Dissertation

Modellbildung in der algebraischen Kryptoanalyse

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

(Dr. rer. nat.)

im Fachbereich Mathematik und Naturwissenschaften

der Universität Kassel

vorgelegt von

Frank-Michael Quedenfeld

eingereicht bei

Prof. Dr. Wolfram Koepf

im

Januar 2015

Danksagung

Ich hatte die besondere Möglichkeit in zwei Arbeitsgruppen zu wirken. In den ersten zwei Jahren meiner Promotionszeit war ich am Institut für Mathematik der Universität Kassel tätig. Das dritte Jahr habe ich an der Ruhr-Universität Bochum am Lehrstuhl für Kryptologie und IT-Sicherheit gearbeitet.

Zunächst möchte ich meinem Betreuer Prof. Dr. Wolfram Koepf für die Zusammenarbeit in der Lehre in den Fachbereichen 10 und 16 der Universität Kassel und die Anleitung in der Forschung danken. Vor allem seine Weitsicht und sein Führungsvermögen war zu jedem Zeitpunkt sehr hilfreich.

Bereits im ersten Jahr lernte ich Dr. Enrico Thomae, Dr. Christopher Wolf und Marina Efimenko von der Ruhr-Universität Bochum kennen und wir begannen gemeinsam zu forschen. Diese Kooperation war sehr produktiv und ich möchte mich bei jedem der drei dafür bedanken. Insbesondere Dr. Christopher Wolf war an vielen Diskussionen beteiligt, etwa bei unseren zahlreichen gegenseitigen Besuchen bis zum Ende der Promotion.

Falk Schellenberg (Ruhr-Universität Bochum) danke ich wegen der vielen Diskussionen bei der Entwicklung des Fehler-Modells und Dr. Torsten Sprenger (Universität Kassel) für die Zusammenarbeit bei dem *CloudMath*-Projekt.

Darüber hinaus will ich Prof. Dr. Alexander May und dem gesamten Lehrstuhl für Kryptologie und IT-Sicherheit der Ruhr-Universität Bochum dafür danken, dass sie mich so gut in Ihre Arbeitsgruppe aufgenommen haben. Zudem danke ich Dr. Jennylee Müller, Dr. Bertram Poettering, Matthias Fetzer, Kornelia Fischer, Saqib A. Kakvi und Elena Kirshanova für die vielen Anregungen und Kommentare beim Erstellungsprozess von Publikationen und dieser Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einleitung

Kryptographie

Manche Nachrichten sind so wichtig, dass keiner außer dem Sender und dem Empfänger sie lesen dürfen. Schon Julius Cäsar verschob in geheimen Teilen seiner Nachrichten die Buchstaben im Alphabet um 3 Stellen nach rechts. Also ersetzte er den Buchstaben „a“ durch „d“ und so weiter. In Zeiten des Internets, Handys und moderner Computer ist der Bedarf an Verschlüsselung von Nachrichten größer denn je. Während sich die Codierungstheorie damit beschäftigt, Nachrichten über einen unsicheren beziehungsweise unzuverlässigen Kanal wie das Internet oder Funk zu übertragen, wird in der Kryptographie sicher gestellt, dass Nachrichten nicht von Dritten gelesen oder verändert werden können.

Gerade bei Handys oder WLAN benötigen wir schnelle Algorithmen, die dafür sorgen, dass niemand Drittes Zugriff auf die übertragenen Daten erlangt. Dazu wurden symmetrische Kryptosysteme entwickelt, die eine Nachricht mit Hilfe eines vorher festgelegten geheimen Schlüssels verschlüsseln. In sogenannten Stromchiffren wird eine scheinbar zufällige Folge von Schlüsselstrom generiert und mit dieser Folge die Nachricht verändert. In modernen Blockchiffren hingegen wird die Nachricht in Blöcke gleicher Länge geteilt und mit Hilfe von Vermischung oder Permutation sowie Ersetzung, sogenannter Substitution, verschlüsselt. Diese Verfahren sind sehr schnell, wenn wir einmal den geheimen Schlüssel ausgetauscht haben. Mit asymmetrische Verfahren übertragen oder vereinbaren wir einen geheimen Schlüssel. Danach verschlüsseln wir Nachrichten mit einem schnellen symmetrischen Kryptosystem.

Wenn eine neue symmetrische Chiffre eingeführt wird, muss getestet werden, ob diese sicher ist. Leider gibt es dafür häufig keine Beweisführung. Das liegt einfach daran, dass wir nicht wissen, welche Angriffe in Zukunft entwickelt werden. Die einzige Aussage, die wir treffen können, ist, dass wir sicher gegen die Angriffe sind, die es bereits gibt.

Traditionelle Kryptoanalyse

Die traditionellen Angriffe auf symmetrische kryptographische Primitive sind lineare, differentielle und algebraische Angriffe. Lineare Angriffen wurden in [Mat94] eingeführt. Dabei benutzen wir lineare Zusammenhänge zwischen Klartext-Geheimtext-Paaren und

dem geheimen Schlüssel, die mit einer gewissen Wahrscheinlichkeit gelten, um Informationen über den Schlüssel zu erhalten. Wenn wir Zusammenhänge untersuchen, die entstehen, wenn wir nur kleine Änderungen in einem Klartext-Geheimtext-Paar vornehmen, führen wir einen differentiellen Angriff aus, wie zuerst in [BS91] zu lesen war.

Diese statistischen Angriffe waren sehr erfolgreich gegen symmetrische Blockchiffren. Es werden mittlerweile immer Gegenmaßnahmen getroffen, wenn eine neue Chiffre eingeführt wird.

In dieser Arbeit beschäftigen wir uns mit algebraischer Kryptoanalyse. Da eine Chiffre ein algebraisches Objekt mit klar definierten Abbildungen zum Ver- und Entschlüsseln ist, können wir diese Abbildung polynomiell darstellen. Wir stellen zunächst ein Kryptosystem durch ein nichtlineares multivariates Gleichungssystem über einem endlichen Körper dar. Danach versuchen wir, dieses System mit Hilfe geeigneter Algorithmen so zu vereinfachen, dass wir mit einigen Paaren von Klartext und Geheimtext sofort auf den Schlüssel schließen können.

Algebraische Angriffe teilen leider nicht den Erfolg der anderen traditionellen Angriffe. Das liegt daran, dass bereits das Lösen eines multivariaten, quadratischen, polynomiellen Gleichungssystems über einem endlichen Körper Teil des schweren MQ-Problems ist. Es gibt keinen Algorithmus, der ein beliebiges quadratisches Gleichungssystem mit mehreren Variablen in polynomieller Zeit löst, wie in [GJ90] beschrieben ist.

Allerdings haben Gleichungssysteme, die durch die Modellierung von symmetrischen kryptographischen Primitiven entstehen, viel Struktur. Es gibt mehrere algebraische Formulierungen für eine Chiffre. Ein geeignetes polynomielles Gleichungssystem zu wählen, das mit dem benutzten Algorithmus zum Lösen von Gleichungssystemen abgestimmt ist, entscheidet maßgeblich über den Erfolg eines algebraischen Angriffs.

Neue Angriffe auf symmetrische Kryptosysteme

In [DS09] wurde eine Variante eines differenziellen Angriffs hoher Ordnung vorgestellt. Diesen nannten die Autoren *Cube*-Angriff, weil sie eine Summe über einen n-dimensionalen Würfel bilden. Gerade bei modernen Stromchiffren war dieser Angriff erfolgreich. So brach der Angriff Grain-128 und eine reduzierte Variante von Trivium, wie in [DAS11, FV13] dargestellt wird.

Cube-Angriffe sind ein gutes Beispiel einer Variante eines traditionellen Angriffs, der neu durchdacht wurde. Genau das tun wir in dieser Arbeit mit algebraischen Angriffen.

In der heutigen Zeit müssen wir nicht nur die Kommunikation zwischen zwei stationären Computern schützen. Auch Handys, Smartcards oder Fernbedienungen für Autos gilt es zu sichern. Der Unterschied zu stationären Computern ist, dass diese Geräte Verschlüsselungsverfahren fordern, die effizient auf Hardware umsetzbar sind, und dass diese Geräte gestohlen werden können. Damit haben wir das Gerät während des Angriffs in unserem Besitz und können eine sogenannte Fehler-Analyse durchführen. Eingeführt wurden Fehler-Analysen zuerst auf asymmetrischen Kryptoverfahren in [BDL97]. Danach stellten die Autoren von [BS97] differentielle Fehler-Analysen auf symmetrische Chiffren vor. Mit dem Zugang zum Gerät messen wir Energieverbrauch und Datenstrom. Darüber hinaus injizieren wir Fehler in den internen Zustand der Chiffre. Dies können wir etwa mit einem Laser tun. Die sogenannte *Laser Fault Injection* (LFI) ist

die genaueste Technologie für diese Zwecke. Mit dem Fehler im internen Zustand vereinfachen wir die polynomielle Darstellung oder veranlassen die Chiffre, Informationen über den Schlüssel preiszugeben.

Lösen von polynomiellen Gleichungssystemen

Wie bereits erwähnt ist das Lösen von zufälligen nichtlinearen, multivariaten quadratischen Gleichungssystemen über einem endlichen Körper ein NP-schweres Problem, das sogenannte MQ-Problem (siehe zum Beispiel [GJ90]). Wir kennen also keinen Algorithmus mit polynomieller Laufzeit, der ein solches Gleichungssystem löst. Allerdings können manche Gleichungssysteme, die viel Struktur aufweisen, effizient gelöst werden, wie in [FJ03] gezeigt wurde. Von „schlecht“ konstruierten Chiffren können solche Gleichungssysteme manchmal abgeleitet werden.

Bruno Buchberger hat 1965 in seiner Doktorarbeit [Buc85] Gröbnerbasen, benannt nach Buchbergers Doktorvater Wolfgang Gröbner, vorgestellt. Gröbnerbasen sind eine spezielle Basis eines Ideals, das durch das polynomielle Gleichungssystem erzeugt wird. Durch diese Basis können wir bei einer geeigneten Monomordnung die Lösungen des Systems einfach bestimmen. Gröbnerbasen sind das zur Zeit fortschrittlichste Verfahren zur Lösung multivariater Gleichungssysteme. Sie sind mittlerweile ein Standardwerkzeug in der Kryptoanalyse und der algebraischen Geometrie. Des Weiteren sind sie in vielen Computer-Algebra-Systemen (CAS) wie CoCoA, MAGMA, MATHEMATICA und SAGE implementiert.

Gleichungssysteme für Hardware-orientierte Chiffren sind meistens Gleichungssysteme über dem endlichen Körper $\mathbb{F}_2$, da Operationen über $\mathbb{F}_2$ besonders effizient umsetzbar sind. Dafür gibt es das Paket PolyBoRi aus [BD09] für das CAS SAGE, das spezialisiert auf multivariate Polynome über $\mathbb{F}_2$ ist. Selbst diese spezialisierte Variante eines Gröbnerbasen-Algorithmus kann keine Gleichungssysteme mit einer großen Anzahl von Variablen bearbeiten, weil zu viel Speicher benötigt wird. Wir benötigen daher andere Algorithmen, die solche Gleichungssysteme lösen. Vor Kurzem wurde in [CSSV12] ein Algorithmus mit dem Namen „ElimLin“ eingeführt. Das ist ein spezieller Algorithmus für *quadratische*, dünn besetzte, polynomielle Gleichungssysteme. In diesem Algorithmus nutzen die Entwickler lineare Gleichungen im Gleichungssystem, um Variablen zu eliminieren. Dadurch vereinfachen wir das System und erzeugen neue lineare Gleichungen, die wir wieder einsetzen, bis das System gelöst ist oder wir keine linearen Gleichungen mehr erhalten. Mit ElimLin lösen wir nicht alle Gleichungssysteme, aber es benötigt weniger Speicher und arbeitet gut mit vielen Variablen in einem dünn besetzten Gleichungssystem.

Ziel dieser Arbeit

In dieser Arbeit wollen wir Kryptosysteme durch polynomielle Gleichungssysteme in mehreren Variablen darstellen und sie mit Hilfe eines Algorithmus zum Lösen solcher Systeme brechen. Wie oben beschrieben, sind beliebige multivariate, polynomielle Gleichungssysteme sehr schwer zu lösen.

Moderne Algorithmen zum Lösen von polynomiellen Gleichungssystemen basieren

zumeist auf dem Buchberger-Algorithmus zum Auffinden einer Gröbnerbasis. Dieser hat theoretisch über einem beliebigen Körper eine doppelt exponentielle Laufzeit in der Anzahl der Variablen. Auch wenn die durchschnittliche Laufzeit kürzer ist, erfordert das Auffinden einer Gröbnerbasis für ein Ideal im Allgemeinen sehr viel Speicher.

Andererseits werden moderne Kryptosysteme mit relativ einfachen Aktualisierungsfunktionen entworfen, um sie gut in Hardware umzusetzen. Danach werden viele Runden ausgeführt, um eine Nachricht sicher zu verschlüsseln. Dadurch haben die resultierenden polynomiellen Darstellungen für eine einzelne Runde sehr viel Struktur. Wenn wir eine Chiffre so modellieren, dass diese Struktur in dem Gesamtsystem aus multivariaten Polynomen erhalten bleibt, könnten wir diese nutzen, um das Gleichungssystem zu lösen und so die Chiffre zu brechen.

Ziel: Finde Strategien, eine Chiffre so zu modellieren, dass ein angepasster Solver das resultierende polynomielle Gleichungssystem lösen kann.

Dafür benötigen wir aber auch einen Solver, der aus dieser Struktur Vorteile zieht. Auch wenn Gröbnerbasen sehr gut verstanden sind und theoretisch alle Lösungen jedes Gleichungssystems anzeigen können, sind sie dafür zu allgemein. Wir müssen also einen anderen Solver direkt an unsere Systeme anpassen.

Übersicht der Resultate

Wir beschreiben an dieser Stelle die Organisation der Arbeit und kurz die Resultate der einzelnen Kapitel.

Kapitel 2: Im zweiten Kapitel beschreiben wir Grundlagen, insbesondere die Konstruktion von Block- und Stromchiffren. Weiter stellen wir relevante Konzepte der modernen Kryptoanalyse dar. Danach geben wir eine Einführung in die Theorie von Gröbnerbasen und das Lösen von multivariaten, polynomiellen Gleichungssystemen.

Kapitel 3: Wir modellieren mehrere Instanzen der Blockchiffre Katan und der Stromchiffre Trivium mit gleichem Schlüssel und unterschiedlichen öffentlichen Werten wie beispielsweise dem Klartext. Dadurch erhalten wir ein dünn besetztes, multivariates, quadratisches, polynomielles Gleichungssystem. Dabei nutzen wir lineare Beziehungen zwischen diesen Instanzen, um so wenig Zwischenvariablen und Monome wie möglich in diesen Gleichungssystemen zu verwenden. Unsere Techniken lösen eine Sättigung in der Anzahl der Zwischenvariablen und Monome aus, die es uns erlaubt, Instanzen samt der Ausgabe einer Chiffre zu generieren, ohne dafür Zwischenvariablen einzuführen.

(Teile dieses Kapitels wurden in [QW14a] veröffentlicht. Die Idee für das Modell von Trivium entstand in Zusammenarbeit mit Enrico Thomae.)

Kapitel 4: Wir definieren die Technik der Cube-Angriffe und klassifizieren mögliche Varianten der Angriffstechnik aus algebraischer Sicht. Das sogenannte Superpoly erhalten wir aus der Zerlegung der polynomialen Darstellung einer Chiffre. In [DS09] führten die Autoren dies nur als lineares Polynom abhängig vom Schlüssel der Chiffre ein. Wir

erweitern die Definition des Polynoms in Bezug auf Definitionsmenge und Grad des Polynoms. Außerdem wenden wir die Technik des Cube-Angriffs auch auf den internen Zustand einer Chiffre mit dem Modell aus Kapitel 3 an. Dies erlaubt es uns, mehr Beziehungen zwischen einzelnen Instanzen zu finden.

(Dieses Kapitel entstand in Zusammenarbeit mit Enrico Thomae und Christopher Wolf; Teile des Kapitels wurden in [QW14b] veröffentlicht.)

