

Angriffe auf das McEliece Kryptosystem

Diplomarbeit von Markus Peter
Betreuer: Prof. Dr. Johannes Buchmann



Januar 2006

Fachbereich Informatik
Kryptographie und Computeralgebra
Technische Universität Darmstadt

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Januar 2006

Markus Peter

Danksagung

Danken möchte ich meinen Eltern, dafür dass sie mich während des gesamten Studiums unterstützt haben. Bei Prof. J. Buchmann und Raphael Overbeck möchte ich mich für das interessante Thema bedanken. Nicole, Michael, Christoph, Frank, Clemens und Dirk danke ich für eine schöne gemeinsame Studienzeit.

Inhaltsverzeichnis

1	Einleitung	9
2	Grundlagen	11
2.1	Codierungstheorie	11
2.2	Goppa Codes	12
3	McEliece	15
3.1	Schlüsselgenerierung	15
3.2	Verschlüsselung	16
3.3	Entschlüsselung	16
4	Angriffe	19
4.1	GISD	19
4.1.1	Funktionsweise	20
4.1.2	Theoretische Laufzeit	22
4.1.3	Experimentelle Laufzeit	24
4.2	Stern-GS	26
4.2.1	Funktionsweise	26
4.2.2	Theoretische Laufzeit	29
4.2.3	Experimentelle Laufzeit	32
4.3	Statistical Decoding	34
4.3.1	Funktionsweise	34
4.3.2	Theoretische Laufzeit	37
4.3.3	Experimentelle Laufzeit	38
5	Fazit	43
6	Code-Beschreibung	45
6.1	Design	45
6.2	Programmaufruf	46
6.2.1	attack	46

6.2.2	statistic	47
6.3	Dateiformate	49
6.3.1	Aufbau der Log Datei	49
6.3.2	Aufbau der Public Key Datei	50
A	Programm-Code	55
A.1	Attacks	55
A.2	statistic	58
A.3	GISD	66
A.3.1	Header	66
A.3.2	Code	67
A.4	SternGS	69
A.4.1	Header	69
A.4.2	Code	70
A.5	BitProxy	75
A.5.1	Header	75
A.6	BitVector	76
A.6.1	Header	76
A.6.2	Code	78
A.7	BitMatrix	83
A.7.1	Header	83
A.7.2	Code	86
A.8	BitSquareMatrix	95
A.8.1	Header	95
A.8.2	Code	95
A.9	Exceptions	96

Kapitel 1

Einleitung

Die meisten heutzutage eingesetzten Kryptosysteme basieren entweder auf dem Faktorisierungsproblem, wie etwa RSA, oder dem Problem der diskreten Logarithmen, wie z.B. ElGamal [4]. Weil Quantencomputer beide Probleme in Polynomialzeit lösen können [13], werden diese Verfahren automatisch unsicher, sobald Quantencomputer verfügbar sind. Man geht allerdings davon aus, dass \mathcal{NP} -schwere Probleme von Quantencomputern nicht in polynomieller Zeit gelöst werden können [1]. Dieser Umstand erfordert es, alternative Kryptosysteme zu erforschen, die durch den Einsatz von Quantencomputern nicht gebrochen werden können.

Das McEliece Kryptosystem ist ein solches. McEliece benutzt lineare Fehlerkorrekturcodes, für die schnelle Decodieralgorithmen vorhanden sind, die Goppa Codes. Die Sicherheit des Verfahrens basiert darauf, dass das Decodierproblem für generelle lineare Codes \mathcal{NP} -schwer ist [3].

In dieser Arbeit wird die Funktionsweise des McEliece Kryptosystems näher erläutert. Außerdem werden Angriffe auf das System behandelt und deren Laufzeit analysiert. Ziel soll es sein abzuschätzen, für welche Parameter man das McEliece Kryptosystem als sicher bezeichnen kann. Dabei soll sich nicht nur auf die theoretische Laufzeit der Angriffe bezogen werden, da diese durch die O -Notation Ungenauigkeiten enthält. Dies wird daran deutlich, dass sowohl $5n^3$ als auch $1000n^3$ beide in $O(n^3)$ sind. Der Vorfaktor der in der O -Notation wegfällt, hat folglich einiges zur tatsächlichen Laufzeit beizutragen.

In Kapitel 2 werde ich zunächst das Grundlegende zur Codierungstheorie darlegen. Kapitel 3 enthält eine Beschreibung des McEliece Kryptosystems und in Kapitel 4 werden die Angriffe darauf beschrieben und durchgeführt. Kapitel 5 beinhaltet mein Fazit und Kapitel 6 beschreibt die benutzten Programme.

Kapitel 2

Grundlagen

2.1 Codierungstheorie

Die Codierungstheorie ist die mathematische Theorie über die Erkennung und Korrektur von Fehlern in digitalen Daten. Ihr Einsatzzweck ist folglich der Schutz digitaler Daten vor Fehlern bei deren Speicherung oder Übertragung. Um dies zu erreichen, werden die ursprünglichen Daten, auch Informationssequenz genannt, mittels der Codierungsfunktion in ein Codewort transformiert. Dieses Codewort enthält Redundanzen, die es ermöglichen sollen eine bestimmte Anzahl von Fehlern zu erkennen oder gar zu beheben. Dadurch ist das Datenvolumen des Codeworts größer als das der Informationssequenz.

Es lassen sich allerdings nicht beliebig viele Fehler korrigieren. Ein wichtiges Maß, das zur Beschreibung dieser Code Eigenschaft herangezogen wird, ist die Hamming Distanz.

Definition 1 Die Hamming Distanz zweier Codevektoren $x = (x_1, x_2, \dots, x_n)$ und $y = (y_1, y_2, \dots, y_n)$ entspricht:

$$d(x, y) = |\{x_i | x_i \neq y_i\}|.$$

Die Hamming Distanz beschreibt also die Anzahl der unterschiedlichen Einträge beider Vektoren.

Definition 2 Das Hamming Gewicht gibt den Abstand eines Vektors zum Nullvektor an:

$$|x| = d(x, 0) = |\{x_i | x_i \neq 0\}|$$

Die minimale Distanz d_{min} zwischen zwei Codewörtern bestimmt nun wie gut der Code Fehler erkennen oder korrigieren kann. Je größer die minimale Distanz ist, umso mehr Fehler kann der Code beheben. Die Zahl der maximal erkennbaren Fehler f_e ergibt sich aus $f_e = d_{min} - 1$. Maximal können $f_k = \lfloor (d_{min} - 1)/2 \rfloor$ Fehler korrigiert werden.

Es gibt verschiedene Arten von Codes, die nach ihrer Codierungsfunktion unterschieden werden. Als einige Beispiele wären die Gruppencodes, die linearen Codes und zyklischen Codes zu nennen. Im weiteren Verlauf werden wir uns nur mit einem bestimmten linearen Code beschäftigen, dem Goppa Code.

2.2 Goppa Codes

In diesem Abschnitt wird der Goppa Code definiert. Ebenso wird erklärt wie man eine Generatormatrix und eine Parity Check Matrix für Goppa Codes konstruiert. Goppa Codes wurden 1970 von V.D. Goppa definiert [10].

Definition 3 Ein linearer Code \mathcal{C} der Länge n und Dimension k über $GF(q)$ ist ein k -dimensionaler Untervektorraum von $GF(q)^n$.

Der binäre Goppa Code operiert auf einem binären Alphabet im endlichen Körper $GF(2)$. Da es sich dabei um einen linearen Code handelt, kann man die Codierungsfunktion als eine Matrix G darstellen. G bildet einen Vektor $m \in GF(2)^k$ auf einen Vektor $c \in GF(2)^n$ ab. Um eine k -bit Informationssequenz m in ihr korrespondierendes n -bit Codewort c zu transformieren berechnet man: $m * G = c$.

In der Kryptographie sind nur die irreduziblen Goppa Codes interessant. Um einen irreduziblen Goppa Code zu erzeugen, wählt man zuerst ein normiertes, irreduzibles Polynom

$$g(x) = \sum_{i=0}^t g_i * x^i \text{ über } GF(2^m) \text{ vom Grad } t.$$

Der Goppa Code kann dann maximal t Fehler korrigieren und hat die Codelänge $n = 2^m$. Die Dimension k des Codes liegt bei $k \geq n - t * m$.

Als nächstes wählt man n paarweise verschiedene Elemente

$$L = \{\gamma_1, \gamma_2, \dots, \gamma_n\}$$

aus $GF(2^m)$. Da $g(x)$ irreduzibel ist, gilt für alle Elemente $\gamma \in GF(2^m)$, dass $g(\gamma) \neq 0$ ist. Die Menge L benötigt man, um das Syndrom eines Vektors zu berechnen.

Definition 4 Das Syndrom eines Vektors $c \in GF(2)^n$ ist definiert durch:

$$S_c(x) = - \sum_{i=0}^{n-1} \frac{c_i}{g(\gamma_i)} * \frac{g(x) - g(\gamma_i)}{x - \gamma_i}$$

Sei $y = g + e$ ein durch den Fehlervektor e korrumpierter Codevektor g . Das Produkt $s = y * H^t$ wird als Syndromvektor von y bezeichnet. $s = y * H^t = (g + e) * H^t = e * H^t$. Das Syndrom hängt also nur vom Fehlervektor e ab und ist unabhängig vom Codevektor g .

Der binäre Goppa Code besteht nun aus allen Vektoren $c \in GF(2)^n$ für die gilt:

$$S_c(x) = 0 \text{ oder äquivalent } S_c(x) \equiv \sum_{i=0}^{n-1} \frac{c_i}{x - \gamma_i} \equiv 0 \pmod{g(x)}.$$

Mit der Parity Check Matrix H kann man testen, ob es sich bei einem Vektor $c \in GF(2)^n$ um ein gültiges Codewort handelt. Denn für alle Codewörter g des Goppa Codes gilt $g * H^t = 0$. Die Parity Check Matrix H übernimmt folglich den Test, ob das Syndrom $S_c(x) = 0$ ist. Daraus ergibt sich der Aufbau von H wie folgt:

$$H = \begin{pmatrix} g_t g(\gamma_0)^{-1} & \cdots & g_t g(\gamma_{n-1})^{-1} \\ (g_{t-1} + g_t \gamma_0) g(\gamma_0)^{-1} & \cdots & (g_{t-1} + g_t \gamma_{n-1}) g(\gamma_{n-1})^{-1} \\ \vdots & \ddots & \vdots \\ (\sum_{j=1}^t g_j \gamma_0^{j-1}) g(\gamma_0)^{-1} & \cdots & (\sum_{j=1}^t g_j \gamma_{n-1}^{j-1}) g(\gamma_{n-1})^{-1} \end{pmatrix} = XYZ$$

wobei

$$X = \begin{pmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{pmatrix}, Y = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \gamma_0 & \gamma_1 & \cdots & \gamma_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_0^{t-1} & \gamma_1^{t-1} & \cdots & \gamma_{n-1}^{t-1} \end{pmatrix},$$

$$Z = \begin{pmatrix} \frac{1}{g(\gamma_0)} & & & \\ & \frac{1}{g(\gamma_1)} & & \\ & & \ddots & \\ & & & \frac{1}{g(\gamma_{n-1})} \end{pmatrix}.$$

Die Zeilen von H spannen einen Untervektorraum von $GF(2)^n$ auf. Da für alle Codevektoren gilt, dass $g * H^t = 0$ ist, bildet der Goppa Code einen zu H dualen Vektorraum. Wir erhalten also die Generatormatrix G des Goppa Codes, indem wir die Basisvektoren dieses dualen Vektorraums berechnen. Sie bilden die Zeilen der Generatormatrix G .

Ist ein Codevektor g durch einen Fehler e zu einem korrumpierten Codevektor $y = g + e$ geworden, dann kann man aus y effizient g ermitteln, wenn man das Generatorpolynom $g(x)$ kennt. Dazu existieren mehrere Algorithmen. Ein solcher ist der Peterson Algorithmus, der auch als Matrix Methode bekannt ist. Seine Laufzeit beträgt $O(n^3)$. Seine Funktionsweise wird in [8] näher beschrieben. Ein weiterer Algorithmus zum schnellen Decodieren ist der Berlekamp-Massey Algorithmus, der auf der Euklidischen Division basiert [8].

Kapitel 3

McEliece

In diesem Kapitel wird erläutert, wie das McEliece Kryptosystem funktioniert.

Das McEliece Kryptosystem basiert auf einem \mathcal{NP} -schweren Problem aus der Informations- und Codierungstheorie, dem Decodieren von generellen linearen Codes.

Bei der Nachrichtenübermittlung werden die Informationen für gewöhnlich codiert, damit der Empfänger in der Lage ist, Übertragungsfehler herauszufiltern. Die ursprüngliche Nachricht wird dabei durch Redundanzinformationen vergrößert. Durch die Redundanzen können eine bestimmte Anzahl von Übertragungsfehlern ausgeglichen werden.

Diese Fehlerkorrekturmechanismen macht sich McEliece in seinem Kryptosystem zu nutze. Im McEliece Kryptosystem fügt der Versender der Nachricht selbst Fehler zur codierten Nachricht c hinzu. Wie wir später sehen werden, ist es für einen Angreifer schwierig, diese zufällig hinzugefügten Fehler aus der Nachricht zu entfernen.

3.1 Schlüsselgenerierung

Um sich ein McEliece Private Key / Public Key Schlüsselpaar zu erzeugen, wählt man ein zu den Rahmenparametern n und t passendes irreduzibles Generatorpolynom $g(x)$ vom Grad t über $GF(2^m)$. Die Chance, dass ein zufällig gewähltes Polynom $g(x)$ irreduzibel ist, beträgt etwa $1/t$. Um dies zu überprüfen existieren schnelle Test-Algorithmen [2].

Anschließend berechnet man die $k \times n$ Generatormatrix G' , indem man k linear unabhängige Vektoren g_i aus $GF(2)^n$ wählt. Um die Kryptoanalyse zu erschweren, wird die Generatormatrix dann verschleiert. Dazu wählt man eine zufällige, invertierbare, dichte $k \times k$ Matrix S und eine zufällige $n \times n$

Permutationsmatrix P . Die verschleierte Generatormatrix G ergibt sich dann aus $G = S * G' * P$. G generiert einen permutationsäquivalenten linearen Code zu G' mit denselben Eigenschaften, insbesondere ist auch der minimale Abstand identisch. G wird dann als der öffentliche Schlüssel bekannt gegeben, während S , P , G' und $g(x)$ geheim bleiben und somit den Private Key bilden.

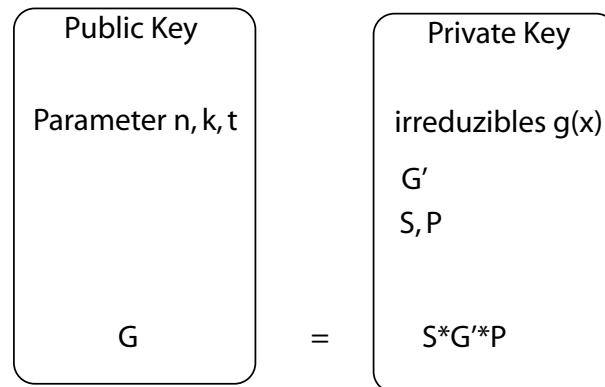


Abbildung 3.1: Public Key / Private Key Übersicht

3.2 Verschlüsselung

Um eine k -bit Nachricht m zu verschlüsseln, berechnet man zunächst das korrespondierende Codewort $g = m * G$. Anschließend addiert man einen zufällig gewählten n -bit Fehlervektor e mit $|e| \leq t$ hinzu: $c = g \oplus e = m * G \oplus e$. c ist dann die verschlüsselte Version von m . Die Fehlerstellen in e sollten gleichverteilt gewählt werden, da man sonst Nullstellen in e raten könnte. Es ist für die Sicherheit des Verfahrens wichtig, dass der gewählte Fehlervektor geheim bleibt und für jede Nachricht ein neuer Fehlervektor bestimmt wird.

3.3 Entschlüsselung

Um eine verschlüsselte Nachricht c zu entschlüsseln, berechnet man zunächst $c' = c * P^{-1}$. Anschließend ermittelt man mit einem schnellen Decodieralgo-

rithmus, z.B. dem Peterson Algorithmus, $m * S = m'$. Daraus erhält man dann $m = m' * S^{-1}$. Wie man sieht, ist der Fehlervektor e zum Decodieren tatsächlich nicht nötig. Allein durch die Kenntnis des Goppa Polynoms $g(x)$ ist es möglich die Fehler zu korrigieren. Dies geschieht mit dem Peterson Algorithmus, der eine Laufzeit von $O(n^3)$ hat [8]. Ohne Kenntnis des passenden Goppa Polynoms $g(x)$, kann man die Fehler nicht effizient entfernen.

Kapitel 4

Angriffe

Für einen Angreifer gibt es zwei potentielle Herangehensweisen, um das McEliece Kryptosystem zu brechen. Einerseits könnte er versuchen $g(x)$ aus G abzuleiten, um dann den Peterson Algorithmus zum schnellen Decodieren benutzen zu können. Die zweite Möglichkeit ist, m aus c zu konstruieren ohne G' bzw. $g(x)$ zu kennen.

Die erste Attacke wird durch geeignete Wahl von n und t vereitelt. Für $n = 1024 = 2^{10}$ und $t = 50$ existieren allein etwa 2^{149} verschiedene lineare Codes. Hinzu kommen noch unzählige Variationsmöglichkeiten für S und P . Ein Brute-Force Angriff auf die geheime Matrix G würde also viel zu lange dauern. Und eine effizientere Methode, $g(x)$ aus den öffentlich bekannten Informationen zu ermitteln, ist bisher nicht bekannt.

