Gitterbasierende Kryptosysteme

Georg Schönberger

BACHELORARBEIT

Nr. 239-006-024-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Computer- und Mediensicherheit

in Hagenberg

im März 2009

Diese Arbeit entstand im Rahmen des Gegenstands

Kryptosysteme und digitale Signaturen

im

Wintersemester 2008/09

Betreuer:

Jürgen Fuß

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 24. August 2014

Georg Schönberger

Inhaltsverzeichnis

Kurzfassung

Ursprünglich waren Gitter ein wichtiges Werkzeug der Kryptoanalyse und wurden unter anderem dazu verwendet, um kryptographische Verfahren anzugreifen. Mit der Erforschung der Schwierigkeit von Gitterproblemen entwickelten sie sich jedoch auch zur Grundlage von kryptographischen Public-Key-Systemen und anderen Verfahren.

Der „IEEE P1363.1TM/D9 draft standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices“ spezifiziert drei Teile, mit denen ein Kryptosystem, dessen Sicherheit auf Gitterproblemen basiert, implementiert werden kann: (1) eine Referenz von Techniken für Applikationen, (2) den entsprechenden mathematischen Hintergrund und (3) eine Abhandlung über auftretende sicherheitsrelevante Themen. Auch wenn bei der Implementierung eigentlich nur mit Polynomen gearbeitet wird, lässt sich ein Zusammenhang zu Gittern herstellen.

Die Sicherheit des Systems beruht dabei auf der Schwierigkeit der Lösung bekannter Gitterprobleme, wie z. B. dem *closest vector problem* (CVP) oder dem *shortest vector problem* (SVP). Viele Arbeiten zeigten bereits, dass es schwierig ist, Lösungen für ein CVP oder ein SVP zu berechnen. Genau diese Schwierigkeit nutzt auch NTRUsign, der das Signieren einer Nachricht mittels Polynomen erlaubt.

Nicht nur die Schwierigkeit der Gitterprobleme, die es von anderen Systemen wie RSA unterscheidet, sondern auch Effizienz machen die gitterbasierenden Kryptosysteme für die Zukunft populär.

Abstract

Primary lattices were used in cryptoanalysis as a tool to break cryptographic algorithms. Exciting results on the complexity of lattice problems opened the door to cryptographic public-key cryptosystems relying on the computational difficulty of these problems.

The "IEEE P1363.1TM/D9 draft standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices" defines three parts to implement a cryptosystem whose security is based on the complexity of lattice problems: (1) a reference for specification of techniques from which application may select, (2) the relevant number theoretic background and (3) a discussion of security and implementation considerations. Even if the implementation only uses polynomials for calculation, there is a connection to lattices.

The security of the system relies on the complexity of computing solutions for lattice problems, like the closest vector problem (CVP) or the shortest vector problem (SVP). Many papers have already studied the computational difficulty to find solutions for the CVP and SVP. Exactly this difficulty is also used by NTRUsign to digitally sign a message using polynomials.

Not only the fact, that the system's security is based on the complexity of lattices, what is a difference to e.g. RSA, but also efficiency increases the popularity of lattice based cryptosystems.

Kapitel 1

Einführung

1.1 Kryptosysteme und Gitter

Kryptographiesysteme sind aus der heutigen Informationstechnologie nicht mehr wegzudenken. Die Anzahl der Menschen, die sensitive Daten über Netzwerke oder das Internet austauschen, steigt permanent. Eine Gewährleistung der sicheren Übertragung dieser Informationen, ist ohne gute Kryptosysteme nicht mehr möglich.

Im Gegensatz zu einem Public-Key-System, gleicht ein symmetrisches Kryptosystem einem Safe, wobei der geheime Schlüssel die Kombination ist [20, S. 31ff]. Jeder, der die Kombination besitzt, kann den Safe öffnen und erhält Zugriff auf den Inhalt. Jemand, der die Kombination nicht kennt, wird an den Inhalt auf vorgesehenem Weg nicht kommen.

Bei der Public-Key-Kryptographie werden zwei unterschiedliche Schlüssel verwendet, ein öffentlicher und ein privater. In diesem Zusammenhang ist es schwierig, falls jemand nur den öffentlichen Schlüssel besitzt, daraus den privaten zu berechnen. Ein jeder kann mit dem öffentlichen Schlüssel eine Nachricht verschlüsseln, aber nicht wieder entschlüsseln. Nur die Person, die Zugriff auf den privaten Schlüssel hat, kann aus der verschlüsselten Nachricht wieder den Klartext berechnen. Im folgenden Beispiel wird erklärt, wie Alice Bob eine Nachricht mittels Public-Key-Kryptographie schicken kann:

- Alice und Bob einigen sich auf ein Public-Key-Kryptosystem.
- Bob sendet an Alice seinen öffentlichen Schlüssel.
- Alice verschlüsselt mit Bobs öffentlichen Schlüssel ihre Nachricht und schickt diese an Bob.
- Bob entschlüsselt die von Alice erhaltene Nachricht mit seinem privaten Schlüssel.

In disem Beispiel lässt sich erkennen, dass das Problem des sicheren Schlüsselaustausches, welches bei einem symmetrischen Kryptosystem entsteht, nicht mehr vorhanden ist. Bei einem symmetrischen System hätte

Alice einen zufälligen Schlüssel generiert und diesen auf sicherem Weg zu Bob bringen müssen. Public-Key-Kryptographie ermöglicht es Alice, ohne vorher über sichere Kommunikationswege einen geheimen Schlüssel austauschen zu müssen, eine verschlüsselte Nachricht an Bob zu schicken[1]. Eve, die den Nachrichtenaustausch von Alice und Bob abhört, kann zwar den öffentlichen Schlüssel und die verschlüsselte Nachricht mitlesen, kann daraus aber weder auf den privaten Schlüssel von Bob, noch auf den ursprünglichen Klartext zurückschließen.

Einwegfunktionen: Eine Möglichkeit, ein Public-Key-Kryptosystem zu konstruieren, sind Einwegfunktionen. Dazu wird eine Einwegfunktion $f(x)$ benötigt für die es einfach ist, für ein gegebenes x, $f(x)$ zu berechnen. Für ein gegebenes $f(x)$ muss es jedoch, z. B. für einen Angreifer, ausreichend schwierig sein den zugehörigen x-Wert zu finden. Üblicherweise handelt es sich bei $f(x)$ um eine Einwegfunktion mit einer Hintertür[2]. D. h. es gibt eine einfache Möglichkeit, unter Zuhilfenahme einer geheimen Information y, um zu $f(x)$ das zugehörige x zu finden. Diese geheime Information y ist normalerweise nur dem bekannt, der auch über den geheimen privaten Schlüssel verfügt [20, S. 30].

Bei RSA[3] ist $f(x) = x^e \pmod{n}$, für ein geeignetes n und e, die Einwegfunktion und die geheime Information für die Hintertür ist die Primfaktorzerlegung von n. Im IEEE-P1363.1-Standard steckt die geheime Information in zwei kleinen Polynomen[4].

Wird bedacht, wie wichtig Public-Key-Systeme sind, so ist es erstaunlich, dass das Augenmerk nur auf einige wenige Verfahren gelegt wird, die sich in der Vergangenheit bewährt haben. Noch dazu beruht die Sicherheit dieser Systeme zum größten Teil auf der Schwierigkeit der Berechnung von Problemen in endlichen Gruppen, wie z. B. die Primfaktorzerlegung oder das Finden von diskreten Logarithmen. Es stellt sich in diesem Zusammmenhang die Frage, ob es sinnvoll und zukunftssicher ist, die Sicherheit eines Verfahrens durch die Vergrößerung der Parameter zu gewährleisten.

Aus diesem Grund müssen neue Kryptosysteme entwickelt werden, deren Sicherheit auf anderen mathematischen Problemen basiert, wie z. B. die Gitterprobleme CVP und SVP. Die Aufgabe der Kryptoanalyse ist es dann, zu zeigen, dass sich auch die Kryptosysteme, die auf Gitter basieren oder direkt mit Gittern arbeiten, ebenso bewähren können, wie RSA in den letzten Jahren.

---

[1] Vorraussetung ist, dass Alice den öffentlichen Schlüssel von Bob besitzt.

[2] Diese wird im Englischen auch als *trapdoor one-way function* bezeichnet.

[3] Für eine Erklärung von RSA siehe [15, S. 284ff].

[4] Diese beiden Polynome tauchen aneinandergekettet im entsprechenden Gitter auf (vgl. Abschnitt 3.1.2).

1.2 Zielsetzung

Im Zuge dieser Arbeit soll das im „IEEE P1363.1TM/D9 draft standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices“ spezifizierte Ver- und Entschlüsselungsschema vorgestellt, implementiert und analysiert werden. Dazu werden die im Standard angeführten Funktionen programmiert und danach auf ihre Performanz getestet. Am Ende soll gezeigt werden, dass es mittels dem implementierten Standard möglich ist, eine Nachricht erfolgreich zu ver- und entschlüsseln.

Als Einstieg liefert das Kapitel 2 eine Einführung in die Gittertheorie und stellt die wichtigsten Gitterprobleme vor. Außerdem wird kurz behandelt, mit welchen Mitteln ein auf Gittern basierendes System angegriffen werden kann. Kapitel 3 stellt das im Standard definierte Schema vor und beschreibt dessen implementierte Funktionen in der Programmiersprache *C++*. Außerdem wird, jedoch ohne einer Implementierung, die Funktionsweise von NTRUsign beschrieben. Kapitel 4 enthält die Testergebnisse zu den im Kapitel 3 erstellten Funktionen und zeigt außerdem Eingenschaften auf, die charakteristisch für die Algorithmen sind. Abschließend werden in Kapitel 5 zukünftige Entwicklungen angeführt und ein Resümee gezogen.

Kapitel 2

Grundlagen

2.1 Relevante Begriffe und Definitionen

Im Zusammenhang mit gitterbasierenden Kryptographiesystemen tauchen mathematische Ausdrücke auf, die in diesem Abschnitt zum besseren Verständnis erklärt werden. Wenn nicht weiter angegeben beruhen die Definitionen auf [15, S. 63–78].

1. **Die ganzen Zahlen modulo n**

   (a) $\mathbb{N}$ kennzeichnet die Menge der positiven ganzen Zahlen $\{1, 2, 3, \ldots\}$.

   (b) $\mathbb{Z}$ kennzeichnet die Menge der ganzen Zahlen $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.

   (c) Die Menge der Äquivalenz- bzw. Restklassen modulo einer Zahl $n \in \mathbb{N}$ ist gekennzeichnet durch $\mathbb{Z}_n$, also ist

   $$\mathbb{Z}_n = \{[0]_n, [1]_n, \ldots, [n-1]_n\}.$$

   Addition, Subtraktion und Multiplikation in $\mathbb{Z}_n$ werden modulo n durchgeführt.

2. **Polynomringe**

   (a) Ist R ein kommutativer Ring, so wird unter einem Polynom $f(x)$ in R ein Ausdruck der Form

   $$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{2.1}$$

   verstanden. Das Element a_i kennzeichnet den Koeffizienten von x^i in $f(x)$, mit $a_i \in R$ und $n \geq 0$. Die größte ganze Zahl m mit $a_m \neq 0$ wird als Grad von $f(x)$ bezeichnet.

(b) Sei R ein kommutativer Ring, so bildet die Menge aller Polynome mit der Variable x und Koeffizienten in R den Polynomring $R[x]$. Die beiden arithmetischen Operationen des Polynomrings sind die Polynomaddition und die Polynommultiplikation, wobei die Berechnung der Koeffizienten in R stattfindet.
Hierzu ein Beispiel: Seien $f(x) = x^3 + x + 1$ und $g(x) = x^2 + x$ Elemente des Polynomrings $\mathbb{Z}_2[x]$, so ist

$$f(x) + g(x) = x^3 + x^2 + 1$$

und

$$f(x) * g(x) = x^5 + x^4 + x^3 + x.$$

(c) Ist $f(x)$ ein Element eines Polynomrings $R[x]$ und besitzt wenigstens Grad 1, dann wird es als irreduzibel über R bezeichnet, falls es nicht möglich ist, $f(x)$ als Produkt zweier Polynome in $R[x]$, beide mit positivem Grad, darzustellen.

(d) $R[x]/f(x)$ kennzeichnet die Menge der Äquivalenz- bzw. Restklassen der Polynome in $R[x]$ modulo $f(x)$. $R[x]/f(x)$ ist außerdem ein kommutativer Ring. Gilt für $f(x)$, dass es sich um ein irreduzibeles Polynom über R handelt, dann ist $R[x]/f(x)$ ein Körper.

3. **Polynommultiplikation in $\mathbb{Z}_q[x]/x^N - 1$**

Die Multiplikation von Polynomen $c = a*b$ in $\mathbb{Z}_q[x]/x^N - 1$ kann durch den Ausdruck

$$c_k = \sum_{i+j=k \pmod N} a_i b_j : 0 \leq k \leq N-1, 0 \leq i,j < N \qquad (2.2)$$

beschrieben werden. Mittels Rotationen der Koeffizienten von c kann das Ergebnis auch durch

$$\begin{aligned} &a_0 * (b_0, b_1, b_2, \ldots, b_{N-1}) + \\ &a_1 * (b_{N-1}, b_0, b_1, \ldots, b_{N-2}) + \\ &a_2 * (b_{N-2}, b_{N-1}, b_0, \ldots, b_{N-3}) + \\ &\vdots \\ &= (c_0, \ldots, c_{N-1}) \end{aligned}$$

berechnet werden (vgl. Abschnitt 4.1).

4. ***Plaintext awareness***

Ein Public-Key-System wird als *plaintext aware* bezeichnet, falls es für einen Angreifer unmöglich ist, ohne Kenntniss des Klartextes, einen gültigen chiffrierten Text zu erzeugen [15, S. 311-312].

2.2 Gitter

Bevor auf einzelne Algorithmen näher eingegangen wird, gibt der nächste Abschnitt einen Überblick über Gitter und deren Komplexität. Die folgende Einführung in die Gittertheorie beruht, wenn nicht weiter vermerkt, auf [17, Kap. 1].

Ist $\mathbb{R}^m$ ein m-dimensionaler euklidischer Raum, so ist ein Gitter in $\mathbb{R}^m$ die Menge aller Linearkombinationen von n linear unabhängigen Vektoren $b_1, \ldots, b_n$ in $\mathbb{R}_m (m \geq n)$:

$$\mathcal{L}(b_1, \ldots, b_n) := \left\{ \sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z} \right\}. \tag{2.3}$$

Die Parameter m und n werden entsprechend Ordnung bzw. Dimension des Gitters genannt. Ist $m = n$ so wird das Gitter als vollständig oder volldimensional bezeichnet. Die Folge $b_1, \ldots, b_n$ wird als Gitterbasis bezeichnet, welche als Matrix

$$B := [b_1, \ldots, b_n] \in \mathbb{R}^{m \times n} \tag{2.4}$$

mit den Basisvektoren als Spalten dargestellt werden kann. In Matrixschreibweise kann (2.3) in kompakterer Form als

$$\mathcal{L}(B) := \{Bx : x \in \mathbb{Z}^n\} \tag{2.5}$$

aufgeschrieben werden.

Grafisch kann ein Gitter als die Menge der Punkte eines unendlichen, regelmäßigen n-dimensionalen Rasters beschrieben werden. In Abbildung 2.1 wird ein 2-dimensionales Gitter, erzeugt von den Basisvektoren

$$b_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \tag{2.6}$$

dargestellt. Hervorzuheben ist, dass ein Gitter keine eindeutigen Basisvektoren besitzt, sondern unendlich viele verschiedene Basen hat. So ist z. B.

$$b'_1 = b_1 + b_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \qquad b'_2 = 2b_1 + b_2 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \tag{2.7}$$

ebenfalls eine Basis für $\mathcal{L}(b_1, b_2)$. Der von den beiden Basisvektoren erzeugte Raster wird in Abbildung 2.2 dargestellt. Zu beachten ist, dass, obwohl die beiden Raster unterschiedlich sind, sich die Mengen der erzeugten Gitterpunkte gleichen. D. h. $\{b_1, b_2\}$ und $\{b'_1, b'_2\}$ sind verschiedene Basen für das gleiche Gitter: $\mathcal{L}(b_1, b_2) = \mathcal{L}(b'_1, b'_2)$.