Kapitel 5: Wir stellen ein realistisches Fehler-Modell für Angriffe auf symmetrische Kryptosysteme vor. Das Fehler-Modell beschreibt, welche Voraussetzungen wir bei einem Angriff haben. Für dieses 3-Bit-Fehler-Modell können wir den Fehler mit Hilfe der Techniken aus Kapitel 3 und 6 für Katan32 berechnen, falls wir den Fehler in einer Runde $R^* \geqslant 241$ von insgesamt 254 Runden injizieren. Daraufhin beschreiben wir einen Filter, mit dem wir Rundenschlüsselbits mit Hilfe von Fehlern in dem internen Zustand der Chiffre effektiv raten können. Wir rechnen die Ausgaben bis zur Runde R^* zurück, in der wir den Fehler injiziert haben, und überprüfen, ob wir einen gültigen Fehlervektor erhalten. Mit diesem Filter vermögen wir einen Angriff auf Katan mit $4,33$ bis $7,33$ Fehlern in 2^{29} bis 2^{74} Katan-Berechnungen durchzuführen. Das ist eine starke Verbesserung der früheren Fehler-Angriffe auf Katan in [ALRSS12, SH13], die 115 beziehungsweise 140 Fehler und bis zu 2^{59} Katan-Berechnungen benötigt haben.

(Teile dieses Kapitels wurden in [Que14] veröffentlicht. Das Fault-Modell entstand in Zusammenarbeit mit Falk Schellenberg)

Kapitel 6: In diesem Kapitel beschreiben wir einen Solver, der in der Lage ist, viele Variablen und Monome für unsere polynomiellen Gleichungssysteme aus Kapitel 3 zu verarbeiten. Der Solver basiert auf der Eliminierung von linearen Variablen und der Linearisierung des polynomiellen Gleichungssystems. Wir passen beide Methoden an, so dass wir die Struktur der Gleichungssysteme soweit wie möglich erhalten und die Modellierung aus Kapitel 3 effizient nutzen.

Die folgenden Daten sind wie folgt zu verstehen. Für einen validen Angriff müssen wir mit einer Wahrscheinlichkeit von $1-\delta$ für ein kleines $\delta > 0$ den Schlüssel finden und dabei weniger als die Größe des Schlüsselraums $|\mathcal{K}|$ Chiffren-Berechnungen und Chiffren-Daten benötigen. Damit wir besser mit dem Wert $|\mathcal{K}| = 2^{80}$ für beide Chiffren vergleichen können, bringen wir die Anzahl der benötigten Berechnungen und Ausgabebits in die Form 2^x für ein $x \in \mathbb{Q}$. Genauere Details dazu finden wir in Abschnitt 2.1.2 und 6.1.2.

Danach greifen wir die beiden in Kapitel 3 modellierten Chiffren an. Dabei können wir unseren algebraischen Angriff auf Trivium als Härtetest für unseren Solver ansehen. Wir brechen 625 von 1152 Runden Triviums in $2^{42.2}$ Trivium-Berechnungen und benötigen dafür 2^{12} Ausgabebits. Andere Angriffe brechen mehr Runden Triviums (799/1152), sind dafür aber nicht praktikabel in Bezug auf die 2^{62} benötigten Trivium-Berechnungen und benötigen viel mehr, nämlich 2^{40}, Ausgabebits. Frühere rein algebraische Angriffe auf Trivium hatten darüber hinaus gar keinen Erfolg, die Chiffre zu brechen. Unser Solver ist dabei in der Lage, Gleichungssysteme mit mehr als einer Million Monomen und mehr als 30.000 Variablen zu verarbeiten. Das ist weit außerhalb der Reichweite moderner Methoden für Gröbnerbasen.

In unserem Angriff auf Katan, der das „Gewählter Klartext“-Szenario verwendet, verbuchen wir weniger Erfolg. Wir brechen 80 der 254 Runden in $2^{72.8}$ Katan-Berechnungen, benötigen dafür 128 Klartextblöcke und raten 40 Variablen. Das ist der beste zur Zeit bekannte algebraische Angriff und besser als der Cube-Angriff auf Katan, der nur eine Variante von Katan mit $R = 65$ Runden zu brechen vermag. Allerdings brechen Albrecht und Leander in [AL12] in demselben Szenario 115 Runden in praktikablen 2^{32} Katan-Berechnungen und benötigen dafür 2^{32} Klartextblöcke mit einem differentiellen Angriff. Die Entwickler von Katan behaupten in [CDK], dass kein rein algebraischer Angriff mehr als die Hälfte aller Runden brechen kann. Dies können wir hier einmal mehr bestätigen.

Auf Katan führen wir im „Fehler“-Szenario einen weiteren Angriff wie in Kapitel 5 aus. Anstatt falsche Rundenschlüsselbits zu entdecken, filtern wir falsche Gleichungssysteme, um damit Fehler in kleinen Fehler-Runden $R^* < 200$ zu injizieren. Der Angriff bricht Katan mit durchschnittlich $7,5$ Fehlern in $2^{27.2}$ Katan-Berechnungen. Dieses Ergebnis erhöht den Abstand zu vorherigen Angriffen noch weiter.

(Teile dieses Kapitels wurden in [QW14a] und [Que14] veröffentlicht. Die Datenstruktur für den Solver und der Ausgangsalgorithmus von ElimLin wurde von Christopher Wolf in erster Version implementiert.)

Hauptresultate der Arbeit:

In dieser Arbeit gibt es zwei Resultate, die von besonderem Interesse sind.
Zum Einen ist das die Entwicklung und Implementierung des gesamten Solvers. Dieser kann dünn besetzte, quadratische, polynomielle Gleichungssysteme sehr effizient lösen. Die Idee, die zwei Algorithmen ElimLin und SL zu verbinden, wurde hier erfolgreich umgesetzt und führt zu einer Verbesserung der beiden Ursprungsalgorithmen mit viel Kontrolle durch die eingefügte Monomordnung, wie in Abschnitt 6.2 beschrieben ist. Ferner gab es unseres Wissens nach vor dieser Arbeit keine Implementierung eines Solvers, der über eine Million Monome und über 30.000 Variablen auf einem Standard-Computer mit 16Gb RAM verarbeiten kann. Der algebraische Angriff auf Trivium kann als Stresstest für den Solver angesehen werden. Dabei ist der eigentliche Angriff nicht so interessant wie die Funktionsweise und Flexibilität des Solvers.

Das andere Resultat, das besonders heraussticht, ist die Modellierung der Chiffren beziehungsweise die in Kapitel 6 gezeigte Flexibiltät dieses Modells. Die Modelle sind an den Cube-Angriff angelehnt. Dadurch nutzen wir die Vorteile dieses Angriffs, wie das Benutzen mehrerer Instanzen und Zerlegen des vollen Polynoms der Chiffre in Polynome kleinen Grades, auf natürliche Weise. Darüber hinaus können wir nicht nur einen algebraischen Angriff im „Gewählter Klartext“-Szenario durchführen. Wenn sich das Szenario ändert, ist es sehr einfach das Modell anzupassen. So war es möglich aus dem Modell einen Filter für das „Fehler“-Szenario abzuleiten, obgleich das Szenario ganz andere Bedingungen an den Angreifer stellt. In diesem Zusammenhang ist die Idee sehr erfolgreich, falsche Gleichungssysteme zuzulassen, um zur Lösung zu kommen.

Kapitel 2

Grundlagen

In diesem Kapitel erläutern wir die Grundlagen dieser Arbeit. In Abschnitt 2.1 lernen wir die Prinzipien der Kryptographie und insbesondere der Krytpoanalyse kennen. Dabei führen wir symmetrische kryptographische Primitive aus theoretischer Sicht ein. Danach beschäftigen wir uns in Abschnitt 2.2 mit dem Lösen von polynomiellen, multivariaten Gleichungssystemen über einem endlichen Körper. Dabei greifen wir in Abschnitt 2.2.1 die Notation für multivariate Polynome auf und beschäftigen uns mit dem Buchberger-Algorithmus zur Berechnung von Gröbnerbasen. Moderne Algorithmen zur Berechnung von Gröbnerbasen basieren auf diesem einfachen Algorithmus. Als Beispiel beschreiben wir den F_4-Algorithmus in Abschnitt 2.2.2 ein, der den Buchberger-Algorithmus durch Techniken der linearen Algebra beschleunigt. Zuletzt stellen wir in Abschnitt 2.2.3 den XL-Algorithmus vor. Dieser verfolgt einen alternativen Ansatz, um ein polynomielles Gleichungssystem zu lösen.

2.1 Grundlagen der Kryptologie

Während sich die Codierungstheorie mit dem zuverlässigen Übertragen von Daten über einen eventuell verrauschten Kanal beschäftigt, stellen wir in der Kryptographie sicher, dass die Daten nicht von Dritten gelesen oder verändert werden können. In Abschnitt 2.1.1 führen wir die Grundlagen der Kryptographie ein. Dabei stellen wir symmetrische Kryptosysteme vor und führen die sogenannten Block- und Stromchiffren ein. Blockchiffren teilen eine Nachricht in Blöcke auf und verschlüsseln jeden Block einzeln. Im Gegensatz dazu liefern Stromchiffren einen langen, scheinbar zufälligen Schlüsselstrom, mit dem eine Nachricht additiv verknüpft wird. Danach stellen wir Schieberegister vor. Dies sind Objekte, die sich sehr gut eignen, Schlüsselstrom zu produzieren. Aber auch Blockchiffren lassen sich mit Hilfe von Schieberegistern konstruieren, wie wir in Abschnitt 3.2 sehen.

Wir führen in Abschnitt 2.1.2 Sicherheitsspiele ein, die beschreiben, welche Informationen wir bei einem Angriff auf ein Kryptosystem erhalten und wann wir eine Chiffre gebrochen haben. Außerdem dürfen wir für einen erfolgreichen Angriff nur eine bestimmte Anzahl an Chiffre-Berechnungen ausführen und Daten der Chiffre nutzen.

2.1.1 Kryptographie

In diesem Abschnitt stellen wir Grundprinzipien der Kryptographie vor. Wir wollen über einen abhörbaren und manipulierbaren Kanal kommunizieren. Dabei soll ein möglicher Angreifer Nachrichten weder entschlüsseln noch unbemerkt verändern können. Dabei folgen wir grob den Ausführungen in [Buc03]. In der nächsten Definition lernen wir ein Kryptosystem beziehungsweise eine Chiffre kennen, das es uns ermöglicht, über einen abhörbaren Kanal zu kommunizieren.

Definition 2.1.1. *Ein Kryptosystem ist ein* 5*-Tupel* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$*, so dass die folgenden Regeln gelten:*

a) $\mathcal{P}$ *ist eine endliche Menge von Klartexten.*

b) $\mathcal{C}$ *ist eine endliche Menge von Geheimtexten.*

c) $\mathcal{K}$ *ist eine Menge aus endlich vielen Schlüsseln, genannt der Schlüsselraum.*

d) $\mathcal{E} = \{\mathtt{enc}_K \mid K \in \mathcal{K}\}$ *ist eine Familie von Funktionen* $\mathtt{enc}_K : \mathcal{P} \to \mathcal{C}$*. Ihre Elemente heißen Verschlüsselungsfunktionen.*

e) $\mathcal{D} = \{\mathtt{dec}_K \mid K \in \mathcal{K}\}$ *ist eine Familie von Funktionen* $\mathtt{dec}_K : \mathcal{C} \to \mathcal{P}$*. Ihre Elemente heißen Entschlüsselungsfunktionen.*

f) Für jedes $e \in \mathcal{K}$ *gibt es* $d \in \mathcal{K}$*, so dass für alle* $p \in \mathcal{P}$ *die Gleichung* $\mathtt{dec}_d(\mathtt{enc}_e(p)) = p$ *gilt.*

Die Definition 2.1.1 ist die klassische Definition eines Kryptosystems. Sie umfasst keine modernen Signatur-Verfahren und Hash-Funktionen. Auch gibt es Chiffren, für die die Bedingung *f)* nur mit hoher Wahrscheinlichkeit gilt. Für symmetrische Kryptoanalyse reicht diese Definition aus [Buc03] vollkommen aus. Auch ist die Definition sehr viel klarer als moderne Definitionen, weil sie weniger Spezialfälle abdeckt. Eine gute moderne Definition wird in [KL07] gegeben.

Die Bedingung *f)* der Definition besagt Folgendes: Wenn ein Klartext $p \in \mathcal{P}$ mit Hilfe von $\mathtt{enc}_e$ verschlüsselt wird, dann muss der Geheimtext $\mathtt{enc}_e(p) \in \mathcal{C}$ mit Hilfe von $\mathtt{dec}_d$ wieder nach p entschlüsselt werden können. Dies bedeutet, dass die Abbildung $\mathtt{enc}_e$ für jedes $e \in \mathcal{K}$ injektiv sein muss.

Wie wir im nächsten Abschnitt sehen, benutzen moderne Chiffren $\mathcal{P} = \mathcal{C}$ und Polynome $f : \mathcal{P} \to \mathcal{P}$ von niedrigem totalen Grad und evaluieren diese mehrfach, um zu verschlüsseln. Die polynomiale Darstellung von `enc` können wir dann selbst auf einem Cluster nicht mehr darstellen, weil der Grad zu hoch ist und die Darstellung des Polynoms dadurch zu viel Speicher erfordert. In Kapitel 3 teilen wir das Polynom mit Hilfe von Zwischenvariablen, um es darzustellen. Weiter zerlegen wir in Kapitel 4 dieses Polynom durch sogenannte Cube-Summen, so dass wir durch Summation für verschiedene Werte Polynome von kleinem Grad erhalten, die Teile der Verschlüsselung beschreiben.

Die Elemente aus $\mathcal{P}$ und $\mathcal{C}$ können sowohl einzelne Buchstaben oder Zahlen, als auch ganze Blöcke von Buchstaben oder Zahlen sein.

Die Nachricht $m = m_0 m_1 \dots m_n$ mit $m_i \in \mathcal{P}$ und den Geheimtext $m' = m'_0 m'_1 \dots m'_k$ mit $m'_i \in \mathcal{C}$ stellen wir über einem Alphabet dar. Unter einem Alphabet verstehen wir eine endliche, nichtleere Menge Σ. Die Kardinalität von Σ ist die Anzahl der Elemente in Σ und diese Elemente heißen Zeichen, Buchstaben oder Symbole von Σ.

Beispiel 2.1.2.

- *Im weiteren Verlauf dieser Arbeit verwenden wir das binäre Alphabet* $\{0, 1\}$.
- *Das lateinische Alphabet ist* $\mathcal{A} = \{a, b, \dots, z\}$. *Es hat die Länge* $|\mathcal{A}| = 26$.
- *Wir können jedem Zeichen eines Alphabets eine ganze Zahl aus* $\mathbb{Z}/k\mathbb{Z}$ *mit der Länge des Alphabets* k *zuordnen. Verwenden wir keine Sonderzeichen, können wir zum Beispiel für das lateinische Alphabet* $\mathcal{A}$ *die Zuordnung*

$$\begin{aligned} c : \{a, b, \dots, z\} &\to \{0, 1, \dots, 25\}, \\ a &\mapsto 0, \\ b &\mapsto 1, \\ \vdots &\quad \vdots \quad \vdots \\ z &\mapsto 25 \end{aligned}$$

  benutzen. Unser Alphabet ist dann $\mathcal{B} = \mathbb{Z}/26\mathbb{Z}$. *Also können wir die Nachricht* m *mit Zeichen aus dem Alphabet* $\mathcal{A}$ *mittels der bijektiven Abbildung* c *in das Alphabet* $\mathcal{B}$ *überführen. Die Hinrichtung dieser Übersetzung kennzeichnen wir im Folgenden mit* $=_c$ *und die Rückrichtung mit* $=_{c^{-1}}$.

Um mit Hilfe eines Alphabets Klar- und Geheimtexte darstellen zu können, benötigen wir noch folgende Definition.

Definition 2.1.3. *Sei* Σ *ein Alphabet.*

1. *Als Wort oder String über* Σ *bezeichnen wir eine endliche Folge von Zeichen aus* Σ *einschließlich der leeren Folge, die wir mit* ϵ *bezeichnen und leeres Wort nennen.*
2. *Die Länge eines Wortes* $\boldsymbol{w}$ *über* Σ *ist die Anzahl seiner Zeichen, die wir mit* $|\boldsymbol{w}|$ *bezeichnen. Das leere Wort hat die Länge* 0.
3. *Die Menge aller Wörter über* Σ *einschließlich des leeren Worts bezeichnen wir als* Σ^*.
4. *Sind* $\boldsymbol{v}, \boldsymbol{w} \in \Sigma^*$, *dann ist der String* $\boldsymbol{vw} = \boldsymbol{v} \circ \boldsymbol{w}$, *den man durch Hintereinanderschreiben von* $\boldsymbol{v}$ *und* $\boldsymbol{w}$ *erhält, die Konkatenation von* $\boldsymbol{v}$ *und* $\boldsymbol{w}$. *Insbesondere schreiben wir* $\boldsymbol{v} \circ \epsilon = \epsilon \circ \boldsymbol{v} = \boldsymbol{v}$.
5. *Ist* n *eine nicht-negative ganze Zahl, dann bezeichnen wir mit* Σ^n *die Menge aller Wörter der Länge* n *über dem Alphabet* Σ.

