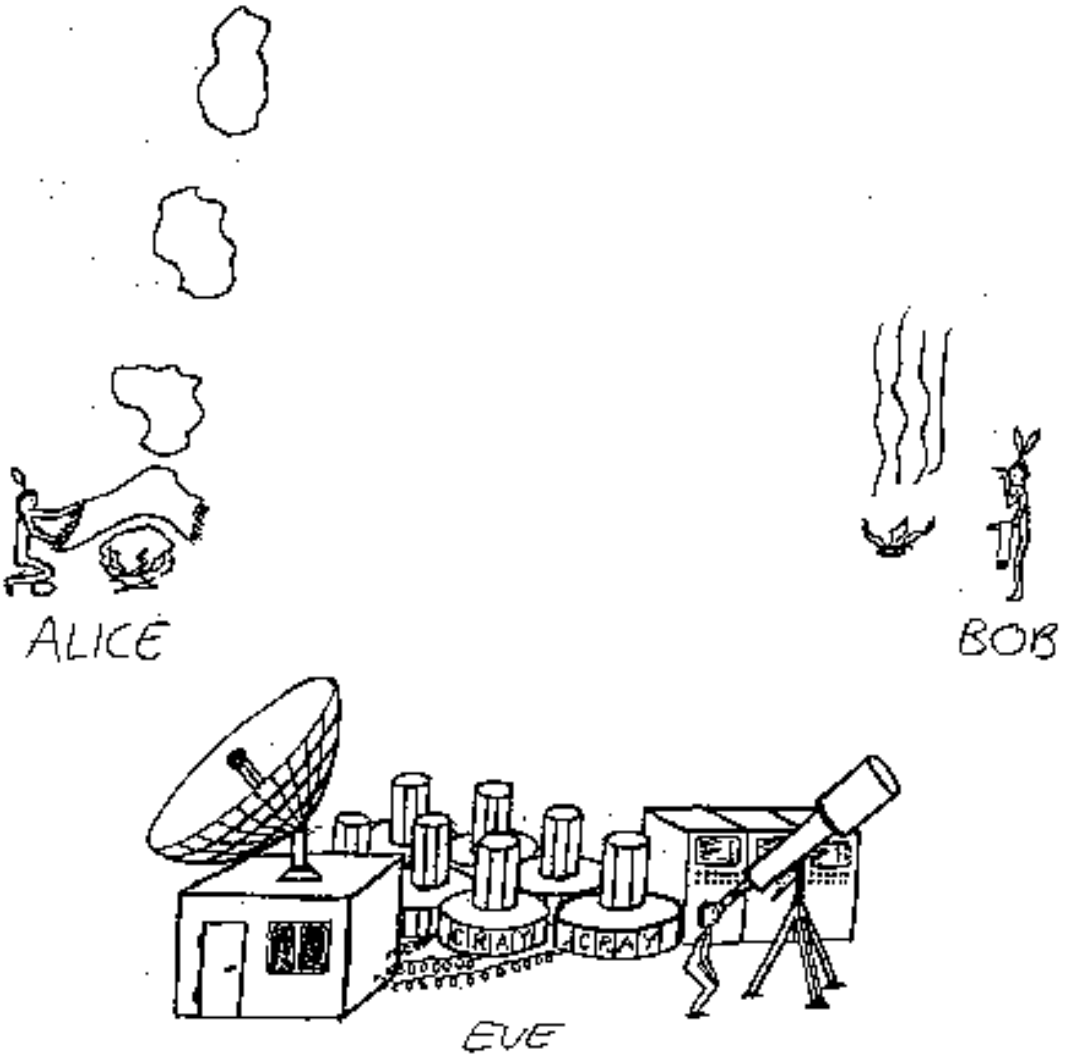
Analyse beweisbar sicherer Schlüsselgenerierungsprotokolle

Martin Gander
Institut für Theoretische Informatik
ETH Zürich
CH-8092 Zürich



6. April 1993

Inhaltsverzeichnis

1 Einleitung

Eines der fundamentalen Probleme in der Kryptographie ist die Übermittlung einer geheimen Nachricht M von einem Sender Alice zu einem Empfänger Bob über einen unsicheren Kanal, welcher von einem Feind Eve abgehört wird, ohne dass Eve brauchbare Information über M erhält.

Im klassischen Model von Shannon [6] erzeugt Alice aus der Nachricht M und dem Schlüssel K eine verschlüsselte Nachricht C. Weil Eve perfekten Zugriff auf den unsicheren Kanal hat, erhält sie eine identische Kopie der Nachricht C, die Bob bekommt. Shannon zeigte, dass perfekte Sicherheit in der Übertragung erreicht wird, wenn die verschlüsselte Nachricht C statistisch unabhängig ist von der ursprünglichen Nachricht M, d.h. wenn die Mutual Information $I(M;C) = 0$. Dies wird zum Beispiel mit dem von Shannon eingeführten *"one-time pad"* erreicht, indem der geheime Schlüssel K einfach mindestens so lang sein muss wie die Nachricht M, also die Entropie $H(K) >= H(M)$. Leider ist meistens nur ein kurzer geheimer Schlüssel vorhanden, sodass die Lösung in der Praxis unbrauchbar ist.

Eine Arbeit [1] von Maurer zeigt, dass die Annahme von Shannon, dass der Feind Eve eine identische Kopie der verschlüsselten Nachricht C bekommt wie Bob, in der Praxis zu restriktiv ist. Vielmehr muss man stets damit rechnen, dass ein Übertragungskanal mit Fehlern behaftet ist. Diese werden zwar im Normalfall mit fehlerkorrigierenden Codes [5] behoben, weil sie nicht erwünscht sind. Sie können aber, wie in [1] gezeigt wird, dazu verwendet werden, einen beliebig langen, beweisbar sicheren Schlüssel K über den unsicheren Kanal aufzubauen. Das Verfahren funktioniert selbst dann, wenn Eve einen besseren, also weniger fehlerbehafteten Kanal besitzt als Bob. Dabei wird vorausgesetzt, dass Eve keine der abgehörten Nachrichten ändern, oder falsche Nachrichten in den Kanal einspeisen kann. Falls diese Annahmen in einem System in der Realität nicht garantiert werden können, so müssen sie durch ein beweisbar sicheres Authentifikationsprotokoll sichergestellt werden, wie es zum Beispiel in [7] beschrieben wird.

Die folgende Arbeit untersucht Protokolle, welche das Erzeugen eines beliebig langen, beweisbar sicheren Schlüssels K über einen vom Feind abgehörten Kanal erlauben. Es wird eine numerische Methode hergeleitet, mit der die Bitfehlerwahrscheinlichkeit von Eve auf den Bits des Schüssels K für diese Protokolle berechnet werden kann. Weiter wird theoretisch die optimale Strategie von Eve zum Dekodieren des Schlüssels gezeigt, aus der sich eine geschlossene Formel für ihre Bitfehlerwahrscheinlichkeit berechnen lässt.

2 Modelle

In diesem Kapitel werden zwei Modelle eines Übertragungskanals vorgestellt, die in allen folgenden Berechnungen verwendet werden. Es wird gezeigt, dass man die beiden Modelle ineinander überführen kann, und es werden Formeln dazu hergeleitet.

2.1 Ein verrauschter Kanal

Ein Übertragungskanal wird in diesem Modell als eine Black Box betrachtet, welche einen Eingang hat, an dem Alice, der Sender, angeschlossen ist, und zwei Ausgänge, mit welchen Bob und Eve verbunden sind, wie in Abbildung 1 dargestellt.

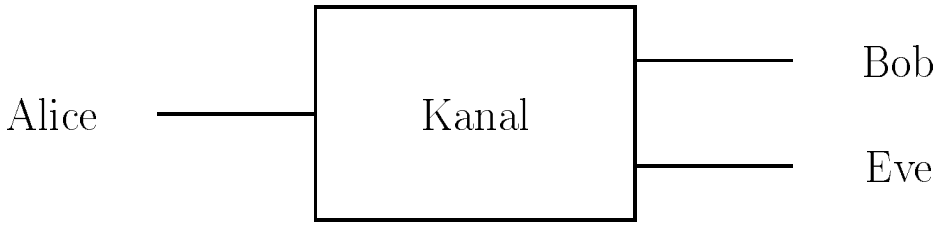


Abbildung 1: Ein verrauschter Kanal

Dieser Kanal wird durch die Übergangswahrscheinlichkeiten $\alpha_{i,j}$ beschrieben. $\alpha_{i,j}$ ist die bedingte Wahrscheinlichkeit, dass Bob ein Bit i und Eve ein Bit j empfängt mit $i, j \in \{0, 1\}$, wenn Alice ein Bit 0 gesendet hat. Der Kanal ist durch die vier Wahrscheinlichkeiten $\alpha_{0,0}$, $\alpha_{0,1}$, $\alpha_{1,0}$ und $\alpha_{1,1}$ eindeutig bestimmt. So wird zum Beispiel die Wahrscheinlichkeit, dass Bob ein Bit 1 empfängt, wenn Alice ein Bit 0 gesendet hat, gleich $\alpha_{1,0}+\alpha_{1,1}$. Weil der Kanal binär symmetrisch ist, genügen die vier Parameter für eine vollständige Beschreibung. Es ist sogar einer redundant, da sie zu 1 summieren.

2.2 Ein Satellit, welcher Zufallsbits sendet

Man stelle sich folgende Situation vor: Ein Satellit, der sich auf einer Umlaufbahn um die Erde befindet, sendet mit einer schwachen Sendeleistung zufällige Bits aus (vergleiche dazu Abbildung 2).

Auf der Erde befinden sich Alice, Bob und Eve, welche alle mit einer Antenne ausgerüstet sind, um diese Bits zu empfangen. Allerdings sind die gesendeten Bits so schwach, dass ein Empfang ohne Fehler nicht möglich ist. Die Bitfehlerwahrscheinlichkeiten sind ϵ_A für Alice, ϵ_B für Bob und ϵ_E für Eve. Weiter sind Alice und Bob mit einer Leitung verbunden, welche fehlerfreie Übertragung garantiert, aber von Eve auch ohne Fehler abgehört werden kann.

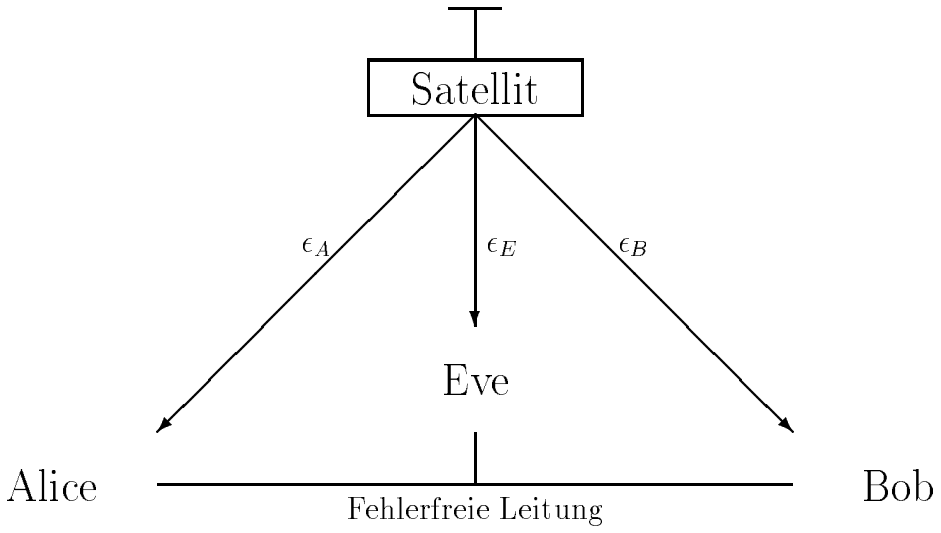


Abbildung 2: Das Satellitenmodell

Wir betrachten nun eine Nachricht M, welche Alice zu Bob schicken möchte. Um auf der fehlerfreien Leitung ein Rauschen zu simulieren, addiert Alice zu jedem Bit von M das eben empfangene Zufallsbit des Satelliten modulo 2, und Bob subtrahiert es wieder. Auch Eve subtrahiert nach dem Empfangen von M die jeweiligen Bits des Satelliten. Dadurch wird die Übertragung mit einem Fehler behaftet, welcher durch die Parameter ϵ_A, ϵ_B und ϵ_E bestimmt ist.

2.3 Umrechnung von einem Modell ins andere

Um eine Abbildung von einem Szenario ins andere zu finden, berechnen wir die Übergangswahrscheinlichkeiten $\alpha_{i,j}$ im Satellitenbild, vorausgesetzt, dass Alice eine 0 sendet. Dabei soll $\delta_A = 1 - \epsilon_A$, $\delta_B = 1 - \epsilon_B$ und $\delta_E = 1 - \epsilon_E$ sein. Man findet

$$\begin{aligned} \alpha_{0,0} &= \delta_A\delta_B\delta_E + \epsilon_A\epsilon_B\epsilon_E \\ \alpha_{0,1} &= \delta_A\delta_B\epsilon_E + \epsilon_A\epsilon_B\delta_E \\ \alpha_{1,0} &= \delta_A\epsilon_B\delta_E + \epsilon_A\delta_B\epsilon_E \\ \alpha_{1,1} &= \delta_A\epsilon_B\epsilon_E + \epsilon_A\delta_B\delta_E. \end{aligned} \tag{1}$$

Die Gleichungen sind leicht zu verstehen, denn wenn Alice eine 0 gesendet hat, dann haben zum Beispiel Bob und Eve genau dann beide eine 0, wenn alle drei das Bit vom Satelliten entweder mit oder ohne Fehler erhalten haben. Man kann somit anstelle des Satellitenmodells durch einfache Umrechnung auch das Modell des verrauschten Kanals verwenden. Die Rückrechnung ist etwas aufwendiger, aber nicht schwierig. Man löst das

Gleichungssystem 1 nach den Variablen ϵ_A, ϵ_B und ϵ_E auf und findet

$$\epsilon_E = \frac{1}{2} - \frac{1}{2}\sqrt{-2\frac{(\alpha_{0,1} + \alpha_{1,0} - \frac{1}{2})(\alpha_{0,1} + \alpha_{1,1} - \frac{1}{2})}{\alpha_{1,0} + \alpha_{1,1} - \frac{1}{2}}}$$

$$\epsilon_A = \frac{\alpha_{0,1} + \alpha_{1,1} - \epsilon_E}{1 - 2\epsilon_E} \tag{2}$$

$$\epsilon_B = \frac{\alpha_{0,1} + \alpha_{1,0} - \epsilon_E}{1 - 2\epsilon_E}$$

Falls $\epsilon_E = \frac{1}{2}$ ist, werden die Nenner von ϵ_A und ϵ_B in 2 null, also kann man die oberen Formeln nicht mehr verwenden. In der Tat legt das Gleichungssystem nur noch die Beziehung zwischen ϵ_A und ϵ_B fest. In diesem Fall kann man ϵ_A wählen und ϵ_B durch die folgende Gleichung bestimmen.

$$\epsilon_B = \frac{\alpha_{1,0} - \frac{1}{2}\epsilon_A}{\frac{1}{2} - \epsilon_A}$$

Für die folgenden Rechnungen ist es oft nützlich, nur die Fälle $\epsilon_A = \epsilon_B$ zu betrachten. In diesem Fall berechnet sich bei $\epsilon_E = \frac{1}{2}$ die Fehlerwahrscheinlichkeit ϵ_A zu

$$\epsilon_A = \frac{1 - \sqrt{1 - 4\alpha_{1,0}}}{2}.$$

3 Einfache Protokolle

In diesem Kapitel werden einfache Protokolle vorgestellt, bei denen keine Abhängigkeit zwischen den einzelnen Bits besteht.

3.1 Verkleinerung der Bitfehlerwahrscheinlichkeit

3.1.1 Repeatcodes

Der Repeatcode ist eine einfache Art, die Bitfehlerwahrscheinlichkeit auf einem verrauschten Kanal zu verkleinern. Jedes Bit wird nicht nur einmal, sondern n mal nacheinander über den Kanal gesendet. Wenn der Kanal binär symmetrisch ist mit der Bitfehlerwahrscheinlichkeit ϵ und der Empfänger ein Bit nur akzeptiert, wenn er n mal das gleiche Bit erhalten hat, so wird die Bitfehlerwahrscheinlichkeit $\epsilon^{'}$ im so neu entstandenen Kanal

$$\epsilon^{'} = \frac{\epsilon^n}{\epsilon^n + (1-\epsilon)^n}.$$

Sei zum Beispiel $\epsilon = 0.2$ und $n = 5$, dann wird $\epsilon^{'} = 0.00098$.

Betrachten wir nun das Satellitenmodell. Die Wahrscheinlichkeit, dass Bob und Alice das Bit vom Satelliten verschieden empfangen, ist

$$\epsilon = \delta_A \epsilon_B + \epsilon_A \delta_B, \tag{3}$$

mit $\delta_A = 1-\epsilon_A$ und $\delta_B = 1-\epsilon_B$ wie im Abschnitt 2.3. Wenn Alice nun einen Repeatcode der Länge n verwendet, so wird die Bitfehlerwahrscheinlichkeit β von Bob

$$\beta = \frac{\epsilon^n}{\epsilon^n + (1-\epsilon)^n}. \tag{4}$$

Die Wahrscheinlichkeit, dass Bob ein Bit akzeptiert, beträgt

$$p_{accept} = \epsilon^n + (1-\epsilon)^n. \tag{5}$$

Berechnet man mit Hilfe der $\alpha_{i,j}$ aus dem Abschnitt 2.3 die Wahrscheinlichkeiten p_ω, dass ein Bit 0, n mal von Alice gesendet und von Bob akzeptiert, von Eve als Bitsequenz mit ω Einern empfangen wurde, so erhält man

$$p_\omega = \alpha_{0,0}^{n-\omega}\alpha_{0,1}^{\omega} + \alpha_{1,0}^{n-\omega}\alpha_{1,1}^{\omega}.$$

Eve kann nun aufgrund der empfangenen Daten das Bit von Alice schätzen, indem sie einen Mehrheitsentscheid fällt: wenn sie mehr Nullen als Einer empfangen hat, so hat Alice wahrscheinlich eine Null gesendet, im anderen Fall eine Eins. Das enspricht für ungerade n einem binär-symmetrischen Kanal mit der Bitfehlerwahrscheinlichkeit

$$\gamma_{odd} = \frac{1}{p_{accept}} \sum_{\omega=\lceil n/2 \rceil}^{n} \binom{n}{\omega} p_\omega \tag{6}$$

mit p_{accept} aus Gleichung 5 und für gerade n

$$\gamma_{even} = \gamma_{odd} - \frac{1}{2}\frac{1}{p_{accept}} \left(\begin{array}{c} n \\ n/2 \end{array} \right) p_{n/2}, \tag{7}$$

weil in diesem Fall Eve bei gleich vielen Nullen wie Einern beim Raten nur die Hälfte im Durchschnitt falsch rät. Interessanterweise kann n immer so gewählt werden, dass $\gamma > \beta$, wie in [1] gezeigt wird, sodass also Eve stets auf ihren Bits eine grössere Bitfehlerwahrscheinlichkeit hat als Bob.

3.1.2 Paritycodes

Den Paritycode versteht man am besten im Satellitenmodell. Sobald Alice zwei zufällige Bits vom Satelliten empfangen hat, berechnet sie das Parity von diesem Zweierblock, also die Summe modulo 2, und sendet das Resultat zu Bob. Wenn Bob für seine zwei zufälligen Bits vom Satelliten das gleiche Parity berechnet, dann behalten Alice und Bob das linke der beiden Bits, sonst werden die zwei Bits weggeworfen und ein neuer Zweierblock abgewartet. Sei ϵ nach Gleichung 3 die Wahrscheinlichkeit, dass ein Bit vom Satelliten von Alice und Bob verschieden empfangen wurde. Dann wird die Bitfehlerwahrscheinlichkeit ϵ' zwischen Alice und Bob nach einem Parityprotokollschritt

$$\epsilon^{'} = \frac{\epsilon^2}{\epsilon^2 + (1-\epsilon)^2}. \tag{8}$$

Wenn man zum Beispiel $\epsilon_A = \epsilon_B = 0.2$ wählt, dann wird $\epsilon = 0.32$ und $\epsilon^{'} = 0.18$, also stimmen die Bits von Alice und Bob eher überein nach einem Parityprotokollschritt als vorher.

3.1.3 Vergleich von Parity- und Repeatcode

Nun kann man natürlich das Parityprotokoll mehrmals anwenden und es drängt sich ein Vergleich mit dem Repeatcode auf. Dabei kann man zwei Dinge vergleichen: Erstens die entstehenden Bitfehlerwahrscheinlichkeiten zwischen Alice und Bob sowie Alice und Eve; zweitens die Kapazität des neuen Kanals, definiert als Verhältnis Anzahl brauchbarer Bits zu Anzahl gesendeter Bits.

Zuerst zeige ich die Äquivalenz der Bitfehlerwahrscheinlichkeit zwischen Alice und Bob. Wenn man das Parityprotokoll einmal anwendet, so beträgt die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob $\epsilon^{'}$ gemäss Gleichung 8. Wendet man das Parityprotokoll erneut auf die mit dem ersten Parityschritt erhaltenen Bits an, so beträgt die Bitfehlerwahrscheinlichkeit $\epsilon^{''}$ zwischen Alice und Bob

$$\epsilon^{''} = \frac{\epsilon^{'2}}{\epsilon^{'2} + (1-\epsilon^{'})^2}.$$

Durch Einsetzen von $\epsilon^{'}$ erhält man

$$\epsilon'' = \frac{(\frac{\epsilon^2}{\epsilon^2+(1-\epsilon)^2})^2}{(\frac{\epsilon^2}{\epsilon^2+(1-\epsilon)^2})^2 + (1-\frac{\epsilon^2}{\epsilon^2+(1-\epsilon)^2})^2},$$

was nach dem Vereinfachen die Formel

$$\epsilon'' = \frac{\epsilon^4}{\epsilon^4 + (1-\epsilon)^4}$$

liefert. Durch Induktion findet man für die Bitfehlerwahrscheinlichkeit β zwischen Alice und Bob nach k Runden des Parityprotokolls

$$\beta = \frac{\epsilon^{2^k}}{\epsilon^{2^k} + (1-\epsilon)^{2^k}}. \tag{9}$$

Dies entspricht aber genau der Bitfehlerwahrscheinlichkeit zwischen Alice und Bob, nachdem ein Repeatcodeprotokoll der Länge $n = 2^k$ durchgeführt worden ist, wie ein Vergleich mit Gleichung 4 zeigt. Insbesondere ist das Parityprotokoll in bezug auf Alice und Bob äquivalent einem Repeatcode der Länge 2.

