiv 1386 \bmod N$ und $y \equiv 566 \bmod N$; besser gesagt die Restklassen $x - y \equiv 820 \bmod N$ und $x + y \equiv 1952 \bmod N$, welche ungleich der Restklasse $\overline{0}$ sind und somit beide einen nicht-trivialen Faktor von $N = 2501$ liefern müssen:

$$ggT(820, 2501) = 41 \qquad \text{und} \qquad ggT(1952, 2501) = 61\,.$$

Unterschied zwischen Zahlkörpersieb und Dixon's Random-Squares-Methode
Bei dieser einfachsten Art und Weise, mit Hilfe von Faktorbasen und Glattheit die Quadrate $x^2 \equiv y^2$ zu konstruieren, brauchten wir nur auf der rechten Seite der Kongruenz die einzelnen Zeilen zu einem Quadrat zu kombinieren. Auf der linken Seite stand in jeder einzelnen Zeile ohnehin schon ein Quadrat, weshalb das Produkt von beliebigen „linken Seiten" immer ein Quadrat ist und wir uns nur auf die rechte Seite konzentrieren brauchten. Dabei haben wir auf beiden Seiten immer gleich im Ring $\mathbb{Z}_N$ gearbeitet.

Beim allgemeinen Zahlkörpersieb arbeiten wir im Unterschied dazu auf beiden Seiten nicht (sofort) im Ring $\mathbb{Z}_N$, sondern in anderen Ringen, die dann später (nachdem die Wurzel gezogen wurden) homomorph auf den Ring $\mathbb{Z}_N$ abgebildet werden:
Auf der linken Seite agieren wir vorerst im Ring $\mathcal{R}_1 = \mathbb{Z}$ der ganzen Zahlen und auf der rechten Seite im Ring $\mathcal{R}_2 = \mathbb{Z}[\rho]$ (siehe dazu weiter unten). Dann konstruieren wir auf *beiden* Seiten (nicht nur auf der rechten) - abhängig von einander und simultan - zwei Quadrate $Q_1 \in \mathcal{R}_1$ und $Q_2 \in \mathcal{R}_2$, von denen wir im Vorhinein (durch die Art und Weise der Konstruktion der Quadrate) schon wissen, dass sie mittels der Homomorphismen

$$\begin{array}{rccl} \varphi_1: & \mathcal{R}_1 = \mathbb{Z} & \to & \mathbb{Z}_N \\ & x & \mapsto & x \bmod N \end{array} \qquad \begin{array}{rccl} \varphi_2: & \mathcal{R}_2 = \mathbb{Z}[\rho] & \to & \mathbb{Z}_N \\ & f(\rho) & \mapsto & f(m) \bmod N \end{array} \tag{3.19}$$

in dieselbe Restklasse mod N abgebildet werden (um Gleichung (3.16) zu erfüllen). Sodann ziehen wir in den beiden Ringen jeweils separat die Wurzel und bilden die Wurzeln $W_1 \in \mathcal{R}_1$ und $W_2 \in \mathcal{R}_2$ ebenfalls mit den Homomorphismen φ_1 bzw. φ_2 nach $\mathbb{Z}_N$ ab, in der Hoffnung, dass für die Bilder der Wurzeln

$$\varphi_1(W_1) \not\equiv \pm\varphi_2(W_2) \mod N$$

gilt, womit auch die Gleichung (3.17) erfüllt wäre und wir so eine Faktorisierung von N erreicht hätten.

Mathematische Grundlagen des Zahlkörpersiebes
Wir werden hier die Theorie, die hinter dem Zahlkörpersieb steckt, nur in jenem Maß anführen, als es für eine Implementierung erforderlich ist. Für ausführlichere Arbeiten dazu siehe [Lj93] (eine gute Zusammenfassung der grundlegendsten Arbeiten zum NFS) sowie die Diplomarbeiten bzw. Dissertationen zum GNFS: [Zay91], [Zay95] und [Bri98].

Definition 3.47 (Zahlkörper)
Sei $\mathbb{K} \subseteq \mathbb{C}$ eine mindestens zwei-elementige Teilmenge von $\mathbb{C}$. $\mathbb{K}$ heißt ein *Zahlkörper* (*number field*), wenn $\forall \alpha, \beta, \gamma \in \mathbb{K}$ mit $\gamma \neq 0$ gilt:

1. $(\alpha + \beta) \in \mathbb{K}$
2. $(\alpha - \beta) \in \mathbb{K}$
3. $(\alpha \cdot \beta) \in \mathbb{K}$
4. $\left(\frac{\alpha}{\gamma}\right) \in \mathbb{K}$.

Definition 3.48 (algebraische Zahl, Grad, Konjugierte)
Sei $\rho \in \mathbb{C}$ eine Nullstelle (Wurzel) des Polynoms

$$f(x) = c_d\, x^d + c_{d-1}\, x^{d-1} + \ldots + c_1\, x + c_0 \qquad \text{mit } c_i \in \mathbb{Q} \quad \forall i = 0, \ldots, d\,,$$

dann nennt man ρ eine *algebraische Zahl*. Ist $f(x)$ zusätzlich irreduzibel über $\mathbb{Q}$ und $c_d \neq 0$, dann nennt man d den *Grad* der algebraischen Zahl ρ, $f(x)$ das *Minimalpolynom* von ρ und die restlichen (komplexen) Nullstellen $\rho_2, \rho_3, \ldots, \rho_d$ heißen die *Konjugierten* von $\rho = \rho_1$.

Lemma 3.49 (Zahlkörper $\mathbb{Q}(\rho)$)
Ist ρ eine algebraische Zahl vom Grad d, dann ist

$$\mathbb{Q}(\rho) \;:=\; \left\{ c_{d-1}\rho^{d-1} + c_{d-2}\rho^{d-2} + \ldots + c_1\rho + c_0 \;\middle|\; c_i \in \mathbb{Q} \quad \forall i = 1, ..., d-1 \right\}$$

ein Zahlkörper und kann als ein d-dimensionaler Vektorraum über $\mathbb{Q}$ gesehen werden, dh.

$$\mathbb{Q}(\rho) \;\cong\; \mathbb{Q}[x] \Big/ \big(f(x)\big) \;\cong\; \mathbb{Q}^d \;.$$

Definition 3.50 (ganze algebraische Zahl)
Sei $f(x) = c_d\, x^d + c_{d-1}\, x^{d-1} + \ldots + c_1\, x + c_0$ mit $c_i \in \mathbb{Z} \;\; \forall i = 0, \ldots, d$ irreduzibel über $\mathbb{Q}$ und sei ρ eine Nullstelle von $f(x)$. Dann heißt ρ eine *ganze algebraische Zahl* (vom Grad d). Die Menge aller ganzen algebraischen Zahlen bezeichnen wir mit $\mathcal{O}_\rho$.

Definition 3.51
Sei $\rho \in \mathcal{O}_\rho$ eine ganze algebraische Zahl vom Grad d. Dann bezeichnen wir mit $\mathbb{Z}[\rho] \subseteq \mathbb{C}$ die Menge

$$\mathbb{Z}[\rho] \;:=\; \left\{ c_{d-1}\rho^{d-1} + c_{d-2}\rho^{d-2} + \ldots + c_1\rho + c_0 \;\middle|\; c_i \in \mathbb{Z} \quad \forall i = 1, ..., d-1 \right\} \;\subseteq\; \mathcal{O}_\rho \;\subseteq\; \mathbb{Q}(\rho)\,.$$

Das folgende Lemma ist sehr hilfreich, um das Zahlkörpersieb in der Praxis zu implementieren, denn es ersetzt Rechnen in $\mathbb{C}$ durch Rechnen in $\mathbb{Z}^d$, was leichter zu programmieren ist [CP00, Kap. 6.2.1, S. 243]:

Lemma 3.52
Sei $\rho \in \mathcal{O}_\rho$ eine ganze algebraische Zahl vom Grad d und $f(x)$ das zugehörige Minimalpolynom. Dann ist $\mathbb{Z}[\rho]$ isomorph zu

$$\mathbb{Z}[\rho] \;\cong\; \mathbb{Z}[x] \Big/ \big(f(x)\big) \;.$$

Definition 3.53 (Norm)
Sei α eine algebraische Zahl. Dann ist die *Norm* $N(\alpha)$ definiert als das Produkt seiner Konjugierten:

$$N(\alpha) \;:=\; \prod_{i=1}^{d} \alpha_i \;.$$

Lemma 3.54 (Eigenschaften der Norm)
Seien α, β algebraische Zahlen. Dann gilt:

1. $N(\alpha) \in \mathbb{Q}$
2. $N(\alpha \cdot \beta) = N(\alpha) \cdot N(\beta) \qquad \forall \alpha, \beta \qquad$ (Multiplikativität)
3. Gilt zusätzlich $\alpha \in \mathcal{O}_\rho$, dh. α ist eine ganze algebraische Zahl, dann gilt sogar: $N(\alpha) \in \mathbb{Z}$.

Konvention
Von nun an sei ρ eine ganze algebraische Zahl und $f(x) \in \mathbb{Z}[x]$ sei das zugehörige Minimalpolynom vom Grad d.

Konstruktion von Quadraten beim Zahlkörpersieb
Mit diesem mathematischen Hintergrund wollen wir uns die Konstruktion der Quadrate beim Zahlkörpersieb nochmals etwas genauer anschauen:

Angenommen wir hätten eine Menge S von Paaren (a, b) gefunden, sodass

1. $\prod_{(a,b)\in S} (a + bm) = \widetilde{W_1}^2 \qquad$ für ein $\widetilde{W_1} \in \mathcal{R}_1 = \mathbb{Z} \qquad$ („linke Seite“) $\qquad$ und

2. $\prod\limits_{(a,b)\in S}(a+b\rho) = \widetilde{W_2}^2$ für ein $\widetilde{W_2} \in \mathcal{R}_2 = \mathbb{Z}[\rho]$ („rechte Seite“)

gilt, dann hätten wir mit $\widetilde{x} := \varphi_1(\widetilde{W_1})$ und $\widetilde{y} := \varphi_2(\widetilde{W_2})$ wegen

$$\begin{aligned}
\widetilde{y}^2 &= \left(\varphi_2\left(\widetilde{W_2}\right)\right)^2 \equiv \varphi_2\left(\widetilde{W_2}^2\right) \equiv \varphi_2\left(\prod_{(a,b)\in S}(a+b\rho)\right) \equiv && \pmod N \\
&\equiv \prod_{(a,b)\in S}\varphi_2(a+b\rho) && \pmod N \\
&\equiv \prod_{(a,b)\in S}(a+bm) && \pmod N \\
&\equiv \prod_{(a,b)\in S}\varphi_1(a+bm) && \pmod N \\
&\equiv \varphi_1\left(\prod_{(a,b)\in S}(a+bm)\right) \equiv \varphi_1\left(\widetilde{W_1}^2\right) \equiv \left(\varphi_1\left(\widetilde{W_1}\right)\right)^2 = \widetilde{x}^2 && \pmod N
\end{aligned}$$

zwei Zahlen $\widetilde{x}, \widetilde{y}$ mod N gefunden, deren Quadrate in derselben Restklasse lägen, dh. die (3.16) erfüllten. Das Problem dabei ist, dass wir nicht davon ausgehen können, dass $\widetilde{W_2} \in \mathbb{Z}[\rho]$ liegt. Bis jetzt können wir mit Sicherheit nur sagen, dass $\widetilde{W_2} \in \mathbb{Q}(\rho)$ liegt. Da wir letztendlich die Wurzel $\widetilde{W_2}$ mittels des Homomorphismus' φ_2 nach $\mathbb{Z}_N$ abbilden möchten, müssen wir aber sicherstellen, dass $\widetilde{W_2}$ im Definitionsbereich von φ_2 liegt, dh. dass $\widetilde{W_2} \in \mathbb{Z}[\rho]$ ist. Dies können wir mit folgendem Lemma [Lj93, S. 60f] bewerkstelligen:

Lemma 3.55
Wenn $\prod\limits_{(a,b)\in S}(a+b\rho) = \alpha^2$ für ein $\alpha \in \mathbb{Q}(\rho)$, dann ist $f'(\rho)\cdot\alpha \in \mathbb{Z}[\rho]$.

Aufgrund von Lemma 3.55 können wir das Produkt $\prod\limits_{(a,b)\in S}(a+b\rho)$ mit $f'(\rho)^2$ multiplizieren und wissen dann, dass W_2 mit

$$W_2{}^2 = Q_2 = f'(\rho)^2 \cdot \prod_{(a,b)\in S}(a+b\rho) \quad \in \mathbb{Z}[\rho]$$

selbst auch im Definitionsbereich $\mathbb{Z}[\rho]$ von φ_2 liegen muss.
Analog müssen wir das Produkt $\prod\limits_{(a,b)\in S}(a+bm)$ mit $f'(m)^2$ multiplizieren um weiterhin zu gewährleisten, dass die Bilder von

$$Q_1 := f'(m)^2 \cdot \prod_{(a,b)\in S}(a+bm) \qquad \text{und} \qquad Q_2 := f'(\rho)^2 \cdot \prod_{(a,b)\in S}(a+b\rho)$$

unter φ_1 bzw. φ_2 dieselbe Restklasse mod N ergeben, denn es gilt dann:

$$\begin{aligned}
\varphi_2(Q_2) &= \varphi_2\left(f'(\rho)^2 \cdot \prod_{(a,b)\in S}(a+b\rho)\right) \equiv \varphi_2\left(f'(\rho)^2\right)\cdot\varphi_2\left(\prod_{(a,b)\in S}(a+b\rho)\right) \equiv && \pmod N \\
&\equiv \left(\varphi_2\left(f'(\rho)\right)\right)^2 \cdot \prod_{(a,b)\in S}\varphi_2(a+b\rho) \equiv f'(m)^2 \cdot \prod_{(a,b)\in S}(a+bm) \equiv && \pmod N \\
&\equiv \left(\varphi_1\left(f'(m)\right)\right)^2 \cdot \prod_{(a,b)\in S}\varphi_1(a+bm) \equiv \varphi_1\left(f'(m)^2\right)\cdot\varphi_1\left(\prod_{(a,b)\in S}(a+bm)\right) \equiv && \pmod N \\
&\equiv \varphi_1\left(f'(m)^2 \cdot \prod_{(a,b)\in S}(a+bm)\right) = \varphi_1(Q_1) && \pmod N\ .
\end{aligned}$$

Deshalb gilt für $x := \varphi_1(W_1)$ und $y := \varphi_2(W_2)$ die Kongruenz

$$x^2 = \left(\varphi_1(W_1)\right)^2 \equiv \varphi_1\left({W_1}^2\right) = \varphi_1(Q_1) \equiv \varphi_2(Q_2) = \varphi_2\left({W_2}^2\right) \equiv \left(\varphi_2(W_2)\right)^2 = y^2 \pmod{N}$$

womit (3.16) weiterhin erfüllt ist und wir somit Kanditaten $x, y \in \mathbb{Z}_N$ für eine Faktorisierung erhalten.

Polynombestimmung
Um im Ring $\mathcal{R}_2 = \mathbb{Z}[\rho]$ arbeiten zu können, benötigen wir ein über $\mathbb{Q}$ irreduzibles Polynom $f(x) \in \mathbb{Z}[x]$ vom Grad d und $\rho \in \mathbb{C}$ sei eine Nullstelle von $f(x)$. Ein Teil der Berechnungen des Zahlkörpersiebes findet im Unterring $\mathbb{Z}[\rho]$ des Zahlkörpers $\mathbb{Q}(\rho)$ statt. Für die konkreten Berechnungen, die tatsächlich durchgeführt werden müssen, ist aber die Kenntnis des Wertes der Nullstelle $\rho \in \mathbb{C}$ nicht nötig. Aus Sicht der Praxis können wir ρ als ein formales Symbol ansehen, von dem wir wisssen, dass $f(\rho) = 0$ ist.

Was hat ein irreduzibles Polynom vom Grad d mit der zu faktorisierenden Zahl N zu tun? Um faktorisieren zu können, brauchen wir keine der Nullstellen von $f(x)$ zu kennen; es reicht, einen Wert $m \in \mathbb{Z}$ zu kennen, sodass m eingesetzt in $f(x)$ (das ganzzahlige Koeffizienten besitzt) ein Vielfaches von N ergibt, dh.

$$f(m) \equiv 0 \mod N \,. \tag{3.20}$$

Andererseits gilt definitionsgemäß: $f(\rho) = 0$ weshalb trivialerweise auch

$$f(\rho) \equiv 0 \mod N$$

gilt, womit der Zusammenhang zwischen dem Polynom $f(x)$ und der zu faktorisierenden Zahl N sichtbar wird.

Um für eine konkrete Zahl $N \in \mathbb{Z}$ ein passendes Polynom $f(x)$ zu bestimmen, kann man die *base-m-Methode* anwenden: Dazu fixiert man zuallererst den Grad $d \in \mathbb{N}$ des Polynoms $f(x)$. Ein heuristisch begründeter Wert [CP00, S. 251, 257] für d ist

$$d := \left\lfloor \left(\frac{3 \ln N}{\ln \ln N} \right)^{\frac{1}{3}} \right\rfloor .$$

Dadurch ergeben sich die in Tabelle 3.1 angeführten Werte für d.

N	Polynomgrad d	RSA-Schlüssellänge
$\leq 10^{13}$	2	
10^{14} bis 10^{42}	3	64, 128 Bit
10^{43} bis 10^{98}	4	256 Bit
10^{99} bis 10^{190}	5	512 Bit
10^{191} bis 10^{329}	6	1024 Bit
10^{330} bis 10^{526}	7	1536 Bit
10^{527} bis 10^{792}	8	2048 Bit
10^{793} bis 10^{1139}	9	3072 Bit
10^{1140} bis 10^{1579}	10	4096 Bit
10^{1580} bis 10^{2125}	11	6144 Bit
10^{2126} bis 10^{2788}	12	8192 Bit

Tabelle 3.1: Zahlkörpersieb - Polynomgrade

N	Polynomgrad d
$\leq 10^{75}$	3
10^{76} bis 10^{200}	5

Tabelle 3.2: Zahlkörpersieb - nur ungerade Polynomgrade

Für eine Verbesserungsmöglichkeit bei der späteren Berechnung der algebraischen Quadratwurzel (vgl. S. 68) ist ein ungerader Polynomgrad d erforderlich. Jörg Zayer [Zay95, Kap. 4.1, S. 93] verwendet dafür die in Tabelle 3.2 angeführten Werte für d.

Ist d einmal festgelegt, so berechnet man sich als Nächstes den Wert

$$m \;:=\; \left\lfloor N^{\frac{1}{d}} \right\rfloor$$

und stellt dann N als Zahl in der Basis m dar:

$$N \;=\; a_d \cdot m^d + a_{d-1} \cdot m^{d-1} + \ldots + a_1 \cdot m + a_0 \qquad \text{mit } a_i \in \{0, \ldots, m-1\} \text{ für } i = 0, \ldots, d$$

Unser gesuchtes Polynom $f(x)$ definiert sich dann durch

$$f(x) \;:=\; a_d \cdot x^d + a_{d-1} \cdot x^{d-1} + \ldots + a_1 \cdot x + a_0 \;.$$

Man beachte, dass bei dieser Art und Weise, $f(x)$ und m zu bestimmen,

$$f(m) \;=\; N$$

gilt, wodurch insbesondere die Kongruenz (3.20) erfüllt ist. Unter der zusätzlichen Voraussetzung, dass auch $N > 2^{d^2}$ und $d > 1$ erfüllt ist (was bei RSA-Moduli in der Praxis erfüllt sein wird), folgt [Lj93, S. 54], dass der Leitkoeffizient $a_d = 1$ (und dass $a_{d-1} \leq d$) sein muss, wodurch wir ein normiertes Polynom erhalten:

$$f(x) \;=\; x^d + a_{d-1} \cdot x^{d-1} + \ldots + a_1 \cdot x + a_0 \;.$$

Eine der Anforderungen an das Polynom $f(x)$ ist ja, dass $f(x)$ irreduzibel über $\mathbb{Q}$ sein muss. Diese Eigenschaft müssen bzw. möchten wir auch überprüfen:
Wenn $f(x)$ irreduzibel über $\mathbb{Q}$ ist, dann müssen wir mit dem Zahlkörpersieb (in seiner vollen Ausformung) wie geplant fortfahren. Dies wird auch der Regelfall sein.
Was aber ist, sollte $f(x)$ unerwarteterweise tatsächlich reduzibel sein?
In diesem Fall lässt sich $f(x)$ schreiben als

$$f(x) \;=\; g(x) \cdot h(x)$$

für zwei Polynome $g(x), h(x)$. Wegen $f(m) = N$ gilt dann aber auch

$$N \;=\; g(m) \cdot h(m)$$

weshalb wir damit ([Lj93, S. 54], [CP00, Kap. 6.2.1, S. 244]) eine nicht-triviale Faktorisierung von N erhalten würden. Somit könnten wir uns alle weiteren Schritte ersparen.

Durch die Anwendung von Lemma 3.55 müssen wir auf der rationalen Seite das Produkt $\prod_{(a,b)\in S} (a+bm)$ mit $f'(m)^2$ multiplizieren. Um dabei die theoretische Erfolgswahrscheinlichkeit des allgemeinen Zahlkörpersiebes durch diese Multiplikation unverändert zu lassen [Zay95, Bem. 2.27, S. 37], ist es notwendig, dass $ggT(f'(m), N) = 1$ ist, was wir leicht überprüfen können. Sollte unerwarteterweise doch $ggT(f'(m), N) > 1$ und $f(x)$ und m mit der base-m-Methode erstellt worden sein, so muss [Lj93, S. 61] wegen $1 < f'(m) < N$ der $ggT(f'(m), N)$ eine nicht-triviale Faktorisierung liefern.

Verwenden von Faktorbasen beim Zahlkörpersieb
Im Beispiel 3.46 haben wir gesehen, wie wir durch Kombination von mehreren Kongruenzen (auf der rechten Seite) ein Quadrat konstruieren konnten. Dieses Prinzip werden wir beim Zahlkörpersieb auf *beiden* Seiten anwenden. Dabei verwenden wir für die beiden Seiten der Kongruenz unterschiedliche Faktorbasen. Um genauer zwischen den beiden Seiten der Kongruenz unterscheiden zu können, sprechen wir, wenn wir den Ring $\mathcal{R}_1 = \mathbb{Z}$ meinen, von der *rationalen Seite* (linke Seite) der Kongruenz, bzw. ist mit der *algebraischen Seite* (rechte Seite) der Kongruenz der Ring $\mathcal{R}_2 = \mathbb{Z}[\rho]$ gemeint.

Rationale Faktorbasis - RFB
Als Erstes müssen wir für die *rationale Faktorbasis* RFB eine obere Schranke B_{rat} in Abhängigkeit von N wählen. Um die (heuristische) asymptotische Laufzeit des Zahlkörpersiebes minimal zu halten [Lj93, S. 80], sollte man B_{rat} in der Größenordnung

$$B_{\text{rat}} \sim e^{\left((\frac{8}{9})^{\frac{1}{3}} \cdot (\ln n)^{\frac{1}{3}} \cdot (\ln \ln n)^{\frac{2}{3}}\right)}$$

wählen.[15] Für den Ring $\mathcal{R}_1 = Z$ (linke Seite) dient uns als Faktorbasis die Menge

$$\text{RFB} := \left\{ p \in \mathbb{P} \;\middle|\; p \leq B_{\text{rat}} \right\}.$$

Da man später im Verlauf des Zahlkörpersiebes den Wert m mod p für alle $p \in$ RFB immer wieder benötigen wird[16], ist es aus praktischer Sicht - dh. um Rechenzeit zu sparen - ratsam, die rationale Faktorbasis alternativ als die Menge der Paare

$$\text{RFB} := \left\{ (p, m \bmod p) \;\middle|\; p \in \mathbb{P},\ p \leq B_{\text{rat}} \right\}$$

zu definieren, wovon ich in weiterer Folge auch ausgehen möchte[17].

Der Zusammenhang zwischen der rationalen Faktorbasis und dem zu konstruierenden Quadrat in $\mathcal{R}_1 = \mathbb{Z}$ spiegelt sich in der folgenden bekannten Tatsache wider:

Lemma 3.56
Für eine Zahl $z \in \mathbb{Z}$ gilt: Ist z ein Quadrat in $\mathbb{Z}$, dann muss in der eindeutigen Primfaktorzerlegung von $|z| = {p_1}^{e_1} {p_2}^{e_2} \cdots {p_r}^{e_r}$ jeder Primfaktor $p_i \in \mathbb{P}$ in gerader Vielfachheit e_i auftreten. Dh. für unser Quadrat $\prod_{(a,b) \in S} a + bm$, das wir in $\mathbb{Z}$ konstruieren wollen, ist es eine notwendige Bedingung, dass die Vielfachheit der Primfaktoren 0 mod 2 ist.
In $\mathbb{Z}$ gilt aber auch die Umkehrung: Sei z das Produkt $\prod_{i=1}^{r} {p_i}^{e_i}$ von r Primfaktoren $p_i \in \mathbb{P}$, wobei jeder Primfaktor p_i in gerader Vielfachheit vorkommt. Dann gibt es eine Zahl $w \in \mathbb{Z}$ mit $w^2 = z$.

Dies garantiert uns, dass die von uns konstruierte Zahl tatsächlich ein Quadrat ist. (Auf der rationalen Seite, dh. in $\mathbb{Z}$ ist diese Äquivalenz für uns eine Selbstverständlichkeit; auf der algebraischen Seite ist die Lage jedoch anders.)

[15] Wie der Leser sicherlich schon bemerkt hat, ist an dieser Stelle mit e^x die Exponentialfunktion gemeint und nicht der öffentliche RSA-Exponent, da diese Faktorisierungsmethode unabhängig vom RSA-Schlüsselpaar funktioniert.

[16] Für jeden b-Wert berechnet man beim rationalen Sieb für jede Primzahl p aus der rationalen Faktorbasis einen Startpunkt, dh. den kleinsten Wert $a \in [-C, C]$ mit $p \mid a + b\,m$. Dies geschieht mit der Formel:

$$\mathit{startpunkt} := -C + \big((C - b\,m) \bmod p\big)$$

[17] Derselben Idee folgend könnte man auch den Wert C mod p für alle $p \in$ RFB einmalig vorausberechnen und dann die Trippel $(p, m \bmod p, C \bmod p)$ als die Elemente der RFB definieren.

Algebraische Faktorbasis - AFB
In dieser Arbeit werden wir uns, was den mathematischen Hintergrund anbelangt - insbesondere im Hinblick auf das Thema „algebraische Faktorbasis - algebraisches Sieben“ - vorwiegend auf den für eine Implementierung notwendigen Aspekt konzentrieren. Für diesbezüglich weiterführendere Literatur sei verwiesen auf [Lj93, S. 50-94], [Bri98] und [Zay91].

Die Elemente in der algebraischen Faktorbasis sind all jene Paare (p, r) mit der Eigenschaft, dass p eine Primzahl unterhalb einer gewissen oberen Schranke B_{alg} ist und dass r eine Nullstelle von $f(x)$ modulo p ist. Die Nullstellen von $f(x)$ modulo einem festen p fassen wir zusammen in der Menge

$$R(p) \;:=\; \left\{\; r \;\middle|\; 0 \le r \le p-1,\; f(r) \equiv 0 \bmod p \right\}.$$

Dh. die *algebraische Faktorbasis* AFB lässt sich somit durch

$$\text{AFB} \;:=\; \left\{(p,r) \;\middle|\; p \in \mathbb{P},\; p \le B_{\text{alg}},\; r \in R(p)\right\}$$

definieren. B_{alg} sollte dabei so gewählt werden, dass die Anzahl der Elemente in AFB ungefähr 2 bis 3 mal so groß als jene in RFB ist [Jen05, Kap. 6.2.2, S. 62]. Dabei ist zu beachten, dass ein Polynom modulo eine Primzahl bekannterweise nur höchstens soviele Nullstellen haben kann, als der Grad des Polynoms angibt. Daraus folgt die Tatsache, dass

$$0 \;\le\; |R(p)| \;\le\; d$$

ist. Für manche p kann $R(p) = \varnothing$ sein, dh. diese p kommen in AFB dann gar nicht vor. Für manche p gibt es genau *ein* r; diese p kommen dann genau einmal in AFB vor. Und für manche p gibt es *mehrere* $r \in R(p)$. Somit kann in AFB ein p öfter als einmal, dafür aber mit unterschiedlichen r vorkommen. Durchschnittlich können wir aber erwarten, dass $R(p)$ ungefähr 1 Element besitzt [Lj93, S. 57]. (Für meine Implementierung in Maple habe ich $B_{\text{alg}} = 3 \cdot B_{\text{rat}}$ gewählt.)

Definition 3.57 (Glattheit einer ganzen algebraischen Zahl)
Eine ganze algebraische Zahl $\alpha \in \mathbb{Z}[\rho]$ heißt *b-glatt*, wenn ihre Norm $N(\alpha) \in \mathbb{Z}$ b-glatt ist.[18]

Definition 3.58 (homogenes Polynom $F(x,y)$)
Sei $f(x) = x^d + a_{d-1} \cdot x^{d-1} + a_{d-2} \cdot x^{d-2} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$ gegeben. Dann definieren wir das homogene Polynom $F(x,y)$ durch

$$F(x,y) \;:=\; x^d + a_{d-1} \cdot x^{d-1}y + a_{d-2} \cdot x^{d-2}y^2 + \ldots + a_2 \cdot x^2 y^{d-2} + a_1 \cdot xy^{d-1} + a_0 \cdot y^d .$$

Satz 3.59
Sei $a + b\rho \in \mathbb{Z}[\rho]$ und sei $F(x,y)$ wie in Definition 3.58. Dann gilt für die Norm von $a + b\rho$:

$$N(a + b\rho) \;=\; F(a, -b).$$

Beweis
Laut Definition 3.53 ist die Norm einer algebraischen Zahl definiert als das Produkt mit all ihren Konjugierten, dh. es gilt

$$\begin{aligned} N(a + b\rho) &= (a + b\rho_1)(a + b\rho_2)\cdots(a + b\rho_d) \;= \\ &= (-b)^d \left(\frac{a}{-b} - \rho_1\right)\left(\frac{a}{-b} - \rho_2\right)\cdots\left(\frac{a}{-b} - \rho_d\right) \;= \\ &= (-b)^d f\left(\frac{a}{-b}\right). \end{aligned} \tag{3.21}$$

[18] Dass für ein $\alpha \in \mathbb{Z}[\rho]$ seine Norm $N(\alpha) \in \mathbb{Z}$ ist, wissen wir schon aus Lemma 3.54 .