Ein Wort über dem oben beschriebenen lateinischen Alphabet $\mathcal{A}$ ist zum Beispiel „beispiel“. Es hat die Länge 8. Ein anderes Wort wäre „gutes“. Die Konkatenation aus beiden ist „gutesbeispiel“. Zur Trennung dieser Wörter könnten wir unser Alphabet durch das Leerzeichen erweitern.

Um eine Nachricht aus einem Wort oder mehreren Wörtern zu verschlüsseln, definieren wir eine injektive Abbildung über einem Alphabet Σ als Verschlüsselungsfunktion, wie wir im nächsten Beispiel sehen.

Beispiel 2.1.4. *Betrachte $\mathcal{P} = \mathcal{C} = \mathcal{B} = \mathbb{Z}/26\mathbb{Z}$ und die Abbildung $\texttt{enc}_{13}(x) = \texttt{dec}_{13}(x) = x + 13 \mod 26$.*

Wir wollen die Nachricht $m = b\ e\ i\ s\ p\ i\ e\ l$ verschlüsseln. Dazu übersetzen wir sie mit Hilfe der Abbildung c in das Alphabet $\mathcal{B}$. Das liefert die Nachricht $m =_c [1, 4, 8, 18, 15, 8, 4, 11]$ im Alphabet $\mathcal{B}$. Wir wenden die Verschlüsselungsfunktion $\texttt{enc}$ auf jede Stelle der Nachricht an und erhalten $[14, 17, 21, 5, 2, 21, 17, 24]$. Wenn wir diese Nachricht im lateinischen Alphabet darstellen, lautet sie orvfcvry*.*

Die Abbildungen $\texttt{enc}$ und $\texttt{dec}$ arbeiten in diesem Beispiel mit dem Schlüssel 13. Der Schlüsselraum ist bei dieser Chiffre $\mathcal{K} = \mathcal{B}$. Für einen Schlüssel $e \in \mathbb{Z}/26\mathbb{Z}$ ist die Verschlüsselungsfunktion $\texttt{enc}_e(x) = x + e \mod 26$. Den Schlüssel der Entschlüsselungsfunktion können wir leicht durch $d = 26 - e \mod 26$ bestimmen. Die Entschlüsselungsfunktion lautet dann $\texttt{dec}_d(x) = x + d \mod 26$.

Diese einfache Verschlüsselung lässt sich aus heutiger Sicht einfach brechen, weil es nur 26 Schlüssel gibt und sie schlechte statistische Eigenschaften aufweist, wie zum Beispiel in [HU07] gezeigt wird. Wenn wir einfach alle Schlüssel ausprobieren und testen, ob wir einen sinnvollen Text erhalten, ist nicht nur die Nachricht entschlüsselt, sondern wir erhalten auch den geheimen Schlüssel und sind selbst in der Lage, verschlüsselte Nachrichten zu verschicken. Mit dieser Chiffre und dem Schlüssel $e = 3$ hat Cäsar seine geheimen Nachrichten verschlüsselt.

Wir merken an, dass wir nicht dasselbe Alphabet für $\mathcal{P}$ und $\mathcal{C}$ verwenden müssen. Zum Beispiel können wir auch von $\mathcal{P} = \mathcal{A}^n$ direkt nach $\mathcal{C} = \mathcal{B}^k$ verschlüsseln, wenn wir über entsprechende Abbildungen $\texttt{enc}$ und $\texttt{dec}$ verfügen.

Im weiteren Verlauf dieser Arbeit beschäftigen wir uns mit *symmetrischen* Kryptosystemen.

Definition 2.1.5. *Sei $F = (\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ ein Kryptosystem. Wir nennen F symmetrisch, wenn beide Abbildungen $\texttt{enc}_e \in \mathcal{E}$ und $\texttt{dec}_d \in \mathcal{D}$ denselben Schlüssel $e = d = K$ verwenden oder sich der Schlüssel der einen Funktion in polynomieller Zeit aus dem Schlüssel der anderen Funktion berechnen lässt.*

Das heißt, in einem symmetrischen Kryptosystem können wir mit einem beliebigen Schlüssel sowohl verschlüsseln als auch entschlüsseln.

Ein wichtiges Problem von symmetrischen Kryptosystemen ist, dass wir den benutzten Schlüssel erst vereinbaren müssen. Für die Schlüsselübergabe verwenden wir asymmetrische Kryptosysteme. Diese Kryptosysteme erfüllen folgende Bedingungen.

1. Für alle Schlüsselpaare $(e, d) \in \mathcal{K}_\infty \times \mathcal{K}_\in$ und alle Klartexte $x \in \mathcal{P}$ lässt sich der Geheimtext $\texttt{enc}_e(x)$ mit Hilfe des sogenannten öffentlichen Schlüssels e schnell berechnen.

2. Ohne den geheimen Schlüssel d lässt sich das Urbild eines Geheimtextes $y \in \mathcal{C}$ unter $\texttt{enc}_e$ nicht in polynomieller Zeit berechnen. Selbst dann nicht, wenn der öffentliche Schlüssel e und die Abbildung $\texttt{enc}_e$ bekannt sind.

3. Mit dem geheimen Schlüssel d lässt sich $\texttt{dec}_d(y)$ für alle Geheimtexte $y \in \mathcal{C}$ in polynomieller Zeit berechnen.

Für ein solches Kryptosystem sind $\texttt{enc}_e$ und der öffentliche Schlüssel e bekannt. Es reicht aus, den geheimen Schlüssel d geheim zu halten, um eine Nachricht zu verschlüsseln. Es ist üblich, für ein asymmetrisches Kryptosystem zwei Schlüsselräume für den öffentlichen und den privaten Schlüssel zu vereinbaren. Dadurch ändert sich die Definition von Kryptosystemen aber nicht wesentlich.

Der Vorteil von asymmetrischen Kryptosystemen ist, dass wir keine geheimen Schlüsselabsprachen benötigen. Wir halten den Teilschlüssel d geheim. Mit Hilfe von e und $\texttt{enc}_e$ kann uns jeder eine Nachricht schicken, ohne dass sie jemand entschlüsseln kann. Nur wir können dies.

Solche Systeme sind in Bezug auf die benötige Rechenzeit zum Verschlüsseln und Entschlüsseln viel langsamer als symmetrische Kryptosysteme.

In der modernen Kryptographie werden asymmetrische Verfahren dazu eingesetzt, geheime Schlüssel für ein symmetrisches Verfahren zu übertragen beziehungsweise solche Schlüssel zu vereinbaren. Danach verschlüsseln wir Nachrichten mit einem symmetrischen Kryptosystem. Für den Rest dieser Arbeit werden wir davon ausgehen, dass wir den geheimen Schlüssel sicher austauschen können, und uns allein mit symmetrischen Kryptosystemen beschäftigen.

Bei einem symmetrischen Verfahren werden zumeist viele Nachrichten mit demselben Schlüssel verschlüsselt. Also muss die Verschlüsselung mit einem Startwert - Initial Value (IV) randomisiert werden. Wenn nämlich ein Klartext oft mit demselben Schlüssel verschlüsselt wird und immer der gleiche Geheimtext entsteht, kennt ein Angreifer sofort die Nachricht, sobald er einmal die Nachricht entschlüsselt hat. Folgendes Beispiel aus [Buc03] erläutert dieses Prinzip.

Beispiel 2.1.6. *Ein Bankkunde handelt mit Aktien und sendet der Bank jeweils die Anweisungen „Kaufen“, „Halten“ oder „Verkaufen“. Diese Anweisungen werden der Bank verschlüsselt übermittelt. Wenn ein Angreifer einmal merkt, dass nach dem Geheimtext zu „Kaufen“ ein Kauf getätigt wurde, so kann er diese Nachricht jedesmal entschlüsseln.*

Ist die Chiffre randomisiert, so sieht der Geheimtext mit an Sicherheit grenzender Wahrscheinlichkeit bei jedem Durchlauf der Chiffre anders aus.

Bei der Stromchiffre Trivium, die wir in Abschnitt 3.1 behandeln, werden wir als IV einen 80 Bit langen Initialisierungsvektor benutzen.

Blockchiffren

Kryptosysteme, in denen wir nicht ein Zeichen verschlüsseln, sondern viele Zeichen gleichzeitig behandeln, nennen wir Blockchiffren. Dabei bilden wir Blöcke fester Länge auf Blöcke derselben Länge ab.

Definition 2.1.7. *Sei $F = (\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ ein Kryptosystem. Wir nennen F eine Blockchiffre, wenn der Klartext- und der Geheimtextraum die Menge Σ^n aller Wörter der Länge n über einem Alphabet Σ sind. Der Parameter $n \in \mathbb{N}$ heißt die Blocklänge.*

Wir veranschaulichen das Prinzip am Beispiel des sogenannten *Vigenère-Kryptosystems.*

Beispiel 2.1.8. *Sei m eine feste, positive Zahl. Im* Vigenère-Kryptosystem *seien $\mathcal{P} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}/26\mathbb{Z})^m$. Für einen Schlüssel $K = (k_1, \ldots, k_m) \in \mathcal{K}$ definieren wir*

$$\texttt{enc}_K : \mathcal{P} \to \mathcal{C},$$
$$\texttt{enc}_K(x_1, \ldots, x_m) = (x_1 + k_1, x_2 + k_2, \ldots, x_m + k_m)$$

und

$$\texttt{dec}_K : \mathcal{C} \to \mathcal{P},$$
$$\texttt{dec}_K(y_1, \ldots, y_m) = (y_1 - k_1, y_2 - k_2, \ldots, y_m - k_m).$$

Dabei finden alle Rechnungen über $\mathbb{Z}/26\mathbb{Z}$ statt.

Wir vereinbaren die Blocklänge $m = 5$ und das Schlüsselwort vigen $=_c [21, 8, 6, 4, 13]$. Wir wollen die Nachricht

$$e\ i\ n\ b\ e\ i\ s\ p\ i\ e\ l\ f\ u\ e\ r\ b\ l\ o\ c\ k\ c\ h\ i\ f\ f\ r\ e\ n =_c$$
$$[4, 8, 13, 1, 4, 8, 18, 15, 8, 4, 11, 5, 20, 4, 17, 1, 11, 14, 2, 10, 2, 7, 8, 5, 5, 17, 4, 13]$$

verschicken.

Dazu zerlegen wir sie in Klartextblöcke der Länge 5. Falls der letzte Block nicht mit der eigentlichen Nachricht abschließt, ergänzen wir die Nachricht mit seltenen Buchstaben, wie $x =_c 23$. Auf diese Weise erhalten wir die Blöcke $[4, 8, 13, 1, 4]$, $[8, 18, 15, 8, 4]$, $[11, 5, 20, 4, 17]$, $[1, 11, 14, 2, 10]$, $[2, 7, 8, 5, 5]$, $[17, 4, 13, 23, 23]$.

Jetzt addieren wir das Schlüsselwort zu jedem der Blöcke. Wir erhalten die Geheimtextblöcke

$$[25, 16, 19, 5, 17], [3, 0, 21, 12, 17], [6, 13, 0, 8, 4], [22, 19, 20, 6, 23], [23, 15, 14, 9, 18],$$
$$[12, 12, 19, 1, 10] =_{c^{-1}} z\ q\ t\ f\ r\ d\ a\ v\ m\ r\ g\ n\ a\ i\ e\ w\ t\ u\ g\ x\ x\ p\ o\ j\ s\ m\ m\ t\ b\ k.$$

Um die Nachricht zu entschlüsseln, zerlegen wir die Nachricht wieder in Blöcke der Länge 5 und wenden die Abbildung $\texttt{dec}_K$ an.

Wenn wir einen Block betrachten, können wir die Abbildung für Verschlüsselung beziehungsweise Entschlüsselung mehrfach und mit unterschiedlichen Schlüsseln anwenden. Durch diese R Runden gewährleisten wir mehr Sicherheit, müssen aber auch mehr Schlüssel speichern. Um das zu vermeiden, verwenden wir sogenannte Rundenschlüssel K_r für $0 \leqslant r < R$, die aus dem Schlüssel $K \in \mathcal{K}$ mittels eines Algorithmus zur Aufstellung der Rundenschlüssel entstehen. Dieser Algorithmus wird für jede moderne Blockchiffre fest aber beliebig gewählt, wenn eine Blockchiffre konstruiert wird.

Moderne Blockchiffren verwenden Substitution, wie in Beispiel 2.1.8 beschrieben, und Permutation für eine bestimmte Rundenanzahl R.

Für die Substitution ersetzen wir einzelne Einträge des Blocks durch andere. Dafür verwenden wir, anders als in Beispiel 2.1.8, keine linearen Abbildungen. Lineare Substitution kann leicht gebrochen werden. Wir wollen den totalen Grad des resultierenden polynomiellen Gleichungssystems, das die Chiffre beschreibt, so hoch wie möglich setzen. Dafür benötigen wir in einer Runde keine Funktion hohen Grades, wenn wir viele Runden durchführen. In Abschnitt 3.2 lernen wir ein gutes Beispiel für eine Blockchiffre kennen, das viele Runden und eine quadratische Funktion als Substitution verwendet.

In dem Permutationsschritt vermischen wir einzelne Zeichen des Blocks, wie wir am folgenden Beispiel sehen.

Beispiel 2.1.9. *Sei $F = (\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ ein Kryptosystem mit $\mathcal{P} = \mathcal{C} = \{a, \ldots, z\}$. Die Menge der Schlüssel $\mathcal{K}$ besteht aus allen möglichen Permutationen der Menge $\{1, \ldots, n\}$ einer bestimmten Länge n. Für dieses Beispiel setzen wir $n = 5$.*

Für jede Permutation $\pi \in \mathcal{K}$ sei $\mathtt{enc}_\pi = \pi$ und $\mathtt{dec}_\pi = \pi^{-1}$. Sei weiter $\pi : \{1, \ldots, 5\} \to \{1, \ldots, 5\}$ die bijektive Abbildung mit

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix}.$$

Dabei ist π eine Permutation der Position der Buchstaben in einem Klartextblock. Wir ersetzen also keinen Buchstaben, sondern sortieren die Buchstaben um.

Wenn wir die Nachricht e i n b e i s p i e l f u e r b l o c k c h i f f r e n verschlüsseln wollen, teilen wir die Nachricht in Blöcke der Länge 5 und wenden π auf jeden Block an. Wir erhalten folgende Verschlüsselung der Nachricht.

Klartext	*e i n b e*	*i s p i e*	*l f u e r*	*b l o c k*	*c h i f f*	*r e n x x*
Permutation	*3 4 2 5 1*	*3 4 2 5 1*	*3 4 2 5 1*	*3 4 2 5 1*	*3 4 2 5 1*	*3 4 2 5 1*
Geheimtext	*n b i e e*	*p i s e i*	*u e f r l*	*o c l k b*	*i f h f c*	*n x e x r*

Also erhalten wir den Geheimtext n b i e e p i s e i u e f r l o c l k b i f h f c n x e x r.

Modernen Blockchiffren sind mit Hilfe von Permutation und Substitution aufgebaut. Bei sogenannten *Substitution-Permutation-Netzwerken* (SPN) gehen wir in einer Runde wie folgt vor.

1. Addiere den (numerischen) Klartextblock m_0 zu einem Rundenschlüssel K_i.

2. Substituiere die Zeichen von $m_1 = m_0 + K_i$ mit Hilfe einer nichtlinearen Funktion $S : \Sigma^n \to \Sigma^n$. S ist die sogenannte S-Box.

3. Permutiere die Zeichen von $m_2 = S(m_1)$ mit Hilfe einer Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.

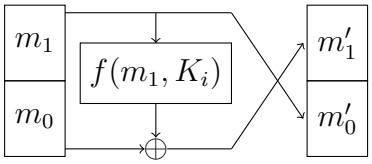


Abbildung 2.1: Schematische Darstellung einer Runde eines Feistel-Netzwerks

Dabei kennzeichnet n die Blocklänge des SP-Netzwerks. Wir führen so viele Runden aus, bis bestimmte Sicherheitsparameter erfüllt sind.

Abbildung 2.1 zeigt eine Runde eines Feistel-Netzwerks, die zum Beispiel in [KL07] ausführlich beschrieben sind. Der Eingangsblock m wird in zwei Blöcke m_0 und m_1 geteilt. Dann addieren wir m_0 zu der Ausgabe einer (nichtlinearen) Aktualisierungsfunktion $f(m_1, K_i) : \Sigma^{|m_1|} \times \Sigma^{|K_i|} \to \Sigma^{|m_0|}$ und tauschen die beiden Blöcke m_0 und m_1. Die Funktion f dient dabei als Substitution.

Stromchiffren

Eine andere Klasse von symmetrischen Kryptosystemen bilden die Stromchiffren. Wir folgen der Notation aus [HU07].