Der zweite Weg sieht interessanter aus. Allerdings ist das zu lösende Basisproblem das allgemeine Decodierproblem für lineare Codes, welches \mathcal{NP} -schwer ist [3]. Trotzdem hat dieser Angriff bessere Erfolgsaussichten als die erste Methode.

Im folgenden werden drei Varianten der zweiten Methode vorgestellt. Zunächst wird die Funktionsweise einer Variante erläutert und dann auf ihre theoretische Laufzeit eingegangen. Anschließend werden die auf einem Athlon XP 2100+ mit 512 MB RAM gemessenen Laufzeiten präsentiert und analysiert.

4.1 GISD

Die erste Variante ist der General Information Set Decoding (GISD) Angriff. Bereits McEliece hat in seinem ursprünglichen Paper [10] diese Idee angesprochen. Lee und Brickell haben dann noch eine Verbesserung vorgenommen [9].

4.1.1 Funktionsweise

Die Idee des Angriffs besteht darin, eine Menge \mathcal{J} von k Spalten aus der $k \times n$ Generatormatrix G so zu wählen, dass diese k Spalten eine invertierbare $k \times k$ Matrix $G_{\mathcal{J}}$ bilden. Ebenso bildet man aus den gleichen k Spalten des Codevektors c den k -bit Vektor $c_{\mathcal{J}}$. Dadurch wählt man implizit auch den k -bit Vektor $e_{\mathcal{J}}$ mit.

Sind die gewählten Spalten fehlerfrei, d.h. $e_{\mathcal{J}} = 0$, wird $c_{\mathcal{J}} = m * G_{\mathcal{J}} \oplus e_{\mathcal{J}}$ zu $c_{\mathcal{J}} = m * G_{\mathcal{J}}$. Daraus ergibt sich $m = c_{\mathcal{J}} * G_{\mathcal{J}}^{-1}$ unter der Bedingung, dass $G_{\mathcal{J}}$ invertierbar ist.

Dies ist allerdings nur korrekt, falls der Fehler $e_{\mathcal{J}} = 0$ ist. Die Lee und Brickell Erweiterung [9] addiert alle möglichen Fehlervektoren $e_{j_{\mathcal{J}}}$ mit Gewicht kleiner als j zu $c_{\mathcal{J}} * G_{\mathcal{J}}^{-1}$ dazu, um Fehlervektoren $e_{\mathcal{J}}$ zu eliminieren, deren Gewicht maximal j ist. Falls nämlich der Fehlervektor e_j dem $e_{\mathcal{J}}$ entspricht, wird dieser dadurch eliminiert und man kann nun $m = c_{\mathcal{J}} * G_{\mathcal{J}}^{-1}$ berechnen. Die Anzahl fehlerbehafteter Spalten wird durch das Informationsfenster bestimmt.

Definition 5 Sei $N = \{0, 1, \dots, n\}$ die Menge der Spaltenindizes von G . \mathcal{J} ist ein Informationsfenster von G , falls $G = (V, W)_{\mathcal{J}}$ mit $V = (G_i)_{i \in \mathcal{J}}$ und $W = (G_j)_{j \in N \setminus \mathcal{J}}$. G_i beschreibt die i -te Spalte in G . Die Komplementärmenge $\mathcal{R} = N \setminus \mathcal{J}$ nennt man Redundanzfenster.

Man kann nicht effizient im voraus ermitteln, ob die gewählte $k \times k$ Matrix $G_{\mathcal{J}}$ invertierbar ist. Zum Testen auf Invertierbarkeit benötigt man einen ähnlichen Aufwand, wie zum Invertieren selbst. Deshalb spart man sich den Test und versucht die Matrix direkt zu invertieren. Zum Invertieren wird der Gauss Algorithmus verwendet, der einen Aufwand von $O(n^3)$ hat. Tritt in dessen Verlauf eine Null-Zeile oder -Spalte auf, ist die Matrix nicht invertierbar. Der Algorithmus wird dann abgebrochen und das Verfahren beginnt mit einer neuen Auswahl von Spalten von Vorne. Das bedeutet, dass man mindestens den Aufwand $O(n^3)$ pro Spaltenauswahl \mathcal{J} aufwenden muss unabhängig davon, wie viele Fehlerspalten die Menge \mathcal{J} letztendlich enthält. Ist die Matrix invertierbar, nennt man die Menge der k Spalten \mathcal{J} Informationsfenster von G .

Wenn man ein Informationsfenster gefunden hat, kann man nun testen, ob es maximal j Fehler enthält. Die Rechenzeit für den Test auf Fehlerfreiheit der gewählten Spalten ist sehr gering. Man kann diesen Test allerdings erst durchführen, wenn die Matrix invertiert ist.

Durch die Verbesserung von Lee und Brickell kann man auch eine geringe Anzahl an Fehlerspalten tolerieren. Dadurch kann man ein Informationsfens-

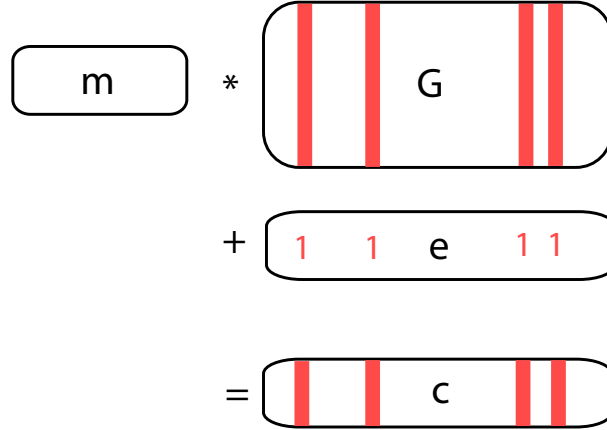


Abbildung 4.1: GISD Fehlerspalten

ter quasi mehrfach verwenden. Lee und Brickell schlagen als maximale Anzahl fehlerbehafteter Spalten $j = 2$ vor. Diese Optimierung erhöht den Aufwand pro Informationsfenster nur gering. Dafür steigt die Wahrscheinlichkeit erheblich, c decodieren zu können. Das bedeutet, dass man im Schnitt weniger Informationsfenster und somit Matrixinvertierungen berechnen muss. Der komplette Ablauf ist in Algorithmus 1 beschrieben.

Der Algorithmus terminiert erfolgreich, wenn $|z \oplus e_j * Q_2| \leq t$ ist. Warum e_j dann der gesuchte Fehlervektor ist, werden wir uns jetzt etwas näher anschauen. Setzt man für $z = c \oplus c_{\mathcal{J}} * Q_2$ und für $Q_2 = G_{\mathcal{J}}^{-1} * G$ ein, erhält man:

$$|(c \oplus c_{\mathcal{J}} * Q_2) \oplus e_j * G_{\mathcal{J}}^{-1} * G| \leq t$$

bzw.

$$|(c \oplus c_{\mathcal{J}} * G_{\mathcal{J}}^{-1} * G) \oplus e_j * G_{\mathcal{J}}^{-1} * G| \leq t.$$

Da $c_{\mathcal{J}} = m * G_{\mathcal{J}} \oplus e_{\mathcal{J}}$ ist, erhält man

$$|c \oplus (m * G_{\mathcal{J}} \oplus e_{\mathcal{J}}) * G_{\mathcal{J}}^{-1} * G \oplus e_j * G_{\mathcal{J}}^{-1} * G| \leq t.$$

Wenn man $m * G_{\mathcal{J}}$ aus der Klammer zieht und anschließend $G_{\mathcal{J}}^{-1} * G$ ausklammert, wird die Ungleichung zu:

$$|c \oplus m * G \oplus (e_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t.$$

Algorithmus 1 GISD

Input:

- $k \times n$ Generatormatrix G
- die maximale Anzahl an Fehlern t
- Schlüsseltext $c = m * G \oplus e$, wobei m der Klartext und e ein Fehlervektor mit $|e| \leq t$ ist
- positiver Integer $j \leq t$

Output:

- Klartext m

while true do

wähle zufälliges $\mathcal{J} \subseteq \{0, 1, \dots, n-1\}$, $|\mathcal{J}| = k$

$$Q_1 = G_{\mathcal{J}}^{-1}$$

$$Q_2 = Q_1 * G$$

$$z = c \oplus c_{\mathcal{J}} * Q_2$$

for $i = 0$ to j **do**

for alle e_j mit $|e_j| < i$ **do**

if $|z \oplus e_j * Q_2| \leq t$ **then**

return $(c_{\mathcal{J}} \oplus e_j) * Q_1$

end if

end for

end for

end while

Da $c = m * G \oplus e$ ist und $m * G \oplus m * G = 0$ folgt:

$$|e \oplus (e_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t.$$

Und das ergibt sich zu

$$|e| \leq t,$$

falls $e_j = e_{\mathcal{J}}$.

4.1.2 Theoretische Laufzeit

Die durchschnittliche Laufzeit des Algorithmus hängt stark davon ab, wie wahrscheinlich es ist, eine Menge \mathcal{J} zu wählen, die maximal j Fehler enthält. Es existieren insgesamt

$$\binom{n}{k}$$

Algorithmus 2 GISD Abbruch-Test Umformung

$$\begin{aligned}
 &|e| \leq t, \text{ wenn } e_{\mathcal{J}} = e_j \text{ ist, dann gilt} \\
 &|e \oplus (e_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t \\
 &|m * G \oplus e \oplus m * G \oplus (e_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t \text{ gdw} \\
 &|c \oplus m * G \oplus (e_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t \text{ gdw} \\
 &|c \oplus (m * G_{\mathcal{J}} \oplus e_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t \text{ gdw} \\
 &|c \oplus (c_{\mathcal{J}} \oplus e_j) * G_{\mathcal{J}}^{-1} * G| \leq t \text{ gdw} \\
 &|c \oplus c_{\mathcal{J}} * G_{\mathcal{J}}^{-1} * G \oplus e_j * G_{\mathcal{J}}^{-1} * G| \leq t \text{ gdw} \\
 &|c \oplus c_{\mathcal{J}} * Q_2 \oplus e_j * Q_2| \leq t \text{ gdw} \\
 &|z \oplus e_j * Q_2| \leq t
 \end{aligned}$$

Mengen \mathcal{J} mit $|\mathcal{J}| = k$. Die Zahl der Mengen mit exakt i Fehlern betragt:

$$\binom{t}{i} * \binom{n-t}{k-i}.$$

Daraus ergibt sich die Wahrscheinlichkeit eine Menge \mathcal{J} zu wahlen, die maximal j Fehler enthalt zu:

$$p_j = \sum_{i=0}^j \frac{\binom{t}{i} * \binom{n-t}{k-i}}{\binom{n}{k}}.$$

Das bedeutet, dass man im Schnitt $1/p_j$ Matrizen $G_{\mathcal{J}}$ invertieren muss, um erfolgreich zu sein. Zum Invertieren einer $k \times k$ Matrix benotigt man k^3 Operationen. Der Aufwand um festzustellen, ob der resultierende k -bit Co-devektor $c_{\mathcal{J}}$ maximal j Fehler enthalt, setzt sich folgendermaen zusammen. Die Zahl der Fehlervektoren e mit einem Gewicht kleiner als j betragt

$$N_j = \sum_{i=0}^j \binom{k}{i}.$$

Fur jeden Fehlervektor werden k Bitoperationen durchgefuhrt. Also mussen insgesamt $k * N_j$ Operationen pro Durchgang durchgefuhrt werden. Somit erhalt man als durchschnittliche Laufzeit

$$L_j = \alpha * 1/p_j * (k^3 + N_j * k)$$

mit einer kleinen Konstante α .

4.1.3 Experimentelle Laufzeit

Für die experimentellen Tests wurden für jedes m fünf verschiedene Schlüsselpaare generiert. Deren Parameter entsprechen den optimalen Parametern, die in Tabelle 4.1 aufgeführt sind. Diese Parameter sind dahingehend optimal, als dass für sie die durchschnittliche theoretische Laufzeit des GISD am größten ist. P bezeichnet die Wahrscheinlichkeit, dass man ein Informationsfenster mit weniger als $j = 2$ Fehlern wählt. Für $m = 4$ bzw. $m = 5$ wäre der beste Wert für $t = 1$ gewesen, allerdings ließ sich zu diesem Parametersatz keine Generatormatrix erzeugen.

m	t	k	P
4	2	8	100%
5	3	17	100%
6	5	34	44%
7	8	72	7,1%
8	13	152	0,1%

Tabelle 4.1: optimale GISD Parameter

Die Anzahl der Tests ist die Gesamtzahl der durchgeführten Tests. Mit jedem Schlüssel wurden gleich viele Versuche durchgeführt. Für jeden Test wurde eine zufällig gewählte Klartextnachricht m verschlüsselt und der zugehörige Schlüsseltext c mit dem GISD Algorithmus wieder decodiert. Der GISD Algorithmus wurde immer mit dem Parameter $j = 2$ durchgeführt. Eine Übersicht über die erzielten Ergebnisse liefert Tabelle 4.2. Die Messwerte wurden für $m \leq 6$ auf Millisekunden genau gerundet. Für größere m sind die Messwerte auf Sekunden gerundet.

Man sieht, dass für $m = 4$ im Schnitt nur knapp 2 Informationsfenster berechnet werden müssen. Hierbei ist die Chance 100%, dass ein Informationsfenster über maximal j Fehler verfügt. Das bedeutet, dass circa die Hälfte der gewählten Informationsfenster nicht invertierbar waren.

Auch für $m = 5$ ist die Erfolgswahrscheinlichkeit 100%, wenn das Informationsfenster invertierbar ist. Dies ist im Schnitt bei jedem dritten auch der Fall. Im weiteren Verlauf fällt die Wahrscheinlichkeit drastisch, ein Informationsfenster mit maximal j Fehlern zu erhalten.

Da angenommen werden kann, dass die Laufzeit exponentiell verläuft, ist die zugehörige Funktion von der Form $f(x) = e^{g(x)}$. $g(x)$ ist allerdings unbekannt. Als Arbeitshypothese wird angenommen, dass $g(x)$ entweder eine lineare oder eine quadratische Funktion ist. Um $g(x)$ näher zu untersuchen, betrachten wir nun den natürlichen Logarithmus der Laufzeiten.

m	4	5	6	7
Anzahl der Tests	5000	5000	5000	500
Laufzeit (Durchschnitt)	1,2ms	18,2ms	0,698s	64,4s
Informationsfenster	1,9	3,1	7,7	44,7
Median	0s	10ms	0,45s	44,5s
5%-Quantil	0s	0s	30ms	4,3s
10%-Quantil	0s	0s	40ms	7,7s
Theoretische Laufzeit	$4,7 * 10^3$	$5,0 * 10^4$	$1,6 * 10^6$	$4,2 * 10^8$

Tabelle 4.2: GISD Ergebnisse

Zunächst werden Funktionen für das 5%-Quantil $q_5(x)$ und das 10%-Quantil $q_{10}(x)$ ermittelt. Sie dienen als Abschätzung der Laufzeit nach unten. Wenn man diese Zeit aufwendet, kann man 5% bzw 10% der verschlüsselten Nachrichten in dieser Zeit decodieren. Für $m = 4$ und $m = 5$ konnten allerdings keine Werte in die Berechnung mit eingehen, da der natürliche Logarithmus von Null nicht definiert ist. Die Laufzeiten sind natürlich nicht exakt Null, aber die Messgenauigkeit liegt bei 10 ms und Werte darunter werden mit einer Laufzeit von 0 ms registriert. Dadurch können diese Messwerte nicht weiter verarbeitet werden.

Für $m = 8$ ist die durchschnittliche Laufzeit zu hoch, um ausreichend Messwerte zu erzielen. Deshalb wurden 100 Tests mit $m = 8$ nach jeweils 20 Minuten abgebrochen, falls sie noch nicht erfolgreich decodiert hatten. Dadurch ließ sich zumindest ein Wert für das 5%-Quantil ermitteln. Es liegt bei 16 Minuten. Somit stehen zur Extrapolation des 5%-Quantils 3 Messpunkte zur Verfügung, für das 10%-Quantil nur deren zwei. Als Funktionen für die Quantile erhält man dann $q_{10}(x) = 5,06581x - 33,5481$ und $q_5(x) = 4,97097x - 33,3324$.

In Tabelle 4.3 sieht man, wie sich der Verlauf der Quantile für größere Schlüssel entwickelt.

Damit Ausreißern nach unten oder oben bei den Messwerten keine so große Bedeutung beigemessen wird, bestimmt man den Erwartungswert nicht als arithmetisches Mittel, sondern über den Median. Die lineare Schätzung für den Median ergibt $m_l(x) = 4,20075x - 25,7403$, die quadratische $m_q(x) = 0,39405x^2 - 0,52785x - 11,8172$.

Extrapoliert man diese Kurven über $m = 8$ hinaus, sieht man (in Tabelle 4.3), dass der lineare Median ab $m = 10$ kürzere Rechenzeiten liefert als die beiden Quantile. Hier scheint der quadratisch extrapolierte Median die bessere Wahl zu sein.