Andererseits kennen wir auch einen Zusammenhang zwischen $f(x)$ und $F(x,y)$:

$$\begin{aligned} y^d \cdot f\left(\frac{x}{y}\right) &= y^d \cdot \left[\left(\frac{x}{y}\right)^d + a_{d-1}\left(\frac{x}{y}\right)^{d-1} + a_{d-2}\left(\frac{x}{y}\right)^{d-2} + \ldots + a_2\left(\frac{x}{y}\right)^2 + a_1\left(\frac{x}{y}\right) + a_0\right] \\ &= x^d + a_{d-1}\cdot x^{d-1}y + a_{d-2}\cdot x^{d-2}y^2 + \ldots + a_2\cdot x^2 y^{d-2} + a_1\cdot xy^{d-1} + a_0\cdot y^d \\ &= F(x,y) \end{aligned}$$

weshalb wir daraus

$$(-y)^d f\left(\frac{x}{-y}\right) = F(x,-y) \tag{3.22}$$

schließen können. Aus den Gleichungen (3.21) und (3.22) erhalten wir mit $x = a$ und $y = b$ schließlich die gewünschte Gleichung:

$$N(a+b\rho) = F(a,-b)\,.$$

□

Dh. eine ganze algebraische Zahl $\alpha = a + b\rho$ ist B_{alg}-glatt, wenn $F(a,-b)$ B_{alg}-glatt ist. In diesem Fall sagen wir, α *zerfällt* über der algebraischen Faktorbasis AFB.

Wie wir aus Lemma 3.54 schon wissen ist die Norm einer algebraischen Zahl multiplikativ. Daraus erhalten wir folgendes Korollar:

Korollar 3.60
Sei $\alpha = \beta^2$ ein Quadrat mit $\beta \in \mathbb{Z}[\rho]$. Dann ist dessen Norm

$$N(\alpha) = N\left(\beta^2\right) = \left(N(\beta)\right)^2 \quad \in \mathbb{Z}$$

ein Quadrat in $\mathbb{Z}$.

Damit haben wir eine weitere notwendige Bedingung für die von uns zu konstruierenden Quadrate gefunden:

Bemerkung 3.61
Wir wissen, dass wir auf der Suche nach dem algebraischen Quadrat $\prod_{(a,b)\in S} a+b\rho$ nur Kandidaten in Betracht ziehen brauchen, deren Norm

$$N\left(\prod_{(a,b)\in S} a+b\rho\right) = \prod_{(a,b)\in S} N(a+b\rho) = \prod_{(a,b)\in S} F(a,-b)$$

ein Quadrat in $\mathbb{Z}$ ist.

Dh. wir suchen Relationen (a,b), sodass $F(a,b)$ jeweils über der algebraischen Faktorbasis zerfällt. Dann wählen wir davon eine Teilmenge S aus, sodass $\prod_{(a,b)\in S} F(a,-b)$ ein Quadrat in $\mathbb{Z}$ ergibt.
Leider ist die Bedingung, dass die Norm einer algebraische Zahl α ein Quadrat in $\mathbb{Z}$ ist, bei weitem noch nicht hinreichend dafür, dass α selbst ein Quadrat in $\mathbb{Z}[\rho]$ ist. Die Vorgangsweise beim Zahlkörpersieb ist jene, diese Wahrscheinlichkeit dafür aber so weit zu erhöhen, dass sie für die Praxis ausreichend ist.

Nachdem definitionsgemäß eine algebraische Zahl über einer algebraischen Faktorbasis genau dann zerfällt, wenn ihre Norm darüber zerfällt, ist es wichtig zu wissen, wann eine Primzahl p die Norm $N(a+b\rho)$ teilt, worüber uns der folgende Satz Auskunft gibt [Lj93, Kap. 5, S. 57], [CP00, Kap. 6.2.2, S. 247]:

Satz 3.62
Seien a, b teilerfremd und $p \nmid b$. Dann gilt

$$p \,\big|\, N(a+b\alpha) \quad \Leftrightarrow \quad \exists r \in R(p):\ a \equiv -br \mod p\,.$$

Für $p \,|\, b$ kann es keinen Wert a geben, sodass sowohl $N(a+b\alpha) \equiv 0 \mod p$ als auch $ggT(a,b) = 1$ gilt.

Aus diesem Satz können wir zwei Informationen gewinnen:

1. Prinzipiell brauchen wir uns nur auf jene Primzahlen p zu konzentrieren, für die $R(p) \neq \varnothing$ ist. Andere Werte für p können ohnehin gar nicht in Frage kommen.

2. Für sog. *line sieving* (vgl. S. 58) können wir für ein fixiertes b daraus auch gleich den Startpunkt berechnen, von dem aus wir später im Abstand p fortschreiten werden, um die geeigneten a-Werte zu finden sodass *alle* Werte $N(a + b\rho)$ jeweils durch p teilbar sind.

Quadratische Charaktere Basis - QCB
Die Einführung von sog. quadratischen Charakteren hat einen doppelten Grund [Lj93, Kap. 6 u. 7], [CP00, Kap. 6.2.4]: Zum einen werden dadurch gewisse mathematische Probleme, die sich bei der dem Zahlkörpersieb zugrundeliegenden Theorie ergeben, überwunden. Zum anderen wird dadurch aber auch die Wahrscheinlichkeit für eine erfolgreiche Faktorisierung von N erhöht.

Ein sehr gutes, konkretes Beispiel, um die Idee hinter den quadratischen Charakteren zu beschreiben, bringen Crandall und Pomerance in [CP00, S. 253] der allgemeinen Idee aus [Lj93, S. 67f] folgend:

Beispiel 3.63
Stellen wir uns vor, wir könnten in $\mathbb{Z}$ zwar schnell die Primfaktoren einer beliebigen Zahl z bestimmen, aber wir hätten keine Möglichkeit, die Vorzeichen zu erkennen. Dh. wir können die Zahlen 4 und -4 nicht unterscheiden, da sie (bis auf das Vorzeichen) dieselbe Primfaktorenzerlegung besitzen. Nehmen wir nun an, dass uns jemand eine Zahl $z \in \mathbb{Z}$ vorlegt, in der alle Primzahlen in gerader Vielfachheit auftreten. Die Frage, die wir klären wollen ist, ob z tatsächlich ein Quadrat in $\mathbb{Z}$ ist oder nicht. (noch einmal: Wichtig dabei ist, dass wir das Vorzeichen von z nicht bestimmen können.)
Angenommen z ist tatsächlich ein Quadrat, dann existiert ein $w \in \mathbb{Z}$ mit $w^2 = z$. Somit ist die Restklasse $\overline{z}$ modulo *jeder* beliebigen Primzahl p ein quadratischer Rest, da aus $w^2 = z$ natürlich folgt: $\overline{w}^2 \equiv \overline{z} \mod p \quad \forall\ p \in \mathbb{P}$. In diesem Fall muss dann das Legendre-Symbol $\left(\frac{z}{p}\right) = 1$ sein, und zwar für *alle* $p \in \mathbb{P}$.
Zum Beispiel ist $\left(\frac{4}{13}\right) = 1$. Genauso erhalten wir für $p = 17$: $\left(\frac{4}{17}\right) = 1$. Nun gilt aber auch für $z = -4$, dass $\left(\frac{-4}{13}\right) = 1 = \left(\frac{-4}{17}\right)$ ist.
Wenn wir aber ein p finden würden, sodass $\left(\frac{z}{p}\right) = -1$ ist, dann könnten wir daraus schließen, dass z kein Quadrat sein kann: Mit der Wahl $p = 19$ erhalten wir $\left(\frac{-4}{19}\right) = -1$. Somit wissen wir, dass -4 kein Quadrat sein kann.
Dh. je mehr Primzahlen p wir für diesen Test mit einer gegebenen Zahl z heranziehen, desto größer wird die Wahrscheinlichkeit, dass wir z, sofern es kein Quadrat sein sollte, auch als solches erkennen [Lj93, S. 68, Lemma 8.2].

Die Elemente in der *quadratischen-Charaktere-Basis* QCB sind ganz ähnlich jenen in der AFB definiert:

$$\text{QCB} \;:=\; \left\{(q,s) \;\middle|\; q \in \mathbb{P},\ \exists s \in R(q):\ f'(s) \not\equiv 0 \mod q\right\},$$

bloß mit der zusätzlichen Bedingung, dass $f'(s) \not\equiv 0 \mod q$ gelten muss und dass q größer als die größte in RFB und AFB vorkommende Primzahl ist.

Was die Anzahl der Elemente in QCB betrifft, so gibt es unterschiedliche Angaben dazu, wobei dieser Wert (genauso wie auch die Größe von RFB und AFB) prinzipiell nach eigenem Belieben bzw. technischen Einschränkungen verändert werden kann, sofern man sich der Auswirkungen bewusst ist. In einer der grundlegendsten Arbeiten zum Zahlkörpersieb [Lj93, S. 69] wird die Anzahl der Elemente in QCB mit $\left\lfloor \frac{3 \ln N}{\ln 2} \right\rfloor$ angegeben. [CP00, Alg. 6.2.5, S. 257] verwendet die ebenfalls von N abhängige Anzahl $\lfloor 3 \ln N \rfloor$ an quadratischen Charakteren.
Hingegen sollte für die Praxis eine Größe von etwa 50-100 ausreichend sein, da selbst für Rekordfaktorisierungen nicht mehr als 100 quadratische Charaktere verwendet werden [Jen05, Kap. 5.2.4.1, S. 54], [Lj93, S. 108, Kap. 5].

Sieben
Um unsere beiden Quadrate $Q_1 \in \mathbb{Z}$ und $Q_2 \in \mathbb{Z}[\rho]$ zu konstruieren, suchen wir Relationen $(a, b) \in \mathbb{Z} \times \mathbb{N}$, sodass $a + bm$ über RFB und $a + b\rho$ über AFB zerfällt. Die ursprüngliche und einfachste Art und Weise, diese Paare (a, b) zu suchen, ist sog. *line sieving*. Das bedeutet, dass man vorerst den Wert b festhält und a im *Siebintervall* $[-C, C]$ laufen lässt, wobei C im Vorhinein genügend groß festgelegt wurde. Von diesem Bereich an (a, b)-Werten filtern wir jene Paare heraus, die gewisse Eigenschaften (siehe 3.64) erfüllen. Anschließend erhöht man den Wert b um eins und wiederholt das Ganze für den Wert $b+1$. Dies führt man solange fort, bis man genügend geeignete Paare (a, b) gefunden hat.

Für eine optimale (asymptotische) Laufzeit sollte auch C in derselben Größenordnung wie die beiden Schranken für RFB bzw. AFB liegen, nämlich:

$$C \sim e^{\left(\left(\frac{8}{9}\right)^{\frac{1}{3}} \cdot (\ln n)^{\frac{1}{3}} \cdot (\ln \ln n)^{\frac{2}{3}}\right)}.$$

Kommen wir nun zu den Eigenschaften, die ein Paar (a, b) erfüllen muss um als geeignet zu gelten:

Definition 3.64 (gute Relation)
Sei $a, b \in Z$. Die Relation (a, b) heißt eine *gute Relation*, wenn folgende 3 Eigenschaften erfüllt sind:

1. $ggT(a, b) = 1$,
2. $a + bm$ zerfällt über der rationalen Faktorbasis RFB,
3. $a + b\rho$ zerfällt über der algebraischen Faktorbasis AFB.

Die guten Relationen sind jene, aus denen wir dann später versuchen werden, eine Teilmenge S davon zu finden, sodass $f'(m)^2 \cdot \prod_{(a,b)\in S} (a + bm)$ ein Quadrat in $\mathbb{Z}$ und $f'(\rho)^2 \cdot \prod_{(a,b)\in S} (a + b\rho)$ ein Quadrat in $\mathbb{Z}[\rho]$ ist.

Die wesentliche Grundidee des Siebens (die auch erst die optimalere Laufzeit ermöglicht) ist, dass man *NICHT*, wie es vielleicht intuitiv wäre, der Reihe nach ein konkretes Paar (a, b) aus dem Siebintervall nimmt und

1. man überprüft, ob der $ggT(a, b) = 1$ ist,
2. man für dieses konkrete (a, b) alle $p \in$ RFB nimmt, die höchste Potenz von p aus $a + b\,m$ herausdiviert und so feststellt, ob $a + b\,m$ über RFB zerfällt,

3. man der Reihe nach alle $p \in$ AFB nimmt, die höchste Potenz von p aus $N(a+b\rho) = F(a,-b)$ herausdiviert und so feststellt, ob $a+b\rho$ über AFB zerfällt.

Bei der soeben beschriebenen Vorgehensweise würden viele unnötige Berechnungen durchgeführt werden. Zum Beispiel brauchen jene a-Werte, die zu einem vorgegebenen b nicht teilerfremd sind, nicht durch probieren (dh. durch Berechnung von $ggT(a,b) \ \forall \ a \in [-C,C]$) herausgefunden werden, sondern hinter diesen Werten steckt ein System, welches uns das nun folgende Beispiel illustrieren soll und die Kernidee des Siebens darstellt:

Beispiel 3.65 (Grundprinzip des Siebens)
Sei $b = 10$. Für welche a im Siebintervall $[-1000, 1000]$ ist $ggT(a,b) = 1$?
Dazu betrachten wir die Primfaktorenzerlegung von $b = 10 = 2 \cdot 5$. Klarerweise gilt mit $k \in \mathbb{Z}$ für jedes Vielfache $k \cdot 2$ von 2, dass $ggT(k \cdot 2, 10) > 1$ ist. Somit können wir alle Vielfachen von 2 schon als geeignete a-Werte ausschließen. Genauso wissen wir, dass auch kein Vielfaches von 5 als ein a-Wert mit $ggT(a,b) = 1$ in Frage kommen kann.
Nachdem wir dieses *Sieb* für alle Primfaktoren von $b = 10$ angewendet haben, wissen wir, dass die übrigen a-Werte, die also weder ein Vielfaches von 2 noch ein Vielfaches von 5 sind, teilerfremd zu 10 sein müssen. Wir brauchen also nicht 2001 teure ggT-Berechnungen durchführen (für jedes $a \in [-1000, 1000]$: $ggT(a,b)$ berechnen), um unsere gesuchten Werte für a zu erhalten, sondern wir gehen nach dem Ausschließungsprinzip vor.

Sieben bedeutet also, dass man das Wissen, dass jene Elemente, die eine bestimme Eigenschaft besitzen, immer in einem bestimmen Abstand vorkommen, ausnützt, indem man sich nur *einen einzigen* Wert (bzw. ein nur ein paar wenige) mit einer gewünschten Eigenschaft ausrechnet und dann die restlichen Werte ganz einfach durch sukzessives Addieren bzw. Subtrahieren des bekannten Abstandes erhält. Dabei läuft für ein festgehaltenes Siebintervall die äußerste Schleife über alle Primzahlen (entweder aus RFB oder aus AFB) und erst die innere Schleife über die a-Werte, wobei man hierbei die für ein p unpassenden a-Werte gar nicht betrachtet, sondern (mit Schrittweite p) überspringt.

In den nachfolgenden 3 Siebschritten (ggT-Sieb, rationales Sieb, algebraisches Sieb) wollen wir beschreiben, was beim „line sieving“ für einen einzigen, festgehaltenen Wert b passiert:

ggT-Sieb
Wollen wir also das schon in Bsp. 3.65 erkennbare Vorgehen zum Auffinden jener Werte $a \in [-C,C]$ mit $ggT(a,b) = 1$ allgemein zusammenfassen:

Für ein festes b stellen wir als Erstes ein boolsches Array B auf, das gleich groß wie das Siebintervall ist und an allen Stellen $B[a]$ mit dem Wert $true \ \forall a \in [-C,C]$ initialisiert wird. $B[a]$ soll am Ende des ggT-Siebes angeben, ob $ggT(a,b) = 1$ ist. Dieses Array B wird später auch noch in den beiden folgenden Siebschritten weiterverwendet werden.
Für ein konkretes b berechnen wir uns im ggT-Sieb-Schritt die Primfaktorzerlegung $b = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$. Für jeden Primfaktor p_i von b erkennen wir, dass p_i und alle seine Vielfachen zumindest den Faktor p_i gemeinsam haben und somit nicht teilerfremd zu b sein können. Daher können wir $B[a] := false$ setzen für alle $a = k \cdot p_i \quad \forall \, k \in \mathbb{Z}$ mit $a \in [-C,C]$. Dazu berechnen wir uns für jedes p_i den *Startpunkt* a_0, dh. jenen kleinsten Wert $a_0 \in [-C,C]$, sodass $ggT(a_0,b) > 1$ ist. Diesen kann man zB. mittels der Formel

$$a_0 \ := \ -C + \big(C \bmod p_i\big)$$

berechnen. Anschließend können wir im Array B bei a_0 beginnend und im Abstand p_i fortschreitend alle Einträge im Bereich $[-C,C]$ auf $false$ setzen.
Dies machen wir für alle Primfaktoren p_i von b. Jene Werte a, für die dann noch immer

input : $b \in \mathbb{N}$,
Siebintervallgrenze C
output : boolsches Array B mit: $B[a] = true \Leftrightarrow ggT(a,b) = 1 \quad \forall\ a \in [-C, C]$

$B[a] := true \quad \forall\ a \in [-C, C]$; `// Initialiserung des Ergebnisarrays`
Berechnen der Primfaktorzerlegung $b = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$
for $i = 1, \dots, r$ **do**
  $a_0 := -C + \big(C \bmod p_i\big)$; `// Berechnung des Startpunktes`
  for $a = a_0$ **by** p_i **to** C **do**
    $B[a] := false$; `// Ausschließen der Vielfachen von` p_i
  end
end
return B

Algorithmus 3.66: ggT-Sieb

$B[a] = true$ gilt, sind folglich sicherlich teilerfremd zu b.
Das soeben beschriebene Vorgehen ist in Algorithmus 3.66 zusammengefasst.

Rationales Sieb
Nachdem wir für einen festgehaltenen Wert b das ggT-Sieb angewandt haben, ist es nun das Ziel des rationalen Siebens, Paare (a, b) zu finden, sodass $a + bm$ über der rationalen Faktorbasis RFB zerfällt, dh. dass jede Zahl $a + bm$ nur Primfaktoren besitzt, die (als 1. Komponente) in RFB vorkommen. Dabei werden wir das boolsche Array B, welches wir im ggT-Sieb-Schritt zuvor gerade erstellt haben, weiterverwenden, um unnötige Rechenschritte zu vermeiden.
Dazu erstellen wir uns ein weiteres Array R, das für alle $a \in [-C, C]$ den Wert

$$R[a] := a + bm$$

enthält. Man beachte, dass die Differenz zweier benachbarter Werte in R immer genau 1 ist. Somit braucht man nicht für jedes $a \in [-C, C]$ die Berechnung von $a+bm$ extra durchzuführen, sondern es reicht, sich den Wert für $R[-C] = -C + bm$ zu berechnen, und jeder weitere Wert in R ergibt sich durch Addition von 1 zu seinem Vorgänger.
Ob man, wie soeben beschrieben, das Array R für *alle* (also auch für die wegen des ggT-Siebes schon auszuschließenden a-Werte) initialisiert oder ob man nur für jene a mit $B[a] = true$ die Berechnung $R[a] = a + bm$ durchführt, hängt von den Gegebenheiten des Programms bzw. der Programmiersprache sowie der Hardwareressourcen ab, mit denen man arbeitet.[19]

Bei der Suche nach jenen Werten in R, die komplett über RFB zerfallen, brauchen wir - wie auch schon beim ggT-Sieb - *nicht* für jeden Werte $a + bm$ einzeln zu testen, ob er durch einen bestimmten Primfaktor teilbar ist. Denn ähnlich wie beim ggT-Sieb wissen wir: Angenommen $R[a] = a + bm$ ist durch p teilbar ist, dann ist auch $R[a + p] = (a + p) + bm$ durch p teilbar. Und alle Zahlen zwischen diesen beiden Werten können *nicht* durch p teilbar sein.
Dh. um zu testen, welche der Zahlen im Array R komplett über der RFB zerfallen, nehmen wir uns der Reihe nach alle Primfaktoren p_i aus RFB her. Für jedes p_i berechnen wir ein a_0 sodass $a_0 + bm$ durch p_i teilbar ist und dividieren aus $R[a_0]$ die höchste Potenz von p_i heraus. Anschließend schreiten wir im Array R im Abstand p_i fort und dividieren $R[a + k \cdot p_i]$ ebenfalls durch die höchst mögliche Potenz von p_i $\forall\ k \in \mathbb{Z}$ mit $a_0 + k \cdot p_i \in [-C, C]$. Dies führt man für

[19] Bei meiner Implementierung in Maple ist eine Initialisierung des *gesamten* Arrays $R :=$ Array$\big(-C\,..\,C\,, [\$\text{-}C{+}b{*}m\,..\,C{+}b{*}m]\big)$ schneller als eine for-Schleife mit einer if-Abfrage.

alle Primzahlen p_i in RFB durch.
Wenn dann für ein a: $R[a] = 1$ gilt, dann zerfällt $a + b\,m$ offensichtlich über der rationalen Faktorbasis. In diesem Fall besteht für das Paar (a, b) auch weiterhin die Möglichkeit, etwas zu einer erfolgreichen Faktorisierung von N beizutragen.
Sollte für ein a: $R[a] > 1$ gelten[20], dann wissen wir, dass es einen Primfaktor in $a + b\,m$ gibt, der nicht in der rationalen Faktorbasis enthalten ist. Somit zerfällt $a + b\,m$ nicht über RFB und wir brauchen das Paar (a, b) bei unseren weiteren Überlegungen nicht mehr betrachten. Deshalb ändern wir den Wert des boolschen Arrays an solchen Stellen a auf $B[a] := false$.

Wir müssen zum Glück nicht p_i verschiedene Werte durchprobieren, um einen Ausgangspunkt a_0 zu finden, von dem aus wir in Schrittweite p_i fortschreiten können. Wegen

$$p_i \,|\, a + b\,m \quad \Leftrightarrow \quad a + b\,m \equiv 0 \mod p_i \quad \Leftrightarrow \quad a \equiv -b\,m \mod p_i$$

wissen wir, dass $-b\,m \mod p_i$ ein geeigneter Wert für a ist, von dem aus wir mit Schrittweite p_i alle restlichen Werte in R erreichen, die durch p_i teilbar sind.
Der Startpunkt a_0 soll bei uns nicht irgendein Wert sein mit $p_i \,|\, a_0 + bm$, sondern wir wollen wiederum - analog zum ggT-Sieb - a_0 als den *kleinsten* Wert im Intervall $[-C, C]$ mit $p_i \,|\, a_0 + b\,m$ definieren. Den Startpunkt a_0 für eine for-Schleife im Abstand p_i können wir dabei mit folgender Formel berechnen:

$$\begin{aligned} seed &:= -b\,m \mod p_i \\ a_0 &:= -C + \Big((C + seed) \bmod p_i\Big) \end{aligned}$$

Für die Berechnung des Wertes *seed* benötigen wir für verschiedene b immer denselben Wert $m \mod p_i$, weshalb wir diesen schon bei der Erstellung der rationalen Faktorbasis einmalig vorweg berechnet und als zweite Komponente der Elemente in RFB gespeichert haben. Zur Erinnerung: RFB $= \Big\{(p, \underbrace{m \mod p}_{=:r}) \;\Big|\; p \in \mathbb{P},\ p \le B_{\text{rat}}\Big\}$. Tatsächlich berechnen wir *seed* mittels

$$seed := -b\,r_i \mod p_i\,,$$

wobei die Einführung der zweiten Komponente r_i der RFB-Elemente keinen tieferen mathematischen Hintergrund hat, sondern nur dazu dient, etwas Rechenzeit einzusparen.

Nachdem wir nun den Startpunkt a_0 berechnet haben, schreiten wir das Array R bei a_0 beginnend mit Schrittweite p_i ab. Das Herausdividieren der höchsten Potenz $p_i^{e_i}$ von $R[a]$ findet natürlich nur dann statt, wenn a noch ein möglicher Kanditat für eine gute Relation ist. Dh. nur dann, wenn $B[a] = true$ gilt, führen wir die relativ „teuren“ Divisionen durch, überschreiben den Wert in $R[a]$ und speichern[21] für jedes Paar (a, b) die Vielfachheit e_i des Primfaktors p_i in $R[a]$ in einer schwach besetzten Matrix, (die für jedes neue b neu leer initialisiert wird).

Nachdem wir dies für alle Primfaktoren p_i in RFB durchgeführt haben, sehen wir uns die Einträge im Array R der Reihe nach an:
Für jene Werte a, für die dann $R[a] = 1$ gilt, wissen wir, dass $a + b\,m$ über der rationalen Faktorbasis zerfällt. Das Paar (a, b) bleibt deshalb auch weiterhin ein möglicher Kanditat für eine gute Relation und deshalb lassen wir den Eintrag $B[a] = true$ unverändert.
Für jene a mit $R[a] > 1$ wissen wir, dass $a + b\,m$ (mindestens) einen Primfakor p mit $p > B_{\text{rat}}$ enthält. Deshalb zerfällt $a + b\,m$ *nicht* über RFB und wir setzen für diese a das boolsche Array B auf $B[a] := false$, da wir das Paar (a, b) schon ausschließen können.

Algorithmus 3.67 fasst das rationale Sieb nochmals zusammen.

[20] Bei unserer Wahl der Parameter gilt für realistische Faktorisierungsaufgaben immer $C < m$, weshalb $a + b\,m$ immer positiv ist und wir deshalb das Vorzeichen von $a + b\,m$ nicht berücksichtigen müssen.

[21] Sollte die Relation (a, b) auch den algebraischen Siebschritt überleben und später sogar in S aufgenommen werden, so können wir die rationale Wurzel leicht anhand der hier berechneten Vielfachheiten e_i erzeugen.

input : $b \in \mathbb{N}$, $m \in \mathbb{N}$
Siebintervallgrenze C
boolsches Array B
rationale Faktorbasis RFB
output : boolsches Array B

$R[a] := a + b\,m \qquad \forall\ a \in [-C, C]$
(Initialisierung evt. einschränken auf nur jene a mit $B[a] = true$)

for (p, r) **in** RFB **do**
  $seed := -b\,r \mod p$; // beliebiger passender Wert für a
  $a_0 := -C + \Big((C + seed) \mod p\Big)$; // Startpunkt berechnen
  for $a = a_0$ **by** p **to** C **do** // nur für potentielle a
    if $B[a] = true$ **then** // nur wenn a das ggT-Sieb bestanden hat
      $e := 0$
      while $p \mid R[a]$ **do** // durch die höchste Potenz von p dividieren
        $R[a] := \frac{R[a]}{p}$
        e := e+1
      end
    end
    if $e \geq 1$ **then**
      Vielfachheit e (von p in $a + b\,m$) speichern
    end
  end
end
for $a = -C$ **to** C **do** // boolsches Array aktualisieren
  if $R[a] > 1$ **then** // wenn $a + b\,m$ nicht über RFB zerfällt ...
    $B[a] := false$; // ... a für das algebraische Sieb ausschließen
  end
end
return B

Algorithmus 3.67: rationales Sieb

Algebraisches Sieb
Beim algebraischen Sieben geht es darum, für unser festgehaltenes b unter den Werten $a \in [-C, C]$, welche sowohl das ggT-Sieb als auch das rationale Sieb überstanden haben, jene zu finden, sodass die Norm $N(a+b\rho) = F(a,-b)$ über der algebraischen Faktorbasis zerfällt. Dazu stellen wir uns ein Array N auf und initialisieren dieses Array nur für jene Indizes $a \in [-C, C]$ mit den Werten

$$N[a] \;:=\; F(a,-b)\,,$$

für die noch immer $B[a] = \mathit{true}$ gilt.[22]

Nach der Arrayinitialisierung nehmen wir der Reihe nach die Elemente der algebraischen Faktorbasis AFB und dividieren „alle" Arrayeinträge in N jeweils durch die höchste Potenz von p. Wiederum braucht man für ein festes p nicht alle Einträge von N durchzuprobieren. Wegen Satz 3.62 und Satz 3.59 wissen wir bereits, dass für ein festes b und ein festes Element $(p,r) \in$ AFB der Primfaktor p genau dann ein Teiler von $F(a,-b)$ ist, wenn $a \equiv -b\,r \mod p$ gilt. Dh. die Restklasse $-b\,r \mod p$ ist ein geeigneter Kanditat für a, von dem aus wir im Array N wieder nur jeden p-ten Eintrag betrachten brauchen. Und auch nur dann, wenn im boolschen Array B an dieser Stelle *true* gespeichert ist.
Im Falle $p \mid b$ kann es kein *teilerfremdes* a geben, sodass $F(a,-b) \equiv 0 \mod p$ gilt. Wenn die erste Komponente p eines Elementes (p,r) in AFB ein Teiler des festgehaltenen b ist, dann kann man also dieses Element der AFB überspringen (s. Satz 3.62).
Für ein p mit $r_1 \not\equiv r_2 \in R(p)$ und mit $p \nmid b$ folgt, dass $b^{-1} \mod p$ existiert und wegen

$$-b\,r_1 \equiv -b\,r_2 \mod p \quad\Leftrightarrow\quad -\left(b\cdot b^{-1}\right) r_1 \equiv -\left(b\cdot b^{-1}\right) r_2 \mod p \quad\Leftrightarrow\quad r_1 \equiv r_2 \mod p$$

können wir schließen, dass sich die beiden arithmetischen Folgen der potentiell geeigneten a-Werte für (p,r_1) und für (p,r_2) nicht überschneiden können. Dh. beim Fortschreiten mit Schrittweite p können wir einerseits keinen durch p teilbaren Arrayeintrag von N übersehen, und andererseits treffen wir für ein p mit mehreren $r \in R(p)$ auch nicht (unnötigerweise) zweimal auf dasselbe Arrayfeld. Wenn also $p \mid N(a+b\rho)$ gilt, dann ist für alle Potenzen von p in $N(a+b\rho)$ ein und dasselbe r dafür „verantwortlich". Beim Konstruieren des algebraischen Quadrates $Q_2 = f'(\rho)^2 \cdot \prod_{(a,b)\in S} (a+b\rho) \in \mathbb{Z}[\rho]$ werden wir dann darauf achten, dass im Faktor $N\left(\prod_{(a,b)\in S} (a+b\rho)\right)$ von $N(Q_2)$ nicht nur jeder Primfaktor p in gerader Vielfachheit auftritt, sondern auch, dass die Vielfachheiten, die auf ein konkretes $(p,r) \in$ AFB zurückzuführen sind, für sich selbst genommen bereits ebenfalls alle gerade sind. Dh. für jede Relation (a,b) speichern wir in einem Exponentenvektor für alle Elemente (p,r) von AFB die jeweilige Vielfachheit von p in $N(a+b\rho)$.