Unabhängig von der Basis kann jedem Gitter eine reelle Zahl zugeordnet werden. Die Gitterdeterminante eines Gitters $\Lambda = \mathcal{L}(B)$, bezeichnet mit $\det(\Lambda)$, ist das n-dimensionale Volumen des von den Basisvektoren aufgespannten Parallelepipeds $\mathcal{P}(B)$ (s. auch [17, S. 6]).

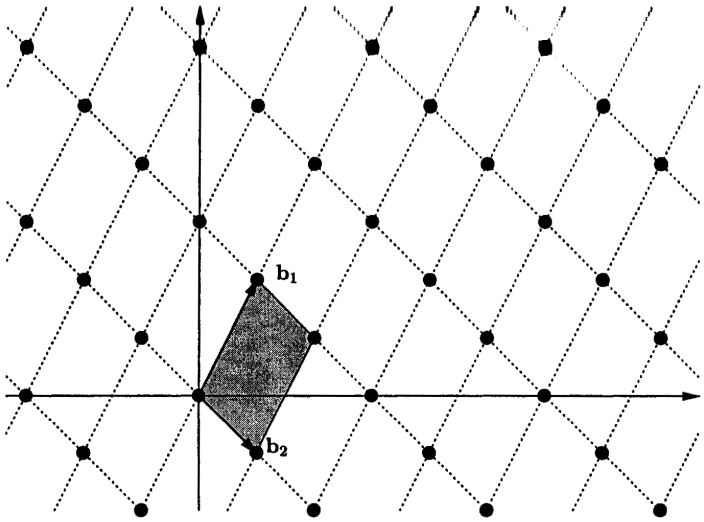

Abbildung 2.1: Ein von b1 und b2 erzeugtes Gitter (aus [17, S. 2]).

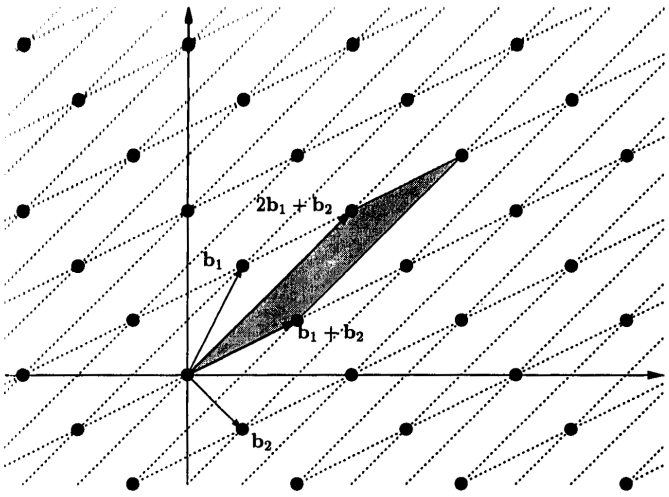

Abbildung 2.2: Eine weitere Basis für $\mathcal{L}(b_1, b_2)$ (aus [17, S. 3]).

Berechnet wird die Gitterdeterminante unter Verwendung der Determinante[1] einer Gram-Matrix $B^T B$, einer $n \times n$-Matrix, die als (i, j)-ten Eintrag das Skalarprodukt $\langle b_i, b_j \rangle$ besitzt:

$$\det(\mathcal{L}(B)) = \sqrt{\det(B^T B)}. \tag{2.8}$$

[1] Die Berechnung einer Determinante von $n \times n$ Matrizen kann in [14, S. 123ff] nachgelesen werden.

2.2.1 Gitterprobleme

Im Zusammenhang mit Gitterproblemen spielen Normen, Abstände und sukzessive Minima[2] eine wichtige Rolle [17, S. 7].

Euklidsche Norm: Die ℓ_2-Norm, oder euklidsche Norm, wird definiert als

$$\|x\|_2 = \sqrt{\langle x, x\rangle} = \sqrt{\sum_{i=1}^{n} x_i^2}. \tag{2.9}$$

Euklidsche Distanz: Entsprechend der ℓ_2-Norm berechnet sich der Abstand zweier Vektoren x, y durch

$$\mathrm{dist}(x, y) = \|x - y\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{2.10}$$

Sukzessive Minima: Für ein n-dimensionales Gitter Λ sind die sukzessiven Minima $\lambda_1, \ldots, \lambda_n$. Das i-te Minimum $\lambda_i(\Lambda)$ ist hierbei der Radius der Kugelfläche um den Ursprung, welche i unabhängige Gittervektoren enthält [16, S. 20]. Insbesondere ist $\lambda_1(\Lambda)$ die Länge des kürzesten, vom Nullvektor verschiedenen Vektor[3] und gleicht außerdem dem Minimalabstand zwischen zwei unterschiedlichen Gitterpunkten:

$$\lambda_1(\Lambda) = \min_{x \neq y \in \Lambda} \|x - y\|_2 = \min_{x \in \Lambda \setminus 0} \|x\|_2 . \tag{2.11}$$

***Shortest vector problem*, SVP:** Das Problem, einen Vektor der Länge λ_1 zu finden, wird auch SVP genannt: Gegeben ist eine Basis $B \in \mathbb{Z}^{m \times n}$. Gesucht ist ein Vektor Bx des Gitters mit $x \in \mathbb{Z}^n \setminus \{0\}$, sodass $\|Bx\| \leq \|By\|$ für jedes andere $y \in \mathbb{Z}^n \setminus \{0\}$ gilt. Grafisch kann ein SVP wie in Abbildung 2.3 visualisiert werden[4].

Zur Zeit sind keine effizienten Algorithmen bekannt mit denen ein SVP in Polynomialzeit gelöst werden kann. Genauer gesagt ist auch kein Algorithmus bekannt mit dem Vektoren einer Länger kleiner der Minkowskischen Grenze[5] gefunden werden können [17, S. 17]. Ajtai zeigt außerdem in [1], dass das SVP unter *randomisierter* Reduktion NP-schwer[6] ist.

Ein weiteres Problem, für das es zur Zeit keinen Algorithmus zur Lösung in Polynomialzeit gibt, ist das *closest vector problem.*

[2] Für eine detaillierte Erklärung von sukzessiver Minima siehe z. B. [17, S. 7ff] oder [6, S. 20f].

[3] Für einen Beweis, dass es so ein λ_1 gibt, siehe z. B. [17, S. 9f].

[4] Abbildung aus einer Präsentation von Adi Shamir, erhältlich unter: http://people.csail.mit.edu/tromer/PKC2003/pkclattice1.pdf.

[5] Die Minkowskische Grenze ist gleich dem Ausdruck $\sqrt{n} \cdot \det(B)^{1/n}$.

[6] Für eine Erklärung der Komplexitätstheorie wird auf [15, Kap. 2.3] verwiesen.

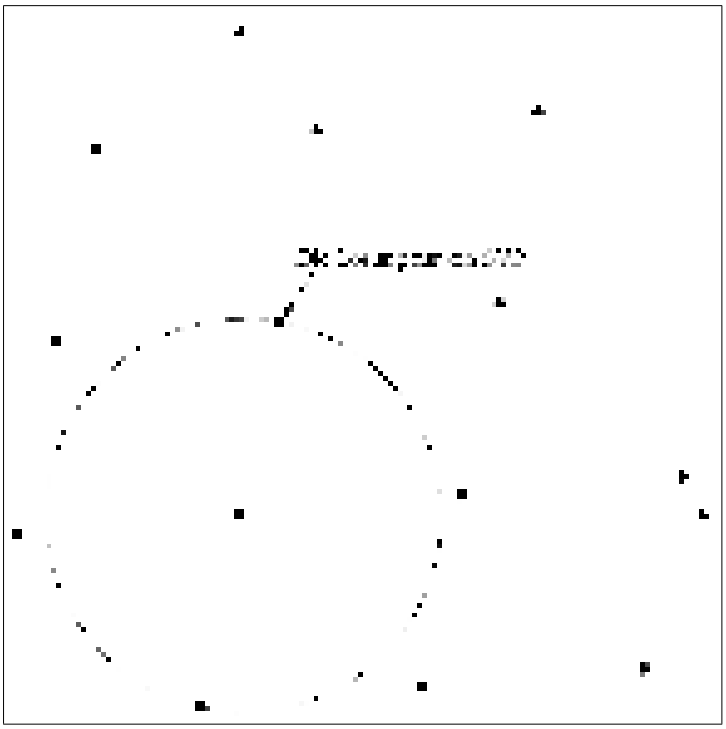

Abbildung 2.3: Darstellung der Lösung eines SVP.

***Closest vector problem*, CVP:** Gegeben ist eine Basis $B \in \mathbb{Z}^{m \times n}$ und ein Zielvektor $t \in \mathbb{Z}^m$. Gesucht ist ein Vektor Bx des Gitters am nächsten zum Zielvektor t, genauer gesagt, finde einen Vektor $x \in \mathbb{Z}^n$, sodass $\|Bx - t\| \leq \|By - t\|$ für jedes andere $y \in \mathbb{Z}^n$ gilt. Grafisch wird die Lösung eines CVP in Abbildung 2.4 dargestellt[7].

Der Abschnitt 2.2 in [17] erwähnt außerdem, im Zusammenhang mit der Komplexität bei Berechnungen von Lösungen für die oben genannten Gitterprobleme, verschiedene Arten der Aufgabenstellungen zu SVP und CVP, die von Algorithmen gelöst werden sollen (in absteigender Reihenfolge der Schwierigkeit):

- *Search Problem*
- *Optimization Problem*
- *Decision Problem*

Das *decision problem* in Verbindung mit dem CVP ist NP-vollständig [17, S. 48ff] und kann daher von keinem bisher bekannten Algorithmus in Polynomialzeit gelöst werden.

[7] Abbildung wiederum aus der Präsentation von Adi Shamir, erhältlich unter: http://people.csail.mit.edu/tromer/PKC2003/pkclattice1.pdf.

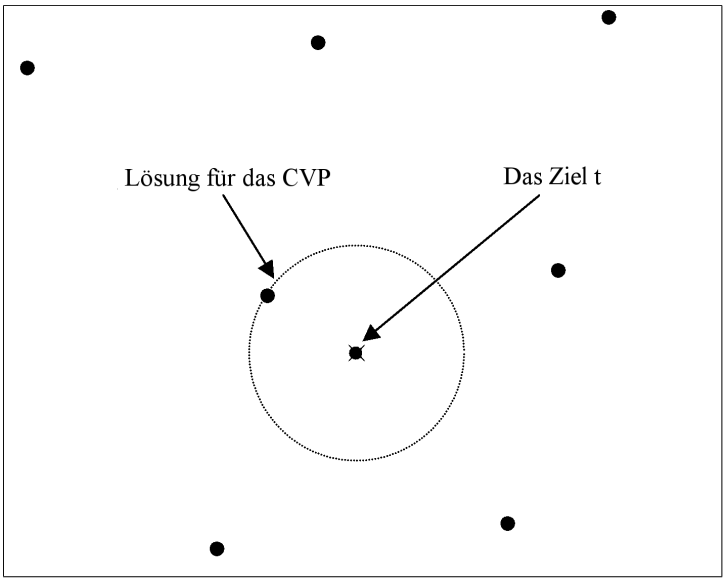

Abbildung 2.4: Darstellung der Lösung eines CVP.

Die Schwierigkeit Lösungen für ein CVP und SVP zu berechnen brachte Algorithmen hervor, mit denen Näherungslösungen bestimmt werden können. Ein Algorithmus, der eine Näherungslösung bestimmt, kann garantieren, dass sein Ergebnis der Berechnung kleiner als ein bestimmer Faktor γ, im Vergleich zur optimalen Lösung, ist. Es folgen die Näherungsversionen des SVP und CVP.

***Approximate* SVP:** Gegeben ist eine Basis $B \in \mathbb{Z}^{m \times n}$. Gesucht ist ein Vektor Bx des Gitters mit $x \in \mathbb{Z}^n \setminus \{0\}$, sodass $\|Bx\| \leq \gamma \cdot \|By\|$ für jedes andere $y \in \mathbb{Z}^n \setminus \{0\}$ gilt.

***Approximate* CVP:** Gegeben ist eine Basis $B \in \mathbb{Z}^{m \times n}$ und ein Zielvektor $t \in \mathbb{Z}^m$. Gesucht ist ein Vektor Bx des Gitters mit $x \in \mathbb{Z}^n$, sodass $\|Bx - t\| \leq \gamma \cdot \|By - t\|$ für jedes andere $y \in \mathbb{Z}^n$ gilt.

Der Näherungsfaktor γ kann bei der Berechnung der Lösung eines *approximate* SVP und *approximate* CVP in Relation zu einem Parameter des verwendeten Gitters stehen. Üblicherweise wird die Dimension des Gitters verwendet, damit die Schwierigkeit des Problems mit der Erhöhung dieses Parameters steigt.

Bei Untersuchungen der Komplexität der Berechnung von Lösungen für Gitterprobleme ist es praktisch, wenn *decision problems* zu *promise problems* verallgemeinert werden. Ein *promise problem* ist ein Paar (Π_{JA}, Π_{NEIN}), für das $\Pi_{JA} \cap \Pi_{NEIN} = \emptyset$ gilt. Ein Algorithmus löst ein *promise problem* (Π_{JA}, Π_{NEIN}), wenn er bei einem Input $I \in \Pi_{JA} \cup \Pi_{NEIN}$ korrekt entschei-

den kann, ob $I \in \Pi_{JA}$ oder $I \in \Pi_{NEIN}$ ist. Falls $I \notin \Pi_{JA} \cup \Pi_{NEIN}$ ist, wird das Verhalten des Algorithmus nicht festgelegt.

Es folgen die *promise problems* in Relation zum *approximate* SVP und CVP (aus [17, S. 20]).

GapSVP$_\gamma$: (γ wird auch *gap*-Funktion genannt, welche in Verbindung mit der Dimension des Gitters steht)

- **JA** Lösungen sind Paare (B, r), wobei $B \in \mathbb{Z}^{m \times n}$ eine Gitterbasis und $r \in \mathbb{Q}$ eine rationale Zahl ist, für die $\|Bz\| \leq r$ für irgendein $z \in \mathbb{Z}^n \setminus \{0\}$ gilt.
- **NEIN** Lösungen sind Paare (B, r), wobei $B \in \mathbb{Z}^{m \times n}$ eine Gitterbasis und $r \in \mathbb{Q}$ eine rationale Zahl ist, für die $\|Bz\| > \gamma \cdot r$ für alle $z \in \mathbb{Z}^n \setminus \{0\}$ gilt.

GapCVP$_\gamma$:

- **JA** Lösungen sind Triplets (B, t, r), wobei $B \in \mathbb{Z}^{m \times n}$ eine Gitterbasis, $t \in \mathbb{Z}^m$ ein Vektor und $r \in \mathbb{Q}$ eine rationale Zahl ist, für die $\|Bz - t\| \leq r$ für irgendein $z \in \mathbb{Z}^n$ gilt.
- **NEIN** Lösungen sind Triplets (B, t, r), wobei $B \in \mathbb{Z}^{m \times n}$ eine Gitterbasis, $t \in \mathbb{Z}^m$ ein Vektor und $r \in \mathbb{Q}$ eine rationale Zahl ist, für die $\|Bz - t\| > \gamma \cdot r$ für alle $z \in \mathbb{Z}^n$ gilt.

Dies bedeutet für das GapSVP$_\gamma$ und das GapCVP$_\gamma$, dass für den Bereich zwischen r und $\gamma \cdot r$ nicht entschieden werden kann, ob es sich um ein Ja oder ein Nein Ergebnis handelt.

Für eine detaillierte Analyse der Komplexität der einzelnen Probleme wird auf Kapitel 3 und 4 in [17] verwiesen. Weiters sind die oben aufgeführten Gitterprobleme nur die bekanntesten und daher ist die Liste der Probleme nicht vollständig. Zusätzliche Informationen zu Gitterproblemen können unter anderem in [21] und [4] gefunden werden.