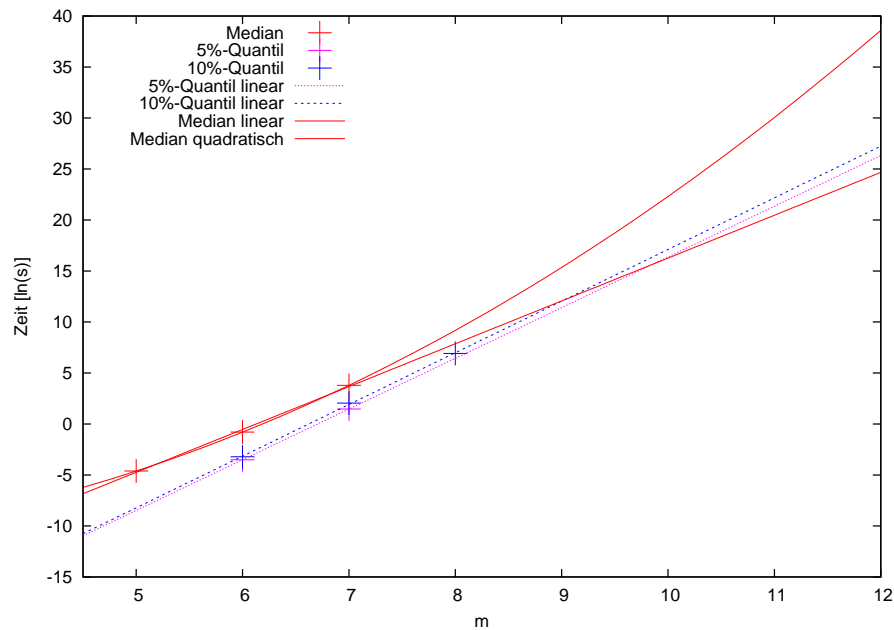


Abbildung 4.2: GISD Laufzeiten

4.2 Stern-GS

Der Stern-GS Algorithmus [6] schlägt einen anderen Weg als der GISD ein, um aus dem Kryptotext $c = m * G \oplus e$ den Klartext m zu erhalten. Er generiert einen neuen Coderaum, in dem das kürzeste Codewort e ist. Ist e gefunden, kann man anschließend leicht m aus dem fehlerfreien Vektor $c' = c \oplus e$ erhalten.

4.2.1 Funktionsweise

Zunächst wird für den neuen Coderaum die Generatormatrix \hat{G} aus den öffentlich bekannten G und c erzeugt, indem c als neue Zeile an G angefügt

m	$e^{q_{10}(m)}$	$e^{q_5(m)}$	$e^{m_l(m)}$	$e^{m_q(m)}$
8	18 min	10 min	43 min	162 min
9	47 h	25 h	48 h	54 d
10	312 d	150 d	134 d	157 y
11	137 y	60 y	25 y	363.543 y
12	21.780 y	8.660 y	1.662 y	$1,9 * 10^9$ y
13	3.452.362 y	1.248.415 y	110.924 y	$2 * 10^{13}$ y

Tabelle 4.3: GISD Laufzeit Extrapolation

wird:

$$\hat{G} = \left(\begin{array}{c} G \\ \hline m * G \oplus e \end{array} \right)$$

Nun wird das kürzeste Codewort des von \hat{G} erzeugten linearen Codes bestimmt. Dabei handelt es sich um e . Denn alle von G generierten Codewörter g haben einen minimalen Abstand von $2t + 1$ und somit ein Gewicht $|g| \geq 2t + 1$. Das Gewicht von e hingegen beträgt $|e| \leq t$. e ist Teil des von \hat{G} erzeugten Coderaums, da für $m' = (m, 1)$ sich $m' * \hat{G}$ zu $m * G \oplus 1 * (m * G \oplus e)$ ergibt. Somit ist $m' * \hat{G} = e$.

Für diese neue Generatormatrix \hat{G} wird dann ein initiales Informationsfenster \mathcal{I} ermittelt. Dazu benutzt man, wie aus dem GISD Algorithmus bekannt, das Gauss Verfahren. Als Ergebnis erhält man eine systematische Generatormatrix $G = (id_k | Z)$ in Abhängigkeit von \mathcal{I} .

Definition 6 Eine $k \times n$ Generatormatrix G heißt systematisch, falls ihre ersten k Spalten die $k \times k$ Einheitsmatrix id_k bilden.

Dieses Informationsfenster \mathcal{I} wird dann zufällig in zwei Teilmengen \mathcal{I}_1 und \mathcal{I}_2 aufgeteilt, so dass $|\mathcal{I}_1| = \lfloor k/2 \rfloor$ und $|\mathcal{I}_2| = \lceil k/2 \rceil$. Dadurch erhält man eine Partitionierung von Z in $Z_{\mathcal{I}_1}$ und $Z_{\mathcal{I}_2}$. Durch die Partitionierung muss man später nicht so viele verschiedene Linearkombinationen berechnen.

Als nächstes wird zufällig eine l -elementige Menge \mathcal{L} bestimmt. In ihr sind nur Spaltenindizes der Teilmatrix Z enthalten, d.h. $\mathcal{L} \cap \mathcal{I} = \emptyset$.

Nun berechnet man alle Linearkombinationen λ_1 von p Zeilen aus $Z_{\mathcal{I}_1}$ und alle Linearkombinationen λ_2 von p Zeilen aus $Z_{\mathcal{I}_2}$.

Stimmen für zwei Linearkombinationen λ_1 und λ_2 alle Spalten aus \mathcal{L} überein, überprüft man ob $|\lambda_1 \oplus \lambda_2| \leq t - 2p$. Wenn dies der Fall ist, ergibt

sich e aus der Linearkombination der $2p$ Zeilen aus \hat{G} die zur Berechnung von λ_1 und λ_2 verwendet wurden.

Um alle Linearkombinationen aus den beiden Mengen miteinander vergleichen zu können, berechnet man zuerst alle Linearkombinationen aus $Z_{\mathcal{I}_1}$ und speichert diese. Zusätzlich zu jeder Linearkombination muss man die Indizes der verwendeten p Zeilen speichern. Auch der Wert der in \mathcal{L} enthaltenen Spalten wird extra gespeichert, um die Suche zu beschleunigen.

Algorithmus 3 Stern-GS

Input:

- $k \times n$ Generatormatrix G
- Anzahl p der Zeilen pro Linearkombination
- l , die Anzahl der Referenzspalten
- die maximale Anzahl an Fehlern t

Output:

- Codewort c , mit dem Gewicht $|c| \leq t$

wähle zufällig eine Informationsmenge \mathcal{I} und wende den Gauss Algorithmus an, um eine systematische Generatormatrix $G = (Id_k | Z)$ zu erhalten.

while true do

verteile alle Elemente aus \mathcal{I} so in zwei Mengen \mathcal{I}_1 und \mathcal{I}_2 auf, dass $|\mathcal{I}_1| = \lfloor k/2 \rfloor$ und $|\mathcal{I}_2| = \lceil k/2 \rceil$. Dadurch erhält man

$$G = \left(\mathcal{J} \mid \begin{array}{c} Z_1 \\ Z_2 \end{array} \right)$$

wähle ein l -elementige Untermenge \mathcal{L} aus $N \setminus \mathcal{I}$

für alle Linearkombinationen λ_1 von p Zeilen aus Z_1 speichere das Tupel $(\lambda_{1_{\mathcal{L}}}, \lambda_1, \text{Indizes der } p \text{ Zeilen})$

für alle Linearkombinationen λ_2 von p Zeilen aus Z_2 berechne $\lambda_{2_{\mathcal{L}}}$

if $(\lambda_{2_{\mathcal{L}}}, \lambda_1, p \text{ Zeilenindizes})$ gespeichert wurde **then**

if $|(\lambda_1 + \lambda_2)_{N \setminus (\mathcal{I} \cup \mathcal{L})}| = t - 2p$ **then**

konstruiere c aus den $2p$ zur λ -Berechnung verwendeten Zeilen

return(c)

end if

end if

bilde ein neues \mathcal{I}

end while

Die Größe von \mathcal{L} hat somit einen gewissen Einfluss darauf, wie viel Speicher man benötigt und andererseits wie wahrscheinlich es ist e zu finden.

Dadurch, dass λ_1 und λ_2 bei diesen l Spalten übereinstimmen, ergibt die Summe λ aus $\lambda_1 \oplus \lambda_2$ dort l Nullen (siehe Abb. 4.3). Umso mehr Nullen dort erzeugt werden, desto geringer ist das Gewicht von λ . Und umso näher kommt man an das Ziel, e zu erhalten. Umso größer jedoch \mathcal{L} wird, umso unwahrscheinlicher wird es, dass man zwei Linearkombinationen findet, die in diesen l Spalten übereinstimmen.

Da in den Spalten des Informationsfensters \mathcal{I} die Einheitsmatrix steht, entstehen dort durch die Addition von $2p$ Zeilen auch exakt $2p$ Einsen. Deshalb dürfen in den restlichen Spalten maximal $t - 2p$ Einsen erzeugt werden.

Findet man keine zwei Linearkombinationen mit dieser Eigenschaft, muss man ein neues Informationsfenster bestimmen. Beim GISD Algorithmus war dies durch den Gauss Algorithmus die teuerste Operation. Der Stern-GS Algorithmus verwendet eine andere Methode, um ein neues Informationsfenster zu erhalten. Und zwar wird dazu das aktuelle Informationsfenster an einer Position verändert. Man erhält das neue Informationsfenster \mathcal{I}' , indem man eine Spalte aus dem Informationsfenster \mathcal{I} entfernt und durch eine andere Spalte ersetzt, die noch nicht Teil des Informationsfensters ist. Man spricht dann von dichten Informationsfenstern.

Definition 7 Zwei Informationsfenster \mathcal{I} und \mathcal{I}' sind dicht, wenn

$$\exists q \in \mathcal{I}, \exists p \in N \setminus \mathcal{I}, \text{ so dass } \mathcal{I}' = (\mathcal{I} \setminus \{q\}) \cup \{p\}.$$

Um den Wechsel von \mathcal{I} zu \mathcal{I}' zu vollziehen, benutzt man den Algorithmus 4. Man erhält \mathcal{I}' also durch den Austausch der p -ten mit der q -ten Spalte. Dies kann man durch eine simple Pivot Operation in der Position (p, q) erhalten. Man addiert folglich zu jeder Zeile i die p -te Zeile dazu, falls das korrespondierende Element $z_{iq} = 1$ ist.

4.2.2 Theoretische Laufzeit

Um die durchschnittliche Laufzeit des Algorithmus zu bestimmen, werden wir wie beim GISD Algorithmus die Anzahl der Iterationen bis zum Erfolg abschätzen sowie den Aufwand pro Iteration. Die Laufzeit ergibt sich dann aus dem Produkt dieser beiden Kennzahlen.

Definition 8 Die Funktion $Pr(a = b | c)$ gibt die Wahrscheinlichkeit an, dass die Variable a den Wert b unter der Bedingung c annimmt.

Algorithmus 4 iterative Informationsfenster**Input:**

- aktuelles Informationsfenster \mathcal{I}
- Spaltenindex $p \in \mathcal{I}$ aus dem aktuellen Informationsfenster
- Spaltenindex $q \notin \mathcal{I}$, der nicht zum aktuellen Informationsfenster gehört, mit $Z_{pq} = 1$

Output:

- neues Informationsfenster \mathcal{I}'

$$Z' = Z$$

$$\forall i \in \mathcal{I}' \setminus \{p\}, \forall j \in N \setminus (\mathcal{I}' \cup \{q\}) \quad z'_{ij} = z_{ij} \oplus z_{ip} * z_{qj}$$

$$\forall i \in \mathcal{I}' \setminus \{p\} \quad z'_{iq} = z_{ip}$$

Um die Anzahl der nötigen Iterationen abzuschätzen, modelliert man den Algorithmus als stationäre Markov Kette $\{X_i\}_{i \in N}$ [5]. Dabei ist X_i eine Zufallsvariable, die die i -te Iteration des Algorithmus repräsentiert. Die Markov Kette kann man sich als endlichen Automaten vorstellen. Der Übergang vom Zustand X_n zum Zustand X_{n+1} hängt nur vom Zustand X_n ab und nicht von vorherigen Zuständen:

$$Pr(X_{n+1} = x | X_0, X_1, X_2, \dots, X_n) = Pr(X_{n+1} = x | X_n).$$

Ist man also in einem Zustand x zum Zeitpunkt n , dann hängt die Wahrscheinlichkeit zum Zeitpunkt $n + 1$ im Zustand y zu sein nur vom Zustand x ab. Ein endlicher Automat besitzt eine endliche Menge \mathcal{E} aller möglichen Zustände. Die Menge \mathcal{S} beschreibt die Menge alle Erfolgszustände und die Menge $\mathcal{F} = \mathcal{E} \setminus \mathcal{S}$ ist die Menge der Fehlerzustände. An der Kante zwischen zwei Zuständen des Automaten steht die Wahrscheinlichkeit für einen Zustandsübergang zwischen diesen Zuständen. Diese Wahrscheinlichkeiten werden in der Zustandsübergangsmatrix P zusammengefasst. Ein Element (u, v) der Matrix stellt die Wahrscheinlichkeit dar, vom Zustand u in den Zustand v zu wechseln. Daraus folgt, dass jeder Eintrag der Matrix nicht negativ ist und die Summer aller Einträge in einer Spalte oder Zeile genau 1 ergibt. Wie die genauen Einträge der Matrix lauten, kann man in [6] nachlesen.

Die Markov Kette ist dann durch ihre Startverteilung π_0 , aufgrund derer man in den ersten Zustand gelangt, und ihre Übergangsmatrix P mit

$$Pr(X_i = u | X_{i-1} = v) = P_{v,u} \quad \forall i, \forall (u, v) \in \mathcal{E}$$

vollständig bestimmt.

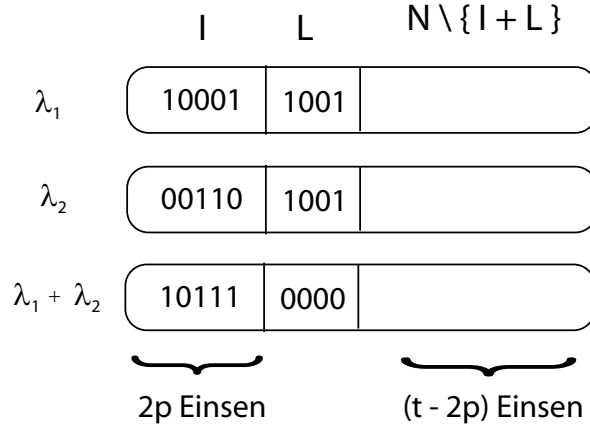


Abbildung 4.3: Stern-GS Linearkombinationen

Definition 9 Eine Markovsche Matrix $(P_{u,v})$ mit $(u, v) \in \mathcal{E}$ besteht aus Elementen $P_{u,v} \geq 0$ und $\sum_{v \in \mathcal{E}} P_{u,v} = 1$ für alle $u \in \mathcal{E}$.

Die i -te Iteration des Algorithmus wird dann durch die Wahrscheinlichkeitsverteilung π_i beschrieben:

$$\forall u \in \mathcal{E}. \pi_i(u) = Pr(X_i = u) \text{ und } \sum_{u \in \mathcal{E}} \pi_i(u) = 1.$$

Sei nun $Q = (P_{u,v})_{u,v \in \mathcal{F}}$ die substochastische Matrix, die mit den Übergängen der Markov Kette in den Zwischenzuständen korrespondiert, dann hat $(Id - Q)$ eine Inverse R :

$$R = \sum_{m=0}^{\infty} Q^m = (Id - Q)^{-1}.$$

Dann ergibt sich die Anzahl der erwarteten Iterationen N aus:

$$N = \sum_{u \in \mathcal{F}} \pi_0(u) * \sum_{v \in \mathcal{F}} R_{i,v}.$$

In jeder Iteration wird zunächst einmal ein neues Informationsfenster bestimmt. Wenn man annimmt, dass in der q -ten Spalte $k/2$ Einsen stehen, muss man zu $k/2$ Zeilen die p -te Zeile hinzuaddieren. Da $G = (id_k | Z)$ eine systematische Matrix bleibt, muss man nur im Z -Teil von G die Additionen durchführen. Das ergibt $k/2 * (n - k)$ Bitoperationen.

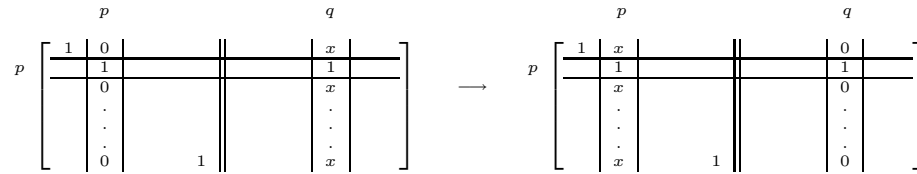


Abbildung 4.4: dichte Informationsfenster

Für die Speicherung der Linearkombinationen aus $Z_{\mathcal{I}_1}$ nimmt man $K * (p * \binom{k/2}{p} + 2^l)$ dynamische Speicherallokationen vor. K ist die Größe eines Wortes (meist 32 oder 64).

Um eine Linearkombination zu berechnen, fallen $2 * p$ Operationen auf l -bit Wörtern an. Insgesamt gibt es jeweils $\binom{k/2}{p}$ verschiedene Linearkombinationen in $Z_{\mathcal{I}_1}$ und $Z_{\mathcal{I}_2}$. Für deren Berechnung benötigt man also $2 * p * \binom{k/2}{p}$ Operationen.

Wenn zwei Linearkombinationen in \mathcal{L} übereinstimmen, berechnet man $(2p - 1)$ Additionen und eine Hamming Distanz eines $(n - k - l)$ -bit Wortes. Im Schnitt gibt es

$$\frac{\binom{k/2}{p}^2}{2^l}$$

solcher Kollisionen.