Im Array N können wegen $N[a] = F(a,-b)$ auch negative Werte auftreten. Deshalb speichern wir auch das Vorzeichen von $N[a]$. Sollte $N[a]$ negatives Vorzeichen haben, so speichern wir uns für dass Paar (a,b) den Wert 1; und sollte $N[a]$ positiv sein, so speichern wir für (a,b) den Wert 0. Damit wird die multiplikative Struktur $\{1,-1\}$ auf die additive Struktur $\{0,1\} = \mathbb{Z}_2$ übertragen, mit der wir dann später arbeiten wollen.
Ob $N[a]$ über AFB zerfällt, sehen wir daran, dass nach dem abarbeiten aller Elemente aus AFB im Arrayfeld $N[a]$ der Wert +1 oder −1 gespeichert ist. Für andere Einträge zerfällt $F(a,-b)$ offensichtlich *nicht* über AFB und für diese a setzen wir $B[a] := \mathit{false}$. Somit gibt uns das Array B nun an, welche Relationen (a,b) „gute Relationen" sind.

Die soeben beschriebenen Schritte werden ebenfalls nochmals, in Algorithmus 3.68 zusammengefasst.

[22] Aufgrund des inhaltlichen Zusammenhangs $\big(N[a] = N(a+b\rho)\big)$ tragen die Normfunktion $N(\alpha)$ für $\alpha \in \mathbb{Z}[\rho]$ und das Array $N[a]$ mit $a \in [-C, C]$ denselben Namen. Man kann aber stets sowohl aus dem Kontext als auch anhand der Schreibweise $\big(N[.]$ vs. $N(.)\big)$ leicht erkennen, um welches der beiden es sich gerade handelt.

input : $b \in \mathbb{N}$, $m \in \mathbb{N}$
Siebintervallgrenze C
boolsches Array B
Polynom $F(x, y)$
algebraische Faktorbasis AFB
output : boolsches Array B

for $a = -C$ **to** C **do**
  if $B[a] = true$ **then** // wenn a ggT- & rat. Sieb bestanden hat ...
    $N[a] := F(a, -b)$ // ... initialisieren der Norm
  end
end

for (p, r) **in** AFB **do**
  $seed := -br \mod p$; // beliebiger passender Wert für a
  $a_0 := -C + \Big((C + seed) \mod p\Big)$; // Startpunkt berechnen
  for $a = a_0$ **by** p **to** C **do** // nur für potentielle a
    if $B[a] = true$ **then** // nur wenn a auch das rat. Sieb bestanden hat
      $e := 0$
      while $p \mid N[a]$ **do** // durch die höchste Potenz von p dividieren
        $N[a] := \frac{N[a]}{p}$
        e := e+1
      end
    end
    if $e \geq 1$ **then**
      Vielfachheit e $\big($von p in $F(a, -b)\big)$ speichern
    end
  end
end
for $a = -C$ **to** C **do** // boolsches Array aktualisieren
  if $N[a] \neq \pm 1$ **then** // wenn $N(a + b\rho)$ nicht über RFB zerfällt ...
    $B[a] := false$; // ... a ausschließen
  end
end
return B

Algorithmus 3.68: algebraisches Sieb

Quadratische Charaktere

Zwar könnten wir an dieser Stelle schon versuchen Quadrate $Q_1 \in \mathcal{R}_1 = \mathbb{Z}$ und $Q_2 \in \mathcal{R}_2 = \mathbb{Z}[\rho]$ zu erstellen, aber wie wir im vergleichenden Bsp. 3.63 schon gesehen haben, wissen wir nicht, ob die von uns konstruierten Objekte auch tatsächlich Quadrate von Elementen in $\mathcal{R}_1$ bzw. $\mathcal{R}_2$ sind. Um diese Wahrscheinlichkeit zu erhöhen verwenden wir - basierend auf einer Idee von Adleman - *quadratische Charaktere*[23]. Der folgende Satz liefert wiederum eine notwendige Bedingung, die das von uns konstruierte Objekt $Q_2 \in \mathcal{R}_2$ erfüllen muss, wenn es ein Quadrat sein soll [Lj93, Prop. 8.3, S. 68], [Bri98, Th. 3.2.1, S. 21]:

Satz 3.69

Sei $q \in \mathbb{P}$ ungerade und $s \in R(q)$ mit

$$\begin{aligned} a + b\,s &\equiv 0 \mod q \qquad \forall\,(a,b) \in S \qquad \text{und} \\ f'(s) &\not\equiv 0 \mod q \end{aligned}$$

sodass $\prod_{(a,b)\in S} (a + b\,\rho)$ ein Quadrat in $Q(\rho)$ ist. Dann gilt

$$\prod_{(a,b)\in S} \left(\frac{a+bs}{q}\right) = 1\,.$$

Auch in diesem Fall sind wir eigentlich an der umgekehrten Richtung des Satzes interessiert, die hier zum Glück in gewisser Weise gilt [Lj93, S. 69]: Wir wählen S so aus, dass $\prod_{(a,b)\in S} \left(\frac{a+bs}{q}\right) = 1$ ist für alle $(q,s) \in$ QCB. Wenn QCB genügend groß ist, dann sollte $\prod_{(a,b)\in S} (a + b\,\rho)$ auch tatsächlich ein Quadrat in $Q(\rho)$ sein.

Um die Menge S so auswählen zu können, dass $\prod_{(a,b)\in S} \left(\frac{a+bs}{q}\right) = 1$ ist, berechnen wir für jedes a mit $B[a] = true$ und jedes Element (q,s) in QCB den Wert des Legendre-Symbols $\left(\frac{a+b\,s}{q}\right)$. Abhängig von diesem Wert speichern wir dann - ganz analog zum Vorzeichen der Norm $N(a+b\rho)$ - den Wert

$$\begin{aligned} 0, &\quad \text{falls } \left(\tfrac{a+b\,s}{q}\right) = +1 \\ 1, &\quad \text{falls } \left(\tfrac{a+b\,s}{q}\right) = -1\,. \end{aligned}$$

Relationenmatrix erstellen

Für ein festes b können wir uns die Einträge, die wir im Zuge des rationalen und des algebraischen Siebes errechnet haben, sowie die soeben beschriebenen Werte für die quadratischen Charaktere, wie in Abbildung 3.4 dargestellt, vorstellen.

Eine Spalte entspricht einem Exponentenvektoren für ein konkretes Paar (a,b). Um in der Teilmenge S eine gerade Anzahl an Elementen auszuwählen[24], speichern wir an der letzten Stelle im Exponentenvektor von (a,b) immer den Wert 1. Wir können also mit den bisher beschriebenen Einträgen für jedes (a,b) einen Exponentenvektor $e_{(a,b)}$ bilden.

All jene Elemente a mit $B[a] = true$ haben alle drei Siebschritte „überlebt“ und liefern uns schließlich eine gute Relation (a,b). Für jede gute Relation speichern wir die bisher über sie gewonnenen Informationen in eine Matrix M. Dh. M enthält der Reihe nach folgende Einträge bzw. Blöcke von Einträgen:

[23] Zwar ist die quadratische-Charaktere-Basis genauso eine „Basis“ wie RFB und AFB auch, jedoch mit dem Unterschied, dass wir ein Element der QCB nicht nur *auf einen Teil* der noch potentiellen a-Werte anwenden, sondern wirklich für *jedes* noch potentielle a und *jeden* quadratischen Charakter (q,s) eine Berechnung durchführen.

[24] [Lj93, S. 85] bzw. [Zay95, S. 17]

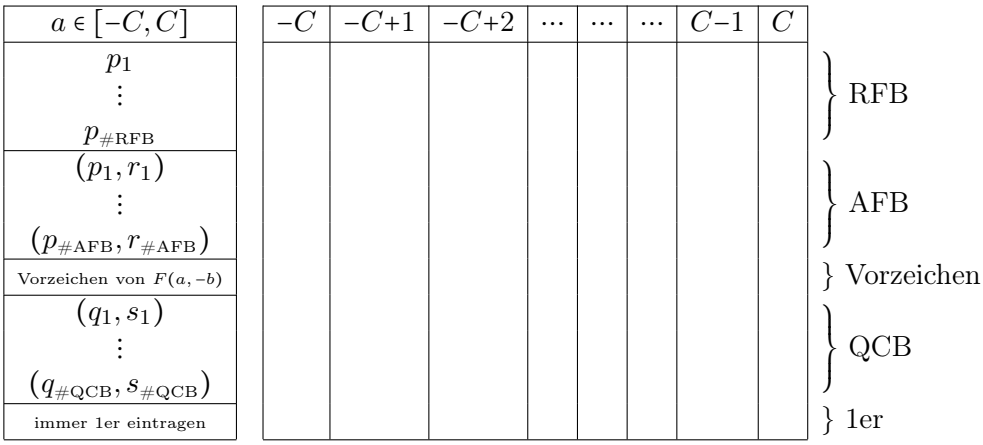


Abbildung 3.4: Zahlkörpersieb - Aufbau der Relationenmatrix M

- Für jedes Element $(p, m \bmod p)$ aus der rationalen Faktorbasis RFB wird die Vielfachheit des Primfaktors p in $a + b\,m$ gespeichert.
- Für jedes Element (p, r) aus der algebraischen Faktorbasis AFB speichern wir die Vielfachheit von p in $F(a, -b)$, sofern $a \equiv -b\,r \mod p$ erfüllt ist.
- Da $F(a, -b)$ auch negativ sein kann, speichern wir für das Vorzeichen von $N[a]$ folgenden Wert:
$$\frac{1 - N[a]}{2} = \begin{cases} 0, & \text{falls } N[a] = +1 \\ 1, & \text{falls } N[a] = -1 \end{cases} .$$
- Für jedes Element (q, s) in der quadratischen-Charaktere-Basis QCB speichern wir den Wert:
$$\frac{1 - \left(\frac{a+b\,s}{q}\right)}{2} = \begin{cases} 0, & \text{falls } \left(\frac{a+b\,s}{q}\right) = +1 \\ 1, & \text{falls } \left(\frac{a+b\,s}{q}\right) = -1 \end{cases} ,$$
wobei $\left(\frac{a+b\,s}{q}\right)$ das Legendre-Symbol bezeichnet.
- An letzter Stelle speichern wir immer den Wert 1.

Sei k definiert als die Anzahl der soeben beschriebenen Einträge:

$$k := \#\mathrm{RFB} + \#\mathrm{AFB} + 1 + \#\mathrm{QCB} + 1 .$$

Nachdem wir die Informationen der guten Relationen in die Matrix $M \in \mathbb{N}^{k \times (k+1)}$ übertragen haben, erhöhen wir den Wert von b um eins und wiederholen ab dem ggT-Sieb dieselben Schritte für $b + 1$. Dies führen wir solange aus, bis wir $k + 1$ Relationen gefunden haben. Da wir um eine Relation mehr als k gefunden haben, besteht unter den Spalten in M jedenfalls lineare Abhängigkeit.

Matrixschritt

Nachdem wir nun solange b erhöht und anschließend gesiebt haben, bis genügend gute Relationen gefunden wurden, ist es unser eigentliches Ziel, eine Teilmenge S der guten Relationen zu finden, sodass die Summe der Exponentenvektoren $\sum_{(a,b) \in S} e_{(a,b)}$ ein Vektor mit lauter geraden Einträgen ist (siehe 3.56, 3.61 und 3.69). Dh. es reicht, wenn wir die Matrix M modulo 2 betrachten:

$$M_2 := M \mod 2 \in (\mathbb{Z}_2)^{k \times (k+1)} .$$

Aufgrund der linearen Abhängigkeit der Spalten von M_2 können wir eine nicht-triviale Linearkombination l des Nullvektors in M_2 (modulo 2) finden:

$$M_2 \cdot l \;=\; \mathbf{0} \;\in\; {\mathbb{Z}_2}^k$$

Die $k+1$ Koeffizienten dieses Vektors l geben uns an, welche Teilmenge S der $k+1$ guten Relationen wir auswählen müssen, sodass $\prod_{(a,b)\in S}(a+bm)$ über der rationalen Faktorbasis und $\prod_{(a,b)\in S}(a+b\rho)$ über der algebraischen Faktorbasis zerfällt.

In der Regel finden wir aber nicht nur eine einzige nicht-triviale Linearkombination, sondern wir finden einen ganzen Lösungsraum, von dem wir eine Lösungsbasis berechnen. Für jeden Vektor der Lösungsbasis führen wir die restlichen Schritte durch, solange bis wir eine erfolgreiche Faktorisierung von N erreicht haben.
Sollte kein einziger Vektor der Lösungsbasis zum Erfolg führen, so kann dies mehrere Ursachen haben. Abhängig von der Ursache des Misserfolges haben wir mehrere Möglichkeiten zu reagieren: Wir können noch zusätzliche Elemente zur QCB hinzufügen, um die Wahrscheinlichkeit zu erhöhen, dass die von uns konstruierten Produkte auch tatsächlich Quadrate sind. Oder wir gehen noch einen Schritt weiter zurück und suchen noch mehr gute Relationen.

Die einfachste Methode zum Berechnen der Lösungsbasis ist Gauß-Elimination. Wenn die Matrix M klein genug ist, ist dieser Algorithmus auch praktikabel.
Da die Matrix M nur dünn besetzt ist, ist auch M_2 dünn besetzt, weil beim Übergang von M zu M_2 auch noch Einträge der ohnehin schon dünn besetzten Matrix M zu 0 werden können.
Es gibt auch eine Version der Gauß-Elimination, die auf die besondere Struktur der Matrix M_2 Rücksicht nimmt und versucht, im Verlauf der Berechnung der Lösungsbasis möglichst lange möglichst wenige Einträge zu bewahren [Odl85], [PS92], [LO91]. Diese werden auch *structured Gauss Methoden* genannt.

Es gibt aber spezielle, für dünn besetzte Matrizen über endlichen Körpern ausgelegte Algorithmen, die eine asymptotisch bessere Laufzeit haben. Dazu gehören die *conjugate gradient Methode* [Odl85], die *Block-Lanczos Methode* [Cop93] und der *Block Wiedemann Algorithmus* [Cop94].

Bemerkung 3.70
Der Matrix-Schritt wird natürlich umso aufwendiger, je größer die Matrix M_2 ist. Um diese Komplexität zu verringern kann man natürlich die Größe der Faktorbasen RFB und AFB verringern, was sich direkt auf die Größe von M_2 auswirkt. Dadurch aber wird auch die Wahrscheinlichkeit, dass $a+bm$ und $a+b\rho$ über RFB bzw. AFB zerfallen, kleiner und wir müssen eine längere Siebphase in Kauf nehmen.
Umgekehrt können wir die Siebphase verkürzen indem wir die Faktorbasen vergrößern, was wiederum mehr Speicherplatz benötigt und einen längeren Matrix-Schritt verursacht.
Diese beiden Parameter beeinflussen sich gegenseitig und können je nach den eigenen Gegebenheiten angepasst werden.

Rationale Wurzel berechnen
Nachdem wir nun einen Vektor l berechnet haben, der uns die Teilmenge S der guten Relationen definiert, sodass

$$\varphi_1\Bigg(\underbrace{f'(m)^2 \cdot \prod_{(a,b)\in S}(a+bm)}_{Q_1 \,=\, {W_1}^2}\Bigg) \;=\; \varphi_2\Bigg(\underbrace{f'(\rho)^2 \cdot \prod_{(a,b)\in S}(a+b\rho)}_{Q_2 \,=\, {W_2}^2}\Bigg) \;\in\; \mathbb{Z}_N$$

gilt, wird es unser nächstes Ziel sein, das Bild $\varphi_1(W_1) \in \mathbb{Z}_N$ der rationalen Wurzel $W_1 \in \mathbb{Z}$ zu berechnen. Dies ist eine relativ leichte Aufgabe:
Da jedes einzelne $a+bm$ über RFB zerfällt und die Vielfachheiten der Primfaktoren von $a+bm$ jeweils im Exponentenvektor $e_{(a,b)}$ gespeichert sind, kennen wir bereits auch für das Produkt $\prod_{(a,b)\in S}(a+bm)$ die Primfaktorzerlegung, welche wir anhand des Exponentenvektors

$$e_S := \sum_{(a,b)\in S} e_{(a,b)}$$

ablesen können. e_S lässt sich zwar schreiben als $e_S = M \cdot l$, jedoch ist die normale Matrizenmultiplikation an dieser Stelle unnötig aufwendig. Da wir M selbst schon als dünn besetzte Matrix gespeichert haben, sollten wir diese Eigenschaft auch bei der Berechnung von e_S ausnutzen.

Wenn wir mit $e_S[i]$ den i-ten Eintrag im Exponentenvektor e_S bezeichnen, dann gilt für das rationale Quadrat Q_1

$$Q_1 = f'(m)^2 \cdot \prod_{i=1}^{\#\mathrm{RFB}} {p_i}^{e_S[i]},$$

und somit können wir die rationale Wurzel W_1 durch Halbieren (der Einträge) des Exponentenvektors e_S erhalten:

$$W_1 = f'(m) \cdot \prod_{i=1}^{\#\mathrm{RFB}} {p_i}^{\frac{e_S[i]}{2}}.$$

Bemerkung 3.71
Würden wir dieses Produkt tatsächlich ausmultiplizieren, so erhielten wir eine Zahl, deren Stellenanzahl ein Vielfaches jener von N wäre. Da wir aber letzten Endes nur an der *Restklasse* von W_1 modulo N interessiert sind, können wir auch gleich alle Berechnungen modulo N ausführen und können so die Stelligkeit der Zahlen, mit denen wir arbeiten müssen, in der Größenordnung der zu faktorisierenden Zahl N halten.
Dh. für die rationale Seite wenden wir auch gleich den Homomorphismus φ_1 an.

Bemerkung 3.72
Es ist weder notwendig noch ratsam, Q_1 auszurechnen und anschließend davon die Wurzel in $\mathbb{Z}$ zu ziehen!

Algebraisches Quadrat berechnen
Auf der algebraischen Seite können wir leider nicht anhand des halbierten Exponentenvektors auf die algebraische Wurzel schließen. Deshalb berechnen wir uns als Erstes das algebraische Quadrat

$$Q_2 = f'(\rho)^2 \cdot \prod_{(a,b)\in S} (a+b\rho) \quad \in \mathbb{Z}[\rho].$$

Da wir in $\mathbb{Z}[\rho]$ arbeiten, können wir beim Aufmultiplizieren der linearen Polynome $a+b\rho$ immer gleich modulo $f(x)$ rechnen. Somit können wir den Grad zwar niedrig halten, aber wir müssen trotzdem mit Koeffizienten hantieren, die größenordnungsmäßig weit über N liegen.

Algebraische Wurzel ziehen
Wenn uns das algebraische Quadrat Q_2 als Polynom vom Grad kleiner d vorliegt, stellt sich für uns die Frage, wie wir ein Polynom W_2 (ebenfalls vom Grad kleiner d) finden, sodass ${W_2}^2 = Q_2 \mod f(x)$ gilt (wobei die Koeffizienten der beiden Polynome jeweils alle in $\mathbb{Z}$ liegen).
Um die Idee hinter dem folgenden Wurzel-Schritt zu vermitteln, möchten wir uns ein Beispiel dazu ansehen.

Beispiel 3.73
Sei $N = 181187$ die zu faktorisierende Zahl. Als Parameter seien $d = 3 \Rightarrow m = 56 \Rightarrow f(x) = x^3 + x^2 + 43x + 27$ gewählt. Seien weiters

$$\text{RFB} = \{[2,0],[3,2],[5,1],[7,0],[11,1],[13,4],[17,5],[19,18],[23,10],[29,27],[31,25],[37,19],[41,15],[43,13],[47,9],[53,3]\} \text{ und}$$
$$\text{AFB} = \{[2,1],[3,0],[3,1],[5,2],[7,5],[17,9],[23,12],[31,4],[37,20],[41,13],[43,14],[59,34],[61,11],[71,64],[79,34],[83,14],\ldots$$
$$\ldots[89,15],[101,80],[103,50],[107,67],[109,81],[137,65],[137,77],[137,131],[139,72],[139,84],[139,121],[149,17],\ldots$$
$$\ldots[149,37],[149,94],[151,118],[157,68],[157,103],[157,142],[163,26],[167,108],[167,110],[167,115]\}.$$

Dann ist

$$S = \{[-15,1],[-7,1],[-5,1],[0,1],[13,1],[28,1],[-25,3],[1,3],[2,3],[7,3],[14,3],[-19,4],[-27,5],[14,5]\}$$

eine geeignete Menge von Paaren (a,b) und es gilt

$$\begin{aligned} Q_2 &\equiv \left(3x^2+2x+43\right)^2 \cdot \left(9244462421178240\,x^2 - 147307489438344576\,x - 96683200997591808\right) \equiv \\ &\equiv 90872920988385555456\,x^2 - 991313705212563566592\,x - 662059347915890672640 \mod f(x) \end{aligned}$$

Die Frage ist nun, wie wir von Q_2 eine Quadratwurzel W_2 modulo $f(x)$ finden können.
Da die Koeffizienten von Q_2 etwa 20-stellig sind und bei einer Multiplikation zweier Polynome modulo $f(x)$ nur der Grad, nicht aber die Koeffizienten reduziert werden, würden wir erwarten, dass die Koeffizienten von W_2 in etwa 10-stellig sind. Eine Abschätzung für die Koeffizienten gibt uns der folgende Satz [Cou, 2.2, S. 99]: □

Satz 3.74 (obere Schranke für einzelne Wurzelkoeffizienten)
Sei $W_2 = w_{d-1}\,x^{d-1} + w_{d-2}\,x^{d-2} + \ldots + w_1\,x + w_0$ und sei $f(x) = x^d + a_{d-1}\cdot x^{d-1} + \ldots + a_1 \cdot x + a_0$ mit $\|f\| := \sqrt{\sum_{i=1}^{d} {a_i}^2}$. Weiters sei u eine obere Schranke für alle Beträge von a und b in S, dh. $u \geq \max(|a|,|b|) \quad \forall\ (a,b) \in S$. Dann gilt für den i-ten Koeffizienten w_i der Wurzel W_2

$$|w_i| \ \leq\ d^{\frac{3}{2}} \cdot \|f\|^{d-i} \cdot \left(2u\,\|f\|\right)^{\frac{|S|}{2}} \qquad \forall\ i = 0,\ldots,d-1$$

Korollar 3.75 (obere Schranke für alle Wurzelkoeffizienten)
Wegen $\|f\| \geq 1$ folgt aus Satz 3.74, dass

$$\max_{i=0,\ldots,d-1} |w_i| \quad \leq \quad d^{\frac{3}{2}} \cdot \|f\|^{d} \cdot \left(2u\,\|f\|\right)^{\frac{|S|}{2}} \ =:\ B_{\text{Koeff}} \qquad \forall\ i = 0,\ldots,d-1\,, \tag{3.23}$$

womit wir eine obere Schranke B_{Koeff} für *alle* Koeffizienten der Wurzel W_2 gefunden haben.

Beispiel 3.76 (Fortsetzung von Bsp. 3.73)
In unserem Beispiel gilt $u = 28$ und $\|f\| = \sqrt{1+1+43^2+27^2} = \sqrt{2580}$. Somit erhalten wir aus (3.23) die obere Schranke

$$B_{\text{Koeff}} \ =\ 3^{\frac{3}{2}} \cdot \sqrt{2580} \cdot \left(2 \cdot 28 \cdot \sqrt{2580}\right)^{\frac{14}{2}} \ =\ 22957059911700990394368000 0 \cdot \sqrt{3}\,.$$

□

Wenn wir ein $p \in \mathbb{P}$ finden mit $p > 2 \cdot B_{\text{Koeff}}$, sodass zusätzlich $f(x)$ irreduzibel $\mod p$ ist, dann ist

$$GF\left(p^d\right) \ \cong\ \mathbb{Z}_p[x] \Big/ \left(f(x) \bmod p\right).$$

Wenn das Produkt Q_2 tatsächlich ein Quadrat ist, dann können wir Q_2 auffassen als ein Element von $GF\left(p^d\right)$ - dh. als ein Polynom vom Grad höchstens $d-1$ und mit Koeffizienten in $\mathbb{Z}_p$ - und wir können davon *im Körper* $GF\left(p^d\right)$ die Wurzel $\widetilde{W} \in GF\left(p^d\right)$ ziehen. Die Bedingung $p > 2 \cdot B_{\text{Koeff}}$ benötigen wir, um von den Koeffizienten $r_i \in \mathbb{Z}_p$ von R auf die richtigen Koeffizienten $w_i \in \mathbb{Z}$ von W_2 schließen zu können. Dies kann man sehr gut an unserem Beispiel erkennen:

Beispiel 3.77 (Fortsetzung von Bsp. 3.76)
Sei $p = 678969358892993473514642059973813038909$. Dann ist $p > 2 \cdot B_{\text{Koeff}}$ und $f(x)$ ist irreduzibel mod p. Q_2 aufgefasst als Element von $GF\left(p^d\right)$ liefert uns

$$\begin{aligned} Q_2 &= 90872920988385555456\,x^2 - 991313705212563566592\,x - 662059347915890672640 = \\ &= 90872920988385555456\,x^2 + 678969358892993472523328354761249472317\,x + 678969358892993472852582712057922366269\,. \end{aligned}$$

Nehmen wir einmal an, wir wüssten schon, wie wir in $GF\left(p^d\right)$ eine Quadratwurzel berechnen können und wir erhielten

$$\widetilde{W} = 678969358892993473514642059972695508813\,x^2 + 678969358892993473514642059961379003901\,x + 678969358892993473514642059966425676909$$

als eine Wurzel von Q_2 in $GF\left(p^d\right)$. Eigentlich würden wir für die Koeffizienten der Wurzel erwarten, dass die Stelligkeit in etwa bei 10 läge, was hier offensichtlich aber (noch) nicht zutrifft. Wenn wir aber von den Koeffizienten von $\widetilde{W}$ einzeln jeweils den kleinsten Absolutrest modulo p nehmen, so erhalten wir

$$W_2 = -1117530096\,x^2 - 12434035008\,x - 7387362000 = \widetilde{W} \quad \in GF\left(p^d\right),$$

womit wir auch bei der erwarteten Stelligkeit von etwa 10 liegen. Es lässt sich auch leicht nachrechnen, dass nicht nur in $GF\left(p^d\right)$ sondern auch in $\mathbb{Z}[x]/f(x)$ gilt:

$$\begin{aligned} {W_2}^2 &= \left(-1117530096\,x^2 - 12434035008\,x - 7387362000\right)^2 = \\ &= 1248873515465769216\,x^4 + 27790816672315201536\,x^3 + 17111642531026306406 4\,x^2 + 18370943544953779200 0\,x + 54573117319044000000 \equiv \\ &\equiv 90872920988385555456\,x^2 - 991313705212563566592\,x - 662059347915890672640 = Q_2 \mod f(x) \end{aligned}$$

Mit W_2 ist natürlich auch $-W_2$ eine Wurzel von Q_2 (hierbei ist mit „$-W_2$“ das additiv Inverse in $\mathbb{Z}$ gemeint, nicht in $GF(p^d)$). □

Da wir nun wissen, warum wir $p > 2 \cdot B_{\text{Koeff}}$ wählen, müssen wir uns nur überlegen, *wie* wir im Körper $GF\left(p^d\right)$ eine Quadratwurzel berechnen können. Hierfür werden wir eine Verallgemeinerung des Algorithmus' von Shanks verwenden, die auf der Quadratwurzelberechnung in $\mathbb{Z}_p$ basiert [Zay95, Kap. 5.1.1, S. 107-112].
Dazu wollen wir uns nochmals in Erinnerung rufen, wie wir aus einem Quadrat q modulo $p \in \mathbb{P}$ eine Quadratwurzel r ziehen (siehe auch [CP00, Alg. 2.3.8, S. 94]):

Bemerkung 3.78
Sei $p \in \mathbb{P}$ ungerade und q gegeben mit $\left(\frac{q}{p}\right) = 1$, dh. q ist ein Quadrat in $\mathbb{Z}_p$. Dann lässt sich die Wurzel r von q wie folgt berechnen:

1. Für den Fall $p \equiv 3 \mod 4$ ist die Wurzel r gegeben durch $r := q^{\frac{p+1}{4}} \mod p$.
2. Für den Fall $p \equiv 5 \mod 8$ berechnen wir uns
   - $x := q^{\frac{p+3}{8}} \mod p$
   - Wenn $x^2 \equiv q \mod p$ ist, dann setze $r := x$;
     im Falle $x^2 \not\equiv q \mod p$ setze $r := x \cdot 2^{\frac{p-1}{4}} \mod p$.
3. Für den Fall $p \equiv 1 \mod 8$ stellen wir $p - 1$ als $p - 1 = 2^k \cdot u$ mit ungeradem u dar. Weiters benötigen wir einen quadratischen Nicht-Rest v um $w := v^u$ zu berechnen. e sei dann jener Exponent, sodass $w^e \equiv q^u \mod p$ ist. Dann ist die Wurzel r gegeben durch $r := q^{\frac{u+1}{2}} \cdot w^{\frac{-e}{2}} \mod p$.