2.2.2 Gitterreduktion

Ziel der Gitterbasen- bzw. Gitterreduktion ist es (laut [12, S. 3]):

1. Kurze Vektoren zu Beginn der Matrix anzuführen
2. Dass die Spaltenvektoren paarweise möglichst orthogonal zueinander sind

Der Abschnitt 1.2 in [12] erwähnt außerdem, dass folgende einfache Operationen genügen, um eine Basis Bx eines Gitters $\mathcal{L}$ zu reduzieren:

- Vertausche zwei Spalten der Matrix Bx

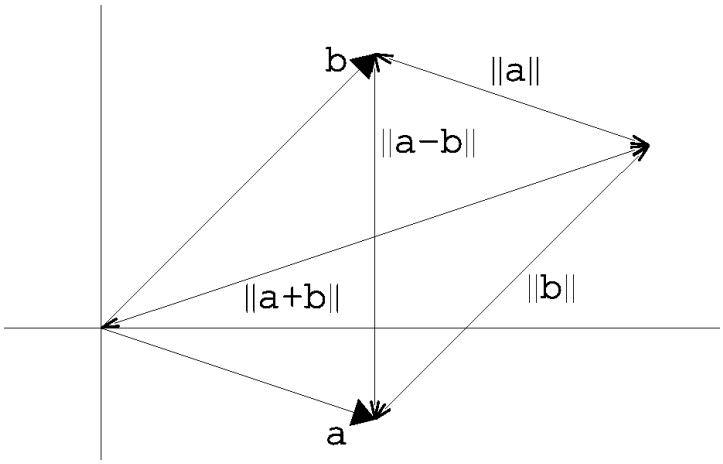

Abbildung 2.5: Eine reduzierte Basis eines Gitters der Dimension 2.

- Addiere ein ganzzahliges Vielfaches einer Spalte zu einer anderen Spalte
- Lösche Nullspalten[8]

Dabei ist es schwierig den Ablauf der oben genannten Schritte so zu wählen, dass die Vektoren möglichst kurz werden. Idealwerweise sollte nach dem Terminieren des Algorithmus die erste Zeile dem kürzesten, vom Nullvektor verschiedenen Vektor gleichen. In der Praxis ist jedoch kein Algorithmus bekannt der diese Aufgabe effizient lösen kann. Es ist lediglich möglich einen Vektor aufzustellen, der nicht länger ist, als die optimale Lösung multipliziert mit einer Konstante abhängig von der Dimension des Gitters.

Für den Spezialfall eines Gitters der Dimension 2 berechnet der Gauss Algorithmus eine reduzierte Basis. Dazu ist nach [17, S. 24] eine Basis $[a, b]$ (bezüglich der Dimension 2) des Gitters auch reduziert, wenn

$$\|a\|, \|b\| \leq \|a + b\|, \|a - b\|.$$

Grafisch kann diese Gleichung wie in Abbildung 2.5 dargestellt werden.

In [17, S. 26] wird außerdem definiert, dass eine Basis dann ideal reduziert ist, wenn a und b die Länge der ersten beiden sukzessiven Minima λ_1, λ_2 besitzen (d. h. auch, dass ein SVP der Dimension 2 gelöst werden kann).

Der LLL-Algorithmus

Der bekannte Lenstra-Lenstra-Lovasz (LLL) Gitterreduktionsalgorithmus (s. auch [13]) berechnet Näherungslösungen von reduzierten Basen in höheren Dimensionen als 2. Mit seiner Hilfe kann z. B. ein SVP mit der Einschränkung, dass die Länge des Lösungsvektors kleiner oder gleich $\gamma(n) \cdot \lambda_1$ mit

[8] Dies ist nur dann möglich, wenn die Spalten nicht linear unabhängig sind.

$\gamma(n) = (2/\sqrt{3})^n$, und nicht gleich λ_1, ist [17, S. 23]. Abschnitt 3 des zweiten Kapitels von [17] zeigt außerdem, wie der LLL-Algorithmus dazu verwendet werden kann ein CVP innerhalb des Faktors $2(2/\sqrt{3})^n$ zu lösen. Dieser Algorithmus wird auch als *nearest plane algorithm* bezeichnet, erstmals vorgestellt in [2]. Inzwischen existieren mehrere Varianten des LLL-Algorithmus, die unter anderem in [5] behandelt werden.

Kapitel 3

Gitterbasierende Verfahren

3.1 Implementierung des IEEE-P1363.1-Standards

Zunächst soll als Einführung erklärt werden, um was es sich bei der IEEE-P1363-Standardgruppe handelt.

IEEE-P1363 ist ein *Institute of Electrical and Electronics Engineers* Projekt für die Entwicklung und Standardisierung von Public-Key-Kryptographie[1]. Das Ziel des Projektes ist es eine Referenz für Spezifikationen der verschiedensten Techniken zur Verfügung zu stellen. Die einzelnen Standards unterteilen sich dabei in folgende Gruppen[2]:

1. ***Traditional Public-Key Cryptography***: (1363-2000, 1363a-2004) Der Standard inkludiert digitale Signaturen und Schlüsselerstellung basierend auf drei mathematischen Problemen:
   - Faktorisierung von ganzen Zahlen (z. B. RSA).
   - Berechnung von diskreten Logarithmen (z. B. Diffie-Hellman).
   - Berechnung von diskreten Logarithmen auf elliptischen Kurven (z. B. MQV).
2. ***Lattice-Based Public-Key Cryptography***: (P1363.1) Der Standard schließt gitterbasierende Systeme zur Verschlüsselung (z. B. NTRUEncrypt) und zum digitalen Signieren (z. B. NTRUsign) mit ein.
3. ***Password-Based Public-Key Cryptography***: (P1363.2) Der Standard beruht auf Passwort-basierenden Kryptosystemen (z. B. EKE, Ford & Kaliski).
4. ***Identity-Based Public-Key Cryptography using Pairings***: (P1363.3) Der Standard spezifiziert idenditätsbasierende Public-Key Systeme.

[1] Siehe auch http://grouper.ieee.org/groups/1363/

[2] Die englischen Namen wurden bewusst nicht ins deutsche übersetzt.

IEEE-P1363.1: Der Standard stellt eine Referenz für die Implementierung eines Public-Key-Kryptographiesystemes zur Verfügung. Er beschreibt zwei unterschiedliche Bausteine, die für eine Applikation nötig sind [11, S. 11].

- *Primitives* sind grundlegende mathematische Operationen, wie z. B. eine Polynommultiplikation modulo $x^N - 1$, basierend auf numerischen Problemen. Mehrere *Primitives* als Einheit ergeben ein *Scheme*.
- *Schemes* verkörpern eine Sammlung von *Primitives* und zusätzlichen Hilfsfunktionen, die Sicherheit gewährleisten können, falls sie standardkonform implementiert werden.

3.1.1 Beschreibung des Kryptosystems

Der folgende Abschnitt erklärt die Funktionalität des IEEE-P1363.1-Kryptosystems im Bezug auf die verwendeten Parameter, die Erstellung eines Schlüsselpaares, der Verschlüsselung einer Nachricht, sowie dessen Entschlüsselung. Außerdem wird erkärt, warum es möglich ist eine verschlüsselte Nachricht erfolgreich zu entschlüsseln.

Parameter:

- $N \in \mathbb{N}$: Parameter für den Grad der Polynome.
- $q \in \mathbb{P}$: Der große Modul, eine Primzahl.
- $p \in \mathbb{N}$: Der kleine Modul, relativ prim zu q.

Der Parameter N wird dazu verwendet, um den Ring zu definieren, in dem die arithmetischen Operationen durchgeführt werden. Die Polynome besitzen außerdem ganzzahlige Koeffizienten modulo q.

Somit kann festgelegt werden, dass die Berechnungen mit Polynomen in $\mathbb{Z}_q[x]/x^N - 1$ stattfinden. Der Parameter p ist normalerweise eine kleine Primzahl, wie z. B. 2 oder 3, und wird bei der Schlüsselerstellung und bei der Entschlüsselung benötigt[3]. Für eine Auflistung der restlichen Parameter siehe Abschnitt 3.1.6 auf Seite 40.

Schlüsselverwaltung: Angenommen Bob will sich ein Schlüsselpaar zur Ver- und Entschlüsselung erzeugen. Er geht dabei wie folgt vor.

Zuerst wählt Bob ein zufällig generiertes binäres Polynom F vom Grad $N - 1$ (die Anzahl der Positionen, welche ungleich 0 sein müssen, wird in

[3] Es ist auch möglich für p ein kleines Polynom zu verwenden (vgl. [9]).

diesem Fall von den Parametersets vorgegeben). Dann berechnet er die Polynome

$$f = 1 + p * F, \tag{3.1}$$

$$f_q = f^{-1} \tag{3.2}$$

in $\mathbb{Z}_q[x]/x^N - 1$. Bob wählt ein weiteres zufällig generiertes, binäres Polynom g (die Anzahl der Positionen, welche ungleich 0 sein müssen, werden wieder von den Parametersets vorgegeben), welches in diesem Ring invertierbar ist und berechnet

$$h = (f_q * g * p) \tag{3.3}$$

in $\mathbb{Z}_q[x]/x^N - 1$. Das Polynom h wird als öffentlicher Schlüssel bezeichnet, die Polynome f und f_q als privater.

Verschlüsselung: Angenommen Alice will Bob eine verschlüsselte Nachricht schicken. Zu diesem Zweck wandelt sie ihren Klartext in ein Polynom m mit Koeffizienten modulo p um. Im nächsten Schritt nimmt Alice ein zufällig generiertes Polynom r (in diesem Zusammenhang auch Blendpolynom genannt) und berechnet unter Verwendung des öffentlichen Schlüssels h von Bob damit

$$e = r * h + m \tag{3.4}$$

in $\mathbb{Z}_q[x]/x^N - 1$. Das Polynom e ist die verschlüsselte Nachricht, die an Bob gesendet werden kann.

Entschlüsselung: Bob erhält von Alice die verschlüsselte Nachricht, um diese zu entschlüsseln. Dazu startet er mit seinem privaten Schlüssel, dem Polynom f, um

$$a = f * e \tag{3.5}$$

in $\mathbb{Z}_q[x]/x^N - 1$ zu berechnen. Im nächsten Schritt reduziert Bob die Koeffizienten des Polynoms a modulo p und erhält damit die ursprüngliche Nachricht m.

Warum die Verschlüsselung funktioniert: Bob startet bei der Verschlüsselung mit der Berechnung von a:

$$a = f * e, \tag{3.6}$$

$$a = g * p * r + m + pFm \tag{3.7}$$

in $\mathbb{Z}_q[x]/x^N - 1$. Durch die Reduzierung von a modulo p wird die ursprüngliche Nachricht m erhalten, falls

$$g * p * r + m + pFm < q$$

ist. Die Paramtersets des IEEE-P1363.1-Standards garantieren die Richtigkeit dieser Gleichung und somit, dass die Verschlüsselung immer funktioniert.

3.1.2 P1363.1 und Gitter

Ein direkter Zusammenhang zwischen dem Kryptosystem und Gittern lässt sich auf den ersten Blick nur schwer erkennen. Eine spezielle Wahl von Matrizen, deren Zeilen als Gitterbasen dienen, stellen eine Verbindung zwischen Polynomen und Gittern her.

Definition: Zyklische Matrix Sei h ein Polynom mit $h = h_0 + \ldots + h_{N-1}X^{N-1}$. Daraus wird die zyklische $N \times N$ Matrix

$$H = \begin{pmatrix} h_0 & h_1 & \cdots & h_{N-1} \\ h_{N-1} & h_0 & \cdots & h_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \cdots & h_0 \end{pmatrix}$$

geformt.

Definition: Modulares Gitter [11, S. 46] Ein modulares Gitter der Dimension $n = 2N$ und dem Modul q ist ein Gitter, erzeugt von den Zeilen einer $N \times N$ Matrix der Form

$$\begin{pmatrix} 1 & \cdots & 0 & h_0 & \cdots & h_{N-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 1 & h_1 & \ldots & h_0 \\ 0 & \cdots & 0 & q & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & q \end{pmatrix}.$$

Oft wird diese Matrix auch in abgekürzter Schreibweise als

$$\begin{pmatrix} I & H \\ 0 & qI \end{pmatrix}$$

angegeben, wobei I die $N \times N$ Einheitsmatrix, 0 die $N \times N$ Nullmatrix und H eine zyklische $N \times N$ Matrix mit Ganzzahlkoeffizienten h_{ij}, für die

$$|h_{ij}| \leq q/2$$

gilt, repräsentiert.

Modulare Gitter und Polynomringe Werden die Polynome

$$f = f_0 + \ldots + f_{N-1}X^{N-1} \text{ und } g = g_0 + \ldots + g_{N-1}X^{N-1}$$

als Zeilenvektor

$$\overline{f} = (f_0, \ldots, f_{N-1}) \text{ und } \overline{g} = (g_0, \ldots, g_{N-1})$$

betrachtet, so lässt sich daraus ein Vektor $[\overline{f}, \overline{g}]$ der Dimension $2N$, durch zusammenhängen von $\overline{f}$ und $\overline{g}$, bilden.

Mit einem normierten Polynom $M(x)$ in $\mathbb{Z}_q[x]$ vom Grad N kann jedes Polynom $h(x)$ im Ring $\mathbb{Z}_q[x]/M(x)$ dazu verwendet werden, um ein modulares Gitter $\mathcal{L}_h$ mit

$$\mathcal{L}_h = \left\{[\overline{f}, \overline{g}] : f * h = g \text{ in } \mathbb{Z}_q[x]/M(x)\right\}$$

zu erzeugen. Im besonderen Fall, dass $M(x)$ gleich $x^N - 1$ ist, ist der rechte obere Teil des modularen Gitters erzeugt von

$$\mathcal{M} = \begin{pmatrix} I & H \\ 0 & qI \end{pmatrix},$$

gleich der zyklischen Matrix H des Polynoms h [11, S. 46].

Ist $\mathcal{L}_h$ nun dieses Modulare Gitter und $g = f * h$ in $\mathbb{Z}_q[x]/x^N - 1$, so kann ein Polynom y berechnet werden, das die Gleichung

$$g = f * h + qy$$

erfüllt. Wird y wiederum als Zeilenvektor $\overline{y}$ betrachtet und an den Zeilenvektor $\overline{f}$ angehängt, um $[\overline{f}, \overline{y}]$ zu formen, so ergibt die Multiplikation

$$[\overline{f}, \overline{y}]M = [\overline{f}, \overline{g}],$$

d. h. $[\overline{f}, \overline{g}]$ befindet sich im Gitter $\mathcal{L}_h$. Es wird also deutlich, dass sich die geheime Information für den privaten Schlüssel f als Teil eines Zeilenvektors in $\mathcal{L}_h$ befindet. Gelingt es also dem Angreifer $[\overline{f}, \overline{g}]$ zu finden und das CVP[4] zu lösen, so ist das Kryptosystem gebrochen [22, S. 389].

3.1.3 Verwendete Bibliotheken bei der Implementierung

C++-Standardbibliotheken

Die Beschreibung der einzelnen Bibliotheken beruht, wenn nicht anders angegeben, auf [23, Kap. 7].

iostream: Ein Vorteil bei der Verwendung von Ein- und Ausgabeströmen gegenüber z. B. einem *printf* in *C* ist die automatisierte Typensicherheit, die der Compiler beim Aufrufen aufgrund des Argumentyps verwendet. Der Hauptanwendungsbereich der *iostream*-Bibliothek liegt bei [23, S. 719]:

- Standardein- bzw. Ausgabe
- Dateiverarbeitung
- Stringverarbeitung im Hauptspeicher

[4] Für eine Erklärung, warum es sich um ein CVP handelt, siehe [10, S. 2ff].

string: In *C* werden zur Darstellung von Zeichenketten *char*-Felder verwendet, einen eigenen Typ *string* gibt es nicht. Das Problem bei der Verwendung der *C*-Funktionen ist, dass die Verantwortung für die Verhinderung von Pufferüberläufen beim Programmierer selbst liegt.

> In *C++* wurde zu diesem Zweck eine auf Klassen basierende Bibliothek für *Strings* implementiert [23, S. 691].