Nun zeige ich die Äquivalenz von Parity- und Repeatcode in bezug auf die Bitfehlerwahrscheinlichkeit zwischen Alice und Eve. Man muss sich dabei vor suboptimalen Strategien für Eve hüten. Ein abschreckendes Beispiel, dem ich zum Opfer fiel, möchte ich genauer erläutern. Wenn Alice und Bob das Parityprotokoll verwenden, so kann Eve folgende Strategie anwenden: immer, wenn die Parity bei Alice und Bob übereinstimmt und sie das linke der beiden Bits, die sie vom Satelliten empfangen haben, behalten, behält auch Eve das linke der beiden Bits, die sie vom Satelliten bekommen hat. Sei $\alpha_{i,j}$ wie im Abschnitt 2.3 definiert. Wenn Alice jetzt in einem Schritt des Parityprotokolls zwei Nullen hat und Bob den Parityschritt akzeptiert, dann kann man für die neue Situation die Wahrscheinlichkeit $p_{i,j}$, dass nach dem Parityschritt Bob das Bit i und Eve das Bit j hat, wiefolgt berechnen:

$$\begin{aligned} p_{0,0} &= \alpha_{0,0}^2 + \alpha_{0,0}\alpha_{0,1} \\ p_{0,1} &= \alpha_{0,1}^2 + \alpha_{0,1}\alpha_{0,0} \\ p_{1,0} &= \alpha_{1,0}^2 + \alpha_{1,0}\alpha_{1,1} \\ p_{1,1} &= \alpha_{1,1}^2 + \alpha_{1,1}\alpha_{1,0}. \end{aligned}$$

Normiert man diese Wahrscheinlichkeiten mit der Wahrscheinlichkeit, dass Bob akzeptiert, also mit

$$p_{accept} = (\alpha_{0,0} + \alpha_{0,1})^2 + (\alpha_{1,0} + \alpha_{1,1})^2,$$

so erhält man neue Werte $\alpha'_{i,j}$

$$
\begin{aligned}
\alpha'_{0,0} &= \frac{\alpha_{0,0}^2 + \alpha_{0,0}\alpha_{0,1}}{p_{accept}} \\
\alpha'_{0,1} &= \frac{\alpha_{0,1}^2 + \alpha_{0,0}\alpha_{0,1}}{p_{accept}} \\
\alpha'_{1,0} &= \frac{\alpha_{1,0}^2 + \alpha_{1,0}\alpha_{1,1}}{p_{accept}} \\
\alpha'_{1,1} &= \frac{\alpha_{1,1}^2 + \alpha_{1,0}\alpha_{1,1}}{p_{accept}}.
\end{aligned}
$$

Diese Formeln kann man rekursiv wieder auf sich anwenden und so die $\alpha_{i,j}$ für eine beliebige Anzahl Parity-Protokollschritte berechnen. Ja, es gelingt sogar, die simultane Rekursion aufzulösen und man findet für die $\alpha_{i,j}$ nach k Protokollschritten die explizite Form

$$
\begin{aligned}
\alpha_{0,0}^k &= \frac{\alpha_{0,0}(\alpha_{0,0} + \alpha_{0,1})^{2^k-1}}{p_{accept}} \\
\alpha_{0,1}^k &= \frac{\alpha_{0,1}(\alpha_{0,0} + \alpha_{0,1})^{2^k-1}}{p_{accept}} \\
\alpha_{1,0}^k &= \frac{\alpha_{1,0}(\alpha_{1,0} + \alpha_{1,1})^{2^k-1}}{p_{accept}} \\
\alpha_{1,1}^k &= \frac{\alpha_{1,1}(\alpha_{1,0} + \alpha_{1,1})^{2^k-1}}{p_{accept}}
\end{aligned}
$$

mit

$$
p_{accept} = (\alpha_{0,0} + \alpha_{0,1})^{2^k} + (\alpha_{1,0} + \alpha_{1,1})^{2^k}.
$$

Die Fehlerwahrscheinlichkeit von Eve beträgt mit dieser Strategie

$$
\gamma_k = \alpha_{0,1}^k + \alpha_{1,1}^k.
$$

Bildet man schliesslich den Grenzwert für k gegen Unendlich, so findet man, dass dieser existiert und folgende einfache Form hat:

$$
\gamma = \lim_{k \to \infty} \gamma_k = \frac{\alpha_{0,1}}{\alpha_{0,0} + \alpha_{0,1}}
$$

Man kann also zeigen, dass Eve bei wiederholtem Anwenden des Parityprotokolls mit dieser Strategie nie besser wird als eine bestimmte Grenze γ. Leider ist diese Strategie nicht optimal, Eve kann es besser tun ! Nämlich nicht sofort bei jedem Bit entscheiden, sondern

alle Paritybits, die über den Kanal gesendet wurden, speichern und erst am Schluss einen globalen Entscheid fällen. Aus den gespeicherten Paritybits lassen sich zwei komplementäre Bitstrings generieren, einer entspricht Bit 0 am Schluss, der andere Bit 1. Eve muss nun dasjenige Bit wählen, dessen String bei dem, den sie vom Satelliten empfangen hat, näher ist, oder genauer, zu dem der Hammingabstand kleiner ist.

Ein Beispiel dazu zeigt die Abbildung 3. In der obersten Zeile steht der Bitstring, welcher Eve vom Satelliten empfangen hat. In den folgenden Zeilen, welche mit *Parity* angeschrieben sind, stehen die Paritychecks, die Eve auf dem Kanal abhört. Wenn nun der letzte Paritycheck gemacht ist, so kann Eve unten mit der Konstruktion der beiden möglichen, inversen Bitstrings beginnen.

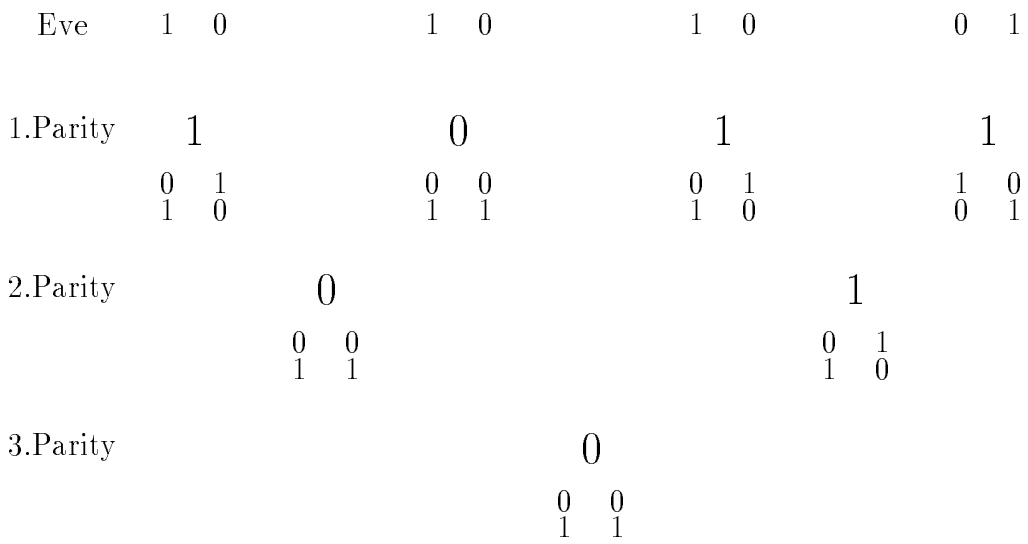


Abbildung 3: Optimale Strategie für Eve

Im Beispiel war der letzte Paritycheck 0, also könnten Alice und Bob entweder 00 (Fall Oben) oder 11 (Fall Unten) als Bitstring haben. Der vorletzte Check gab auf der linken Seite 0, also mussten Alice und Bob 00 oder 11 gehabt haben, und zwar 00 für den Fall Oben und 11 für den Fall Unten. Der vorletzte Check auf der rechten Seite ergab Parity 1, also mussten Alice und Bob entweder 01 für den Fall Oben oder 10 für den Fall Unten gehabt haben. So konstruiert nun Eve weiter nach oben, bis sie die zwei komplementären Bitstrings unter dem ersten Paritycheck gefunden hat. Nun berechnet sie den Hammingabstand von ihrem Bitstring zu diesen beiden und findet für den Fall Unten Abstand 1 und für den Fall Oben 7. Somit ist der Fall Unten wahrscheinlicher, also das Bit von Alice und Bob am Schluss wahrscheinlich 1.

Es ist einfach einzusehen, dass Eve mit dieser Strategie beim Paritycode genau die gleiche Fehlerwahrscheinlichkeit γ hat wie beim Repeatcode. Daraus folgt, dass Paritycode

und Repeatcode in bezug auf die Bitfehlerwahrscheinlichkeiten für Alice, Bob und Eve äquivalent sind.

Das Parityprotokoll hat aber doch einen Vorteil, den man schon intuitiv erahnen kann. Man stelle sich dazu einfach einen Repeatcode der Länge $n = 2^k$ vor gegenüber einem Paritycode, der k mal wiederholt wird. Die Wahrscheinlichkeit p_{repeat}, dass Bob einen Block der Länge $n = 2^k$ akzeptieren kann, beträgt bei einer Bitfehlerwahrscheinlichkeit ϵ

$$p_{repeat} = \epsilon^{2^k} + (1-\epsilon)^{2^k},$$

ist also äusserst gering. Beim Parityprotokoll werden hingegen einzelne Zweierblöcke mit der Wahrscheinlichkeit

$$p_{parity} = \epsilon^2 + (1-\epsilon)^2$$

akzeptiert, sodass Bob auch bei mehrmaligem Anwenden des Protokollschritts weniger Daten verwerfen muss.

Für eine exakte Berechnung des Vorteils führe ich den Begriff des Reduktionsfaktors ein. Der Reduktionsfaktor ist der Faktor, um welchen die Anzahl Bits in einem Bitstring reduziert wird bei der Anwendung eines bestimmten Protokolls. Der Reduktionsfaktor für das Repeatcodeprotokoll, R_{repeat} beträgt

$$R_{repeat} = \frac{\epsilon^{2^k} + (1-\epsilon)^{2^k}}{2^k}, \tag{10}$$

nämlich die Wahrscheinlichkeit, dass ein Block akzeptiert wird, mal den Bruchteil der Bits, die aus einem Block verwendet werden können, also hier geteilt durch die Blocklänge. Der Reduktionsfaktor beim Parityprotokoll beträgt für einen Schritt analog

$$R_{1parity} = \frac{\epsilon^2 + (1-\epsilon)^2}{2}.$$

Für zwei Schritte berechnet sich der Reduktionsfaktor als Produkt des Reduktionsfaktors aus dem ersten Schritt mal den Reduktionsfaktor des zweiten Schrittes, mit einer neuen Fehlerwahrscheinlichkeit $\epsilon^{'}$.

$$R_{2parity} = \frac{\epsilon^2 + (1-\epsilon)^2}{2} \ \frac{\epsilon^{'2} + (1-\epsilon^{'})^2}{2}.$$

Durch Einsetzen von $\epsilon^{'}$ aus der Formel 8 im Abschnitt 3.1.2 ergibt sich nach einigen einfachen Umformungen

$$R_{2parity} = \frac{1}{4} \ \frac{\epsilon^4 + (1-\epsilon)^4}{\epsilon^2 + (1-\epsilon)^2}.$$

Durch Induktion findet man, dass der Reduktionsfaktor R_{parity} für k Schritte des Parityprotokolls

$$R_{parity} = \frac{\epsilon^{2^k} + (1-\epsilon)^{2^k}}{2^k} \prod_{i=1}^{k-1} \frac{1}{\epsilon^{2^i} + (1-\epsilon)^{2^i}}. \tag{11}$$

beträgt. Hier sieht man, dass der Reduktionsfaktor des Repeatcodeprotokolls in der Schlussformel des Reduktionsfaktors des Parityprotokolls vorkommt, aber durch ein Produkt abgeschwächt wird, nämlich

$$R_{parity} = R_{repeat} \prod_{i=1}^{k-1} \frac{1}{\epsilon^{2^i} + (1-\epsilon)^{2^i}}. \tag{12}$$

Ein Zahlenbeispiel zeigt deutlich den Vorteil des Paritycodes gegenüber dem Repeatcode: Sei $\epsilon = 0.2$ und $k = 4$, das heisst 16 Bit Blöcke beim Repeatcode, so wird $R_{repeat} = 0.00176$, während $R_{parity} = 0.0375$ beträgt. Mit dem Paritycode können in diesem Beispiel bei gleicher Bitfehlerwahrscheinlichkeit 20 mal mehr Bits aus der Übertragung verwendet werden als mit dem Repeatcode.

3.2 Vergrösserung der Bitfehlerwahrscheinlichkeit

Wie im Abschnitt 3.1.1 erwähnt, können Alice und Bob immer ein Protokoll finden, sodass sie in ihren Bits eine kleinere Fehlerwahrscheinlichkeit haben als Eve. Der Nachteil aber ist, dass dabei nicht nur Alice und Bob besser werden, sondern auch Eve. Es geht nun darum, Protokollschritte so zu finden, dass zwar beide wieder schlechter werden, aber der Abstand zwischen der Entropie von Bob und derjenigen von Eve immer grösser wird. So wäre dann durch abwechselnde Verbesserung und wieder Verschlechterung ein Aufschaukeln der Entropiedifferenz möglich, bis Eve für ihre Bits eine Entropie von $1-\mu$ hat und Bob μ für μ beliebig klein.

3.2.1 Der Reduktionsschritt

Die einfachste Möglichkeit, die Fehlerwahrscheinlichkeit für alle wieder zu verschlechtern, besteht darin, dass Alice und Bob von entsprechenden Blöcken der Länge n jeweils nur das Parity behalten. Wenn die Bitfehlerwahrscheinlichkeit vor dem Schritt ϵ war, so beträgt sie nacher

$$\epsilon' = 1 - \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n}{2i} (1-\epsilon)^{n-2i} \epsilon^{2i},$$

nämlich die Gegenwahrscheinlichkeit davon, dass eine gerade Anzahl Bits falsch war. Durch geschicktes Addieren der Reihenentwicklungen von $((1-\epsilon)+\epsilon)^n$ und $((1-\epsilon)-\epsilon)^n$ findet man die schöne geschlossene Form

$$\epsilon' = \frac{1}{2} - \frac{1}{2}(1-2\epsilon)^n. \tag{13}$$

Wenn zum Beispiel $\epsilon = 0.05$ ist, so wird nach einem Reduktionsschritt der Länge $n = 8$ die neue Bitfehlerwahrscheinlichkeit $\epsilon' = 0.28$.

3.2.2 Vergleich von Reduktionsschritten verschiedener Länge

Wie beim Vergleich von Parity- und Repeatcode möchte ich hier einen Reduktionsschritt der Länge $n = 2^k$ mit k Reduktionsschritten der Länge 2, kurz Reduceschritten, vergleichen. Zuerst zeige ich, dass für Alice und Bob ein Reduktionsschritt der Länge $n = 2^k$ äquivalent ist mit k Reduceschritten. Sei die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob ϵ. Dann beträgt sie nach einem Reduceschritt

$$\epsilon^{'} = \frac{1}{2} - \frac{1}{2}(1-2\epsilon)^2.$$

Führt man anschliessend einen zweiten Reduceschritt durch, findet man

$$\epsilon^{''} = \frac{1}{2} - \frac{1}{2}(1-2\epsilon^{'})^2,$$

was nach Einsetzen von $\epsilon^{'}$ und Vereinfachen

$$\epsilon^{''} = \frac{1}{2} - \frac{1}{2}(1-2\epsilon)^4$$

ergibt. Durch Induktion findet man sofort, dass ein Reduktionsschritt der Länge 2^k äquivalent ist mit k Reduceschritten, was Alice und Bob betrifft.

Weil Eve aber bei den Reduktionsschritten keine Information über den Kanal bekommt, spielt es auch für sie keine Rolle, ob ein grosser Reduktionsschritt oder nacheinander viele kleine ausgeführt werden.

In der folgenden Arbeit kann ich mich also ganz auf die Untersuchung von Parity- und Reduceschritten beschränken, ohne dass ich die Allgemeinheit von Repeatcode und Reduktionsschritten mit Länge $n = 2^k$ für $k > 1$ verliere.

3.3 Mischen von Parity- und Reduceschritten

Wie oben gezeigt wurde, kann man ohne Probleme Bitfehlerwahrscheinlichkeiten sowohl für Alice und Bob, als auch für Eve angeben, wenn man nur Parityschritte oder nur Reduceschritte durchführt. Schwierig aber wird es, wenn man beginnt, Protokollschritte zu mischen. Für Alice und Bob kann man zwar noch eine einfache Form angeben. Sei s ein String aus den Buchstaben R für Reduce und P für Parity, welcher eine bestimmte Ablaufreihenfolge von Protokollschritten festlegt, und sei ϵ die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob am Anfang. Dann kann man die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob für jedes Protokoll aus Parity- und Reduceschritten rekursiv berechnen mit

$$\begin{aligned} \epsilon(0) &= \epsilon \\ \epsilon(sR) &= \frac{1}{2} - \frac{1}{2}(1-2\epsilon(s))^2 \\ \epsilon(sP) &= \frac{\epsilon(s)^2}{\epsilon(s)^2 + (1-\epsilon(s))^2}. \end{aligned} \tag{14}$$

Ich habe als Beispiel den Fall untersucht, in dem abwechslungsweise je ein Parity- und ein Reduceschritt durchgeführt wird. Ich wollte herausfinden, ob bei gleichmässigem Abwechseln stets die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob auf 0 abnimmt. Dies ist nicht der Fall. Es gibt eine Grenze für die Startkonfiguration ϵ, wie die Abbildung 4 zeigt, bei der gleichmässiges Abwechseln weder zu Bitfehlerwahrscheinlichkeit 0 noch zu 0.5 führt, sondern in einem stabilen Zyklus mit zwei Zuständen bleibt.

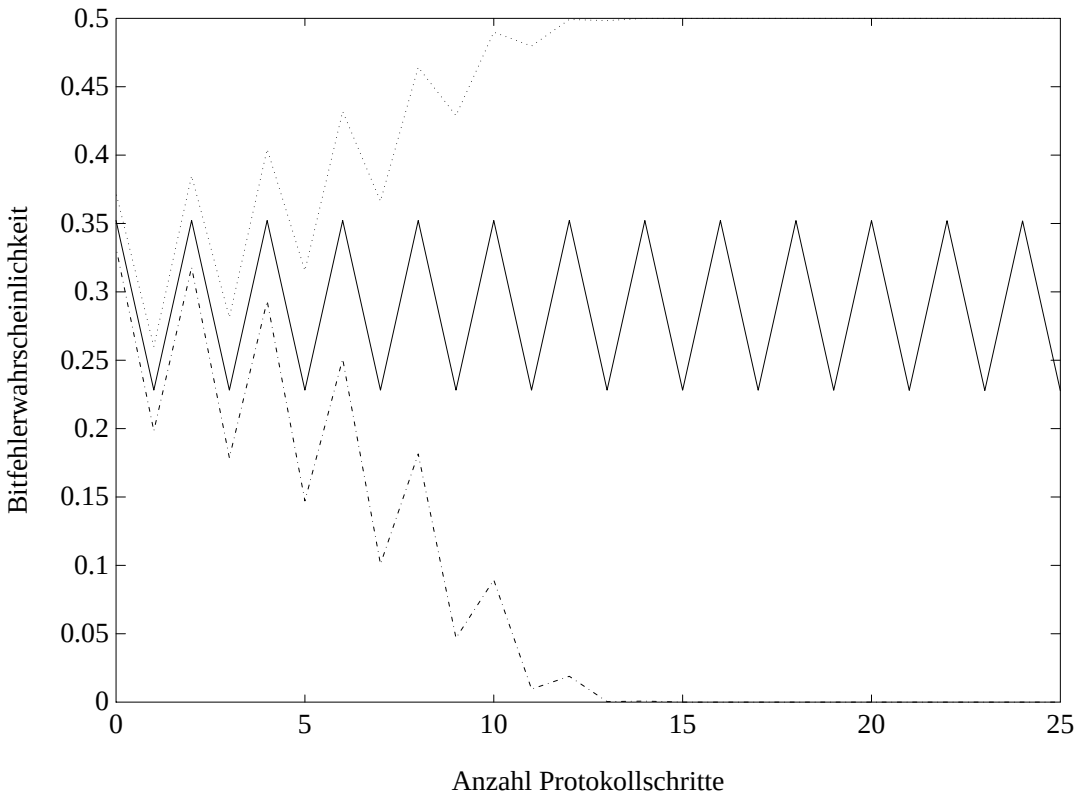


Abbildung 4: Grenze bei abwechslungsweisen Protokollschritten

Der obere der beiden Zustände liegt bei

$$\epsilon = 0.3522$$

und der untere bei

$$\epsilon = 0.2282.$$

Die Ausgangslage ϵ ist somit entscheidend für die Wahl der Protokollschritte. Startet man mit einem Parityschritt unterhalb der oberen Grenze, und wechselt man die Protokollschritte schön ab, so kommt man zu einer Bitfehlerwahrscheinlichkeit 0 für Alice und

Bob. Wenn man weit unterhalb startet, mag es sogar mehr Reduce als Parityschritte ertragen. Startet man hingegen oberhalb der oberen Grenze, so muss man mehr Parity- als Reduceschritte machen, um noch zu einer Bitfehlerwahrscheinlichkeit 0 für Alice und Bob zu kommen. Mit den Programmen aus Anhang A.2 kann man die Grenze für beliebige Protokolle berechnen.

Für Eve wird es aber schwierig, Aussagen über ihre Bitfehlerwahrscheinlichkeit zu machen, da sie nie gezwungen ist, einen Entscheid zu fällen, während das Protokoll läuft. Sie wird vielmehr alle Daten aufbewahren und am Schluss des ganzen Protokolls eine optimale Entscheidung suchen, wenn die Übertragung zwischen Alice und Bob beendet ist.

4 Numerische Analyse

Im Abschnitt 3.1.2 wurde gezeigt, wie gefährlich es ist, für Eve suboptimale Strategien anzunehmen. Um zu vermeiden, überhaupt eine Strategie für Eve anzunehmen, kann man einfach alle möglichen Fälle zwischenspeichern, die im Verlaufe eines Protokolles auftreten könnten, und für jeden weiteren Protokollschritt jeden Fall mit jedem wieder kombinieren. In diesem Kapitel wird mit Hilfe des Computers genau dieser Vorgang untersucht.