Somit ergibt sich der gesamte durchschnittliche Aufwand pro Iteration zu:

$$J = \frac{k * (n - k)}{2} + 2pl * \binom{k/2}{p} + 2p(n - k - l) * \frac{\binom{k/2}{p}^2}{2^l} + K(p * \binom{k/2}{p} + 2^l).$$

4.2.3 Experimentelle Laufzeit

Für die Laufzeitermittlung des Stern Algorithmus wurden die selben Schlüssel wie auch für den GISD verwendet. Deren Eigenschaften und die optimalen Werte für p und l kann man Tabelle 4.4 entnehmen.

Die erzielten Ergebnisse lassen sich in Tabelle 4.5 ablesen. Im Vergleich zum GISD fällt auf, dass für $m \leq 7$ der Stern Algorithmus mehr Informationsfenster benötigt. Dies hängt damit zusammen, dass für viele Informati-

m	n	k	t	p	l
4	16	8	2	1	1
5	32	17	3	1	3
6	64	34	5	1	4
7	128	72	8	1	5
8	256	153	13	1	7

Tabelle 4.4: Stern optimale Parameter

onsfenster keine Linearkombinationen gefunden werden konnten, die in der \mathcal{L} -Menge übereinstimmen, obwohl das Informationsfenster weniger als j Fehler enthält. Trotzdem ist der Stern Algorithmus schneller als der GISD, wenn man den Median der Laufzeiten betrachtet. Der Wegfall des Gauss Algorithmus macht sich folglich positiv bemerkbar.

m	4	5	6	7	8
Anzahl der Tests	5000	5000	5000	5000	50
durchschnittliche Laufzeit	1,1 ms	4,2 ms	55 ms	3,7 s	32 min
Informationsfenster	10,6	9,1	25	278	17540
Median	0 s	0 s	30 ms	2,5 s	23,1 min
5%-Quantil	0 s	0 s	10 ms	120 ms	54 s
10%-Quantil	0 s	0 s	10 ms	260 ms	74 s
Theoretische Laufzeit	$3 * 10^3$	$6 * 10^3$	$3 * 10^5$	$5 * 10^6$	$1 * 10^8$

Tabelle 4.5: Stern Ergebnisse

Als untere Schranken für die Laufzeit dienen auch hier das 10%- und 5%-Quantil. Beim Stern Algorithmus fallen ebenfalls die Werte für $m = 4$ und $m = 5$ den Messungenauigkeiten zum Opfer und können nicht bei der Extrapolation verwendet werden. Die quadratischen Formen für die beiden Quantile scheinen nicht der Realität zu entsprechen, da sie schon bei $m = 10$ (5%-Quantil) bzw $m = 12$ (10%-Quantil) den linear als auch den quadratisch extrapolierten Medianwert klar übersteigen. Als Formeln für die Quantile erhält man die linearen Funktionen $q_{10}(x) = 4,45462x - 31,7317$ und $q_5(x) = 4,29708x - 30,9917$. Für den Median erhält man die beiden Gleichungen $m_l(x) = 5,37109x - 36,0492$ und $m_q(x) = 0,94824x^2 - 7,90427x + 9,78243$.

Was das für die Laufzeiten des Stern-GS Algorithmus bei größeren m bedeutet ergibt sich aus Tabelle 4.6 und der Abb. 4.5.

Sowohl die lineare als auch die quadratische Extrapolation für den Median ergeben eine recht gute Abschätzung der tatsächlichen Laufzeit nach oben.

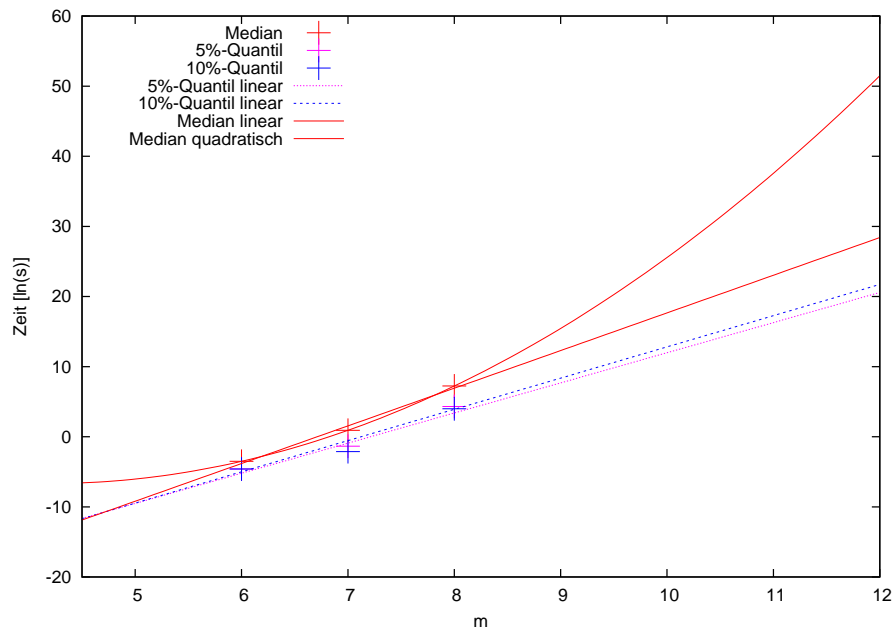


Abbildung 4.5: Stern Laufzeiten

4.3 Statistical Decoding

Der Statistical Decoding Angriff geht auf A Kh. Al Jabri zurück [11]. Die Idee besteht darin, dass das Produkt von Vektoren des von der Generatormatrix G erzeugten Vektorraums und des zugehörigen dualen Vektorraum Null ergibt. Die verschlüsselte Nachricht c ist allerdings ein verfälschter Vektor aus dem von G erzeugten Vektorraum. Das Produkt von c und einem Vektor aus dem dualen Raum ergibt folglich nicht zwangsweise Null. Wenn deren Produkt ungleich Null ist, dann verrät dies etwas über die Position der Fehler in c .

4.3.1 Funktionsweise

Der Algorithmus gliedert sich in zwei Teile. Im ersten Teil wird eine Menge \mathcal{H}_w an Vektoren mit geringem Gewicht aus dem dualen Vektorraum erzeugt. Dies ist pro Schlüssel nur einmal nötig. Anschließend wird im zweiten Teil unter Zuhilfenahme von \mathcal{H}_w eine verschlüsselte Nachricht c zu ihrem korre-

m	$e^{q_5(m)}$	$e^{q_{10}(m)}$	$e^{m_l(m)}$	$e^{m_q(m)}$
9	36 min	71 min	2,5 d	60 d
10	44 h	4,5 d	1,5 y	4.068 y
11	136 d	1 y	323 y	$6,7 * 10^8$ y
12	28 y	88 y	69.635 y	$7,3 * 10^{14}$ y
13	2.034 y	7.521 y	14.978.449 y	$5,310^{21}$ y

Tabelle 4.6: Stern Laufzeit Extrapolation

spondierendem m decodiert.

Für die Bestimmung von \mathcal{H}_w berechnet man zunächst die Parity Check Matrix H . Diese erhält man recht leicht aus der Generatormatrix G , da $G * H = 0$ ist. Wendet man den Gauss Algorithmus auf G an, um diese zu systematisieren, erhält man $G = (id_k | R) * P$. Falls man dazu Spalten in G tauschen muss, werden diese in der Permutationsmatrix P festgehalten. Somit erhält man $(id_k | R) * P * H = 0$. Da die Einträge in den Matrizen binär sind gilt:

$$(id_k | R) * \begin{pmatrix} R \\ id_k \end{pmatrix} = id_k * R + id_k * R = R + R = 0.$$

Folglich ergibt sich

$$H = P * \begin{pmatrix} R \\ id_k \end{pmatrix}.$$

Mit der Generatormatrix H für den dualen Raum werden nun Vektoren mit einem Gewicht von exakt w bestimmt. Dazu geht man analog zum Stern Algorithmus vor. Jeder gefundene Codevektor wird dann in \mathcal{H}_w abgespeichert. Wie das w zu wählen ist, ist fraglich. Es ist kaum etwas über den minimalen Abstand des dualen Codes bekannt [16]. Ebenso ungewiss ist, wie viele Codewörter mit einem bestimmten Gewicht existieren.

Im zweiten Schritt findet das eigentliche Decodieren statt. Dazu wird die verschlüsselte Nachricht $c = (c_0, c_1, \dots, c_{n-1})$ mit jedem Vektor $h = (h_0, h_1, \dots, h_{n-1})$ aus \mathcal{H}_w skalarmultipliziert. Ist das Ergebnis Null, erhält man keine Information über den Fehlervektor. Ist das Ergebnis allerdings 1, dann weiß man, dass ein Fehler an einer der Positionen i sein muss, wo $h_i = 1$ ist. Dies erklärt sich daraus, dass $c = g \oplus e$ ist und $g * h = 0$, da sie aus zueinander orthogonalen Räumen stammen. Somit ergibt sich $c * h$ zu $e * h$. Ist dieses Produkt nun 1 bedeutet das, dass h an derselben Stelle wie e eine 1 stehen hat. Allerdings weiß man nicht, welche Stelle von h das ist. Deshalb zählt man im Vektor v für jede Position von h , wie oft sie bei einem

Skalarprodukt von $c * h = 1$ ebenfalls den Wert von 1 hatte. Interessant sind nun die Positionen, die einen besonders hohen und einen besonders niedrigen Zählerstand haben. e kann auf zwei verschiedene Weisen g verändern. Es kann aus einer 1 eine 0 machen oder umgekehrt aus einer 0 eine 1. Die Positionen mit den hohen Werten spiegeln den Fall e macht 0 zu 1 wider, die niedrigen den anderen Fall. Die Wahrscheinlichkeit, dass ein Vektor h aus \mathcal{H}_w an einer fehlerhaften Position eine 1 hat, beträgt:

$$p = \frac{\sum_{m \leq t}^{m \text{ odd}} \binom{n-t}{w-m} \binom{t-1}{m-1}}{\sum_{m \leq t}^{m \text{ odd}} \binom{t}{m} \binom{n-t}{w-m}}.$$

Die Wahrscheinlichkeit dafür, dass h eine 1 an einer fehlerfreien Stelle hat, ist:

$$q = \frac{\sum_{m \leq t}^{m \text{ odd}} \binom{n-t-1}{w-m-1} \binom{t}{m}}{\sum_{m \leq t}^{m \text{ odd}} \binom{t}{m} \binom{n-t}{w-m}}.$$

Um alle möglichen Fehlermuster herausfiltern zu können, benötigt man $|\mathcal{H}_w| = 625 * 10^{-6} * p(1-p) * \epsilon^{-2}$ viele Vektoren h aus dem dualen Vektorraum. ϵ ist hierbei viel kleiner zu wählen als $|p - q|$.

Algorithmus 5 Statistical Decoding

Input:

- Matrix \mathcal{H}_w , deren Zeilenvektoren aus dem dualen Raum stammen
- den verschlüsselten Codevektor c

Output:

- decodierte Nachricht m

$$v = \sum_{h \in \mathcal{H}_w} (c * h^T \text{ mod } 2); v \in Z^n$$

wähle $\mathcal{I}_1 = \{\text{Positionen der } k \text{ größten Einträge in } v\}$ so, dass $G_{\mathcal{I}_1}$ invertierbar ist

wähle $\mathcal{I}_2 = \{\text{Positionen der } k \text{ kleinsten Einträge in } v\}$ so, dass $G_{\mathcal{I}_2}$ invertierbar ist

$$m_1 = c_{\mathcal{I}_1} * G_{\mathcal{I}_1}^{-1}$$

$$m_2 = c_{\mathcal{I}_2} * G_{\mathcal{I}_2}^{-1}$$

if $|m_1 G \oplus c| \leq t$ **then**

$$m = m_1$$

else

$$m = m_2$$

end if

Wenn man für alle h das Skalarprodukt berechnet hat, wählt man die höchsten k Positionen aus v so, dass die Matrix $G_{\mathcal{I}_1}$ aus den korrespondier-

enden k Spalten von G invertierbar ist. Von c bildet man in gleicher Weise den Subvektor $c_{\mathcal{I}_1}$. Die Klartextnachricht ergibt sich dann aus $m = c_{\mathcal{I}_1} * G_{\mathcal{I}_1}^{-1}$, falls $|m_1 * G \oplus c| \leq t$ ist. Ist die Bedingung nicht erfüllt, vollzieht man das gleiche Prozedere nochmal mit den k niedrigsten Positionen in v .

m	t	w	p	q	p-q
4	2	5	0,5	0,285714285714	0,214285714286
		6	0,5	0,357142857143	0,142857142857
5	3	5	0,331444759207	0,137540295008	0,193904464198
		6	0,329949238579	0,171713635568	0,158235603011
		7	0,327868852459	0,205765969474	0,122102882985
		8	0,325088339223	0,239673449494	0,0854148897283
		9	0,321473951715	0,273408403803	0,0480655479122
		10	0,316872427984	0,306939123031	0,00993330495246
6	5	7	0,207919244255	0,101023161987	0,106896082268
		8	0,2114013812	0,117676280004	0,0937251011956
		9	0,215612136512	0,134266815168	0,081345321344
		10	0,220584186689	0,150791545708	0,0697926409813
		11	0,226343302385	0,167247690011	0,0590956123741
		12	0,23290669974	0,183633021019	0,0492736787215
		13	0,240281389252	0,199945979525	0,0403354097263
7	8	20	0,173391318023	0,155100597043	0,0182907209802
		21	0,178801068925	0,163071245073	0,0157298238512
		22	0,184447226799	0,171025652985	0,013421573814
		25	0,202653593501	0,194800622013	0,00785297148774
		30	0,236245651835	0,23419041113	0,00205524070547
8	13	16	0,0896351814976	0,0610479343878	0,0285872471099
		26	0,112683673945	0,100962958044	0,0117207159009
		40	0,157817370159	0,156129087309	0,00168828285056
		50	0,19398337392	0,19528429398	-0,00130092005956
		60	0,230670537428	0,23437034499	-0,0036998075621

Tabelle 4.7: Statistical Decoding Parameter

4.3.2 Theoretische Laufzeit

Der Aufwand für die Bestimmung der Menge \mathcal{H}_w ist nicht genau bekannt.

Für das Decodieren des Schlüsseltextes c benötigt man zunächst einmal $|\mathcal{H}_w|$ Skalarmultiplikationen von n -bit Vektoren: $O(n^2 * |\mathcal{H}_w|)$. Hinzu kommen

zwei Invertierungen von $k \times k$ Matrizen, was einem Aufwand von $O(2k^3)$ entspricht. Für den abschließenden Test, ob man die richtige Version der Nachricht m erhalten hat, benötigt man $k * n$ Operationen. Insgesamt ergibt sich somit ein Aufwand von

$$O(n^2 * |\mathcal{H}_w| + 2k^3 + kn).$$

4.3.3 Experimentelle Laufzeit

Zunächst einmal werden die Vektoren des dualen Raums ermittelt. Für $m = 6$ wird dafür bei 5000 öffentlichen Schlüsseln versucht, Vektoren des dualen Raums der maximalen Länge von 13 zu finden. In Tabelle 4.8 sind die Gewichte der gefundenen Vektoren aufgezählt und wie oft ein bestimmtes Gewicht registriert wurde (Median). Im Schnitt dauert dies etwa 20 Sekunden pro Schlüssel. Dem gegenübergestellt ist die Menge von nötigen Vektoren $|\mathcal{H}_w|$ bei verschiedenen Werten für ϵ , um alle Codewörter decodieren zu können.

Gewicht	Median	$ \mathcal{H}_w $ für $\epsilon =$		
		$0,01 * p - q $	$0,25 * p - q $	$0,1 * p - q $
8	2	119	2	1
9	1	160	2	1
10	3	221	3	1
11	28	314	4	1
12	132	460	5	1
13	647	702	8	2

Tabelle 4.8: Anzahl der Vektoren des dualen Raums für $m=6$

Der Decodiervorgang wird für jeden öffentlichen Schlüssel und jedes im zugehörigen dualen Raum gefundene Gewicht 100 mal durchgeführt. Die unterschiedliche Anzahl an Tests für die verschiedenen Gewichte ergibt sich folglich daraus, dass nicht zu allen öffentlichen Schlüsseln Vektoren mit allen verschiedenen Gewichten gefunden wurden. Die durchschnittliche Laufzeit einer Decodierung beträgt weniger als 10 ms. Für $m = 6$ ist ein Gewicht von 13 das mit der höchsten Erfolgsquote. Mit etwas mehr Zeitaufwand lassen sich auch noch mehr Vektoren von diesem Gewicht finden, was die Erfolgsrate weiter steigern dürfte.

Für $m = 7$ dauert das Finden von 240.000 Vektoren des dualen Raums etwa fünf Stunden. Diese Vektoren haben ein maximales Gewicht von 30. Wie sich die einzelnen Gewichte zusammensetzen, wird aus Tabelle 4.10 ersicht-

Gewicht	Tests	Erfolgsquote
8	23200	3,3%
9	171300	3,4%
10	481900	3,3%
11	498100	6,3%
12	500000	26,2%
13	500000	67,5%

Tabelle 4.9: Decodierergebnisse für $m=6$

lich. Der Median ergibt sich aus der Berechnung von $|\mathcal{H}_w|$ für 30 verschiedene Public Keys.