Die ersten beiden Fälle sind jene, die am einfachsten zu programmieren sind und die auch am wenigsten Rechenaufwand haben. Zusätzlich enthält der dritte Fall mit dem Suchen nach einem quadratischen Nichtrest einen Bestandteil mit unkontrollierbarer Laufzeit.

Das Prinzip der Fallunterscheidung in 3 Teile bei der Wurzelberechnung lässt sich vom Körper $\mathbb{Z}_p$ mit p Elementen auf den Körper $GF\left(p^d\right) \cong \mathbb{Z}_p[x]/\left(f(x) \mod p\right)$ mit p^d Elementen

input : $f(x)$
obere Schranke B_{Koeff} für Koeffizienten des Wurzelpolynoms W_2
algebraisches Quadrat Q_2 als Polynom in $\mathbb{Z}[x]$ mod $f(x)$
output : algebraische Wurzel W_2 als Polynom in $\mathbb{Z}[x]$ mod $f(x)$

$gefunden := false$
$p := 2 \cdot B_{\text{Koeff}} + 1$
while $gefunden = false$ **do**
  $p := nextprime(p)$; // Maple-interner Befehl
  $rest := (p \mod 8)^d \mod 8$
  if $rest \in \{3, 5, 7\}$ **then**
    if $f(x)$ ist irreduzibel modulo p **then**
      $gefunden := true$
    end
  end
end
if $(rest \mod 4) = 3$ **then**
  $W_2 := (Q_2)^{\frac{p^d+1}{4}}$
else // $(rest \mod 8) = 5$
  if $(Q_2)^{\frac{p^d-1}{4}} = 1$ **then**
    $W_2 := (Q_2)^{\frac{p^d+3}{8}}$
  else
    $W_2 := (Q_2)^{\frac{p^d+3}{8}} \cdot 2^{\frac{p-1}{4}}$
  end
end
return W_2

Algorithmus 3.79: algebraische Quadratwurzel

übertragen. Normalerweise sind p und q gegeben, sodass man auf keinen der 3 Fälle verzichten kann. Für unsere Aufgabenstellung jedoch müssen wir aber erst ein geeignetes p suchen, dh. ein $p > 2 \cdot B_{\text{Koeff}}$ mit $f(x)$ irreduzibel mod p. Daher haben wir beim Zahlkörpersieb die Möglichkeit, für p zusätzlich $p^d \equiv 3, 5$ oder $7 \mod 8$ zu fordern.

Somit ergibt sich für uns Algorithmus 3.79 zur Berechnung der algebraischen Quadratwurzel.

Wurzeln nach $\mathbb{Z}_N$ abbilden

Nachdem wir nun die beiden Wurzeln $W_1 \in \mathcal{R}_1 = \mathbb{Z}$ und $W_2 \in \mathcal{R}_2 = \mathbb{Z}[x] \big/ f(x)$ berechnet haben, brauchen wir nur noch die beiden Homomorphisem φ_1 und φ_2 darauf anzuwenden und zu hoffen, dass

$$\varphi_1(W_1) \not\equiv \varphi_2(W_2) \mod N$$

gilt, sodass die Faktorisierung erfolgreich, dh. nicht-trivial ist (vgl. (3.17) auf S. 47).

Die Berechnung von $\varphi_1(W_1) \in \mathbb{Z}_N$ haben wir aus Performancegründen schon im Schritt „rationale Wurzel berechnen“ vorweg genommen (vgl. 3.71 auf S. 68) und braucht an dieser Stelle also nicht mehr zu erfolgen.

Bleibt uns also nur noch den Homomorphismus auf die algebraische Wurzel W_2 anzuwenden. Dazu nehmen wir das Polynom W_2 und setzen - je nach Sichtweise - m in das Polynom W_2 ein bzw. ersetzen in W_2 die Nullstelle ρ durch den Wert m (vgl. (3.19) auf S. 49). Jedenfalls

erhalten wir als Bild die Restklasse

$$\varphi_2(W_2) \;=\; w_{d-1}\,m^{d-1} + w_{d-2}\,m^{d-2} + \ldots + w_1\,m + w_0 \qquad \bmod N\,.$$

Da wir schlussendlich ohnehin nur an der Restklasse mod N interessiert sind und die Koeffizienten w_i im Allgemeinen viel größere Stelligkeit als N haben, nehmen wir von den Koeffizienten von W_2 zuerst einzeln die Restklasse mod N und berechnen $\varphi_2(W_2)$ in der Praxis durch

$$\varphi_2(W_2) \;=\; (w_{d-1} \bmod N)\cdot m^{d-1} + (w_{d-2} \bmod N)\cdot m^{d-2} + \ldots + (w_1 \bmod N)\cdot m + (w_0 \bmod N) \qquad \bmod N\,.$$

Wenn nun für die Bilder

$$\varphi_1(W_1) \;\not\equiv\; \pm\varphi_2(W_2) \mod N$$

gilt (und die Quadrate davon in derselben Restklasse mod N liegen), dann liefern uns sowohl

$$ggT\big(\varphi_1(W_1) - \varphi_2(W_2), N\big)$$

als auch

$$ggT\big(\varphi_1(W_1) + \varphi_2(W_2), N\big)$$

jeweils einen nicht-trivialen Faktor von N.

Beispiel 3.80 (Fortsetzung von Bsp. 3.77)
Das Bild von W_2 unter φ_2 ergibt

$$\begin{aligned}\varphi_2(W_2) &= (-1117530096)\cdot 56^2 + (-12434035008)\cdot 56 + (-7387362000) \;\equiv \qquad \bmod 181187\\ &\equiv 31320\cdot 3136 + 104054\cdot 56 + 175551 \;=\; 98219520 + 5827024 + 175551 \;\equiv\\ &\equiv 16166 + 29040 + 175551 \;=\; 220757 \;\equiv\; 39570 \qquad \bmod 181187\,.\end{aligned}$$

Wollen wir nun auch noch die restlichen Werte für eine vollständige Faktorisierung unseres Beispiels anführen bzw. berechnen:

$$\begin{aligned}\mathrm{QCB} = \{&[173,10],[179,6],[193,47],[197,120],[211,101],[223,31],[227,86],[229,33],[241,25],[251,20],[257,65],[269,22],\ldots\\ \ldots&[271,206],[281,90],[283,108],[293,68],[307,119],[331,44],[337,186],[347,119],[359,145],[373,154],[389,159],[397,114],[409,56],\ldots\\ \ldots&[421,64],[431,209],[433,276],[443,21],[457,206],[461,252],[463,61],[479,228],[487,153],[491,166],[503,429],[523,463],[541,133],\ldots\\ \ldots&[547,445],[557,104],[563,16],[569,410],[577,93],[599,74],[617,494],[619,101],[641,75],[643,117],[647,52],[653,187],[659,509],\ldots\\ \ldots&[691,74],[709,160],[719,357],[727,318],[733,351],[739,259],[743,183],[769,368],[773,18],[787,245],[809,260],[821,181],[827,308],\ldots\\ \ldots&[839,406],[857,23],[863,224],[877,532],[881,377],[911,892],[929,818],[937,917],[947,8],[967,894],[971,58],[977,275],[983,527],\ldots\\ \ldots&[997,750],[1009,771],[1013,225],[1021,501],[1031,388],[1069,40],[1093,588],[1097,154],[1117,671],[1123,176],[1129,193],[1151,452],\ldots\\ \ldots&[1153,110],[1171,907],[1181,1078],[1193,55],[1201,727],[1223,118],[1231,300],[1237,170],[1259,50],[1291,1161],[1297,1108]\}\end{aligned}$$

Der Exponentenvektor $e_S = \sum_{(a,b)\in S} e_{(a,b)}$ lautet

$$\begin{aligned}e_S = [&\underbrace{8,4,4,8,2,4,2,0,2,0,0,0,2,0,0,0}_{\text{RFB}},\underbrace{42,8,4,4,2,2,0,0,2,2,2,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0}_{\text{AFB}},\ \ldots\\ \ldots\ &\underbrace{8}_{\text{Vorzeichen}},\underbrace{6,6,8,8,8,6,8,8,4,4,8,10,8,6,4,4,6,8,8,6,8,6,8,8,4,8,8,6,8,4,4,8,4,4,8,6,4,6,6,8,6,6,6,10,8,6,6,8,8,6,6,6,6,}_{\text{QCB}}\ \ldots\\ &\underbrace{\ldots\ 8,6,10,8,4,8,8,4,8,8,8,8,4,10,10,8,10,8,8,6,10,6,6,8,4,4,4,8,6,6,8,8,10,4,8,8,6,4,10,8,8,6,8,10,4,4,6,}_{\text{QCB}}\ \underbrace{14}_{\text{1er}}\ ]\end{aligned}$$

Die rationale Wurzel W_1 ergibt mit $f'(x) = 3\,x^2 + 2\,x + 43$:

$$\begin{aligned}W_1 &= \Big(3\cdot 56^2 + 2\cdot 56 + 43\Big)\cdot\Big(2^{\frac{8}{2}}\cdot 3^{\frac{4}{2}}\cdot 5^{\frac{4}{2}}\cdot 7^{\frac{8}{2}}\cdot 11^{\frac{2}{2}}\cdot 13^{\frac{4}{2}}\cdot 17^{\frac{2}{2}}\cdot 23^{\frac{2}{2}}\cdot 41^{\frac{2}{2}}\Big) \;=\\ &= 9563\cdot\Big(2^4\cdot 3^2\cdot 5^2\cdot 7^4\cdot 11^1\cdot 13^2\cdot 17^1\cdot 23^1\cdot 41^1\Big) \;=\\ &= 9563\cdot\Big(16\cdot 9\cdot 25\cdot 2401\cdot 11\cdot 169\cdot 17\cdot 23\cdot 41\Big) \;\equiv\; 9563\cdot 18230 \;\equiv\; 31596 \;=\varphi_1(W_1) \qquad \bmod 181187\,.\end{aligned}$$

Somit können wir in $\mathbb{Z}_N$ die Summe

$$\varphi_1(W_1) + \varphi_2(W_2) \;=\; 31596 + 39570 \;=\; 71166 \qquad \bmod 181187$$

und die Differenz

$$\varphi_1(W_1) - \varphi_2(W_2) \;=\; 31596 - 39570 \;=\; -7974 \;\equiv\; 173213 \qquad \text{mod } 181187$$

bilden und beide liefern uns jeweils einen nicht-trivialen Teiler von N:

$$ggT(71166, 181187) \;=\; 409$$

und

$$ggT(-7974, 181187) \;=\; ggT(173213, 181187) \;=\; 443$$

womit wir $N = 181187 = 409 \cdot 443$ erfolgreich faktorisiert haben.

Kapitel 4

ElGamal

Das ElGamal-Verschlüsselungsverfahren stellt bei geeigneter zugrundeliegender Gruppe die beste Alternative zu RSA dar [KK10, S. 219] und „basiert“ auf dem Problem des Diskreten Logarithmus'. Dh. bisher ist zum Brechen von ElGamal kein anderer Weg als das Berechnen des diskreten Logarithmus' bekannt. Und für das Problem des diskreten Logarithmus' ist bisher auch noch kein (allgemein anwendbarer) Algorithmus bekannt - mit Ausnahme für spezielle Gruppen wie zB. $\mathbb{Z}_p{}^*$.

4.1 Zahlentheoretische Grundlagen von ElGamal

Der mathematische Hintergrund von ElGamal ist in vielen Büchern und Arbeiten beschrieben, zB. in [KK10, S. 167-173, 179], [Buc10, S. 155-162], [Sti06, S. 233-235, 267f], [MvOV97, S. 103f, 113f, 294-298] und [KL08, S. 274-278, 364-369].
Im Folgenden wollen wir die grundlegenden Konzepte für ElGamal zusammenfassen:

4.1.1 Diskreter Logarithmus

Definition 4.1 (DL)
Sei $(G,\cdot)$ eine (multiplikativ geschriebene) Gruppe der Ordnung $|G| = n$ und sei $\gamma \in G$ ein erzeugendes Element von G, dh.

$$\langle\gamma\rangle = G\,.$$

Für ein Element $\alpha \in G$ heißt die kleinste Zahl $m \in \mathbb{N}$ mit

$$\gamma^m = \alpha$$

der *diskrete Logarithmus (DL)* von α zur Basis γ - in Zeichen:

$$m = \log_\gamma \alpha\,.$$

Definition 4.2 (DLP)
Das Problem, für ein gegebenes Element $\alpha \in G$ und einen gegebenen Erzeuger γ von G den diskreten Logarithmus von α zur Basis γ zu berechnen, heißt das *diskrete Logarithmen-Problem (DLP)*.

4.1.2 Diffie-Hellman-Schlüsselaustausch

Der Schlüsselaustausch nach Diffie-Hellman bietet die Möglichkeit, dass sich zwei Kommunikationspartner über einen unsicheren Kanal einen gemeinsamen, geheimen Schlüssel für eine

sichere Kommunikation ausmachen können. Dazu muss man voraussetzen, dass in der gewählten, endlichen zyklischen Gruppe $G = \langle\gamma\rangle$ das diskrete Logarithmen-Problem schwierig zu lösen ist [KK10, S. 179].
Wollen wir uns anhand von Alice und Bob anschauen, wie dieses Schlüsselaustauschprotokoll funktioniert:

- Alice und Bob einigen sich auf eine Gruppe G der Ordnung n und ein erzeugendes Element γ von G. Diese Informationen über G und γ dürfen öffentlich bekannt sein und stellen (zumindest zur Zeit) kein Sicherheitsrisiko dar.

- Nach der Festlegung von (G, γ) wählt sich Alice ein zufälliges $a \in \mathbb{N}$ mit $a \in \{2, \ldots, n-2\}$ und berechnet sich
$$A := \gamma^a .$$
a stellt den geheimen Schlüssel von Alice dar und darf nicht publik werden.
Analog wählt Bob ein zufälliges $b \in \{2, \ldots, n-1\} \subseteq \mathbb{N}$, welches er geheim halten muss und berechnet
$$B := \gamma^b .$$

- Alice sendet an Bob über die unsichere Leitung nur das Element A, nicht aber ihre geheime Information a.
Analog sendet Bob das Ergebnis B an Alice, nicht aber b.

- Alice empfängt von Bob B und berechnet sich den Schlüssel $K_1 := B^a$. Bob berechnet sich $K_2 := A^b$ und tatsächlich gilt
$$K_1 = B^a = \left(\gamma^b\right)^a = \gamma^{b \cdot a} = \gamma^{a \cdot b} = \left(\gamma^a\right)^b = A^b = K_2 ,$$
womit gezeigt wäre, dass dieses System tatsächlich zum Vereinbaren eines gemeinsamen Schlüssels verwendet werden kann.

Ein passiver Angreifer, der die Kommunikation während der Phase der Schlüsselerzeugung belauscht, hat zwar die Informationen über die Gruppe G, deren Erzeuger γ und die beiden Werte A und B, kann aber mit den momentan bekannten Mitteln nicht auf das Element $\gamma^{a\,b}$ schließen.

Definition 4.3 (DHP)
Das Problem, aus der bloßen Kenntnis von $A = \gamma^a$ und $B = \gamma^b$ (ohne die Werte a und b zu kennen) den gemeinsamen Schlüssel $K := \gamma^{a \cdot b}$ zu berechnen, nennt man das *Diffie-Hellman-Problem (DHP)*.

Lemma 4.4 (DL vs. DHP)
Wenn man effizient diskrete Logarithmen berechnen kann, dann kann man auch das Diffie-Hellman-Problem effizient lösen, denn ein Lauscher hört die Informationen (G, γ, A, B) ab, berechnet sich den diskreten Logarithmus a von A zur Basis γ sowie den DL b von B (zur Basis γ) und kann sich mit den Werten a und b deren Produkt $a \cdot b$ und schlussendlich mittels eines square & multiply-Algorithmus' die Potenz $\gamma^{a \cdot b}$ berechnen, womit er den privaten, gemeinsamen Schlüssel von Alice und Bob bestimmt hat.
Dh. der DH-Schlüsselaustausch ist deshalb ein sicheres Verfahren, weil bislang noch kein effizienter und allgemein anwendbarer Algorithmus zur Lösung des DLP bekannt ist. (Zur Lösung des DLP in speziellen Gruppen gibt es spezielle Algorithmen, die zwar etwas schneller sind als allgemeine Algorithmen, dafür aber eben auch nur in diesen Gruppen funktionieren.) Es ist aber auch noch unbekannt, ob man, um das DHP zu lösen, überhaupt diskrete Logarithmen berechnen können muss, oder ob nicht vielleicht jemand einen anderen Weg findet, der ohne DL auskommt [KK10, S. 169].

input : Basis der Potenz: g
Exponent e in Binärdarstellung $e = (e_r\, e_{r-1} \ldots e_1\, e_0)_2$
output : Potenz $A = g^e$

$A := 1$
for $i = r, \ldots, 0$ **do**
  $A := A^2$
  if $e_i = 1$ **then**
    $A := A \cdot g$
  end
end
return A

Algorithmus 4.5: schnelles Potenzieren - left-to-right

4.1.3 Schnelles Potenzieren - Abwehr von side-channel-Attacken

Ebenso wie für das RSA-Verfahren muss man auch für ElGamal schnell große Potenzen eines beliebigen Elements berechnen können. Dazu können wir die schon in Kap. 3.1.2 und 3.1.3 vorgestellte Vorgehensweise verwenden. Das dort beschriebene Vorgehen ist ein sog. *right-to-left-Algorithmus*, da dabei die Binärdarstellung des Exponenten von rechts nach links gelesen den Ablauf mitbestimmt. (Bei einem 1er wird die optionale Multiplikation ausgeführt, bei einem 0er nicht.)

Algorithmus 4.5 stellt einen square & multiply-Algorithmus aus der Kategorie der left-to-right-Algorithmen dar [MvOV97, Sl. 614-617].

Der in Kap. 3.3.10.2 beschriebene power-consumption-Angriff, der die Binärdarstellung des Exponenten rekonstruiert, greift auch bei dem soeben beschriebenen Algorithmus 4.5, da die Strukur gleich ist: Wir quadrieren auf jeden Fall und multiplizieren eventuell, abhängig vom aktuellen Bit im Exponenten.

Auf Algorithmus 4.5 aufbauend wollen wir alternative Methoden zur schnellen Berechnung von Potenzen vorstellen.[1]

4.1.3.1 Window-Methode

Um dem Angriff von Kap. 3.3.10.2 zu entgehen, kann man die Binärdarstellung des Exponenten (von rechts beginnend) in Blöcke von jeweils k Bits einteilen. Oder anders gesagt, wir stellen den Exponenten in der Basis $b := 2^k$ mit Koeffizienten f_i im Bereich von $0 \le f_i \le 2^k - 1$ dar:

$$e \;=\; \Big(e_r \ldots e_6 \underbrace{e_5 e_4 e_3}_{f_1} \underbrace{e_2 e_1 e_0}_{f_0}\Big)_2 \;=\; \big(f_s\, f_{s-1} \ldots f_1\, f_0\big)_b\,.$$

Für jeden (Binär-)Block der Länge k wollen wir die notwendigen Multiplikationen in einem einzigen Schritt durchführen. Dazu müssen wir ein wenig Vorarbeit leisten, deren Ergebnisse wir aber für verschiedene Exponenten immer wiederverwenden können. Dazu berechnen wir vorab in einer Schleife die Werte

$$g_0 := 1, \qquad \text{und} \qquad g_i := g_{i-1} \cdot g \qquad \forall\; i = 1, \ldots, 2^k - 1\,,$$

für die dann die Gleichung $g_i = g^i \;\; \forall\, 0 \le i \le 2^k - 1$ gilt. Die Berechnung der Potenz läuft daraufhin nach Algorithmus 4.6 ab.

[1] Meiner Meinung nach sind aber nicht alle folgenden Methoden gleichermaßen dazu geeignet, sog. side-channel-Attacken (sowohl auf RSA, als auch auf ElGamal) zur Rekonstruktion des geheimen Exponenten abzuwehren.

input : Potenzen $g_i := g^i \quad \forall\, i = 0, \ldots, 2^k - 1$
Exponent $e = (f_s f_{s-1} \ldots f_1 f_0)_b$ in der Darstellung zur Basis $b := 2^k$
output : Potenz $A = g^e$

$A := 1$
for $i = s, \ldots, 0$ **do**
  $A := A^{2^k}$
  if $f_i \neq 0$ **then**
    $A := A \cdot g_{f_i}$
  end
end
return A

Algorithmus 4.6: schnelles Potenzieren - window-Methode

input : Potenzen $g_i := g^i \quad \forall\, i = 1, 3, 5, \ldots, 2^k - 1$
Exponent $e = (f_s f_{s-1} \ldots f_1 f_0)_b$ in der Darstellung zur Basis $b := 2^k$
output : Potenz $A = g^e$

$A := 1$
for $i = s, \ldots, 0$ **do**
  $f_i = 2^{t_i} \cdot u_i$; `// zerlege` f_i `in geraden und ungeraden Anteil`
  $A := A^{2^{k-t_i}}$
  $A := A \cdot g_{u_i}$
  $A := A^{2^{t_i}}$
end
return A

Algorithmus 4.7: schnelles Potenzieren - modifizierte window-Methode

Die Wahrscheinlichkeit, dass bei der window-Variante des square & multiply-Algorithmus' die optionale Multiplikation nicht ausgeführt wird, liegt für einen zufälligen Exponenten f_i nur mehr bei $\frac{1}{2^k}$. Da bei entsprechender Wahl von k nun „fast immer" eine Multiplikation durchgeführt wird, kann man mit der power-consumtion-Attacke nur mehr jene Binärblöcke der Länge k erkennen, die ausschließlich aus Nullen bestehen.

4.1.3.2 Modifizierte window-Methode

Man kann Algorithmus 4.6 noch weiter verbessern, indem man ihn ein wenig abändert, sodass beim einmaligen Vorbereitungsschritt nur jene g_i berechnet werden müssen, die einen ungeraden Index i besitzen. Dazu berechnet man die Werte

$$g_0 := 1, \quad g_1 := g, \quad g_2 := g^2, \quad g_{2i+1} := g_{2i-1} \cdot g_2 \qquad \forall\ i = 1, \ldots, 2^{k-1} - 1\,.$$

Weiters zerlegen wir (mit den bisherigen Bezeichnungen) ein $f_i \in \{0, \ldots, 2^k - 1\}$ in seinen geraden und seinen ungeraden Anteil

$$f_i \;=\; 2^{t_i} \cdot u_i \qquad \text{mit } t_i \text{ maximal und } u_i \text{ ungerade}\,.$$

Diese Zerlegung nützen wir aus, indem wir für ein konkretes f_i zuerst so oft quadrieren, als der ungerade Anteil Binärstellen hat, nämlich $k - t_i$ Mal. Dann Multiplizieren wir g_{u_i} darauf und anschließend führen wir die restlichen t_i Quadrierungen durch. Dadurch erhalten wir den Algorithmus 4.7 .

input : Potenzen $g_i := g^i \quad \forall\, i = 1, 3, 5, \ldots, 2^k - 1$
Exponent $e = (e_r\, e_{r-1} \ldots e_1\, e_0)_2$ in Binärdarstellung
output : Potenz $A = g^e$

$A := 1$
$i := r$
while $i \geq 0$ **do**
  if $e_i = 0$ **then**
    $A := A^2$
    $i := i - 1$
  else
    // finde längsten Block $e_i\, e_{i-1} \ldots e_l$ mit $i - l + 1 \leq k$ und $e_l = 1$
    $A := A^{2^{i-l+1}}$
    $A := A \cdot g_{e_i\, e_{i-1} \ldots e_l}$
    $i := l - 1$
  end
end
return A

Algorithmus 4.8: schnellen Potenzieren - sliding-window-Methode

Zwar erspart man sich durch diese Modifikation die Hälfte der notwendigen Vorausberechnungen, jedoch birgt dieser Algorithmus meiner Meinung nach die Gefahr in sich, dass ein Angreifer, der zwischen dem Quadrieren und einer allgemeinen Multiplikation unterscheiden kann, dadurch von jedem Binärblock der Länge k den Wert modulo 2^{t_i+1} kennt, nämlich $f_i \equiv 2^{t_i} \mod 2^{t_i+1}$. Dh. ein Angreifer würde dann vom Exponenten in jedem Binärblock der Länge k die Länge der hintersten Nullfolge kennen, oder erkennen, dass f_i ungerade ist.

4.1.3.3 Sliding-window-Methode

Bei der sliding-window-Methode wird (mit den bisherigen Bezeichnungen) die Binärdarstellung des Exponenten nicht mehr in Blöcke der fixen Länge k unterteilt, sondern die Länge der Blöcke ist maximal k und wird durch Blöcke von Nullen beliebiger Länge getrennt.

Wiederum berechnet man sich im Vorhinen - genauso wie zuvor in Kap. 4.1.3.2 - die Werte für g_i mit ungeradem Index i. Die Abarbeitung eines Exponenten $e = (e_r\, e_{r-1} \ldots e_1\, e_0)_2$ in Binärdarstellung läuft wie folgt: Angenommen der Exponent wurde schon von links bis zum Bit e_{i+1} abgearbeitet.

- Wenn $e_i = 1$ ist, dann sucht man von links beginnend den längsten Block $e_i\, e_{i-1} \ldots e_l$ mit $e_l = 1$ und $i - l + 1 \leq k$, dh. der ausgewählte Block $e_i\, e_{i-1} \ldots e_l$ soll Länge höchstens k haben. Entsprechend der Blocklänge wird A dann $i-l+1$ Mal quadriert und anschließend mit $g_{e_i\, e_{i-1} \ldots e_l}$ multipliziert, was nach unserer Konstruktion ungerade sein muss. Zuletzt setzen wir i noch auf $i := l - 1$.

- Wenn $e_i = 0$ ist, dann quadrieren wir A einfach und setzen $i := i - 1$

Algorithmus 4.8 fasst dieses Vorgehen zusammen.

4.1.3.4 NAF-Darstellung des Exponenten

Ähnlich der Binärdarstellung $n = \sum_{i=0}^{r} a_i 2^i$ mit $a_i \in \{0, 1\}$ ist die *NAF-Darstellung* einer natürlichen Zahl n jene eindeutige Darstellung $n = \sum_{i=0}^{s} b_i 2^i$ mit $b_i \in \{-1, 0, 1\}$ und dem Zusatz, dass von je zwei benachbarten Zeichen in $\left(b_s\, b_{s-1} \ldots, b_1\, b_0\right)$ mindestens eine gleich 0 sein muss. Dh. es gilt

$$b_i \cdot b_{i-1} \;=\; 0 \qquad \forall\, i = 1, \ldots, s\,.$$

Von dieser Eigenschaft, dass nie zwei Einträge ungleich 0 nebeneinander stehen, leitet sich auch die Bezeichnung NAF-Darstellung (**N**on-**A**djacent-**F**orm) ab, manchmal auch *signed-digit-Darstellung* genannt, vgl. [MvOV97, Kap. 14.7.1, S. 627f].

Wenn man in der gegebenen Gruppe G schnell das additiv Inverse eines Elements berechnen kann, so kann man die NAF-Darstellung auch in einem square&multiply-Algorithmus einsetzen. Diese Variante hat gegenüber der Verwendung der Binärdarstellung zwei Vorteile:

- Im Gegensatz zur Binärdarstellung, bei der durchschnittlich die Hälfte der Binärzeichen a_i eines zufällig gewählten $n = \left(a_r\, a_{r-1} \ldots a_1\, a_0\right)_2$ Ziffern ungleich 0 sind, sind bei der NAF-Darstellung durchschnittlich nur $\frac{1}{3}$ der Ziffern b_i ungleich 0. Dadurch müssen beim square & multiply-Algorithmus weniger optionale Multiplikationen ausgeführt werden, was Zeit- bzw. Rechenersparnis mit sich bringt.
- Zusätzlich kann ein Angreifer bei der Power-consumption-Attacke alleine vom Erkennen des Auftretens einer optionalen Multiplikation nicht auf das Vorzeichen des zugehörigen Zeichens b_i schließen.

In [MvOV97] sind zwei weitere Methoden angeführt, die im Zusammenhang mit der Berechnung modularer Potenzen in Zusammenhang stehen:

- *Montgomery-Potenzbildung*
  In [MvOV97, S.619f] wird ein Algorithmus zur Berechnung einer modularen Potenz angegeben, der den klassischen Reduktionsschritt vermeidet bzw. umgeht. Grundlage dafür bildet die *Montgomery-Reduktion*, vgl. [MvOV97, Kap. 14.3.2, S. 600-603].
- Die *Barrett-Reduktion* [MvOV97, Kap. 14.3.3 S.603f] berechnet von einer gegebenen Zahl x die Restklasse $r \equiv x \mod n$, in der sie liegt, ebenfalls ohne den klassischen Reduktionsalgorithmus anzuwenden.

4.2 Funktionsweise von ElGamal

Das ElGamal-Verschlüsselungssystem beruht auf dem Diffie-Hellman-Problem. Wollen wir uns zuerst die Funktionsweise von ElGamal ansehen und anschließend den Zusammenhang mit dem DHP.