4.1 Protokollschritte als Abbildung von Verteilungen

Betrachten wir die Ausgangslage im Satellitenmodell mit den Bitfehlerwahrscheinlichkeiten ϵ_A für Alice, ϵ_B für Bob und ϵ_E für Eve. Daraus kann man die Wahrscheinlichkeit $\alpha_{i,j,k}$, dass Alice das Bit i, Bob das Bit j und Eve das Bit k hat, leicht berechnen. Sei $\delta_A = 1-\epsilon_A$, $\delta_B = 1-\epsilon_B$ und $\delta_E = 1-\epsilon_E$. Analog wie in Gleichung 1 findet man

$$\begin{aligned}
\alpha_{0,0,0} &= \frac{1}{2}(\delta_A\delta_B\delta_E + \epsilon_A\epsilon_B\epsilon_E) \\
\alpha_{0,0,1} &= \frac{1}{2}(\delta_A\delta_B\epsilon_E + \epsilon_A\epsilon_B\delta_E) \\
\alpha_{0,1,0} &= \frac{1}{2}(\delta_A\epsilon_B\delta_E + \epsilon_A\delta_B\epsilon_E) \\
\alpha_{0,1,1} &= \frac{1}{2}(\delta_A\epsilon_B\epsilon_E + \epsilon_A\delta_B\delta_E) \\
\alpha_{1,0,0} &= \frac{1}{2}(\epsilon_A\delta_B\delta_E + \delta_A\epsilon_B\epsilon_E) \\
\alpha_{1,0,1} &= \frac{1}{2}(\epsilon_A\delta_B\epsilon_E + \delta_A\epsilon_B\delta_E) \\
\alpha_{1,1,0} &= \frac{1}{2}(\epsilon_A\epsilon_B\delta_E + \delta_A\delta_B\epsilon_E) \\
\alpha_{1,1,1} &= \frac{1}{2}(\epsilon_A\epsilon_B\epsilon_E + \delta_A\delta_B\delta_E).
\end{aligned}$$

Der Faktor $\frac{1}{2}$ kommt daher, dass Alice mit gleicher Wahrscheinlichkeit eine 0 oder eine 1 sendet. Nun betrachten wir die Situation aus der Sicht von Eve. Falls Eve eine 0 sieht, ist die Wahrscheinlichkeit $\phi_{i,j}$, dass Alice das Bit i und Bob das Bit j hat

$$\begin{aligned}
\phi_{0,0} &= \alpha_{0,0,0} \\
\phi_{0,1} &= \alpha_{0,1,0} \\
\phi_{1,0} &= \alpha_{1,0,0} \\
\phi_{1,1} &= \alpha_{1,1,0}.
\end{aligned}$$

Genauso kann man für den dualen Fall, dass Eve eine 1 sieht, die $\phi_{i,j}$ angeben

$$\begin{aligned}\phi_{0,0} &= \alpha_{0,0,1}\\ \phi_{0,1} &= \alpha_{0,1,1}\\ \phi_{1,0} &= \alpha_{1,0,1}\\ \phi_{1,1} &= \alpha_{1,1,1}.\end{aligned}$$

Aus einer Situation im Satellitenmodell entstehen also zwei duale Fälle in der Verteilung der $\phi_{i,j}$. Nun kann man für einen Protokollschritt alle möglichen Kombinationen aus Fällen von $\phi_{i,j}$ verwenden und damit einen neuen Fall mit Parametern $\phi'_{i,j}$ berechnen. Somit wird ein Protokollschritt eine Abbildung von einer Verteilung der $\phi_{i,j}$ auf sich selbst.

4.1.1 Der Parityschritt

Es genügt, nur den Fall zu betrachten, in dem das Paritybit 0 ist, da der Fall mit Paritybit 1 die gleichen Wahrscheinlichkeiten liefert. Die Wahrscheinlichkeit von jedem Fall muss dann aber doppelt gezählt werden. Seien zwei Situationen gegeben durch die Koordinaten $\phi^1_{i,j}$ und $\phi^2_{i,j}$, dann werden die Koordinaten der neuen Situation $\phi'_{i,j}$ berechnet durch

$$\begin{aligned}\phi'_{0,0} &= \phi^1_{0,0}\phi^2_{0,0}\\ \phi'_{0,1} &= \phi^1_{0,1}\phi^2_{0,1}\\ \phi'_{1,0} &= \phi^1_{1,0}\phi^2_{1,0}\\ \phi'_{1,1} &= \phi^1_{1,1}\phi^2_{1,1}.\end{aligned} \tag{15}$$

Diese Formeln sind leicht zu verstehen, denn falls das Paritybit 0 ist, können Alice und Bob zum Beispiel genau dann je eine 0 haben, wenn sie vorher je 00 gehabt haben.

4.1.2 Der Reduceschritt

Ganz ähnlich kann man auch eine Abbildung für den Reduceschritt im Raum der $\phi_{i,j}$ beschreiben. Sei die erste Situation wiederum gegeben durch die Koordinaten $\phi^1_{i,j}$ und die zweite durch $\phi^2_{i,j}$, so erhält man die Koordinaten der neuen Situation $\phi'_{i,j}$ durch die Formeln

$$\begin{aligned}\phi'_{0,0} &= \phi^1_{0,0}\phi^2_{0,0} + \phi^1_{0,1}\phi^2_{0,1} + \phi^1_{1,0}\phi^2_{1,0} + \phi^1_{1,1}\phi^2_{1,1}\\ \phi'_{0,1} &= \phi^1_{0,0}\phi^2_{0,1} + \phi^1_{0,1}\phi^2_{0,0} + \phi^1_{1,0}\phi^2_{1,1} + \phi^1_{1,1}\phi^2_{1,0}\\ \phi'_{1,0} &= \phi^1_{1,0}\phi^2_{0,0} + \phi^1_{0,0}\phi^2_{1,0} + \phi^1_{0,1}\phi^2_{1,1} + \phi^1_{1,1}\phi^2_{0,1}\\ \phi'_{1,1} &= \phi^1_{0,0}\phi^2_{1,1} + \phi^1_{1,1}\phi^2_{0,0} + \phi^1_{0,1}\phi^2_{1,0} + \phi^1_{1,0}\phi^2_{0,1}.\end{aligned} \tag{16}$$

Auch diese Formeln sind einleuchtend, denn Alice und Bob können zum Beispiel genau dann je eine 0 haben, wenn sie vorher je 00 oder 11 gehabt haben.

4.2 Auswertung

Die oben hergeleiteten Formeln kann man verwenden, um, ausgehend von einer konkreten Ausgangslage, das heisst einer konkreten Verteilung der ϵ, verschiedene Protokollabläufe im Raum der ϕ durchzuführen und die entstehende Verteilung auszuwerten. Die Tabelle 1 zeigt die Resultate für alle Protokolle, die aus höchstens drei Protokollschritten bestehen, mit den Startwerten $\epsilon_A = \epsilon_B = 0.2$ und $\epsilon_E = 0.15$. Nach dem letzten Protokollschritt wurden für Eve und Bob ihre Bitfehlerwahrscheinlichkeiten über die ganze Verteilung gemittelt. In der dritten Kolonne ist die Entropiedifferenz zwischen Bob und Eve aufgetragen.

Protokoll	γ	β	$h(\gamma) - h(\beta)$
-	0.29000	0.32000	-0.03566
P *	0.24717	0.18130	0.12385
R	0.41180	0.43520	-0.01041
PP *	0.11457	0.04675	0.24112
PR	0.37215	0.29686	0.07489
RP	0.40202	0.37254	0.01952
RR	0.48444	0.49160	-0.00050
PPP	0.02966	0.00240	0.16834
PPR *	0.20289	0.08913	0.29411
PRP *	0.26335	0.15129	0.21870
PRR	0.46731	0.41747	0.01666
RPP	0.33244	0.26064	0.08969
RPR	0.48080	0.46751	0.00198
RRP	0.48418	0.48321	0.00009
RRR	0.49952	0.49986	-0.00000

Tabelle 1: Bitfehlerwahrscheinlichkeiten aus der Verteilungsabbildung

Das Protokoll aus lauter Parityschritten erzeugt am Anfang die grösste Entropiedifferenz für diese Startsituation. Schon bei drei Protokollschritten aber sieht man deutlich, wie sich die Protokolle PPR und PRP durch eine höhere Entropiedifferenz auszeichnen als das Protokoll PPP. Es lohnt sich also, Reduceschritte zu verwenden. Die Protokolle PPR und PRP habe ich noch weiter untersucht. Das beste Protokoll mit fünf Schritten ist für diese Startsituation das Protokoll $PPRPR$, wie aus Tabelle 2 hervorgeht.

4.3 Probleme

Die Formeln in Abschnitt 4.1 lassen vermuten, dass die Anzahl der zu behandelnden Fälle mit jedem Protokollschritt enorm zunimmt, entsteht doch bei einem Protokollschritt aus jeder Zweierkombination von Situationen eine neue. Man erwartet, dass die Anzahl Fälle $A(k)$ folglich nach k Protokollschritten gemäss der rekursiven Formel

Protokoll	γ	β	$h(\gamma) - h(\beta)$
PRPR	0.38800	0.25680	0.14162
PRPP *	0.13328	0.03080	0.36798
PPRR	0.32345	0.16237	0.26814
PPRP *	0.08110	0.00948	0.32872
PRPPP	0.04479	0.00100	0.25243
PRPPR *	0.23103	0.05970	0.45355
PPRPR *	0.14904	0.01879	0.47284
PPRPP	0.01900	0.00009	0.13445

Tabelle 2: Weiterverfolgen der grössten Entropiedifferenz

$$\begin{aligned} A(0) &= 2 \\ A(k+1) &= \binom{A(k)}{2} + A(k) \end{aligned} \tag{17}$$

zunimmt, nämlich die mögliche Anzahl Zweierkombinationen plus jede Situation mit sich selbst kombiniert. Führt man nur Parity- oder nur Reduceschritte aus, zeigt es sich, dass die Annahme zu pessimistisch ist, da viele Kombinationen aus Symmetriegründen wieder auf eine gleiche Situation führen. Mischt man aber Protokollschritte, so zeigt die Abbildung 5, dass die Anzahl Fälle weit über exponentiell steigt, selbst wenn man nur noch die Fälle betrachtet, in welchen die Bitfehlerwahrscheinlichkeiten von Alice und Bob gleich sind, das heisst $\epsilon_A = \epsilon_B$. Die durchgezogene Linie stellt die berechnete Anzahl Fälle nach Gleichung 17 dar. Gepunktet ist ein Protokoll ausschliesslich aus Reduceschritten, gestrichelt eines aus Parityschritten und mit Punkt-Strich dargestellt ein gemischtes. Deutlich wird, wie durch das Mischen von Protokollschritten die Symmetrie gebrochen wird. Man beachte, dass die Skalierung der Anzahl Fälle logarithmisch ist.

Das für Tabelle 1 und 2 verwendete Testprogramm ergab, dass man auf der Sun nicht mehr als 6 Protokollschritte berechnen kann, wenn man alle auftretenden Fälle behalten will. Selbst massiver Einsatz von Grosscomputern oder Parallelrechnern brächte nur einige Protokollschritte mehr. Es ist aussichtslos, mit dem Computer alle Fälle behalten zu wollen.

4.4 Gerasterte Verteilung

Eine Möglichkeit, die Fälle zu reduzieren, besteht darin, die auftretenden Fälle auf eine beschränkte Anzahl Gitterpunkte zu runden, ohne dabei einen Vorteil für Alice und Bob oder einen Nachteil für Eve einzuführen. Dies geht am einfachsten im Raum der ϵ, denn wenn ϵ_A und ϵ_B stets aufgerundet werden und ϵ_E stets abgerundet wird, kann für Eve kein Nachteil entstehen. Wenn $\epsilon_A = \epsilon_B$ und die Anzahl Rasterpunkte für ϵ_A und ϵ_E gleich n

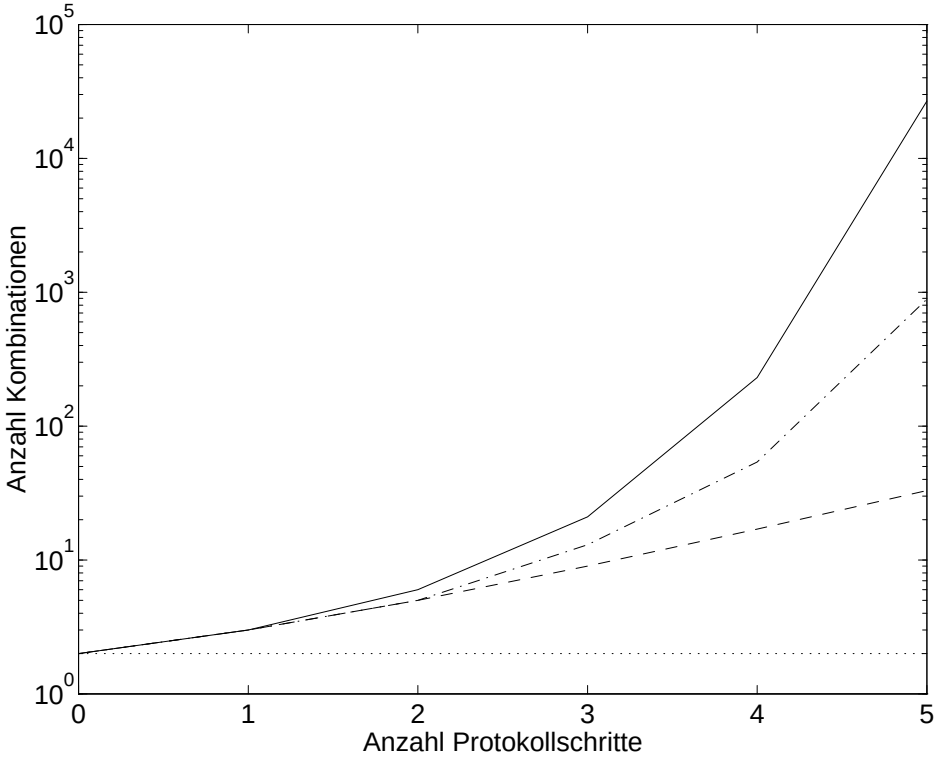


Abbildung 5: Anwachsen der zu betrachtenden Fälle

ist, dann kann die Anzahl der auftretenden Fälle nicht mehr grösser werden als n^2. Der Algorithmus, welcher die Abbildung vornimmt, ist somit höchstens von der Ordnung n^4 und nicht mehr abhängig von der Anzahl Protokollschritte.

Sobald nun im Programm die Anzahl Fälle einen Grenzwert überschreitet, wird die Verteilung aus dem ϕ-Raum in den ϵ-Raum übersetzt, dort auf diskrete Punkte gerundet, und wieder zurückübersetzt.

4.4.1 Mehr Protokollschritte

Durch das Runden wird es möglich, längere Protokolle zu untersuchen. Von der gleichen Ausgangslage ausgehend wie im Abschnitt 4.2 habe ich die Protokolle mit der grössten Entropiedifferenz, deren Verteilung die Abbildungen 6 und 7 zeigen, weiter untersucht mit einem Gitter aus $n^2 = 2500$ Rasterpunkten. Zur Beschriftung verwende ich wie im Abschnitt 3.3 einen String aus R für den Reduce- und P für den Parityschritt.

Es zeigt sich ganz klar, dass sich die Verteilung der ϵ_A bei einem Wert nahe bei 0 konzentriert, während die Verteilung der ϵ_E breit gestreut ist. Durch Weiterführen des Protokolles mit dem Ziel, die Entropiedifferenz zu maximieren, sind aus den Verteilungen in den Abbildungen 6 und 7 die Verteilungen der Abbildungen 8 und 9 entstanden.

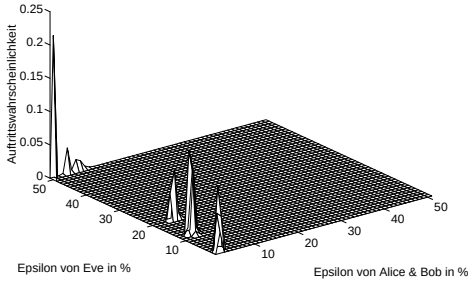


Abbildung 6: Protokoll PRPPR

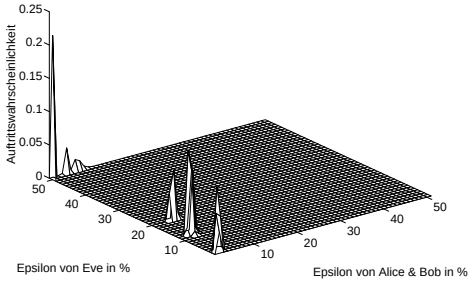


Abbildung 7: Protokoll PPRPR

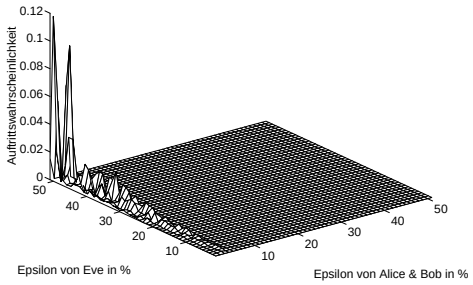


Abbildung 8: Protokoll PRPPRRPRPR

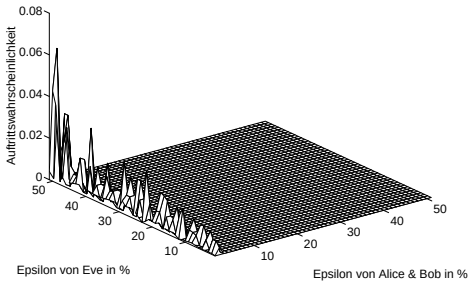


Abbildung 9: Protokoll PPRPRRRPRP

Bitfehlerwahrscheinlichkeiten und Entropiedifferenz dieser Verteilungen sind in der Tabelle 3 aufgetragen. Dies sind jedoch nicht mehr exakte, sondern auf die sichere Seite gerundete Werte.

Protokoll	γ	β	$h(\gamma) - h(\beta)$
PPRPRRRPRP	0.33078	0.01557	0.79994
PRPPRRPRPR	0.40535	0.02703	0.79472

Tabelle 3: Weiterverfolgen der grössten Entropiedifferenz

4.4.2 Extreme Ausgangslage

Die Ausgangslage $\epsilon_A = \epsilon_B = 0.2$ und $\epsilon_E = 0.15$ ist kein grosser Vorteil für Eve. Den Faktor f, um den ihr Kanal zu Beginn besser ist als derjenige von Bob, berechnet man mit

$$f = \frac{1 - h(\epsilon_E)}{1 - h(\epsilon_B)},$$

wobei $h(\epsilon)$ die binäre Entropiefunktion

$$h(\epsilon) = -\epsilon \log_2(\epsilon) - (1-\epsilon)\log_2(1-\epsilon)$$

ist. Im erwähnten Fall ergibt $f = 1.403$. Es fragt sich nun, wieviel man mit dem Protokoll aus Parity- und Reduceschritten erreichen kann, wenn f in die Grössenordnung von 10 oder sogar 100 kommt. Als Beispiel zeige ich den Fall mit Startsituation $\epsilon_A = \epsilon_B = 0.4$ und $\epsilon_E = 0.1$. Eve hat hier am Anfang einen um den Faktor $f = 18.279$ besseren Kanal. Die Abbildungen 10 und 11 zeigen für die Protokolle mit der grössten Entropiedifferenz, $PPPPPPRR$ und $PPPPPRPR$, wie stark sich das Runden auf die sichere Seite auf die Resultate auswirkt.

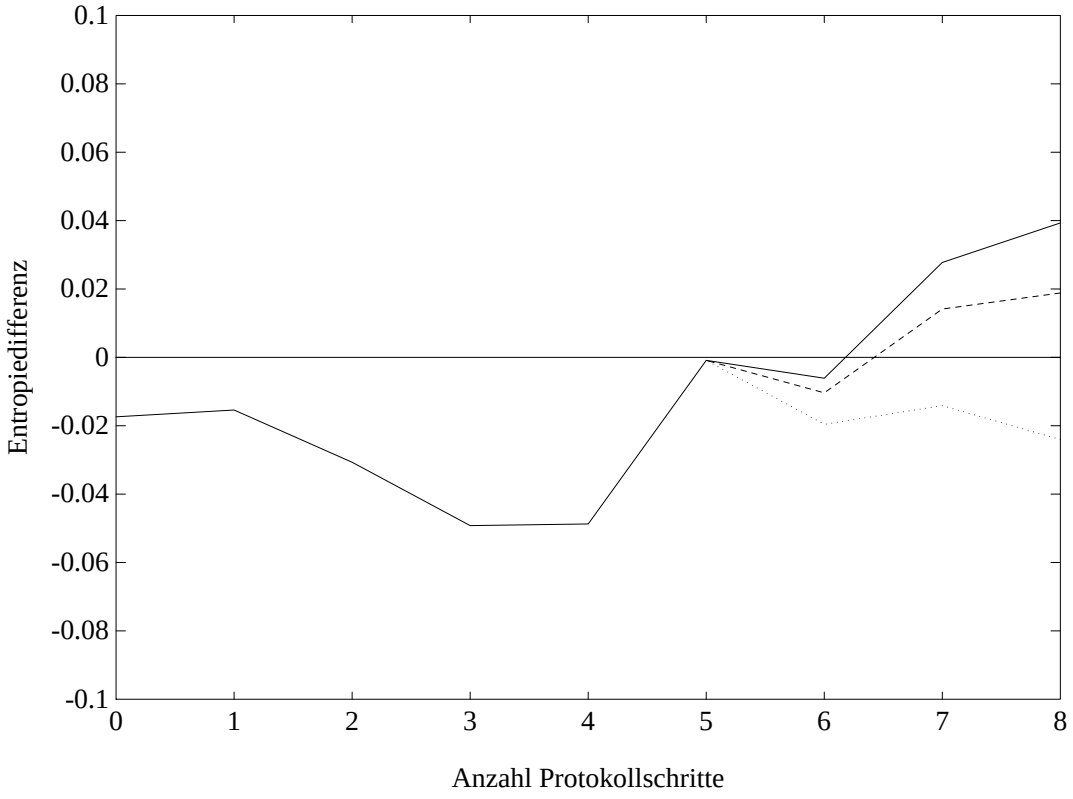


Abbildung 10: Wirkung des Rundens beim Protokoll PPPPPRPR

Bis zum fünften Protokollschritt können alle Kombinationen exakt berechnet werden. Nachher musste ich runden. Man sieht deutlich, wie die Gitterauflösung von $n^2 = 2500$ (gepunktete Linie) nicht mehr reicht, um die Entropiedifferenz zwischen Bob und Eve grösser als null zu machen. Auch die Auflösung von $n^2 = 10000$ (gestrichelte Linie) ist für das Protokoll $PPPPPPRR$ zu wenig. Ich musste n^2 auf den Wert 40000 erhöhen (durchgezogene Linie), was die Rechenzeit enorm verlängerte. Aber noch bei dieser Auflösung sind

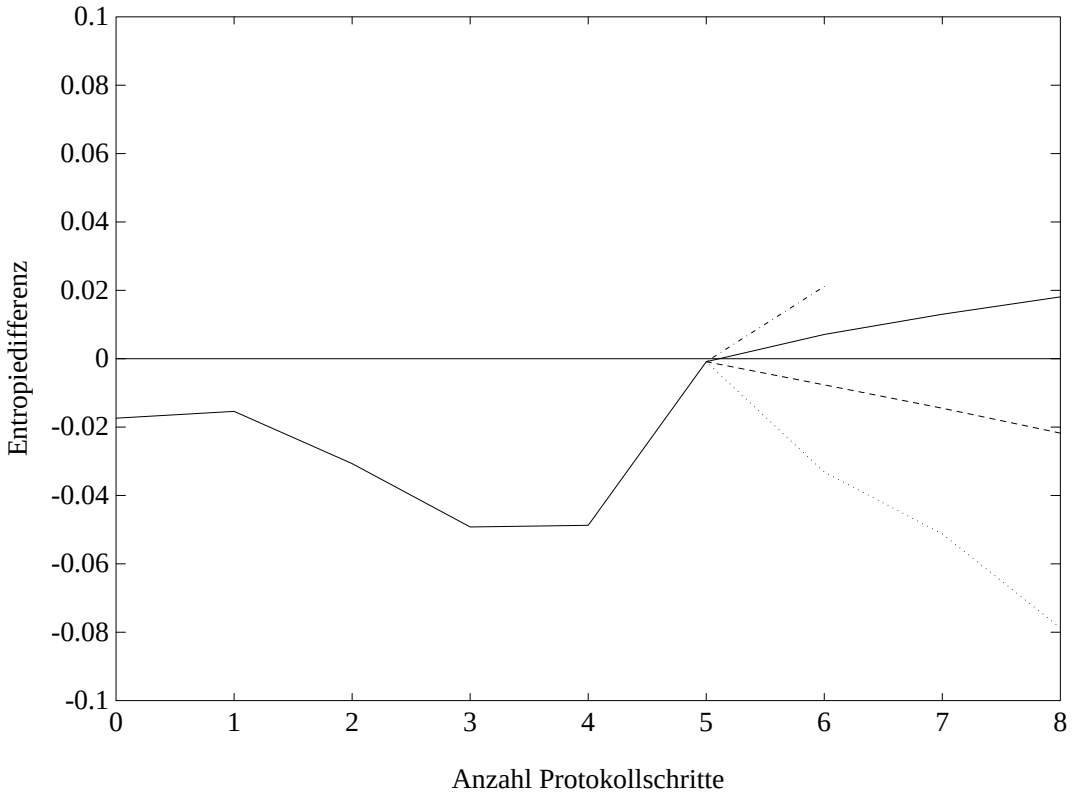


Abbildung 11: Wirkung des Rundens beim Protokoll PPPPPPRR

die Resultate viel zu pessimistisch, wie die Punkt-Strichlinie in 11 zeigt, welche durch die Formeln 4 und 7 aus Abschnitt 3.1.1 exakt berechnet werden kann.