Definition 2.1.10. *Eine Stromchiffre ist ein 7-Tupel $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{L}, \mathcal{F}, \mathcal{E}, \mathcal{D})$ mit*

a) $\mathcal{P}$ ist eine endliche Menge von Klartexten.

b) $\mathcal{C}$ ist eine endliche Menge von Geheimtexten.

c) $\mathcal{K}$ ist eine endliche Menge von Schlüsseln.

d) $\mathcal{L}$ ist eine endliche Menge, genannt das Schlüsselstromalphabet.

e) $\mathcal{F} = (f_1, f_2, \ldots)$ ist der Schlüsselstrom-Erzeuger. Für $i \geqslant 1$ ist f_i eine Abbildung $f_i : \mathcal{K} \times \mathcal{P}^{i-1} \to \mathcal{L}$.

f) Für jedes $z \in \mathcal{L}$ gibt es eine Chiffrierungsregel $\mathtt{enc}_z \in \mathcal{E}, \mathtt{enc}_z : \mathcal{P} \to \mathcal{C}$ und eine Dechiffrierungsregel $\mathtt{dec}_z \in \mathcal{D}, \mathtt{dec}_z : \mathcal{C} \to \mathcal{P}$, so dass für alle $x \in \mathcal{P}$ gilt: $\mathtt{dec}_z(\mathtt{enc}_z(x)) = x$.

Das folgende Beispiel zeigt, wie wir mit Stromchiffren arbeiten.

Beispiel 2.1.11. *Wir betrachten das sogenannte Selbstschlüssel-Kryptosytem mit $\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathcal{L} = \mathbb{Z}/26\mathbb{Z}$ und einer Nachricht $m = m_0 m_1 \ldots$. Der Algorithmus für den Schlüsselstrom-Erzeuger ist*

$$z_0 = K$$

$$z_i = m_{i-1} \text{ für } i \geqslant 1.$$

Für $0 \leqslant z \leqslant 25$ definieren wir die Abbildungen

$$\texttt{enc}_z(x) = (x + z) \mod 26 \text{ und}$$
$$\texttt{dec}_z(y) = (y - z) \mod 26.$$

Wir verschlüsseln die Nachricht

$$e\ i\ n\ b\ e\ i\ s\ p\ i\ e\ l\ f\ u\ e\ r\ s\ t\ r\ o\ m\ c\ h\ i\ f\ f\ r\ e\ n =_c$$
$$[4, 8, 13, 1, 4, 8, 18, 15, 8, 4, 11, 5, 20, 4, 17, 18, 19, 17, 14, 12, 2, 7, 8, 5, 5, 17, 4, 13]$$

mit dem Schlüssel $K = 8$. Dazu müssen wir den nötigen Schlüsselstrom erzeugen. Wir verwenden den Schlüssel als erstes Zeichen $z_0 = K = 8$ und folgen mit dem Klartext $z_1 = m_0 = 4, z_2 = m_1 = 8, \ldots, z_{28} = m_{27} = 4$. Also ist der Schlüsselstrom

$$[8, 4, 8, 13, 1, 4, 8, 18, 15, 8, 4, 11, 5, 20, 4, 17, 18, 19, 17, 14, 12, 2, 7, 8, 5, 5, 17, 4].$$

Es folgt der Geheimtext

$$[12, 12, 21, 14, 5, 12, 0, 7, 23, 12, 15, 16, 25, 24, 21, 9, 11, 10, 5, 0,$$
$$14, 9, 15, 13, 10, 22, 21, 17] =_{c^{-1}} mmvofmahxmpqzyvjlkfaojpnkwvr.$$

Wenn wir die Nachricht wieder entschlüsseln wollen, subtrahieren wir den Schlüssel von der ersten Stelle des Geheimtexts $m_1 = \texttt{dec}_{z_0=K} = 12 - 8 \mod 26 = 4$ und erhalten damit auch das zweite Zeichen des Schlüsselstroms $z_1 = 4$. So fahren wir fort und erhalten wieder unsere Ausgangsnachricht.

Das Selbstschlüssel-Kryptosystem ist unter anderem unsicher, weil der Schlüsselraum zu klein ist. Wir können einfach alle möglichen Schlüssel ausprobieren, um den Schlüssel zu finden. In der nächsten Definition lernen wir zwei Eigenschaften von Stromchiffren kennen.

Definition 2.1.12. *a) Eine Stromchiffre heißt synchron, wenn der Schlüsselstrom unabhängig vom Klartext ist.*

b) Eine Stromchiffre heißt periodisch mit Periode d, falls $z_{i+d} = z_i$ für alle $i \geqslant 0$.

Beispiel 2.1.13. *a) Das Selbstschlüssel-Kryptosystem ist nicht synchron.*

b) Wir können das Vigenère-Kryptosystem aus Beispiel 2.1.8 als eine synchrone Stromchiffre interpretieren. Wenn wir den Schlüssel $K = [k_1, \ldots, k_m]$ wählen, so können wir $z = [k_1, \ldots, k_m, k_1, \ldots, k_m, \ldots]$ als Schlüsselstrom mit $z_j = k_i$ für $i \mod m = j$ betrachten. Die Periode dieser Stromchiffre ist m.

Im folgenden Abschnitt und in Abschnitt 3.1 lernen wir weitere Stromchiffren kennen. Dabei ver- beziehungsweise entschlüsseln wir stets, indem wir den Schlüsselstrom zur Nachricht addieren beziehungsweise subtrahieren. Das setzt voraus, dass wir nur synchrone Stromchiffren betrachten.

Lineare Schieberegister

Ein Weg, Stromchiffren zu konstruieren, sind lineare Schieberegister. Sie sind für den endlichen Körper $\mathbb{F}_2$ effizient in Hardware implementierbar und haben gute statistische Eigenschaften als Schlüsselstromgenerator.

Definition 2.1.14. *Sei $\mathbb{F}$ ein endlicher Körper. Ein lineares Schieberegister der Länge n besteht aus einem internen Zustand $S = (s_i, s_{i-1}, \ldots, s_{i-(n-1)}) \in \mathbb{F}^n$ und einer Schiebe- oder Aktualisierungsfunktion $f(\mathbf{x}) = \sum_{i=1}^{n} \lambda_{i-1} \cdot x_i \in \mathbb{F}[x_1, \ldots, x_n]$ mit Koeffizienten $\lambda_k \in \mathbb{F}$. Ein lineares Schieberegister wird rundenweise aktualisiert. Pro Runde wird ein Eintrag des internen Zustands ausgegeben und der interne Zustand mit Hilfe einer Funktion f aktualisiert. Sei $S_0 = (s_0, s_{-1}, \ldots, s_{-(n-1)}) \in \mathbb{F}^n$ der anfängliche interne Zustand des Schieberegisters und $L(x_1, \ldots, x_n) := (f(x_1, \ldots, x_n), x_1, \ldots, x_{n-1})$. In jeder Runde werden zwei Schritte ausgeführt:*

1. *Gib einen Eintrag des internen Zustands aus.*

2. *Aktualisiere den internen Zustand S_t zu S_{t+1} mit Hilfe von L:*

$$S_{t+1} := L(S_t) = (f(S_t), s_t, \ldots, s_{t-(n-2)}) = (\sum_{k=0}^{n-1} \lambda_k \cdot s_{t-k}, s_t, \ldots, s_{t-(n-2)}).$$

Die Aktualisierungsfunktion f schreiben wir auch als

$$s_{t+1} = \sum_{k=0}^{n-1} \lambda_k \cdot s_{t-k},$$

um die Rekursion stärker zu betonen.

Für diesen Abschnitt nehmen wir an, dass wir stets den letzten Eintrag ausgeben. In den folgenden Kapiteln geben wir hingegen keinen oder bestimmte Werte aus. Wenn wir nicht den letzten Eintrag als Ausgabe verwenden, weisen wir darauf hin. Das folgende Beispiel für ein lineares Schieberegister entnehmen wir [Arm06].

Beispiel 2.1.15. *Sei $\mathbb{F} := \mathbb{F}_2, n := 2$ und $f(x_1, x_2) = x_1 + x_2$. Wir initialisieren das Schieberegister mit $(s_0, s_{-1}) = (1, 1)$. Die folgende Tabelle zeigt die Werte des internen Zustands für die ersten 9 Runden. Mit diesem Schieberegister erhalten wir den Schlüssel-*

t	0	1	2	3	4	5	6	7	8	...
S_t	1	0	1	1	0	1	1	0	1	...
	1	1	0	1	1	0	1	1	0	

strom $1, 1, 0, 1, 1, 0, 1, 1, 0, \ldots$ mit einer Periode von 3.

Wenn wir $\mathbb{F} = \mathbb{F}_3$ wählen und die sonstigen Werte beibehalten, so erhalten wir in der nachfolgenden Tabelle den internen Zustand der ersten 10 Runden und den Schlüsselstrom $1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2, \ldots$. Dieser hat eine Periode von 8.

t	0	1	2	3	4	5	6	7	8	9	10	...
S_t	1	2	0	2	2	1	0	1	1	2	0	...
	1	1	2	0	2	2	1	0	1	1	2	

In dieser Arbeit beschäftigen wir uns mit Kryptosystemen, die mit Hilfe nichtlinearer Schieberegister konstruiert werden. Ein nichtlineares Schieberegister besitzt eine nichtlineare Aktualisierungsfunktion. Ein vollständiger Überblick über lineare Stromchiffren ist in [LN86] gegeben.

Schieberegister, mit denen wir Stromchiffren konstruieren, initialisieren wir mit dem geheimen Schlüssel und einem Startwert.

Wenn die Aktualisierungsfunktion „zurückrechenbar“ ist, können wir das Schieberegister invertieren. Sei dazu $S = (s_i, \ldots, s_{i-(n-1)})$ der interne Zustand eines Schieberegisters der Länge n über einem endlichen Körper $\mathbb{F}$ mit einer invertierbaren Aktualisierungsfunktion f und dem internen Zustand des Registers $S_t = (s_t, \ldots, s_{t-(n-1)})$ im Schritt t. Wir definieren L als $L(x_1, \ldots, x_n) := (x_2, \ldots, x_n, f^{-1}(x_1, \ldots, x_n))$. In jeder Aktualisierungsrunde führen wir die folgenden zwei Schritte aus:

1. Gib den ersten Eintrag s_i aus.

2. Aktualisiere den internen Zustand S_t zu S_{t-1} mit Hilfe von L:

$$S_{t-1} := L(S_t) = (s_{t+1}, \ldots, s_{t-(n-2)}, f^{-1}(S_t)).$$

Mit Hilfe von Schieberegistern konstruieren wir in Kapitel 3 sowohl eine Block- als auch eine Stromchiffre.

2.1.2 Kryptoanalyse

In diesem Abschnitt legen wir den theoretischen Grundstein für unsere Angriffe auf Kryptosysteme. Wir gehen darauf ein, in welchen Szenarien und mit welchen Methoden wir Angriffe durchführen und wann diese als erfolgreich gelten.

Jeder Angriff muss in einem festgelegten Angriffsszenario und mit einer Angriffsmethode ablaufen. Dabei beschreiben wir, welche Informationen der Angreifer von einem Orakel bekommt, welche Daten er wählen kann und wann der Angriff erfolgreich ist. Um Angriffsszenarien und -methoden zu beschreiben, verwenden wir Sicherheitsspiele, die in [KL07] beschrieben sind.

Definition 2.1.16. *Ein Sicherheitsspiel ist eine Sammlung der folgenden Algorithmen, die ein Herausforderer bereitstellt und ein Angreifer abfragen kann.*

`init` *Initialisiert das Spiel mit den dafür notwendigen Werten. Diese Funktion muss am Anfang einmal ausgeführt werden.*

`orakel` *Die Orakel-Algorithmen können vom Angreifer eine bestimmte Anzahl oft ausgeführt werden. Sie liefern Daten für die Analyse.*

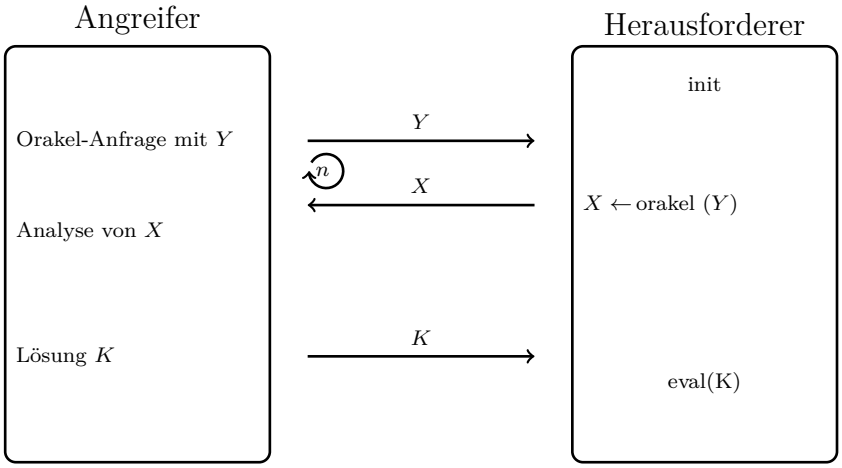


Abbildung 2.2: Allgemeines Sicherheitsspiel

`eval` *Die Funktion wird am Ende einmal ausgeführt und gibt an, ob der Angreifer gewonnen hat oder nicht.*

Ein Sicherheitsspiel stellen wir in graphischer Form mit stark vereinfachtem Pseudocode dar, wie wir in Abbildung 2.2 sehen. Dabei bezeichnet n die maximale Anzahl an Fragen, die ein Angreifer einem Orakel stellen kann. Der Angreifer sendet dem Herausforderer Orakel-Anfragen und analysiert diese. Danach schickt er die Lösung K, die mit `eval`(K) evaluiert wird. Falls `eval` „wahr“ ausgibt, hat der Angreifer gewonnen.

Ausgehend von Sicherheitsspielen beschreiben wir unsere Angriffsmethoden und -szenarien gleichzeitig. Die Sicherheitsspiele sind an unsere Zwecke angepasst, so dass wir kryptographische Protokolle definieren, die triviale Angriffe auf das Protokoll zulassen. Da wir keine Protokolle definieren wollen, sondern nur die Methoden und Szenarien von Angriffen beschreiben, haben wir die Sicherheitsmaßnahmen gegen solche Angriffe der Lesbarkeit halber weggelassen.

Wir gehen in unseren Angriffen immer davon aus, dass der Angreifer das Kryptosystem kennt. Damit befolgen wir das Prinzip von Kerkhoff:

„Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung der Chiffrierungsregel abhängen, sondern darf nur auf der Geheimhaltung des Schlüssels beruhen.“

Bevor wir zu der Erklärung der eingesetzten Methoden kommen, erklären wir ausgewählte Angriffszenarien.

Bekannter Geheimtext (Ciphertext-only -KC) Der Angreifer bekommt von einem Orakel eine bestimmte, große Anzahl an Geheimtexten.

Bekannter Klartext (Known Plaintext - KP) Der Angreifer bekommt von einem Orakel zusammengehörenden Klar- und Geheimtext.

Gewählter Klartext (Chosen Plaintext - CP) Der Angreifer wählt Klartexte und bekommt von einem Orakel die dazugehörigen Geheimtexte.

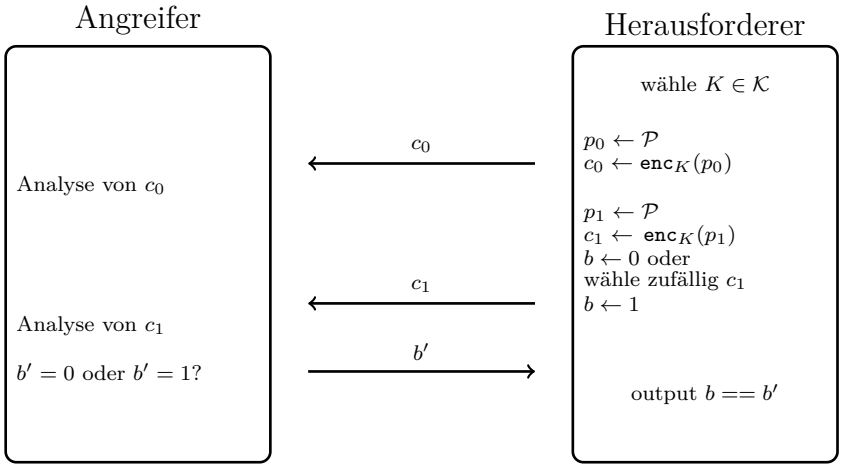


Abbildung 2.3: Sicherheitsspiel für einen Distinguisher in dem „Bekannter Geheimtext"-Szenario

Neben dem Angriffsszenario gibt es auch Methoden oder Ziele, die wir mit unserem Angriff erreichen möchten.