Für jeden Schlüssel und jedes Gewicht erfolgen wiederum 100 Decodierungsversuche. Wie man in Tabelle 4.11 sehen kann, ist die Erfolgsquote von 1% sehr gering. Ob man bei größeren Gewichten oder mit mehr Vektoren pro Gewicht, das Ergebnis in gewünschte Regionen befördern kann, ist fraglich.

Für $m = 8$ sieht es noch schlechter aus. In 92 Stunden ließen sich 330.000 Vektoren mit einem Gewicht kleiner als 50 finden. Jedoch langt es bei keinem Gewicht zu einem Erfolg beim Decodieren bei jeweils 1000 Versuchen. In den 92 Stunden kann man mit dem Stern Algorithmus ca. 240 Codewörter decodieren. Da man für $m = 8$ noch erheblich mehr Zeit in das Konstruieren von \mathcal{H}_w investieren muss, lohnt sich das nur, wenn man eine größere Menge an Nachrichten decodieren möchte. Auch hier sind jedoch die Erfolgsaussichten unbekannt.

Gewicht	Median	$ \mathcal{H}_w $ für $\epsilon =$		
		$0,01 * p - q $	$0,25 * p - q $	$0,1 * p - q $
16	1	830	9	2
17	1	1091	11	2
18	1,5	1449	15	3
19	3	1948	20	4
20	21,5	2650	27	5
21	91,5	3651	37	6
22	471,5	5095	51	9
23	1569	7208	73	12
24	4339	10345	104	17
25	9638,5	15075	151	25
26	17819	22326	224	36
27	29232,5	33644	337	54
28	43560,5	51653	517	83
29	59169,5	80925	810	130
30	74082,5	129624	1297	208

Tabelle 4.10: Anzahl der Vektoren des dualen Raums für $m=7$

Gewicht	Tests	Erfolgsquote
16	100	1,0%
17	400	0,3%
18	1400	0,2%
19	2900	0,1%
20	3000	0,1%
21	3000	0,4%
22	3000	1,1%
23	3000	1,6%
24	3000	1,3%
25	3000	1,0%
26	3000	0,8%
27	3000	0,1%
28	3000	0,2%
29	3000	0,2%
30	3000	0,1%

Tabelle 4.11: Decodierergebnisse für $m=7$

Gewicht	Median	$ \mathcal{H}_w $ für $\epsilon =$		
		$0,01 * p - q $	$0,1 * p - q $	$0,25 * p - q $
32	1	18008	181	29
33	1	23046	231	37
34	3	29625	297	48
35	4	38251	383	62
36	17	49611	497	80
37	48	64638	647	104
38	86	84609	847	136
39	232	111274	1113	179
40	445	147045	1471	236
41	945	195269	1953	313
42	1818	260608	2607	417
43	3574	349592	3496	560
44	6559	471421	4715	755
45	11802	639130	6392	1023
46	19911	871292	8713	1395
47	32935	1194540	11946	1912
48	52131	1647303	16474	2636
49	80303	2285391	22854	3657
50	119185	3190411	31905	5105

Tabelle 4.12: Anzahl der Vektoren des dualen Raums für $m=8$

Kapitel 5

Fazit

Wenn man eine Nachrichten decodieren möchte, ist dazu am besten der Stern-GS Algorithmus geeignet. Er ist klar schneller als der GISD Algorithmus und da er im Gegensatz zum Statistical Decoding auch immer erfolgreich ist, ist er klar die erste Wahl. Falls es gelingt, genügend Vektoren für die Menge \mathcal{H}_w zu finden, ist das Statistical Decoding eine Alternative zum Stern Algorithmus. Allerdings ist fraglich, ob es überhaupt genügend geeignete Vektoren gibt und wie lange es dauert sie zu berechnen. Hinzu kommt, dass sich dieser große Aufwand nur dann lohnt, wenn man auch eine hinreichend große Zahl an Nachrichten decodieren möchte, die alle von demselben Schlüssel generiert wurden.

Die erzielten Ergebnisse der Laufzeitmessung reichen nicht aus, um einen bestimmten Parametersatz als sicher zu deklarieren. Allerdings kann man sagen, dass für $m \leq 10$ das McEliece Kryptosystem nicht sicher ist. Da die Laufzeit für das 10%-Quantil bei $m = 11$ nur ein Jahr beträgt und der lineare Median bei 323 Jahren liegt, kann man davon ausgehen, dass auch für $m = 11$ das McEliece System nicht sicher ist.

Kapitel 6

Code-Beschreibung

In diesem Abschnitt werden die grundlegenden Designentscheidungen beschrieben, die bei der Implementierung der Angriffe getroffen wurden. Des Weiteren wird erläutert, wie das Programm zu benutzen ist.

6.1 Design

Die Implementierung gliedert sich in zwei Abschnitte. Zum einen die mathematischen Hilfsklassen, die Vektoren und Matrizen abbilden. Zum anderen die eigentlichen Angriffe GISD und Stern-GS. Einen Überblick über die Klassenstruktur liefert die Abbildung 6.1.

Beim mathematischen Teil stellen die Klassen *BitVector* und *BitMatrix* grundlegende Operationen für binäre Vektoren und Matrizen bereit. Dazu gehören neben Matrix-Matrix Multiplikation, Matrix-Vektor Multiplikation und dem Skalarprodukt von Vektoren auch die Möglichkeit, den Gauss Algorithmus auf eine Matrix anzuwenden. Eine $n \times m$ Matrix ist als n Zeilenvektoren der Länge m implementiert. Da das Invertieren von Matrizen nur bei quadratischen Matrizen funktionieren kann, existiert dafür eine extra Klasse *BitSquareMatrix*, die von der Klasse *BitMatrix* erbt.

Die *BitProxy* Klasse ermöglicht es, ein binäres Datum in einem Bit zu speichern. In C++ wird ein bool nämlich intern als int betrachtet. Somit wird für ein binäres Datum als bool-Wert ein Wort benötigt, was auf gängigen Computern 32 Bit entspricht. Doch das hat nicht nur Auswirkungen auf den verwendeten Speicherplatz, sondern auch auf die Rechengeschwindigkeit. Wird ein Binärwert als bool repräsentiert, kann nur eine binäre Operation auf einmal in einem Register ausgeführt werden. Ein int und somit ein Register bietet jedoch Platz für 32 Binärwerte. Es sind also theoretisch 32 Binäroperationen in derselben Zeit möglich. Um dies zu erreichen, speichert die Klasse

BitProxy zusammengehörende Binärwerte eines Vektors intern in einem `int`. Nach außen hin sieht es jedoch so aus, als ob jedes Bit seinen eigenen Speicherplatz hat. Verfügt man über einen Computer mit 64-bit Registern, erhöht sich dieser Vorteil noch mehr.

Bei den Angriffsklassen gibt die Basisklasse *Attack* das Interface für die Ausführung der Angriffe vor. Die Klassen *GISD* und *SternGS* implementieren dann ihre jeweiligen Angriffe. Der GISD Angriff kann komplett mit den mathematischen Grundlagen der Basisklassen implementiert werden. Beim Stern-GS Angriff kommt noch die Speicherung der Linearkombinationen hinzu. Dazu wurde eine *Multimap* verwendet, da sie in $O(\log n)$ gefüllt und gelesen werden kann. Ein weiterer Punkt bei der Implementierung des Stern Algorithmus ist, dass die Generatormatrix nicht in dem Sinne systematisiert wurde, dass im vorderen Teil die Identität steht. Die einzelnen Teile der Identität sind über die komplette Generatormatrix verstreut. Dies ändert allerdings nichts an dem eigentlichen Algorithmus.

Für den Statistical Decoding Angriff existiert keine eigene Klasse. Zur Konstruktion der \mathcal{H}_w Menge wird allerdings der Stern-GS Algorithmus verwendet.

6.2 Programmaufruf

Dieser Abschnitt erläutert wie die Programme zu verwenden sind. Das Programm *attack* führt Angriffe mit den Algorithmen GISD und Stern-GS durch. Das Programm *statistic* erledigt die verschiedenen Schritte des Statistical Decoding.

6.2.1 attack

Das Programm ist dazu bestimmt, die Zeit zum Decodieren einer verschlüsselten Nachricht c zu messen. Es wird folgendermaßen aufgerufen:

```
attack keyfile logfile #runs (GISD j | STERN p l | GISD_TIME j
                             stop_time)
```

Beim ersten Parameter *keyfile* handelt es sich um den Namen der Datei, die den öffentlichen Schlüssel enthält, mit dem Nachrichten verschlüsselt werden. Der Aufbau dieser Datei ist in Kapitel 6.3.2 beschrieben. Die Schlüssel wurden mit dem Flexiprovider [17] erzeugt. Der nächste Parameter gibt die Log-Datei an, in der alle Messergebnisse protokolliert werden. Deren Aufbau wird in Kapitel 6.3.1 behandelt. Der Parameter *#runs* enthält die Anzahl an

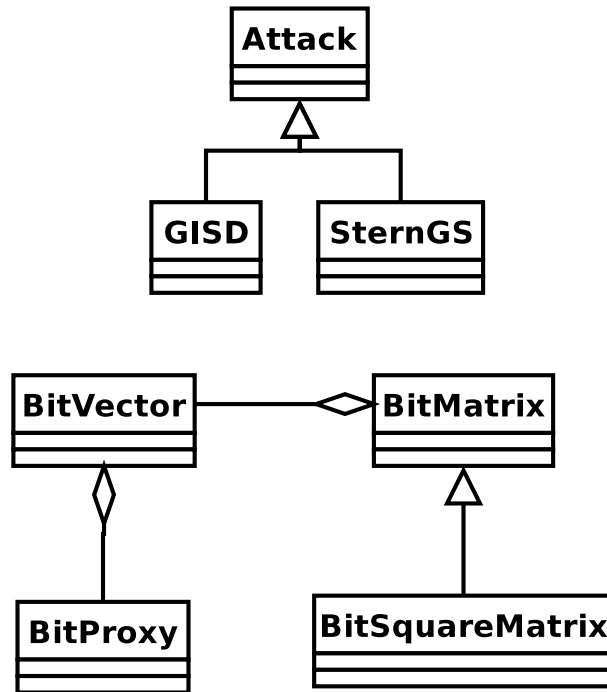


Abbildung 6.1: UML Klassendiagramm

Tests, die durchgeführt werden sollen. Für jeden Test wird ein neues Klartext - Schlüsseltext Paar erzeugt. Nun folgt ein Schlüsselwort, das signalisiert, welcher Angriffsalgorithmus benutzt werden soll. *GISD* für den GISD Algorithmus und *STERN* für den Stern-GS Algorithmus. Das Schlüsselwort *GISD_TIME* aktiviert den GISD Algorithmus und bricht ihn nach einer gewissen Anzahl an Sekunden ab (*stop_time*), falls bis dahin die Nachricht nicht entschlüsselt werden konnte. Anschließend werden die für den jeweiligen Angriff nötigen Parameter angegeben.

6.2.2 statistic

Das Statistical Decoding ist in zwei Arbeitsschritte gegliedert. Zunächst einmal muss die \mathcal{H}_w -Datei mit geeigneten Vektoren aus dem dualen Raum angelegt werden. Im nächsten Schritt wird dann mit dieser Vektormenge der

Angriff vollzogen.

Um die Vektoren aus dem dualen Raum zu erhalten, starten man das Programm wie folgt:

```
statistic FIND keyfile H-File #nr #stop weight p l [MAX]
```

keyfile enthält den öffentlichen Schlüssel, der wie in Abschnitt 6.3.2 beschrieben aufgebaut ist. In das *H-File* werden alle gefundenen Codewörter gespeichert. Dabei enthält eine Zeile dieser Datei genau ein Codewort. *#nr* gibt an, wie viele Codewörter vom Gewicht *weight* insgesamt gefunden werden sollen. Wird der optionale Parameter *MAX* verwendet, werden auch Codewörter mit einem geringeren Gewicht als *weight* gespeichert. *p* und *l* sind die Parameter, mit denen der Stern Algorithmus zum Finden der kurzen Codewörter aufgerufen wird. Über den *#stop* Parameter wird ein Abbruchkriterium definiert. Werden *#stop* viele Codewörter hintereinander ermittelt, die alle im Laufe der Suche bereits errechnet wurden, dann wird das Programm beendet. Dieser Parameter ist nötig, da nicht bekannt ist, wie viele Vektoren des gewünschten Gewichts im dualen Raum überhaupt vorkommen. Somit kann das Programm terminieren, obwohl nicht genügend Vektoren im dualen Raum vorhanden sind.

Hat man den Parameter *MAX* verwendet, kann man das *H-File* noch nicht direkt für Angriffe nutzen, da dafür im *H-File* nur Codewörter desselben Gewichts vorkommen sollten. Um die Datei dahingehend zu transformieren, startet man das Programm so:

```
statistic CONVERT H-File max_weight
```

Dabei wird für jedes vorkommende Gewicht *i* eine eigene Datei erzeugt und alle Codewörter mit diesem Gewicht dort gespeichert. Der Dateiname ergibt sich aus dem Dateinamen vom *H-File* durch das Anhängen von "*_hi*". Falls schon eine Datei mit diesem Namen existiert, so wird sie überschrieben. *max_weight* gibt das höchste vorkommende Gewicht in *H-File* an. Will man eine Statistik darüber, welche Gewichte wie oft im *H-File* vertreten sind, erhält man diese über:

```
statistic ANALYZE H-File max_weight
```

Hat man sich ein passendes *H-File* erzeugt, kann man nun darangehen, einen Angriff zu starten:

```
statistic ATTACK keyfile H-File #nr
```

keyfile gibt dabei den Public Key an und *H-File* steht für die Datei mit den Vektoren aus dem dualen Raum. *#nr* steht für die Anzahl der durchzuführenden Angriffe. Als Ausgabe erhält man für jeden Angriff eine solche Zeile:

File m7/m7t8v1006.public.h_h20 time: 0 status: failed

Sie beinhaltet das verwendete *H-File*, die Laufzeit in Sekunden und ob das Decodieren geklappt hat ("ok") oder nicht ("failed").

6.3 Dateiformate

6.3.1 Aufbau der Log Datei

```

GISD j=2 Public Key: m5t3v1.public IWs = 1 ok time: 0.03
GISD j=2 Public Key: m5t3v1.public IWs = 1 ok time: 0.16
GISD j=2 Public Key: m5t3v1.public IWs = 3 ok time: 0.22
GISD j=2 Public Key: m5t3v1.public IWs = 1 ok time: 0.04
GISD j=2 Public Key: m5t3v1.public IWs = 1 ok time: 0.23
GISD j=2 Public Key: m5t3v1.public IWs = 6 ok time: 0.13
GISD j=2 Public Key: m5t3v1.public IWs = 5 ok time: 0.21
GISD j=2 Public Key: m5t3v1.public IWs = 2 ok time: 0.25
GISD j=2 Public Key: m5t3v1.public IWs = 2 ok time: 0.15
GISD j=2 Public Key: m5t3v1.public IWs = 1 ok time: 0.2
STERN p=1 l=4 Public Key: m5t3v1.public IWs = 6 ok time: 0.1
STERN p=1 l=4 Public Key: m5t3v1.public IWs = 13 ok time: 0.17
STERN p=1 l=4 Public Key: m5t3v1.public IWs = 13 ok time: 0.17
STERN p=1 l=4 Public Key: m5t3v1.public IWs = 11 ok time: 0.15
STERN p=1 l=4 Public Key: m5t3v1.public IWs = 19 ok time: 0.23
GISD_TIME j=2 stop: 600 Public Key: m8t13v3.public IWs = 74 failed
time: 632

```

Abbildung 6.2: Log File

Jede Zeile der Log Datei steht für einen Durchlauf. Zunächst wird über die Schlüsselwörter *GISD* oder *STERN* angegeben, welcher Angriff durchgeführt wurde. Danach folgen die für den jeweiligen Angriff spezifischen Parameter. Im Fall von *GISD* also nur j , bei Stern-GS wird zuerst der Wert für p und dann der von l aufgeführt. Anschließend kommt *Public Key*: gefolgt vom Dateinamen des verwendeten öffentlichen Schlüssels. $IWs = 5$ bedeutet, dass

zur Decodierung 5 Informationsfenster berechnet wurden. Das *ok* bedeutet, dass die Decodierung erfolgreich war, also auch die korrekte Klartextnachricht ermittelt wurde. Weicht das berechnete m vom Ursprungs- m ab, würde anstelle von *ok* ein *failed* erscheinen. Das sollte allerdings nicht passieren. Kommt es doch vor, deutet das auf einen Bug in der Programmierung hin oder die angriffsspezifischen Parameter waren unzulässig. Die *time* Passage gibt schlussendlich an, wie lange die Decodierung der Nachricht gedauert hat. Für $2^m = n \leq 6$ wird die Zeit millisekundengenau gemessen. Für größere Codes wird die Zeitmessung nur noch sekundengenau betrieben. Falls der *GISD_TIME* Modus benutzt wird, bedeutet ein *failed* Eintrag, dass der Algorithmus vorzeitig beendet wurde, während *ok* wie gewöhnlich anzeigt, dass die Decodierung gelungen ist. Die Laufzeit des Algorithmus wird immer etwas länger sein als die vorgegebene Stop-Zeit, da die Zeitüberschreitung nur bei der Berechnung eines neuen Informationsfensters stattfindet.