Die Tabelle 4 zeigt die exakten Resultate so weit wie möglich und dann die gerundeten für die Auflösung $n^2 = 40000$.

Leider konnte ich in der zur Verfügung stehenden Zeit nicht mehr längere Protokolle berechnen. Schon für den nächsten Reduceschritt im Protokoll $PPPPPRPR$ muss der Computer 2.5 Milliarden Kombinationen berechnen bei einer Auflösung von $n^2 = 40000$. Eine Sun, welche während 6 Tagen ununterbrochen rechnete, konnte in dieser Zeit nicht einmal 10 Prozent der ganzen Aufgabe lösen. Hier würde nun ein schnellerer Computer nützen, da der Aufwand für weitere Schritte nicht mehr zunimmt.

Die durch die letzten beiden Protokolle von Tabelle 2 entstandenen Verteilungen im Raum der ϵ zeigen die Abbildungen 12 und 13. Man sieht, dass sich die wahrscheinlichsten Fälle um einen Peak in der Nähe von (0,0) scharen, während der Rest der Verteilung leer zu sein scheint. In Wirklichkeit ist er aber nicht leer, nur die Wahrscheinlichkeit, dass einer dieser Fälle auftritt, ist verschwindend klein. Man könnte nun die Anzahl Fälle drastisch reduzieren, indem man die mit kleiner Auftrittswahrscheinlichkeit weiter auf die sichere Seite von einem Gitterpunkt zum nächsten rundet. Ein Programm, welches genau dieses

Protokoll	Auflösung n^2	γ	β	$h(\gamma) - h(\beta)$
-	exakt	0.4200	0.4800	-0.01739
P	exakt	0.41693	0.46006	-0.01540
PP	exakt	0.37052	0.42063	-0.03068
PPP	exakt	0.29933	0.21743	-0.04921
PPPP	exakt	0.19253	0.21743	-0.04873
PPPPP	exakt	0.07143	0.07167	-0.00088
PPPPPR	40000	0.13279	0.13505	-0.00610
PPPPPP	40000	0.01070	0.00962	0.00710
PPPPPP	exakt	0.00891	0.00592	0.02114
PPPPPRP	40000	0.03196	0.02647	0.02776
PPPPPPR	40000	0.02100	0.01869	0.01301
PPPPPRPR	40000	0.06151	0.05183	0.03932
PPPPPPRR	40000	0.04089	0.03698	0.01809

Tabelle 4: Weiterverfolgen der grössten Entropiedifferenz

Runden vornimmt, ist bereits in Arbeit.

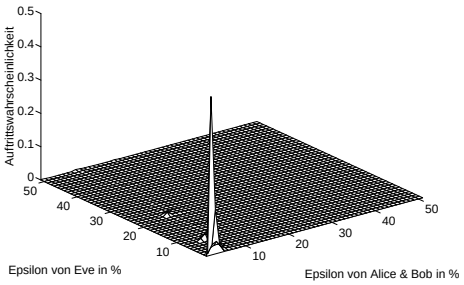


Abbildung 12: Verteilung nach dem Protokoll PPPPPRPR

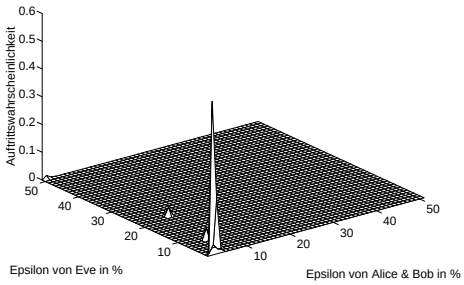


Abbildung 13: Verteilung nach dem Protokoll PPPPPPRR

5 Theoretische Analyse

Weil sich das Problem numerisch nur näherungsweise lösen lässt, habe ich mich auch sehr damit befasst, es theoretisch zu verstehen. Es gelingt, das Problem umzuformulieren auf ein bekanntes Problem, dessen Lösung im allgemeinen aber NP-vollständig ist.

5.1 Eve's Information als linearer Code

Um die Transformation leicht einzusehen, ist die Betrachtungsweise aus dem Abschnitt 3.1.3, welche die Äquivalenz von Parity- und Repeatcode plausibel macht, sehr nützlich.

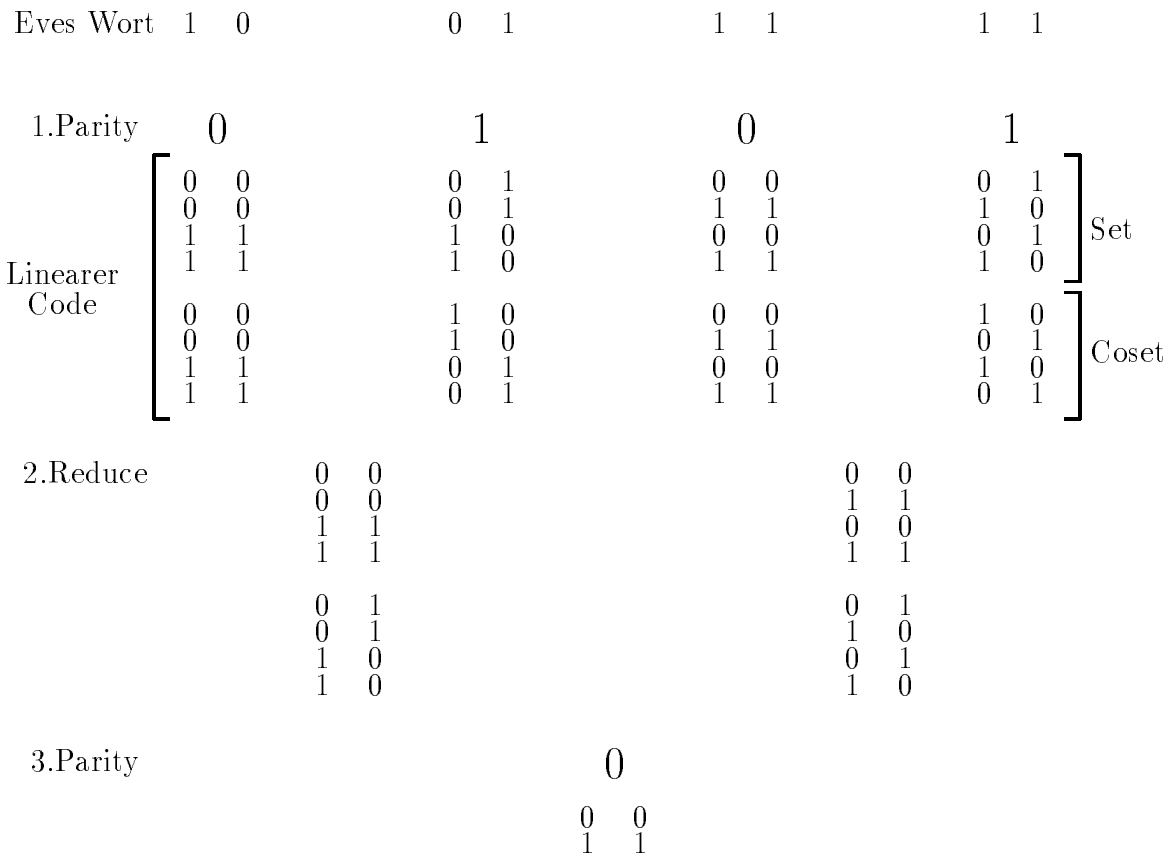


Abbildung 14: Aus einem Protokoll entstandener linearer Code

Versuchen wir, alle möglichen Bitstrings von Alice und Bob zu finden, die zu den gleiche Paritychecks in einem gegebenen Protokoll führen, so finden wir, dass Eve am Schluss eines Protokolls für Alice und Bob einen linearen Code sieht und selber ein Wort besitzt, das es zu dekodieren gilt. Ein Beispiel dazu zeigt Abbildung 14. Man sieht, wie sich Eve aus

der Kenntnis des Protokolls und den Paritywerten die Codewörter eines linearen Codes aufbauen kann. Wie in der Abbildung 3 gibt es auch hier einen Fall Oben und einen Fall Unten. Dadurch, dass der letzte Paritycheck 0 ist, weiss Eve, dass Alice am Schluss entweder 00 (Fall Oben) oder 11 (Fall Unten) gehabt hat. Weil der vorletzte Schritt ein Reduceschritt gewesen ist, kann im Fall Oben die 0 links aus entweder zwei Nullen oder zwei Einern entstanden sein. Das Gleiche gilt auch für die 0 rechts. Somit kann der Fall Oben aus 0000, 0011, 1100 oder 1111 entstanden sein. Den Fall Unten findet man analog.

Eve kann sich so alle möglichen Wörter konstruieren, aus denen das Protokoll entstanden sein könnte und muss sich am Ende der Übertragung mit ihrem Wort, das nicht unbedingt mit einem der gefundenen Wörter übereinstimmt, für den Fall Oben oder Unten entscheiden.

5.2 Analyse des speziellen linearen Codes

5.2.1 Anzahl der entstehenden Codewörter

Zuerst habe ich die Anzahl Codewörter des Codes untersucht. Man sieht, dass sich durch jeden Protokollschritt, egal ob Parity oder Reduce, die Codewortlänge verdoppelt. Beim Parityschritt bleibt die Anzahl der Codewörter konstant, während sie beim Reduceschritt drastisch zunimmt. Es gibt dabei für jedes Bit zwei Entstehungsmöglichkeiten, zum Beispiel kann das Bit 0 entstanden sein aus 00 oder 11. Sei die Anzahl der Codewörter nach dem Reduceschritt n, und ihre Länge m, so muss Eve vor dem Reduceschritt mit Codewörtern der Länge

$$m^{'} = 2m$$

in einer Anzahl

$$n^{'} = 2^{m}n$$

rechnen, denn für jedes Codewort kann jedes Bit aus zwei möglichen Kombinationen entstanden sein. Sei wiederum s ein String aus P für Parity und R für Reduce, so lässt sich die Anzahl der Codewörter rekursiv berechnen mit den Formeln

$$\begin{aligned} m(0) &= 1 \\ n(0) &= 2 \\ m(Ps) &= 2m(s) \\ m(Rs) &= 2m(s) \\ n(Ps) &= n(s) \\ n(Rs) &= 2^{m(s)}n(s). \end{aligned} \tag{18}$$

Man beachte dabei, dass der String s in dieser Rekursion von rechts nach links aufgebaut wird im Gegensatz zu früher, weil wir uns hier vom Protokollende zurückbewegen zur Ausgangssituation. Das Anwachsen der Anzahl Codewörter ist in der Abbildung 15 dargestellt.

Die drei gezeichneten Funktionen zeigen das maximale Wachstum bei ausschliesslich Reduceschritten, das mittlere Wachstum bei regelmässigem Abwechseln zwischen Parity- und Reduceschritten und das minimale Wachstum bei ausschliesslich Parityschritten, bei dem es stets nur zwei Codewörter gibt, wie schon in 3.1.3 gezeigt wurde.

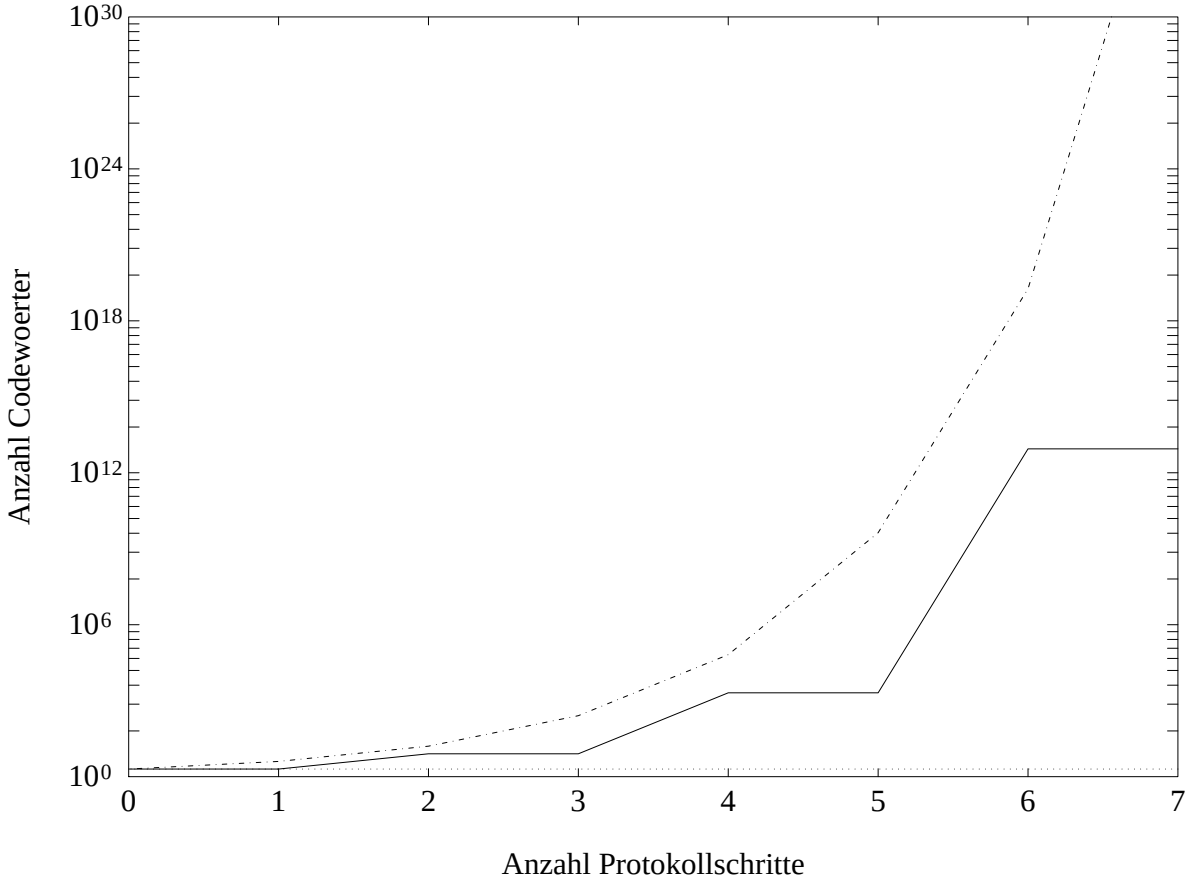


Abbildung 15: Anwachsen der Anzahl Codewörter

Jeweils die Hälfte der Codewörter gehört zum Set, die andere Hälfte zum Coset.

5.2.2 Distanzprofil des Codes

Interessant ist auch das Distanzprofil des Codes, insbesondere, da es bei diesem Code ja nur darum geht, ob ein Wort eher zu dem Codeteil gehört, welcher zu 0 führt, oder zu dem, der zu 1 führt. Ich bezeichne im Folgenden den Teil des Codes, der zu 0 führt, mit Set und den Teil, der zu 1 führt, mit Coset wie in Abbildung 14. Betrachten wir den Code, der entsteht, wenn alle ausgetauschten Paritychecks 0 sind. Dies ist zwar ein Spezialfall, aber die Distanzeigenschaften dieses Codes unterscheiden sich nicht von denen eines Codes mit anderen Paritychecks. Als Referenzwort aus dem Set, von dem aus ich alle Distanzen

berechne, verwende ich das Wort aus lauter Nullen. Schreibt man das Distanzprofil als liegenden Vektor $\vec{d}$ mit einem Index von $0..n$ und den Elementen d_i als Anzahl Codewörter mit dem Abstand i vom Referenzwort, so kann man $\vec{ds}(0)$, das Distanzprofil für den Set ohne Protokollschritt, angeben als

$$\vec{ds}(0) = [1, 0]\,,$$

da dieser Set nur aus dem Nullwort der Länge 1, also aus einem Bit besteht mit dem Wert 0. Der Distanzvektor für den Coset ist somit

$$\vec{dc}(0) = [0, 1]\,,$$

nämlich das Bit 1, also ein Wort mit Abstand 1 vom Referenzwort 0 im Set.

Wenn wir wiederum den String s als Notation für die Protokollabfolge verwenden, können wir die Wirkung eines Parityschrittes auf einen Distanzprofilvektor $\vec{d}$ der Länge $n+1$ angeben:

$$\vec{d}(Ps) = [d_0(s), 0, d_1(s), 0, \ldots, d_n(s)]\,. \tag{19}$$

Man sieht nämlich leicht, dass der Parityschritt nur die Anzahl der 0-Bits und die Anzahl der 1-Bits verdoppelt, wenn alle Paritychecks 0 sind. Damit ändert sich das Distanzprofil vom Referenzwort aus lauter Nullen, indem sich einfach alle Distanzen verdoppeln.

Die Wirkung des Reduceschrittes ist allerdings nicht so einfach einzusehen. Eine 1 kann durch einen Reduceschritt aus dem Paar 10 oder 01 entstanden sein, also aus zwei Worten mit Abstand 1 vom Nullwort. Zwei Einer, also 11 können durch einen Reduceschritt entstanden sein aus den Quadrupeln 1010, 1001, 0110 und 0101, also aus vier Worten mit Abstand 2 vom Nullwort. Allgemein können n Einer aus 2^n Worten mit Abstand n vom Nullwort entstanden sein. Eine 0 kann durch einen Reduceschritt aus dem Paar 00 oder 11 entstanden sein, also aus einem Wort mit Abstand 0 vom Nullwort und einem Wort mit Abstand 2 vom Nullwort. Zwei Nullen, also 00 können durch einen Reduceschritt entstanden sein aus den Quadrupeln 0000, 0011, 1100 und 1111, also aus vier Worten mit den Abständen $0, 2, 2, 4$ vom Nullwort. Allgemein können n Nullen aus 2^n Worten mit den Abständen $0, 2, 4, \ldots, 2n$ vom Nullwort entstanden sein, und zwar verteilt nach den Binomialkoeffizienten, das heisst $\binom{n}{0}$ mal Abstand 0, $\binom{n}{1}$ mal Abstand 2 und so weiter bis $\binom{n}{n}$ mal Abstand $2n$.

Schreibt man die Binomialkoeffizienten in eine Matrix, ähnlich dem Pascaldreieck

$$\begin{aligned}
Pascal_0 &= [1] \\
Pascal_1 &= \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \\
Pascal_2 &= \begin{bmatrix} 1 & 0 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\
\vdots \quad & \quad \vdots \qquad\qquad \vdots
\end{aligned}$$

und schreibt man die Zweierpotenzen in Diagonalmatrizen der Form

$$Pot_n = \begin{bmatrix} 2^0 & 0 & \dots & 0 \\ 0 & 2^1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 2^n \end{bmatrix},$$

so lässt sich auch die Abbildung des Distanzprofils d der Länge $n+1$ durch einen Reduceschritt direkt schreiben als

$$\vec{d}(Rs) = \vec{d}(s) Pot_n Pascal_n. \tag{20}$$

Die Matrix der Zweierpotenzen Pot_n erzeugt dabei alle möglichen Entstehungskombinationen der Einer, die Pascalmatrix die Entstehungskombinationen der Nullen. In den Tabellen 5 und 6 ist die Entwicklung der Distanzprofile $\vec{ds}$ für den Set und $\vec{dc}$ für den Coset für einige Protokolle dargestellt. Das verwendete Protokoll wird wiederum mit String s bestehend aus P und R bezeichnet.

Protokoll	Distanzen vom Codewort 0								
	0	2	4	6	8	10	12	14	16
P	1								
RP	1	2	1						
PRP	1		2		1				
RPRP	1	8	60	184	518	184	60	8	1
...	...								

Tabelle 5: Entwicklung des Distanzprofils $\vec{ds}$ vom Set

Protokoll	Distanzen vom Codewort 0								
	0	2	4	6	8	10	12	14	16
P		1							
PR		4							
PRP			4						
PRPR			64	256	384	256	64		
...	...								

Tabelle 6: Entwicklung des Distanzprofils $\vec{dc}$ vom Coset

Mit dem Distanzprofil kann man die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob berechnen. Wenn Alice das Referenzwort aus lauter Nullen aus dem Set hat, so

ist die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob einfach die Summe über das Distanzprofil des Coset mal die Wahrscheinlichkeit, dass ein Wort mit der entsprechenden Distanz auftritt. Sei die Bitfehlerwahrscheinlichkeit vor dem Protokoll ϵ, so ist sie nach dem Protokoll s mit k Schritten

$$\epsilon(s) = \sum_{i=0}^{n} \vec{dc}_i(s)\epsilon^i(1-\epsilon)^{n-i} \quad \text{mit} \quad n = 2^k.$$

In 3.3 wurde schon die Formel 14 für $\epsilon(s)$ hergeleitet, denn die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob kann direkt berechnet werden. Mit Maple konnte ich zeigen, dass die beiden Formeln nicht nur numerisch, sondern sogar symbolisch gleich sind. Also muss das kompliziert zu berechnende Distanzprofil auch in den früher hergeleiteten Formeln enthalten sein. Ich habe aus der rekursiven Formel 14 das Distanzprofil hergeleitet, bin aber auf die gleich komplizierte Form gestossen wie oben.

Die Abbildungen 16 und 17 zeigen die Entwicklung des Distanzprofils von einem Protokoll, das schon in der numerischen Analyse untersucht worden ist.