Dazu nehmen wir wieder an, dass Bob an Alice eine verschlüsselte Nachricht senden möchte:

- Schlüsselerzeugung:
  - Alice wählt eine endliche zyklische Gruppe G mit $n = |G|$ Elementen und einem Erzeuger γ von G.

  - Als geheimen Schlüssel wählt Alice ein $a \in \mathbb{N}$ mit $2 \leq a \leq n-1$ und berechnet sich damit den Wert $A := \gamma^a$.
  - Alice macht als öffentlichen Schlüssel das Trippel (G, γ, A) publik.

- Verschlüsseln:

  - Bob, der eine Nachricht senden möchte, besorgt sich Alice' öffentlichen Schlüssel und stellt die Nachricht als Element $M \in G$ dar.
  - Bob wählt ein zufälliges $b \in \{2, \ldots, n-1\}$, das nur für die Übertragung dieser einen (!) Nachricht M verwendet werden darf.
  - Bob berechnet
    * $B := \gamma^b$,
    * den temporären Sitzungsschlüssel $K_E := A^b$
    * und „maskiert" damit die Nachricht M:

$$c \;:=\; K_E \cdot M\,.$$

  - Als Chiffrat übermittelt Bob an Alice das Paar

$$C := (B, c)\,.$$

- Entschlüsseln:

  - Alice empfängt von Bob $C = (B, c)$ und berechnet sich $K_D := B^{n-a}$.
  - Damit berechnet sie sich

$$\widetilde{M} := K_D \cdot c\,.$$

  - Tatsächlich gilt $\widetilde{M} = M$, womit sie Bobs ursprüngliche Nachricht M lesen kann.

4.2.1 Korrektheit des Verfahrens

Wollen wir uns kurz überlegen, dass das soeben beschriebene Verfahren tatsächlich wieder den ursprünglichen Klartext M hervorbringt:

Als Chiffrat sendet Bob

$$C \;=\; (B, c) \;=\; \left(\gamma^b, A^b \cdot M\right) \;=\; \left(\gamma^b, (\gamma^a)^b \cdot M\right) \;=\; \left(\gamma^b, \gamma^{a \cdot b} \cdot M\right)$$

an Alice, welche sich daraus als Erstes den Schlüssel

$$K_D \;=\; B^{n-a} \;=\; B^n \cdot B^{-a} \;=\; 1 \cdot B^{-a} \;=\; \left(\gamma^b\right)^{-a} \;=\; \gamma^{-ba}$$

und in weiterer Folge

$$\widetilde{M} \;=\; K_D \cdot c \;=\; \left(\gamma^{-ba}\right) \cdot \left(\gamma^{a \cdot b} \cdot M\right) \;=\; M$$

berechnet, womit die korrekte Dechiffrierung gezeigt ist.

4.2.2 ElGamal-Problem

Das Problem, aus den Werten $\left(G, \gamma, A = \gamma^a, B = \gamma^b, c = \gamma^{a\,b} \cdot M\right)$ den Wert M zu berechnen, nennen wir das *ElGamal-Problem (EGP)*.

Satz 4.9 (EGP $\Leftrightarrow$ DHP)
Das ElGamal-Problem ist äquivalent zum Diffie-Hellman-Problem [KK10, S. 172], oder anders formuliert:

1. Wenn wir das Diffie-Hellman-Problem lösen können, dann können wir auch das ElGamal-Problem lösen.
2. Wenn wir das ElGamal-Problem lösen können, dann können wir auch das Diffie-Hellman-Problem lösen.

Beweis

1. Wenn wir das DHP lösen können, dann können wir aus dem Wert $A = \gamma^a$ des öffentlichen Schlüssels und dem abgehörten Wert $B = \gamma^b$ den Wert $\gamma^{a\,b}$ berechnen. Durch Berechnung der Inversen von $\gamma^{a\,b}$ können wir
$$\gamma^{-a\,b} \cdot c \;=\; \gamma^{-a\,b} \cdot \left(\gamma^{a\,b} \cdot M\right) \;=\; M$$
bilden, womit wir das EGP gelöst haben.
2. Wenn wir das EGP lösen können, dann können wir uns den Wert M berechnen und folglich auch den Wert M^{-1}. Dies können wir mit c multiplizieren womit wir
$$c \cdot M^{-1} \;=\; \left(\gamma^{a\,b} \cdot M\right) \cdot M^{-1} \;=\; \gamma^{a\,b}$$
erhalten und somit das DHP gelöst haben.

□

4.2.3 Günstige Gruppen

Ein Vorteil von ElGamal ist, dass sich dieses Kryptosystem auf alle zyklischen Gruppen verallgemeinern lässt, in denen die Gruppenoperation effizient berechenbar ist (dh. man kann die Vorbereitungen und die Durchführung einer verschlüsselten Nachrichtenübertragung schnell ausführen), aber das Lösen des Diffie-Hellman-Problems bzw. des ElGamal-Problems schwierig ist. Insbesondere bedeutet dies, dass man keine effiziente Berechnung des diskreten Logarithmus' in dieser Gruppen kennen darf [Buc10, S. 158 bzw. 161f].

Ein Beispiel für eine *ungünstige* Gruppe ist $(\mathbb{Z}_n, +)$, weil man dort mittels des Erweiterten Euklidischen Algorithmus' das DLP effizient lösen kann [KK10, Kap. 9.1.4, S. 170f].

[MvOV97, Kap. 8.4.2, S. 297] gibt eine Übersicht über jene Gruppen, die für ElGamal verwendet werden können. Der wohl bekannteste Vertreter ist $\mathbb{Z}_p{}^* = \mathbb{Z}_p \backslash \{0\}$ als Spezialfall von $GF(p^m)^*$.

4.2.3.1 $\mathbb{Z}_p{}^*$

Bei der Verwendung von $G = \mathbb{Z}_p^*$ als zugrundeliegende Gruppe für das ElGamal-Verschlüsselungssystem sollte p zur Zeit ebenfalls mindestens 2048 Bit lang sein[2] (vgl. auch Kap. 3.2.2) und $p - 1$ sollte einen „großen" Primfaktor besitzen [Sti06, S. 235].

[2] Stand 20. Mai 2011:
http://www.bundesnetzagentur.de/DE/Sachgebiete/QES/Veroeffentlichungen/Algorithmen/algorithmen_node.html

4.2.3.2 $GF(2^m)^*$

Auch der Fall, dass die dem ElGamal-Verfahren zugrundeliegende Gruppe $G = GF(2^m)^*$ ist, wird häufig verwendet und ist genauso, wie der vorige Abschnitt ($G = \mathbb{Z}_p^*$), ein Spezialfall von $G = GF(p^m)^*$. Es gibt eine Attacke, die speziell für diese Gruppe optimiert ist - vgl. S. 90.

4.2.3.3 Weitere mögliche Gruppen

1. elliptische Kurven über endlichen Körpern
2. Jakobische Varietät hyperelliptischer Kurven über endlichen Körpern
3. Klassengruppen imaginär-quadratischer Ordnungen

4.3 Attacken auf ElGamal

Die meisten Attacken auf das ElGamal-Kryptosystem versuchen, dieses durch Berechnen des diskreten Logarithmus' zu brechen. Wir werden uns auf allgemeine Methoden zum Berechnen des DL konzentrieren, die in *jeder* Gruppe funktionieren.

4.3.1 Allgemeine Attacken für DLP

4.3.1.1 Enumeration

Der naive Ansatz, um den diskreten Logarithmus m eines Elementes α in einer zyklischen Gruppe G der Ordnung n zu berechnen, besteht darin, einfach der Reihe nach alle Potenzen

$$\gamma^i \qquad \forall i \in \{0, \ldots, n-1\}$$

eines Erzeugers γ von G zu berechnen und zu vergleichen, ob $\gamma^i = \alpha$ ist. Falls ja, dann ist das aktuelle i der diskrete Logarithmus von α zur Basis γ. Da dieses Vorgehen eine Laufzeit von $\mathcal{O}(|G|)$ hat, ist es nur für kleine Gruppengrößen anwendbar und im Allgemeinen nicht effizient [KK10, Kap. 10.1.2, S. 180].

4.3.1.2 Baby-Step-Giant-Step-Angriff von Shanks

Das Prinzip hinter der Baby-Step-Gigant-Step-Methode von Shanks kann man mit folgenden zwei Ideen beschreiben:

- Wenn man auf einem Schachbrett die Menge M_1 als die Felder in einer beliebigen Reihe des Schachbretts definiert und die Menge M_2 als die Felder einer beliebigen Spalte, dann gibt es genau ein Feld am Schachbrett, das in der Schnittmenge $M_1 \cap M_2$ liegt.
- Und wenn man die Zahlen von 1 bis n der Reihe nach in ein quadratisches Schachbrett mit Seitenlänge $k \geq \sqrt{n}$ legt, so kann man aus der Angabe der Zeile i und der Spalte j, in der sich eine Zahl $a \in \{1, \ldots, n\}$ befindet, durch

  $$a = (i-1) \cdot k + j$$

  sofort auf die Zahl a selbst schließen.

Für den Baby-Step-Gigant-Step-Angriff [KK10, Kap. 10.2, S. 180f], [Buc10, Kap. 11.3, S.178-180] stellen wir uns also die Elemente der Gruppe $G = \langle\gamma\rangle$ als Potenzen des Erzeugers γ vor und ordnen die $n = |G|$ Gruppenelemente der Reihe nach (dh. nach aufsteigenden Exponenten bezüglich γ) zeilenweise wie auf einem Schachbrett mit Seitenlänge $k := \lceil\sqrt{n}\rceil$ an, beginnend mit γ. Sei weiters $\alpha := \gamma^a$ für $1 \le a \le n$ ein gegebenes Element, von dem wir uns den diskreten Logarithmus zur Basis γ ausrechnen wollen:

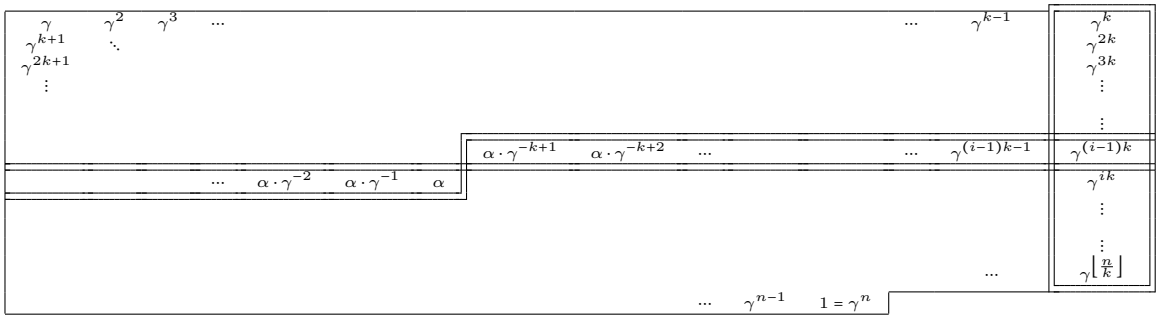
γ	γ^2	γ^3	γ^4	...					...	γ^{k-2}	γ^{k-1}	γ^k
γ^{k+1}	γ^{k+2}	γ^{k+3}									γ^{2k-1}	γ^{2k}
γ^{2k+1}	γ^{2k+2}											γ^{3k}
γ^{3k+1}												$\vdots$
$\vdots$												
					$\alpha = \gamma^a$							
												$\vdots$
$\vdots$											...	$\gamma^{\lfloor\frac{n}{k}\rfloor}$
$\gamma^{\lfloor\frac{n}{k}\rfloor+1}$	...					...	γ^{n-1}	$1 = \gamma^n$				

Als Spalte für die Menge M_2 wollen wir uns die letzte Spalte wählen, dh.

$$M_2 := \left\{\gamma^k, \gamma^{2\,k}, \gamma^{3\,k}, \ldots, \gamma^{\lfloor\frac{n}{k}\rfloor}\right\}. \tag{4.1}$$

Da wir nicht wissen, an welcher Stelle sich unser gegebenes Element α im Schachbrettquadrat befindet, wählen wir als „Zeile“ einfach jene k direkt aufeinander folgenden Elemente, deren letztes Element α ist:

$$M_1 := \left\{\alpha\cdot\gamma^{-k+1},\ \alpha\cdot\gamma^{-k+2},\ \ldots,\ \alpha\cdot\gamma^{-2},\ \alpha\cdot\gamma^{-1},\ \alpha\right\}.$$



Wie schon zuvor gezeigt muss es ein Element im Schnitt von M_1 und M_2 geben, nämlich

$$\gamma^{(i-1)\,k} \in \left(M_1 \cap M_2\right).$$

Der Vorteil dieser Vorgehensweise im Vergleich zur Enumeration besteht darin, dass man nicht (bis zu) n Gruppenelemente berechnen muss, sondern man braucht nur höchstens $2\cdot\lceil\sqrt{n}\rceil$ Elemente berechnen. Weiters kann man für eine Gruppe G die Spaltenliste M_2 ein für alle Mal vorweg berechnen und die Elemente von M_2 für *jede* DL-Berechnung in *dieser* Gruppe wiederverwenden.

Wollen wir den Ablauf für die Berechnung des diskreten Logarithmus' nach der *Baby-Step-Gigant-Step-Methode* von Shanks zusammenfassen:

- Suchen eines erzeugenden Elementes γ der Gruppe G mit n Elementen, in der der diskrete Logarithmus berechnet werden soll.

- Berechnen von $k := \lceil \sqrt{n} \rceil$
- Bestimmung der Elemente in $M_2 = \left\{\gamma^k, \gamma^{2\,k}, \gamma^{3\,k}, \ldots, \gamma^{\lfloor \frac{n}{k} \rfloor}\right\}$
- Bestimmung der Elemente in $M_1 := \left\{\alpha \cdot \gamma^{-k+1},\ \alpha \cdot \gamma^{-k+2},\ \ldots,\ \alpha \cdot \gamma^{-2},\ \alpha \cdot \gamma^{-1},\ \alpha\right\}$
- Stelle für jedes Element aus M_2 fest, ob es in der Menge M_1 vorkommt.
  - Falls ja, dann gibt es zwei Werte i und j mit $1 \le i, j \le k$ und
    $$\gamma^{i \cdot k} = \alpha \cdot \gamma^{-j}\,.$$
    Wegen $\gamma^{i \cdot k} = \alpha \cdot \gamma^{-j} \Leftrightarrow \gamma^{i\,k+j} = \alpha = \gamma^a$ haben wir deshalb mit
    $$i \cdot k + j = a$$
    den diskreten Logarithmus von α zur Basis γ berechnet.
  - Falls nein, dann versuche das nächste Element von M_2 in M_1 zu finden.

Bemerkung 4.10

- Das Element $\gamma^n = 1$ befindet sich bei der Wahl $k := \lceil \sqrt{n} \rceil$ jedenfalls in der letzten oder in der vorletzten Zeile unseres Schachbretts mit Seitenlänge k. Dh. die möglichen Felder für $\gamma^n = 1$ sind jene, die der nachfolgenden Abbildung in der doppelten Umrahmung liegen:

γ	γ^2	γ^3	$\cdots$		$\cdots$	γ^{k-1}	γ^k
γ^{k+1}	γ^{k+2}						γ^{2k}
γ^{2k+1}		$\ddots$					$\vdots$
$\vdots$			$\ddots$				
				$\ddots$			
					$\ddots$		$\vdots$
						$\ddots$	$\gamma^{(k-1)k}$
					$\cdots$	γ^{k^2-1}	γ^{k^2}

  Daraus folgt, dass $\lfloor \frac{n}{k} \rfloor$ nur die Werte k, $k-1$ oder $k-2$ annehmen kann. Der Fall $\lfloor \frac{n}{k} \rfloor = k$ kann nur dann auftreten, wenn $k = \sqrt{n}$ ist.

- Es gibt auch eine alternative Version der Baby-Step-Gigant-Step-Methode, die zwar die gleiche Idee verwendet, aber bei der die beiden folgenden Mengen verwendet werden [CP00, Kap. 5.3, S. 200-202]:
  - Nur ein einziges Mal im Voraus berechnet wird dabei die Zeile
    $$M_2 := \left\{\gamma, \gamma^2, \gamma^3, \ldots, \gamma^k\right\}.$$
  - Für ein konkretes $\alpha \in G$, von dem man den DL (zur Basis γ) berechnen möchte, werden dann erst sukzessive die Elemente der Menge
    $$M_1 := \left\{\alpha, \gamma^{-k}\alpha, \gamma^{-2k}\alpha, \gamma^{-3k}\alpha, \ldots, \gamma^{-(k-1)k}\alpha\right\}$$
    berechnet und für jedes Element aus M_1 sofort verglichen, ob es in M_2 vorkommt.

- Weiters gibt es noch unwesentliche Abwandlungen [KL08, Kap. 8.2.1, S.307f], [MvOV97, Kap. 3.6.2, S. 104f], bei denen zB. ...
  - ... das erste Element im Schachbrett nicht γ sondern $1 = \gamma^0$ ist,
  - ... k als $k := \lfloor \sqrt{n} \rfloor$ definiert, dh. nicht auf- sondern abgerundet wird,
  - ... die Menge M_1 bei α *beginnt* und die $k-1$ *nachfolgenden* Elemente genommen werden.

4.3.1.3 Pollard'sche ρ-Methode für DLP

Die *Pollard'sche ρ-Methode* für das Problem des diskreten Logarithmus' geht in ihrer Grundversion (genauso wie die Pollard'sche ρ-Methode zur Faktorisierung natürlicher Zahlen) von einer unendlichen Folge von Elementen einer endlichen Gruppe aus und verwendet das Argument, dass in dieser Folge zwei Folgenglieder gleich sein müssen [Buc10, Kap. 11.4, S.180-183].

Dazu unterteilen wir die Menge G in 3 möglichst gleich große Teilmengen G_1, G_2, G_3, für die die beiden Bedingungen

1. $G_i \cap G_j = \varnothing \quad \forall i \neq j$
2. $G_1 \cup G_2 \cup G_3 = G$

erfüllt sein müssen (vgl. Bem. 4.11). Weiters sei γ wiederum ein Erzeuger der Gruppe G mit Ordnung $n := |G|$ und sei $\alpha \in G$ jenes Element, von dem wir den diskreten Logarithmus (zur Basis γ) berechnen wollen.
Dazu wählen wir eine natürliche Zahl x_0 zufällig aus dem Bereich $0 \leq x_0 \leq n-1$ und bestimmen damit (zufällig) ein Element β_0 als erstes Element unserer unendlichen Folge:

$$\beta_0 := \gamma^{x_0}.$$

Mit der Funktion $f : G \mapsto G$, die durch

$$f(\beta) := \begin{cases} \gamma \cdot \beta & \beta \in G_1 \\ \beta^2 & \beta \in G_2 \\ \alpha \cdot \beta & \beta \in G_3 \end{cases}$$

definiert ist, berechnen wir uns mittels $\beta_{i+1} := f(\beta_i)$ die gewünschte, unendliche Folge. Weil wir den Exponenten x_0 vom Startwert $\beta_0 = \gamma^{x_0}$ kennen und das Abbilden mit f gleichsam eine Multiplikation mit passenden Potenzen von α und γ darstellt, können wir von jedem Glied β_i unserer unendlichen Folge eine Darstellung

$$\beta_i = \gamma^{x_i}\alpha^{y_i} \qquad \forall i \geq 0$$

angeben. Mit dieser Darstellung der Folgenglieder lässt sich f schreiben als

$$f(\gamma^x\alpha^y) := \begin{cases} \gamma^{x+1}\alpha^y & \gamma^x\alpha^y \in G_1 \\ \gamma^{2x}\alpha^{2y} & \gamma^x\alpha^y \in G_2 \\ \gamma^x\alpha^{y+1} & \gamma^x\alpha^y \in G_3, \end{cases}$$

womit wir die beiden rekursiven Folgen

$$x_{i+1} := \begin{cases} x_i + 1 & \beta \in G_1 \\ 2\,x_i & \beta \in G_2 \\ x_i & \beta \in G_3 \end{cases} \qquad \text{und} \qquad y_{i+1} := \begin{cases} y_i & \beta \in G_1 \\ 2\,y_i & \beta \in G_2 \\ y_i + 1 & \beta \in G_3 \end{cases}$$

erhalten. Wie schon festgestellt müssen zwei Folgenglieder β_i und β_{i+k} mit $0 \leq i < k$ existieren, für die

$$\beta_i = \beta_{i+k}$$

gilt. Wegen

$$\beta_i = \beta_{i+k} \quad \Leftrightarrow \quad \gamma^{x_i}\alpha^{y_i} = \gamma^{x_{i+k}}\alpha^{y_{i+k}} \quad \Leftrightarrow \quad \gamma^{x_i - x_{i+k}} = \alpha^{y_{i+k} - y_i}$$

und $\alpha = \gamma^a$ gilt für die Exponenten im Allgemeinen zwar keine Gleichheit, aber zumindest die folgende Kongruenz:

$$x_i - x_{i+k} \equiv a \cdot (y_{i+k} - y_i) \mod n. \tag{4.2}$$

Der diskrete Logarithmus a von α zur Basis γ muss somit in der Lösungsmenge aller möglichen a, die die Kongruenz (4.2) erfüllen, liegen.

Bemerkung 4.11
Die Funktion f ist so gewählt, dass die Folge $\left(\beta_i\right)_{i\in\mathbb{N}}$ einer Zufallsfolge aus G sehr ähnlich ist. Deshalb muss man bei der Partitionierung von G beachten, dass das Element $1 \notin G_2$ liegt, da sonst gelten würde: $\beta_{i_0} = 1 \;\Rightarrow\; \beta_i = 1 \;\; \forall i \geq i_0$. [Sti06, Kap. 6.2.2, S. 239]

Bemerkung 4.12 (2 Verbesserungen)

- Bei der bis jetzt beschriebenen Vorgehensweise müssten wir alle berechneten Folgenglieder β_i abspeichern und ein neu berechnetes β_{i+1} mit all ihren Vorgängern vergleichen. Um sich Speicherplatz und Rechenzeit zu sparen, halten wir immer nur ein Element β_j im Speicher und vergleichen alle β_i für $j < i \leq 2j$ immer nur mit β_j. Sollten wir bis $i = 2j$ keine Übereinstimmung finden, so verwerfen wir β_j und speichern uns β_{2j} als neues Element zum Vergleichen [Buc10, S. 181f].

- Eine andere Verbesserung schlägt vor, dass man die Paare (β_i, β_{2i}) für $i \geq 1$ berechnet und prüft, ob $\beta_i = \beta_{2i}$ ist. Fall ja, dann hat man die gewünschte Übereinstimmung gefunden, und falls nicht, dann berechnet man sich aus (β_i, β_{2i}) die Werte $(\beta_{i+1}, \beta_{2i+2})$ durch
$$(\beta_{i+1}\,,\;\beta_{2i+2}) \quad := \quad \Big(f(\beta_i)\,,\; f\big(f(\beta_{2i})\big)\Big)$$

- Bei beiden Verbesserungsvorschlägen können wir mit der angenommenen Zufälligkeit der Folgenglieder das Geburtstagsparadoxon zur Laufzeitbestimmung heranziehen und erhalten für die adaptierten Versionen eine Laufzeit von $\mathcal{O}(\sqrt{n})$ und einen Speicheraufwand von $\mathcal{O}(1)$ [Buc10, S. 183].

Wollen wir noch kurz näher darauf eingehen, wie wir die Lösungen der Kongruenz (4.2) bestimmen können:
Bekannterweise gilt, dass die Kongruenz $b\,x \equiv c \mod n$ genau dann eine Lösung für x besitzt, wenn $ggT(b,n)\big|c$ gilt. In diesem Fall sei $d := ggT(b,n)$ und t jene ganze Zahl mit $d\,t = c$. Dann liefert der Erweiterte Euklidische Algorithmus zwei Zahlen $r, s \in \mathbb{Z}$ mit $b\,r + n\,s = d$. Diese Gleichung mulitpliziert mit t liefert die Gleichung $b\,(r\,t) + n\,t\,s = \underbrace{d\,t}_{=c}$ womit wir mit

$$x \;:=\; r\,t \mod n$$

eine Lösung von $b\,x \equiv c \mod n$ gefunden haben. *Alle* Lösungen werden dann genau durch die Menge

$$\left\{\left(x + m\,\frac{n}{d}\right) \mod n \;\middle|\; m \in \mathbb{Z}\right\} \;=\; \left\{\left(x + m\,\frac{n}{d}\right) \mod n \;\middle|\; 0 \leq m < d\right\}$$

repräsentiert [KK10, Kap. 10.3, S. 183].

Algorithmen und Beispiele finden sich sowohl in den schon genannten Quellen als auch in [MvOV97, S. 106f] und [CP00, Kap. 5.2.2, S. 197f].

4.3.1.4 Pohlig-Hellman-Algorithmus

Den *Pohlig-Hellman-Algorithmus* (manchmal auch *Silver-Pohlig-Hellman-Algorithmus* genannt) kann man verwenden, wenn man von der Gruppe G, in der der diskrete Logarithmus berechnet werden soll, eine Faktorisierung der Gruppenordnung $n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$ kennt. Diese Faktorisierung muss nicht unbedingt vollständig sein, um das DLP in G zu vereinfachen [KL08, S. 306]. Dabei reduzieren wir das Problem des DL in einer Gruppe mit Ordnung n auf r DL-Probleme in Gruppen mit $p_i^{e_i}$ Elementen.

Angenommen wir wollen in der Gruppe $G = \langle\gamma\rangle$ vom Element

$$\alpha = \gamma^a$$

den diskreten Logarithmus $a = \log_\gamma(\alpha)$ berechnen. Wenn die Gruppenordnung $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ ist, definieren wir uns für jeden Primteiler $p_i \,|\, n$ die beiden Werte[3]

$$n_p := \frac{n}{p_i^{e_i}} \qquad \text{und} \qquad \gamma_p := \gamma^{n_p} \qquad \forall\, p \,|\, n \text{ mit } p \in \mathbb{P}\,.$$

Da $\mathrm{ord}(\gamma) = n$ ist, hat das Element $\gamma_p = \gamma^{n_p}$ die Ordnung

$$\mathrm{ord}\,(\gamma_p) = p_i^{e_i}\,.$$

Wegen ${\gamma_p}^a = (\gamma^{n_p})^a = (\gamma^a)^{n_p} = \alpha^{n_p}$ liegt das Element α^{n_p} in der von γ_p erzeugten Untergruppe mit $p_i^{e_i}$ Elementen und der folgende Satz liefert die Grundlage für unser weiteres Vorgehen:

Satz 4.13 (Reduktion auf Primpotenzordnung)
Mit den bisherigen Bezeichnungen lässt sich der diskrete Logarithmus a von $\alpha = \gamma^a$ wie folgt berechnen:
Sei a_p definiert als der diskrete Logarithmus von α^{n_p} zur Basis γ_p in der Untergruppe $\langle\gamma_p\rangle$ mit $p_i^{e_i}$ Elementen. Dann lässt sich a berechnen als die (mod n) eindeutige Lösung des Kongruenzensystems

$$\begin{aligned} a &\equiv a_{p_1} \mod p_1^{e_1} \\ a &\equiv a_{p_2} \mod p_2^{e_2} \\ &\vdots \\ a &\equiv a_{p_r} \mod p_r^{e_r}\,. \end{aligned} \tag{4.3}$$

Beweis
Sei $\tilde{a}$ die durch den chinesischen Restsatz erhaltene, eindeutige Lösung des Systems (4.3). Dann gilt es zu zeigen, dass $\gamma^{\tilde{a}} = \alpha$ ist. Wegen der Äquivalenzen:

$$\begin{array}{lrcll} & \gamma^{\tilde{a}} & = & \alpha = & \\ & & = & \gamma^a & \mid \mathrm{ord}(\gamma) = n \\ \iff & \tilde{a} & \equiv & a \mod n & \mid \text{chinesischer Restsatz, allgemein} \\ \iff & \tilde{a} & \equiv & a \mod p_i^{e_i} \quad \forall\, p_i \,|\, n,\ p \in \mathbb{P} & \mid \text{chinesischer Restsatz, (4.3)} \\ \iff & a_p & \equiv & a \mod p_i^{e_i} \quad \forall\, p_i \,|\, n,\ p \in \mathbb{P} & \mid \mathrm{ord}(\gamma_p) = p_i^{e_i} \\ \iff & {\gamma_p}^{a_p} & = & {\gamma_p}^a = & \\ & & = & (\gamma^{n_p})^a = (\gamma^a)^{n_p} = \alpha^{n_p} & \mid \text{Def. von } a_p \end{array}$$

reicht es zu zeigen, dass ${\gamma_p}^{a_p} = \alpha^{n_p}$ ist, was aufgrund der Definition von a_p laut Voraussetzung erfüllt ist.

□

Bei dieser Vorgehensweise ist die Laufzeit vom größten *bekannten* Teiler von n abhängig.
In einem zweiten Schritt kann man ein einzelnes DLP mit Gruppenordnung $p_i^{e_i}$ auf e DLPs mit Ordnung p zurückführen. Eine explizite Beschreibung hiervon ist in [Buc10, Kap. 11.5.2, S. 185f] und [KK10, Kap. 10.4, S. 186-190] zu finden; implizit ist sie auch im Algorithmus in [MvOV97, Kap. 3.6.4, S. 107-109] wieder zu finden.

[3] Aus Gründen der leichteren Lesbarkeit verzichten wir auf den Subindex i und schreiben anstatt „$n_{p_i} := \dots$“ nur „$n_p := \dots$“ sowie „γ_p“ statt „γ_{p_i}“und später auch „a_p“ anstelle von „a_{p_i}“ .