- Wir versuchen, einen Teil oder den ganzen Schlüssel eines Kryptosystems zu berechnen. Diese Methode bezeichnet man als **Schlüsselsuche - Key Recovery (KR)**. Damit hätten wir direkte Kontrolle über die Chiffre. Wir können entschlüsseln, verschlüsseln oder uns als jemand anderes ausgeben.
- Mit einem **Distinguisher** unterscheiden wir die Chiffre vom Zufall und beschleunigen so einen Brute-Force-Angriff. In der Abbildung 2.3 ist ein Sicherheitsspiel für einen Distinguisher in einem „Bekannter Geheimtext"-Szenario abgebildet.
- Ein einfacheres Ziel ist das Entschlüsseln von Geheimtext. In unseren Beispielen für Kryptosysteme aus Abschnitt 2.1.1 lässt sich dies durch eine statistische Analyse bewerkstelligen.

Es gibt auch Angriffe, bei denen wir nicht eindeutig zwischen Methode und Szenario unterscheiden können. Dazu gehört der **Verwandte Schlüssel-Angriff - Related Key (RK)**. Wir gehen zusätzlich zu einem normalen Szenario davon aus, dass sich der Schlüssel in gewissen Abständen (q Orakel-Aufrufe im Sicherheitsspiel) nur in einem geringen Maß ändert.

In den folgenden Kapiteln dieser Arbeit versuchen wir, den Schlüssel eines Kryptosystems zu berechnen. Dabei verschlüsseln wir gewählte Klartexte, benutzen also das „Gewählter Klartext"-Szenario. Das Pendant bei Stromchiffren ist das „Gewählter IV"-Szenario. Dabei generieren wir Schlüsselstrom mit anderen öffentlichen Startwerten. Die Anforderungen der beiden Szenarien sind gleich, weil wir den IV für das Kryptosystem wählen können. Als Klartext benutzen wir das leere Wort; lassen uns also nur den

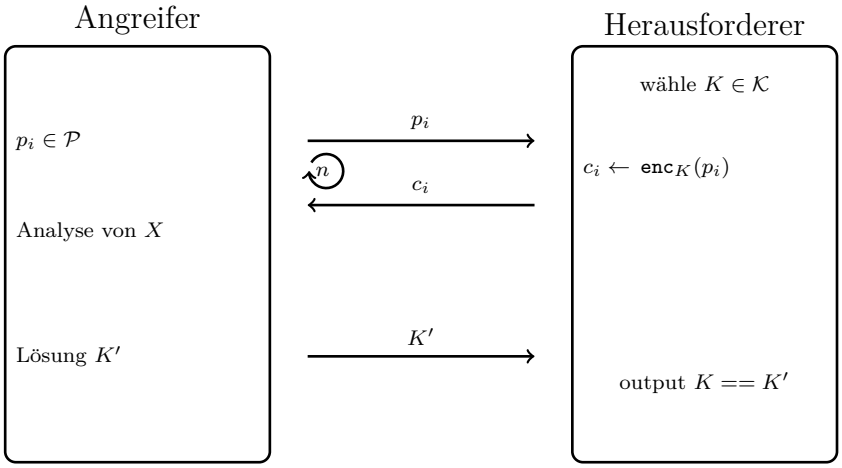


Abbildung 2.4: Sicherheitsspiel für eine KR in dem „Gewählter Klartext“-Szenario

Schlüsselstrom ausgeben. Das zugehörige Sicherheitsspiel sehen wir in Abbildung 2.4. Im weiteren Verlauf dieser Arbeit erklären wir diese Analysetechnik noch genauer.

In dieser Arbeit greifen wir konkrete Chiffren an. Dazu klären wir zunächst, was eine sichere Chiffre ist und wann wir eine Chiffre als „gebrochen“ betrachten.

Das ist kein einfaches Problem. Allein die Definition einer praktisch sicheren Chiffre können wir nicht formalisieren, weil wir nicht wissen, welche Angriffe wir betrachten müssen. Auch die benutzte Zeit messen wir nicht in Sekunden, weil die Rechenleistung stetig steigt und dieses Maß damit verzerrt. Aus diesem Grund leiten wir für diese Arbeit eine angepasste Definition ab.

So wollen wir stets den Schlüssel finden. In dieser Arbeit werden wir Geheimtexte einer Chiffre nicht vom Zufall unterscheiden oder geheime Nachrichten entschlüsseln. Das sind auch valide Angriffe, mit denen wir uns aber nicht beschäftigen.

Weiter müssen wir definieren, was Zeit ist. Wir müssen dazu eine Definition finden, die auch mit wachsender Computerleistung valide ist. Dazu nehmen wir Chiffren-Berechnungen als Einheit. Wie wir Zeit in Sekunden zu Zeit in Berechnungen einer Chiffre umrechnen, lernen wir in Abschnitt 6.1.3.

Diese Punkte fassen wir in Definition 2.1.17 zusammen. Anstatt zu definieren, wann ein Angriff erfolgreich ist, definieren wir zunächst, wann eine Chiffre sicher ist. Dazu nutzen wir als Grundlage die Definition einer (t, ϵ)-sicheren Chiffre aus [KL07] und ergänzen sie unseren Zwecken entsprechend.

Definition 2.1.17. *Sei $F = (\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ ein Kryptosystem.*

1. *Das Kryptosystem F ist gebrochen, wenn ein Angreifer mit hoher Wahrscheinlichkeit Zugang zu dem Schlüssel $K \in \mathcal{K}$ für alle Abbildungen $\mathtt{enc}_K$ und $\mathtt{dec}_K$ erlangen kann.*

2. *Wir messen die Zeit t in Berechnungen einer Chiffre.*

3. *Ein Kryptosystem F ist (t, ϵ)-sicher, wenn ein Angreifer in höchstens t Zeiteinheiten das Kryptosystem mit einer Wahrscheinlichkeit von höchstens ϵ brechen kann.*

Um aus dieser Definition einen erfolgreichen Angriff zu definieren, müssen wir zunächst ϵ bestimmen. Im Folgenden setzen wir $\epsilon = \frac{t}{|\mathcal{K}|}$, wie in [KL07] geraten wird. Im nächsten Beispiel erläutern wir, warum dieser Parameter gut gewählt ist.

Beispiel 2.1.18. *Die in dieser Arbeit betrachteten Chiffren haben einen Schlüsselraum der Größe* $\mathcal{K} = 2^{80}$.

1. *Der Parameter* $\epsilon = \frac{t}{|\mathcal{K}|}$ *schließt den Brute-Force-Angriff durch* $t = |\mathcal{K}|$ *ein. In einem Brute-Force-Angriff testen wir alle Möglichkeiten für den Schlüssel. So finden wir sicher den richtigen. Also sind die Chiffren nicht* $(2^{80}, 1)$*-sicher. Das ist die natürliche Grenze eines jeden Kryptosystems.*

2. *Wenn es keinen Angriff gibt, der weniger als* 2^{50} *Chiffren-Berechnungen benötigt, ist die Chiffre* $(2^{50}, \delta)$*-sicher für ein kleines* $\delta > 0$. *Mit unserem Parameter ist* $\delta = \frac{2^{50}}{2^{80}} = 2^{-30}$.

   Das heißt, wenn wir einen Angriff in unter 2^{50} *Chiffre-Berechnungen finden, der eine Erfolgswahrscheinlichkeit von über* 2^{-30} *hat, so ist der Angriff erfolgreich.*

Das Beispiel motiviert die folgende Definition eines erfolgreichen Angriffs.

Definition 2.1.19. *Ein Angriff ist erfolgreich, wenn wir die Chiffre mit hoher Wahrscheinlichkeit* $\mathcal{P}(_{success}) = 1 - \delta$ *für ein kleines* $\delta > 0$ *brechen und dabei weniger als* $|\mathcal{K}|$ *Zeit und weniger als* $|\mathcal{K}|$ *Daten benötigen.*

Im Prinzip drehen wir dabei einfach die Definition 2.1.17 um. Anstatt zu fordern, dass jeder Angriff eine geringe Erfolgswahrscheinlichkeit hat, müssen wir zeigen, dass es einen Angriff in „wenig“ Zeit gibt, der mit hoher Wahrscheinlichkeit den Schlüssel findet.

Die Datenkomplexität nehmen wir in unsere Definition mit auf, weil wir ansonsten zu jeder Chiffre einen trivialen Angriff konstruieren können. Wir speichern zu jedem Schlüssel und einem Klartext und Startwert den Geheimtext. Danach schauen wir anhand eines Referenzpaars nach, welche Kandidaten wir für den Schlüssel erhalten und testen diese wenigen Kandidaten. Das Nachschauen in einer Liste ist sehr effizient und benötigt weniger Laufzeit als das Ausführen einer Chiffre, so dass wir einen erfolgreichen Angriff hätten. Aber so viele Daten zu speichern ist nicht möglich. Die Datenkomplexität eines Angriffs ist genauso ein limitierender Faktor wie die Zeit. Daten messen wir in der Ausgabe der Chiffre. Also in Bits für Stromchiffren und Klartext-Geheimtext-Paaren für Blockchiffren.

Unsere Angriffe haben meistens eine Erfolgswahrscheinlichkeit von $\mathcal{P}(_{success}) \approx 1$. Wir gehen davon aus, dass $\delta \leqslant 0,05$ sein muss, damit ein Angriff erfolgreich ist.

2.2 Lösen von polynomiellen Gleichungssystemen

In dieser Arbeit führen wir sogenannte algebraische Kryptoanalyse durch. Dabei stellen wir ein Kryptosystem durch ein nichtlineares multivariates Gleichungssystem über einem

endlichen Körper dar. Danach versuchen wir, diese Systeme mit Hilfe bestimmter Algorithmen so zu vereinfachen, dass wir mit einigen Paaren von Klartext und Geheimtext schnell auf den Schlüssel schließen können.

Das Lösen von zufälligen nichtlinearen, multivariaten quadratischen Gleichungssystemen über einem endlichen Körper ist ein NP-schweres Problem, das sogenannte MQ-Problem (siehe zum Beispiel [GJ90]). Wir kennen also keinen Algorithmus mit polynomieller Laufzeit, der ein solches Gleichungssystem löst. Allerdings können manche Systeme, die viel Struktur aufweisen, effizient gelöst werden, wie in [FJ03] gezeigt wurde. Solche Gleichungssysteme können manchmal von schlecht konstruierten Chiffren abgeleitet werden.

Im weiteren Verlauf dieser Arbeit meinen wir immer, wenn wir von einem Gleichungssystem oder System schreiben, ein multivariates, polynomielles Gleichungssystem über einem endlichen Körper.

In diesem Abschnitt beschreiben wir die Theorie für ausgewählte Algorithmen zum Lösen von polynomiellen Gleichungssystemen, sogenannten Solvern. Dafür stellen wir in Abschnitt 2.2.1 Monomordnungen vor. Diese erlauben es uns, in einem Gleichungssystem Monome zu sortieren. Dadurch können wir auch eine Division für multivariate Polynome definieren. Dieses Wissen benutzen wir, um in Abschnitt 2.2.2 einen modernen Algorithmus von Faugère aus [Fau02], genannt $\mathbf{F_4}$, vorzustellen. F_4 liefert uns eine Gröbnerbasis, mit der wir ein Gleichungssystem lösen können. Gröbnerbasen-Algorithmen liefern uns immer eine Lösung. Das Berechnen einer Gröbnerbasis verbraucht aber viel Zeit und Speicher und ist in vielen Fällen nicht nötig, um das System zu lösen. Dafür stellen wir in Abschnitt 2.2.3 einen alternativen Algorithmus für die Lösung eines Gleichungssystems vor.

2.2.1 Multivariate Polynome

In diesem Abschnitt wiederholen wir kurz wichtige Definitionen und legen den theoretischen Grundstein für das Lösen von Gleichungssystemen. Dabei folgen wir grob den Ausführungen von [Que11].

Sei $\mathbb{F}$ ein Körper. Wir betrachten Polynome $f(x_1, \ldots, x_n)$ in n Variablen mit Koeffizienten in $\mathbb{F}$. Solche Polynome sind endliche Summen von sogenannten Termen der Form $ax_1^{\beta_1} \cdots x_n^{\beta_n}$ mit $a \in \mathbb{F}$ und $\beta_i \in \mathbb{N}_0, i = 1, \ldots, n$. Wir nennen einen Term $x_1^{\beta_1} \cdots x_n^{\beta_n}$ mit Koeffizient $a = 1$ ein Monom. Die Menge aller Polynome in n Variablen mit Koeffizienten in $\mathbb{F}$ bezeichnen wir mit $\mathbb{F}[x_1, \ldots, x_n]$. In $\mathbb{F}[x_1, \ldots, x_n]$ haben wir die bekannte Addition und Multiplikation von Polynomen und bezüglich dieser Operationen ist $\mathbb{F}[x_1, \ldots, x_n]$ ein kommutativer Ring.

Das Erzeugnis der Polynome $f_1, \ldots, f_s$

$$I = \langle f_1, \ldots, f_s \rangle = \left\{ \sum_{i=1}^{s} u_i f_i \mid u_i \in \mathbb{F}[x_1, \ldots, x_n], i = 1, \ldots, s \right\}$$

ist offensichtlich ein Ideal in $\mathbb{F}[x_1, \ldots, x_n]$. Für $f, g \in I \neq \emptyset$ und $h \in \mathbb{F}[x_1, \ldots, x_n]$ gilt also $f + g \in I$ und $hf \in I$. Die Menge $\{f_1, \ldots, f_s\}$ heißt ein Erzeugendensystem von I. Ein Ideal I hat im Allgemeinen viele verschiedene Erzeugendensysteme mit verschiedenen Eigenschaften.

Im weiteren Verlauf benötigen wir noch folgende Eigenschaften des Rings $\mathbb{F}[x_1, \ldots, x_n]$.

Satz 2.2.1. *Folgende Bedingungen sind für einen kommutativen Ring R äquivalent.*

1. *Sei $I \subset R$ ein beliebiges Ideal in R. Dann existieren Elemente $f_1, \ldots, f_s \in R$ mit $I = \langle f_1, \ldots, f_s \rangle$.*
2. *Sei $I_1 \subset I_2 \subset \ldots \subset I_n \subset \ldots$ eine aufsteigende Kette von Idealen in R. Dann existiert ein N, so dass $I_N = I_{N+1} = I_{N+2} = \ldots$. Die Kette von Idealen wird also stationär.*

Beweis: Wir führen den Beweis hier nicht an und verweisen auf [LA94]. □

Definition 2.2.2. *Ein Ring R, der die Bedingungen in Satz 2.2.1 erfüllt, heißt noethersch.*

In einem noetherschen Ring sind nach Satz 2.2.1 alle Ideale endlich erzeugt. Dass auch $\mathbb{F}[x_1, \ldots, x_n]$ ein noetherscher Ring ist, zeigt der Hilbertsche Basissatz.

Satz 2.2.3. *Sei $\mathbb{F}$ ein Körper. Dann ist $\mathbb{F}[x_1, \ldots, x_n]$ ein noetherscher Ring.*

Beweis: Auch hier verzichten wir auf den Beweis und verweisen auf [LA94]. □

Um multivariate Polynome in einem Gleichungssystem zu sortieren, verwenden wir Monomordnungen. Ein Monom können wir schreiben als $x^\alpha = x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ mit Exponenten $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}_0^n$. Dies liefert einen Isomorphismus zwischen der Menge der Monome x^α und $\mathbb{N}_0^n$, wie in [CLO07] gezeigt wird. Damit brauchen wir Ordnungen auf $\mathbb{N}_0^n$, die die algebraische Struktur des Polynomrings $\mathbb{F}[x_1, \ldots, x_n]$ erhalten.

Definition 2.2.4. *Eine Monomordnung $<$ auf $\mathbb{F}[x_1, \ldots, x_n]$ ist eine Ordnungsrelation $<$ auf $\mathbb{N}_0^n$ oder äquivalent auf der Menge der Monome $x^\alpha, \alpha \in \mathbb{N}_0^n$, die folgenden Bedingungen genügt.*

1. *$<$ ist eine totale Ordnung auf $\mathbb{N}_0^n$, das heißt, für ein Paar von Monomen x^α und x^β gilt immer genau eine der Aussagen $x^\alpha > x^\beta, x^\alpha = x^\beta$ oder $x^\alpha < x^\beta$.*
2. *Wenn $\alpha, \beta, \gamma \in \mathbb{N}_0^n$ und $\alpha > \beta$, dann ist $\alpha + \gamma > \beta + \gamma$.*
3. *$<$ ist eine Wohlordnung auf $\mathbb{N}_0^n$, das heißt, jede nicht leere Untermenge von $\mathbb{N}_0^n$ hat ein kleinstes Element bezüglich $<$.*

Im Folgenden werden wir eine Monomordnung der Einfachheit halber Ordnung nennen.

Folgende Ordnungen haben eine besondere Bedeutung in der Praxis.