Diese Bibliothek erlaubt einen komfortablen Umgang mit Zeichenketten und bietet zusätzlich eine Speicherverwaltung an, mit der das Schreiben über Feldergrenzen hinaus verhindert wird.

cmath: In *C* als *math.h* bekannt kann die Headerdatei in *C++* unter dem Namen *<cmath>* weiterverwendet werden. Darin enhalten sind diverse mathematische Funktionen wie z. B. die Berechnung von Winkelfunktionen, das Ermitteln von Logarithmen, Wurzel Ziehen oder auch Potenzieren.

cstdlib: Die Funktionen

- *srand(unsigned int seed);* und
- *int rand();*

für die Generierung von Pseudozufallszahlen stammen aus der Headerdatei *<cstdlib>*. Mit *srand* wird der Zufallszahlengenerator initialisiert und der Aufruf von *rand* gibt im Anschluss eine Pseudozufallszahl zwischen 0 und RAND_MAX zurück. RAND_MAX ist in diesem Fall eine von *<cstdlib>* vorgegebene Konstante.

ctime: Für die Initialisierung des Pseudozufallszahlengenerators wird ein entsprechender *seed* benötigt. Mittels der Headerdatei *<ctime>* wird die aktuelle Systemzeit ermittelt und für den Aufruf von *srand* als *seed* verwendet. Somit ist sicher gestellt, dass durch die sekündliche Änderung des *seeds* eine andere Pseudozufallszahl erstellt wird:

```
#include <ctime>
#include <cstdlib>

int main(void){
  int random;
  srand((unsigned)time(0));
  random = rand();
}
```

NTL

NTL ist eine *C++*-Bibliothek für mathematische Berechnungen[5]. Für die Implementierung des Standards ist nur ein Algorithmus der Bibliothek benötigt worden, nämlich der Algorithmus zum Invertieren einer Zahl modulo einer Zahl N. NTL bietet noch viele weitere Algorithmen an, die für eine Implementierung im Bereich der Kryptographie interessant sein können.

3.1.4 Unterstützende Algorithmen

Bei der Implementierung des IEEE-P1363.1-Standards wird bei manchen Funktionen die Berechnung eines Hashwertes benötigt. Eine der im Standard empfohlenen Hashfunktionen ist hierbei SHA-1.

SHA-1

Als Implementierung der Hashfunktion SHA-1 wird die freie *CSHA1-Klasse* von Dominik Reichl verwendet. Der Quellcode sowie eine Testapplikation stehen auf der *Codeproject*-Webseite [19] zur Verfügung. Der Vollständigkeit halber sollen die wichtigsten Funktionen der *CSHA1*-Klasse kurz erklärt werden.

- void Update(unsigned char* data, unsigned int len);

  Die Funktion *Update* wird aufgerufen, um einen Datenstrom, für den der Hashwert berechnet werden soll, hinzuzufügen. *Data* ist ein Zeiger auf den Beginn der Zeichenkette und *len* ist die Länge der Daten in Bytes.

- void Final();

  Nachdem alle Daten erfolgreich durch die *Update*-Funktion hinzugefügt wurden, berechnet die *Final*-Funktion den SHA-1-Wert.

- void ReportHash(char *szReport, unsigned char uReportType);

  Nach dem Aufruf der *Final*-Funktion kann mittels *ReportHash* der Hashwert geholt werden. Dieser wird als Zeichenkette in *szReport* gespeichert. Mit *uReportType* kann die Ausgabeform des Hashwertes beeinflusst werden. Hierbei sind zwei verschiedene Typen gültig. *REPORT_HEX* gibt den Hashwert in hexadezimaler Form aus, im Gegensatz dazu kann mit *REPORT_DIGIT* der Hashwert als Zahlenkette zur Basis 256 ausgegeben werden.

- void GetHash(unsigned char *uDest);

  Will man den Hashwert nicht in vorformatierter Weise, so wie er bei *ReportHash* erstellt wird, so kann durch die *GetHash*-Funktion der Hashwert in Byte Form erhalten werden.

[5] Siehe auch http://www.shoup.net/ntl/.

3.1.5 Implementierte Funktionen

Für eine übersichtliche Programmierung sind die einzelnen Funktionen auf mehrere Quelldateien, die auch als Module gesehen werden können, aufgeteilt. In diesem Zusammenhang existiert zu jeder Quelldatei eine öffentliche Schnittstelle, die Headerdatei (.h bzw..hpp). Diese enthält sämtliche Funktionsdeklarationen, Strukturen und Kommentare, welche dem Anwender bei der Verwendung behilflich sind. Die zugehörige .cpp realisiert die eigentliche Ausarbeitung der Schnittstellen.

Die einzelnen Methodendeklarationen sind auf folgende Headerdateien aufgeteilt:

- *polynom.h*: Definiert den Aufbau der Polynom Klasse.
- *auxiliaryAlgorithms.h*: Definiert selbst erstellte Hilfsfunktionen, z. B. die Erzeugung eines zufälligen *Octet Strings*.
- *mathematFoundation.h*: Definiert die Algorithmen 1 bis 5 des Standards.
- *dateTypeConversion.h* Definiert die Algorithmen 6 bis 17 des Standards.
- *supportingAlgorithms.h* Definiert die Algorithmen 18 bis 21 des Standards.
- *encryption.h* Definiert die Algorithmen 22 bis 26 des Standards.

Im folgenden Abschnitt werden mit vereinzelten Beispielen die realisierten Funktionen beschrieben und deren Deklaration als Unterstützung für eine korrekte Verwendung angeführt. Bei den Algorithmen des P1363.1-Standards ist in der Deklaration außerdem die zugehörige Nummerierung mitangeführt.

1. **Die *polynom*-Klasse**

   Ohne weiteres kann ein Polynom auch als eine Reihe von Koeffizienten dargestellt werden. Dazu soll das Polynom

   $$f = 2 + 3x + 4x^2 + 3x^3$$

   als Feld von vier Koeffizienten dargestellt werden:

   $$f = [2, 3, 4, 3].$$

   Es liegt nahe, dass in *C++* für die Repräsentierung eines Polynoms ein Feld für Integer-Elemente angelegt wird und die Koeffizienten darin gespeichert werden. Vom Standpunkt der Plausibilität und Effizienz wäre dies sicher die beste Wahl, doch hat sie einen entscheidenen Nachteil. Ein Feld wird immer eine fixe Größe besitzen und ein dynamisches Anpassen der Elementkapazität wäre nur durch umständliche *Wrapper*-Funktionen möglich.

```
#include <iostream>
#include <vector>
using namespace std;

int main (void){
  vector<int> testVector (10);
  int i;
  for(i=0;i<10;i++){
  //Der Vektor wird mit Werten von 0 bis 9 befüllt
    testVector.at(i)=i;
  }
  for(i=0;i<10;i++){
  //Wiederum wird 0 bis 9 an den Vektor angehängt
   testVector.push_back(i);
  }
  for(i=0;i<20;i++){
    cout<<testVector.at(i)<<" ";
  }
}
```

Programm 3.1: Bsp. der Vektor Klasse

Eine nützliche *C++*-Klasse ist die Klasse *<vector>*. Ein Vektor verhält sich im Grunde wie ein statisches Feld, bietet jedoch auch verschiedene Möglichkeiten, mit denen der Vektor bearbeitet und verändert werden kann. Eine dieser Funktionen stellt das einfache Erweitern um Elemente dar, das am Anfang und Ende des Vektors sogar in konstanter Zeit ausgeführt wird (d. h. es dauert beim jedem Einfügungsvorgang gleich lange). Ein einfaches Beispiel wie ein Vektor verwendet werden kann zeigt das Programmstück 3.1.

Da durch die Template Klasse *vector<T>* das Arbeiten mit Listen stark vereinfacht wird und definierte Schnittstellen zur Verfügung gestellt werden, eignet sie sich zur Darstellung von Koeffizienten der Polynome. Die Nachteile der höheren Zeit beim Einfügen von Elementen in den Vektor und beim Suchen von Elementen im Vektor, kann bei Polynomen außer Acht gelassen werden. Die Klasse *polynom* hat daher als einziges Attribut einen Vektor, der die Koeffizienten der Polynome repräsentiert.

2. **Konstruktoren der *polynom*-Klasse**

   Die Klasse stellt drei verschiedene Varianten zur Verfügung, mit der Polynome erzeugt werden können. Die erste Methode erzeugt ein Polynom, das nur einen Koeffizienten, der gleich Null ist, besitzt. Die zweite Konstruktorart erzeugt ein Polynom mit $N+1$ Elementen gleich 0. Der dritte und letzte Konstruktor ist hilfreich, wenn ein Polynom kopiert

werden soll.

```
//Erstellt ein Polynom gleich 0
Polynom();

/*Erstellt ein Polynom mit N
 Koeffizienten gleich 0*/
Polynom(int N);

//Kopiert das Polynom poly in das aufrufende Polynom
Polynom(const Polynom& poly);
```

3. **Berechnung des Grades eines Polynoms**

   Eine Funktion zur Berechnung des Grades eines Polynoms ist vor allem in jenen Fällen wichtig, bei denen es sich um besondere Polynome handelt. Als einfaches Beispiel dient die nächste Gleichung, welche ein Polynom vom Grad 4 darstellt:

   $$f = 1 + 2x^2 + 4x^4.$$

   Festgelget werden muss der Grad eines Polynoms, das nur aus einer Konstante besteht bzw. der des Nullpolynoms. Da der Grad z. B. bei der Schleifenbedingung des erweiterten euklidschen Algorithmus als Abbruchbedingung dient, sind die Rückgabewerte der Funktion nicht unrelevant.

   In der *polynom*-Klasse wird der Grad eines Polynoms, das nur aus einer Konstante besteht, mit 0 festgelegt. Das Nullpolynom erhält den Grad -1.

```
/*
*Berechnung des Grades eines Polynoms.
*@this Polynom, für das der Grad berechnet wird
*@return Der Grad des Polynoms oder −1,
* falls es sich um das Nullpolynom handelt.
*/
int deg()const;
```

4. **Multiplikation eines Polynoms mit einem Polynom f in Binärform**

   Die Funktion multipliziert ein Polynom a mit einem zweiten Polynom f, dessen Koeffizienten nur aus Einsen und Nullen besteht (Binärform). Wichtig ist, dass an die Funktion nicht das ganze Polynom f übergeben wird, sondern ein Feld der Positionen jener Koeffizienten, welche ungleich 0 sind. So wird z. B. für das Polynom $1 + x^2 + x^5$ das Feld

   $$[1, 2, 5] \tag{3.8}$$

   an die Funktion übergeben. Das Ergebnis der Multiplikation wird in einem weiteren Polynom $c = a * f$ in $\mathbb{Z}_q[x]/x^N - 1$ gespeichert.

```
/*
*Algorithmus 1 – Multiplikation eines Polynoms a mit einem Polynom f
*in Binärform in Z_q[x]/x^N - 1.
*@int N Maximaler Grad der Polynome
*@int q Der Modul
*@int b[] Feld der Koeffizientenposition von f,
*die ungleich 0 sind
*@int df Länge des Feldes b
*@const Polynom& a Referenz auf das Polynom a
*@Polynom& c Das resultierende Produkt a * f
*/
void multiplyByBinaryPolynomZqXN1(int N, int q, int b[], int df,
    const Polynom &a, Polynom &c);
```

5. **Multiplikation eines Polynoms mit einem Polynom f in Produktform**

   Die Funktion multipliziert ein Polynom a mit einem Polynom f, welches sich aus drei Polynomen f_1, f_2 und f_3 in Binärform mit

   $$f = f_1 * f_2 + f_3$$

   zusammensetzt. Wiederum werden die Polynome nicht direkt übergeben, sondern nur die Positionen der Koeffizienten von f_1, f_2 und f_3, die ungleich 0 sind (siehe Gleichung 3.8). Das Produkt der beiden Polynome a und f wird in einem Polynom c mit $c = a * f$ in $\mathbb{Z}_q[x]/x^N - 1$ gespeichert.

```
/*
*Algorithmus 2 – Multiplikation eines Polynoms a mit einem Polynom f
*in Produktform in Z_q[x]/x^N - 1.
*@int N Maximaler Grad der Polynome
*@int q Der Modul
*@int b1[], b2[], b3[] Felder der Koeffizientenpositionen
*von f1,f2 und f3, die ungleich 0 sind
*@int df1, df2, df3 Länge der Felder b1, b2, b3
*@const Polynom& a Referenz auf das Polynom a
*@Polynom& c Das resultierende Produkt a * f
*/
void multiplyByProductFormPolynomZqXN1 (int N, int q, int b1[],
    int b2[], int b3[], int df1, int df2, int df3, const Polynom&
    a, Polynom& c);
```

6. **Multiplikation eines Polynoms mit einem Polynom f in $\mathbb{Z}_q[x]$**

   Die *polynom*-Klasse stellt eine Funktion zur Verfügung, mit der ein Polynom a mit einem zweiten Polynom f multipliziert werden kann. Das Ergebnis ist

   $$c = a * f \text{ in } \mathbb{Z}_q[x],$$

   d. h. c weißt Koeffizienten zwischen 0 und $q - 1$ auf.

```
/*
*Multiplikation eines Polynoms mit einem Polynom b in Z_q[x].
*@this Polynom, das mit b multipliziert wird
*@const Polynom& b Polynom, das mit dem Aufrufenden multipliziert wird
*@int q Der Modul
*@Polynom& c Das resultierende Produkt a * b
*/
void multiplyPolyZq(const Polynom& b, Polynom& c, int q);
```

7. **Polynomdivision in $\mathbb{Z}_p[x]$**

   Die Funktion berechnet Quotient und Rest einer Division zweier Polynome modulo einer Primzahl p. Zum Beispiel ist der Quotient der Division von

   $$\frac{x^3 - 1}{4 + x + x^2}$$

   gleich $4 + x$ und der Rest gleich $3 + 2x$ in $\mathbb{Z}_5[x]$.

```
/*
*Algorithmus 3 – Polynomdivision in Z_p[x].
*@int N Maximaler Grad von b
*@int p Der prime Modul
*@const Polynom& a Das zu dividierende Polynom
*@const Polynom& b Der Dividend, ein Polynom vom Grad N - 1
*und führendem Koeffizienten ungleich 0
*@Polynom& q,r Resultierende Polynome mit
*a = b * q + r und deg(r) < deg(b)
*/
void polynomialDivisionZpX(int N, int p, const Polynom& a, const
    Polynom& b, Polynom& q, Polynom& r);
```

8. **Erweiterter Euklidscher Algorithmus in $\mathbb{Z}_p[x]$**

   Die Funktion berechnet den größten gemeinsamen Teiler d zweier Polynome a und b, sowie die Polynome u und v, sodass

   $$a * u + b * v = d \tag{3.9}$$

   gilt. Zur Veranschaulichung werden u, v und d der Polynome $3+x^3+x^5$ und $4 + x^3$ berechnet:

   $$u = 3x \tag{3.10}$$
   $$v = 1 + x + 2x^2 \tag{3.11}$$
   $$d = 4 + x \tag{3.12}$$

```
/*
*Algorithmus 4 – Erweiterter Euklidscher Algorithmus in Z_p[x].
*@int N Maximaler Grad von b
*@int p Der prime Modul
```

```
*@const Polynom& a Ein Polynom ungleich dem Nullpolynom
*@const Polynom& b Ein Polynom ungleich dem Nullpolynom
*@Polynom& u,v,d Polynome mit a*u+b*v=d und d=GCD(a,b)
*/
void polynomialGCDZpX(int N, int p, Polynom& a, Polynom& b,
    Polynom& u, Polynom& v, Polynom& d);
```

9. **Berechnung von Inversen in $\mathbb{Z}_p[x]/x^N - 1$**

   Im Grunde können mit dem erweiterten Euklidschen Algorithmus bereits Polynome invertiert werden. Koppelt man die Rückgabewerte der *polynomialGCDZpX*-Funktion an die Bedingung, dass der Wert d eine Konstante sein muss, so ist das Inverse, falls es existiert, schon gefunden. Wird die Gleichung 3.9 betrachtet, so ist

   $$v = b^{-1} \text{ in } \mathbb{Z}_p[x]/a,$$

   falls für das Polynom d gilt, dass es Grad 0 hat. So gilt z. B. für die Gleichung

   $$(2 + x^2 + x^3) * (3 + x) + (1 + x + x^2) * (1 + 2x + 4x^2) = 2.$$

   Bei der Berechnung des erweiterten Euklidschen Algorithmus ist daher $u = 3 + x$, $v = 1 + 2x + 4x^2$ und $d = 2$.