6.3.2 Aufbau der Public Key Datei

Der generelle Aufbau der Datei entspricht einer Folge von Kommentaren und Daten. Zuerst kommt eine Kommentarzeile, die angibt, um was für Daten es sich im Folgenden handelt. Kommentarzeilen erkennt man daran, dass sie mit zwei Sternchen begonnen und abgeschlossen werden. Am Anfang der Datei steht die Länge n des Codes. Anschließend wird die Dimension k des Coderaums gelistet. Es folgt die maximale Anzahl t von Fehlern, die der Code korrigieren kann. Diese Daten sind alle einzeilig. Die folgenden zwei Zeilen sind für das Programm nicht von Bedeutung und werden folglich ignoriert. Den Abschluss bildet die Generatormatrix G . Jede der nun folgenden k Zeilen enthält auch eine Zeile der Generatormatrix.

```
**n**  
16  
**k**  
8  
**t**  
2  
**fieldPoly**  
11001  
**PKFileGPub**  
0100010110000101  
1010010001100100  
0101000010100001  
1010000010100010  
0110110000100001  
1010000010001101  
0110000000110110  
1100011000100000
```

Abbildung 6.3: Public Key File

Literaturverzeichnis

- [1] C.H. Bennet, E. Bernstein, G. Brassard und U. Vazirani: Strengths and Weaknesses of Quantum Computing. preprint 1994
- [2] E.R. Berlekamp: Algebraic Coding Theory McGraw-Hill, 1968
- [3] E.R. Berlekamp, R.J. McEliece, and H.C.A. Van Tilborg: On the inherent intractability of certain coding problems. IEEE Trans. Inform. Theory, IT-24(3), pp. 384-386, 1978.
- [4] Johannes Buchmann: Einführung in die Kryptographie, 2. erweiterte Auflage ISBN 3-540-41283-2, Springer 2001
- [5] Anne Canteaut, Florent Chabaud: Improvements of the Attacks on Cryptosystems Based on Error-correcting Codes Rapport interne du Département Mathématiques et Informatique LIENS-95-21, Ecole Normale Supérieure, Paris, July 1995
- [6] Anne Canteaut, Florent Chabaud: A New Algorithm for Finding Minimum-Weight Words in a Linear Code: Application to McEliece's Cryptosystem and to Narrow-Sense BCH Codes of Length 511 In: IEEE Transaction on Information Theory, Vol 44, No 1, 1998
- [7] Florent Chabaud: On the Security of Some Cryptosystems Based on Error-correcting Codes In: EUROCRYPT 94, LNCS 950, pp. 131-139, 1995
- [8] Ernst M. Gabidulin: Public-key cryptosystems based on linear codes In: Reports of the Faculty of Technical Mathematics and Informatics 95-30, Delft University of Technology, ISSN 0922-5641, Delft, 1995
- [9] P.J. Lee, E.F. Brickell: An Observation on the Security of McEliece's Public-Key Cryptosystem In: EUROCRYPT 88, LNCS 330, pp. 175-280, 1988

- [10] R.J. McEliece: A Public-Key Cryptosystem Based On Algebraic Coding Theory In: DSN Progress Report 42-44, 1978
- [11] A Kh. Al Jabri: A New Class Of Attacks On McEliece Public-Key And Related Cryptosystems
- [12] Thomas Johansson, Fredrik Jönsson: On the Complexity of Some Cryptographic Problems Based on the General Decoding Problem
- [13] P.W. Shor: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer SIAM J. of Comput., 26(5), pp. 1484-1509, 1997.
- [14] J. Stern: A method of finding codewords of small weight In: Coding Theory and Applications, LNCS 388, pp. 106-113, 1989
- [15] Johan van Tilburg: On the McEliece Public-Key Cryptosystem In: S. Goldwasser (editor) Advances in Cryptology - CRYPTO '88, LNCS 403, pp. 119-131, 1990
- [16] F.Levy dit Vehel, S. Litsyn: Parameters of Goppa Codes revisited. In: IEEE Transactions on Information Theory, Vol 43, No 6, pp. 1811-1819, 1997
- [17] <http://flexiprovider.cdc.informatik.tu-darmstadt.de/>

Anhang A

Programm-Code

A.1 Attacks

```
#include <numeric>
#include <iterator>
#include <ctime>

int main(int argc, char *argv[])
{
    BitVector::InitStatics();
    if (argc < 5) {
        cout << "usage: attack keyfile logfile #runs (GISD j | STERN p l
| GISD_TIME j stop_time) " << endl;
        return 0;
    }

    //Preparation steps
    //
    //read G
    ifstream in(argv[1]);
    if (!in.is_open()) {
        cout << "File reading error" << endl;
        return 0;
    }
    BitMatrix G;
    unsigned int t = G.ReadFile(in);
    in.close();

    //open logfile
    ofstream log(argv[2], ofstream::app | ofstream::out);
```

```

srand (time(0));

//set parameters and start timer
int nrRuns = atoi(argv[3]);
double total_time = 0.0;
Attack* attack;
string log_prefix;
unsigned int stop_time = -1;

if (strcmp(argv[4], "GISD") == 0) {
    if (argc < 6)
        cout << "using default parameter for j: 2" << endl;
    int j = (argc < 6) ? 2 : atoi(argv[5]);
    attack = new GISD(G, t, j);
    ostringstream ost;
    ost << "GISD j=" << j << " ";
    log_prefix = ost.str();
}
else if (strcmp(argv[4], "GISD_TIME") == 0) {
    if (argc < 6)
        cout << "using default parameter for j: 2" << endl;
    if (argc < 7)
        cout << "using default parameter for stop_time: 600" << endl;
    int j = (argc < 6) ? 2 : atoi(argv[5]);
    stop_time = (argc < 7) ? 600 : atoi(argv[6]);
    attack = new GISD(G, t, j);
    ostringstream ost;
    ost << "GISD_TIME j=" << j << " stop: " << stop_time << " ";
    log_prefix = ost.str();
}
else if (strcmp(argv[4], "STERN") == 0) {
    if (argc < 7)
        cout << "using default parameter for l: 4" << endl;
    int l = (argc < 7) ? 4 : atoi(argv[6]);
    if (argc < 6)
        cout << "using default parameter for p: 2" << endl;
    int p = (argc < 6) ? 2 : atoi(argv[5]);
    attack = new SternGS(G, t, p, l);
    ostringstream ost;
    ost << "STERN p=" << p << " l=" << l << " ";
    log_prefix = ost.str();
} else {

```

```

    cout << "GISD or STERN expected\n";
    return 1;
}

//start the attack
for (int nr = 1; nr <= nrRuns; ++nr) {
    //create m <-> c pair
    unsigned int n = G.Rows();
    unsigned int k = G.Columns();
    // cout << "n: " << n << " k: " << k << endl;
    BitVector m(n), e(k);
    do
        for(unsigned int i = 0; i < n; ++i)
            m[i] = rand()%2;
        while (m.weight() == 0);
        while (e.weight() < t)
            e[rand()%k]=1;
#ifdef verbose
    cout << "e: " << e << " |e| = " << e.weight() << endl;
#endif
    BitVector c = m*G + e;
#ifdef verbose
    cout << "m: " << m << "\nc: " << c << " |c| = " << c.weight() <<
endl;
#endif
    time_t tstart = time(0);
    clock_t start = clock();
    BitVector m2 = attack->run(c,stop_time);
    clock_t stop = clock();
    time_t tstop = time(0);
    //if m <= 6 clock counter normally is ok, for bigger ms it
overflows
    double runtime;
    if (n < 128)
        runtime = static_cast<double>(stop-start) / CLOCKS_PER_SEC;
    else
        runtime = static_cast<double>(tstop-tstart);

    total_time += runtime;

    string ok = (m == m2) ? "ok" : "failed";
#ifdef verbose
    cout << "m: " << m << endl << "m2:" << m2 << " time: " <<

```

```

runtime << "\n";
#endif
    log << log_prefix << "Public Key: " << argv[1];
    log << " IWs = " << attack->getRounds() << " " << ok << " time:
" << runtime << endl;
    }
    delete attack;
    cout << "File " << argv[1] << " Total Runs: " << nrRuns;
    cout << " Total Time: " << total_time << " Average Time: " <<
total_time/nrRuns << endl;
}

```

A.2 statistic

```

#include "BitMatrix.h"
#include "SternGS.h"
#include "BitSquareMatrix.h"

int main(int argc, char *argv[])
{
    BitVector::InitStatics();
    if (argc < 4) {
        cout << "usage: statistic FIND keyfile H-File #nr #stop weight p
l [MAX]" << endl;
        cout << "\t keyfile:\t Public Key File" << endl;
        cout << "\t H-File:\t the computed codewords get stored in here"
<< endl;
        cout << "\t #nr:\t \t how many words should be found << endl;
        cout << "\t #stop:\t\t the program aborts when this number of
duplicated codewords is found in a row << endl;
        cout << "\t weight:\t the weight of the codewords to be found"
<< endl;
        cout << "\t p,l:\t\t Stern-GS Parameter" << endl;
        cout << "\t MAX:\t\t optimal; all codewords with weight lower
than 'weight' are stored too" << endl;
        cout << "usage: statistic ATTACK keyfile H-File #nr<< endl;
        cout << "\t keyfile:\t Public Key File" << endl;
        cout << "\t H-File:\t the computed codewords get stored in here"
<< endl;
        cout << "\t #nr:\t\t number of ciphertexts to be decoded" <<
endl;
    }
}

```

```

    cout << "usage: statistic CONVERT H-File max_weight" << endl;
    cout << "\t H-File:\t the computed codewords get stored in here"
<< endl;
    cout << "\t max_weight:\t maximum weight of a Codeword found in
'H-File'" << endl;
    cout << "usage: statistic ANALYZE H-File max_weight" << endl;
    cout << "\t H-File:\t the computed codewords get stored in here"
<< endl;
    cout << "\t max_weight:\t maximum weight of a Codeword found in
'H-File'" << endl;
    return 0;
}
//read public key file and H-File
BitMatrix G;
unsigned int t = 0;
set<BitVector> short_words; //store the short words
if (strncmp(argv[1], "FIND", 4) == 0 || strncmp(argv[1], "ATTACK",
6) == 0) {

    //init first 2 params stuff
    //read G
    ifstream in(argv[2]);
    if (!in.is_open()) {
        cout << "File reading error " << argv[2] << endl;
        return 0;
    }
    t = G.ReadFile(in);
    in.close();

    //read H-File
    in.open(argv[3], ifstream::in);
    if (in.is_open()) {
        string tmp;
        while (getline(in, tmp)) {
            BitVector c(tmp);
            short_words.insert(c);
        }
    }
    in.close();
}
#ifdef verbose
    cout << "Loaded " << short_words.size() << " words from file\n";
#endif

```

```

srand (time(0));
//switch action via keywords FIND,ATTACK,CONVERT,ANALYZE
if (strncmp(argv[1], "FIND", 4) == 0) {
    if (argc < 9) {
        cout << "usage: statistic FIND keyfile H-File #nr #stop weight
p l [MAX]" << endl;
        return 0;
    }
    unsigned int nr = atoi(argv[4]);
    unsigned int red_stop = atoi(argv[5]);
    unsigned int weight = atoi(argv[6]);
    unsigned int p = atoi(argv[7]);
    unsigned int l = atoi(argv[8]);
    bool accurate = true;
    if ((argc == 10) && (strncmp(argv[9], "MAX", 3) == 0))
        accurate = false;

    //step 1: generate dual code generator matrix H

    //create systematic matrix
    BitMatrix G_tmp(G);
    BitMatrix P(G_tmp.GaussP());
    for (unsigned int i = G_tmp.Rows()-1; i > 0; --i)
        G_tmp.ZeroColAboveRow(i, i);

    set<unsigned int> cols;
    for (unsigned int i= G_tmp.Rows(); i < G_tmp.Columns(); ++i)
        cols.insert(i);
    BitMatrix H = G_tmp.SubMatrixByCols(cols);
    vector<BitVector> a(G_tmp.Columns() - G_tmp.Rows());
    BitVector v(G_tmp.Columns() - G_tmp.Rows());
    for (unsigned int i = 0; i < v.length(); ++i) {
        v[i] = 1;
        a[i] = v;
        v[i] = 0;
    }
    for (unsigned int i = 0; i < a.size(); ++i)
        H.appendRow(a[i]);
    H = P*H;
    {
        BitMatrix M = G*H;
        //check for correctness
        for (unsigned int i = 0; i < M.Rows(); ++i)

```

```

        if (M[i].weight() > 0) {
            cout << "H not correct!" << endl;
            exit(0);
        }
    }

    //step 2: find short words
    unsigned int total = 0; //total nr of found words
    unsigned int red = 0; // nr of found words, which are already
known, since last new word
#ifdef verbose
    unsigned int zehntel = nr / 10;
    unsigned int actual = zehntel;
#endif
    time_t tstart = time(0);
#ifdef verbose
    cout << tstart << " Start \n";
#endif
    while (short_words.size() < nr) {
        SternGS s(H, t, p, l);
        BitVector low = s.find_shortest(weight, p, l);
        if (accurate && low.weight() < weight) {
            cout << "zu klein" << endl;
            continue;
        }
        total++;
        pair<set<BitVector>::iterator,bool> res =
short_words.insert(low);
        if (res.second) {
            red = 0; // new word found
#ifdef verbose
            if (short_words.size() > actual) {
                cout << time(0) << " Found codeword nr " << actual <<
endl;
                actual += zehntel;
            }
#endif
        } else
            red++;
        if (red >= red_stop) {
#ifdef verbose
            cout << "redundancy stop mark reached!\n";
#endif
        }
    }
}

```

```

        break;
    }
}
time_t tend = time(0);
#ifdef verbose
    cout << tend << " Stop" << endl;
    cout << "Duration: " << tend - tstart;
    cout << " Total: " << total << " unique: " << short_words.size()
<< endl;
#endif
    //save result
    ofstream h_file(argv[3], ofstream::trunc | ofstream::out);
    for (set<BitVector>::iterator i = short_words.begin();
        i != short_words.end(); ++i)
        h_file << *i << endl;
    h_file.close();
    cout << "File " << argv[1] << " : " << short_words.size() << " /
" << nr;
    cout << " words of weight " << weight << " found in time: " <<
tend-tstart << " status: ";
    if (red >= red_stop)
        cout << "aborted";
    else
        cout << "ok";
    cout << endl;
} //if argv[3] == FIND

if (strncmp(argv[1], "ATTACK", 6) == 0) {
    if (argc < 5) {
        cout << "usage: statistic ATTACK keyfile H-File #nr<< endl;
        return(0);
    }
    unsigned int nr = atoi(argv[4]);
    for (unsigned int d = 0; d < nr; ++d) {
        //create m <-> c pair
        unsigned int k = G.Rows();
        unsigned int n = G.Columns();
        BitVector m(k), e(n);
        do
            for(unsigned int i = 0; i < k; ++i)
                m[i] = rand()%2;
        while (m.weight() == 0);
        while (e.weight() < t)

```

```

        e[rand()%n]=1;
        BitVector c = m*G + e;

#ifdef verbose
cout << "m: " << m << endl;
cout << "e: " << e << " |e| = " << e.weight() << endl;
cout << "c: " << c << endl;
#endif

        time_t tend;
        time_t tstart=time(0);
        //step 1:  $v = \sum_{h \in H_w} (c \cdot h^T \bmod 2) \cdot h$ 
        vector<unsigned int> v(n);
        BitVector h(n);
        ifstream h_file(argv[3]);
        string tmp;
        while (getline(h_file, tmp)) {
            BitVector h(tmp);
            if (c*h == 1)
                for (unsigned int i = 0; i < v.size(); ++i)
                    if (h.toBool(i))
                        v[i]++;
        }

        //step pre 2: order v positions
        typedef multimap<unsigned int, unsigned int> ORDERMAP;
        ORDERMAP ordered_v;
        for (unsigned int i = 0; i < v.size(); ++i) {
            ordered_v.insert(pair<unsigned int, unsigned int>(v[i], i));
        }

        //step 3: choose k largest entries in v
        set<unsigned int> iw;
        iw = G.invertibleIW(ordered_v, k);
        BitSquareMatrix G_k = G.SubMatrixByCols(iw);
        BitSquareMatrix G_1 = G_k.Inverse();

        BitVector m_1 = c.SubVector(iw)*G_1, m_2;
        BitVector test = m_1*G+c;
        if (test.weight() <= t) {
            tend = time(0);
#ifdef verbose
            cout << "m = " << m_1 << endl;
#endif

```

```

        if (m_1 == m) cout << "ok"; else cout << "failed";
        cout << endl;
    #endif
    } else {
        //step 4: choose k smallest entries in v
        iw = G.invertibleIW(ordered_v, k, 0);
        BitSquareMatrix G_k = G.SubMatrixByCols(iw);
        BitSquareMatrix G_2 = G_k.Inverse();

        m_2 = c.SubVector(iw)*G_2;
        test = m_2*G+c;
        tend = time(0);
        if (test.weight() > t) {
    #ifdef verbose
            cout << "statistical decoding failed!!!!\n";
    #endif
        }
    #ifdef verbose
        cout << "m = " << m_2 << endl;
        if (m_2 == m) cout << "ok"; else cout << "failed";
        cout << endl;
    #endif
    #endif
    }
    cout << "File " << argv[3] << " time: " << tend - tstart << "
status: ";
    if (m_1 == m || m_2 == m)
        cout << "ok";
    else
        cout << "failed";
    cout << endl;
}
}
if (strncmp("CONVERT", argv[1], 7) == 0) {
    //cout << "usage: statistic CONVERT H-File max_weight" << endl;
    ifstream in(argv[2]);
    if (!in.is_open()) {
        cout << "File reading error " << argv[2] << endl;
        return 0;
    }
    unsigned int max = atoi(argv[3]);
    vector<set<BitVector> > weights(max+1);
    string tmp;
    while (getline(in, tmp)) {