4.3.2 Spezielle Attacke für DLP

4.3.2.1 Index-calculus-Methode

Die *Index-calculus-Methode* ist die bisher schnellste Möglichkeit um diskrete Logarithmen zu berechnen, aber leider ist sie nicht auf alle Gruppen anwendbar. Wollen wir uns die Index-calculus-Methode anhand der Gruppe $\mathbb{Z}_p{}^* = \mathbb{Z}_p \backslash \{0\}$ anschauen [Buc10, Kap. 11.6, S.187-190]:

Dazu berechnen wir - ähnlich dem Zahlkörpersieb aus Kap. 3.3.13.1 - abhängig von einer oberen Schranke b die Faktorbasis

$$B := \{q \in \mathbb{P} \mid q \leq b\},$$

vgl. Def. 3.44. Um den diskreten Logarithmus in der Gruppe $\langle g \rangle = \mathbb{Z}_p{}^*$ zu berechnen, berechnen wir in einem ersten Schritt den DL für alle Basiselemente $q \in B$. (Dieser Schritt ist noch unabhängig von einem konkreten Element, dessen DL wir erhalten wollen.) Um für ein konkretes $\alpha \in G$ den DL zur Basis g zu erlangen, wählen wir ein zufälliges $k \in [0, p-2] \subseteq \mathbb{N}$, sodass $\alpha \cdot g^k \mod p$ b-glatt ist, dh. $\alpha \cdot g^k \mod p$ *zerfällt* über der Faktorbasis B. Da wir für die Faktorbasiselemente schon die diskreten Logarithmen kennen, können wir, nachdem wir ein passendes k gefunden haben, leicht den diskreten Logarithmus von $\alpha \cdot g^k$, und somit auch jenen von α berechnen. Soweit ein Kurzüberblick über das Vorgehen.

Wollen wir uns nun ansehen, wie wir zu einer gegebenen Faktorbasis $B = \{q_1, q_2, \ldots, q_r\}$ die diskreten Logarithmen ihrer Elemente $q_i \in B$ berechnen können.
Dies geschieht nicht einzeln für jedes q_i extra sondern für alle Basiselemente simultan. Dazu stellen wir ein System von Kongruenzen $(\mod p-1)$ in den Unbekannten

$$\log_g(q_1),\ \log_g(q_2),\ \ldots,\ \log_g(q_r)$$

auf, indem wir einen zufälligen Exponenten $l \in [1, p-2]$ wählen und schauen, ob $g^l \mod p$ über B zerfällt. Falls ja, dann gilt

$$g^l \equiv q_1^{e_1} q_2^{e_2} \cdots q_r^{e_r} \mod p$$

mit geeigneten Exponenten e_i für $i = 1, \ldots, r$ und wir können die Kongruenz

$$l \equiv e_1 \cdot \log_g(q_1) + e_2 \cdot \log_g(q_2) + \ldots + e_r \cdot \log_g(q_r) \mod (p-1)$$

in unser Kongruenzensystem aufnehmen. Nachdem wir auf diese Weise mindestens r verschiedene Kongruenzen bestimmt haben, lösen wir das System nach unseren Unbekannten auf und erhalten so die Werte

$$x_i := \log_g(q_i) \quad \text{für } i = 1, \ldots, r\,.$$

Je nach Wahl der Schranke b für die Elemente in der Faktorbasis stellt der bis jetzt beschriebene Teil den Großteil des Rechenaufwandes dar.

Nun sind wir gerüstet um für ein einzelnes, konkretes $\alpha \in G$ relativ schnell den diskreten Logarithmus zu berechnen. Wie schon erwähnt wählen wir dazu ein zufälliges $k \in [0, p-2] \subseteq \mathbb{N}$, sodass der Wert $\alpha \cdot g^k \mod p$ nur Primfaktoren kleiner oder gleich der Schranke b besitzt. Dann zerfällt das Element $\alpha \cdot g^k$ über der Faktorbasis und wir können es anschreiben als

$$\alpha\, g^k \equiv q_1^{f_1} q_2^{f_2} \cdots q_r^{f_r} \mod p\,.$$

Daraus können wir wegen $x^{p-1} \equiv 1 \mod p$ für $p \in \mathbb{P}$ (vgl. Korollar 3.5) und wegen $\alpha\, g^k = g^{a+k}$ für die Exponenten die Kongruenz

$$a + k \equiv f_1 \log_g(q_1) + f_2 \log_g(q_2) + \ldots + f_r \log_g(q_r) \mod p-1$$

aufstellen, womit wir mit der Formel

$$a \;:=\; \left(\sum_{i=1}^{r} f_i\, x_i\right) - k \qquad \bmod p-1$$

den diskreten Logarithmus a von $\alpha = g^a$ berechnen können.

Eine Variation der Index-calculus-Methode für die Gruppe $\mathbb{Z}_p{}^*$ wird ebenfalls *Zahlkörpersieb*[4] genannt und ist das asymtotisch schnellste Verfahren für die Berechnung des DL in $\mathbb{Z}_p{}^*$. Auch für die Gruppe $GF\,(2^m)^*$ stellt eine spezialisierte Abwandlung der Index-calculus-Methode den schnellsten Angriff auf das DLP dar und heißt *Coppersmith-Algorithmus* [MvOV97, Bem. 3.72, S.112].

Weitere Literatur findet sich in [Sti06, Kap. 6.2.4, S. 244-246] bzw. [KL08, Kap. 8.2.4, S. 311-313]. Zwei anschauliche Beispielen bringt [MvOV97, Kap. 3.6.5, S. 109-112].

[4] Die Ähnlichkeit mit dem in Kap. 3.3.13.1 beschriebenen Verfahren zur Faktorisierung natürlicher Zahlen ist durchaus beabsichtigt.

Kapitel 5

Implementierung in Maple

Für jede der beiden beschriebenen Kryptosysteme (RSA und ElGamal) habe ich den jeweils wichtigsten bzw. effizientesten Angriff in Maple implementiert. Die verwendete Softwareversion ist „Maple 14.01“ in Form einen Studentenlizenz.

5.1 Zahlkörpersieb - (RSA)

5.1.1 Literatur

Für die Implementierung des Zahlkörpersiebes zur Faktorisierung natürlicher Zahlen waren [Lj93], [Stö07] und [Zay95] sehr nützlich.

5.1.2 Anmerkungen

Die meisten der mir zur Verfügung gestandenen Quellen haben den Schritt des Ziehens der algebraischen Quadratwurzel (siehe Seite 68ff) nur sehr knapp ausgeführt. Dieser Teilschritt stellte deshalb eine der größeren Herausforderungen bei der Implementierung des Zahlkörpersiebes dar.

Weiters habe ich bei genau diesem Schritt eine Variante verwendet, die ich zwar in keiner Literatur gefunden habe, die aber den Programmieraufwand verringert hat, indem man bei der Verallgemeinerung von Bem. 3.78 den komplizierteren Fall 3 vermeidet und einen der beiden leichteren Fälle 1 oder 2 erzwingt. Dh. die geeignete Primzahl p muss $p \equiv 3, 5, 7 \mod 8$ erfüllen. Somit wird das algebraische Wurzelziehen auf ein schnelles Potenzieren eines Polynoms mit Grad kleiner d und Koeffizienten kleiner p reduziert, wobei der Exponent im Allgemeinen sehr, sehr groß ist.
Um diesen Exponenten kleiner zu halten, könnte man versuchen nach dem Suchen der linear abhängigen Teilmengen mit jenen zu beginnen, ...

- ... die weniger der gefundenen Relationen benötigen, oder
- ... deren verwendete Relationen (a, b) „näher“ bei $(0, 0)$ liegen.

Der Programmcode meiner Implementierung des Zahlkörpersiebes findet sich im Anhang A ab Seite 93.

5.2 Index-calculus-Methode - (ElGamal)

5.2.1 Anmerkungen

Auch bei der zweiten Implementierung, der Index-calculus-Methode, in Maple ist mir eine Tatsache aufgefallen, die in keiner meiner Quellen beschrieben wurde:
Nachdem man zur Berechnung der diskreten Logarithmen der Elemente der Faktobasis B etwas mehr als $|B|$ Relationen gefunden hat, kann man im Allgemeinen damit nur einen Teil (!) der benötigten Werte $\log_g(p)$ für $p \in B$ berechnen. Deshalb muss man in der Praxis dann eine zweite Faktorbasis B_2 aufstellen, in der nur jene Primzahlen vorkommen, für die tatsächlich der diskrete Logarithmus bekannt ist. Und für das Element $\beta = \alpha \cdot g^k$ wird geprüft, ob es über dieser *neuen Faktorbasis* B_2 zerfällt, und nicht über B.

Den Faktor '-1' in eine Faktorbasis aufzunehmen ist prinzipiell nichts Neues, diesen Umstand haben wir schon beim Zahlkörpersieb kennen gelernt (und wird auch beim Quadratischen Sieb verwendet). In meiner Implementierung habe ich eine Idee von Wikipedia[1] aufgegriffen und bei der Implementierung der Index-calculus-Methode auch den Faktor '-1' in die Faktorbasis aufgenommen. (Auch diese Variante habe ich in keiner Literatur gefunden.) In Folge habe ich beim Test der Glattheit über der Faktorbasis immer die kleinsten Absolutreste verwendet.

Der Programmcode meiner Implementierung der Index-calculus-Methode findet sich im Anhang B ab Seite 121.

[1] http://en.wikipedia.org/wiki/Index_calculus_algorithm

Anhang A

Programmcode - Zahlkörpersieb

Zahlkörpersieb

Setup

Rechengenauigkeit festlegen

```
> setup_rechengenauigkeit := proc(n, Rechengenauigkeit)

    Rechengenauigkeit := ceil(evalf(log[10](n))*1.1+20);

  end proc;
```

Vortests mit n durchführen

n gerade?

```
> setup_vortests_gerade := proc(n, trivial, trivial_Faktoren)

    if (type(n,even)) then
      print("TRIVIALE FAKTORISIERUNG GEFUNDEN: n=", n, " ist gerade!");
      trivial := true;
      trivial_Faktoren := [ 2 , n/2 ];
    end if;

  end proc;
```

n Potenz?

```
> setup_vortests_potenz := proc(n, Rechengenauigkeit, trivial,
  trivial_Faktoren)

    local höchst_mögliche_potenz;          # Obergrenze für
  Schleifenvariable
    local e:=1;                            # Schleifenvariable
    local ist_potenz::boolean := false;    # gibt am Ende an, ob n eine
  Potenz ist
    local basis_approx;

    höchst_mögliche_potenz := floor(evalf(log[2](n)));

    while (e <= höchst_mögliche_potenz) do

      # brauche nur Primzahlen als Potenzen testen
      e := nextprime(e);

      # ungefähre Berechnung der e-ten Wurzel von n
      basis_approx := floor(evalf[ Rechengenauigkeit ](root[e](n)));

      #testen, ob die aktuelle Approximation der e-ten Wurzel eine ganze
  Zahl ist?
      if (basis_approx^e = n) then
        print("TRIVIALE FAKTORISIERUNG GEFUNDEN: n ist eine Potenz: Basis:
  ",basis_approx,", Exponent: ",e);
        trivial := true;
        trivial_Faktoren := [seq(basis_approx, i=1..e)];
        break;
        #return;
      end if;
```

```
    end do;

  end proc;
```

Test, ob n groß genug für Kriterium "n > d^(2d^2)" mit d=3 ist?

```
> setup_vortests_zuklein := proc(n, d, trivial, trivial_Faktoren)

    local zerlegung;

    if ( n <= 3^(2*3^2) ) then
    print("Hinweis:  n=", n, " ist zu klein für das Kriterium: n > d^(2d^2)
  mit d=3. Die Maple-interne Funktion ifactors liefert das Ergebnis:");

    # Aufruf der Maple-internen Funktion "ifactors"
    zerlegung := ifactors(n)[2];
    trivial := true;
    trivial_Faktoren := [seq(seq(potenz[1],j=1..potenz[2]),potenz in
  zerlegung)];
  end if;

  end proc;
```

Polynomgrad d bestimmen (ungerade, >=3)

```
> setup_polynomgrad := proc(n, d)

    local d_temp;

    # erster Vorschlag für d
    d_temp := floor(evalf((( 3*ln(n) )/( ln(ln(n)) ))^(1/3)));

    # auf ungeraden Polynomgrad abrunden
    if (modp(d_temp,2)=0) then
      d_temp := d_temp - 1;
    end if;

    # sicherstellen, dass Polynomgrad mindestens 3 beträgt
    d_temp := max( d_temp , 3 );

    d := d_temp;

  end proc;
```

m bestimmen

```
> setup_m := proc(n, d, Rechengenauigkeit, m)

    m := floor(evalf[ Rechengenauigkeit ](root[d](n)));

  end proc;
```

Polynom f(x) bestimmen - mittels base-m-Methode

```
> setup_polynom := proc(n, m, f_koeffs, f_koeffs_anzahl, f)
```

```
  local f_koeffs_temp;
  local f_koeffs_anzahl_temp;
  local i;                        # Schleifenvariable

  # Bestimmung der Koeffizienten von f(x)
  f_koeffs_temp := convert(n,base,m);
  f_koeffs := f_koeffs_temp;

  # Anzahl der Koeffizienten von f(x):
  f_koeffs_anzahl_temp := nops(f_koeffs_temp);
  f_koeffs_anzahl := f_koeffs_anzahl_temp;

  # Aufstellen des Polynoms f(x) an sich
  f := unapply( sum( x^(i-1) * f_koeffs_temp[i] , i=1..f_koeffs_anzahl_temp )
, x );

end proc;
```

Tests mit f(x) machen

```
> setup_polynom_tests := proc(n, d, m, f, f_koeffs, f_koeffs_anzahl,
f_abgeleitet, trivial, trivial_Faktoren)

  local f_abgeleitet_temp;
  local ggt;

  # Probe, ob Polynom f(x) vollen Grad d hat?
  if (f_koeffs_anzahl <> d+1) then
    print("Warnung: Das Polynom f(x) hat nicht vollen Grad d!");
  end if;

  # Probe, ob Polynom f(x) normiert ist?
  if (f_koeffs[-1] <> 1) then
    print("Warnung: Das Polynom f(x) ist nicht normiert!")
  end if;

  # Probe, ob f(m)=n ist?
  if(f(m) <> n) then
    print("Warnung: m ist keine Nullstelle des Polynoms f(x)!");
  end if;

  # Probe, ob f(x) über Z zerfällt?
  if (not(irreduc(f(x)))) then
    #print("f(x) IST REDUZIBEL UND SOMIT HABEN WIR EINE FAKTORISIERUNG VON
n:");
    #print(subs(x=m, factors(f(x))[2][1..2,1] ));
    trivial := true;
    trivial_Faktoren := subs(x=m, factors(f(x))[2][1..2,1] );
  end if;

  # Probe, ob ggT( f'(m) , n ) > 1 ist?
  f_abgeleitet_temp := unapply( diff( f(x) , x ) , x);
  f_abgeleitet := f_abgeleitet_temp;
  ggt := igcd( f_abgeleitet_temp(m) , n );
  if (ggt>1) then
    #print("DER ggT( f'(m) , n ) IST GRÖßER ALS 1: !!!", ggt, n/ggt);
    trivial := true;
    trivial_Faktoren := sort([ggt, n/ggt]);
  end if;
```

```
  end proc;
```

homogenisiertes Polynom F(x,y) aufstellen

```
> setup_homogenesPolynom := proc(f_koeffs, f_koeffs_anzahl, F)

    F := unapply( sum( x^(i-1) * y^(f_koeffs_anzahl-i) * f_koeffs[i] ,
  i=1..f_koeffs_anzahl ) , [x,y] );

  end proc;
```

Faktorbasen aufstellen

Laufzeitfunktion L_n[u,v] aufstellen

```
> fb_laufzeitfunktion := proc(n, L)

    L := (u,v) -> exp( v * (ln(n))^u * (ln(ln(n)))^(1-u) );

  end proc;
```

RFB - rationale Faktorbasis

obere Schranke für die RFB berechnen

```
> fb_rfb_schranke := proc(L, Rechengenauigkeit, RFB_Schranke)

    RFB_Schranke :=       floor(evalf[ Rechengenauigkeit
  ](L(1/3,(8/9)^(1/3))));
    #RFB_Schranke := min ( floor(evalf[ Rechengenauigkeit
  ](L(1/3,(8/9)^(1/3))) , 1800 );
    #RFB_Schranke := min ( max ( floor(evalf[ Rechengenauigkeit
  ](L(1/3,(8/9)^(1/3))) , 50 ) , 1800 );
  end proc;
```

Primzahlen in der RFB berechnen

```
> fb_rfb_primzahlen := proc(RFB_Schranke, RFB_p)

    RFB_p := select( isprime , [$2..RFB_Schranke] );

  end proc;
```

Anzahl der Elemente in RFB berechnen

```
> fb_rfb_anzahl := proc(RFB_p, RFB_Anzahl)

    RFB_Anzahl := nops(RFB_p);

  end proc;
```

m mod p für alle p in RFB berechnen

```
> fb_rfb_paare := proc(RFB_p, m, RFB)

    RFB := [seq( [p,modp(m,p)] , p in RFB_p )];

  end proc;
```

größtes Element in der rationalen Faktorbasis berechnen

```
> fb_rfb_maximum := proc(RFB_p, RFB_max)

    RFB_max := RFB_p[-1];

  end proc;
```

AFB - algebraische Faktorbasis

obere Schranke für Primzahlen in AFB

```
> fb_afb_schranke := proc(RFB_Schranke, Faktor_AFB_zu_RFB, AFB_Schranke)

    AFB_Schranke := Faktor_AFB_zu_RFB * RFB_Schranke;

  end proc;
```

zusätzliche Primzahlen in AFB, die nicht schon in RFB vorkommen berechnen und zusammenfügen der Primzahlen aus RFB und den neuen in AFB zu den potentiellen(!) Primzahlen in AFB. "Potentiell" deshalb, weil später nur jene mit R(p) ungleich der leeren Menge interessant sind.

```
> fb_afb_primzahlen := proc(RFB_Schranke, AFB_Schranke, RFB_p,
  AFB_p_möglich)

    AFB_p_möglich := [ op(RFB_p) , op(select( isprime ,
  [$RFB_Schranke+1..AFB_Schranke] )) ];

  end proc;
```

Hilfsfunktion zur Fortschrittsanzeige beim Berechnen der R(p) = { 0 <= r <= p-1 | f(r) = 0 mod p }

```
> fb_afb_anzeige := proc(AFB_p_möglich, AFB_Anzahl_möglicher_r,
  AFB_p_möglich_Anzahl)

    AFB_Anzahl_möglicher_r := Threads[Add]( p , p in AFB_p_möglich );
    AFB_p_möglich_Anzahl := nops(AFB_p_möglich);

  end proc;
```

R(p) berechnen

```
> fb_afb_R_p_berechnen := proc(AFB_p_möglich, AFB_p_möglich_Anzahl,
  AFB_Anzahl_möglicher_r, f, d, ausgabe_r, ausgabe_p, AFB_pEinfach_rListe,
  AFB_p_Anzahl, AFB_pMehrfach_rEinfach, AFB_Anzahl)

    local AFB_pEinfach_rListe_temp;
    local AFB_p_Anzahl_temp;
    local AFB_pMehrfach_rEinfach_temp;
```

```
  local st, st_p_alt, st_p_neu, st_r_alt, st_r_neu, zähler_r;      #
Hilfsvariablen zur Fortschrittsanzeige
  local zähler_p, r, i;                                  # Schleifenvariablen
  local p, liste;       # Hilfsvariablen in Schleife


  # erste Form der AFB leer initialisieren
  AFB_pEinfach_rListe_temp := Array(1..0,1..2);
  AFB_p_Anzahl_temp := 0;

  # Zeitmessung zur Fortschrittsanzeige
  st := time();
  st_p_alt := st;
  st_r_alt := st;
  zähler_r := 0;

  for zähler_p from 1 to AFB_p_möglich_Anzahl do

    p := AFB_p_möglich[zähler_p];

    liste := []:   # hier kommen (für jedes p) die r mit "f(r) = 0 mod p"
hinein

    for r from 0 to p-1 do

      if ( modp(f(r),p)=0 ) then

        liste := [ op(liste) , r ];

        if (nops(liste)=d) then
          zähler_r := zähler_r + p-r;   # fehlende r für Zähler noch
dazuzählen
          break;
        end if; #if nops(liste) = d

      end if; #if modp(f(r),p) = 0

      # r-Zähler erhöhen und ggf. r-Fortschrittsanzeige ausgeben
      zähler_r := zähler_r+1;
      if ( modp(zähler_r,ausgabe_r)=0 ) then
        st_r_neu := time();
        print( "r   " , evalf[5](zähler_r/AFB_Anzahl_möglicher_r*100), "%
der r,   ", evalf[5](zähler_p/AFB_p_möglich_Anzahl*100), "% der p,   ", "
Zeit (r): ", st_r_neu - st_r_alt, " Sekunden" );
        st_r_alt := st_r_neu;

      end if; #if modp(zähler_r,ausgabe_r) = 0

    end do; #for r = 0..p-1

    #print(p,liste);

    # Falls es ein oder mehrere passende r gibt, dann diese in die
Ergebnisliste eintragen
    if (nops(liste)>0) then
      AFB_p_Anzahl_temp := AFB_p_Anzahl_temp + 1;
      AFB_pEinfach_rListe_temp :=
Array(1..AFB_p_Anzahl_temp,1..2,AFB_pEinfach_rListe_temp):
      AFB_pEinfach_rListe_temp[AFB_p_Anzahl_temp,1] := p;
      AFB_pEinfach_rListe_temp[AFB_p_Anzahl_temp,2] := liste;
    end if; #if nops(liste) > 0
```

```
    # ggf. p-Fortschrittsanzeige ausgeben
    if ( modp(zähler_p,ausgabe_p)=0) then
      st_p_neu := time();
      print( "AFB aufstellen: " ,
 evalf[5](zähler_r/AFB_Anzahl_möglicher_r*100), "% der r,   ",
 evalf[5](zähler_p/AFB_p_möglich_Anzahl*100), "% der p,   ", "  Zeit (p):
", st_p_neu - st_p_alt, " Sekunden" );
      st_p_alt := st_p_neu;
    end if;

  end do: #for zähler_p = 1 .. AFB_p_möglich_Anzahl



  #printlevel := 100;



  # zweite Form der AFB leer initialisieren
  AFB_pMehrfach_rEinfach_temp := [];

  for i from 1 to AFB_p_Anzahl_temp do

    p     := AFB_pEinfach_rListe_temp(i,1);
    liste := AFB_pEinfach_rListe_temp(i,2);

    AFB_pMehrfach_rEinfach_temp := [ op(AFB_pMehrfach_rEinfach_temp) ,
 seq( [p,r] , r in liste ) ];

  end do; #for i = 1..AFB_p_Anzahl_temp

  AFB_Anzahl := nops(AFB_pMehrfach_rEinfach_temp);


  # lokale, temporäre Variablen zurückgeben/zurückschreiben

  AFB_pEinfach_rListe := AFB_pEinfach_rListe_temp;
  AFB_p_Anzahl := AFB_p_Anzahl_temp;
  AFB_pMehrfach_rEinfach := AFB_pMehrfach_rEinfach_temp;

end proc;
```

QCB - quadratische Charaktere Basis

```
> fb_qcb := proc(AFB_Schranke, QCB_Anzahl, f, f_abgeleitet, QCB_p_getestet,
QCB)

  local QCB_p_getestet_temp := [];
  local QCB_temp := [];

  local anzahl_qcb_gefunden := nops(QCB_temp);
  local q := AFB_Schranke;
  local f_mod_q;
  local f_abgeleitet_mod_q;
  local s;                                          # Schleifenvariable

  while ( anzahl_qcb_gefunden < QCB_Anzahl ) do

    q := nextprime(q);
```

```
      QCB_p_getestet_temp := [ op(QCB_p_getestet_temp) , q ];
      f_mod_q := unapply( f(x) mod q , x );
      f_abgeleitet_mod_q := unapply( f_abgeleitet(x) mod q , x );

      for s from 0 to q-1 do
        if (modp( f_mod_q(s) , q )=0) then
          if (modp( f_abgeleitet_mod_q(s) , q )>0) then
            QCB_temp := [ op(QCB_temp) , [q,s] ];
            anzahl_qcb_gefunden := nops(QCB_temp);
            #print("quadratischen Charakter gefunden:     q=", q, "  s=", s);
            break;
          end if;
        end if;
      end do;

    end do:


    # Rückgabe der temporären, lokalen Variablen
    QCB_p_getestet := QCB_p_getestet_temp;
    QCB := QCB_temp;

  end proc;
```

Probedivision mit Primzahlen aus allen (Faktor-)Basen

Probedivision mit allen(!) Primzahlen bis zum größten Element aus der QCB

```
> probedivision := proc(AFB_p_möglich, QCB_p_getestet, n,
  alle_möglichen_Primzahlen, trivial, trivial_Faktoren)

    local alle_möglichen_Primzahlen_temp;

    local p;                                  # Schleifenvariable
    local ggt;

    alle_möglichen_Primzahlen_temp := [ op(AFB_p_möglich) , op(QCB_p_getestet) ];
    alle_möglichen_Primzahlen := alle_möglichen_Primzahlen_temp;

    for p in alle_möglichen_Primzahlen_temp do
      ggt := igcd(p,n);
      if (ggt>1) then
        print("PRIMFAKTOR BEI PROBEDIVISION GEFUNDEN !!!    p=", p, "   n/p=",
  n/p, "   ggt=", ggt);
        trivial := true;
        trivial_Faktoren := sort([p, n/p]);
        break;
      end if;
    end do:

  end proc;
```

Vorbereitungen fürs Sieben

Anzahl der zu suchenden Relationen

```
> siebvorbereitungen_anzahlRelationenNotwendig := proc(RFB_Anzahl, AFB_Anzahl,
  QCB_Anzahl, anzahlRelationenZusätzlich, anzahlRelationenNotwendig,
  Schleifenmatrix_AnzahlZeilen)

    anzahlRelationenNotwendig :=
    RFB_Anzahl  # Anzahl Primzahlen in der RFB
  + AFB_Anzahl  # Anzahl der Paare (p,r) in der AFB
  + 1           # Vorzeichen von N(a+bm)
  + QCB_Anzahl  # Anzahl quadratischer Charaktere (q,s) in der QCB
  + 1           # "lauter 1er"-Spalte, damit eine gerade Anzahl an Paaren (a,b)
  ausgewählt wird
  + 1           # 1 mehr, damit man tatsächlich eine linear abhängige Teilmenge
  finden kann
  + anzahlRelationenZusätzlich;   # Sicherheitspuffer

    Schleifenmatrix_AnzahlZeilen := RFB_Anzahl + AFB_Anzahl + 1 + QCB_Anzahl +
  1;

  end proc;
```

Schranke für Siebbereich (für a) festlegen

```
> siebvorbereitungen_siebschranke := proc(alle_möglichen_Primzahlen, m, L,
  Rechengenauigkeit, C);

    #C := min( floor(m/2) , alle_möglichen_Primzahlen[-1] );
    C := min( floor(m/2) , max ( alle_möglichen_Primzahlen[-1] , 3 *
  floor(evalf[ Rechengenauigkeit ](L(1/3,(8/9)^(1/3)))) + 200 * floor(evalf[
  Rechengenauigkeit ](ln(L(1/3,(8/9)^(1/3))))) ) , 10^5 );
    #C := 3 * floor(evalf[ Rechengenauigkeit ](L(1/3,(8/9)^(1/3)))) + 200 *
  floor(evalf[ Rechengenauigkeit ](ln(L(1/3,(8/9)^(1/3))))) ;

  end proc;
```

Sieben

Initialisierung der Variablen fürs Sieben

```
> sieben_variableninitialisierung := proc(RelationenMatrix,
  Schleifenmatrix_AnzahlZeilen, anzahlRelationenNotwendig, C, Siebintervall)

    # Relationenmatrix (leer) initialisieren - speichert die Exponenten jener
  Paare (a,b), die komplett über den beiden Faktorbasen RFB und AFB zerfallen
    RelationenMatrix :=
  Matrix(Schleifenmatrix_AnzahlZeilen,anzahlRelationenNotwendig,storage=sparse)
  ;

    Siebintervall := [$-C..C];


  end proc;
```