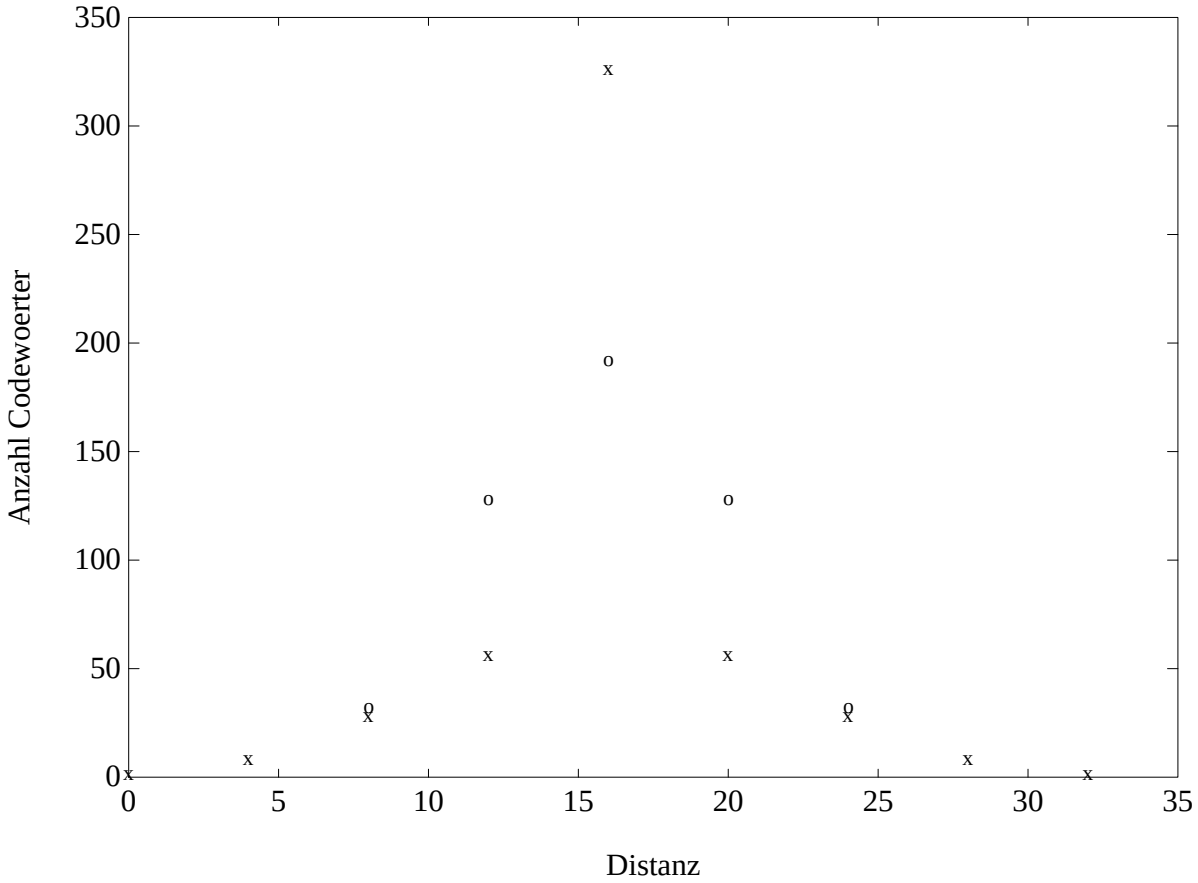


Abbildung 16: Distanzprofil von Protokoll PRPPR

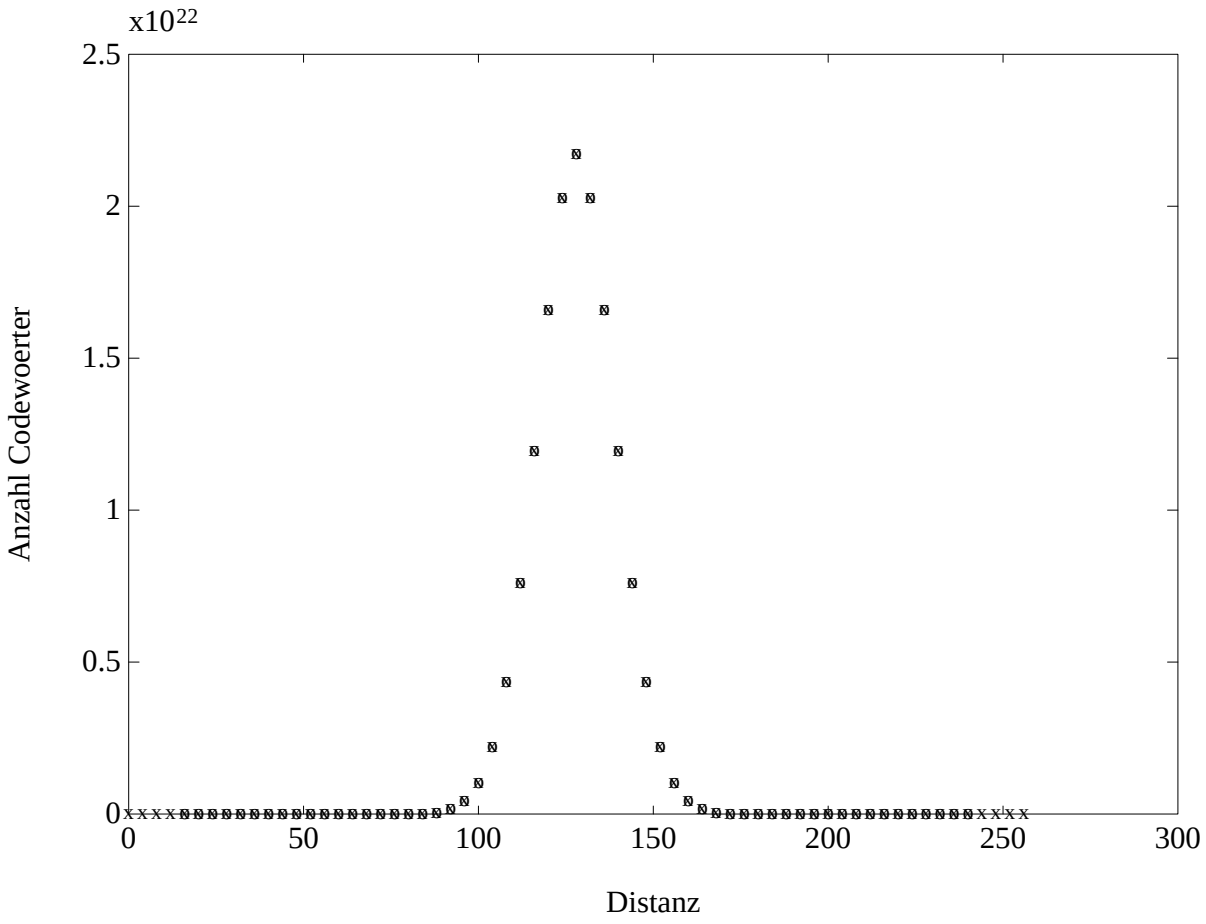


Abbildung 17: Distanzprofil von Protokoll PRPPRRPR

Man sieht, dass sich die Distanzprofile des Set (mit 'x' markiert) und des Coset (mit 'o' markiert) nähern, ihre Verteilung im mittleren Bereich sogar ineinander übergeht. Am Rand hingegen gibt es nur Codewörter vom Set und keine vom Coset. Ziel sollte somit sein, Bob im Bereich des Randes zu halten, während Eve in den Bereich gedrängt wird, in dem Wörter aus dem Set und Coset gleichermassen vertreten sind. Damit wird die Entscheidung für Eve schwierig, ob Alice ein Wort aus dem Set oder eines aus dem Coset hat. Dies ist aber erst möglich, wenn Bob einen Vorteil hat gegenüber Eve. Dann ist die Wahrscheinlichkeit für Bob, eine kleine Distanz vom richtigen Wort zu haben, grösser als die von Eve.

5.2.3 Generatormatrix des Codes

Um die Generatormatrix G für den Code zu finden, ist es in diesem Fall am einfachsten, über die Paritycheckmatrix H zu gehen. H kann man sofort hinschreiben, da die Paritychecks durch die Parityschritte explizit bekannt sind. So ist zum Beispiel H für das Protokoll aus Abbildung 14 gegeben durch

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Wenn die Paritiycheckmatrix bekannt ist, so lässt sich die Generatormatrix G' eines äquivalenten Codes daraus sofort berechnen, weil die folgende Beziehung gilt (vergleiche [5])

$$G = \left[I \ \vdots \ P \right] \Longleftrightarrow H = \left[-P^T \ \vdots \ I \right].$$

Dabei bezeichnet I die Identität. Für unser Beispiel findet man für G'

$$G' = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

Somit ist der entstehende Code für jede Kombination von Parity- und Reduceschritten durch sein Äquivalent bestimmt.

Leider findet man durch diese Art der Berechnung die Struktur der eigentlichen Matrix G nicht. So habe ich untersucht, was ein Protokollschritt auf einer bestehenden Matrix direkt bewirkt. Ähnlich wie im Abschnitt 5.2.2 gehe ich von einem Code aus, der aus lauter Paritychecks entstanden ist, die 0 sind. Sei G eine gegebene Matrix zu einem linearen Code, dann sind die Zeilenvektoren von G eine Basis für die Codewörter. Macht man einen Parityschritt, so kann man mit den gleichen Argumenten wie in Abschnitt 5.2.2 einfach die 0-Bits und 1-Bits der Basisvektoren in G verdoppeln und erhält die neue Generatormatrix

G'. Macht man einen Reduceschritt, so muss man sicher für jeden Basisvektor in G einen Basisvektor in G' aufnehmen, der den Basisvektor von G durch einen Reduceschritt erzeugt. Weiter kann der 0-Vektor nicht nur durch Multiplikation mit 0 erzeugt werden, sondern durch alle Vektoren, welche an gerader Position ein 00- oder 11-Paar haben. Man muss also noch erzeugende Vektoren für diese in der Matrix G' aufnehmen.

Nun kann die Abbildung formuliert werden. Sei D_n eine doppelte Identitätsmatrix der Höhe n , also zum Beispiel

$$D_4 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix},$$

und E_k eine ähnliche Matrix, aber mit jeweils nur einem 1-Bit, im Beispiel

$$E_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

dann kann G folgendermassen rekursiv berechnet werden. Sei s wieder ein String aus P für Parityschritt und R für Reduceschritt, k die Anzahl Protokollschritte und $n = 2^k$,

$$\begin{aligned} G(0) &= [1] \\ G(Ps) &= G(s)D_n \\ G(Rs) &= \begin{bmatrix} D_{2n} \\ \cdots \\ G(s)E_n \end{bmatrix}. \end{aligned} \tag{21}$$

Die Matrix G für unser Beispiel lautet somit

$$G = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

Man sieht sofort, dass sich die Matrix G in die aus der Paritycheckmatrix H berechnete Matrix G' durch elementare Zeilenkombinationen und durch Spaltentauschen überführen lässt. Die beiden Codes sind also äquivalent.

Lässt man in der Generatormatrix den untersten Basisvektor weg, so werden nur noch Codewörter aus dem Set erzeugt, was direkt aus der Konstruktion von G hervorgeht.

5.3 Entscheidungsverfahren für Eve

Die Entscheidung, zu welchem Codewort ein Wort dekodiert, ist für allgemeine Codes NP-vollständig. Dazu kommt das Problem, dass Eve nicht eigentlich an einem Codewort interessiert ist, zu dem ihr Wort dekodiert, sondern vielmehr daran, ob ihr Wort eher zum Set passt oder zum Coset.

5.3.1 Eine ausschöpfende Methode

Nehmen wir an, dass Eve den Bitstring e vom Satelliten erhalten hat und aus dem von Alice und Bob durchgeführten Protokoll den Set und Coset des entstandenen linearen Codes aufgebaut hat. Dann kann sie die Wahrscheinlichkeit $P(i,j,e)$, dass sie das Wort e, Alice das Bit i und Bob das Bit j hat mit $i,j \in \{0,1\}$, wiefolgt ausrechnen:

$$\begin{aligned} P(0,0,e) &= \sum_{\substack{a\in Set\\ b\in Set}} \prod_j \alpha_{a_j,b_j,e_j} \\ P(0,1,e) &= \sum_{\substack{a\in Set\\ b\in Coset}} \prod_j \alpha_{a_j,b_j,e_j} \\ P(1,0,e) &= \sum_{\substack{a\in Coset\\ b\in Set}} \prod_j \alpha_{a_j,b_j,e_j} \\ P(1,1,e) &= \sum_{\substack{a\in Coset\\ b\in Coset}} \prod_j \alpha_{a_j,b_j,e_j} \end{aligned} \tag{22}$$

Dabei bedeutet $\alpha_{i,j,k}$ die Wahrscheinlichkeit, dass Alice das Bit i, Bob das Bit j und Eve das Bit k hat, wie im Kapitel 4.1 definiert, und a_j bezeichnet zum Beispiel das Bit Nummer j des Codewortes a. Um nun zu entscheiden, welches Bit für Alice wahrscheinlicher ist, wenn Eve den Bitstring e hat, genügt es, die Summen

$$P_0(e) = P(0,0,e) + P(0,1,e)$$

und

$$P_1(e) = P(1,0,e) + P(1,1,e)$$

gegeneinander abzuwägen. Ist $P_0(e)$ grösser als $P_1(e)$, so ist es wahrscheinlicher, dass Alice das Bit 0 hat, wenn Eve das Wort e gesehen hat und umgekehrt. Den Fehler, den Eve mit dieser Strategie macht, entspricht genau dem Minimum von $P_0(e)$ und $P_1(e)$. Die totale Bitfehlerwahrscheinlichkeit γ von Eve lässt sich jetzt leicht berechnen. Man summiert über alle Wörter e, welche Eve beobachten kann, die Bitfehlerwahrscheinlichkeit, die Eve in diesem Fall macht. Das Resultat muss man mit der Wahrscheinlichkeit p_{accept}, dass Bob ein Wort von Alice akzeptiert, skalieren. Man findet

$$\gamma = \frac{1}{p_{accept}} \sum_e Min(P_0(e), P_1(e)). \tag{23}$$

Die Wahrscheinlichkeit p_{accept} kann man auf verschiedene Arten berechnen, jedoch in einem Programm zur Berechnung von γ geht es am einfachsten mit

$$p_{accept} = \sum_e P_0(e) + P_1(e).$$

Protokoll	γ	β	$h(\gamma) - h(\beta)$
-	0.29000	0.32000	-0.03566
P	0.24717	0.18130	0.12385
R	0.41180	0.43520	-0.01041
PP	0.11457	0.04675	0.24112
PR	0.37215	0.29686	0.07489
RP	0.40202	0.37254	0.01952
RR	0.48444	0.49160	-0.00050
PPP	0.02966	0.00240	0.16834
PPR	0.20289	0.08913	0.29411
PRP	0.26335	0.15129	0.21870
PRR	0.46731	0.41747	0.01666
RPP	0.33244	0.26064	0.08969
RPR	0.48080	0.46751	0.00198
RRP	0.48418	0.48321	0.00009
RRR	0.49952	0.49986	-0.00000

Tabelle 7: Theoretisch hergeleitete Bitfehlerwahrscheinlichkeiten

Die Tabelle 7 zeigt als Beispiel die berechneten Bitfehlerwahrscheinlichkeiten γ für Eve und β für Bob für die Ausgangslage $\epsilon_A = \epsilon_B = 0.2$ und $\epsilon_E = 0.15$. Sie stimmen mit den numerisch berechneten Werten aus Tabelle 1 überein.

Die Formel 23 gibt geschlossen die Bitfehlerwahrscheinlichkeit γ von Eve für ein bestimmtes Protokoll aus Parity- und Reduceschritten. Praktisch sind aber wieder Grenzen durch die Rechenleistung der heutigen Computern gesetzt. Eine obere Grenze für die Anzahl Codewörter c nach k Protokollschritten kann man aus der rekursiven Formel 18 in Abschnitt 5.2.1 herleiten. Man findet

$$c \leq 2^{2^k}.$$

Gleichheit gilt, wenn aussschliesslich Reduceschritte durchgeführt werden, wodurch alle Wörter der Länge 2^k zu Codewörtern werden. Der Aufwand $A(e)$ zur Berechnung der $P_1(e)$ und $P_0(e)$ für ein gegebenes e ist somit begrenzt durch

$$A(e) \leq c^2 = (2^{2^k})^2$$

weil jedes Codewort von Alice nach Formel 22 mit jedem von Bob kombiniert werden muss. Für den totalen Aufwand A für die Bitfehlerwahrscheinlichkeit γ von Eve muss das für alle Wörter e der Länge 2^k gerechnet werden. Somit hat A die obere Schranke

$$A \leq A(e)2^{2^k} = (2^{2^k})^2 2^{2^k} = O(2^{2^k}). \tag{24}$$

Im Extremfall ist der Aufwand für die Berechnung von γ mit der Formel 23 doppelt exponentiell in der Anzahl Protokollschritte k.

5.3.2 Ausnützen der Codeeigenschaften

Die Aufwandsabschätzung 24 zeigt, dass man an zwei Stellen einen doppelt exponetiellen Aufwand reduzieren muss. Eine Verbesserung an einem Faktor erreicht man dadurch, dass $A(e)$ nicht für alle Wörter e von Eve berechnet werden muss, sondern nur für diejenigen mit verschiedenem Syndrom. Das Syndrom s von einem Wort x ist definiert als $s = xH^T$. Sei die Paritycheckmatrix H eine m mal n Matrix. Um für jedes Syndrom ein Codewort zu finden, löst man für jedes Syndrom s das unterbestimmte Gleichungssystem

$$Hx = s,$$

indem man Komponenten von x für m linear unabhängige Spaltenvektoren von H berechnet und die übrigen Komponenten von x null setzt. Nun gibt es aber 2^m Syndrome und die Anzahl Codewörter ist in diesem Fall 2^{n-m}. Der reduzierte Aufwand A_r beträgt noch immer

$$A_r = (2^{n-m})^2 2^m,$$

mit $n = 2^k$. Ist zum Beispiel $m = n/2$, so beträgt der Aufwand $A_{\frac{1}{2}}$

$$A_{\frac{1}{2}} = (2^{\frac{n}{2}})^2 2^{\frac{n}{2}} = (2^{2^{k-1}})^2 2^{2^{k-1}} = O(2^{2^k}).$$

Man sieht, dass diese Verbesserung an der Ordnung der Berechnung nichts ändert. Man muss weitere Symmetrien im Code finden, um eine echte Verbesserung zu erreichen. In der in Abschnitt 5.2.3 hergeleiteten Generatormatrix sind offensichtlich noch viele Symmetrien vorhanden. Ich habe einen Teil davon untersucht, bin aber noch zu keinem vollständigen Resultat gekommen.

6 Ausblick

6.1 Weiterarbeit

6.1.1 Programmbeweis

Das Programm aus Anhang A.3 verwendet die in Modula-2 auf Unix zur Verfügung stehende Longrealarithmetik. Die Resultate sind also mit Rundungsfehlern behaftet. Dadurch ist es nicht möglich, einen Beweis mit Hilfe der berechneten Resultate zu führen, obwohl sie sicher die richtige Entwicklung der Situation zeigen.

Um einen Beweis über das Programm führen zu können, ist es notwendig, alle Realoperationen unter Kontrolle zu halten und stets auf die sichere Seite, also zu Gunsten von Eve und zu Ungunsten von Alice und Bob zu runden. Dazu müsste man eine eigene Realarithmetik implementieren.

Es gibt aber auch fertige Produkte, wie zum Beispiel Pascal-XSC [8], welche auf solche Probleme spezialisiert sind. Pascal-XSC stellt eine vollkommene Intervallarithmetik zur Verfügung mit allen Realoperationen. Wenn man mit Intervallvariablen rechnet, so ist das wirkliche Resultat garantiert im Intervall der Variablen zu finden. Es wäre interessant, das bestehende Modulaprogramm leicht abzuändern und damit Pascal-XSC zu testen.

6.1.2 Beweis über den linearen Code

Die explizite Formel für die Bitfehlerwahrscheinlichkeit γ von Eve ist in dieser Form nicht geeignet für einen Beweis, da einerseits numerisch die Auswertung aufwendig ist, andererseits auch mathematisch ein Grenzwert für lange Protokolle nicht gebildet werden kann. In der Formel sind aber noch nicht alle Symmetrien des speziellen linearen Codes ausgenutzt. Falls es gelingt, eine entscheidende Symmetrie zu finden, könnte eine theoretische und numerische Grenzwertbildung möglich sein, was für einen Beweis die sauberste Lösung wäre. Ich habe schon daran gearbeitet, konnte bisher aber nur die in der Arbeit vorgestellten Vereinfachungen vornehmen.

6.1.3 Vom linearen Code zum Protokoll mit mehreren Runden

Dieser Gedanke ist noch nicht konkret, aber eigentlich natürlich: wenn es einen Weg gibt von Protokollschritten, welche in mehreren Runden ausgetragen zu einem linearen Code führen, gibt es dann nicht auch den Rückweg von einem linearen Code zu Protokollschritten ? Könnte man für einen bestehenden, guten Code damit ein einfach zu verwendendes Protokoll finden ? Vielleicht gäbe sogar diese Betrachtung mehr Einsicht in das Wesen von guten linearen Codes.

6.2 Dank

Ganz herzlich möchte ich meinem Diplomprofessor Ueli Maurer danken für die stets offene Tür, und für die Geduld, mir Dinge zu erklären, die mir am Anfang völlig fremd waren.

Auch hat er mir immer geholfen, einen neuen Weg zu finden, wenn ich in eine Sackgasse geraten war und nicht mehr weiterkam. Ich habe zum erstenmal eine so intensive Zusammenarbeit erlebt, bei welcher der Betreuer derart aktiv mit bei der Sache war.

Weiter danke ich Dominik Gruntz für seine Hilfe bei den Postscriptbildern in Latex und Dr. Urs von Matt, der mir Matlab Version 4 zur Verfügung stellte, womit ich die dreidimensionalen Abbildungen der Verteilungen erstellt habe.

Ganz besonders danken möchte ich an dieser Stelle auch meinem Vater, der sämtliche Formeln in meiner Diplomarbeit von Hand auf ihre Richtigkeit überprüfte und die mühevolle Aufgabe des Korrekturlesens auf sich nahm.

Literatur

[1] U.M. Maurer, Secret key agreement by public discussion from common information, to appear in *IEEE Transactions on Information Theory*.

[2] U.M. Maurer, Protocols for secret key agrreement by public discussion from common information, to appear in *Advances in Cryptology -CRYPTO '92*, Lecture Notes in Computer Science, Berlin: Springer-Verlag.

[3] C.H.Bennett, G. Brassard and J.-M. Robert, "Privacy amplification by public discussion", *SIAM Jurnal on Computing*, Vol. 17, No. 2,pp.210-229,1988.

[4] R.G. Gallager, *Information Theory and Reliable Communication*, New York: John Wiley & Sons, 1968.

[5] R.E.Blahut, *Principles and Practice of Information Theory*, Reading, MA: Addison-Wesley, 1987.

[6] C.E. Shannon, Communication theory of secrecy systems, *Bell System Technical Journal*, Vol. 28, Oct. 1949, pp. 656-715.

[7] M.N. Wegman and J.L Carter, New hash functions and their use in authentication and set equality,*Journal of Computer and System Sciences*, Vol. 22, 1981, pp. 265-279.

[8] R. Klatte U. Kulisch M.Neaga D. Ratz Ch. Ullrich,*Pascal-XSC Sprachbeschreibung mit Beispielen*, Springer-Verlag.

A Programmbeschreibungen

A.1 Eve mit suboptimaler Strategie

Dieses c-Programm simuliert die Generierung eines Schlüssels im Satellitenmodell mit Hilfe des Parityprotokolls. Eve entscheidet dabei nach jedem Protokollschritt, was natürlich suboptimal ist. Gleichzeitig misst das Programm den Reduktionsfaktor. Alle Resultate werden auch berechnet und zum Vergleich neben den gemessenen dargestellt.

A.2 Alice und Bob

Diese Matlabprogramme dienen der Untersuchung der Bitfehlerwahrscheinlichkeit zwischen Alice und Bob bei Protokollen aus Parity- und Reduceschritten.