```

```

    BitVector test(tmp);
    weights[test.weight()].insert(test);
}

in.close();
for(unsigned i = 0; i < weights.size(); ++i) {
    if (weights[i].size() == 0)
        continue;
    ostringstream outfile;
    outfile << argv[2] << "_h" << i;
    ofstream out(outfile.str().c_str(), ofstream::trunc |
ofstream::out);
    for (set<BitVector>::const_iterator ci = weights[i].begin();
ci != weights[i].end(); ++ci)
        out << *ci << endl;
    out.close();
}
}
if (strcmp("ANALYZE", argv[1], 7) == 0) {
    //cout << "usage: statistic ANALYZE H-File max_weight" << endl;
    ifstream in(argv[2]);
    if (!in.is_open()) {
        cout << "File reading error " << argv[2] << endl;
        return 0;
    }
    unsigned int max = atoi(argv[3]);
    vector<unsigned int> weights(max+1);
    string tmp;
    while (getline(in, tmp)) {
        BitVector test(tmp);
        weights[test.weight()]++;
    }

    in.close();

    cout << "File " << argv[2] << endl;
    cout << "weight\t#\tsum\n";
    unsigned int sum = 0;
    for (unsigned int i = 0; i < max+1; ++i) {
        sum += weights[i];
        if (weights[i] > 0)
            cout << i << "\t" << weights[i] << "\t" << sum << endl;
    }
}

```

```

    }
}

```

A.3 GISD

A.3.1 Header

```

*/
class GISD : public Attack {
public:
    /*
     * @param G: n x k generator matrix
     * @param j: maximum number of errors in the chosen Information
Window
     * @param t: maximum errors allowed in the ECC
     */
    GISD(const BitMatrix& _G, unsigned int _t, unsigned int _j)
        : Attack(_G), c(), t(_t), j(_j) {};
    ~GISD() {};

    /*
     * @param c: ciphertext  $c = m * G + e$ 
     *
     * @returns: plaintext m
     *
     * there is no real computation declared in here, just calling
algo()
     */
    virtual BitVector run(const BitVector& _c, unsigned int
stop_time = -1);

private:

    /*
     * randomly selects an information window and calls algo_step
     */
    BitVector algo(unsigned int t, unsigned int j, unsigned int
stop_time);

    /*

```

```

        * computed the inverse matrix G_k and tests if there are less
        than j errors in c_k
        */
        BitVector algo_step(const set<unsigned int>& cols, unsigned int
t, unsigned int j);

        BitVector c;
        unsigned int t, j;
};

#endif

```

A.3.2 Code

```

}

BitVector GISD::algo(unsigned int t, unsigned int j, unsigned int
stop_time)
{
    time_t tstart = time(0);
    unsigned int n = G.Rows();
    unsigned int k = G.Columns();
    IWcounter = 0;
    while (1) {
        ++IWcounter;

        //step 1: randomly choose information window I
        set<unsigned int> cols;
        while (cols.size() < n)
            cols.insert(rand()%k);

        try {
            return algo_step(cols, t, j);
        } catch (NotFoundException e) {};
        time_t tactual = time(0);
        if ((tactual-tstart) > stop_time) {
            throw NotFoundException();
        }
    }
}
}

```

```

BitVector GISD::algo_step(const set<unsigned int>& cols, unsigned
int t, unsigned int j)
{
    //step 2:  $Q_1 = G_I^{-1}$ 
    BitSquareMatrix G_I(G.SubMatrixByCols(cols));
    BitMatrix Q_1(0, 0);
    try {
        Q_1 = G_I.Inverse();
    } catch (GaussException e) {
        throw NotFoundException();
    }

    //step 3:  $q_2 = Q_1 * G$ 
    BitMatrix Q_2 = Q_1 * G;

    //step 4:  $z = c + c_I * Q_2$ 
    BitVector c_I = c.SubVector(cols);
    BitVector z = c + c_I * Q_2;

    unsigned int n = Q_1.Rows();
    vector<bool> e(n); //all error vectors with  $|e| < j$ 
    for (unsigned int i = 0; i <= j; ++i) {
        for (unsigned int r = 1; r <= i; ++r)
            e[n-r] = 1;
        do {
            BitVector ve(e);

            //step 5: test if  $|z+e*Q_2| < t$ 
            if ((z + ve * Q_2).weight() <= t) {
                //step 6: return the result
                return (c_I + ve)*Q_1;
            }
        } while (next_permutation(e.begin(), e.end()));
    }
    throw NotFoundException();
}

```

A.4 SternGS

A.4.1 Header

```

*/
class SternGS : public Attack {
public:
    /*
     * @param G: n x k generator matrix, if (n > k) G gets
transposed
     * param t: maximum errors allowed in ECC
     * param p: linear combinations of p rows are taken
     * param l: l bits are compared for equality
     */
    SternGS(const BitMatrix& _G, unsigned int _t, unsigned int _p,
unsigned int _l);
    ~SternGS() {};

    /*
     * @param c: ciphertext,  $c = m \cdot G + e$ 
     *
     * @returns plain text m
     */
    virtual BitVector run(const BitVector& _c, unsigned int
stop_time = -1);

    /*
     * returns the shortest code word produced by G
     */
    BitVector find_shortest(unsigned int t, unsigned int p, unsigned
int l);
private:
    /*
     * implementation of the Stern_GS attack
     * @param c: ciphertext
     * @param t: maximum errors correctable by ECC system
     * @param p: use linear combinations of p rows
     * @param l:  $|L| = l$ 
     */
    BitVector algo(const BitVector& _c, unsigned int t, unsigned int
p, unsigned int l);

    BitMatrix original_G;

```

```

    unsigned int t, p, l;
    set<unsigned int> final_I;
};

#endif

```

A.4.2 Code

```

BitVector SternGS::run(const BitVector& _c, unsigned int stop_time)
{
    // generate  $G' = (G \mid c)^T$ 
    G = original_G;
    G.appendRow(_c);
    // start the real work
    BitVector computed_m = algo(_c, t, p, l);

    return computed_m;
}

BitVector SternGS::algo(const BitVector& _c, unsigned int t,
unsigned int p, unsigned int l)
{
    BitVector error = find_shortest(t, p, l);
    BitVector cc = _c + error;

    GISD last(original_G, t, 2);
    BitVector result = last.run(cc); //original_G, cc, t, 2);

    return result;
}

BitVector SternGS::find_shortest(unsigned int t, unsigned int p,
unsigned int l)
{
    unsigned int k = G.Rows();
    unsigned int n = G.Columns();
    unsigned int n_i1 = k >> 1;

    //step 1: randomly choose starting I
    //
    //Gauss without Col replace

```

```

/*
 * i dont swap the cols and remember the swappings, to undo them
 in the end
 * instead i leave everything where it was and get a spreaded
 identity matrix in G
 */
set<unsigned int> I; // contains the indices in G which belong to
the identity matrix
vector<unsigned int> IDCols(n, n); // contains the ordering of
identity entries
for (unsigned int i = 0; i < k; ++i) {
    unsigned int next;
    do {
        next = rand()%n;
    } while (G[i].toBool(next) == 0);
    I.insert(next);
    IDCols[next] = i;
    G.ZeroColBelowRow(i, next);
    G.ZeroColAboveRow(i, next);
}

IWcounter = 1;

set<unsigned int> Zindex; //contains the indices of G cols that
are not part of the identity
{
    vector<unsigned int> N(n);
    for (unsigned int i = 0; i < n; ++i)
        N[i] = i;
    set_difference(N.begin(), N.end(),
        I.begin(), I.end(),
        insert_iterator<set<unsigned int> >(Zindex, Zindex.begin()));
}

while(1) {
    IWcounter++;

    //step 2: split I equally in two parts I_1 and I_2
    set<unsigned int> I_1, I_2;

    while (I_1.size() < n_i1) {
        set<unsigned int>::iterator i = I.find(rand()%n);
        if (i != I.end())

```

```

        I_1.insert(*i);
    }
    set_difference(I.begin(), I.end(),
        I_1.begin(), I_1.end(),
        insert_iterator<set<unsigned int> >(I_2, I_2.begin()));

    set<unsigned int> I_1Rows;
    for (set<unsigned int>::const_iterator i = I_1.begin(); i !=
I_1.end(); ++i)
        I_1Rows.insert(IDCols[*i]);

    //compute a new matrix, containing only rows in I_1 and cols in
Z
    //(i.e. no identity col in here)
    //and then compute and store all linear combinatins of l rows
    BitMatrix Z_i = G.SubMatrix(I_1Rows, Zindex);

    //step 3: choose l-element set L
    set<unsigned int> L;
    while (L.size() < l) {
        unsigned int s = Z_i.Columns();
        L.insert(rand()%s);
    }

    //step 4: compute all linear combinations in Z_{I_1} and store
them in a multimap
    //Z1
    typedef pair<BitVector, set<unsigned int> > DATA;
    typedef multimap<string, DATA > MAP;
    vector<bool> e(n_i1);
    for (unsigned int r = 1; r <= p; ++r)
        e[n_i1-r] = 1;
    MAP linkombis;
    do {
        BitVector bv_e(e);
        BitVector lambda = bv_e*Z_i;
        BitVector lambda_L = lambda.SubVector(L);
        DATA data(lambda, bv_e.Items(1));
        linkombis.insert(pair<string, DATA>(lambda_L.toString(),
data));
    } while (next_permutation(e.begin(), e.end()));

    //step 5: compute all linear combinations in Z_{I_2}

```

```

/*
 * now do the same thing with the rows in I_2
 * find out if a linear combination has been computed with I_1
rows
 */
//Z2
unsigned int n_i2 = k - n_i1;
set<unsigned int> I_2Rows;
for (set<unsigned int>::const_iterator i = I_2.begin(); i !=
I_2.end(); ++i)
    I_2Rows.insert(IDCols[*i]);
Z_i = G.SubMatrix(I_2Rows, Zindex);

e = vector<bool>(n_i2);
for (unsigned int r = 1; r <= p; ++r)
    e[n_i2-r] = 1;
do {
    //step 6: search the linear combinations of Z_{I_1} for a
L-equal one
    BitVector bv_e(e);
    BitVector lambda = bv_e*Z_i;// this is the linear combination
    BitVector lambda_L = lambda.SubVector(L);
    pair<MAP::iterator, MAP::iterator> its =
linkombis.equal_range(lambda_L.toString());
    for (MAP::iterator i = its.first; i != its.second; ++i) {
        //step 7: check if weight of lamda_1 + lambda_2 is <= t-2p
        BitVector test = i->second.first;
        test += lambda;
        if (test.weight() <= t - 2 *p) { // we have a match
            //step 8: calculate and return e
            set<unsigned int> i2set = bv_e.Items();

            BitVector shortest(G.Columns());//the result

            unsigned int cnt = 0;
            set<unsigned int>::iterator i_1, si;
            i_1 = I_1Rows.begin();
            for (si = i->second.second.begin(); si !=
i->second.second.end(); ++si) {
                while (cnt < *si) {
                    cnt++;
                    i_1++;
                }
            }
        }
    }
}

```

```

    }

    shortestest += G[*i_1];
}
cnt = 0;
i_1 = I_2Rows.begin();
for (si = i2set.begin(); si != i2set.end(); ++si) {
    while (cnt < *si) {
        cnt++;
        i_1++;
    }

    shortestest += G[*i_1];
}

for (set<unsigned int>::iterator i = I.begin(); i !=
I.end(); ++i)
    if (IDCols[*i] < G.Rows() - 1)
        final_I.insert(*i);
return shortestest; //victory, leave the loop
}

}
} while (next_permutation(e.begin(), e.end()));

//step 9: switch to a close information set and retry

//compute a new information set by swaping a col p in I with a
col q in Z
//p and q need to have a 1 on the same row
set<unsigned int>::iterator p;
set<unsigned int>::iterator q;
do {
    p = I.begin();
    for(int i = rand()%k; i > 0; --i) {
        p++;
        if (p == I.end()) {
            cout << "Oha PPP";
            exit(1);
        }
    }
}
q = Zindex.begin();
for(int i = rand()%(n-k); i > 0; --i) {

```

```

        if (q == Zindex.end()) {
            cout << "Oha QQQ";
            exit(1);
        }
        q++;
    }
} while (G[IDCols[*p]].toBool(*q) == 0);

//now update the data structures, create a new spreaded
identity,
IDCols[*q] = IDCols[*p];
IDCols[*p] = n;
unsigned int tmp = *p;
I.erase(p);
I.insert(*q);
unsigned int q2 = *q;
Zindex.erase(q);
Zindex.insert(tmp);

G.ZeroColBelowRow(IDCols[q2], q2);
G.ZeroColAboveRow(IDCols[q2], q2);
}
}

```

A.5 BitProxy

A.5.1 Header

```

class BitProxy {
public:
    /*
     * _data and _index represent the mapping of the internal data
     structure of a BitVector.
     * If this data structure changes, BitProxy needs a new
     constructor to keep up.
     * _mask represents the single bit, which should be referenced
     */
    BitProxy(vector<unsigned long>& _data, unsigned int _index,
unsigned long _mask)
        : data(_data), index(_index), mask(_mask) { inverse_mask =
~mask; };

```

```

    BitProxy& operator=(const bool other) { if (other) data[index]
|= mask;
        else data[index] &= inverse_mask;
        return *this; };
    ~BitProxy() {};
    operator bool() const { return (data[index] & mask) > 0; };
private:
    vector<unsigned long>& data;
    unsigned int index;
    unsigned long mask;
    unsigned long inverse_mask;

};

#endif

```

A.6 BitVector

A.6.1 Header

```

* Make sure to use InitStatics() if you need to (if your machine
isn't a 32bit one).
*
* @author Markus Peter
*/
class BitVector {
public:
    BitVector() : size(0), BigData(0) {};
    BitVector(unsigned int _size);
    BitVector(const vector<bool>& _data);
    BitVector(const BitVector& other);
    BitVector(const string& other);
    ~BitVector() {};
    /*
    * per default, an unsigned long holds 32 bits.
    * if this is not true for your machine, you have to run this
function,
    * so the number of bits an unsigned long contains is adopted.
    * its safe to run this on 32 bit unsigned long machines: it has
no effect
    */
}

```

```

static void InitStatics(void) {
    unsigned int s = sizeof(unsigned long);
    unsigned int result = 2;
    while (s > 0) {
        s = s >> 1;
        result++;
    }
    BigDataPotenz = result;
    BigDataBits = 1 << BigDataPotenz;
};

bool operator==(const BitVector& other) const;
bool operator!=(const BitVector& other) const { return
!(operator==(other)); };

/*
 * calculates the scalar product <*this, other>
 */
bool operator*(const BitVector& other) const;

/*
 * calculates the element wise mult of the two vectors
 */
BitVector mult(const BitVector& other) const;
BitVector operator+(const BitVector& other) const;
BitVector& operator=(const BitVector& other);
BitVector& operator+=(const BitVector& other);
BitProxy operator[](unsigned int index);

bool operator<(const BitVector& other) const;

/*
 * this is the const version of operator[]
 * to avoid misunderstandings i used the name toBool
 * there is no reference returned, so you cant cast away
constness
 */
bool toBool(unsigned int index) const;

/*
 * returns a string representation of the vector
 */
string toString() const;

```

```

    /*
     * returns the number of bits stored in this vector
     */
    unsigned int length(void) const { return size; };

    /*
     * returns a new vector, containing the bits indicated by cols
     */
    BitVector SubVector(const set<unsigned int>& cols) const;

    /*
     * returns all indizes of bits containing value
     */
    set<unsigned int> Items(bool value = 1) const;

    /*
     * returns the hammering distance of this vector
     */
    unsigned int weight(void) const;
private:
    /*
     * make sure to update BitProxy if you change BigData type
     */
    unsigned int size; //nr of bits
    vector<unsigned long> BigData; //if size % sizeof(long) != 0
upper bits arent used
    static unsigned int BigDataPotenz; // 2^bigdatapotenz =
bigdatabits
    static unsigned int BigDataBits; //bits per unsigned long,
system dependant
};
ostream& operator<<(ostream& output, const BitVector& v);

#endif

```

A.6.2 Code

```

    BigData.resize(s);
};

```

```

BitVector::BitVector(const BitVector& other)
{
    size = other.size;
    BigData = other.BigData;
}

BitVector::BitVector(const vector<bool>& _data) : size(_data.size())
{
    unsigned int s = size >> BigDataPotenz;
    if (size % BigDataBits > 0)
        s++;
    BigData.resize(s);
    for(unsigned int i = 0; i < size; ++i)
        if (_data[i] == 1)
            BigData[i >> BigDataPotenz] |= (1 << (i % BigDataBits));
}

BitVector::BitVector(const string& other) : size(other.length())
{
    unsigned int s = size >> BigDataPotenz;
    if (size % BigDataBits > 0)
        s++;
    BigData.resize(s);
    for(unsigned int i = 0; i < size; ++i)
        if (other[i] == '1')
            BigData[i >> BigDataPotenz] |= (1 << (i % BigDataBits));
}

BitVector BitVector::operator+(const BitVector& other) const
{
    if (size != other.size)
        throw SizeMismatchException();
    BitVector result(size);
    for (unsigned int i = 0; i < BigData.size(); ++i)
        result.BigData[i] = BigData[i] ^ other.BigData[i];
    return result;
}

bool BitVector::operator==(const BitVector& other) const
{
    return ((size == other.size) && (BigData == other.BigData));
}