Sieben

```
> sieben := proc(Schleifenmatrix_AnzahlZeilen, anzahlRelationenNotwendig, C, m,
  F, RFB_Anzahl, AFB_Anzahl, QCB_Anzahl, RFB, AFB_pMehrfach_rEinfach, QCB,
  Siebintervall, RelationenMatrix, RelationenListe, anzahlRelationenGefunden)
```

```
  local RelationenListe_temp := [];


  local Schleifenmatrix :=
Array(1..Schleifenmatrix_AnzahlZeilen,-C..C,storage=sparse);

  local b;                                      # zweite Koordinate eines
Paares (a,b)
  local anzahlRelationenGefunden_temp := 0;     # Zählvariable, wieviele gute
Relationen schon gefunden wurden
  local B := Array(-C..C,fill=true);            # boolsches Array
  local paar;                                   # Hilfsvariable zur
Initialisierung (Speicherlöschung) der Schleifenmatrix o.ä.
  local Schleifenliste;                         # Liste der guten a-Werte für
einen festgehaltene b-Wert
  local primfaktorzerlegung_von_b;              # Hilfsvariable für GGT-Sieb
  local p;                                      # Schleifenvariable im
GGT-Sieb / erster Eintrag für Element in RFB / AFB-Sieb
  local startpunkt;                             # der linkeste(!) Punkt bei
einem Siebdurchgang
  local a;                                      # erste Koordinate eines
Paares (a,b)
  local bm;                                     # Hilfswert für "b*m"
  local R;                                      # Array für rationales Sieb
  local i;                                      # Schleifenvariable
  local r;                                      # zweiter Wert eines Eintrages
(p,r) in der RFB bzw. AFB
  local seed;                                   # ein(!) passender Wert beim
Sieben
  local vielfachheit;                           # Vielfachheit eines
Primfaktors bei RFB- und AFB-Sieb
  local ggt;                                    # Hilfsvariable zur
Faktorisierung über RFB bzw. AFB
  local dummy_rat;                              # dient nur dem Aufspüren
eines Fehlers, sonst nichts
  local N;                                      # Norm-Array für algebraisches
Sieb
  local dummy_alg;                              # dient nur dem Aufspüren
eines Fehlers, sonst nichts
  local gute_a_Werte;                           # jene Werte für a, sodass für
ein festgehaltenes b das Paar (a,b) eine gute(!) Relation ist
  local anzahlRelationenNeu;                    # Anzahl der guten a-Werte
(=nops(gute_a_Werte)) pro festem b-Wert
  local j;                                      # Schleifenvariable
  local q;                                      # erste Komponente eines
Paares (q,s) in QCB
  local s;                                      # zweite Komponente eines
Paares (q,s) in QCB
  local i_indizes;                              # Hilfsvariable zum Übertragen
der Exponenten für RFB und AFB von Schleifenmatrix in die Relationenmatrix


  print("Beginn: Relationen suchen");

  for b from 1 to infinity while (anzahlRelationenGefunden_temp <
anzahlRelationenNotwendig) do

    print("b=",b);

    # boolsches Array "B" initialisieren
    for i from -C to C do
```

```
      B[i] := true;
    end do;

    # Schleifenmatrix initialisieren
    for paar in indices(Schleifenmatrix) do
      Schleifenmatrix[paar[1],paar[2]] := 0;
    end do;

    # Schleifenliste initialisieren - speichert für dieses eine konkrete "b"
alle jene Paare (a,b), die über RFB und AFB komplett zerfallen
    Schleifenliste := [];


    #GGT-SIEB

    #print("Beginn: GGT-Sieb");
    primfaktorzerlegung_von_b := (ifactors(b)[2])[1..-1,1];
    for p in primfaktorzerlegung_von_b do
      startpunkt := -C + (C mod p);
      for a from startpunkt by p to C do
        B[a] := false;
      end do; #a by p in [-C..C]
    end do; #p primfakt. von b
    #print("Ende: GGT-Sieb");


    # RATIONALES SIEBEN

    # Vorbereitungen für rationales Sieb

    bm := b*m;
    R := Array(-C..C,[$((-C)+bm)..(C+bm)]):   # R ... rationales Sieben

    # Vorzeichen von "a+bm" ist in unserem Fall immer positiv -> kein
Vorzeichen notwendig

    # Schleife über alle Primzahlen bzw. Paare (p,r) in RFB

    #print("Beginn: rat. Sieb");
    for i from 1 to RFB_Anzahl do

      paar := RFB[i]:
      p := paar[1]:
      r := paar[2]:
      seed := modp( modp(-b,p) * r , p ):   # ist ein passender a-Wert im
Bereich zw. 0 und p-1
      startpunkt := -C + modp( seed + modp(C,p) , p ):

      # Schleife=Sieb über alle passenden a-Werte
      for a from startpunkt by p to C do

        # nur wenn das ggT-Sieb es zulässt
        if (B[a]) then

          # höchste Potenz von p aus R[a] herausdividieren
          vielfachheit := 0;   # zählt die Vielfachheit des Primfaktors p in
R[a]
          ggt := igcd( R[a] , p );
          while (ggt=p) do
            R[a] := R[a]/p;
            vielfachheit := vielfachheit + 1;
            ggt := igcd( R[a] , p );
```

```
          end do; #while: höchste Potenz von p aus R[a] herausdividieren

          #evt. Eintragen in die Schleifenmatrix
          if (vielfachheit > 0) then
            Schleifenmatrix[ i , a ] := vielfachheit;
          end if; #if vielfachheit > 0

        end if; #if: wenn ggT-Sieb es für ein konkretes a zulässt

      end do; #for a by p

    end do: #for i = 1 .. RFB_Anzahl / dh. Schleife über alle (p,r) in RFB
    #print("Ende: rat. Sieb");

    #Sieb aktualisieren

    # Schleife über alle a in [-C..C]
    for a from -C to C do

      # wenn für dieses a nicht! R[a]=1 übrig geblieben ist, dann das
boolsche Array für dieses a auf false setzen
      if (R[a]>1) then
        B[a] := false;
      elif (R[a]<1) then
        print("FEHLER: R[a] < 1,    b=", b, "    a=", a, "   R[a]=", R[a], "
Schleifenmatrix an der Stelle (a,b):");
        print(op(Schleifenmatrix[1..Schleifenmatrix_AnzahlZeilen,a]));
      # else: R[a]=1, dh. "a+bm" zerfällt über der rationalen Faktorbasis
      end if;

    end do; #for a in [-C..C]

    # ALGEBRAISCHES SIEB

    # Array für Normen leer erstellen (N ... Norm)
    N := Array(-C..C,storage=sparse):

    # Array mit Normen initialisieren - nur wenn das Sieb a noch als sinnvoll
erachtet
    for a from -C to C do
      if (B[a]) then
        N[a] := F(a,-b);
      end if;
    end do;

    # tatsächliches algebraisches Sieben

    #print("Beginn: alg. Sieb");
    # Schleife über alle (p,r) in AFB
    for i from 1 to AFB_Anzahl do

      paar := AFB_pMehrfach_rEinfach[i]:
      p := paar[1]:
      r := paar[2]:

      # wenn p teilt b, dann kann es kein a geben mit p teilt N(a,b)  (s. LN
S.57)
      if ( igcd(b,p) = p ) then
        #print("Wegen ggt(b,p)=p Sprung zum nächsten Primfaktor (p,r) in
AFB");
        next;
      end if;
```

```
      seed := modp( modp(-b,p) * r , p ):   # ist ein passender a-Wert im
Bereich zw. 0 und p-1
      startpunkt := -C + modp( seed + modp(C,p) , p ):

      for a from startpunkt by p to C do

        # nur wenn das rat. Sieb es noch zulässt
        if (B[a]) then

          # höchste Potenz von p aus N[a] herausdividieren
          vielfachheit := 0;
          ggt := igcd( N[a] , p );

          while (ggt=p) do
            #print("TEST: b=", b, "   a=", a, "   p=", p, "   N[a]", N[a]);
            N[a] := N[a]/p;
            vielfachheit := vielfachheit + 1;
            ggt := igcd( N[a] , p );
          end do; #while: höchste Potenz von p aus N[a] herausdividieren

          #evt. Eintragen in die Schleifenmatrix
          if (vielfachheit > 0) then
            Schleifenmatrix[ RFB_Anzahl + i , a ] := vielfachheit;

          end if; #if vielfachheit > 0

        end if; #if B[a]

      end do; #for a by p in -C..C

    end do: #for (p,r) in AFB
    #print("Ende: alg. Sieb");


    #Sieb aktualisieren (nach alg.Sieb)

    # Schleife über alle a in [-C..C]
    for a from -C to C do

      # nur die noch sinnvollen a betrachten
      if (B[a]) then

        # wenn für dieses a nicht! N[a]=+-1 übrig geblieben ist, dann das
boolsche Array B für dieses a auf false setzen
        if ( abs(N[a]) > 1 ) then
          B[a] := false;
        elif (N[a]=0) then
          print("FEHLER: N[a] = 0,    b=", b, "    a=", a, "   N[a]=", N[a],
"   Schleifenmatrix an der Stelle (a,b):");
          print(op(Schleifenmatrix[1..Schleifenmatrix_AnzahlZeilen,a]));
        elif (N[a]=-1) then

          # N(a+bm) zerfällt über der alg. Faktorbasis und hat negatives
Vorzeichen
          # negatives Vorzeichen in Schleifenmatrix eintragen
          Schleifenmatrix[ RFB_Anzahl + AFB_Anzahl + 1 , a ] := 1;

        # else: N[a]=+1, dh. "N(a+bm)" zerfällt über der algebraischen
Faktorbasis (mit positivem Vorzeichen)
        end if; #if N[a]=?
```

```
      end if; #if B[a]

    end do; #for a in [-C..C] - boolsches Sieb nach AFB-Sieb aktualisieren +
Vorzeichen von N(a,b) eintragen

    # ENDE: SIEBEN  -  Filtern jener a's, die in die Relationenmatrix
aufgenommen werden sollen
    gute_a_Werte := select( x -> B[x] , Siebintervall):   # jene a's, sodass
(a,b) eine brauchbare Relation ist
    anzahlRelationenNeu := nops(gute_a_Werte);
    print("Anzahl neuer Relationen = ", anzahlRelationenNeu);


    # QUADRATISCHE CHARAKTERE + ÜBERTRAGEN DER GUTEN A-WERTE-REALTIONEN

    #print("Beginn: quadr.Charakt. + Übertragen");
    # für alle noch sinnvollen a die quadratischen Charaktere berechnen, den
"ewigen 1er" eintragen und auch gleich in die Relationenmatrix übertragen
    for j from 1 to anzahlRelationenNeu do

      a := gute_a_Werte[j]:

      # quadr.Charaktere berechnen + eintragen in Schleifenmatrix
      for i from 1 to QCB_Anzahl do

        paar := QCB[i];
        q := paar[1];
        s := paar[2];

        Schleifenmatrix[ RFB_Anzahl + AFB_Anzahl + 1 + i , a ] := ( 1 -
numtheory[legendre](a+b*s,q) )/2;

      end do; #for (q,s) in QCB

      # "ewigen 1er" eintragen
      Schleifenmatrix[ Schleifenmatrix_AnzahlZeilen , a ] := 1;

      # Übertragen der Informationen von Schleifenmatrix zur
Relationenmatrix

      i_indizes := lhs~(op(Schleifenmatrix[1..-1,a])[2]);

      # Schleife über alle Einträge <> 0 im Exponentenvektor
      for i in i_indizes do
        RelationenMatrix[ i , anzahlRelationenGefunden_temp + 1 ] :=
Schleifenmatrix[ i , a ];
      end do; #for i in i_indizes

      # Hinzufügen des Paares (a,b) zur Relationenliste
      RelationenListe_temp := [ op(RelationenListe_temp) , [a,b] ];

      anzahlRelationenGefunden_temp := anzahlRelationenGefunden_temp + 1;

      #Abbrechen wenn genügend viele Paare (a,b) gefunden wurden
      if ( anzahlRelationenGefunden_temp = anzahlRelationenNotwendig) then
        break;
      end if;

    end do; #for j = 1 .. anzahlRelationenNeu, dh. for a in gute_a_Werte

    #print("Ende: quadr.Charakt. + Übertragen");
```

```
    print("Fortschritt:  ", anzahlRelationenGefunden_temp, "/",
anzahlRelationenNotwendig, "  =  ",
evalf[5](anzahlRelationenGefunden_temp/anzahlRelationenNotwendig*100), "%");

  end do;

  if (anzahlRelationenGefunden_temp >= anzahlRelationenNotwendig) then
    print("SIEBEN FERTIG: genügend Relationen gefunden");
  end if;

  # Rückgabe der temporären lokalen Variablen
  RelationenListe := RelationenListe_temp;
  anzahlRelationenGefunden := anzahlRelationenGefunden_temp;


end proc;
```

Matrix-Schritt & Lösungen durchprobieren

Matrix-Schritt: linear abhängige Teilmenge suchen

```
> matrixschritt := proc(RelationenMatrix, lösungsbasis)

  local st;   # Hilfsvariable zur Zeitnehmung

  print("Beginn: Suche lin. abh. Teilmengen");
  st := time():

  lösungsbasis := Nullspace(RelationenMatrix) mod 2;

  print("Ende: Suche lin. abh. Teilmengen");
  print("Rechenzeit für Matrix-Schritt:  ", time() - st, " Sekunden");

end proc;
```

Lösungsbasisvektoren durchprobieren

```
> lösungen_probieren := proc(lösungsbasis, Schleifenmatrix_AnzahlZeilen,
RelationenMatrix, RelationenListe, RFB, RFB_Anzahl, n, m, d, f, f_koeffs,
f_koeffs_anzahl, f_abgeleitet, anzahlRelationenGefunden, Faktoren)

  local lösungsdimension := nops(lösungsbasis);     # Anzahl der linear
unabhängigen Lösungsvorschläge
  local faktorisierungErfolgreich := false;         # Indikator, ob
Faktorisierung schon erfolgreich war
  local anzahl_versuche;                            # Nummer des aktuellen
Lösungsversuches
  local spaltenauswahl;                             # Liste der ausgewählten
Spalten der RelationenMatrix für den aktuellen Lösungsversuch
  local exponentenvektor;                           # Vektor mit den Summen
der Exponenten der momentan ausgewählten Spalten von RelationenMatrix
  local j;                                          # Schleifenvariable
  local spaltenIndizes;                             # Indizes einer Spalte(!)
aus RelationenMatrix mit Einträgen ungleich 0
  local i;                                          # Schleifenvariable
  local vorkommendeIndizes;                         # Indizes des
aufsummierten Exponentenvektors(!) mit Einträgen ungleich 0
```

```
  local vorkommendeExponenten;                    # Einträge des
aufsummierten Exponentenvektors ungleich 0
  local ratWurzel;                                # Wert der Wurzel der
rationalen Seite
  local vorkommendeIndizes_RFB;                   # Teilmenge von
vorkommendeIndizes, die zu RFB gehören
  local paar;                                     # Hilfsvariable beim
Auslesen der Elemente der Faktorbasen RFB, Relationenliste, AFB
  local p;                                        # erste Komponente der
Hilfsvariable "paar" beim Auslesen der Elemente der Faktorbasen RFB und AFB
  local e;                                        # Hilfsvariabel für
Exponenten
  local f_temp;                                   # halber Exponent beim
Bilden der rationalen Wurzel
  local p_primzahlpotenz;                         # Teilfaktor beim square
& multiply-Algorithmus, der die "richtigen" 2er-Potenzen "aufsammelt"
  local p_aktuellesQuadrat;                       # Hilfsvariable beim
square & multiply-Algorithmus, die jede Runde quadriert wird
  local ratQuadrat;                               # zur Kontrolle ob die
Quadrate in derselben Restklasse mod n liegen
  local algQuadrat;                               # algebraisches Quadrat
als Polynom in x mod f(x)
  local a, b;                                     # Komponenten der guten
Relationen (a,b)
  local f_Norm;                                   # Wert für Bestimmung
einer oberen Schranke für die Wurzelkoeffizienten
  local u;                                        # Wert für Bestimmung
einer oberen Schranke für die Wurzelkoeffizienten - Maximum der verwendeten
Parre (a,b)
  local spaltenauswahl_anzahl;                    # Anzahl der im aktuellen
Lösungsversuch verwendeten Spalten von RelationenMatrix
  local obereSchranke_Wurzelkoeffizienten;        # obere Schranke für die
Koeffizienten der algebraischen Wurzel als Polynom in x
  local p_gefunden;                               # Indikator, ob schon
eine passende Primzahl für den algebraischen Wurzelschritt gefunden wurde
  local rest;                                     # für Suche nach
geeigneter Primzahl für algebraischen Wurzelschritt
  local G;                                        # Galois-Feld für alg.
Wurzelschritt
  local q;                                        # algebraisches Quadrat
im Galois-Feld
  local r1, r2;                                   # die beiden Wurzeln im
Galois-Feld
  local R1, R2;                                   # Koeffizienten der
beiden alg. Wurzeln
  local S1, S2;                                   # kleinster Absolutrest
der Koeffizienten der beidne alg. Wurzeln
  local T1, T2;                                   # alg. Wurzeln als
Polynome (in Maple)
  local algWurzel1, algWurzel2;                   # Bild der beiden alg.
Wurzeln in Z_n
  local wurzeln_getestet;                         # jene Bilder der alg.
Wurzeln in Z_n, deren Quadrat gleich ratQuadrat ist
  local algWurzel;                                # Laufvariable in
wurzeln_getestet
  local differenz;                                # algWurzel - ratWurzel
  local summe;                                    # algWurzel + ratWurzel
  local ggt;                                      # für ggT( x-y, n ) bzw.
ggT( x+y , n )


  for anzahl_versuche from 1 to lösungsdimension while
```

```
(not(faktorisierungErfolgreich)) do

    print(anzahl_versuche, "-ter Lösungsbasisvektor von ",
lösungsdimension);
    spaltenauswahl := lhs~(op(lösungsbasis[anzahl_versuche])[2]):

    # EXPONENTENVEKTOR BERECHNEN

    # selbst programmierte sparse-Version

    # Exponentenvektor leer initialisieren
    exponentenvektor := Array(1..Schleifenmatrix_AnzahlZeilen):

    # Schleife über alle in dieser Kombination/Lösungsversuch ausgewählten
Spalten
    for j in spaltenauswahl do

      # Indizes der ausgewählten Spalte/Relation
      spaltenIndizes := lhs~(op(RelationenMatrix[ 1..-1 , j ])[2]):

      # Exponenten der aktuellen Spalte/Relation zur Gesamtsumme hinzufügen
      for i in spaltenIndizes do
        exponentenvektor[i] := exponentenvektor[i] + RelationenMatrix[ i , j
];
      end do; #for i in spaltenIndizes

    end do; #for spalte in spaltenauswahl


    vorkommendeIndizes := lhs~(op(exponentenvektor)[2]):
    vorkommendeExponenten := rhs~(op(exponentenvektor)[2]):

    # Fehlerabfrage, ob Exponentenvektor eh keine ungeraden Einträge besitzt
    if ( (vorkommendeExponenten mod 2) = {0} ) then
      # alle Exponenten sind wie zu erwarten gerade, dh. es passt eh alles
    else
      print("FEHLER:  DER EXPONENTENVEKTOR HAT NICHT NUR GERADE SONDERN AUCH
UNGERADE (!) EINTRÄGE !!!");
    end if;

    # BERECHNEN DER RATIONALEN WURZEL

    # rationale Wurzel initialisieren
    ratWurzel := 1;

    # Indizes für die rationale Wurzel
    vorkommendeIndizes_RFB := vorkommendeIndizes intersect {$1..RFB_Anzahl};

    for i in vorkommendeIndizes_RFB do

      paar := RFB[i];
      p := paar[1];
      e := exponentenvektor[i];

      # initialisieren der Variablen für square & multiply - Algorithmus
      f_temp := e/2;
      p_primzahlpotenz := 1; # hier soll das ergebnis des
square-and-multiply-Algorithmus gespeichert werden
      p_aktuellesQuadrat := p;

      while (f_temp > 0) do
```

```
        # multiply-Schritt
        if ( modp(f_temp,2) = 1 ) then
          p_primzahlpotenz := modp( p_primzahlpotenz * p_aktuellesQuadrat , n
);
          f_temp := f_temp - 1;
        end if;

        # square-Schritt
        p_aktuellesQuadrat := modp( p_aktuellesQuadrat * p_aktuellesQuadrat ,
n );

        f_temp := f_temp / 2;

      end do; #while f_temp > 0

      # aktuelle Primzahlpotenz zum laufenden Ergebenis der rat. Wurzel
dazumultiplizieren
          ratWurzel := modp( ratWurzel * p_primzahlpotenz , n );

    end do; #for i in vorkommendeIndizes_RFB

    # f'(m) zur rationalen Wurzel dazumultiplizieren
    ratWurzel := modp( ratWurzel * f_abgeleitet(m) , n ):

    # Berechnung des rationalen Quadrates (dient nur der Kontrolle)
    ratQuadrat := modp( ratWurzel * ratWurzel , n ):


    # BERECHNEN DES ALGEBRAISCHEN QUADRATS

    # Aufmultiplizieren des algebraischen Quadrats mod f(x) - zur Kontrolle
ob die Quadrate mod n eh kongruent sind?
    # alg. Quadrat MIT REDUKTION MOD f(x) berechnen

    # initialisieren der Ergebnisvariable des alg. Quadrats
    algQuadrat := 1:

    # aufmultiplizieren der linearen Polynome der gewählten Spalten
    for j in spaltenauswahl do

      paar := RelationenListe[j];
      a := paar[1];
      b := paar[2];

      algQuadrat := rem( algQuadrat * (a+b*x) , f(x) , x);

    end do: #for j in spaltenauswahl

    # dazumultiplizieren von f'(x)^2
    algQuadrat := rem( algQuadrat * (f_abgeleitet(x))^2 , f(x) , x):

    # Kontrolle, ob die Quadrate kongruent sind
    if (  modp( subs(x=m,algQuadrat) , n ) <> ratQuadrat ) then
      print("FEHLER:  DAS BILD DES ALGEBRAISCHEN QUADRATS UND DAS RATIONALE
QUADRAT LIEGEN MOD n NICHT IN DERSELBEN RESTKLASSE!!! ");
    end if;


    # ALGEBRAISCHE WURZEL ZIEHEN

    # OBERE SCHRANKE FÜR WURZELKOEFFIZIENTEN - siehe: Lecture Notes 1554,
Seite 99
```

```
    f_Norm := sqrt(add( (f_koeffs[i])^2 , i=1..f_koeffs_anzahl ));

    # obere Schranke für die Absolutbeträge (der Komponenten) der Paare
(a,b), die in der Auswahl der guten Relationen vorkommen
    u := max( seq( max( abs~( RelationenListe[i] ) ) ,
i=1..anzahlRelationenGefunden ) );

    # Anzahl der Paare (a,b) in der aktuellen Auswahl
    spaltenauswahl_anzahl := nops(spaltenauswahl);

    #b_0 := (d^(3/2) * f_Norm^(d-0) * (2 * u *
f_Norm)^(spaltenauswahl_anzahl/2));
    obereSchranke_Wurzelkoeffizienten := max(seq(
ceil(evalf[ceil(evalf(log[10]((d^(3/2) * f_Norm^(d-i) * (2 * u *
f_Norm)^(spaltenauswahl_anzahl/2))))*1.1)](   (d^(3/2) * f_Norm^(d-i) * (2 *
u * f_Norm)^(spaltenauswahl_anzahl/2))   )) , i=1..d ));

    # Suchen einer geeigneten Primzahl p für den alg.Wurzelschritt in
Z_p[x]/(f(x) mod p)
    p := 2*obereSchranke_Wurzelkoeffizienten + 1:

    p_gefunden := false:

    while (not(p_gefunden)) do

      p := nextprime(p);

      # Test ob p^d kongruent 3,5,7 mod 8 ist - wir wollen nur die leichten
Fälle
      rest := modp( modp(p,8)^d , 8 );
      if ( (modp(rest,4)=3)  or (rest=5) ) then

        # Test ob f(x) mod p irreduzibel ist
        if (Irreduc(f(x)) mod p) then
          print("passendes p gefunden !!!");
          p_gefunden := true;
          #print("p = ", p);
        end if; #if irreduc f(x) mod p

      end if; #if rest = ... ?

    end do: #while not(p_gefunden)

    # Konstruktion von GF(p^d)
    G := GF(p,d,f(x));

    # Reduktion des alg. Quadrats von Z[x]/f(x) nach Z_p[x]/(f(x) mod p)
    q := G:-ConvertIn(algQuadrat);


    # 1. Fall: p^d = 3 mod 4
    if ( modp(rest,4)=3 ) then

      # Berechnung der beiden alg. Wurzeln für den Fall: p^d = 3 mod 4
      r1 := G:-`^`(q,(p^d+1)/4);
      r2 := G:-`-`(r1);

    elif ( rest=5 ) then
    # 2. Fall: p^d = 5 mod 8

      if ( G:-`^`(q,(p^d-1)/4) = G:-one ) then
```

```
        #print("5 mod 8: 1. Fall");
        r1 := G:-`^`(q,(p^d+3)/8);
        r2 := G:-`-`(r1);
      else
        #print("5 mod 8: 2. Fall");
        r1 := G:-`*`(  G:-`^`(G:-ConvertIn(2),(p-1)/4)  ,
G:-`^`(q,(p^d+3)/8)  );
        r2 := G:-`-`(r1);
      end if;

    else
      print("FEHLER: p^d ist nicht kongruent 3,5,7 mod 8 !!!");
    end if;


    # Probe, ob die beiden Wurzeln zum Quadrat eh das ursprüngliche Polynom
(in GF(p^d) ergeben)
    if ( (G:-`^`(r1,2) <> q) or (G:-`^`(r2,2) <> q) ) then
      print("FEHLER: die Probequadrierung (innerhalb GF_p^d) hat nicht auf
das ursprüngliche Quadrat geführt !!!");
    end if;

    # Konvertieren der beiden Wurzeln aus GF(p^d) in ein "normales" Polynom
und herausholen der Koeffizienten der Polynome
    R1 := [coeffs(G:-ConvertOut(r1))];
    R2 := [coeffs(G:-ConvertOut(r2))];

    # Reduktion der Koeffizienten der alg. Wurzeln MODULO n !!!
    S1 := mods(R1,n);
    S2 := mods(R2,n);

    # Aufstellen der beiden Wurzelpolynome als Polynome in Z[x]/f(x)
    T1 := unapply( add( x^(i-1) * S1[i] , i=1..d ) , x );
    T2 := unapply( add( x^(i-1) * S2[i] , i=1..d ) , x );

    # Abbilden der beiden Wurzeln nach Z_n
    algWurzel1 := T1(m) mod n;
    algWurzel2 := T2(m) mod n;

    # Kontrolle, welches Quadrat (welcher Wurzel) mit dem rationalen Quadrat
übereinstimmt (mod n)?

    wurzeln_getestet := select( x -> evalb( modp(x^2,n) = ratQuadrat ) ,
[algWurzel1, algWurzel2] );
    if ( nops(wurzeln_getestet)=0 ) then
      print("Warnung: beim ", anzahl_versuche, "-ten Lösungsbasisvektor von
", lösungsdimension, " war keines der beiden Quadrate der abgebildeten
algebraischen Wurzeln mod n kongruent dem rationalen Quadrat!!!");
    end if;

    # für jede algebraische Wurzel (mod n), deren Quadrat mit dem rationalen
Quadrat übereinstimmt, versuche n zu faktorisieren
    for algWurzel in wurzeln_getestet do

      differenz := modp( algWurzel - ratWurzel , n );
      ggt := igcd( differenz , n );
      if ( (ggt>1) and (ggt<n) ) then
        print("Faktorisierung war ERFOLGREICH (differenz) !!!!!     Die
beiden Faktoren von n lauten:   ", [ggt, n/ggt]);
        faktorisierungErfolgreich := true;
        Faktoren := sort([ggt, n/ggt]);
        return [ggt, n/ggt];
```

```
        end if; #if 1 < ggt < n

        summe := modp( algWurzel + ratWurzel , n );
        ggt := igcd( summe , n );
        if ( (ggt>1) and (ggt<n) ) then
          print("Faktorisierung war ERFOLGREICH (summe) !!!!!     Die beiden
   Faktoren von n lauten:   ", [ggt, n/ggt]);
          faktorisierungErfolgreich := true;
          Faktoren := sort([ggt, n/ggt]);
          return [ggt, n/ggt];
        end if; #if 1 < ggt < n

      end do; #for algWurzel in wurzeln_getestet

      if (not(faktorisierungErfolgreich)) then
        print("Der ", anzahl_versuche, "-te Lösungsbasisvektor von ",
   lösungsdimension, "war erfolglos.");
      end if; #if not(faktorisierungErfolgreich)

    end do: #for anzahl_versuche = 1 .. lösungsdimension


  end proc;
```