- alicebob :

  Berechnet die Bitfehlerwahrscheinlichkeit zwischen Alice und Bob für eine gegebene Bitfehlerwahrscheinlichkeit am Anfang und ein auszuführendes Protokoll aus Parity- und Reduceschritten.

- limit :

  Berechnet für ein bestimmtes Protokoll aus Parity- und Reduceschritten, wie gross die Bitfehlerwahrscheinlichkeit von Alice und Bob am Anfang sein darf, damit sie durch das Protokoll nicht schlechter werden. Die Entwicklung der Bitfehlerwahrscheinlichkeit während des Protokolls wird auch graphisch aufgetragen.

A.3 Abbildung von Verteilungen

Mit diesem Programm kann man interaktiv für eine beliebige Anfangssituation die Entwicklung für ein bestimmtes Protokoll berechnen lassen. Das Programm meldet, wenn die Anzahl Situationen zu gross wird und empfiehlt, den Raster einzuschalten, welcher die Werte auf diskrete Gitterpunkte rundet.

Das Programm ist in Modula geschrieben, die Abbildung 18 zeigt seinen Aufbau.

- MathConst :

  Berechnet MachEps und legt die Anzahl Nachkommastellen der Ausgabe fest.

- Distributions :

  Stellt den Datentyp Distribution, welcher eine diskrete Verteilung in zwei Dimensionen aufnimmt, zur Verfügung und die dazugehörenden grundlegenden Operationen.

- Chains :

  Stellt den Datentyp Chain zur Verfügung, welcher eine Verteilung in vier Dimensionen exakt festhalten kann. Grundlegende Operationen wie Einfügen, Löschen und Ausgeben werden auch exportiert.

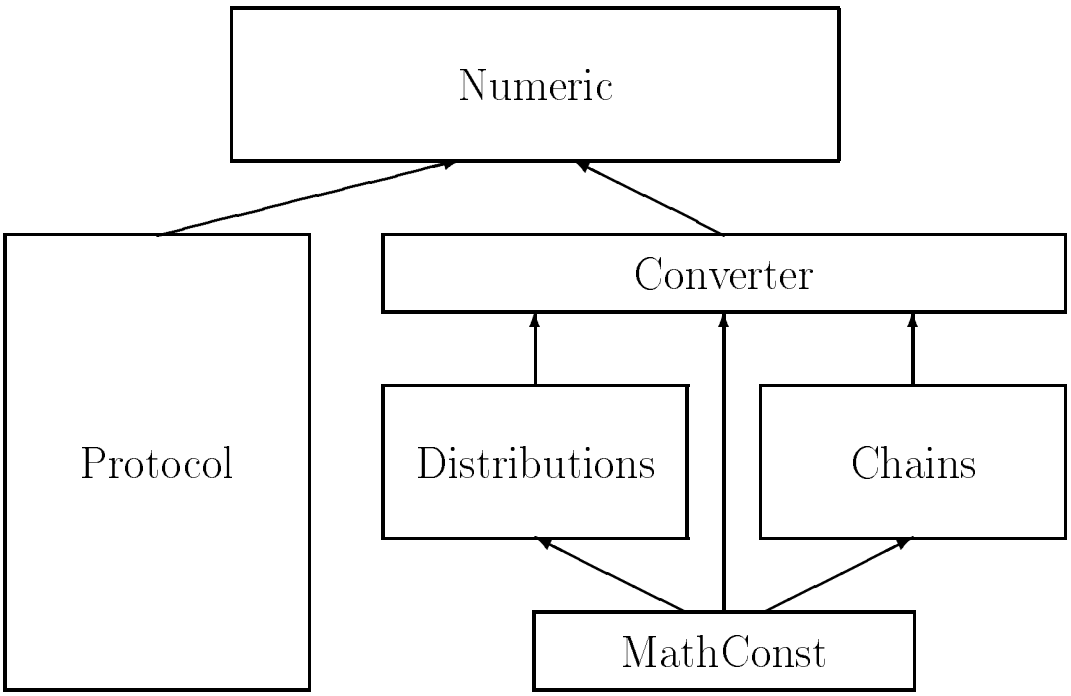


Abbildung 18: Aufbau des Programms Numeric

- Converter :

  Besorgt Umrechnungen vom ϵ-Raum in den α-Raum und Umrechnungen von der exakten Verteilung in einer Chain in eine gerundete Distribution.

- Protocol :

  Implementiert die möglichen Protokollschritte, im Moment den Parityschritt und den Reduceschritt jeweils aus zwei Bits.

- Numeric :

  Einfaches Benutzerinterface, welches interaktives Testen einzelner Protokolle erlaubt, oder für eine gegebene Anfangssituation das Protokoll mit der grössten Entropiedifferenz sucht. Falls die Entropiedifferenz negativ ist, werden Parityschritte bevorzugt.

A.4 Distanzprofil

Berechnet für ein gegebenes Protokoll das entstehende Distanzprofil. Das Programm besteht aus Matlabprozeduren, welche auch einzeln verwendet werden können. Es ist stets eine Onlinehelp eingebaut, sodass man durch Eingabe von 'help Prozedur' eine Beschreibung der Prozedur bekommt.

- distanceprofile :

  Berechnet das Distanzprofil des entstehenden linearen Codes für ein bestimmtes Protokoll aus Parity- und Reduceschritten.

- parity :

  Berechnet für ein gegebenes Distanzprofil das neue, das nach einem Parityschritt entsteht.

- reduce :

  Berechnet für ein gegebenes Distanzprofil das neue, das nach einem Reduceschritt entsteht. Verwendet intern die Prozeduren pascal und power2, deren eigenständige Verwendung keinen Sinn macht.

- condens :

  Macht aus einem Distanzprofilvektor, der für jede mögliche Distanz einen Eintrag hat, ein kompakteres Format.

- numberofcodewords :

  Berechnet die Anzahl der entstehenden Codewörter für ein gegebenes Protokoll aus Parity- und Reduceschritten. Die Anzahl wird nach jedem Schritt in einer Graphik eingezeichnet.

A.5 Berechnung der Generatormatrix

Das Programm generator berechnet für ein gegebenes Protokoll aus Parity- und Reduceschritten die Generatormatrix des entsprechenden linearen Codes. Intern verwendet das Programm zwei Hilfsprozeduren D.m und E.m, deren eigenständige Verwendung keinen Sinn macht.

A.6 Erzeugter Linearer Code

Dieses Programm berechnet die Bitfehlerwahrscheinlichkeit von Eve unter Verwendung des entstandenen linearen Codes für ein Protokoll aus Parity- und Reduceschritten und eine beliebige Anfangssituation. Das Protokoll muss bei diesem Programm in der umgekehrten Reihenfolge eingegeben werden. Ohne Anpassung der Konstanten können nur drei Schritte lange Protokolle berechnet werden. Für längere Protokolle ist mit langen Antwortzeiten zu rechnen, da keine Optimierungen vorgenommen wurden.

B Programme

B.1 Eve mit suboptimaler Strategie

```
#include <stdio.h>
#include <time.h>
#include <strings.h>

#define N 10000                 /* no of bits to start with */
#define SCREENWIDTH 60

char s[N],a[N],b[N],e[N];

void fill(s,n)                  /* fill bitstring s of length n */
  char *s;                      /* with random bits */
  int n;
{
  int i;

  for(i=1;i<=n;i++) s[i]=(char)(random()&01);
};

float percent_fill(s,a,n,p)     /* fill bitstring a of length n */
  char *s,*a;                   /* with bits of s, but approx. */
  float p;                      /* p% of the bits change the value */
{                               /* returns actual percent of changed bits */
  int i,e=0;
  long l=2*32768*32768-1;

  for(i=1;i<=N;i++)
  {
    if(random()>p*l)
      a[i]=s[i];
    else
    {
      e++;
      a[i]=1-s[i];
    };
  };
  return (float)(e*100)/N;
};

void print_seq(a,str,n,p)       /* print bitstring a of length n */
  char *a;                      /* with label str and p% wrong bits */
  char *str;                    /* if n>SCREENWIDTH, the rest of a is */
  int n;                        /* skipped */
  float p;
{
  int i,m;

  if(n<SCREENWIDTH) m=n;
```

```
  else m=SCREENWIDTH;
  printf(str);
  for(i=1;i<=m;i++)
    printf("%d",a[i]);
  if(p>=0) printf(" error = %2.1f percent",p);
};

char eor(a,b)                   /* exclusiv or with characters of */
  char a,b;                     /* value 0 or 1 */
{
  int i;

  i=a+b;
  if(i>1) i=0;
  return i;
};

void parity_step(a,b,e,n,eb,ee) /* simulate paritystep */
  char *a,*b,*e;                /* n = length of bitstrings */
  int *n;                       /* a = bitstring of Alice */
  float *eb, *ee;               /* b = bitstring of Bob */
{                               /* e = bitstring of Eve */
  int i,m=1;                    /* eb,ee = resulting bit-error- */
                                /* probabilities of Bob and Eve */
  (*eb)=0;
  (*ee)=0;
  for(i=1;i<(*n);i=i+2)
  {
    if(eor(a[i],a[i+1])==eor(b[i],b[i+1]))
    {
      a[m]=a[i];
      b[m]=b[i];
      e[m]=e[i];                /* !!! Eve makes a decision !!! */
      if(a[m]!=b[m]) (*eb)++;
      if(a[m]!=e[m]) (*ee)++;
      m++;
    };
  };
  (*n)=m-1;
  (*eb)=((*eb)*100/(*n));
  (*ee)=((*ee)*100/(*n));
};

float power(x,n)                /* result=x^n for n>=0 */
  float x;
  long n;
{
  long i;
  float p;

  p=1;
```

```
  for(i=1;i<=n;i++) p=p*x;
  return p;
};

char sa[]="Dies ist der zu verschluesselnde Testtext";
char sb[100],se[100];

int main()
{
  int n,o,j;
  long i;
  float ea,eb,ee,da,db,de,a00,a01,a10,a11,p,pa,pb,pe;
  float beta,gamma,divisor,rf,rft,prod;

  srandom((long)time(0));

  printf("\n\n\n\nSimulation of the parity-protocol:\n");
  printf("_________________________________\n\n");
  printf("Simulation and calculation of reduce-factors and \n");
  printf("bit-error-probabilities doing the parity-protocol\n");
  printf("in the satelite-model.\n\n");
  printf("warning : suboptimal strategy for Eve ! \n");
  printf("          simulated and calculated error-probabilities\n");
  printf("          of Eve are incorrect.\n\n");
  printf("please open window as large as possible.\n\n");
  printf("\nerror-probability of Alice ea = ");
  scanf("%f",&ea);
  printf("error-probability of Bob   eb= ");
  scanf("%f",&eb);
  printf("error-probability of Eve   ee= ");
  scanf("%f",&ee);
  da=1-ea;
  db=1-eb;
  de=1-ee;
  p=ea*eb+da*db;
  a00=ea*eb*ee+da*db*de;
  a01=ea*eb*de+da*db*ee;
  a10=ea*db*ee+da*eb*de;
  a11=ea*db*de+da*eb*ee;
  do
  {
    n=N;
    fill(s,n);
    printf("\n\nStart-configuration : \n\n");
    pa=percent_fill(s,a,n,ea);
    pb=percent_fill(s,b,n,eb);
    pe=percent_fill(s,e,n,ee);
    print_seq(s,"Sat   : ",n,-1.0);
    printf("\n");
    print_seq(a,"Alice : ",n,pa);
    printf("\n");
```

```
    print_seq(b,"Bob   : ",n,pb);
    printf("\n");
    print_seq(e,"Eve   : ",n,pe);
    printf("\n");
    printf("started with %d bits\n\n",n);
    printf("\nSimulation:\n\n");
    i=1;
    j=1;
    prod=1;
    rf=1;                       /* simulated reduce factor */
    rft=1;                      /* theoretical reduce factor */
    while(n>100)
    {
      o=n;
      parity_step(a,b,e,&n,&pb,&pe);
      printf("%d. step:\n\n",j);
      j=j+1;
      print_seq(a,"Alice : ",n,-1);
      printf("\n");
      print_seq(b,"Bob   : ",n,pb);
      i=i*2;
      divisor=power(a00+a01,i)+power(a10+a11,i);
      beta=(a10*power(a10+a11,i-1)+a11*power(a11+a10,i-1))/divisor;
      printf(" (theory: %2.1f)\n",beta*100);
      print_seq(e,"Eve   : ",n,pe);
      gamma=(a01*power(a01+a00,i-1)+a11*power(a11+a10,i-1))/divisor;
      printf(" (theory: %2.1f)\n\n",gamma*100);
      rf=rf/((float)o/n);
      printf("reduced to %d bits, by factor %1.1f",n,(float)o/n);
      printf(", reduced since start by factor %1.4f",rf);
      rft=(power(p,i)+power(1-p,i))/i*prod;
      printf(" (theory: %1.4f)\n\n\n",rft);
      prod=prod/(power(p,i)+power(1-p,i));
    }
    printf("\n\ndone: 1 -> again   0 -> quit : ");
    scanf("%d",&i);
  }
  while(i>0);
  return 0;
}
```

B.2 Alice und Bob

```
function [e]=alicebob(s,p)

% ALICEBOB Compute bit-error-probabilities for a given protocol
%          ALICEBOB(s,p) computes for the errorprobability s between
%          Alice and Bob at thje beginning and the vector p
%          containing the protocolsteps (1=Parity 0=Reduce) the vector
%          e containing the errorprobabilities after each step.
%
```

```
% example  >> e=alicebob(0.32,[1 0 1 0])
%
%          e =
%
%              0.3200    0.1813    0.2969    0.1513    0.2568
%

[m,n]=size(p);
e=zeros(1,n+1);
c=zeros(1,n+1);
for i=0:n, c(1,i+1)=i; end;
e(1,1)=s;
for i=1:n,
  if p(1,i)==0,
    e(1,i+1)=0.5-0.5*(1-2*e(1,i))^2;
  else
    e(1,i+1)=e(1,i)^2/(e(1,i)^2+(1-e(1,i))^2);
  end;
end;
```

```
function [r]=limit(p,ex)

% LIMIT Compute maximal errorprobability between Alice and Bob
%       LIMIT(p,ex) computes for the given protocolvector p
%       (1=paritystep, 0=reducestep) the maximal errorprobability,
%       that Alice and Bob may have at the beginning without being
%       worse at the end. The real result lies between r-ex and r+ex.
%       A plot shows the protocol with three different startpoints.
%
% example >> limit([1 0 1 0 1 0 1],0.0001)
%
%         ans =
%
%             0.3522

[m,n]=size(p);
c=zeros(1,n+1);
for i=0:n, c(1,i+1)=i; end;
a=0.0;
b=0.5;
while abs(a-b)>ex,                % bisect
  s=(a+b)/2;
  e=alicebob(s,p);
  if e(1,n)>s, b=s; else a=s; end;
end;
r=s;
if s-0.02>0, e1=alicebob(s-0.02,p); else e1=alicebob(0,p); end;
if s+0.02<0.5, e2=alicebob(s+0.02,p); else e2=alicebob(0.5 ,p); end;
axis([0 n 0 0.5]);
plot(c,e,'-',c,e1,'-.',c,e2,':');
xlabel('protocolsteps');
ylabel('bit-error-probability');
```