```

```

BitVector& BitVector::operator=(const BitVector& other)
{
    if (this == &other)
        return *this;
    size = other.size;
    BigData = other.BigData;
    return *this;
}

```

```

BitVector& BitVector::operator+=(const BitVector& other)
{
    if (size != other.size)
        throw SizeMismatchException();
    for (unsigned int i = 0; i < BigData.size(); ++i)
        BigData[i] ^= other.BigData[i];
    return *this;
}

```

```

bool BitVector::operator*(const BitVector& other) const
{
    if (size != other.size)
        throw SizeMismatchException();
    unsigned int count = 0;
    for (unsigned int i = 0; i < BigData.size(); ++i) {
        unsigned long t = BigData[i] & other.BigData[i];
        //count 1's in t
        unsigned long t2 = t;
        do {
            t = t2;
            t2 = t << 1;
            if (t2 < t)
                count++;
        } while (t2 != t);
    }
    return (count % 2);
}

```

```

BitVector BitVector::mult(const BitVector& other) const
{
    BitVector res(length());
    for (unsigned int i = 0; i < length(); ++i)
        res[i] = toBool(i)*other.toBool(i);
    return res;
}

```

```

}

BitProxy BitVector::operator[](unsigned int index)
{
    if (index < 0 || index >= size)
        throw IndexOutOfBoundsException();
    return BitProxy(BigData, index >> BigDataPotenz, 1 << (index %
BigDataBits));
}

bool BitVector::toBool(unsigned int index) const
{
    unsigned long mask = 1 << (index % BigDataBits);
    return ((BigData[index >> BigDataPotenz] & mask) > 0);
}

bool BitVector::operator<(const BitVector& other) const
{
    if (this->length() != other.length()) {
        return (length() < other.length());
    }
    for (unsigned int i = 0; i < length(); ++i) {
        if (toBool(i) != other.toBool(i))
            return (toBool(i) < other.toBool(i));
    }
    return false;
}

BitVector BitVector::SubVector(const set<unsigned int>& cols) const
{
    BitVector result(cols.size());
    unsigned int count=0;
    for (set<unsigned int>::const_iterator i = cols.begin(); i !=
cols.end(); ++i)
        result[count++] = toBool(*i);
    return result;
}

set<unsigned int> BitVector::Items(bool value) const
{
    set<unsigned int> result;
    for(unsigned int i = 0; i < size; ++i) {

```

```

        if (toBool(i) == 1)
            result.insert(i);
    }
    return result;
}

string BitVector::toString() const
{
    string result;
    for(unsigned int i = 0; i < BigData.size(); ++i) {
        unsigned long value = BigData[i];
        int j = 0;
        for (unsigned long mask = 1; mask != 0; mask = mask << 1, ++j)
        {
            if (i*BigDataBits+j >=size) return result;
            result.append(((value & mask) > 0) ? "1" : "0");
        }
    }
    return result;
}

unsigned int BitVector::weight(void) const
{
    unsigned int result = 0;
    for (unsigned int i = 0; i < BigData.size(); ++i) {
        unsigned long t = BigData[i];
        unsigned long t2 = t;
        do {
            t = t2;
            t2 = t << 1;
            if (t2 < t)
                result++;
        } while (t2 != t);
    }
    return result;
}

ostream& operator<<(ostream& output, const BitVector& v)
{
    output << v.toString();
    return output;
}

```

A.7 BitMatrix

A.7.1 Header

```

*/
class BitMatrix {
public:
    BitMatrix(void) : rows(0), cols(0), data(0) {};
    BitMatrix(unsigned int _rows, unsigned int _cols);

    /*
     * this constructor has been used for testing purposes only.
     * could be removed in final version to keep the interface
small.
     */
    BitMatrix(const vector<BitVector>& other);

    BitMatrix(const BitMatrix& other) : rows(other.rows),
cols(other.cols), data(other.data) {};

    /*
     * reads the McEliece Public Key data from the stream and
     * returns the maximum error correction capability t
     * the file should look like this:
     * *n*
     * 16
     * *k*
     * 8
     * *t*
     * 2
     * *fieldPoly*
     * 11001
     * *PKFileGPub*
     * 0100010110000101
     * 1010010001100100
     * 0101000010100001
     * 1010000010100010
     * 0110110000100001
     * 1010000010001101
     * 0110000000110110

```

```

    * 1100011000100000
    */
    unsigned int ReadFile(ifstream& in);
    BitMatrix SubMatrixByCols(const set<unsigned int>& columns)
const;
    BitMatrix SubMatrixByRows(const set<unsigned int>& rows) const;

    /*
    * combines SubMatrixByCols and subMatrixByRows
    */
    BitMatrix SubMatrix(const set<unsigned int>& rows, const
set<unsigned int>& columns) const;
    virtual ~BitMatrix() {};

    bool operator==(const BitMatrix& other);
    BitMatrix& operator=(const BitMatrix& other);
    const BitVector& operator[](const unsigned int& index) const;
    BitVector& operator[](const unsigned int& index);
    BitMatrix operator*(const BitMatrix& other) const;
    friend BitVector operator*(const BitMatrix& m, const BitVector&
v);
    friend BitVector operator*(const BitVector& v, const BitMatrix&
m);

    /*
    * Appends the Matrix by another row (at bottom)
    */
    virtual void appendRow(const BitVector& r);

    /*
    * adds matrix row "row" to all rows with higher index as "row"
(i.e. below "row")
    * that have a value of "1" in col "col"
    */
    void ZeroColBelowRow(unsigned int row, unsigned int col);

    /*
    * adds row "row" to all rows with smaller index as row with
value 1 in col "col"
    */
    void ZeroColAboveRow(unsigned int row, unsigned int col);

    /*

```

```
    * swaps the contents of rows a and b
    */
void swapColumns(unsigned int a, unsigned int b);

/*
 * returns total number of columns
 */
unsigned int Columns(void) const { return cols; };

/*
 * returns total number of rows
 */
unsigned int Rows(void) const { return rows; };

/*
 * returns column col
 */
BitVector Column(unsigned int col) const;

/*
 * apply the gaussian elimination algorithm
 */
void Gauss(void);

/*
 * gauss algo
 * stores column swaps in the returned matrix
 */
BitMatrix GaussP(void);

/*
 * creates a matrix, with 1s on the diagonal and 0s everywhere
else
 */
void Identity(void);

/*
 * get the transposed matrix
 */
BitMatrix Transposed(void) const;

/*
 * used in statistical decoding
```

```

        * takes ordered col indices and builds an invertible submatrix
of size k
        */
        set<unsigned int> invertibleIW(const multimap<unsigned int,
unsigned int>& cols,
            unsigned int k, bool start_front = 1);

        friend ostream& operator<<(ostream& output, const BitMatrix& m);
protected:

        /*
        * creates a 1 in position row, col
        * returns row if nothing had to be done,
        * otherwise the return value is the index of the row added to
row row
        */
        unsigned int Pivot(unsigned int row, unsigned int col);
        unsigned int rows, cols;
        vector<BitVector> data; //1st index is row , 2nd is col
    };

#endif

```

A.7.2 Code

```

{
}

/*
* creates a n x k Marix
* file format is presented in the header file
* */
unsigned int BitMatrix::ReadFile(istream& in)
{
    string tmp;
    getline(in, tmp); // "**n**"
    getline(in, tmp);
    istringstream ss(tmp);
    ss >> cols; // n
    getline(in, tmp); // "**k**"
    getline(in, tmp);

```

```

    ss.clear();
    ss.str(tmp);
    ss >> rows; // k
    getline(in, tmp); // ***t**
    getline(in, tmp); // t
    ss.clear();
    ss.str(tmp);
    int t;
    ss >> t;
    getline(in, tmp); // ***fieldpoly**
    getline(in, tmp); // some bitvector
    getline(in, tmp); // ***PKFileGPub**
    // and finally the matrix data, one row per line
    data = vector<BitVector>(rows, BitVector(cols));
    for (unsigned int j = 0; j < rows; ++j) {
        getline(in, tmp);
        ss.clear();
        ss.str(tmp);
        for (unsigned int i = 0; i < cols; ++i) {
            char b;
            ss >> b;
            if (b == '1')
                data[j][i] = 1;
        }
    }
    return t;
}

void BitMatrix::Identity(void)
{
    BitVector tmp(cols);
    unsigned int min = (rows < cols) ? rows : cols;
    for (unsigned int i = 0; i < min; ++i) {
        tmp[i] = 1;
        data[i] = tmp;
        tmp[i] = 0;
    }
}

BitMatrix BitMatrix::Transposed(void) const
{
    BitMatrix r(0, rows);
    for (unsigned int i=0; i < cols; ++i) {

```

```

    r.appendRow(Column(i));
}
return r;
}

```

```

BitMatrix BitMatrix::SubMatrixByCols(const set<unsigned int>&
columns) const
{
    BitMatrix result(rows, columns.size());
    unsigned int count = 0;
    for(set<unsigned int>::const_iterator i = columns.begin(); i !=
columns.end(); ++i) {
        for(unsigned int j = 0; j < rows; ++j)
            result.data[j][count] = data[j].toBool(*i);
        count++;
    }
    return result;
}

```

```

BitMatrix BitMatrix::SubMatrixByRows(const set<unsigned int>& rows)
const
{
    BitMatrix result(rows.size(), cols);
    unsigned int count = 0;
    for(set<unsigned int>::const_iterator i = rows.begin(); i !=
rows.end(); ++i)
        result.data[count++] = data[*i];
    return result;
}

```

```

BitMatrix BitMatrix::SubMatrix(const set<unsigned int>& rows, const
set<unsigned int>& columns) const
{
    BitMatrix result(rows.size(), columns.size());
    unsigned int i = 0;
    for (set<unsigned int>::const_iterator r = rows.begin(); r !=
rows.end(); ++r) {
        unsigned int j = 0;
        for(set<unsigned int>::const_iterator c = columns.begin(); c !=
columns.end(); ++c) {
            result.data[i][j] = data[*r].toBool(*c);
            ++j;
        }
        ++i;
    }
}

```

```
        ++i;
    }
    return result;
}

bool BitMatrix::operator==(const BitMatrix& other)
{
    if ((rows != other.rows) || (cols != other.cols))
        return false;
    for (unsigned int i = 0; i < rows; ++i)
        if (data[i] != other.data[i])
            return false;
    return true;
}

BitMatrix& BitMatrix::operator=(const BitMatrix& other)
{
    if (this == &other)
        return *this;
    rows = other.rows;
    cols = other.cols;
    data = other.data;
    return *this;
}

const BitVector& BitMatrix::operator[](const unsigned int& index)
const
{
    if (index < 0 || index >= rows)
        throw SizeMismatchException();
    return data[index];
}

BitVector& BitMatrix::operator[](const unsigned int& index)
{
    if (index < 0 || index >= rows)
        throw SizeMismatchException();
    return data[index];
}

BitMatrix BitMatrix::operator*(const BitMatrix& other) const
{
    if (cols != other.rows)
```

```

        throw SizeMismatchException();
    BitMatrix result(rows, other.cols);
    for (unsigned int i = 0; i < result.rows; ++i)
        for (unsigned int j = 0; j < result.cols; ++j)
            result.data[i][j] = data[i]*other.Column(j);
    return result;
}

BitVector operator*(const BitMatrix& m, const BitVector& v)
{
    if (m.Columns() != v.length())
        throw SizeMismatchException();
    BitVector result(m.Rows());
    for(unsigned int i = 0; i < m.Rows(); ++i)
        result[i] = m.data[i]*v;
    return result;
}

BitVector operator*(const BitVector& v, const BitMatrix& m)
{
    if (m.Rows() != v.length())
        throw SizeMismatchException();
    BitVector result(m.Columns());
    for(unsigned int i = 0; i < m.Columns(); ++i)
        result[i] = v*m.Column(i);
    return result;
}

void BitMatrix::appendRow(const BitVector& r)
{
    if (r.length() != cols)
        throw SizeMismatchException();
    data.push_back(r);
    rows++;
}

void BitMatrix::swapColumns(unsigned int a, unsigned int b)
{
    for (unsigned int i = 0; i < rows; ++i) {
        bool tmp = data[i].toBool(a);
        data[i][a] = data[i].toBool(b);
        data[i][b] = tmp;
    }
}

```

```

}

BitVector BitMatrix::Column(unsigned int col) const
{
    if (col >= cols)
        throw IndexOutOfBoundsException();
    BitVector tmp(rows);
    for (unsigned int i = 0; i < rows; ++i)
        tmp[i] = data[i].toBool(col);
    return tmp;
}

void BitMatrix::Gauss(void)
{
    for(unsigned int i = 0; i < rows; ++i) {
        unsigned int j;
        bool done;
        do {
            //find the next position in this col containing a 1
            try {
                done = true;
                j = Pivot(i, i);
            } catch (GaussException e) {//if none is found swap cols
                for (unsigned int i = e.row + 1; i < cols; ++i)
                    if (data[e.row][i] == 1) {
                        swapColumns(e.row, i);

                        done = false;
                        break;
                    }
                if (done) {
                    done = false;
                    throw;
                }
            }
        } while (!done);
        // if (i != j)
        // --j;
        ZeroColBelowRow(i, i);
    }
}

BitMatrix BitMatrix::GaussP(void)

```

```

{
  BitMatrix P(cols, cols);
  P.Identity();

  for(unsigned int i = 0; i < rows; ++i) {
    unsigned int j;
    bool done;
    do {
      //find the next position in this col containing a 1
      try {
        done = true;
        j = Pivot(i, i);
      } catch (GaussException e) {//if none is found swap cols
        for (unsigned int i = e.row + 1; i < cols; ++i)
          if (data[e.row][i] == 1) {
            swapColumns(e.row, i);
            BitMatrix P_n(cols, cols);
            P_n.Identity();
            P_n.swapColumns(e.row, i);
            P = P * P_n;
            done = false;
            break;
          }
        if (done) {
          done = false;
          throw;
        }
      }
    } while (!done);
    ZeroColBelowRow(i, i);
  }
  return P;
}

void BitMatrix::ZeroColBelowRow(unsigned int row, unsigned int col)
{
  for (unsigned int i = row + 1; i < rows; ++i)
    if (data[i][col] == 1)
      data[i] += data[row];
}

void BitMatrix::ZeroColAboveRow(unsigned int row, unsigned int col)

```

```

{
  for (unsigned int i = 0; i < row; ++i)
    if (data[i][col] == 1)
      data[i] += data[row];
}

unsigned int BitMatrix::Pivot(unsigned int row, unsigned int col)
{
  //make sure row,col is 1
  if (data[row][col] == 1)
    return row;
  for (unsigned int i = row + 1; i < rows; ++i)
    if(data[i][col] == 1) {
      data[row] += data[i];
      return i;
    }
  throw GaussException(row);
}

set<unsigned int> BitMatrix::invertibleIW(const multimap<unsigned
int, unsigned int>& v,
  unsigned int k, bool start_front)
{
  typedef multimap<unsigned int, unsigned int> ORDERMAP;
  set<unsigned int> iw;
  ORDERMAP::const_iterator f = v.begin();
  ORDERMAP::const_reverse_iterator r = v.rbegin();
  //start set
  while (iw.size() < k) {
    if (start_front) {
      iw.insert(f->second);
      ++f;
    } else {
      iw.insert(r->second);
      ++r;
    }
  }
}

bool done = true;
do {
  BitMatrix result = SubMatrixByCols(iw);
  for(unsigned int i = 0; i < k; ++i) {
    unsigned int j;

```

```

//find the next position in this col containing a 1
try {
    done = true;
    j = result.Pivot(i, i);
} catch (GaussException e) {//if none is found change iw
    set<unsigned int>::iterator del = iw.begin();
    for (unsigned int l = 0; l < e.row; ++l)
        del++;
    iw.erase(del);

    if (start_front) {
        if (f != v.end()) {
            iw.insert(f->second);
            ++f;
        } else throw;
    } else {
        if (r != v.rend()) {
            iw.insert(r->second);
            ++r;
        } else throw;
    }
    done = false;
    break;
}
result.ZeroColBelowRow(i, i);
} while (!done);

return iw;
}

ostream& operator<<(ostream& output, const BitMatrix& m)
{
    for(vector<BitVector>::const_iterator i=m.data.begin(); i <
m.data.end(); ++i)
        output << "(" << (*i) << ")\n";
    return output;
}

```

A.8 BitSquareMatrix

A.8.1 Header

```

~BitSquareMatrix() {};

/*
 * returns the inverse of the matrix or throws
 * a GaussException if there is none
 */
BitSquareMatrix Inverse(void) const;

private:
/*
 * if this function is called it would remove the square
probability,
 * so its hidden from the outer world.
 */
void appendRow(const BitVector& r) {};
};
#endif

```

A.8.2 Code

```

}

int next = i + 1;
if (add != i ) {
    result.data[i] += result.data[add];
    next = add;
}
for(unsigned int j = next; j < rows; ++j)
    if (tmp.data[j][i] == 1) {
        result.data[j] += result.data[i];
        tmp.data[j] += tmp.data[i];
    }
}

//zero upper triangle, from right to left, to avoid creation of
further 1s ;)
for(int i = cols - 1; i >= 0; --i) {
    for(int j = i-1; j >= 0; --j)

```

```
        if (tmp[j].toBool(i) == 1)
            result.data[j] += result.data[i];
    }
    return result;
}
```

A.9 Exceptions

```
};

class IndexOutOfBoundsException
{
    public:
        IndexOutOfBoundsException() {};
};

class NotFoundException
{
    public:
        NotFoundException() {};
};

#endif
```