Definition 2.2.5. *Seien $\alpha = (\alpha_1, \ldots, \alpha_n), \beta = (\beta_1, \ldots, \beta_n) \in \mathbb{N}_0^n$. Wir nennen eine Ordnung:*

1. *lexikografische Ordnung $<_{lex}$, oder kurz* lex-*Ordnung, falls gilt:*

$$x^\alpha >_{lex} x^\beta \Leftrightarrow \textit{Der erste nicht verschwindende Eintrag (von links) der Differenz } \alpha - \beta \in \mathbb{Z}^n \textit{ ist positiv.}$$

2. *gradlexikografische Ordnung* $<_{deglex}$*, oder kurz* deglex-*Ordnung, falls gilt:*

$$x^\alpha >_{deglex} x^\beta \Leftrightarrow |\alpha| = \sum_{i=1}^n \alpha_i > \sum_{i=1}^n \beta_i = |\beta| \text{ oder}$$
$$|\alpha| = |\beta| \text{ und } \alpha >_{lex} \beta.$$

3. *gradreverslexikografische Ordnung* $<_{degrevlex}$*, oder kurz* degrevlex-*Ordnung, falls gilt:*

$$x^\alpha >_{degrevlex} x^\beta \Leftrightarrow |\alpha| > |\beta| \text{ oder } |\alpha| = |\beta| \text{ und der erste}$$
$$\text{nicht verschwindende Eintrag (von rechts)}$$
$$\text{der Differenz } \alpha - \beta \in \mathbb{Z}^n \text{ ist negativ.}$$

Diese Ordnungen sind Monomordnungen, wie in [CLO07] gezeigt wird. Mit Hilfe einer *lex*-Ordnung kann man die Lösung von polynomiellen Gleichungssystemen sofort an einer Gröbnerbasis ablesen. Jedoch sind Gröbnerbasen zu einer *lex*-Ordnung nur schwer zu berechnen. Eine Gröbnerbasis zu einer *degrevlex*-Ordnung hingegen ist normalerweise schneller berechenbar. Man kann eine Gröbnerbasis für eine Ordnung mit Hilfe des FGLM-Algorithmus aus [FGLM93] oder mit dem „Gröbner Walk“ aus [CKM97] in eine Gröbnerbasis bezüglich einer anderen Ordnung konvertieren. Bevor wir uns einem Beispiel zuwenden, folgen noch eine paar Notationen für Polynome in mehreren Variablen.

Definition 2.2.6. *Sei* $f = \sum_\alpha a_\alpha x^\alpha \neq 0$ *ein Polynom in* $\mathbb{F}[x_1, \ldots, x_n]$ *und* $<$ *eine Monomordnung.*

1. *Der Multigrad* $multideg(f)$ *von* f *ist*

$$multideg(f) = \max_{<}\{\alpha \in \mathbb{N}_0^n \mid a_\alpha \neq 0\}.$$

2. *Der Leitkoeffizient* $LC(f)$ *von* f *ist definiert durch*

$$LC(f) = a_{multideg(f)} \in \mathbb{F}.$$

3. *Das Leitmonom* $LM(f)$ *von* f *ist*

$$LM(f) = x^{multideg(f)}.$$

4. *Der Leitterm* $LT(f)$ *von* f *ist gegeben durch*

$$LT(f) = LC(f) \cdot LM(f).$$

Im Rest dieser Arbeit bezeichnen wir den totalen Grad eines Polynoms $\deg(f) = \max\{\sum_{i=1}^n \alpha_i \,|\, a_\alpha \neq 0\}$ der Einfachheit halber als Grad des Polynoms. Im folgenden Beispiel sehen wir, dass Ordnungen auch verändert werden können, indem man die Variablen umsortiert.

Beispiel 2.2.7. *Wir betrachten das Polynom* $f = 5xy^2z+3z^2-5x^2y+7x^2z^2 \in \mathbb{Q}[x,y,z]$. *Sortieren wir die Monome von* f *in absteigender Reihenfolge bezüglich der* lex-*Ordnung mit* $x > y > z$, *dann ergibt sich*

$$f = -5x^2y + 7x^2z^2 + 5xy^2z + 3z^2$$

mit Multigrad $multideg(f) = (2,1,0)$ *und Leitkoeffizient* $LC(f) = -5$.
Sortieren wir die Monome von f *wieder bezüglich der* lex-*Ordnung, aber mit* $y > x > z$, *so ist*

$$f = 5y^2xz - 5yx^2 + 7x^2z^2 + 3z^2$$

mit Multigrad $(2,1,1)$ *und Leitmonom* xy^2z.
Bezüglich der degrevlex-*Ordnung mit* $x > y > z$ *erhalten wir*

$$f = 5xy^2z + 7x^2z^2 - 5x^2y + 3z^2$$

mit Leitterm $LT(f) = 5xy^2z$.

Für die Algorithmen aus den folgenden Abschnitten benötigen wir eine Division für Polynome mehrerer Veränderlicher. Die Idee ist, Terme von einem Polynom $f \in \mathbb{F}[x_1,\ldots,x_n]$ bei der Division durch Polynome $f_1,\ldots,f_s$ mit Hilfe der Leitterme $LT(f_i), i = 1,\ldots,n$ so zu eliminieren, dass die neuen Terme kleiner bezüglich der gewählten Ordnung sind.

Zunächst betrachten wir die Division von $f \in \mathbb{F}[x_1,\ldots,x_n]$ durch ein Polynom $g \in \mathbb{F}[x_1,\ldots,x_n]$. Dazu sei $<$ eine feste Monomordnung.

Definition 2.2.8. *Seien* $f,g,h \in \mathbb{F}[x_1,\ldots,x_n]$ *mit* $g \neq 0$. *Wir sagen* f *reduziert zu* h *modulo* g *und schreiben* $[f]_g = h$ *genau dann, wenn* $LM(g)$ *einen nicht verschwindenden Term* $a_\alpha x^\alpha$ *von* f *teilt und*

$$h = f - \frac{a_\alpha x^\alpha}{LT(g)} g$$

ist.

In dieser Definition haben wir den Term $a_\alpha x^\alpha$ von f subtrahiert und durch Terme von strikt kleinerer Ordnung ersetzt. Die Algorithmen zum Lösen von multivariaten, polynomiellen Gleichungssystemen dividieren Polynome zur selben Zeit durch mehrere Polynome. In der nächsten Definition erweitern wir dazu Definition 2.2.8.

Definition 2.2.9. *Seien* $f,h,f_1,\ldots,f_s \in \mathbb{F}[x_1\ldots,x_n]$ *Polynome mit* $f_i \neq 0, 1 \leqslant i \leqslant s$ *und sei* F *die Menge aller Polynome im System* $F = \{f_1,\ldots,f_s\}$.

1. *Ein Polynom* f *reduziert zu* h *modulo* F, *geschrieben* $[f]_F = h$, *genau dann, wenn eine Sequenz von Indizes* $i_1,i_2,\ldots,i_t \in \{1,\ldots,s\}$ *und eine Sequenz von Polynomen* $h_1,\ldots,h_{t-1} \in \mathbb{F}[x_1\ldots,x_n]$ *existieren, so dass*

$$[f]_{f_{i_1}} = h_1, [h_1]_{f_{i_2}} = h_2, \ldots, [h_{t-1}]_{f_{i_t}} = h.$$

2. *Ein Polynom r heißt reduziert bezüglich F, wenn $r = 0$ gilt oder kein Monom in r durch ein $LM(f_i), i = 1, \ldots, s$ teilbar ist, also wenn r modulo F nicht reduziert werden kann.*

3. *Wenn $[f]_F = r$ und r reduziert ist bezüglich F, dann nennen wir r Rest von f bei Division durch F.*

In dieser Definition hängen die Ergebnisse stark von der Reihenfolge der Polynome ab, mit denen wir reduzieren, wie wir in Beispiel 2.2.10 sehen. Es gibt aber auch bestimmte Erzeugendensysteme eines Ideals, bei denen das nicht so ist. Diese Erzeugendensysteme heißen Gröbnerbasen und werden später vorgestellt.

Der Reduktionsprozess erlaubt es uns, den Divisionsalgorithmus 2.1 einzuführen. Die Funktion `div` bekommt das Polynom f und die Divisoren $f_1, \ldots, f_s$ und wendet Definition 2.2.8 iterativ an. Danach gibt die Funktion Koeffizienten $u_1, \ldots, u_s \in \mathbb{F}[x_1, \ldots, x_n]$ und $r \in \mathbb{F}[x_1, \ldots, x_n]$ aus, so dass wir f als $f = u_1 f_1 + \ldots + u_s f_s + r$ schreiben können. Dabei ist r reduziert bezüglich $\{f_1, \ldots, f_s\}$ und es gilt

$$\max\{LM(u_1)LM(f_1), \ldots, LM(u_s)LM(f_s), LM(r)\} = LM(f).$$

Algorithmus 2.1 Divisionsalgorithmus für multivariate Polynome

function `div`$(f, f_1, \ldots, f_s)$:
$h \leftarrow f$
$r \leftarrow 0$
for $i \leftarrow 1$ **to** s **do**
  $u_i \leftarrow 0$
end for
while $h \neq 0$ **do**
  if $\exists i : LM(f_i) | LM(h)$ **then**
    choose smallest $i : LM(f_i) | LM(h)$
    $u_i = u_i + \frac{LT(h)}{LT(f_i)}$
    $h = h + \frac{LT(h)}{LT(f_i)} f_i$
  else
    $r = r + LT(h)$
    $h = h - LT(h)$
  end if
end while
return $r, \{u_i | i = 1, \ldots, s\}$

Für den Beweis der Korrektheit des Algorithmus zeigen wir als erstes, dass der Algorithmus terminiert. Bei jedem Durchlauf subtrahieren wir den Leitterm von h, bis dies nicht mehr möglich ist. So gibt der Algorithmus eine Sequenz $h_1, h_2, \ldots$ von h aus, wobei h_{i+1} aus h_i hervorgeht, indem wir $LT(h_i)$ und eventuell Terme kleinerer Ordnung subtrahieren. Für alle i erhalten wir also $LM(h_{i+1}) > LM(h_i)$. Da wir eine Wohlordnung voraussetzen, muss die Sequenz der h_i stoppen.

Aus dem ersten Teil des Beweises und da wir am Anfang des Algorithmus $h = f$ setzen, haben wir in jedem Durchlauf des Algorithmus $LM(h) \leqslant LM(f)$. Für jedes i erhalten wir u_i durch Addition von $\frac{LT(h)}{LT(f_i)}$, wobei $\frac{LT(h)}{LT(f_i)} f_i$ den Leitterm von h eliminiert. Also ist $LM(u_i)LM(f_i) \leqslant LM(f)$. Weiter erhalten wir r durch Addition der Terme $LT(h)$. Damit ist $LM(r) \leqslant LM(f)$. □

Mit Hilfe der Darstellung $f = u_1 f_1 + \ldots + u_s f_s + r$ von f aus Algorithmus 2.1 können wir testen, ob das Polynom f im Ideal I enthalten ist. Es ist $f - r \in \langle f_1, \ldots, f_s \rangle$. Damit ist $f \in \langle f_1, \ldots, f_s \rangle$, wenn $r = 0$. Die Rückrichtung ist im Allgemeinen nicht richtig, wie der zweite Teil von Beispiel 2.2.10 zeigt.

Beispiel 2.2.10. *1. Betrachten wir die Polynome $f = 2xy^2 + x - 6y^3 + 1$ und $g = xy - 2y^3$ in $\mathbb{Q}[x, y]$ bezüglich* lex-*Ordnung mit $x > y$. Dann ist $[f]_g = x + 4y^4 - 6y^3 + 1$, da wir $LT(f) = 2xy^2$ mit Hilfe von $LT(g) = xy$ eliminiert haben.*

Wenn wir die deglex-*Ordnung mit $x > y$ verwenden, ist $[f]_g = 2xy^2 - 3xy + x + 1$ und wir haben $-6y^3$ mit Hilfe von $LT(g) = -2y^3$ eliminiert. Dadurch haben wir den Leitterm von f nicht eliminiert.*

2. Seien nun das Polynom $f = y^3x^2 - x \in \mathbb{Q}[x, y]$ und das Ideal $I = \langle f_1, f_2 \rangle \subset \mathbb{Q}[x, y]$ mit $f_1 = yx^2 - 2y$ und $f_2 = 2y^3 - x$ gegeben. Wir setzen $F = \{f_1, f_2\}$. Bezüglich der deglex-*Ordnung mit $y > x$ erhalten wir mit dem Divisionsalgorithmus $[f]_{f_1} = 2y^3 - x$ und $[2y^3 - x]_{f_2} = 0$. Also ist $[f]_F = 0$ und damit $f = y^2 f_1 + f_2 \in I$.*

Wenn wir die Reihenfolge der Reduktion vertauschen, also f zuerst mit f_2 reduzieren, ergibt sich $[f]_{f_2} = x^3/2 - x$ und $x^3/2 - x$ ist der Rest von f bei Division mit F. Damit erhalten wir einen nicht verschwindenden Rest, aber f ist im Ideal I.

Unser Divisionsalgorithmus ist, wie im Beispiel 2.2.10 gezeigt, nicht eindeutig. Das können wir auch im eindimensionalen Fall beobachten. Wählen wir zum Beispiel $f = x, f_1 = x^2$ und $f_2 = x^2 - x$. Dann ist f bereits der Rest bei Division mit $\{f_1, f_2\}$, aber es ist $f = f_1 - f_2 \in \langle f_1, f_2 \rangle$. Dieses Problem wird gelöst, indem wir ein besseres Erzeugendensystem für $\langle f_1, f_2 \rangle$ angeben. Dieses Erzeugendensystem ist der größte gemeinsame Teiler $x = ggT(x^2, x^2 - x)$ der beiden Polynome f_1 und f_2.

Da der Polynomring in mehreren Veränderlichen kein Hauptidealring ist, können wir dieses Vorgehen nicht übertragen. Wenn wir über eine Gröbnerbasis für ein Ideal verfügen, wird der Divisionsalgorithmus 2.1 eindeutig und das Problem ist gelöst.

2.2.2 Ein moderner Gröbnerbasen-Algorithmus

In diesem Abschnitt führen wir Gröbnerbasen für Ideale ein und zeigen die Existenz einer Gröbnerbasis für nichtleere Ideale des $\mathbb{F}[x_1, \ldots, x_n]$. Daraufhin führen wir den Buchberger-Algorithmus ein, auf dem alle modernen Gröbnerbasen-Algorithmen basieren. Danach stellen wir Faugères F_4-Algorithmus vor. Dieser beschleunigt den Buchberger-Algorithmus mit Hilfe linearer Algebra und wird heutzutage in vielen Implementierungen für Gröbnerbasen benutzt.

Der Buchberger-Algorithmus

Wir beginnen direkt mit der Definition von Gröbnerbasen gemäß [LA94].

Definition 2.2.11. *Sei $I \subset \mathbb{F}[x_1, \ldots, x_n]$ ein Ideal. Eine Menge von nicht verschwindenden Polynomen $G = \{g_1, \ldots, g_t\} \subset I$ heißt genau dann Gröbnerbasis von I, wenn für alle $f \in I, f \neq 0$ ein $i \in \{1, \ldots, t\}$ existiert, so dass $LM(g_i)$ ein Teiler von $LM(f)$ ist.*

Nach Definition 2.2.11 existieren zu einer Gröbnerbasis G für ein Ideal I keine nicht verschwindenden Polynome in I, die reduziert bezüglich G sind.

Aus der Definition geht weder die Existenz noch ein Algorithmus zur Berechnung einer Gröbnerbasis hervor. Damit beschäftigen wir uns im Folgenden. Wir beginnen mit einer Charakterisierung von Gröbnerbasen. Dabei bezeichnet $LT(S) = \{LT(s) \mid s \in S\}$ die Menge der Leitterme für eine Menge $S \subset \mathbb{F}[x_1, \ldots, x_n]$.

Satz 2.2.12. *Sei $I \neq \{0\}$ ein Ideal und f ein Polynom in $\mathbb{F}[x_1, \ldots, x_n]$. Dann sind folgende Aussagen für eine Menge von nicht verschwindenden Polynomen $G = \{g_1, \ldots, g_t\} \subset I$ äquivalent.*

1. *G ist eine Gröbnerbasis von I.*
2. *Es ist $f \in I$ genau dann, wenn $[f]_G = 0$*
3. *Es ist $f \in I$ genau dann, wenn $f = \sum_{i=1}^{t} h_i g_i$ mit $h_i \in \mathbb{F}[x_1, \ldots, x_n]$ und $LM(f) = \max_{1 \leqslant i \leqslant t}\{LM(h_i)LM(g_i)\}$.*
4. *Es gilt $\langle LT(G)\rangle = \langle LT(I)\rangle$.*

Beweis: Den Beweis finden wir in [LA94]. □

Wenn $G = \{g_1, \ldots, g_t\}$ eine Gröbnerbasis für ein Ideal I ist, dann folgt aus dem zweiten Teil von Satz 2.2.12, dass $I = \langle G\rangle$. Also ist eine Gröbnerbasis für ein Ideal I immer ein Erzeugendensystem von I.