B.3 Abbildung von Verteilungen

```
MODULE Numeric;

  IMPORT
    InOut,LongRealIO,MathConst,Distributions,Chains,Converter,Protocol,
    MathLib;

  CONST
    Parity=1;
    Reduce=2;
    StartRasteringInteractiv=150;
    StartRasteringFindBest=150;

  TYPE
    Prot=ARRAY[1..20] OF INTEGER;

  VAR
    field,digits:INTEGER;
    ea,eb,ee,a00,a01,a10,a11:LONGREAL;
    d:Chains.Chain;
    inp:INTEGER;
    done:BOOLEAN;
    ch:CHAR;
    dist:Distributions.Distribution;

  PROCEDURE DoProtocol(VAR d,dn:Chains.Chain;step:INTEGER;raster,proz:BOOLEAN);
    VAR
      ea,eb,ee,ps,p,p00,p01,p10,p11:LONGREAL;
      d1,d2:Chains.Chain;
      n,i,j:LONGINT;
  BEGIN
    IF raster THEN Distributions.Clear(dist) END;
    d1:=d;
    dn:=NIL;
    IF proz THEN                          (* show % already processed *)
      n:=Chains.NoOfEntry(d);
      IF n>30 THEN i:=0;j:=0 ELSE proz:=FALSE END;
    END;
    WHILE d1<>NIL DO
      d2:=d1;
      WHILE d2<>NIL DO
        p00:=d1^.p00;
        p01:=d1^.p01;
        p10:=d1^.p10;
        p11:=d1^.p11;
        p:=d1^.p*d2^.p;
        IF d1<>d2 THEN p:=p*2.0 END;     (* symmetry *)
        IF step=Reduce THEN
          Protocol.Reduce(p00,p01,p10,p11,d2^.p00,d2^.p01,d2^.p10,d2^.p11);
        ELSE
          Protocol.Parity(p00,p01,p10,p11,d2^.p00,d2^.p01,d2^.p10,d2^.p11,ps);
```

```
        p:=p*ps;
      END;
      IF raster THEN
        Converter.AlphaToEps(p00,p11,p01,p10,ea,eb,ee);
        Distributions.Insert(dist,ea,ee,p);
      ELSE
        Chains.Insert(dn,p00,p01,p10,p11,p,TRUE);
      END;
      d2:=d2^.next;
    END;
    d1:=d1^.next;
    IF proz THEN                        (* show % already processed *)
      i:=i+1;
      IF FLOAT(i)>=FLOAT(n)/10.0*FLOAT(j+1) THEN
        j:=j+1;
        InOut.WriteInt(j*10,3);
        InOut.WriteString(" %");
        InOut.WriteLn;
      END;
    END;
  END;
  IF raster THEN
    Converter.DistToChain(dist,dn);
  END;
  IF step=Parity THEN Chains.Rescale(dn) END;
  IF proz THEN InOut.WriteLn;InOut.WriteLn END;
END DoProtocol;

PROCEDURE Interactiv(d:Chains.Chain);
  VAR
    errB,errE,hB,hE:LONGREAL;
    inp,n:INTEGER;
    raster:BOOLEAN;
    dn:Chains.Chain;
BEGIN
  InOut.WriteLn;
  InOut.WriteString("interactiv mode :");
  InOut.WriteLn;
  InOut.WriteLn;
  raster:=FALSE;
  REPEAT
    Chains.Entropy(d,errB,errE,hB,hE);
    n:=Chains.NoOfEntry(d);
    InOut.WriteString("error-probability of Bob : ");
    LongRealIO.WriteLongReal(errB,field,digits);
    InOut.WriteLn;
    InOut.WriteString("error-probability of Eve : ");
    LongRealIO.WriteLongReal(errE,field,digits);
    InOut.WriteLn;
    InOut.WriteString("entropy difference       : ");
    LongRealIO.WriteLongReal(hE-hB,field,digits);
```

```
    InOut.WriteLn;
    InOut.WriteString("number of entries        : ");
    InOut.WriteInt(n,field);
    InOut.WriteLn;
    InOut.WriteLn;
    IF NOT raster AND (n>StartRasteringInteractiv) THEN
      InOut.WriteString("raster recommended to go on interactiv !");
      InOut.WriteLn;
      InOut.WriteLn;
    END;
    IF raster THEN
      InOut.WriteString("0=END 1=Paritystep 2=Reducestep 3=SaveMatrix 4=Save: 5=Round : ");
    ELSE
      InOut.WriteString("0=END 1=Paritystep 2=Reducestep 3=SaveMatrix 4=Save  5=Round 6=RasterOn : ")
    END;
    InOut.ReadInt(inp);
    InOut.WriteLn;
    IF (inp=Parity) OR (inp=Reduce) THEN
      DoProtocol(d,dn,inp,raster,TRUE);
      Chains.Deallocate(d);
      d:=dn;
    ELSIF inp=3 THEN
      Converter.ChainToDist(d,dist);
      Distributions.SaveMatrix(dist);
    ELSIF inp=4 THEN
      Chains.Save(d);
    ELSIF inp=5 THEN
      Converter.ChainToDist(d,dist);
      Distributions.Round(dist,0.0001);
      Converter.DistToChain(dist,d);
    ELSIF NOT raster AND (inp=6) THEN
      raster:=TRUE;
      InOut.WriteString("raster activated: ");
      InOut.WriteLn;
      InOut.WriteLn;
    END;
  UNTIL inp=0;
END Interactiv;

PROCEDURE WriteProtocol(p:Prot;steps:INTEGER);
  VAR
    i:INTEGER;
BEGIN
  FOR i:=1 TO steps DO
    IF p[i]=Reduce THEN InOut.Write("R");
    ELSIF p[i]=Parity THEN InOut.Write("P");
    ELSE InOut.Write("?");             (* error! *)
    END;
  END;
END WriteProtocol;
```

```
PROCEDURE FindBestProtocol(d:Chains.Chain);
  VAR
    bestD:ARRAY[1..4] OF Chains.Chain;
    bestH:ARRAY[1..4] OF LONGREAL;
    bestP:ARRAY[1..4] OF Prot;
    raster:ARRAY[1..4] OF BOOLEAN;
    n,steps,i,j,m1,m2:INTEGER;
    p1,p2:Prot;
    d1,d2:Chains.Chain;
    errB,errE,hB,hE:LONGREAL;
BEGIN
  InOut.WriteLn;
  InOut.WriteString("find best protocol for given situation :");
  InOut.WriteLn;
  InOut.WriteLn;
  steps:=2;
  bestP[1][1]:=Parity;                (* start with PP,PR,RP,RR *)
  bestP[1][2]:=Parity;
  bestP[2][1]:=Parity;
  bestP[2][2]:=Reduce;
  bestP[3][1]:=Reduce;
  bestP[3][2]:=Parity;
  bestP[4][1]:=Reduce;
  bestP[4][2]:=Reduce;
  FOR i:=1 TO 4 DO
    raster[i]:=FALSE;
    DoProtocol(d,d1,bestP[i][1],raster[i],FALSE);
    DoProtocol(d1,bestD[i],bestP[i][2],raster[i],FALSE);
    Chains.Entropy(bestD[i],errB,errE,hB,hE);
    bestH[i]:=hE-hB;
    WriteProtocol(bestP[i],steps);
    InOut.WriteString(" : entropy difference = ");
    LongRealIO.WriteLongReal(bestH[i],field,digits);
    InOut.WriteLn;
  END;
  REPEAT
    IF (bestH[1]<0.0) AND (bestH[2]<0.0) AND
       (bestH[3]<0.0) AND (bestH[4]<0.0)  THEN
      m1:=1;m2:=2;                    (* Parity is better, if H<0 *)
    ELSE
      IF bestH[1]>bestH[2] THEN m1:=1;m2:=2 ELSE m2:=1;m1:=2 END;
      FOR i:=3 TO 4 DO
        IF bestH[i]>bestH[m1] THEN m2:=m1;m1:=i
        ELSIF bestH[i]>bestH[m2] THEN m2:=i
        END;
      END;
    END;
    d1:=bestD[m1];                    (* m1 and m2 are the best *)
    d2:=bestD[m2];
    p1:=bestP[m1];
    p2:=bestP[m2];
```

```
      bestP[1]:=p1;
      bestP[2]:=p1;
      bestP[3]:=p2;
      bestP[4]:=p2;
      bestD[1]:=d1;
      bestD[2]:=d1;
      bestD[3]:=d2;
      bestD[4]:=d2;
      steps:=steps+1;
      bestP[1][steps]:=Parity;           (* add new protocol step *)
      bestP[2][steps]:=Reduce;
      bestP[3][steps]:=Parity;
      bestP[4][steps]:=Reduce;
      FOR i:=1 TO 4 DO
        d1:=bestD[i];
        DoProtocol(d1,bestD[i],bestP[i][steps],raster[i],TRUE);
        Chains.Entropy(bestD[i],errB,errE,hB,hE);
        bestH[i]:=hE-hB;
        WriteProtocol(bestP[i],steps);
        InOut.WriteString(" : bob = ");
        LongRealIO.WriteLongReal(errB,field,digits);
        InOut.WriteString(" : eve = ");
        LongRealIO.WriteLongReal(errE,field,digits);
        InOut.WriteString(" : entropy difference = ");
        LongRealIO.WriteLongReal(bestH[i],field,digits);
        n:=Chains.NoOfEntry(bestD[i]);
        InOut.WriteString(" : n=");
        InOut.WriteInt(n,field);
        InOut.WriteLn;
        IF NOT raster[i] THEN
          raster[i]:=n>StartRasteringFindBest;
          IF raster[i] THEN
            InOut.WriteString("from now on raster enabled for this protocol.");
            InOut.WriteLn;
          END;
        END;
      END;
    UNTIL steps=20;
  END FindBestProtocol;

BEGIN
  digits:=MathConst.Digits;
  field:=digits+3;
  InOut.WriteLn;
  InOut.WriteLn;
  InOut.WriteLn;
  InOut.WriteString("compute distributions in phi- and epsilon space");
  InOut.WriteLn;
  InOut.WriteString("_____________________________________________");
  InOut.WriteLn;
  InOut.WriteLn;
```

```
  InOut.WriteString("Bit-error-probabilities beta of Bob and gamma of Eve");
  InOut.WriteLn;
  InOut.WriteString("are computed using the distribution of situations");
  InOut.WriteLn;
  InOut.WriteString("for Eve.");
  InOut.WriteLn;
  InOut.WriteLn;
  d:=NIL;
  InOut.WriteString("0=End 1=Load 2=New : ");
  InOut.ReadInt(inp);
  InOut.WriteLn;
  IF inp=1 THEN
    Chains.Load(d);
  ELSIF inp=2 THEN
    InOut.WriteString("error-probability of Alice ea = ");
    LongRealIO.ReadLongReal(ea,done);
    InOut.Read(ch);
    InOut.WriteString("error-probability of Bob   eb = ");
    LongRealIO.ReadLongReal(eb,done);
    InOut.Read(ch);
    InOut.WriteString("error-probability of Eve   ee = ");
    LongRealIO.ReadLongReal(ee,done);
    InOut.Read(ch);
    InOut.WriteLn;
    Converter.EpsToAlpha(ea,eb,ee,a00,a01,a10,a11);
    Chains.Insert(d,a00,a10,a11,a01,0.5,TRUE);
    Chains.Insert(d,a01,a11,a10,a00,0.5,TRUE);
  END;
  IF inp<>0 THEN
    InOut.WriteString("0=END 1=Interactiv  2=Find best protocol : ");
    InOut.ReadInt(inp);
    IF inp=1 THEN
      Interactiv(d);
    ELSIF inp=2 THEN
      FindBestProtocol(d);
    END;
  END;
END Numeric.


DEFINITION MODULE Protocol;

  PROCEDURE Parity(VAR a00,a01,a10,a11:LONGREAL;b00,b01,b10,b11:LONGREAL;
                   VAR p:LONGREAL);
            (* combine two situations aij and bij from phi-space *)
            (* with a parity-step. p equals the probability of   *)
            (* this combination.                                 *)

  PROCEDURE Reduce(VAR a00,a01,a10,a11:LONGREAL;b00,b01,b10,b11:LONGREAL);
            (* combine two situations aij and bij from phi-space *)
            (* with a parity-step.                               *)
```

```
END Protocol.

IMPLEMENTATION MODULE Protocol;

  IMPORT
    Protocol;

  PROCEDURE Parity(VAR a00,a01,a10,a11:LONGREAL;b00,b01,b10,b11:LONGREAL;
                   VAR p:LONGREAL);
    VAR
      p00,p01,p10,p11:LONGREAL;
  BEGIN
    p00:=a00*b00;
    p01:=a01*b01;
    p10:=a10*b10;
    p11:=a11*b11;
    p:=p00+p01+p10+p11;
    a00:=p00/p;                          (* rescale probability to 1 *)
    a01:=p01/p;                          (* because not any combination *)
    a10:=p10/p;                          (* is accepted by Bob *)
    a11:=p11/p;
    p:=p*2.0;                            (* include paritycheck = 1 *)
  END Parity;

  PROCEDURE Reduce(VAR a00,a01,a10,a11:LONGREAL;b00,b01,b10,b11:LONGREAL);
    VAR
      p00,p01,p10,p11:LONGREAL;
  BEGIN
    p00:=a00*b00+a01*b01+a10*b10+a11*b11;
    p01:=a00*b01+a01*b00+a10*b11+a11*b10;
    p10:=a00*b10+a10*b00+a01*b11+a11*b01;
    p11:=a00*b11+a11*b00+a01*b10+a10*b01;
    a00:=p00;
    a01:=p01;
    a10:=p10;
    a11:=p11;
  END Reduce;

END Protocol.

DEFINITION MODULE Converter;

  IMPORT
    Chains,Distributions;

  VAR
    warning:BOOLEAN;  (* to toggle warnings for float-operations *)

  PROCEDURE EpsToAlpha(ea,eb,ee:LONGREAL;VAR a00,a01,a10,a11:LONGREAL);
            (* convert from epsilon-space to alpha-space *)

  PROCEDURE AlphaToEps(a00,a01,a10,a11:LONGREAL;VAR ea,eb,ee:LONGREAL);
```

```
            (* convert from alpha-space to epsilon-space *)

  PROCEDURE ChainToDist(c:Chains.Chain;VAR d:Distributions.Distribution);
            (* round precise values from c in the raster of d *)

  PROCEDURE DistToChain(VAR d:Distributions.Distribution;VAR c:Chains.Chain);
            (* convert just non zero values from d to c *)

END Converter.

IMPLEMENTATION MODULE Converter;

  IMPORT
    LongRealIO,InOut,LongMathLib,MathConst,Chains,Distributions,Converter;

  VAR
    field,digits:INTEGER;

  PROCEDURE EpsToAlpha(ea,eb,ee:LONGREAL;VAR a00,a01,a10,a11:LONGREAL);
    VAR
      da,db,de:LONGREAL;
  BEGIN
    de:=1.0-ee;
    da:=1.0-ea;
    db:=1.0-eb;
    a00:=da*db*de+ea*eb*ee;
    a01:=da*db*ee+ea*eb*de;
    a10:=da*eb*de+ea*db*ee;
    a11:=da*eb*ee+ea*db*de;
  END EpsToAlpha;

  PROCEDURE AlphaToEps(a00,a01,a10,a11:LONGREAL;VAR ea,eb,ee:LONGREAL);
    VAR
      divisor,rad:LONGREAL;
  BEGIN
    divisor:=2.0*(a10+a11-0.5);          (* difficult evaluation *)
    IF ABS(divisor)<MathConst.MachEps THEN (* normally FALSE *)
      IF warning THEN
        InOut.WriteString("divisor near zero : divisor = ");
        LongRealIO.WriteLongReal(divisor,field,digits);
        InOut.WriteString(" corrected : 0.0");
        InOut.WriteLn;
      END;
      ee:=0.5;
      ea:=0.5;
      eb:=0.5;
    ELSE
      rad:=-(a01+a10-0.5)*(a01+a11-0.5)/divisor;
      IF rad<0.0 THEN                    (* theoretically not possible *)
        IF warning THEN
          InOut.WriteString("radikand for ee < 0.0 : radikand = ");
          LongRealIO.WriteLongReal(rad,field,digits);
```

```
    InOut.WriteString(" corrected : 0.0");
    InOut.WriteLn;
  END;
  rad:=0.0
END;
ee:=0.5-LongMathLib.Sqrt(rad);
IF ABS(1.0-2.0*ee)<MathConst.MachEps THEN
  IF warning THEN
    InOut.WriteString("ee ~ 0.5 : ee = ");
    LongRealIO.WriteLongReal(ee,field,digits);
    InOut.WriteString(" ee not changed,but ea,eb calculated with 0.5");
    InOut.WriteLn;
  END;
  rad:=1.0-4.0*a10;
  IF rad<0.0 THEN                  (* theoretically not possible *)
    IF warning THEN
      InOut.WriteString("radikand for ea < 0.0 : radikand = ");
      LongRealIO.WriteLongReal(rad,field,digits);
      InOut.WriteString(" corrected : 0.0");
      InOut.WriteLn;
    END;
    rad:=0.0;
  END;
  ea:=0.5*(1.0-LongMathLib.Sqrt(rad));
  eb:=ea;
ELSE
  ea:=(a01+a11-ee)/(1.0-2.0*ee);
  eb:=(a01+a10-ee)/(1.0-2.0*ee);
END;
IF ea>0.5 THEN
  ea:=1.0-ea                       (* take dual case *)
END;
IF eb>0.5 THEN
  eb:=1.0-eb                       (* take dual case *)
END;
IF ea<0.0 THEN                     (* theoretically not possible *)
  IF warning THEN
    InOut.WriteString("ea < 0.0 : ea = ");
    LongRealIO.WriteLongReal(ea,field,digits);
    InOut.WriteString(" corrected 0.0");
    InOut.WriteLn;
  END;
  ea:=0.0;
END;
IF eb<0.0 THEN                     (* theoretically not possible *)
  IF warning THEN
    InOut.WriteString("eb < 0.0 : eb = ");
    LongRealIO.WriteLongReal(eb,field,digits);
    InOut.WriteString(" corrected 0.0");
    InOut.WriteLn;
  END;
```

```
        eb:=0.0;
      END;
    END;
  END AlphaToEps;

  PROCEDURE ChainToDist(c:Chains.Chain;VAR d:Distributions.Distribution);
    VAR
      ea,eb,ee:LONGREAL;
  BEGIN
    Distributions.Clear(d);
    WHILE c<>NIL DO
      AlphaToEps(c^.p00,c^.p11,c^.p01,c^.p10,ea,eb,ee);
      Distributions.Insert(d,ea,ee,c^.p);
      c:=c^.next;
    END;
  END ChainToDist;

  PROCEDURE DistToChain(VAR d:Distributions.Distribution;VAR c:Chains.Chain);
    VAR
      i,j:INTEGER;
      ea,eb,ee,a00,a01,a10,a11:LONGREAL;
  BEGIN
    c:=NIL;
    FOR i:=0 TO Distributions.N DO
      FOR j:=0 TO Distributions.N DO
        IF d[i,j]<>0.0 THEN              (* include only non zero values *)
          Distributions.CoordToEps(i,j,ee,ea);
          eb:=ea;
          EpsToAlpha(ea,eb,ee,a00,a01,a10,a11);
          Chains.Insert(c,a00,a10,a11,a01,0.5*d[i,j],FALSE);
          Chains.Insert(c,a01,a11,a10,a00,0.5*d[i,j],FALSE);   (* dual *)
        END;
      END;
    END;
  END DistToChain;

BEGIN
  warning:=FALSE;                        (* no warnings by default *)
  digits:=MathConst.Digits;
  field:=digits+3;
END Converter.

DEFINITION MODULE Chains;

  TYPE
    Chain=POINTER TO ChainDesc;
    ChainDesc=RECORD
      p00,p01,p10,p11:LONGREAL;          (* coordinate in phi-space *)
      p:LONGREAL;                        (* probability of given coordinate *)
      next:Chain;
    END;
```

```
  PROCEDURE Deallocate(VAR d:Chain);
            (* remove chain d from memory *)

  PROCEDURE Write(d:Chain);
            (* write content of chain d formatted on default output *)

  PROCEDURE Save(d:Chain);
            (* prompt for filename and save d in file *)

  PROCEDURE Load(VAR d:Chain);
            (* prompt for filename and load d from file *)

  PROCEDURE Rescale(d:Chain);
            (* rescale d so that it sums to 1 *)

  PROCEDURE Insert(VAR d:Chain;p00,p01,p10,p11,p:LONGREAL;unique:BOOLEAN);
            (* unique : if d does not contain such a coordinate, insert *)
            (*          it else add given probability to found coordinate *)
            (* ~unique: insert *)

  PROCEDURE Entropy(d:Chain;VAR errB,errE,hB,hE:LONGREAL);
            (* compute error probability and entropy of Bob and Eve *)

  PROCEDURE CollEntropy(d:Chain;VAR errB,errE,hB,hE:LONGREAL);
            (* compute error probability and collisionentropy of Bob and Eve *)

  PROCEDURE NoOfEntry(d:Chain):INTEGER;
            (* count number of entries in chain d *)

  PROCEDURE SumProbability(d:Chain):LONGREAL;
            (* sum up the probabilities in chain d *)

END Chains.

IMPLEMENTATION MODULE Chains;

  IMPORT
    Chains,Storage,InOut,LongRealIO,MathConst,LongMathLib;

  VAR
    field,digits:INTEGER;

  PROCEDURE Deallocate(VAR d:Chain);
    VAR
      d1:Chain;
  BEGIN
    WHILE d<>NIL DO
      d1:=d;
      d:=d^.next;
      Storage.DEALLOCATE(d1,SIZE(ChainDesc));
    END;
  END Deallocate;
```

```
PROCEDURE Write(d:Chain);
BEGIN
  WHILE d<>NIL DO
    InOut.WriteString("p00=");
    LongRealIO.WriteLongReal(d^.p00,field,digits);
    InOut.WriteString("  p01=");
    LongRealIO.WriteLongReal(d^.p01,field,digits);
    InOut.WriteString("  p10=");
    LongRealIO.WriteLongReal(d^.p10,field,digits);
    InOut.WriteString("  p11=");
    LongRealIO.WriteLongReal(d^.p11,field,digits);
    InOut.WriteString("  p=");
    LongRealIO.WriteLongReal(d^.p,field,digits);
    InOut.WriteLn;
    d:=d^.next;
  END;
  InOut.WriteLn;
END Write;

PROCEDURE Save(d:Chain);
BEGIN
  InOut.OpenOutput("Chain");
  IF InOut.Done THEN
    WHILE d<>NIL DO
      LongRealIO.WriteLongReal(d^.p00,25,-16);
      LongRealIO.WriteLongReal(d^.p01,25,-16);
      LongRealIO.WriteLongReal(d^.p10,25,-16);
      LongRealIO.WriteLongReal(d^.p11,25,-16);
      LongRealIO.WriteLongReal(d^.p,25,-16);
      d:=d^.next;
    END;
    InOut.WriteLn;
    InOut.CloseOutput;
  ELSE
    InOut.WriteLn;
    InOut.WriteString(" save error");
  END;
  InOut.WriteLn;
END Save;

PROCEDURE Load(VAR d:Chain);
  VAR
    done:BOOLEAN;
    p00,p01,p10,p11,p:LONGREAL;
BEGIN
  InOut.OpenInput("Chain");
  IF InOut.Done THEN
    Deallocate(d);
    d:=NIL;
    LongRealIO.ReadLongReal(p00,done);
```

```
    WHILE done DO
      LongRealIO.ReadLongReal(p01,done);
      LongRealIO.ReadLongReal(p10,done);
      LongRealIO.ReadLongReal(p11,done);
      LongRealIO.ReadLongReal(p,done);
      Insert(d,p00,p01,p10,p11,p,FALSE);
      LongRealIO.ReadLongReal(p00,done);
    END;
    InOut.CloseInput;
  ELSE
    InOut.WriteLn;
    InOut.WriteString(" not found");
  END;
  InOut.WriteLn;
END Load;

PROCEDURE Rescale(d:Chain);
  VAR
    sum:LONGREAL;
    p:Chain;
BEGIN
  sum:=0.0;
  p:=d;
  WHILE p<>NIL DO
    sum:=sum+p^.p;
    p:=p^.next;
  END;
  p:=d;
  WHILE p<>NIL DO
    p^.p:=p^.p/sum;
    p:=p^.next;
  END;
END Rescale;

PROCEDURE Find(VAR d:Chain;p00,p01,p10,p11:LONGREAL);

  PROCEDURE Eq(a,b:LONGREAL):BOOLEAN;
  BEGIN
    RETURN ABS(a-b)<=ABS(a)*MathConst.MachEps; (* probably to strict *)
  END Eq;

BEGIN
  WHILE (d<>NIL) AND (NOT Eq(p00,d^.p00) OR NOT Eq(p01,d^.p01) OR
                      NOT Eq(p10,d^.p10) OR NOT Eq(p11,d^.p11)) DO
    d:=d^.next;
  END;
END Find;

PROCEDURE Insert(VAR d:Chain;p00,p01,p10,p11,p:LONGREAL;unique:BOOLEAN);
  VAR
    d1:Chain;
```

```
BEGIN
  d1:=d;
  IF unique THEN Find(d1,p00,p01,p10,p11) END;
  IF NOT unique OR (d1=NIL) THEN             (* insert new, if not found *)
    Storage.ALLOCATE(d1,SIZE(ChainDesc));
    IF d1=NIL THEN
      InOut.WriteString("out of memory");
      InOut.WriteLn;
    ELSE
      d1^.p00:=p00;
      d1^.p01:=p01;
      d1^.p10:=p10;
      d1^.p11:=p11;
      d1^.p:=p;
      d1^.next:=d;
      d:=d1;
    END;
  ELSE
    d1^.p:=d1^.p+p;                    (* just add probability *)
  END;
END Insert;

PROCEDURE Entropy(d:Chain;VAR errB,errE,hB,hE:LONGREAL);
  VAR
    sum:LONGREAL;
BEGIN
  sum:=0.0;
  errB:=0.0;
  errE:=0.0;
  WHILE d<>NIL DO
    errB:=errB+(d^.p10+d^.p01)*d^.p;
    IF d^.p01+d^.p00>d^.p10+d^.p11 THEN (* Eve's majority decision *)
      errE:=errE+(d^.p10+d^.p11)*d^.p;
    ELSE
      errE:=errE+(d^.p01+d^.p00)*d^.p;
    END;
    sum:=sum+d^.p;
    d:=d^.next;
  END;
  errB:=errB/sum;
  errE:=errE/sum;
  hE:=-errE*LongMathLib.Ln(errE)-(1.0-errE)*LongMathLib.Ln(1.0-errE);
  hB:=-errB*LongMathLib.Ln(errB)-(1.0-errB)*LongMathLib.Ln(1.0-errB);
  hE:=hE/LongMathLib.Ln(2.0);          (* use dual logarithms *)
  hB:=hB/LongMathLib.Ln(2.0);
END Entropy;

PROCEDURE CollEntropy(d:Chain;VAR errB,errE,hB,hE:LONGREAL);
  VAR
    sum:LONGREAL;
BEGIN
```

```
    sum:=0.0;
    errB:=0.0;
    errE:=0.0;
    WHILE d<>NIL DO
      errB:=errB+(d^.p10+d^.p01)*d^.p;
      IF d^.p01+d^.p00>d^.p10+d^.p11 THEN (* Eve's majority decision *)
        errE:=errE+(d^.p10+d^.p11)*d^.p;
      ELSE
        errE:=errE+(d^.p01+d^.p00)*d^.p;
      END;
      sum:=sum+d^.p;
      d:=d^.next;
    END;
    errB:=errB/sum;
    errE:=errE/sum;
    hE:=-LongMathLib.Ln(errE*errE+(1.0-errE)*(1.0-errE));
    hB:=-LongMathLib.Ln(errB*errB+(1.0-errB)*(1.0-errB));
    hE:=hE/LongMathLib.Ln(2.0);          (* use dual logarithms *)
    hB:=hB/LongMathLib.Ln(2.0);
  END CollEntropy;

  PROCEDURE NoOfEntry(d:Chain):INTEGER;
    VAR
      n:INTEGER;
  BEGIN
    n:=0;
    WHILE d<>NIL DO
      n:=n+1;
      d:=d^.next;
    END;
    RETURN n;
  END NoOfEntry;

  PROCEDURE SumProbability(d:Chain):LONGREAL;
    VAR
      sum:LONGREAL;
  BEGIN
    sum:=0.0;
    WHILE d<>NIL DO
      sum:=sum+d^.p;
      d:=d^.next;
    END;
    RETURN sum;
  END SumProbability;

BEGIN
  digits:=MathConst.Digits;
  field:=digits+3;
END Chains.

DEFINITION MODULE Distributions;
```

```
  CONST
    N=200;
    DFact=FLOAT(N)*2.0;

  TYPE
    Distribution=ARRAY[0..N] OF ARRAY[0..N] OF LONGREAL;

  VAR
    warning:BOOLEAN;  (* to toggle warnings for float-operations *)

  PROCEDURE Clear(VAR d:Distribution);
            (* fill distribution d with zero *)

  PROCEDURE Write(VAR d:Distribution);
            (* write content of distribution d formatted on default output *)

  PROCEDURE SaveMatrix(VAR d:Distribution);
            (* promt for filename and save d in file *)

  PROCEDURE Rescale(VAR d:Distribution);
            (* rescale d so that it sums to 1 *)

  PROCEDURE Round(VAR d:Distribution;min:LONGREAL);
            (* ronnd values in d, so that no value is smaller than min *)

  PROCEDURE Insert(VAR d:Distribution;ea,ee,p:LONGREAL);
            (* insert probability p at coordinate (ee,ea) *)
            (* round on the safe side if needed *)

  PROCEDURE Entropy(VAR d:Distribution;VAR errB,errE,hB,hE:LONGREAL);
            (* compute error probability and entropy of Bob and Eve *)

  PROCEDURE SumProbability(VAR d:Distribution):LONGREAL;
            (* sum up the probabilities in chain d *)

  PROCEDURE EpsToCoord(ee,ea:LONGREAL;VAR n,m:INTEGER);
            (* convert real coordinates ea and ee to integer *)
            (* coordinates n,m. Round on the save side if needed *)
            (* and perform rangecheck, if n or m exceeds dimensions *)

  PROCEDURE CoordToEps(n,m:INTEGER;VAR ee,ea:LONGREAL);
            (* convert coordinates n and m to ea end ee *)

END Distributions.

IMPLEMENTATION MODULE Distributions;

  IMPORT
    InOut,LongRealIO,Distributions,MathConst,LongMathLib;

  VAR
    field,digits:INTEGER;
```

```
  fiarr,diarr:INTEGER;

PROCEDURE Clear(VAR d:Distribution);
  VAR
    i,j:INTEGER;
BEGIN
  FOR i:=0 TO N DO
    FOR j:=0 TO N DO
      d[i,j]:=0.0;
    END;
  END;
END Clear;

PROCEDURE Write(VAR d:Distribution);
  VAR
    i,j,k:INTEGER;
BEGIN
  FOR j:=1 TO fiarr DO InOut.Write(' ') END;
  FOR j:=0 TO N DO                      (* write ea axis *)
    IF j<=N THEN
      LongRealIO.WriteLongReal(FLOAT(j)/DFact,fiarr,diarr);
    END;
  END;
  InOut.WriteLn;
  FOR i:=0 TO N DO
    LongRealIO.WriteLongReal(FLOAT(i)/DFact,fiarr,diarr); (* write ee axis *)
    FOR j:=0 TO N DO
      IF d[i,j]=0.0 THEN                (* don't write zeros *)
        FOR k:=1 TO fiarr DO InOut.Write(' ') END;
      ELSE
        LongRealIO.WriteLongReal(d[i,j],fiarr,diarr);
      END;
    END;
    InOut.WriteLn;
  END;
END Write;

PROCEDURE SaveMatrix(VAR d:Distribution);
  VAR
    i,j:INTEGER;
BEGIN
  InOut.OpenOutput("Dat");
  IF InOut.Done THEN
    FOR i:=0 TO N DO
      FOR j:=0 TO N DO
        LongRealIO.WriteLongReal(d[i,j],11,8);
      END;
      InOut.WriteLn
    END;
    InOut.CloseOutput;
  ELSE
```

```
    InOut.WriteLn;
    InOut.WriteString(" save error");
  END;
  InOut.WriteLn;
END SaveMatrix;

PROCEDURE Rescale(VAR d:Distribution);
  VAR
    i,j:INTEGER;
    sum:LONGREAL;
BEGIN
  sum:=0.0;
  FOR i:=0 TO N DO
    FOR j:=0 TO N DO
      sum:=sum+d[i,j];
    END;
  END;
  FOR i:=0 TO N DO
    FOR j:=0 TO N DO
      d[i,j]:=d[i,j]/sum;
    END;
  END;
END Rescale;

PROCEDURE Round(VAR d:Distribution;min:LONGREAL);
  VAR
    i,j:INTEGER;
BEGIN
  FOR i:=0 TO N DO
    FOR j:=N TO 1 BY -1 DO
      IF d[j,i]<min THEN
        d[j-1,i]:=d[j-1,i]+d[j,i];
        d[j,i]:=0.0;
      END;
    END;
  END;
  FOR i:=0 TO N DO
    FOR j:=0 TO N-1 DO
      IF d[i,j]<min THEN
        d[i,j+1]:=d[i,j+1]+d[i,j];
        d[i,j]:=0.0;
      END;
    END;
  END;
END Round;

PROCEDURE Insert(VAR d:Distribution;ea,ee,p:LONGREAL);
  VAR
    n,m:INTEGER;
BEGIN
  EpsToCoord(ee,ea,n,m);              (* convert safely *)
```

```
  d[n,m]:=d[n,m]+p;
END Insert;

PROCEDURE Entropy(VAR d:Distribution;VAR errB,errE,hB,hE:LONGREAL);
  VAR
    i,j:INTEGER;
    ea,ee:LONGREAL;
BEGIN
  errB:=0.0;
  errE:=0.0;
  FOR i:=0 TO N DO
    FOR j:=0 TO N DO
      CoordToEps(i,j,ee,ea);
      errE:=errE+(ea*(1.0-ee)+(1.0-ea)*ee)*d[i,j];
      errB:=errB+(ea*(1.0-ea)+(1.0-ea)*ea)*d[i,j];
    END;
  END;
  hE:=-errE*LongMathLib.Ln(errE)-(1.0-errE)*LongMathLib.Ln(1.0-errE);
  hB:=-errB*LongMathLib.Ln(errB)-(1.0-errB)*LongMathLib.Ln(1.0-errB);
  hE:=hE/LongMathLib.Ln(2.0);          (* use dual logarithm *)
  hB:=hB/LongMathLib.Ln(2.0);
END Entropy;

PROCEDURE SumProbability(VAR d:Distribution):LONGREAL;
  VAR
    sum:LONGREAL;
    i,j:INTEGER;
BEGIN
  sum:=0.0;
  FOR i:=0 TO N DO
    FOR j:=0 TO N DO
      sum:=sum+d[i,j];
    END;
  END;
  RETURN sum;
END SumProbability;

PROCEDURE EpsToCoord(ee,ea:LONGREAL;VAR n,m:INTEGER);
BEGIN
  n:=LongMathLib.Entier(ee*DFact);     (* round on the safe side *)
  m:=-LongMathLib.Entier(-ea*DFact);
  IF (m>N) THEN                        (* rounding exceeds dimension *)
    IF warning THEN
      InOut.WriteString("Rangecheck m=");
      InOut.WriteInt(m,field);
      InOut.WriteString(" corrected : m=");
      InOut.WriteInt(N,field);
      InOut.WriteLn;
    END;
    m:=N;
  END;
```

```
    IF (n>N) THEN
      IF warning THEN
        InOut.WriteString("Rangecheck n=");
        InOut.WriteInt(n,field);
        InOut.WriteString(" corrected : n=");
        InOut.WriteInt(N,field);
        InOut.WriteLn;
      END;
      n:=N;
    END;
    IF (m<0) THEN
      IF warning THEN
        InOut.WriteString("Rangecheck m=");
        InOut.WriteInt(m,field);
        InOut.WriteString(" corrected : m=");
        InOut.WriteInt(0,field);
        InOut.WriteLn;
      END;
      m:=0;
    END;
    IF (n<0) THEN
      IF warning THEN
        InOut.WriteString("Rangecheck n=");
        InOut.WriteInt(n,field);
        InOut.WriteString(" corrected : n=");
        InOut.WriteInt(0,field);
        InOut.WriteLn;
      END;
      n:=0;
    END;
  END EpsToCoord;

  PROCEDURE CoordToEps(n,m:INTEGER;VAR ee,ea:LONGREAL);
  BEGIN
    ee:=FLOAT(n)/DFact;
    ea:=FLOAT(m)/DFact;
  END CoordToEps;

BEGIN
  warning:=FALSE;                       (* no warnings by default *)
  digits:=MathConst.Digits;
  field:=digits+3;
  diarr:=2;                             (* formatted output with 2 digits *)
  fiarr:=diarr+3;
END Distributions.

DEFINITION MODULE MathConst;

  CONST
    Digits=5;                           (* digits for float-output *)

  VAR
```

```
    MachEps:LONGREAL;                    (* 1.0 + MachEps / 2 = 1.0 *)

END MathConst.

IMPLEMENTATION MODULE MathConst;

BEGIN
  MachEps:=1.0;                          (* compute macheps *)
  WHILE 1.0-MachEps<>1.0 DO MachEps:=MachEps/2.0 END;
  MachEps:=MachEps*2.0;
END MathConst.
```