   Nach einer Normierung der Gleichung durch die Multiplikation mit 3 ergibt sich das Inverse von $1+x+x^2$ als $3+x+2x^2$ in $\mathbb{Z}_5[x]/2+x^2+x^3$.

```
/*
*Algorithmus 5 - Berechnung von Inversen in Z_p[x]/x^N-1.
*@int N Dimension der Polynome
*@int p Der prime Modul
*@const Polynom& a Polynom, das invertiert werden soll
*@const Polynom& b Das Inverse mit a*b=1 in Z_p[x]/x^N-1
*/
void polynomialInverseZpXN1(int N, int p, Polynom& a, Polynom& b);
```

10. **Umwandlung von Dezimalzahlen und *Bit Strings***

    (a) Die Umwandlung einer positiven Dezimalzahl in Binärschreibweise (d. h. zur Basis zwei) kann durch den Aufruf der *I2BSP*-Funktion durchgeführt werden. Die resultierende Binärzahl wird in diesem Fall als Feld von *char*-Elementen gespeichert, weshalb beim Aufruf der Funktion auch die Länge des Feldes mitgegeben werden muss, um eine saubere Nullterminierung garantieren zu können. Zu beachten ist, dass die Binärzahl als Zeichenkette (in *C* sind *Strings* ja nichts anderes als eine Folge von Elementen des Typs *char* und der Nullterminierung) abgelegt wird. Daher ist das Zeichen 0 nicht mit der Zahl 0, bzw. das Zeichen 1 mit der Zahl

1 (s. auch ASCII-Tabelle), gleichzusetzen. Dieser Umstand muss bei der Benutzung eines *Bit Strings* beachtet werden.

```
/*
*Algorithmus 6 – Umwandlung einer positiven Dezimalzal in einen
*Bit String bestimmter Länge.
*@unsigned int x Die zu konvertierende Zahl
*@int blen Länge des resultierenden Bit Strings
*@char* str Char–Feld, das Bit String enthält
*/
void I2BSP(unsigned int x, int blen, char* str);
```

(b) Die Zurückwandlung eines *Bit Strings* bzw. einer Binärzahl in eine vorzeichenlose Dezimalzahl geschieht durch die Funktion *BS2IP*. Die Binärzahl wird als Zeiger auf ein *char*-Feld an die Funktion übergeben und der Rückgabewert ist die gewünschte Dezimalzahl.

```
/*
*Algorithmus 7 – Umwandlung eines Bit Strings in eine
* positive Dezimalzahl.
*@char* str Char feld, das Bit String enthält
*@int blen Länge des Bit Strings
*@return Resultierende Dezimalzahl
*/
unsigned int BS2IP(char* str,int blen);
```

11. **Umwandlung von Dezimalzahlen und *Octet Strings***

Als *Octet String* wird eine Folge von Zahlen zwischen 0 und 256 verstanden, d. h. jede Zahl kann mit acht Bit dargestellt werden. Bei der Implementierung wird ein *Octet String* als *unsigned-char*-Feld dargestellt, um Speicherplatz zu sparen und weil die Hashfunktion für diesen Datentyp ausgelegt ist.

(a) Die *I2OSP*-Funktion wandelt eine positive Dezimalzahl in einen *Octet String* um. Wie schon bei der Umwandlung in eine Binärzahl wird die Länge des resultierenden *Octet Strings* mitübergeben.

```
/*
*Algorithmus 8 – Umwandlung einer positiven Dezimalzal in einen
*Octet String bestimmter Länge.
*@unsigned int x Die zu konvertierende Zahl
*@int olen Länge des resultierenden Octet Strings
*@unsigned char* ostr Unsigned–char–Feld, das Octet String enthält
*/
void I2OSP(unsigned int x, int olen, unsigned char* ostr);
```

(b) Soll ein *Octet String* zurück in eine positive Dezimalzahl umgewandelt werden, so kann dafür die Funktion *OS2IP* benutzt werden. Der Funktion wird der *Octet String* und dessen Länge übergeben, um daraus den Dezimalwert zu berechnen.

```
/*
*Algorithmus 9 - Umwandlung eines Octet Strings in eine
*positive Dezimalzahl.
*@unsigned char* str Unsigned-char-Feld, das Octet String enthält
*@int olen Länge des Octet Strings
*@return Resultierende Dezimalzahl
*/
unsigned int OS2IP(unsigned char* ostr,int olen);
```

12. **Umwandlung von *Bit Strings* und *right–padded Octet Strings***

    (a) Wird ein *Bit String* in einen *Octet String* konvertiert, so muss zuvor ein Padding des *Bit Strings* erfolgen, damit dessen Länge ein Vielfaches von 8 ist. Die Funktion *BS2ROSP* übernimmt dieses Auffüllen sowie die Umrechnung zur Basis 256.

```
/*
*Algorithmus 10 - Umwandlung eines Bit Strings in einen
*Octet String bestimmter Länge.
*@char* str Der zu konvertierende Bit String
*@int olen Länge des resultierenden Octet Strings
*@unsigned char* ostr Feld, das den Octet String enthält
*/
void BS2ROSP(char* str, int olen, unsigned char* ostr);
```

    (b) Wird ein *Octet String* in einen *Bit String* umgewandelt, so erfolgt dies durch umrechnen der einzelnen Elemente in die Binärschreibweise. Benötigt ein Element des *Octet Strings* weniger als 8 Bits, so werden führende Nullen angehängt, um das Byte zu vervollständigen.

```
/*
*Algorithmus 11 - Umwandlung eines Octet Strings in
*einen Bit String bestimmter Länge.
*@unsigned char* ostr Der zu konvertierende Octet String
*@int olen Länge des Octet Strings
*@int blen Länge des Bit Strings
*@char* str Char-Feld, das Bit String enthält
*/
void ROS2BSP(unsigned char* ostr, int olen, int blen, char*
    str);
```

13. **Umwandlung von Polynomen und *Octet Strings***

    (a) Um aus einem Polynom einen *Octet String* zu formen muss beachtet werden, dass der Modul der Koeffizienten in der Darstellung zur Basis 256 auch mehr als ein Oktet benötigen kann. Ist der Modul q z. B. 351, so können zwei Oktetten pro Koeffizient benötigt werden, der resultierende *Octet String* wird daher, wenn

das Polynom Grad N besitzt, $N * 2$ Elemente haben. Die Funktion *RE2OSP* berechnet wieviel Oktetten pro Koeffizient benötigt werden und wandelt das Polynom in einen *Octet String* um. Angenommen das Polynom

$$220 + 297x + 312x^2 + 25x^3$$

soll umgewandelt werden. Die Koeffizienten 297 und 312 benötigen zwei Oktetten, der *Octet String* ist daher

$$\begin{bmatrix} 0 & 220 & 1 & 41 & 1 & 56 & 0 & 25 \end{bmatrix}_{256}. \tag{3.13}$$

```
/*
*Algorithmus 12 – Umwandlung eines (q,N) Polynoms
*in einen Octet String.
*@const Polynom& a Das zu konvertierende Polynom
*@int N Maximaler Grad
*@int q Modul (Koeffizienten des Polynoms zwischen
*0 und q−1)
*@return Der resultierende Octet String
*/
unsigned char* RE2OSP(const Polynom& a, int N, int q);
```

(b) Der umgekehrte Weg, aus einem *Octet String* ein Polynom zu erzeugen, wirft wieder die Problematik auf, dass der Modul q mehr als ein Oktet zur Darstellung benötigen kann. Je nach Größe des Moduls wird immer die entsprechende Anzahl an Elementen des *Octet Strings* genommen und in einen Dezimahlwert umgewandelt.

```
/*
*Algorithmus 13 – Umwandlung eines Octet Strings in ein
*Polynom.
*@unsigned char* ostr Der zu konvertierende Octet String
*@int olen Länge des Octet Strings
*@int N Maximaler Grad
*@int q Modul (Koeffizienten des Polynoms zwischen
*0 und q−1)
*@Polynom& a Das resultierende Polynom
*/
void OS2REP(unsigned char* ostr, int olen, int N, int q,
    Polynom& a);
```

14. **Umwandlung von Polynomen und *Bit Strings***

(a) Die Funktion *RE2BSP* wandelt ein Ringelement in *Bit Strings* mit einer bestimmten Länge um. Dabei wird berechnet wieviele Bits nötig sind, um den Modul q darzustellen und anschließend

jeder Koeffizient der Methode *I2BSP* übergeben. Die Gleichung 3.14 zeigt ein Beispiel mit $N = 3$ und $q = 11$.

$$3 + 12x + 7x^2 = \begin{bmatrix} 0011 & 0001 & 0111 \end{bmatrix}_2 \tag{3.14}$$

```
/*
*Algorithmus 14 - Umwandlung eines (q,N) Polynoms
*in einen Bit String.
*@const Polynom& a Das zu konvertierende Polynom
*@int N Maximaler Grad
*@int q Modul (Koeffizienten des Polynoms zwischen
*0 und q-1)
*@return Der resultierende Bit String
*/
char* RE2BSP(const Polynom& a, int N, int q);
```

(b) Genauso kann ein *Bit String* wieder zurück in ein Polynom umgewandelt werden. Dabei werden wiederum die Parameter N und q an die Funktion übergeben, um aus den einzelnen Bits ein Polynom mit den richtigen Koeffizienten herzustellen.

```
/*
*Algorithmus 15 - Umwandlung eines Bit Strings in ein
*Polynom.
*@char* str Der zu konvertierende Bit String
*@int N Maximaler Grad
*@int q Modul (Koeffizienten des Polynoms zwischen
*0 und q-1)
*@Polynom& a Das resultierende Polynom
*/
void BS2REP(char* str, int N, int q, Polynom& a);
```

15. **Umwandlung von Polynomen in Binärform und *Octet Strings***

(a) Bei der Konvertierung eines Polynoms in Binärform in einen *Octet String* werden immer acht Koeffizienten, die acht Bits repräsentieren, in eine Dezimalzahl umgerechnet und nacheinander in einem *Octet String* gespeichert.

```
/*
*Algorithmus 16 - Umwandlung eines Polynoms in
*Binärform in einen Octet String.
*@const Polynom& a Das zu konvertierende Polynom
*@int N Maximaler Grad
*@int* oLen Zeiger auf die resultierende Länge
*des Octet Strings
*@return Der resultierende Octet String oder NULL
*/
unsigned char* BRE2OSP(const Polynom& a, int N, int* oLen);
```

(b) Bei der Konvertierung eines *Octet Strings* in ein binäres Polynom a muss die Zuordnung von *low-order* und *high-order* beachtet werden. Hierzu ein Beispiel der Funktion *OS2BREP*:

$$\begin{bmatrix}197 & 25\end{bmatrix}_{256} = \begin{bmatrix}11000101 & 00011001\end{bmatrix}_2 \tag{3.15}$$

Anschließend werden die Koeffizienten von a mit den einzelnen Bits der Bytes belegt. Es wird hierbei mit dem ersten Byte

$$o_0 = (b_{0,7}, b_{0,6}, \cdots, b_{0,0})$$

begonnen, dessen Bits dem Polynom a wie folgt zugeordnet werden:

$$a_0 = b_{0,7}, a_1 = b_{0,6}, \cdots, a_7 = b_{0,0}.$$

Die Zuordnung der restlichen Bits wird analog durchgeführt. Der oben angeführte *Bit String*, mit $N = 16$, ergibt daher das Polynom

$$1 + x^2 + x^6 + x^7 + x^8 + x^{11} + x^{12}. \tag{3.16}$$

```
/*
*Algorithmus 17 - Umwandlung eines Octet Strings in ein
*Polynom in Binärform.
*@unsigned char* ostr Der zu konvertierende Octet String
*@int olen Länge des Octet Strings
*@int N Maximaler Grad
*@Polynom& a Das resultierende Polynom
*/
void OS2BREP(unsigned char* ostr, int olen, int N, Polynom& a
    );
```

16. ***Mask generation function***

Eine *Mask generation function* (MGF) ist vergleichbar mit einer Hashfunktion. Der Unterschied besteht darin, dass eine MGF, im Gegensatz zu der Hashfunktion, keinen Wert von fixer Länge produziert. Kernbestandteil der MGF ist eine Hashfunktion, die als Parameter einen *Octet String* und eine Zählervariable erhält, um daraus einen Hashwert zu bilden. Die einzelnen Hashwerte werden in jeder Runde, die die Funktion durchläuft, aneinandergehängt und am Ende wird die gewünschte Anzahl an Bytes in Form eines *Octet Strings* zurückgegeben.

Die Parametersets geben vor welche Hashfunktion mit welchen Parametern verwendet werden darf. In Tabelle 3.1 auf Seite 40 wird z. B. vorgegeben, dass SHA-1 als Hashfunktion verwendet werden muss.

```
/*
*Algorithmus 18 - Mask generation function mit
*SHA-1 als Hashfunktion.
```

```
*@unsigned char* Z Der Input Octet String
*@int zLen Länge des Octet Strings
*@unsigned char* mask Die Maske in Form eines Octet Strings
*@int oLen Gewünschte Länge der Maske
*/
void MGF1(unsigned char* Z, int zLen, unsigned char* mask, int
    olen);
```

17. ***Index generation function***

Eine *Index generation function* (IGF) erzeugt bei jedem Aufruf einen ganzzahligen Wert, der im Intervall $[0, N-1]$ liegt. Die IGF wird mit einem *Octet String* als *seed* initialisiert, um beim ersten Aufruf den internen Status, siehe Programmstück 3.2, mit Werten zu belegen. Bei mehrmaligem Aufrufen wird der Funktion nur mehr der aktuelle Status übergeben, dies ist auch der Grund warum für die IGF zwei seperate Funktionen existieren.

Wichtig ist, dass sich die IGF bei einer Initialisierung mit gleichem *seed* deterministisch verhält. Dies garantiert auch, dass beim Ver- und Entschlüsseln dasselbe Blendpolynom zur Kontrolle erstellt werden kann.

```
/*
*Algorithmus 19 - IGFMGF1: Index generation function mit
*SHA-1 als Hashfunktion.
*IGF-MGF1 wird dazu benutzt, um den internen Status zu
* initialisieren .
*@unsigned char* Z Ein Octet String als seed
*@int zLen Länge von Z
*@int N Der Modul
*@int c Die index generation Konstante
*@int minCalls Anzahl der Aufrufe der Hashfunktion
*@unsigned int* iResult Die resultierende Dezimalzahl
*@return Der initialisierte Status
*/
struct state* IGFMGF1(unsigned char* Z, int zLen, int N, int c,
    int minCalls, unsigned int* iResult);
```

```
/*
*Algorithmus 19 - IGFMGF2: Die Index generation function
*nach Initialisierung des internen Status.
*@struct state* s Der aktuelle Status
*@unsigned int* iResult Die resultierende Dezimalzahl
*/
void IGFMGF2(struct state* s, unsigned int* iResult);
```

18. **Erstellung eines Blendpolynoms**

(a) Im Verschlüsselungsschema des IEEE-P1363.1-Standards wird ein Blendolynom r in Binärform verwendet, um *plaintext awareness* (siehe Abschnitt 2.1 und 3.1.1) zu garantieren. Für die Erzeugung

```
struct state{
  int totLen;
  int remLen;
  unsigned char* buf;
  int bufLen;
  int counter;
  int N;
  int NLen;
  int c;
  unsigned char* seed;
  int sLen;
};
```

Programm 3.2: Der interne Status der IGF

des Polynoms wird die IGF aufgerufen, um die Positionen der Koeffizienten von r zu erhalten, die gleich 1 sein sollen.

```
/*
*Algorithmus 20 - Erstellung eines Blendpolynoms r
*in Binärform.
*@unsigned char* seed Ein Octet String als seed
*@int sLen Länge des seeds
*@int N Maximaler Grad
*@int c Die index generation Konstante
*@int dr Anzahl der Koeffizienten von r, die
*ungleich 0 sind
*@int coeffs [] Feld der Koeffizientenposition von r,
*die ungleich 0 sind
*@int minCalls Minimale Anzahl der Aufrufe der Hashfunktion
*/
void LBPBPGM1(unsigned char* seed, int sLen, int N, int c,
    int dr, int coeffs[], int minCalls);
```

(b) Die zweite Funktion erstellt ein Blendpolynom r in Produktform, d. h. es werden 3 binäre Polynome $r1, r2$ und $r2$ erstellt und

$$r1 * r2 + r3$$

berechnet.

```
/*
*Algorithmus 21 - Erstellung eines Blendpolynoms r
*in Produktform (r=r1*r2+r3).
*@unsigned char* seed Ein zufälliger Octet String
*@int sLen Länge des Octet Strings
*@int N Maximaler Grad
*@int c Die index generation Konstante
*@int dr1,dr2,dr3 Anzahl der Koeffizienten von r1, r2 und r3,
*die ungleich 0 sind
*@int minCalls Minimale Anzahl der Aufrufe der Hashfunktion
```

```
*@int coeffs1 [], coeffs2 [], coeffs3 [] Felder der
* Koeffizientenpositionen von r1,r2 und r3, die ungleich 0 sind
*/
void LBPBPGM2(unsigned char* seed, int sLen, int N, int c,
    int dr1, int dr2, int dr3, int minCalls, int coeffs1[],
    int coeffs2[], int coeffs3[]);
```

19. **Erstellung der Schlüssel**

Der IEEE-P1363.1-Standard stellt zwei verschiedene Arten der Schlüsselerstellung zur Verfügung. Grundsätzlich kann als privater Schlüssel jedes kleine Polynom f genommen werden, dass in $\mathbb{Z}_p[x]/x^N - 1$ und $\mathbb{Z}_q[x]/x^N - 1$ invertierbar ist. Nichtsdestotrotz schreibt der Standard zur Verbesserung der Effektivität vor, dass das Polynom f die Form $f = 1 + pF$, wobei p der kleine Modul und F ein Polynom vom Grad $N - 1$ ist, hat [11, S. 31].