GNFS

```
> GNFS := proc(n::posint, Faktor_AFB_zu_RFB := 3, anzahlRelationenZusätzlich :=
  20, QCB_Anzahl := 100, ausgabe_r := 10^7, ausgabe_p := 50)

  # Beschreibung der Eingabeparameter:

  # n ............................ zu faktorisierende Zahl
  # Faktor_AFB_zu_RFB ............. Faktor Größenunterschied der oberen Schranken
  (!) von AFB zu RFB
  # anzahlRelationenZusätzlich ... Anzahl der Relationen, die mehr als
  eigentlich notwendig zu suchen sind

  # QCB_Anzahl ................... Größe der quadratischen Charaktere Basis QCB
  # ausgabe_r .................... Anzahl der getesteten "r", nach denen bei der
  Berechnung der Mengen R(p) eine (Bildschirm-)Ausgabe erfolgen soll
  # ausgabe_p .................... Anzahl der getesteten "p", nach denen bei der
  Berechnung der Mengen R(p) eine (Bildschirm-)Ausgabe erfolgen soll


  # Variablendeklaration für "Setup"

  local Rechengenauigkeit;              # Genauigkeit für Maple-Funktion
  "evalf[..]"
  local trivial::boolean := false;      # Indikator, ob eine triviale
  Faktorisierung von n möglich ist
  local trivial_Faktoren;               # Faktoren von n, falls n schon bei den
  Vortests auch ohne dem Zahlkörpersieb faktorisiert werden kann
  local d;                              # Polynomgrad von f(x)
  local m;                              # f(m)=n
  local f;                              # Polynom f(x)
  local f_koeffs;                       # Koeffizienten des Polynoms f(x)
  local f_koeffs_anzahl;                # Anzahl der Koeffizienten des Polynoms
  f(x)
  local f_abgeleitet;                   # Ableitung von f(x) nach x
```

```
  # Variablendeklaration für "Faktorbasen aufstellen"

  local L;                              # Laufzeitfunktion L_n[u,v]

  local RFB_Schranke;                   # obere Schranke für Primzahlen in RFB
  local RFB_p;                          # Primzahlen in rationaler Faktorbasis
  local RFB_Anzahl;                     # Anzahl der Primzahlen in RFB
  local RFB;                            # rationale Faktorbasis - als Liste von
Paaren (p,r)
  local RFB_max;                        # größte Primzahl in der rationalen
Faktorbasis
  local AFB_Schranke;                   # obere Schranke für die Primzahlen in
AFB
  local AFB_p_möglich;                  # potentielle (!) Primzahlen für die
AFB
  local AFB_Anzahl_möglicher_r;         # Anzahl der in Frage kommenden "r" für
Fortschrittsanzeige bei AFB-Berechnung
  local AFB_pEinfach_rListe;            # AFB in der Form: für jede Primzahl in
AFB gibt es nur eine(!) Liste mit allen passenden r dazu
  local AFB_pMehrfach_rEinfach;         # AFB in der Form: für jedes Paar [p,r]
gibt es einen eigenen Eintrag
  local AFB_p_möglich_Anzahl;           # Anzahl der potentiellen(!) Primzahlen
in AFB
  local AFB_p_Anzahl;                   # Anzahl der Primzahlen(!) in AFB
  local AFB_Anzahl;                     # Anzahl der Paare [p,r] in AFB - p
mehrfach, r einfach

  local QCB_p_getestet;                 # Primzahlen, die für die QCB zumindest
getestet (vielleicht aber auch nicht aufgenommen) wurden
  local QCB;                            # quadratische Charaktere-Basis als
Paare [q,s]

  # Variablendeklaration für Probedivision

  local alle_möglichen_Primzahlen;

  # Variablendeklaration für Vorbereitungen fürs Sieben

  local anzahlRelationenNotwendig;      # Anzahl der zu suchenden Relationen (=
"unbedingt notwendig" + "20 (oder vom User eingegebene Anzahl) zusätzliche
Relationen zur Sicherheit")
  local Schleifenmatrix_AnzahlZeilen;   # Anzahl der Einträge pro Relation
  local C;                              # Schranke für Siebintervall

  # Variablendeklaration für Sieben

  local RelationenMatrix;               # speichert die Exponenten der
Zerlegungen über den Basen mod 2
  local RelationenListe;                # speichert die guten gefundenen
Relationen (a,b)
  local anzahlRelationenGefunden;       # Anzahl der guten gefundenen
Relationen
  local B;                              # boolsches Array
  local Schleifenmatrix;                # speichert für ein Siebintervall einen
Teil der Einträge für die Relationenmatrix
  local Siebintervall;                  # [-C..C]

  # Variablendeklaration für Matrix-Schritt und Lösungsversuche

  local lösungsbasis;                   # linear unabhängige Lösungsvorschläge
für linear abhängige Teilmengen der Spalten von RelationenMatrix
```

```
  local Faktoren;                          # Endergebnis: die beiden Faktoren des
Eingabeparameters n



  ##### SETUP #####



  # Rechengenauigkeit für Maple festlegen
  setup_rechengenauigkeit(n, Rechengenauigkeit);


  # Testen, ob n gerade ist?
  setup_vortests_gerade(n, 'trivial', trivial_Faktoren);
  # Falls ja, dann Rückgabe der Faktoren
  if (trivial) then

    return (trivial_Faktoren);
  end if;

  # Testen, ob n eine Potenz ist?
  setup_vortests_potenz(n, Rechengenauigkeit, 'trivial', trivial_Faktoren);

  # Falls ja, dann Rückgabe der Faktoren
  if (trivial) then

    return (trivial_Faktoren);
  end if;

  # Testen, ob n groß genug für das Kriterium "n > d^(2d^2)" mit d=3 ist?
  setup_vortests_zuklein(n, d, 'trivial', trivial_Faktoren);

  # Falls nein, dann Rückgabe der Faktoren durch "ifactors"
  if (trivial) then
    return (trivial_Faktoren);
  end if;

  # Polynomgrad d bestimmen (ungerade, >=3)
  setup_polynomgrad(n, d);

  # m bestimmen
  setup_m(n, d, Rechengenauigkeit, m);

  # Polynom f(x) bestimmen - mittels base-m-Methode
  setup_polynom(n, m, f_koeffs, f_koeffs_anzahl, f);

  # Tests mit f(x) machen
  setup_polynom_tests(n, d, m, f, f_koeffs, f_koeffs_anzahl, f_abgeleitet,
'trivial', trivial_Faktoren);
  if (trivial) then
    return (trivial_Faktoren);
  end if;

  # homogenisiertes Polynom F(x,y) aufstellen
  setup_homogenesPolynom(f_koeffs, f_koeffs_anzahl, F);



  ##### FAKTORBASEN AUFSTELLEN #####
```

```
  # Laufzeitfunktion L_n[u,v] bestimmen
  fb_laufzeitfunktion(n, L);


  ### RFB ###


  # obere Schranke für die RFB berechnen
  fb_rfb_schranke(L, Rechengenauigkeit, RFB_Schranke);

  # Primzahlen in der RFB berechnen
  fb_rfb_primzahlen(RFB_Schranke, RFB_p);

  # Anzahl der Elemente in der RFB berechnen
  fb_rfb_anzahl(RFB_p, RFB_Anzahl);

  # Paare in RFB berechnen
  fb_rfb_paare(RFB_p, m, RFB);

  # größtes Element in der rationalen Faktorbasis berechnen
  fb_rfb_maximum(RFB_p, RFB_max);


  ### AFB ###


  # obere Schranke für Primzahlen in AFB
  fb_afb_schranke(RFB_Schranke, Faktor_AFB_zu_RFB, AFB_Schranke);

  # potentielle (!) Primzahlen für AFB berechnen
  # bisherie Primzahlen aus RFB hernehmen und nur die neuen (!) hinzufügen, die
nicht schon in RFB vorkommen
  fb_afb_primzahlen(RFB_Schranke, AFB_Schranke, RFB_p, AFB_p_möglich);

  # Hilfsfunktion zur Fortschrittsanzeige beim Berechnen der R(p) = { 0 <= r <=
p-1  |  f(r) = 0 mod p }
  fb_afb_anzeige(AFB_p_möglich, AFB_Anzahl_möglicher_r, AFB_p_möglich_Anzahl);

  # R(p) berechnen
  fb_afb_R_p_berechnen(AFB_p_möglich, AFB_p_möglich_Anzahl,
AFB_Anzahl_möglicher_r, f, d, ausgabe_r, ausgabe_p, AFB_pEinfach_rListe,
AFB_p_Anzahl, AFB_pMehrfach_rEinfach, AFB_Anzahl);


  ### QCB ###


  fb_qcb(AFB_Schranke, QCB_Anzahl, f, f_abgeleitet, QCB_p_getestet, QCB);



  ##### Probedivision mit Primzahlen aus den Faktorbasen #####


  probedivision(AFB_p_möglich, QCB_p_getestet, n, alle_möglichen_Primzahlen,
'trivial', trivial_Faktoren);
  if (trivial) then

    return (trivial_Faktoren);
  end if;
```

```
  ##### Vorbereitungen fürs Sieben #####


  # Anzahl der zu suchenden Relationen berechnen
  siebvorbereitungen_anzahlRelationenNotwendig(RFB_Anzahl, AFB_Anzahl,
QCB_Anzahl, anzahlRelationenZusätzlich, anzahlRelationenNotwendig,
Schleifenmatrix_AnzahlZeilen);

  # Schranke für Siebbereich (für a) festlegen
  siebvorbereitungen_siebschranke(alle_möglichen_Primzahlen, m, L,
Rechengenauigkeit, C);


  ##### SIEBEN #####


  # Variableninitialisierung fürs Sieben
  sieben_variableninitialisierung(RelationenMatrix,
Schleifenmatrix_AnzahlZeilen, anzahlRelationenNotwendig, C, Siebintervall);

  # tatsächliches Sieben
  sieben(Schleifenmatrix_AnzahlZeilen, anzahlRelationenNotwendig, C, m, F,
RFB_Anzahl, AFB_Anzahl, QCB_Anzahl, RFB, AFB_pMehrfach_rEinfach, QCB,
Siebintervall, RelationenMatrix, RelationenListe, anzahlRelationenGefunden);


  ##### MATRIX-SCHRITT: LINEAR ABHÄNGIGE TEILMENGE SUCHEN UND
DURCHPROBIEREN#####


  matrixschritt(RelationenMatrix, lösungsbasis);
  lösungen_probieren(lösungsbasis, Schleifenmatrix_AnzahlZeilen,
RelationenMatrix, RelationenListe, RFB, RFB_Anzahl, n, m, d, f, f_koeffs,
f_koeffs_anzahl, f_abgeleitet, anzahlRelationenGefunden, Faktoren);

  return Faktoren;




  #dummy;

end proc;
```

Programmaufruf

n zufällig erzeugen

Stelligkeiten festlegen (Dezimalstelligkeit)

Stelligkeit der 1. Primzahl p festlegen

```
> #e1 := 3;
  #e1 := 4;
  e1 := 5;
  #e1 := 6;
  #e1 := 7;
```

Stelligkeit der 2. Primzahl q festlegen

```
> if e1=5 then
    e2:=6
  elif e1=3 then
    e2:=5
  else
    e2:=e1+10
  end if;
```

Zufallsfunktionen erzeugen

```
> rollp := rand(10^e1..10^(e1+1)):
> rollq := rand(10^e2..10^(e2+1)):
```

Zufallszahlen erzeugen

```
> #P := numtheory[safeprime](rollp());
  P := nextprime(rollp());
> #Q := numtheory[safeprime](rollq());
  Q := nextprime(rollq());
```

*Produkt n=p*q berechnen*

```
> n := P*Q;
```

Programmaufruf

```
> GNFS(n);
```

Anhang B

Programmcode - Index calculus

Index calculus - Teilprogramme

Setup

p prim?

```
> setup_Primalität := proc(p::posint)::boolean;

    global Fehler, Fehlerbeschreibungen;

    if isprime(p) then
      #print("p ist (ziemlich sicher) eine Primzahl");
      return true;
    else
      print("p ist keine Primzahl");
      Fehler := true;
      Fehlerbeschreibungen := [ op(Fehlerbeschreibungen) , ["das eingegebene
  p", p, "ist keine Primzahl"] ];
      return false;
    end if;

  end proc;
```

g erzeugendes Element?

```
> setup_Erzeuger := proc(p::posint, g::posint)::boolean;

    global Fehler, Fehlerbeschreibungen;

    if (numtheory[order](g,p) = p-1) then
      #print("g =", g, "ist ein Erzeuger von Z_p*");
      return true;
    else
      print("g =", g, "ist KEIN Erzeuger von Z_p*");
      Fehler := true;
      Fehlerbeschreibungen := [ op(Fehlerbeschreibungen) , ["das eingegebene g
  =", g, "ist kein Erzeuger von Z_p* mit p =", p] ];
    end if;

  end proc;
```

DL von alpha trivial oder unmöglich, dh. alpha = 0,1,g?

```
> setup_trivial := proc(p::posint, g::posint, alpha::integer, DL)::boolean;

    global Fehler, Fehlerbeschreibungen;

    if (alpha=0) then
      print("von alpha =", alpha, "; und von Null kann kein disk. Log.
  berechnet werden");
      Fehler := true;
      Fehlerbeschreibungen := [ op(Fehlerbeschreibungen) , ["das eingegebene
  alpha =", alpha, "ist gleich Null und von Null kann kein disk. Log. berechnet
  werden"] ];
      return false;
    elif (alpha=1) then
      print("vom eingegebene alpha =", alpha, "kann man ohne Rechnung den DL=0
  berechnen");
```

```
      DL := 0;
      return true;
    elif (alpha=g) then
      print("das eingegebene alpha =", alpha, "stimmt mit dem erzeugenden
  Element g =", g, "überein, weshalb man ohne weitere Rechnung DL=1 angeben
  kann");
      DL := 1;
      return true;
    end if;

  end proc;
```

Schranke für Faktorbasis berechnen

```
> setup_Schranke := proc(p::posint, Basis_Schranke);

    #Basis_Schranke := max( floor(sqrt(p)/10) , 3);
    #Basis_Schranke := max( floor(10*ln(p)^3) , 3);
    Basis_Schranke := max( floor(p^(0.35)) , 3);

  end proc;
```

Elemente der Faktorbasis berechnen

```
> setup_Faktorbasis := proc(Basis_Schranke::posint, Basis, Basis_Anzahl)

    local tempBasis := [ -1, op(select(isprime,[$2..Basis_Schranke])) ];

    Basis := tempBasis;
    Basis_Anzahl := nops(tempBasis);

  end proc;
```

Kongruenzensystem für DL der Faktorbasis aufstellen

```
> faktorbasis_System_aufstellen := proc(Basis, Basis_Anzahl, Relationenliste,
  Relationenmatrix, AnzahlRelationenZusätzlich)

    local zufallszahl := rand(1..p-2);
    local AnzahlRelationenNotwendig := Basis_Anzahl + AnzahlRelationenZusätzlich;
    local AnzahlRelationenGefunden := 0;
    local Schleifenliste;
    local e, E, E_temp, vielfachheit, p_akt, indizes, ggt;
    local i;

    # leer initialisieren
    Relationenliste := Vector(1..Basis_Anzahl);
    Relationenmatrix := Matrix(Basis_Anzahl + AnzahlRelationenZusätzlich,
  Basis_Anzahl, storage=sparse);

    # solange suchen, bis genügend Relationen gefunden wurden
    while (AnzahlRelationenGefunden < AnzahlRelationenNotwendig) do

      # hier werden für einen einzigen Versuch die Vielfachheiten der Primzahlen
  aus der Faktorbasis gespeichert
      Schleifenliste := Array(1..Basis_Anzahl, storage=sparse);
```

```
    # zufälligen Exponenten wählen
    e := zufallszahl();
    E := mods(g &^ e, p);
    E_temp := E;

    # Vorzeichen bestimmen
    if (E_temp < 0) then
      Schleifenliste[1] := 1;
      E_temp := abs(E_temp);
    else
      Schleifenliste[1] := 0;
    end if;

    # durch alle Primzahlen der Faktorbasis durchdividieren
    for i from 2 to Basis_Anzahl do

      p_akt := Basis[i];

      # Zähler der Vielfachheit des Primfaktors "p_akt" in "E" bzw. "E_temp"
      vielfachheit := 0;

      # Bestimmung der Vielfachheit des Primfaktors "p_akt" in "E" bzw.
"E_temp"
      ggt := igcd(p_akt, E_temp);
      while (ggt = p_akt) do
        E_temp := E_temp/p_akt;
        vielfachheit := vielfachheit + 1;
        ggt := igcd(p_akt, E_temp);
      end do;

      # wenn Vielfachheit eines Primfaktors > 0 ist ...
      if (vielfachheit > 0) then
        # ... dann merken der Vielfachheit
        Schleifenliste[i] := vielfachheit;
        if (E_temp = 1) then
          break;
        end if;
      end if;

    end do; #for i = 2..Basis_Anzahl

    # Wenn "E" über der Faktorbasis zerfällt, ...
    if (E_temp = 1) then

      # ... dann nur als neu eintragen, wenn das konkrete "e" noch nicht bekannt
ist bzw.
      if (member(e,[entries(Relationenliste, 'nolist')])) then
        # doch nicht eintragen in Relationenliste, sondern weitersuchen
        next;
      end if;

      # Erhöhen des Zählers
      AnzahlRelationenGefunden := AnzahlRelationenGefunden + 1;

      # Übertragen des Exponentenvektors vom temprären Schleifenspeicher in die
Relationenmatrix
      indizes := lhs~(op(Schleifenliste)[2]);
      for i in indizes do
        Relationenmatrix[AnzahlRelationenGefunden,i]:= Schleifenliste[i];

      end do; #for i in indizes
```

```
        # Eintragen des glatten Wertes in die Relationenliste
        Relationenliste(AnzahlRelationenGefunden) := e;

        #print("Relation gefunden: ", AnzahlRelationenGefunden, "/",
    AnzahlRelationenNotwendig,
    "=",evalf[4](AnzahlRelationenGefunden/AnzahlRelationenNotwendig*100),  "%, e=",
    e, "E=", E, "=", ifactor(E));

      end if; #if E_temp = 1

    end do; #while noch nicht genug gefunden

  end proc;
```

DL für Faktorbasis berechnen

```
> faktorbasis_System_lösen := proc(p::posint, Relationenmatrix::Matrix,
  Relationenliste::Vector, Basis_Anzahl, Basis_DL)

    local r := LinearAlgebra[RowDimension](Relationenmatrix);
    local c := LinearAlgebra[ColumnDimension](Relationenmatrix);
    local i, j;         # Schleifenlaufvariable
    local Ergebnis;     # speichert das Ergebnis des Maple-Aufrufs "msolve"
    local glg, var;

    i := 'i';
    j := 'j';
    Ergebnis :=
  msolve({seq(add(Relationenmatrix[i,j]*temp[j],j=1..c)=Relationenliste[i],i=1..r)
  },p-1);

    # für jedes Teilergebnis von "msolve" ...
    i := 'i';
    for glg in Ergebnis do

      var := lhs(glg);

      # nur eintragen, wenn eine echte ganze Zahl als Ergebnis herauskommt
      if (type(rhs(glg),integer)) then
        Basis_DL[op(var)] := rhs(glg);
        #print(Basis_DL[op(var)]);
      end if;

    end do;

  end proc;
```

jene (Indizes der) Faktorbasiselemente herausfiltern, für die ein DL tatsächlich bekannt ist

```
> faktorbasis_nur_berechnete := proc(Basis_DL, Indizes_echt)

    Indizes_echt := sort([indices(Basis_DL,'nolist')],`<`);

  end proc;
```

konkreten DL für eingegebenes Element berechnen

```
> DL_berechnen := proc(Basis, Basis_DL, Basis_Anzahl, Indizes_echt, DL)

    local zufallszahl := rand(0..p-2);
    local zerfällt := false;
    local e;                              # zufälliger Exponent
    local E;                              # zufälliges Element g^e
    local beta;                           # beta = alpha * g^e
    local beta_temp;                      # Arbeitsvariable beim Testen der
  Glattheit von beta
    local i, j;                           # Schleifenvariable
    local Exponentenliste;                # enthält die Vielfachheiten der
  Primfaktoren
    local vielfachheit;                   # Schleifenvariable
    local ggt;                            # Hilfsvariabel beim Testen auf
  Glattheit
    local p_akt;                          # Primzahlen beim Testen auf Glattheit
    local Ergebnis;                       # speichert das Endergebnis

    # solange kein passender Exponent gefunden wurde, ...
    while (not(zerfällt)) do

      # Liste der Exponenten leer initialisieren
      Exponentenliste := Array(1..Basis_Anzahl, storage=sparse);

      # zufälligen Exponenten wählen
      e := zufallszahl();                 # zufälligen Exponenten wählen

      #berechnung des zufälligen Elements sowie des maskierten Eingabeelements
      E := modp(g&^e,p);
      beta := mods(alpha * E, p);
      beta_temp := beta;

      # Vorzeichen bestimmen
      if (beta_temp < 0) then
        Exponentenliste[1] := 1;
        beta_temp := abs(beta_temp);
      end if;

      # für alle Faktorbasiselemente, für die tatsächlich ein diskreter Logarthmus
  berechnet werden konnte (!), ...
      for i in Indizes_echt do

        p_akt := Basis[i];
        vielfachheit := 0;

        # Teilbarkeit durch den aktuellen Primfaktor "p_akt" testen
        ggt := igcd(p_akt, beta_temp);
        while (ggt = p_akt) do
          beta_temp := beta_temp/p_akt;
          vielfachheit := vielfachheit + 1;
          ggt := igcd(p_akt, beta_temp);
        end do;

        # wenn das masierte Element durch "p_akt" teilbar ist, ...
        if (vielfachheit > 0) then
          Exponentenliste[i] := vielfachheit;
          if (beta_temp = 1) then

            zerfällt := true;
            #print("passendes Zufallselement gefunden: e=", e, " E=", E, " beta=",
  beta, "=", ifactor(mods(beta,p)));
```

```
            # Berechnung des gewünschten diskreten Logarithmus'
            Ergebnis := modp( add(Exponentenliste[j]*Basis_DL[j],j in
   Indizes_echt) - e ,p-1);
            DL := Ergebnis;

            break;
          end if; #beta_temp = 1

        end if; #vielfachheit > 0

      end do; #for i in Indizes_echt

    end do; #while(not(zerfällt))

  end proc;
```

Index calculus - Gesamtprogramm

```
> ic := proc(p::posint, g::posint, alpha::integer, AnzahlRelationenZusätzlich := 5)

  global Fehler := false;
  global Fehlerbeschreibungen := "";

  local DL;                # Endergebnis

  local Basis_Schranke;    # obere Schranke für die Faktorbasis
  local Basis;             # Faktorbasis = [-1, 2, 3, 5, ... ]
  local Basis_Anzahl;      # Anzahl der Elemente in der Faktorbasis (inkl. "-1")
  local Basis_DL;          # diskrete Logarithmen der Elemente der Faktorbasis

  local Relationenmatrix;  # speichert für jeden Exponenten e die Primfaktoren von
g^e mod p
  local Relationenliste;   # speichert die Exponenten e, sodass g^e mod p über der
Faktorbasis zerfällt

  local Indizes_echt;      # Indizes jener Faktorbasiselemente, für die tatsächlich
ein konkreter diskreter Logarithmus berechnet werden konnte


  setup_Primalität(p);
  setup_Erzeuger(p,g);
  setup_trivial(p,g,alpha,DL);

  # wenn ein Fehler/Trivialfall aufgetreten ist, dann ausgabe der
Fehlerbeschreibung am Bildschirm
  if Fehler then
    print("FEHLER: ", Fehlerbeschreibungen);
    return [false, Fehlerbeschreibungen];
  end if;

  setup_Schranke(p, Basis_Schranke);
  setup_Faktorbasis(Basis_Schranke, Basis, Basis_Anzahl);
  #print("Basis_Anzahl=",Basis_Anzahl);
  #print("Basis=", Basis);

  #print("Zeit FB Rel suchen: ", time(faktorbasis_System_aufstellen(Basis,
Basis_Anzahl, Relationenliste, Relationenmatrix, AnzahlRelationenZusätzlich)));
  faktorbasis_System_aufstellen(Basis, Basis_Anzahl, Relationenliste,
Relationenmatrix, AnzahlRelationenZusätzlich);
```

```
  #print("Zeit FB berechnen: ",
time(faktorbasis_System_lösen(p,Relationenmatrix,Relationenliste,Basis_Anzahl,Basis
_DL)));

faktorbasis_System_lösen(p,Relationenmatrix,Relationenliste,Basis_Anzahl,Basis_DL);


  # übernimmt nur jene Faktorbasiselemente, für die tatsächlich ein DL berechnet
werden konnte, in die künftig verwendetet Basis
  faktorbasis_nur_berechnete(Basis_DL, Indizes_echt);

  #print("Zeit konkret: ", time(DL_berechnen(Basis, Basis_DL, Basis_Anzahl,
Indizes_echt, DL)));
  DL_berechnen(Basis, Basis_DL, Basis_Anzahl, Indizes_echt, DL);

  return DL;

end proc;
```

Index calculus - Programmaufruf mit zufälligen Werten und Kontrolle des Ergebnisses

```
> p := nextprime(rand(10^10)());
  print("Stelligkeit: ", evalf[5](log[10](%)));
  g := numtheory[primroot](p);
  alpha := g+1;
  ERGEBNIS := ic(p,g,alpha,10);
  PROBE := modp(g&^ERGEBNIS, p);
  DIFFERENZ := alpha - PROBE;
```

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Literaturverzeichnis

[Bau97] Bauer, Friedrich L.: *Entzifferte Geheimnisse - Methoden und Maximen der Kryptologie.* Springer-Verlag, Zweite erweiterte Auflage, 1997.

[BB05] Bakirli, Nilgün und Hüsna Barlak: *RSA - Asymmetrische Konzelation & Digitale Signatur.* WS 2004/05.

[BD97] Boneh, Dan und Glenn Durfee: *New Results on the Cryptanalysis of Low Exponent RSA (Extend Abstract, Preprint).* 1997.

[BDF98] Boneh, D., G. Durfee und Y. Frankel: *An attack on RSA given a fraction of the private key bits.* In: *AsiaCrypt '98*, Band 1514 der Reihe *Lecture Notes in Computer Science*, Seiten 25–34. Springer-Verlag, 1998.

[Ble98] Bleichenbacher, Daniel: *Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS # 1.* In: *CRYPTO '98*, Band 1462 der Reihe *Lecture Notes in Computer Science*, Seiten 1–12. Springer-Verlag, 1998.

[Bon98] Boneh, Dan: *Twenty Years of Attacks on the RSA Cryptosystem.* dabo@cs.standord.edu, 1998.

[Bri98] Briggs, Matthew E.: *An Introduction to the General Number Field Sieve.* Master's thesis, 17. April 1998.

[Buc10] Buchmann, Johannes: *Einführung in die Kryptographie.* Springer-Verlag, Fünfte Auflage, 2010.

[CFPR96] Coppersmith, D., M. Franklin, J. Patarin und M. Reiter: *Low-exponent RSA with related messages.* In: *EUROCRYPT '96*, Band 1070 der Reihe *Lecture Notes in Computer Science*, Seiten 1–9. Springer-Verlag, 1996.

[Cop93] Coppersmith, Don: *Solving linear equations over GF(2): Block Lanczos algorithm.* In: *Linear Algebraand its Applications 192*, Seiten 33–60. 1993.

[Cop94] Coppersmith, Don: *Solving homogeneous linear equations over GF(2) via Block Wiedemann algorithm.* In: *Mathematics of Computation 62*, Nummer 205, Seiten 333–350. 1994.

[Cop97] Coppersmith, Don: *Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities.* Band 10 der Reihe *Journal of Cryptology*, Seiten 233–260. International Association for Cryptologic Research, 1997.

[Cou] Couveignes, Jean-Marc: *Computing a square root for the number field sieve.* In: *The development of the number field sieve*, Band 1554 der Reihe *Lecture Notes in Mathematics*, Seiten 95–102.

[CP00] Crandall, Richard und Carl Pomerance: *Prime Numbers - A Computational Perspective*. Springer Verlag, 2000.

[dW02] Weger, Benne de: *Cryptanalysis of RSA with Small Prime Difference*. In: *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, Band 13, Seiten 17–28. Springer-Verlag, 2002.

[Geh07] Gehrs, Kai: *Elementare Kryptographie*. Paderborn, 9. Juli 2007.

[Hås88] Håstad, Johan: *Solving Simultaneous Modular Equations of Low Degree*. In: *SIAM Journal on Computing*, Band 17, Nr. 2, Seiten 336–341, 1988.

[Jen05] Jensen, Per Leslie: *Integer Factorization*. Master Thesis, Department of Computer Science, University of Copenhagen, 2005.

[KK10] Karpfinger, Christian und Hubert Kiechle: *Kryptologie - Algebraische Methoden und Algorithmen*. Vieweg+Teubner, GWV Fachverlage GmbH, 1. Auflage, 2010.

[KL08] Katz, Jonathan und Yehuda Lindell: *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[Lj93] Lenstra, A. K. und H. W. Lenstra jun. (Herausgeber): *The development of the number field sieve*, Band 1554 der Reihe *Lecture Notes in Mathematics*. Springer-Verlag, 1993.

[LO91] LaMacchia, B. A. und A. M. Odlyzko: *Solving Large Sparse Linear Systems Over Finite Fields*. In: *Advances in Cryptology - Proceedings of Crypto '90*, Band 537 der Reihe *Lecture Notes in Computer Science*, Seiten 72–82. Springer-Verlag, New York, 1991.

[LZWD09] LUO, Ping, HaiJian ZHOU, DaoShun WANG und YiQi DAI: *Cryptanalysis of RSA for a special case with $d > e$*. In: *Science in China Series F: Information Sciences*, Band 52, Seiten 609–616. Springer-Verlag, April 2009.

[MvOV97] Menezes, Alfred J., Paul C. von Oorschot und Scott A. Vanstone: *Handbokk of applied cryptography*. Discrete mathematics and its applications. CRC Press, Inc., 1997.

[Odl85] Odlyzko, A.: *Discrete logarithms in finite fields and their cryptographic significance*. In: *Advances in Cryptology, Proc. Eurocrypt '84*, Band 209 der Reihe *Lecture Notes in Computer Science*, Seiten 224–313. 1985.

[Pom03] Pommerening, Klaus: *Asymmetrische Verschlüsselung*. Saarstraße 21, D-55099 Mainz, 9. August 2003.

[PS92] Pomerance, Carl und J.W. Smith: *Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes*. In: *Experimental Mathematics*, Band 1, Seiten 89–94. 1992.

[RL02] RSA-Laboratories: *PKCS #1 v2.1: RSA Cryptography Standard*, 14. Juni 2002.

[RSA78] Rivest, R. L., A. Shamir und L. Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Band 21 der Reihe *Communications of the ACM*, Seiten 120–126. Februar 1978.

[SDO06] Salah, Imad Khaled, Abdullah Darwish und Saleh Oqeili: *Mathematical Attacks on RSA Cryptosystem.* Band 8 der Reihe *Jornal of Computer Science 2*, Seiten 665–671. Science Publications, 2006.

[SS02] Schmidt-Samoa, Katja: *Das* Number Field Sieve - *Entwicklung, Varianten und Erfolge.* Diplomarbiet, Universität Kaiserslautern, März 2002.

[Stö07] Stöffler, Christian: *Faktorisieren mit dem General Number Field Sieve.* Diplomarbeit, Universität Mannheim, Fakultät für Informatik, ADRESS, 11. Januar 2007.

[Sti06] Stinson, Douglas R.: *Cryptography - Theory and Practice.* Chapman & Hall/CRC, 3. Auflage, 2006.

[VvT97] Verheul, Eric R. und Henk C. A. van Tilborg: *Cryptanalysis of 'Less Short' RSA Secret Exponents*, Band 8 der Reihe *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, Seiten 425–435. Springer-Verlag, 7. März 1997.

[Wie90] Wiener, Michael J.: *Cryptanalysis of short RSA secret exponents*, Seiten 553–558. 1990.

[Wie99] Wiesenbauer, Johann: *Public Key Kryptosysteme in Theorie und Programmierung.* Didaktikhefte der Österreichischen Mathematischen Gesellschaft, 30, 1999.

[Zay91] Zayer, Jörg: *Die Theorie des Number Field Sieve.* Diplomarbeit, Universität des Saarlandes, Saarbrücken, Mai 1991.

[Zay95] Zayer, Jörg: *Faktorisieren mit dem Number Field Sieve.* Dissertation, Universität des Saarlandes, 1995.

Index

math. Symbole

A

B

C

D

E

F

G

H

Q

R

S

T

V

W

Z