B.4 Distanzprofil

```
function [set,coset]=distanceprofile(p)

% DISTANCEPROFILE show distacedistribution between codewords
%         DISTANCEPROFILE(p) computes the distanceprofile
%         of the linear code generated by the protocolvector p
%         containing 1 for parity- and 0 for reducesteps.
%         Results are the distanceprofiles of the set and the
%         coset after the protocol p. The results are also ploted.
%
% example >> [set,coset]=distanceprofile([1 0 1 0 1])
%
%         set =
%
%              1     8    60   184   518   184    60     8     1
%              0     4     8    12    16    20    24    28    32
%
%
%         coset =
%
%             64   256   384   256    64
%              8    12    16    20    24

set=[1 0];
coset=[0 1];
set=condens(set);
coset=condens(coset);
[m,n]=size(p);
for i=1:n,
  if p(1,n-i+1)==0,
    coset=reduce(coset,2^(i-1));
    set=reduce(set,2^(i-1));
  else
    coset=parity(coset);
    set=parity(set);
  end;
end;
```

```
plot(set(2,:),set(1,:),'x',coset(2,:),coset(1,:),'o');
xlabel('distance');
ylabel('number of codewords');


function [p]=parity(d)

% PARITY  compute new codeword distances after a paritystep
%         PARITY(d,s) computes for the distanceprofile d
%         the new distanceprofile after a paritystep.
% example >> d=[1 1; 0 2]
%
%         d =
%
%              1     1       % no of codewords
%              0     2       % and their correspondent distance from 0
%
%         >> parity(d)
%
%         ans =
%
%              1     1
%              0     4
%

[m,n]=size(d);
p=d;
for i=1:n,
  p(2,i)=p(2,i)*2;
end;

function [newd]=reduce(d,s)

% REDUCE  compute new codeword distances after a reducestep.
%         REDUCE(d,s) computes for the distanceprofile d
%         the new distanceprofile after a reducestep.
%         s must be the length of the old codewords.
%
% example >> d=[1 1; 0 2]
%
%         d =
%
%              1     1       % no of codewords
%              0     2       % and their correspondent distance from 0
%
%         >> reduce(d,2)
%
%         ans =
%
%              1     6     1
%              0     2     4
%
```

```
r=d(1,:).*power2(d)*pascal(d,s);
[m,n]=size(r);
newd=zeros(2,2*n-1);
for i=1:n,
  newd(1,i)=r(1,i);
  newd(1,2*n-i)=r(1,i);
end;
newd=condens(newd);

function [p]=pascal(d,s)

% PASCAL Used by REDUCE, must not be used allone.
%        PASCAL(d,s) computes for the distanceprofile d the
%        corresponding pascalmatrix. Only needed rows are included
%        and the right halve of the matrix is omitted.
%        s must be the length of the old codewords.

[m,n]=size(d);
s=s+1;
p=zeros(n,s);
q1=zeros(1,s);
q1(1,s)=1;
k=n;
if d(2,k)==s-1,
  p(k,:)=q1;
  k=k-1;
end;
for i=s-1:-1:d(2,1)+1,
  q2=zeros(1,s);                    % new row
  q2(1,i)=1;
  for j=i+2:2:s,
    if j+1>s,
      q2(1,j)=2*q1(1,j-1);
    else
      q2(1,j)=q1(1,j-1)+q1(1,j+1);
    end
  end
  q1=q2;
  if d(2,k)==i-1,                   % include row, if needed
    p(k,:)=q1;
    k=k-1;
  end;
end;

function [r]=power2(d)

% POWER2 Used by REDUCE, must not be used allone.
%        compute powers of 2 for a distanceprofile.

[m,n]=size(d);
r=zeros(1,n);
```

```
for i=1:n,
  r(1,i)=2^(d(2,i));
end;

function [newd]=condens(d)

% CONDENS convert a fullsize distanceprofile in a compact format
%         CONDENS(d) removes all zero values in the full size
%         distanceprofile d and puts the distance in a second row.
%
% example >> condens([1 0 6 0 1])  % distance corresponds row-index
%
%         ans =
%
%              1     6     1
%              0     2     4       % explicit distance

[m,n]=size(d);
m=0;
for i=1:n,
  if d(1,i)>=1,
    m=m+1;
  end;
end;
newd=zeros(2,m);
m=1;
for i=1:n,
  if d(1,i)>=1,
    newd(1,m)=d(1,i);
    newd(2,m)=i-1;
    m=m+1;
  end;
end;

function [r]=numberofcodewords(p)

% NUMBEROFCODEWORDS compute total number of codewords
%         NUMBEROFCODWORDS(p) computes the number of codewords
%         of the code that corresponds to the protocol p
%         (1=paritystep, 0=reducestep). A plot shows how the
%         number grows during the protocol. Minimal and maximal
%         growth are also shown.
%
% example >> numberofcodewords([0 1 0 1])
%
%         ans =
%
%                 2048

[m,n]=size(p);
n1=zeros(1,n+1);
n2=zeros(1,n+1);
```

```
n3=zeros(1,n+1);
c=zeros(1,n+1);
m=1;
n1(1,1)=2;
n2(1,1)=2;
n3(1,1)=2;
for i=1:n,
  c(1,i+1)=i;
  n2(1,i+1)=2^m*n2(1,i);
  n3(1,i+1)=2;
  if p(1,n-i+1)==1,
    n1(1,i+1)=n1(1,i);
    m=2*m;
  else
    n1(1,i+1)=2^m*n1(1,i);
    m=2*m;
  end;
end;
axis([0 n 0 log(n1(1,n+1))]);
semilogy(c,n1,'-',c,n2,'-.',c,n3,':');
xlabel('number of protocolsteps');
ylabel('number of codewords');
r=n1(1,n+1);
```

B.5 Berechnung der Generatormatrix

```
function [G]=generator(d)

% GENERATOR compute the generator-matrix for a given protocol
%         GENERATOR(d) computes the generator-matrix of
%         the linear code generated by the protocolvector p
%         containing 1 for parity- and 0 for reducesteps.
%
% example >> G=generator([1 0 1])
%
%          G =
%
%               1     1     1     1     0     0     0     0
%               0     0     0     0     1     1     1     1
%               1     1     0     0     1     1     0     0
%

[m,n]=size(d);

G=[1];
for i=0:n-1,
  if d(1,n-i)==1,
    G=G*D(2^i);
  else
    G=[D(2^i);G*E(2^i)];
  end;
```

```
end;

function [D]=D(n)

% D used by generator

D=zeros(n,2*n);
for i=1:n,
  D(i,2*i-1)=1;
  D(i,2*i)=1;
end;

function [E]=E(n)

% E used by generator

E=zeros(n,2*n);
for i=1:n,
  E(i,2*i-1)=1;
end;
```

B.6 Erzeugter Linearer Code

```
MODULE Theory;

  IMPORT
    InOut,LongRealIO;

  CONST
    WordLength=8;                        (* maximal length of a codeword *)
    CodeLength=256;                      (* must be 2^WordLength *)

  TYPE
    CodeWord=ARRAY[1..WordLength] OF INTEGER;
    Code=ARRAY[1..CodeLength] OF CodeWord;

  VAR
    set,coset:Code;
    n,m,i,j:INTEGER;
    done:BOOLEAN;
    ea,eb,ee,da,db,de,gamma,beta:LONGREAL;
    a:ARRAY[0..1] OF ARRAY [0..1] OF ARRAY [0..1] OF LONGREAL;
    protocol:CodeWord;
    ch:CHAR;

  PROCEDURE WriteCodeWord(w:CodeWord;m:INTEGER);
    VAR
      i:INTEGER;
  BEGIN
    FOR i:=1 TO m DO
      InOut.WriteInt(w[i],2);
```

```
  END;
END WriteCodeWord;

PROCEDURE WriteCode(VAR c:Code;n,m:INTEGER);
  VAR
    i:INTEGER;
BEGIN
  FOR i:=1 TO n DIV 2 DO
    WriteCodeWord(c[i],m);
    InOut.WriteLn;
  END;
END WriteCode;

PROCEDURE Zero(VAR z:CodeWord);        (* set codeword z to zero *)
  VAR
    i:INTEGER;
BEGIN
  FOR i:=1 TO WordLength DO z[i]:=0 END;
END Zero;

PROCEDURE Increment(VAR z:CodeWord);
  VAR
    i:INTEGER;
BEGIN
  i:=1;
  z[i]:=z[i]+1;
  WHILE (i<WordLength) AND (z[i]>1) DO
    z[i]:=0;
    i:=i+1;
    z[i]:=z[i]+1;
  END;
END Increment;

PROCEDURE Power2(m:INTEGER):INTEGER;   (* compute 2^m, m>=0  *)
  VAR
    p:INTEGER;
BEGIN
  p:=1;
  WHILE m>0 DO p:=p*2;m:=m-1 END;
  RETURN p;
END Power2;

PROCEDURE ParityStep(VAR s,c:Code;VAR n,m:INTEGER);
  VAR
    w:CodeWord;
    i:INTEGER;
    j,k:INTEGER;
BEGIN
  FOR i:=1 TO n DIV 2 DO
    k:=1;
    FOR j:=1 TO m DO                   (* double zeros and ones *)
```

```
      w[k]:=c[i][j];
      w[k+1]:=c[i][j];
      k:=k+2;
    END;
    c[i]:=w;
    k:=1;
    FOR j:=1 TO m DO
      w[k]:=s[i][j];                    (* double zeros and ones *)
      w[k+1]:=s[i][j];
      k:=k+2;
    END;
    s[i]:=w;
  END;
  m:=m*2;
END ParityStep;

PROCEDURE ReduceStep(VAR s,c:Code;VAR n,m:INTEGER);
  VAR
    i,i1,i2:INTEGER;
    j,k:INTEGER;
    z:CodeWord;
    ws,wc:ARRAY[0..1] OF CodeWord;     (* base to construct new s and c *)
    ns,nc:Code;
BEGIN
  i2:=0;
  FOR i:=1 TO n DIV 2 DO
    k:=1;
    FOR j:=1 TO m DO
      IF s[i][j]=0 THEN
        ws[0][k]:=0;ws[0][k+1]:=0;     (* 0 comes from 00 or 11 *)
        ws[1][k]:=1;ws[1][k+1]:=1;
      ELSE
        ws[0][k]:=1;ws[0][k+1]:=0;     (* 1 comes from 10 or 01 *)
        ws[1][k]:=0;ws[1][k+1]:=1;
      END;
      IF c[i][j]=0 THEN
        wc[0][k]:=0;wc[0][k+1]:=0;
        wc[1][k]:=1;wc[1][k+1]:=1;
      ELSE
        wc[0][k]:=1;wc[0][k+1]:=0;
        wc[1][k]:=0;wc[1][k+1]:=1;
      END;
      k:=k+2;
    END;
    Zero(z);
    FOR i1:=0 TO Power2(m)-1 DO        (* take any combination of ws[0] *)
      k:=1;                            (* and ws[1] for the new s *)
      FOR j:=1 TO m DO                 (* do the same with c *)
        ns[i2+i1+1][k]:=ws[z[j]][k];
        ns[i2+i1+1][k+1]:=ws[z[j]][k+1];
        nc[i2+i1+1][k]:=wc[z[j]][k];
```

```
          nc[i2+i1+1][k+1]:=wc[z[j]][k+1];
          k:=k+2;
        END;
        Increment(z);
      END;
      i2:=i2+Power2(m);
    END;
    n:=2*i2;
    c:=nc;
    s:=ns;
    m:=2*m;
  END ReduceStep;

  PROCEDURE CalcErrorProbability(VAR s,c:Code;n,m:INTEGER;VAR gamma:LONGREAL);
    VAR
      i,j,k,l:INTEGER;
      ps,pc,prod1,prod2,prod3,prod4,paccept:LONGREAL;
      z:CodeWord;
  BEGIN
    gamma:=0.0;
    paccept:=0.0;
    Zero(z);
    FOR i:=0 TO Power2(m)-1 DO
      ps:=0.0;                          (* probability, that word of Alice *)
      pc:=0.0;                          (* is in set respectively in coset *)
      FOR j:=1 TO n DIV 2 DO
        FOR k:=1 TO n DIV 2 DO
          prod1:=1.0;
          prod2:=1.0;
          prod3:=1.0;
          prod4:=1.0;
          FOR l:=1 TO m DO
            prod1:=prod1*a[s[j][l]][s[k][l]][z[l]];
            prod2:=prod2*a[s[j][l]][c[k][l]][z[l]];
            prod3:=prod3*a[c[j][l]][s[k][l]][z[l]];
            prod4:=prod4*a[c[j][l]][c[k][l]][z[l]];
          END;
          ps:=ps+prod1+prod2;
          pc:=pc+prod3+prod4;
        END;
      END;
      IF pc>ps THEN gamma:=gamma+ps ELSE gamma:=gamma+pc END;
      paccept:=paccept+ps+pc;
      Increment(z);
    END;
    gamma:=gamma/paccept;               (* error-probability of Eve *)
  END CalcErrorProbability;

BEGIN
  InOut.WriteLn;
  InOut.WriteLn;
```

```
InOut.WriteString("Calculation of the linear code generated by a protocol");
InOut.WriteLn;
InOut.WriteString("_____________________________________________________");
InOut.WriteLn;
InOut.WriteLn;
InOut.WriteString("Bit-error-probabilities beta of Bob and gamma of Eve");
InOut.WriteLn;
InOut.WriteString("are calculated using the linear code of a protocol.");
InOut.WriteLn;
InOut.WriteLn;
InOut.WriteString("Caution: protocols of more than 3 steps need more");
InOut.WriteLn;
InOut.WriteString("         memory. Change constant CodeSize !");
InOut.WriteLn;
InOut.WriteLn;
InOut.WriteString("error-probability of Alice ea = ");
LongRealIO.ReadLongReal(ea,done);
InOut.Read(ch);
InOut.WriteString("error-probability of Bob   eb = ");
LongRealIO.ReadLongReal(eb,done);
InOut.Read(ch);
InOut.WriteString("error-probability of Eve   ee = ");
LongRealIO.ReadLongReal(ee,done);
InOut.Read(ch);
InOut.WriteLn;
da:=1.0-ea;
db:=1.0-eb;
de:=1.0-ee;
a[0][0][0]:=0.5*(ea*eb*ee+da*db*de);
a[0][0][1]:=0.5*(ea*eb*de+da*db*ee);
a[0][1][0]:=0.5*(ea*db*ee+da*eb*de);
a[0][1][1]:=0.5*(ea*db*de+da*eb*ee);
a[1][0][0]:=0.5*(da*eb*ee+ea*db*de);
a[1][0][1]:=0.5*(da*eb*de+ea*db*ee);
a[1][1][0]:=0.5*(da*db*ee+ea*eb*de);
a[1][1][1]:=0.5*(da*db*de+ea*eb*ee);
beta:=a[0][1][0]+a[0][1][1]+a[1][0][0]+a[1][0][1];
gamma:=a[0][0][1]+a[0][1][1]+a[1][0][0]+a[1][1][0];
m:=1;                                  (* current codeword-length *)
n:=2;                                  (* current no of codewords *)
set[1][1]:=0;
coset[1][1]:=1;
InOut.WriteString("Start-configuration:");
InOut.WriteLn;
InOut.WriteLn;
Zero(protocol);
i:=0;
REPEAT
  InOut.WriteString("Set:");
  InOut.WriteLn;
  WriteCode(set,n,m);
```

```
    InOut.WriteString("Coset:");
    InOut.WriteLn;
    WriteCode(coset,n,m);
    InOut.WriteLn;
    InOut.WriteString("protocol : ");
    FOR j:=i TO 1 BY -1 DO
      IF protocol[j]=1 THEN
        InOut.WriteString("P");
      ELSE
        InOut.WriteString("R");
      END;
    END;
    InOut.WriteLn;
    InOut.WriteLn;
    InOut.WriteString("beta     =");
    LongRealIO.WriteLongReal(beta,10,5);
    InOut.WriteLn;
    CalcErrorProbability(set,coset,n,m,gamma);
    InOut.WriteString("gamma    =");
    LongRealIO.WriteLongReal(gamma,10,5);
    InOut.WriteLn;
    InOut.WriteLn;
    i:=i+1;
    InOut.WriteString("0=End 1=Parity 2=Reduce  :");
    InOut.ReadInt(protocol[i]);
    InOut.WriteLn;
    IF protocol[i]<>0 THEN
      IF protocol[i]=1 THEN
        ParityStep(set,coset,n,m);
      ELSE
        ReduceStep(set,coset,n,m);
      END;
      beta:=a[0][1][0]+a[0][1][1]+a[1][0][0]+a[1][0][1];
      FOR j:=i TO 1 BY -1 DO
        IF protocol[j]=1 THEN            (* compute Bob's error-probability *)
          beta:=beta*beta/(beta*beta+(1.0-beta)*(1.0-beta));
        ELSE
          beta:=0.5-0.5*(1.0-2.0*beta)*(1.0-2.0*beta);
        END;
      END;
    END;
  UNTIL protocol[i]=0;
END Theory.
```