Für die Rückgabe der Schlüssel an die aufrufende Funktion wird ein Zeiger auf ein struct des Typs *keypairT1* oder *keypairT2* verwendet (vgl. Programmstück 3.3).

(a) Typ-1: Ein privater Schlüssel dieses Typs wird erstellt aus einem Polynom F vom Grad $N-1$, das *df* zufällige Koeffizientenpositionen ungleich 0 besitzt. Der Paramter *df* wird von den Parametersets vorgegeben. Im nächsten Schritt wird der eigentliche private Schlüssel, das Polynom f, aus $1+p*F$ in $\mathbb{Z}_q[x]/x^N-1$ berechnet. Zur Berechnung des öffentlichen Schlüssels siehe Gleichung 3.3 auf Seite 16.

```
/*
*Algorithmus 22 - Erstellung eines Schlüsselpaares vom Typ-1.
*@domain parameters: N,q,p,dF,dg
*@int dF Anzahl der Koeffizienten in F ungleich 0
*@int dg Anzahl der Koeffizienten in g ungleich 0
*@struct keypairT1* k Struct in dem die Schlüssel gespeichert
    werden
*/
void LBPKGP1(int N, int q, int p, int dF, int dg, struct
    keypairT1* k);
```

(b) Typ-2: Ein privater Schlüssel vom Typ-2 wird erstellt aus einem Polynom F in Produktform. Dazu werden die Parameter $df1, df2$ und $df3$, welche die Anzahl der Koeffizienten ungleich 0 vorgeben, verwendet, um die Polynome f_1, f_2 und f_3 zu erstellen. Aus diesen wird das Polynom $F = f_1 * f_2 + f_3$ in $\mathbb{Z}_q[x]/x^N - 1$ berechnet. Der eigentliche private Schlüssel f wiederum, ergibt sich aus $1 + p * F$ in $\mathbb{Z}_q[x]/x^N - 1$.

Die Berechnung des öffentlichen Schlüssels erfolgt analog zu jener, die bei der Erstellung eines Typ-1 Schlüsselpaares verwendet wird.

```
struct keypairT1{
  int* dfs;//Positionen von F ungleich 0
  int dF;//Anzahl der Positionen von F ungleich 0
  Polynom f; //Teil des privanten Schlüssels
  Polynom h; //Öffentlicher Schlüssel
  Polynom finv; //Das Inverse von f
  Polynom g; //Teil des öffentlichen Schlüssels
  int* dgs; //Positionen von g ungleich 0
  int dg;//Anzahl der Positionen von g ungleich 0
};

struct keypairT2{
  Polynom f1; //Teil des privaten Schlüssels
  int* df1s; //Positionen von f1 ungleich 0
  int df1;//Anzahl der Positionen von f1 ungleich 0
  Polynom f2; //Teil des privaten Schlüssels
  int* df2s; //Positionen von f2 ungleich 0
  int df2;//Anzahl der Positionen von f2 ungleich 0
  Polynom f3; //Teil des privaten Schlüssels
  int* df3s; //Positionen von f3 ungleich 0
  int df3;//Anzahl der Positionen von f3 ungleich 0
  Polynom h; //Öffentlicher Schlüssel
  Polynom finv; //Das Inverse von f
  Polynom g; //Teil des öffentlichen Schlüssels
  int* dgs; //Positionen von g ungleich 0
  int dg;//Anzahl der Positionen von g ungleich 0
};
```

Programm 3.3: Aufbau der structs für die Schlüssel

```
/*
*Algorithmus 23 - Erstellung eines Schlüsselpaares vom Typ-2.
*@domain parameters: N,q,p,df1,df2,df3,dg
*@int df1 Anzahl der Koeffizienten in f_1 ungleich 0
*@int df2 Anzahl der Koeffizienten in f_2 ungleich 0
*@int df3 Anzahl der Koeffizienten in f_3 ungleich 0
*@int dg Anzahl der Koeffizienten in g ungleich 0
*@struct keypairT2* k Struct in dem die Schlüssel gespeichert
    werden
*/
void LBPKGP2(int N, int q, int p, int df1, int df2, int df3,
    int dg, struct keypairT2* k);
```

20. **Berechnung des Klartextkandidaten:**

    (a) Die erste Funktion berechnet den Klartextkandidaten

$$a = f * e$$

in $\mathbb{Z}_q[x]/x^N - 1$ mit anschließender Reduktion modulo p, falls der richtige private Schlüssel übergeben wurde.
Bei der Implementierung wird darauf geachtet, dass $f = 1 + p * F$ und F ein binäres Polynom ist. Das heißt a kann auch durch $a = e + e * p * f$ dargestellt werden. Dies ist insofern von Bedeutung, da nun für die Multiplikation von e und F die Funktion *multiplyByBinaryPolynomZqXN1* verwendet werden kann, dieser Ausdruck dann mal p gerechnet und mit e addiert wird (die Berechnungen finden jeweils in $\mathbb{Z}_q[x]$ statt). Im letzten Schritt wird das Ergebnis der Addition modulo p gerechnet. So kommt die Berechnung von a ohne Reduktion in $\mathbb{Z}_q[x]/x^N - 1$ aus, was eine höhere Effizienz garantiert.

```
/*
*Algorithmus 24 - Berechnung des
*Klartextkandidaten Typ-1.
*@domain parameters: N,q,p
*@int dF Anzahl der Positionen von F ungleich 0
*@int* dfs Positionen von F ungleich 0
*@Polynom& e Die verschlüsselte Nachricht
*@Polynom& i Der Klartextkandidat
*/
void LBPDP1(int N, int q, int p, int dF, int* dfs, Polynom& e
    , Polynom& i);
```

(b) Die zweite Funktion berechnet den Klartextkandidaten für eine Verschlüsselung, bei der Schlüssel vom Typ-2 verwendet werden. In diesem Zusammenhang wird daher für die Berechnung von a die Funktion *multiplyByProductFormPolynomZqXN1* verwendet. Ansonsten wird a analog zur Variante mit Schlüsseln des Typs-1 berechnet.

```
/*
*Algorithmus 24 - Berechnung des
*Klartextkandidaten Typ-2.
*@domain parameters: N,q,p
*@int df1, df2, df3 Anzahl der Positionen
*von f_1, f_2 und f_3 ungleich 0
*@int* df1s, df2s, df3s Positionen von f_1, f_2
*und f_3 ungleich 0
*@Polynom& e Die verschlüsselte Nachricht
*@Polynom& i Der Klartextkandidat
*/
void LBPDP2(int N, int q, int p, int df1, int* df1s, int df2,
     int* df2s, int df3, int* df3s, Polynom& e, Polynom& i);
```

21. **Verschlüsselung:**

(a) Die erste Art der Verschlüsselung verwendet ein Schlüsselpaar des Typs-1, ein Blendpolynom in Binärform und die Funktion

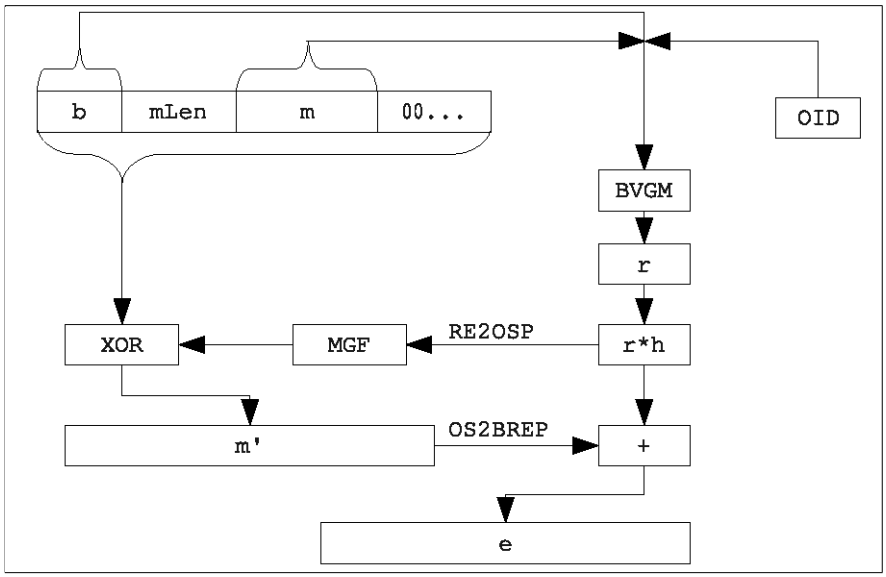

Abbildung 3.1: Operationen bei der Verschlüsselung (nach [11, S. 39]).

LBPDP1 zur Berechnung des Klartextkandidaten. Es ist wichtig darauf zu achten, wenn ein bestimmtes Parameterset des Standards verwendet wird, mit welcher Art von Schlüsseln und Polynomen der Algorithmus durchgeführt wird. Um nach der Entschlüsselung überprüfen zu können, ob diese richtig funktioniert hat, wird der *Octet String sData* aus folgenden Teilen geformt:

$$sData = [b||octL||m||00...].$$

Hierbei ist b ein zufällig gewählter *Octet String*, octL die Länge von m in Bytes, m die Nachricht selbst und anschließend eine bestimmte Anzahl von Nullbytes. Im Anschluss daran wird *sData* als *seed* für die Erstellung des Blendpolynoms r in Binärform verwendet, welches mit dem öffentlichen Schlüssel h multipliziert wird. Danach wird der Ausdruck $r * h$ modulo 2 gerechnet und der *Mask generation function* übergeben und mit Teilen von *sData* XOR gerechnet. Der daraus resultierende *Octet String* wird in ein binäres Polynom umgerechnet und zu $r * h$ modulo q addiert, was als Ergebnis die verschlüsselte Nachricht e ergibt.

Grafisch kann die Verschlüsselungsoperation wie in Abbildung 3.1 visualisiert werden.

```
/*
*Algorithmus 26 – Encryption operation: Verschlüsselung eines
*Octet Strings (Schlüssel vom Typ−1).
*@int N Maximaler Grad der Polynome
```

```
*@int q Der Modul
*@int db Größe des Zufallsbuffers in Bits
*@int pkLen Anzahl der Bits des öffentlichen Schlüssels,
*die in den Hashwert miteinfließen sollen
*@unsigned char* oid Octet String, der Parameterset kennzeichnet
*@int c Die index generation Konstante
*@int dr Anzahl der Koeffizienten ungleich 0 in r
*@int minCalls Minimale Anzahl der Aufrufe der Hashfunktion
*@int dm0 Untere Schranke für das vorkommen von 1
*im Klartextkandidaten
*@unsigned char* m Die Nachricht, ein Octet String
*@int mLen Die Länge von m
*@Polynom& h Der öffentliche Schlüssel (Typ-1)
*@Polynom& e Die resultierende verschlüsselte Nachricht
*/
void encryptionT1(int N, int q, int db, int pkLen, unsigned
     char* oid, int c, int dr, int minCalls, int dm0, unsigned
      char* m, int mLen, Polynom& h, Polynom& e)
```

(b) Die zweite Variante der Verschlüsselung erwartet einen öffentlichen Schlüssel, der zu einem privaten Schlüssel des Typs-2 gehört, ansonsten schlägt die Entschlüsselung fehl (schließlich wird für die Berechnung des Klartextkandidaten die Funktion *LBPDP2* verwendet).

```
/*
*Algorithmus 26 - Encryption operation: Verschlüsselung eines
*Octet Strings (Schlüssel vom Typ-2).
*@int N Maximaler Grad der Polynome
*@int q Der Modul
*@int db Größe des Zufallsbuffers in Bits
*@int pkLen Anzahl der Bits des öffentlichen Schlüssels,
*die in den Hashwert miteinfließen sollen
*@unsigned char* oid Octet String, der Parameterset kennzeichnet
*@int c Die index generation Konstante
*@int dr Anzahl der Koeffizienten ungleich 0 in r
*@int minCalls Minimale Anzahl der Aufrufe der Hashfunktion
*@int dm0 Untere Schranke für das vorkommen von 1
*im Klartextkandidaten
*@unsigned char* m Die Nachricht, ein Octet String
*@int mLen Die Länge von m
*@Polynom& h Der öffentliche Schlüssel (Typ-2)
*@Polynom& e Die resultierende verschlüsselte Nachricht
*/
void encryptionT2(int N, int q, int db, int pkLen, unsigned
     char* oid, int c, int dr, int minCalls, int dm0, unsigned
      char* m, int mLen, Polynom& h, Polynom& e)
```

22. **Entschlüsselung:**

(a) Wird eine Nachricht m mittels *encryptionT1* verschlüsselt, so wird bei der Entschlüsselung von e die Funktion *LBPDP1* auf-

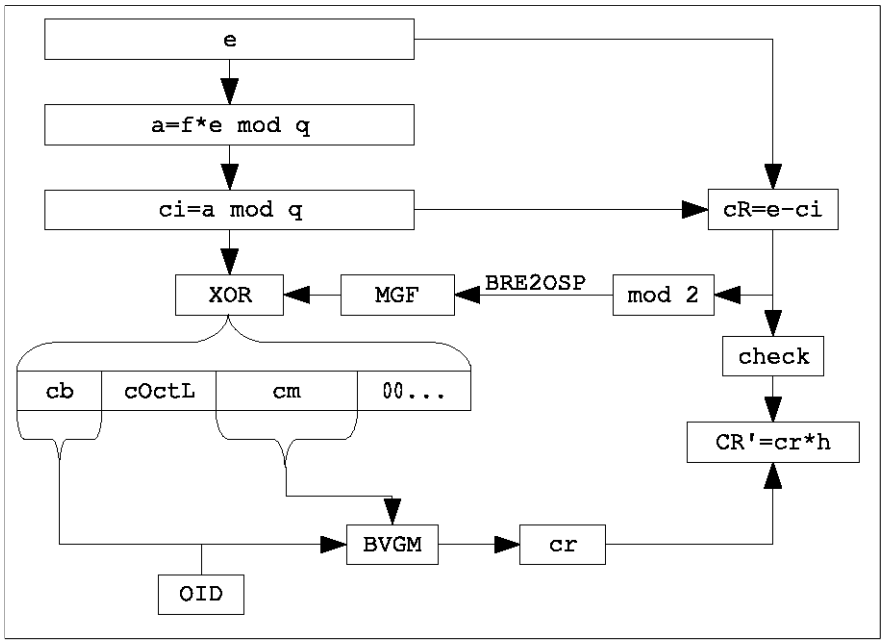

Abbildung 3.2: Operationen bei der Entschlüsselung (nach [11, S. 41]).