Um die Existenz einer Gröbnerbasis für ein Ideal I zu zeigen, brauchen wir noch Kenntnisse über Ideale, die von Monomen erzeugt werden.

Lemma 2.2.13. *Sei I ein von einer Menge S aus nicht verschwindenden Monomen erzeugtes Ideal und $f \in \mathbb{F}[x_1, \ldots, x_n]$. Dann ist $f \in I$ genau dann, wenn für jedes Monom x^α aus f ein Monom $x^\beta \in S$ existiert, so dass x^β ein Teiler von x^α ist. Darüber hinaus existiert eine endliche Untermenge $S_0 \subset S$ mit $I = \langle S_0\rangle$.*

Beweis: Sei $f \in I$. Dann gilt

$$f = \sum_{i=1}^{l} h_i x^{\beta_i}$$

mit $h_i \in \mathbb{F}[x_1, \ldots, x_n]$ und $x^{\beta_i} \in S$ für $i = 1, \ldots, l$. Auf der rechten Seite dieser Gleichung wird jeder Term von einem $x^{\beta_i} \in S$ geteilt. Also wird auch die linke Seite von x^{β_i} geteilt.

Umgekehrt, wenn jedes Monom x^α aus f von einem Monom $x^\beta \in S$ geteilt wird, dann ist $x^\alpha \in I = \langle S\rangle$ und damit $f \in I$.

Nach dem Hilbertschen Basissatz 2.2.3 ist jedes betrachtete Ideal I endlich erzeugt. Nach dem ersten Teil dieses Lemmas wird jedes Monom in dem endlichen Erzeugendensystem durch ein Element in S geteilt. Die endliche Menge S_0 dieser Teiler ist ein Erzeugendensystem von I. □

Nach diesen Vorbereitungen zeigen wir im nächsten Korollar die Existenz einer Gröbnerbasis für ein beliebiges Ideal.

Korollar 2.2.14. *Jedes Ideal $I \neq 0$ von $\mathbb{F}[x_1, \ldots, x_n]$ hat eine Gröbnerbasis.*

Beweis: Nach Lemma 2.2.13 hat $\langle LT(I)\rangle$ ein endliches Erzeugendensystem. Dieses stellen wir als $\{LT(g_1), \ldots, LT(g_t)\}$ mit $g_1, \ldots, g_t \in I$ dar. Sei $G = \{g_1, \ldots, g_t\}$. Dann ist $LT(G) = LT(I)$ und G nach Satz 2.2.12 Teil 4 eine Gröbnerbasis. □

Der nächste Satz zeigt, dass der Divisionsalgorithmus 2.1 eindeutig ist, wenn wir durch eine Gröbnerbasis teilen.

Satz 2.2.15. *Sei $G = \{g_1, \ldots, g_t\} \subset \mathbb{F}[x_1, \ldots, x_n]$ eine Menge von nicht verschwindenden Polynomen. Genau dann ist G eine Gröbnerbasis zu dem Ideal $\langle G\rangle$, wenn für alle $f \in \mathbb{F}[x_1, \ldots, x_n]$ der Rest bei Division mit G eindeutig ist.*

Beweis: Wir verweisen auf [LA94]. □

Eine Gröbnerbasis bezüglich einer Monomordnung ist im Allgemeinen keine Gröbnerbasis bezüglich einer anderen Monomordnung.

Unser nächstes Ziel ist die Berechnung einer Gröbnerbasis für ein Ideal I bezüglich einer Monomordnung $<$. Sei dazu $I = \langle f_1, \ldots, f_s\rangle \subset \mathbb{F}[x_1, \ldots, x_n]$ ein Ideal und $F = \{f_1, \ldots, f_s\}$ mit $f_i \neq 0, i = 1, \ldots, s$. Die Menge F ist genau dann eine Gröbnerbasis, wenn für alle $f \in I$ ein $i \in 1, \ldots, s$ existiert, so dass $LM(f_i)$ ein Teiler von $LM(f)$ ist. Also müssen wir den Fall ausschließen, dass das Leitmonom eines Elements aus I nicht durch $LM(f_i)$ geteilt wird. Wegen $f \in I$ gilt $f = \sum_{i=1}^{s} h_i f_i$ für $h_i \in \mathbb{F}[x_1, \ldots, x_n]$. Das heißt, der Fall tritt ein, wenn der größte der $LM(h_i f_i) = LM(h_i)LM(f_i)$ sich aus der Summe kürzt. Diesen Fall betrachten wir.

Definition 2.2.16. *Sei $0 \neq f, g \in \mathbb{F}[x_1, \ldots, x_n]$ und bezeichne $L = kgV(LM(f), LM(g))$. Das Polynom*

$$S(f, g) = \frac{L}{LT(f)} f - \frac{L}{LT(g)} g$$

heißt das S*-Polynom von f und g.*

Im nächsten Satz, der nach seinem Begründer Bruno Buchberger benannt ist, zeigen wir, dass uns S-Polynome einen Algorithmus für die Berechnung einer Gröbnerbasis liefern.

Satz 2.2.17. *Sei $G = \{g_1, \ldots, g_t\} \subset \mathbb{F}[x_1, \ldots, x_n]$ eine Menge von Polynomen ungleich Null. Genau dann ist G eine Gröbnerbasis des Ideals $I = \langle g_1, \ldots, g_t\rangle$, wenn für alle $i \neq j$*

$$[S(g_i, g_j)]_G = 0$$

gilt.

Beweis: Für den Beweis verweisen wir auf [LA94]. □

Der Satz von Buchberger liefert uns eine Strategie, mit der wir Gröbnerbasen berechnen. Wir reduzieren die S-Polynome eines Erzeugendensystems für ein Ideal. Wenn ein Rest ungleich Null ist, nehmen wir diesen Rest in das Erzeugendensystem auf. Dieses Vorgehen wiederholen wir solange, bis alle S-Polynome zu Null reduzieren, also eine Gröbnerbasis vorliegt.

Im Algorithmus 2.2 beschreiben wir den sogenannten Buchberger-Algorithmus. Die Funktion `groebner` berechnet für ein Erzeugendensystem $F = \{f_1, \ldots, f_s\} \in \mathbb{F}[x_1, \ldots, x_n]$ eines Ideals $I \neq 0$ eine Gröbnerbasis G. Die Menge B sortiert dabei die zu betrachtenden Paare von Polynomen, für die ein S-Polynom berechnet werden muss.

Algorithmus 2.2 Buchberger-Algorithmus zur Erstellung einer Gröbnerbasis

```
function groebner(F):
G ← F
B ← {(i, j) | 1 ⩽ i < j ⩽ |G|}
while B ≠ ∅ do
  choose (i, j) ∈ B
  B = B\{(i, j)}
  h ← [S(f_i, f_j)]_G
  if h ≠ 0 then
    B = B ∪ {(i, |G| + 1) | i = 1, ..., |G|}
    G.append(h)
  end if
end while
return G
```

Wir zeigen, dass der Algorithmus terminiert. Dazu nehmen wir an, dass der Algorithmus nicht abbricht. Dann konstruiert der Algorithmus eine strikt aufsteigende Kette von Mengen

$$G_1 \subset G_2 \subset G_3 \subset \ldots.$$

Für jedes G_i gilt $G_i = G_{i-1} \cup h$ mit einem nicht verschwindenden Rest $h \in I$. Dieses $h \in I$ entsteht bei der Reduktion eines S-Polynoms durch G_{i-1}. Also ist $LT(h) \notin \langle LT(G_{i-1})\rangle$ und wir erhalten eine strikt aufsteigende Kette von Idealen

$$\langle LT(G_1)\rangle \subset \langle LT(G_2)\rangle \subset \langle LT(G_3)\rangle \subset \ldots.$$

Dies ist ein Widerspruch zum Hilbertschen Basissatz 2.2.3.

Es ist $F \subset G \subset I$ und damit $I = \langle f_1, \ldots, f_s\rangle \subset \langle g_1, \ldots, g_t\rangle \subset I$. Also ist G ein Erzeugendensystem für I. Weiterhin gilt für zwei Polynome $g_i, g_j \in G$ durch Konstruktion $[S(g_i, g_j)]_G = 0$. Also ist G eine Gröbnerbasis von I nach dem Satz von Buchberger 2.2.17. □

Zum besseren Verständnis betrachten wir ein Beispiel.

Beispiel 2.2.18. *Sei* $f_1 = xy^2 + y^3 + x^2y$, $f_2 = y + x$ *und* $f_3 = y^2x \in \mathbb{Q}[x, y]$. *Wir wollen eine Gröbnerbasis* G *für* $I = \langle f_1, f_2, f_3 \rangle$ *bezüglich* lex-*Ordnung mit* $x > y$ *berechnen.*

Wir setzen $G = \{f_1, f_2, f_3\}$ *und* $B = \{(1,2), (1,3), (2,3)\}$. *Als erstes wählen wir* $(1,2) \in B$ *und entfernen das Element aus* B. *Der Rest des* S-*Polynoms bezüglich* G *ist* $f_4 := y^3$. *Das ist ungleich Null, so dass wir* B *und* G *aktualisieren.*

$$B = \{(1,3), (2,3), (1,4), (2,4), (3,4)\}$$
$$G = \{f_i \mid i = 1, 2, 3, 4\}.$$

Das nächste Element aus B *ist* $(1,3)$. *Es gilt* $[S(g_1, g_3)]_G = 0$, *also entfernen wir in diesem Schritt nur* $(1,3)$ *aus* B.

In den nächsten Durchläufen der `while`*-Schleife sind die Reste der* S-*Polynome Null, so dass die Elemente aus* B *entfernt werden und der Algorithmus abbricht. Damit ist* $G = \{f_i \mid i = 1, \ldots, 4\}$ *eine Gröbnerbasis zu* $\langle f_1, f_2, f_3 \rangle$.

Die durch den Buchberger-Algorithmus 2.2 erzeugte Gröbnerbasis ist im Allgemeinen nicht eindeutig. Unter bestimmten Bedingungen kann man diese eindeutig machen. Auch gibt es noch viele wichtige Eigenschaften von Gröbnerbasen sowie Erweiterungen beziehungsweise Spezialisierungen für unterschiedliche Arbeitsgebiete. Dies alles führen wir hier nicht an. Uns reicht zu wissen, dass Gröbnerbasen existieren und wir sie mit Hilfe des Buchberger-Algorithmus 2.2 berechnen können.

Der Buchberger-Algorithmus wurde seit seiner Entwicklung stark beschleunigt. Dabei wurde besonderes Augenmerk auf die zwei folgenden Punkte gelegt.

1. Die Auswahl eines Elements aus B in Zeile 5 des Algorithmus 2.2. Auch wenn diese Auswahl nichts an dem Resultat des Algorithmus ändert, erreichen wir mit einer für das Problem angepassten Heuristik einen großen Geschwindigkeitsvorteil. Buchberger selbst hat die Normalenauswahlstrategie in [Buc85] vorgeschlagen. Dabei wählen wir das Element $(i, j) \in B$ aus, für das der Grad des Polynoms $\deg(\mathrm{kgV}(f_i, f_j))$ minimal ist.

   Ferner benutzen Implementierungen des Buchberger-Algorithmus auch das sogenannte *Sugar*-Maß. Dabei nimmt man das Paar (i, j) mit minimalem totalen Grad $\deg(\mathrm{kgV}(f_i, f_j))$, den das Polynom $\mathrm{kgV}(f_i, f_j)$ hätte, wäre es homogen.

2. Die meiste Zeit verbraucht der Algorithmus in Zeile 7-8 für das Reduzieren von S-Polynomen eines Paares $S(f_i, f_j)$ zu Null. Also wurde nach Paaren gesucht, bei denen man leicht feststellen kann, dass das zugehörige S-Polynom zu Null reduziert. In [Buc79, GM88, Mö88] wurden einfache, implementierbare Bedingungen angegeben, wann das S-Polynom eines Paares $S(f_i, f_j)$ zu Null reduziert.

Den vollen Pseudocode geben wir hier nicht an.

Erweiterung durch lineare Algebra

In diesem Abschnitt verbessern wir die durchschnittliche Laufzeit des Buchberger-Algorithmus mit Hilfe der linearen Algebra. Insbesondere führen wir den Algorithmus F_4 von Faugère aus [Fau02] ein. Dabei folgen wir grob den Ausführungen aus [Cab10].

Der F_4-Algorithmus ähnelt dem Buchberger-Algorithmus 2.2. Auch hier werden alle Elemente der Basis durchlaufen und die S-Polynome berechnet. Der Unterschied ist, dass F_4 viele Elemente gleichzeitig auswählt und bearbeitet. Auch führt F_4 die Reduktion und die Berechnung von S-Polynomen durch Matrizenrechnung durch.

Bevor wir F_4 einführen, benötigen wir noch die folgende Definition.

Definition 2.2.19. *Sei F ein Gleichungssystem mit Gleichungen aus $\mathbb{F}[x_1, \ldots, x_n]$.*

1. *Die Menge aller Monome in F nennen wir $\mu(F)$.*
2. *Die Menge aller Variablen in F nennen wir $\nu(F)$.*
3. *Die Matrix M_F, in der als Spalten die Monome $\mu(F)$ und in den Zeilen alle Polynome $f \in F$ des Gleichungssystems abgetragen sind, heißt die Macaulay-Matrix. Die Koeffizienten der Matrix sind die Koeffizienten der einzelnen Monome.*

In unserer Arbeit wechseln wir immer zwischen der Macaulay-Matrix M_F, also der Matrix, die zu einem Gleichungssystem F gehört, und dem eigentlichen Gleichungssystem. In Abschnitt 6.1.1 führen wir direkt Matrixberechnungen auf dem eigentlichen System ohne Umrechnung aus. In unseren Algorithmen erzeugen wir mit der Funktion `echelonize` immer eine Zeilenstufenform der Matrix mit Hilfe des Gaußschen Algorithmus. Dazu gilt folgendes Lemma:

Lemma 2.2.20. *Sei $A \in \mathbb{F}^{n \times m}$ eine $n \times m$ Matrix. Die Zeilenstufenform von A berechnen wir in $\mathcal{O}(n^2 \cdot m)$ Operationen.*

Beweis: Wir wählen eine Zeile mit einem nichtverschwindenden Pivot-Element aus. In der Spalte des Pivot-Elements erzeugen wir in jeder anderen Zeile eine Null. Dafür benötigen wir m Multiplikation und m Additionen für jede der $n-1$ anderen Zeilen. Also insgesamt $(n-1) \cdot 2m$ Operationen.

Das tun wir für alle n Zeilen. Also benötigen wir insgesamt $n \cdot 2m(n-1) = \mathcal{O}(n^2 m)$ Operationen. □

Die Macaulay-Matrix verdeutlichen wir am folgendem Beispiel.

Beispiel 2.2.21. *Sei $F \subset \mathbb{F}_3[a, b, c]$ ein Gleichungssystem mit*

$$abc + ab + ac + b + 1 = 0$$
$$ac^2 + ab + 2ac + 2 = 0$$
$$a^2b + abc + 2bc + a = 0.$$

Die Menge aller Monome ist

$$\mu(F) = \{a^2b, ac^2, abc, bc, ab, ac, a, b, 1\},$$

wobei wir die 1 mitzählen. Weiter legen wir noch eine Ordnung fest, um die Monome zu sortieren, wie wir in Abschnitt 2.2.1 gesehen haben. Wir begnügen uns für dieses Beispiel mit der Ordnung, die wir in $\mu(F)$ angedeutet haben.

Wir erhalten die Macaulay-Matrix M_F für das System F

$$M_F = \begin{pmatrix} 0\,0\,1\,0\,1\,1\,0\,1\,1 \\ 0\,1\,0\,0\,1\,2\,0\,0\,2 \\ 1\,0\,1\,2\,0\,0\,1\,0\,0 \end{pmatrix}.$$

Um die Zeilenstufenform der Matrix zu berechnen, müssen wir die erste und die dritte Zeile beziehungsweise Gleichung addieren und tauschen. Die Zeilenstufenform $\overline{M_F}$ lautet

$$\overline{M_F} = \begin{pmatrix} 1\,0\,0\,2\,1\,1\,1\,1\,1 \\ 0\,1\,0\,0\,1\,2\,0\,0\,2 \\ 0\,0\,1\,0\,1\,1\,0\,1\,1 \end{pmatrix}.$$

Da es sich um ein Gleichungssystem handelt, ist es in der Praxis nicht nötig, die Zeilen dieser Matrix zu tauschen. Insgesamt erhalten wir das geänderte Gleichungssystem

$$\begin{aligned} a^2b + 2bc + ab + ac + a + b + 1 &= 0 \\ ac^2 + ab + 2ac + 2 &= 0 \\ abc + ab + ac + b + 1 &= 0. \end{aligned}$$

Wenn wir auf eine Macaulay-Matrix eine Mengenoperation wie zum Beispiel die Vereinigung durchführen, meinen wir stets eine Menge aus allen Gleichungen des Systems, also Zeilen dieser Matrix.