gerufen, um den Klartextkandidaten *ci* zu erhalten. Anschließend erfolgt, in umgekehrter Reihenfolge, die Reproduzierung des *Octet Strings sData*, um diesen auf eine fehlerhafte Entschlüsselung zu überprüfen. *sData* wird außerdem dazu verwendet, um das Blendpolynom r zu berechnen und zu kontrollieren, ob $e - ci$ gleich $r * h$ ist. Grafisch kann die Entschlüsselungsoperation wie in Abbildung 3.2 dargestellt werden.

```
/*
*Algorithmus 27 - Decryption operation: Entschlüsselung eines
*Octet Strings (Schlüssel vom Typ-1).
*@int N Maximaler Grad der Polynome
*@int q Der große Modul
*@int p Der kleine Modul
*@int db Größe des Zufallsbuffers in Bits
*@int pkLen Anzahl der Bits des öffentlichen Schlüssels,
*die in den Hashwert miteinfließen sollen
*@unsigned char* oid Octet String, der Parameterset kennzeichnet
*@int c Die index generation Konstante
*@int minCalls Minimale Anzahl der Aufrufe der Hashfunktion
*@int dr Anzahl der Koeffizienten ungleich 0 in r
*@int dm0 Untere Schranke für das vorkommen von 1
*im Klartextkandidaten
*@Polynom& e Die verschlüsselte Nachricht
*@struct keypairT1* k Ein Schlüssel des Typs-1
*@return Ein Zeiger auf die entschlüsselte
```

```
*Nachricht als Octet String
*/
unsigned char* decryptionT1(int N, int q, int p, int db, int
    pkLen, unsigned char* oid, int c,int minCalls, int dr,
    int dm0, Polynom& e, struct keypairT1* k);
```

(b) Die zweite Entschlüsselungsfunktion arbeitet mit einem Schlüssel des Typs-2 und einem Blendpolynom in Produktform. Das heißt für die Berechnung des Klartextkandidaten wird die Funktion *LBPDP2* und für die Berechung des Blendpolynoms *LBPBPGM2* aufgerufen. Da in den Parametersets des Standards alle drei Polynome, aus denen das Blendpolynom erstellt wird, eine gleiche Anzahl an Koeffizienten ungleich 0 aufweisen, genügt es, den Parameter *dr* einmal an die Entschlüsselungsfunktion zu übergeben. Ansonsten erfolgen die Berechnungen analog zu *decryptionT1*.

```
/*
*Algorithmus 27 - Decryption operation: Entschlüsselung eines
*Octet Strings (Schlüssel vom Typ-2).
*@int N Maximaler Grad der Polynome
*@int q Der große Modul
*@int p Der kleine Modul
*@int db Größe des Zufallsbuffers in Bits
*@int pkLen Anzahl der Bits des öffentlichen Schlüssels,
*die in den Hashwert miteinfließen sollen
*@unsigned char* oid Octet String, der Parameterset kennzeichnet
*@int c Die index generation Konstante
*@int minCalls Minimale Anzahl der Aufrufe der Hashfunktion
*@int dr Anzahl der Koeffizienten ungleich 0 in r
*@int dm0 Untere Schranke für das vorkommen von 1
*im Klartextkandidaten
*@Polynom& e Die verschlüsselte Nachricht
*@struct keypairT2* k Ein Schlüssel des Typs-2
*@return Ein Zeiger auf die entschlüsselte
*Nachricht als Octet String
*/
unsigned char* decryptionT2(int N, int q, int p, int db, int
    pkLen, unsigned char* oid, int c,int minCalls, int dr,
    int dm0, Polynom& e, struct keypairT2* k);
```

3.1.6 Wahl der Parameter

Der Annex A.5 des IEEE-P1363.1-Standards stellt verschiedene Parametersets für die Anwendung bei der Implementierung zur Verfügung. Die jeweiligen Gruppen von Parametern sollen als Einheit verwendet und nicht untereinander vermischt werden, um Sicherheit und Effizienz zu gewährleisten. Die im Laufe der Bakkalaureatsrarbeit entstandene Implementierung unterstützt alle Parametersets, die als Hashfunktion SHA-1 verwenden. Die Tabelle 3.1 stellt solch ein Parameterset dar.

$N = 251$	
$p = 2$	
$q = 197$	
Schlüsselerzeugung:	LBP-KGP-1 mit
	$dF = 48$
	$dg = 125$
$lLen = 1$	
$db = 80$	
$dm0 = 70$	
MGF1 mit:	
	SHA-1 (MGF)
LBP-BPGM1 mit:	
	IGF-MGF1 und SHA-1 (IGF)
$dr = 48$	
$c = 8$	
$oLenMin = 120$	
$OID = 000106$	
$pkLen = 0$	
$A = 0$	

Tabelle 3.1: Beispiel für ein Parameterset.

Zur Abbildung 3.1 folgt außerdem eine Beschreibung der verwendeten Parameter (vgl. [11, S. 30ff]).

N ... Maximaler Grad der Polynome
p ... Der kleine Modul
q ... Der große Modul
dF ... Anzahl der Koeffizienten von f ungleich 0
dg ... Anzahl der Koeffizienten von g ungleich 0
$lLen$ ... Die Länge des Feldes für die Länge der kodierten Nachricht
db ... Größe des Zufallsbuffers in Bits
$dm0$ ... Untere Schranke für das Vorkommen von 1 im Klartextkandidaten
dr ... Anzahl der Koeffizienten des Blendpolynoms r ungleich 0
c ... Die *index generation* Konstante für die IGF
$oLenMin$ ... Minimale Anzahl der Oktetten für die Ausgabe der IGF
OID ... Ein drei Byte langer String als *seed* für die IGF
$pkLen$ ... Anzahl der Bits des öffentlichen Schlüssels,
aus denen ein Hashwert erzeugt werden soll
A ... Untere Schranke für eine Reduktion modulo q

3.2 NTRUsign

Neben dem P1363.1-Standard wird als zweiter Algorithmus NTRUsign vorgestellt, mit dem Signaturen, unter Verwendung eines erweiterten privaten Schlüssels nach P1363.1, berechnet werden können (nähere Informationen zum Hintergrund von NTRUsign finden sich in [18, S. 2ff]).

Grundsätzlich arbeitet NTRUsign mit denselben Parametern und Polynomen wie der IEEE-P1363.1-Standard, wenn davon abgesehen wird, dass zur Vervollständigung des privaten Schlüssels noch zwei weitere Teile berechnet werden. Die Sicherheit des Algorithmus wird von der Schwierigkeit der Lösung eines *approximate* CVP in dem modularen Gitter

$$M = \begin{pmatrix} I & H \\ 0 & qI \end{pmatrix}$$

ohne den privaten Schlüssel gewährleistet [8]. Dieses spezielle Gitter wird im Zusammenhang mit NTRUsign auch als NTRU-Gitter bezeichnet (s. auch Abschnitt 3.1.2). Die Kernidee von NTRUsign ist dabei folgende:

- Der private Schlüssel des Signierers ist eine kleine Basis des Gitters.
- Der öffentliche Schlüssel ist eine weitere, größere, sozusagen „schlechtere“ Basis desselben Gitters.

Die generierte Signatur weist folgende Eigenschaften auf:

1. Die Signatur wird dem originalen Dokument beigelegt.
2. Die Signatur ist ein Punkt des Gitters.
3. Die Signatur ist ein Punkt im Gitter, möglichst nahe bei der Nachricht. Die Signatur zeigt daher dem, der die Signatur überprüft, dass der Signierende das CVP näherungsweise lösen kann. Würde der öffentliche Schlüssel zum Signieren verwendet werden, so würde die Fehlerrate bzw. der Betrag des Abstands zum Punkt der Nachricht viel größer sein.

Wie am Anfang erwähnt ist ein in P1363.1 verwendeter privater Schlüssel für die Verwendung in NTRUsign noch nicht vollständig. Es werden zwei weitere Polynome F und G berechnet, sodass

$$f * G - g * F = q$$

in $\mathbb{Z}_q[x]/x^N - 1$ gilt, wobei f und g die bereits bekannten Teile des privaten Schlüssels sind. Dieses Quadrupel ist nun der vollständige private Schlüssel.

Die Matrix R:

$$R = \begin{pmatrix} f_0 & f_1 & \cdots & f_{n-1} & g_0 & g_1 & \cdots & g_{n-1} \\ f_{n-1} & f_0 & \cdots & f_{n-2} & g_{n-1} & g_0 & \cdots & g_{n-2} \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ f_1 & \cdots & f_{n-1} & f_0 & g_1 & \cdots & g_{n-1} & g_0 \\ F_0 & F_1 & \cdots & F_{n-1} & G_0 & G_1 & \cdots & G_{n-1} \\ F_{n-1} & F_0 & \cdots & F_{n-2} & G_{n-1} & G_0 & \cdots & G_{n-2} \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ F_1 & \cdots & F_{n-1} & F_0 & G_1 & \cdots & G_{n-1} & G_0 \end{pmatrix}$$

erzeugt ein Gitter aus den entsprechenden geheimen Komponenten (f_i kennzeichnet den Koeffizienten X^i im Polynom f). Da $f * h = g$ in $\mathbb{Z}_q[x]/x^N - 1$ gilt, ist h Teil einer Gitterbasis, die veröffentlicht werden kann, ohne Information über die geheimen Komponenten preis zu geben. [18, S. 7ff]:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & h_0 & h_1 & \cdots & h_{n-1} \\ 0 & 1 & \cdots & 0 & h_{n-1} & h_0 & \cdots & h_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & h_1 & \cdots & h_{n-1} & h_0 \\ 0 & 0 & \cdots & 0 & q & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & q & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & q \end{pmatrix}.$$

Signieren: Um eine Nachricht m zu signieren wird ein Hashwert (m_1, m_2) von m berechnet, mit m_1 und $m_2 \in \mathbb{Z}_q[x]/x^N - 1$. Anschließend berechnet der Signierer (nähere Informationen dazu finden sich in [8, S. 5ff]) A, C, a und c aus den Gleichungen :

$$\begin{aligned} G * m_1 - F * m_2 &= A + q * C, \\ -g * m_1 + f * m_2 &= a + q * c, \end{aligned}$$

wobei $A, C, a, c \in \mathbb{Z}_q[x]/x^N - 1$ sind. Mit den einzelnen Komponenten wird die Sigantur

$$s = f * C + F * c$$

modulo q berechnet [7].

Verifizieren: Derjenige, der die Signatur überprüfen will, berechnet mit dem zur Verfügung stehenden öffentlichen Schlüssel

$$t = s * h \pmod{q}$$

und überprüft, ob (s,t) nahe genug bei (m_1,m_2) liegt. Genauer gesagt, ob

$$\|s-m_1\|^2+\|t-m_2\|^2 \leq \text{ Normschranke}$$

ist.

Kapitel 4

Eigenschaften der Algorithmen

4.1 Aufwand bei Berechnungen

Die häufigste Operation bei der Verwendung des P1363.1-Standards ist die Multplikation zweier Polynome in $\mathbb{Z}_q[x]/x^N - 1$. Da diese Multiplikation ein wesentlicher Bestandteil bei der Erstellung von Schlüsseln bzw. bei Ver- und Entschlüsselung ist, rückt die Performance wesentlich in den Vordergrund. Deshalb wird im folgenden Abschnitt der Aufwand bei der Multiplikation zweier Polynome analysiert[1]:

Zuerst ein Beispiel mit Polynomen vom Grad 2 in $\mathbb{Z}_{97}[x]/x^3 - 1$, wobei $a = 3 + 5x + 6x^2$ und $b = 5 + 2x + 3x^2$ ist. Das Ergebnis der Multiplikation von a und b kann durch

$$\begin{aligned} &3 * (5 + 2x + 3x^2) + \\ &5 * (3 + 5x + 2x^2) + \\ &6 * (2 + 3x + 5x^2) \end{aligned}$$

berechnet werden, was $42 + 49x + 49x^2$ ergibt. Die Berechnung dieses Ergebnisses benötigt neun Multiplikationen und sechs Additionen. Nun wird analysiert was geschieht, wenn bei einer Multiplikation eines der beiden Polynome in Binärform auftritt (vgl. [3, Kap. 3]). Dazu werden wiederum zwei Polynome c und d, diesmal vom Grad 3, ausgewählt, wobei $c = 1 + x^2 + x^3$ und $d = 3 + 4x + 2x^2 + 5x^3$. Das Ergebnis wird wiederum durch

$$\begin{aligned} &1 * (3 + 4x + 2x^2 + 5x^3) + \\ &0 * (5 + 3x + 4x^2 + 2x^3) + \\ &1 * (2 + 5x + 3x^2 + 4x^3) + \\ &1 * (4 + 2x + 5x^2 + 3x^3) \end{aligned}$$

berechnet, was $9 + 11x + 10x^2 + 12x^3$ ergibt. Die Koeffizientenschreibweise

[1] Aus http://ntru.com/cryptolab/tutorial_hamming.htm

$[9, 11, 10, 12]$ des Ergebnisses kommt durch

$$9 = (1 * 3) + (0 * 5) + (1 * 2) + (1 * 4)$$
$$11 = (1 * 4) + (0 * 3) + (1 * 5) + (1 * 2)$$
$$10 = (1 * 2) + (0 * 4) + (1 * 3) + (1 * 5)$$
$$12 = (1 * 5) + (0 * 2) + (1 * 4) + (1 * 3)$$

zu Stande. Es fällt nun auf, dass die Berechnung keine Multiplikation benötigt. Grundsätzlich kann in so einem Fall, wenn eines der Polynome in Binärform mit d_c Koeffizienten ungleich 0 vorkommt, gesagt werden, dass für jeden Koeffizienten des Ergebnisses d_c Koeffizienten des zweiten Polynoms aufsummiert werden müssen. Im vorigen Beispiel hat das Polynom c drei Koeffizienten ungleich 0, somit müssen für jeden Koeffizienten des Ergebnisses drei Zahlen addiert werden. D. h. es werden zwei Additionen pro Koeffizient benötigt, was bei einem Ergebnis vom Grad 4, acht Additionen ergibt.

Es steht somit fest, dass die Verwendung von binären Polynomen die Berechnung einer Multiplikation erheblich erleichtert. Vom Standpunkt der Sicherheit aus gesehen (es muss bedacht werden, dass ein binäres Polynom Bestandteil des privaten Schlüssels ist) dürfen aber nicht zuwenige Koeffizienten ungleich 0 sein, da sonst ein Angreifer alle möglichen Polynome durchprobieren könnte. Aus diesem Grund werden mehrere kleine binäre Polynome miteinander kombiniert, um die Tatsache nutzen zu können, ohne Multiplikation ein Produkt zu berechnen. Außerdem werden somit auch nicht zu kleine Polynome verwendet, die die Sicherheit beeinträchtigen - in diesem Zusammenhang erklärt sich auch, warum der P1363.1-Standard bei einem Typ-2-Schlüsselpaar den privaten Schlüssel aus drei kleinen binären Polynomen zusammensetzt.

Hierzu ein Beispiel: Sei $a_1 = 1 + x + x^6$ und $a_2 = 1 + x^2 + x^4$ und das Ergebnis der Multiplikation der beiden Polynome

$$A = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^8 + x^{10}.$$

Angenommen es würde z. B. das Kryptosystem mit N=23 verwendet werden, so wären mehr als die Hälfte der Koeffizienten ungleich 0. Wird nun ein Polynom e mit A multipliziert, so werden für die Berechnung des Ergebnisses neun Additionen pro Koeffizient benötigt. Werden jedoch die Polynome a_1 und a_2 und nicht A für die Multiplikation verwendet, so wird im ersten Schritt $e * a_1$ berechnet, wobei hier jeweils drei Additionen pro Koeffizient erforderlich sind. Im zweiten Schritt wird dann $(e * a_1) * a_2$ berechnet, was noch einmal drei Multiplikationen pro Koeffizient benötigt. Insgesamt werden also sechs Additionen, anstatt neun, pro Koeffizient von e ausgeführt, was eine Einsparung von einem Drittel bedeutet (für eine detailliertere Analyse von Optimierungsmöglichkeiten siehe auch [9]).

4.2 Performance

Die Performancetests sollen einen Überblick darüber geben, wie die Veränderungen der Parameter die Laufzeit der Applikation beeinflussen. Zu diesem Zweck werden verschiedene Testprogramme erstellt und deren Ausführungszeit gemessen.

Als Werkzeug zur Performanceanalyse dient der *Intel Vtune Performance Analyzer*[2] in einer kostenlosen Evaluierungsversion.

Aufbau der Tests: Im ersten Test werden Schlüsselpaare unter Verwendung der Parametersets erstellt und die Änderung der benötigten Zeit analysiert. Der zweite Test hat als Ergebnis die Laufzeitveränderung bei der Ver- und Entschlüsselung bei gleichbleibender Nachricht. Da beide Tests die gleichen Parameter verwenden, zeigt die Tabelle 4.1, wie sich diese pro Set ändern. Die Spalte „Typ“ gibt hierbei Auskunft darüber, ob das Parameterset einen privaten Schlüssel und ein Blendpolynom in Produktform (Typ-2) verwendet. Bei den Parametern *df* und *dr* eines Typ-2-Sets, müssen daher eigentlich immer drei Werte angegeben werden. Da diese aber pro Parameterset gleich sind, wird in der Tabelle nur ein Wert angeführt.

Typ	T_1			T_2		
Set	2	4	6	3	5	7
N	251	347	397	251	347	397
q	197	269	307	293	541	659
p	2	2	2	2	2	1
db	80	112	128	80	112	128
df	48	66	74	8	11	12
dg	125	173	198	125	173	198
dr	48	66	74	8	11	12

Tabelle 4.1: Änderung der Parameter der Sets.

Für die Berechnung der Werte bei den Tests wird eine Testapplikation dreimal ausgeführt und der Mittelwert der Ergebnisse berechnet. Anschließend werden die erhaltenen Zeiten in ein Diagramm eingetragen.

1. **Erstellung von Schlüssel:** Interessant ist der Anstieg der Laufzeit im Verhältnis zur Änderung der Parameter. Durch eine Verwendung des Parametersets 6 anstatt 2 lässt die Laufzeit sich mehr als verdreifachen (s. auch Abbildung 4.1). Dieser Umstand mag daran liegen, dass die Applikation nun mit Polynomen vom Grad 396 rechnen muss (schließlich muss für die Berechnung von h das Polynom f auch invertiert

[2] Die Webseite der Software ist zu finden unter:
http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm.

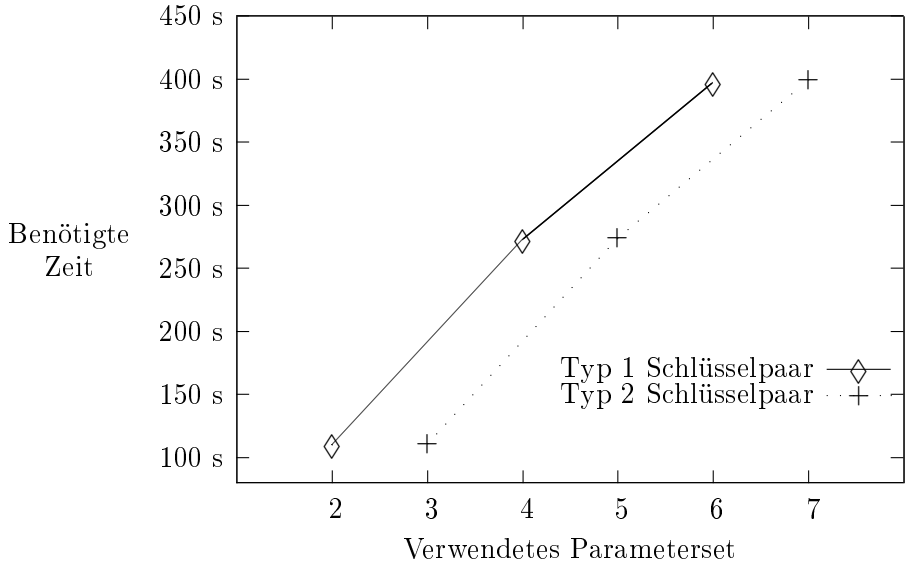

Abbildung 4.1: Laufzeit bei der Erstellung von Schlüsseln.

werden). Darüber hinaus müssen für die Erzeugung von f und g ca. 50 Zufallszahlen mehr erzeugt werden, um die Koeffizientenpositionen, die ungleich 0 sein müssen, bestimmen zu können.

Bemerkenswert ist, dass bei der Verwendung von Parameterset 7 jeder Koeffizient eines Polynoms in $\mathbb{Z}_q[x]/x^N - 1$ einen Wert x mit $0 \leq x \leq 659$ annehmen kann, die Laufzeit aber genauso steigt, als wenn sich q von 197 auf 307 ändert. D. h. bei der Erstellung eines Typ-2-Schlüsselpaares beeinflusst der Anstieg von q, im Gegensatz zur Erhöhung von N, nur unwesentlich die Laufzeit.

2. **Ver- und Entschlüsselung:** Die Unterschiede in den Laufzeiten der Applikation bei Verwendung einer Verschlüsselung des Typs-1 und einer Verschlüsselung vom Typ-2, sind nur sehr gering (s. auch Abbildung 4.2). Das liegt daran, dass bis auf eine unterschiedliche Erstellung des Blendpolynoms r und die unterschiedliche Multiplikation des öffentlichen Schlüssels mit r, sich die Operationen kaum unterscheiden. Der Anstieg der Laufzeit ist einerseits auf den größeren Grad der Polynome und die Größe der Koeffizienten, als auch auf die größere Anzahl von Koeffizienten ungleich 0 im Blendpolynom r, zurückzuführen.

   Bei der Entschlüsselung macht sich ein Performancegewinn bei der Verwendung eines Schlüssels vom Typ-2 deutlich bemerkbar. Dies lässt darauf schließen, dass die Verwendung eines privaten Schlüssels f in der Form $f = f_1 * f_2 + f_3$, wobei f_1, f_2 und f_3 kleine Polynome in Binärform sind, tatsächlich schneller vonstatten geht. Außerdem beschleunigt die

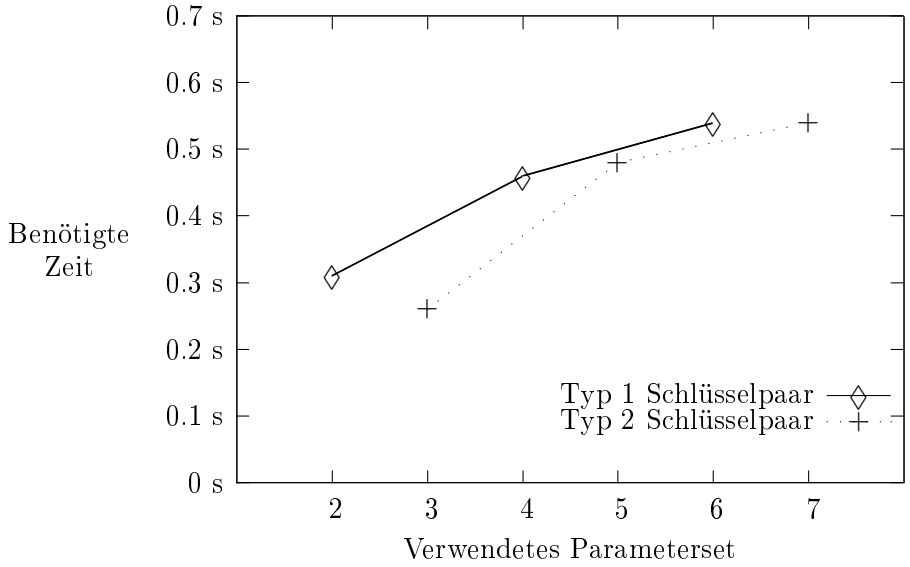

Abbildung 4.2: Laufzeit bei der Verschlüsselung.

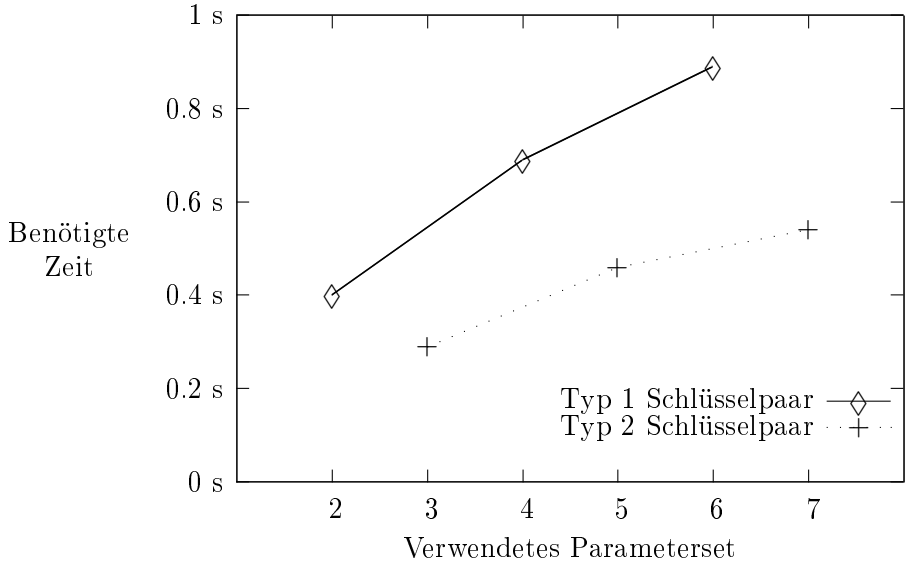

Abbildung 4.3: Laufzeit bei der Entschlüsselung.

Funktion *multiplyByProductFormPolynomZqXN1*, die explizit für eine Multiplikation eines Polynoms mit einem zweiten Polynom in Produktform ausgelegt ist, zusätzlich die Entschlüsselung der Nachricht (siehe Abbildung 4.3).

Kapitel 5

Resümee

5.1 Optimierungsmöglichkeiten

Die vorliegende Implementierung des IEEE-P1363.1-TM/D9-Standards zeigt noch einige Optimierungsmöglichkeiten auf.

- Die Darstellung einer Zahl in Binärform kann für Systeme, wo die Speichergröße begrenzt ist, noch optimiert werden. Anstatt eines Feldes des Datentyps *char*, kann z. B. auf *Bit fields* zurückgegriffen werden, um den Speicherbedarf zu minimieren.

- Für die Darstellung der Polynome muss nicht auf die Klasse *<vector>* zurückgegriffen werden. Die Programmierung unter Zuhilfenahme der Klasse erleichtert zwar die dynamische Verarbeitung der Polynomoperationen, stellt aber nicht die effizienteste Variante dar. Wieviel Performanzsteigerung durch eine alternative Darstellung, z. B. durch ein Feld von Integer-Elementen, erzielt werden kann, müsste überprüft werden.

- Eine effiziente Implementierung eines Algorithmus zum Berechnen des Kehrwertes modulo einer Zahl würde die Verwendung von NTL überflüssig machen. Dies würde auch bedeuten, dass die Applikation ohne die Einbindung einer statischen Bibliothek auskäme.

- Für die Erzeugung von Zufallszahlen wurde bei der Implementierung auf die Headerdatei *<cstdlib>* zurückgegriffen. Hier wäre anzudenken, ob es nicht effizientere und eventuell auch sicherere Methoden für die Erzeugung von Zufallszahlen gäbe.

- Zurzeit werden von der Implementierung nur jene Parametersets unterstützt, die als Hashfunktion SHA-1 verwenden. Für eine komplette Unterstützung aller Parameter müsste eine geeignete Implementierung von SHA-256 gefunden und eingebunden werden.

5.2 Abschließende Erkenntnisse

Überraschenderweise sind für die Implementierung des IEEE-P1363.1-Standards keinerlei kryptographische und mathematische Vorkenntnisse nötig. Die einzelnen Funktionen sind im Großen und Ganzen gut beschrieben, die Schwierigkeit liegt dabei die Beschreibung in die jeweilige Programmiersprache umzusetzen.

Zu beachten ist, dass während des Verfassens dieser Arbeit drei weitere Entwürfe für den IEEE-P1363.1-Standard erschienen sind. Das aktuelle Dokument, der IEEE-P1363.1-TM/D12, kann von der IEEE-Webseite nach Registrierung in deren Mailingliste bezogen werden. Soll die Implementierung auf dem aktuellsten Stand sein, so muss verglichen werden, welche Teile sich geändert haben und ob neue Funktionen hinzugefügt worden sind. Vor allem bezüglich der verwendeten Parameter wird es in Zukunft immer wieder Änderungen geben.

Die Literatur, die ergänzend zum Standard gelesen werden kann und sich detaillierter mit Gittern in der Kryptographie beschäftigt, ist spärlich gesät. Vor allem im deutschsprachigen Raum finden sich kaum Dokumente, die einen adäquaten Überblick über die Thematik geben. Die englischsprachigen Arbeiten sind sehr oft schwer verständlich und ohne entsprechendes Hintergrundwissen kaum begreifbar. Dazu kommt, dass immer wieder Information aus mehreren einzelnen Dokumenten zusammengetragen werden muss, um Zusammenhänge verstehen zu können. Es wurde daher in dieser Arbeit versucht die Methode der Verwendung von Gittern für Kryptosysteme plausibel zu beschreiben und mit einer Implementierung zu zeigen, dass die Theorie auch praktisch umgesetzt werden kann.

Literaturverzeichnis

[1] Ajtai, M.: *Generating Hard Instances of Lattice Problems*. Techn. Ber., ECCC, 1996.

[2] Babai, L.: *On Lovasz lattice reduction and the nearest lattice point problem*. Combinatorica, 6:1–13, 1986.

[3] Buchmann, J., M. Döring und R. Lindner: *Efficiency Improvement for NTRU*. In: *Sicherheit*, 2008.

[4] Cai, J.-Y.: *Some Recent Progress on the Complexity of Lattice Problems*. Techn. Ber., ECCC, 1999. http://eccc.hpi-web.de/eccc-reports/1999/TR99-006/index.html.

[5] Cohen, H.: *A course in computational algebraic number theory*. Springer Verlag, 1995.

[6] Filipovic, B.: *Implementierung der Gitterbasereduktion in Segmenten*. Diplomarbeit, Johann-Wolfgange Goethe -Universität, 2002. http://www.mi.informatik.uni-frankfurt.de/research/mastertheses/filipovic.diplom.2002.pdf.

[7] Gentry, C. und M. Szydlo: *Cryptanalysis of the Revised NTRU Signature Scheme*. Lecture Notes In Computer Science, 2332, 2002.

[8] Hoffstein, J., N. Howgrave-Graham, J. Pipher, J. Silverman und W. Whyte: *NTRUSign: Digital Signatures Using the NTRU Lattice*. CT-RSA, 2003.

[9] Hoffstein, J. und J. Silverman: *Optimizations for NTRU*. Public-Key Cryptography and Computational Number Theory, 2000.

[10] Hoffstein, J., J. Silverman und W. William: *Estimated Breaking times for NTRU lattices*. Techn. Ber., NTRU Cryptosystems, 2003.

[11] IEEE: *P1363.1: Public-Key Cryptographic Techniques based on Hard Problems over Lattices*, 2006.

[12] Joux, A. und J. Stern: *Lattice Reduction: a Toolbox for the Cryptanalyst*. Journal of Cryptology, 11:161–185, 1998.

[13] Lenstra, A. K., W. Lenstra und L. Lovasz: *Factoring Polynomials with Rational Coefficients*. Mathematische Annalen, 261:515–534, 1982.

[14] Manfred, B.: *Mathematik für Informatiker*. Hanser Fachbuch, 2005.

[15] Menezes, A. J., P. C. Van Oorschot und S. A. Vanstone: *Handbook of applied cryptography*. CRC Press, 2001.

[16] Micciancio, D.: *On the Hardness of the Shortest Vector Problem*. Doktorarbeit, Massachusetts Institute of Technology, 1998.

[17] Micciancio, D. und S. Goldwasser: *Complexity of Lattice Problems: a cryptographic perspective*, Bd. 671. Kluwer Academic Publishers, Boston, Massachusetts, 2002.

[18] Nguyen, P. Q. und O. Regev: *Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures*. Lecture Notes in Computer Science, 4004, 2008.

[19] Reichl, D.: *CSHA1 - A C++ class implementation of the SHA-1 hash algorithm*, FEB 2005. http://www.codeproject.com/KB/recipes/csha1.aspx.

[20] Schneier, B.: *Applied Cryptography*. Katherine Schowalter, 1996.

[21] Seifert, J.-P.: *Komplexit¨at von Gitterproblemen: Nicht-Approximierbarkeit und Grenzen der Nicht-Approximierbarkeit*. Doktorarbeit, Johann-Wolfgange Goethe -Universität, 2000. http://www.mi.informatik.uni-frankfurt.de/research/phdtheses.html.

[22] Trappe, W. und L. Washington: *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2005.

[23] Wolf, J.: *C++ von A bis Z*. Galilieo Press, 2006.