In Algorithmus 2.3 ist Faugères F_4-Algorithmus abgebildet. In der Funktion `f4` initialisieren wir genau wie beim Buchberger-Algorithmus 2.2 eine Liste mit allen Paaren und arbeiten diese in der `while`-Schleife ab. Die Menge L enthält von allen ausgewählten Paaren eine Hälfte des S-Polynoms. Sie wird danach in der Funktion `preproc` benutzt, um H zu initialisieren.

Die Funktion `preproc` geht alle Monome t durch, die nicht schon Leitmonom in H sind. Dabei wird nach Polynomen $g \in G$, die t reduzieren, gesucht. Das Polynom $\frac{t}{LM(g)} \cdot g$ hat stets das Leitmonom t. Nachdem wir dieses Polynom in H aufgenommen haben, versuchen wir nicht mehr, mit diesem Polynom zu reduzieren, sondern das Polynom selbst weiter zu vereinfachen. Dabei vergrößern wir die Menge $\mu(H)$, um die Bedingung der `while`-Schleife zu erfüllen.

Nach Ausführung von `preproc` benutzen wir die Macaulay-Matrix, um das System in Zeilenstufenform zu bringen. Dabei reduzieren wir viele implizit gegebenen S-Polynome gleichzeitig, da wir ein S-Polynom von f und g als

$$S(f,g) = \frac{1}{LC(f)} \cdot \frac{\mathrm{kgV}(LM(f), LM(g))}{LM(f)} \cdot f - \frac{1}{LC(g)} \cdot \frac{\mathrm{kgV}(LM(f), LM(g))}{LM(g)} \cdot g$$

schreiben können.

Algorithmus 2.3 F_4-Algorithmus zur Erstellung einer Gröbnerbasis

function `preproc`(L, G):
$H \leftarrow L$
$lm(H) \leftarrow LM(H)$
while $\mu(H) \neq lm(H)$ **do**
  choose $t \in \mu(H) \backslash lm(H)$
  $lm(H) = lm(H) \cup \{t\}$
  if $\exists g \in G : LM(g) | t$ **then**
    choose $g \in G : LM(g) | t$
    $H = H \cup \left\{ \frac{t}{LM(g)} \cdot g \right\}$
  end if
end while
return H

function `f4`(F):
$G \leftarrow F$
$B \leftarrow \{(g_1, g_2) \mid g_1, g_2 \in G, g_1 \neq g_2\}$
while $B \neq \emptyset$ **do**
  choose $B^* \subset B, B^* \neq \emptyset$
  $B = B \backslash B^*$
  $L = \left\{ \frac{\mathrm{kgV}(LM(f), LM(g))}{LM(f)} \cdot f \mid (f, g) \in B^* \right\}$
  $H =$ `preproc`(L, G)
  $M_H =$ `macaulay`(H)
  $\overline{H} =$ `echelonize`(H)
  $\overline{\overline{H}} = \{h \in \overline{H} | LM(h) \notin LM(H)\}$
  $G = G \cup \overline{\overline{H}}$
  $B = B \cup \{(h, g) | h \in \overline{\overline{H}}, g \in G, h \neq g\}$
end while
return G

Danach sammeln wir alle Gleichungen, die ein Leitmonom besitzen, das wir nicht reduzieren konnten, fügen sie in unsere Basis G ein und aktualisieren die Menge aller Paare B entsprechend.

Für den Beweis der Korrektheit des Algorithmus verweisen wir auf [Fau02].

2.2.3 Erweiterte Linearisierung

Gröbnerbasen-Algorithmen verstehen wir sehr gut und haben den Beweis aus Abschnitt 2.2.2, dass der Buchberger-Algorithmus terminiert. Auch für andere Gröbnerbasen-Algorithmen finden wir solche Beweise. Darüber hinaus lösen wir damit nichtlineare Gleichungssysteme sicher. Allerdings ist das Berechnen von S-Polynomen sehr zeitaufwändig. Im schlechtesten Fall haben Gröbnerbasen für polynomielle Gleichungssysteme über endlichen Körpern eine exponentielle Laufzeit in der Anzahl der Variablen

im System. Für Gleichungssysteme über beliebigen Körpern ist die Laufzeit für Gröbnerbasen sogar doppelt exponentiell in der Anzahl der Variablen.

In diesem Kapitel stellen wir eine Alternative zu Gröbnerbasentechniken, genannt Erweiterte Linearisierung (XL), aus [CKPS00b] vor. Der Algorithmus XL löst polynomielle Gleichungssysteme und ist in [CKPS00b] nur für quadratische Systeme ausgelegt. Allerdings wurde der Algorithmus in [Die04] für beliebige Gleichungssysteme erweitert und es wurde in [AFI+04] gezeigt, dass zur Lösung eines Gleichungssystems indirekt wieder eine Gröbnerbasis berechnet wird.

Sei dazu $F = \{f_1, \ldots, f_s\} \subset \mathbb{F}[x_1, \ldots, x_n]$ ein quadratisches Gleichungssystem über dem Körper $\mathbb{F}$. Weiter sei ein Parameter $D \in \mathbb{N}$ gegeben.

In Algorithmus 2.4 ist der XL-Algorithmus dargestellt. Die Funktion `xl` nimmt den Parameter D und das zu lösende Gleichungssystem F. Wir multiplizieren alle Monome bis zu dem Grad $D-2$ mit allen Gleichungen im System und erzeugen die Zeilenstufenform der Macaulay-Matrix. Sollten wir dabei eine univariate Gleichung erhalten, so lösen wir diese mit einem angemessenen Verfahren wie zum Beispiel dem Berlekamp-Algorithmus. Danach werden die Lösungen in das Gleichungssystem eingesetzt und das Verfahren wiederholt, bis das System gelöst ist oder keine univariaten Gleichungen mehr für den gewählten Parameter D hinzukommen.

Algorithmus 2.4 XL-Algorithmus zum Lösen eines quadratischen Gleichungssytems

function `xl`(F, D):
$G \leftarrow F$
flag $\leftarrow True$
while flag == True **do**
  flag = False
  for $k \leftarrow 1$ **to** $D-2$ **do**
    $G = G \cup \{\prod_{l=1}^{k} x_{i_l} \cdot f_j | f_j \in F, x_{i_l} \in \{x_1, \ldots x_n\}\}$
  end for
  $M_G \leftarrow$ `macaulay`(G)
  $M_G =$ `echelonize`(M_G)
  if $\exists g \in G : g \in \mathbb{F}[x_i], x_i \in \{x_1, \ldots, x_n\}$ **then**
    flag = True
    for each $g \in G : g \in \mathbb{F}[x_i], x_i \in \{x_1, \ldots, x_n\}$ **do**
      solve g
      substitute x_i in G
    end for
  end if
end while
return G

Falls die entstandenen univariaten Gleichungen keine Lösung haben, so terminiert der Algorithmus und gibt eine entsprechende Meldung aus. Wie in [Die04] beschrieben, ist der Algorithmus nicht effizient, wenn wir mehrere Lösungen pro univariater Gleichung erhalten. Wir sollten stets Gleichungen mit einer Lösung bevorzugen. Auch macht es

Sinn, den Parameter D am Anfang klein zu wählen und ihn zu erhöhen, falls keine univariaten Gleichungen gefunden werden, ohne die `while`-Schleife zu verlassen. Das tun wir bis zu einem maximalen D_{max}.

Bei der ersten Veröffentlichung des XL-Algorithmus wurde behauptet, wenn genügend Ausgangsgleichungen vorhanden sind, die ein beliebiges quadratisches Gleichungssystem überbestimmen, sei die Laufzeit des Algorithmus subexponentiell. Sowohl in [Die04] als auch in [AFI+04] wurde jedoch festgestellt, dass dies nicht der Fall ist. Falls das Gleichungssystem nur endlich viele Lösungen hat, wurde in [AFI+04] festgestellt, dass XL indirekt eine Gröbnerbasis berechnet.

Dennoch eröffnete XL eine neue Perspektive. Bei der Variante „Erweiterte Linearisierung für dünn besetzte Systeme (XSL)“ aus [CP02] generieren wir keine Gröbnerbasis. Anstatt wie bei XL alle möglichen Monome bis zum Grad $D-2$ mit den Polynomen zu multiplizieren, nehmen wir nur bestimmte. In [Moh11] wird eine gute Übersicht der dazu benutzten Heuristiken gegeben. Produkte von Monomen, die bereits im Gleichungssystem enthalten sind, scheinen demnach am Besten zu funktionieren. Dabei verlieren wir allerdings die Fähigkeit, alle polynomiellen Gleichungssysteme zu lösen.

Auch eine Analyse für die Laufzeit führen wir nicht aus, da sich die Autoren von [CP02] stark auf Heuristiken verlassen. Wie wir bei XL beschrieben haben, kann sich dieses Vorgehen als falsch herausstellen.

Wir versuchen nicht, beliebige Gleichungssysteme zu lösen. Stattdessen stellen wir in Abschnitt 6.1 einen speziellen Solver für dünn besetzte multivariate, quadratische polynomielle Gleichungssysteme mit Struktur vor.

Kapitel 3

Modellierung von Chiffren

In diesem Kapitel stellen wir in Abschnitt 3.1 zunächst die hardware-orientierte Stromchiffre Trivium und die wichtigsten bis dato bekannten Angriffe in Abschnitt 3.1.1 vor. In Abschnitt 3.1.2 modellieren wir Trivium als dünn besetztes quadratisches Gleichungssystem und führen ähnliche Variablen ein, um die Anzahl der benutzten Variablen klein zu halten. Den Effekt solcher ähnlichen Variablen auf das Gleichungssystem beschreiben wir danach in Abschnitt 3.1.3.

Darüber hinaus stellen wir in Abschnitt 3.2 Katan32 vor. Es ist eine Blockchiffre aus der hardware-orientierten Katan-Familie mit einer Blocklänge von 32 Bit. Auch hier beschreiben wir in Abschnitt 3.2.1 die wichtigsten bisherigen Angriffe. Danach modellieren wir Katan32 als Gleichungssystem über $\mathbb{F}_2$ und vergleichen es in Abschnitt 3.2.2 mit dem Gleichungssystem von Trivium.

3.1 Trivium

Trivium ist eine gut studierte hardware-orientierte synchrone Stromchiffre, die in [CP08] vorgestellt wurde. Stromchiffren lernten wir in Abschnitt 2.1.1 kennen. Die Stromchiffre Trivium benutzt einen 80 Bit langen Schlüssel und einen 80 Bit langen öffentlichen Startwert, um einen Schlüsselstrom zu generieren. Vom IV können wir auch weniger Bits benutzen, indem wir den Rest auf Null setzen. Laut den Sicherheitsspezifikationen von Trivium ist ein bis zu 2^{64} Bit langer Schlüsselstrom ohne Sicherheitseinbußen möglich. Die Chiffre besteht aus der Initialisierungsphase, in der R Runden lang der interne Zustand der Chiffre aktualisiert wird, und der Phase, in der der Schlüsselstrom produziert wird. Um Trivium zu beschreiben, benutzen wir eine sehr kompakte Form mit drei rekursiven quadratischen Gleichungen, die die Zustandsänderung pro Runde beschreiben, und einer linearen Gleichung, mit der wir die Ausgabe berechnen. Trivium benutzt nur Multiplikation und Addition über $\mathbb{F}_2$, da diese Operationen sehr effizient durch die logischen Operationen XOR und AND auf Hardware zu implementieren sind.

Wir betrachten die drei Schieberegister $A := (A_i, \ldots, A_{i-92})$, $B := (B_i, \ldots, B_{i-83})$ und $C := (C_i, \ldots, C_{i-110})$. Diese drei Schieberegister bilden den internen Zustand von Trivium. Diesen Zustand initialisieren wir wie folgt.

$$A = (k_0, \ldots, k_{79}, 0, \ldots, 0)$$
$$B = (v_0, \ldots, v_{79}, 0, \ldots, 0)$$
$$C = (0, \ldots, 0, 1, 1, 1)$$

Hier bezeichnet $(k_0, \ldots, k_{79}) \in \mathbb{F}_2^{80}$ den geheimen *Schlüssel* und $(v_0, \ldots, v_{79}) \in \mathbb{F}_2^{80}$ den öffentlichen IV von Trivium. Wenn wir es schaffen, den Schlüssel zu berechnen, so ist die Chiffre gebrochen. Im Folgenden benutzen wir das „Gewählte IV"-Szenario. Das heißt, wir berechnen die Ausgabe von vielen Instanzen von Trivium mit dem gleichen Schlüssel und unterschiedlichem IV.

Den internen Zustand Triviums aktualisieren wir in jeder Runde rekursiv.

$$B_i := A_{i-65} + A_{i-92} + A_{i-90}A_{i-91} + B_{i-77}$$
$$C_i := B_{i-68} + B_{i-83} + B_{i-81}B_{i-82} + C_{i-86}$$
$$A_i := C_{i-65} + C_{i-110} + C_{i-108}C_{i-109} + A_{i-68}$$

Nach der Initialisierungsphase von R Runden wird zusätzlich pro Runde eine Ausgabe durch

$$z_i := C_{i-65} + C_{i-110} + A_{i-65} + A_{i-92} + B_{i-68} + B_{i-83}$$

berechnet.

Das nächste Bit jedes der drei Schieberegister wird durch Einträge eines anderen Schieberegisters berechnet. Dabei werden jeweils fünf Einträge verwendet. Die drei quadratischen Terme sind die einzigen nichtlinearen Komponenten von Trivium. Wir betrachten im Folgenden auf R Runden reduzierte Varianten von Trivium. Die volle Chiffre benutzt $R = 1152$ Runden, bevor Ausgabebits produziert werden. In [CP08] stellen die Autoren zwei Varianten von Trivium vor, genannt Bivium-A und Bivium-B. In denen werden zwei anstatt drei Schieberegister verwendet. Diese betrachten wir nicht, da beide Varianten schon mehrmals gebrochen wurden.

Abbildung 3.1 zeigt das Schaltbild einer Runde Triviums.

3.1.1 Angriffe auf Trivium

Bis jetzt ist Trivium ungebrochen. Dabei war die Chiffre gerade wegen seiner einfachen quadratischen Struktur oft Ziel von Angriffen. Wir wollen hier einige davon vorstellen.

In [DS09, FV13] stellten die Autoren Cube-Angriffe vor beziehungsweise verbesserten sie. Cube-Angriffe betrachten wir ausführlich in Kapitel 4. In beiden Arbeiten wird versucht, den Schlüssel zu berechnen. Um den Schlüssel für eine auf 799 Runden reduzierte Variante von Trivium zu berechnen, werden dort 2^{62} Operationen und 2^{40} Triviuminstanzen benötigt. Dabei haben die Autoren 12 lineare Cubes nach 799 Initialisierungsrunden gefunden und quadratische Cubes aus Runde 784 benutzt, um mehr Informationen über den Schlüssel zu erhalten. Mit dieser Technik ist es möglich 18 Schlüsselbits zu berechnen. Die übrigen Schlüsselbits werden durch einen Brute-Force-Angriff gefunden.

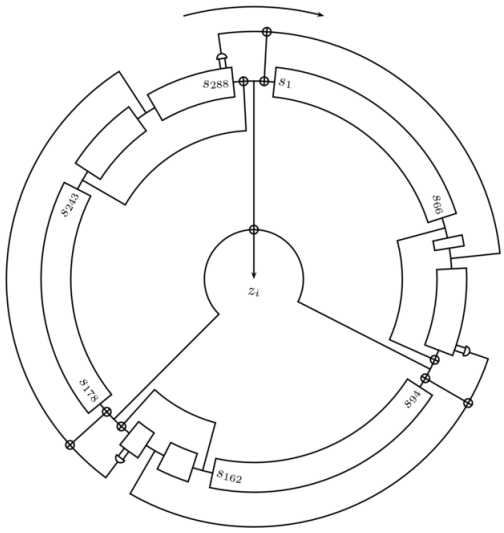


Abbildung 3.1: Schaltbild einer Runde Triviums

Distinguisher, die den Schlüssel nicht berechnen, sondern, wie in Abschnitt 2.1.2 beschrieben, die Chiffre vom Zufall unterscheiden, stellten die Autoren unter anderem in [KMNP11, ADMS09, Sta10] vor. Das Sicherheitsspiel zu Distinguishern ist in Abbildung 2.3 dargestellt. Die erfolgreichsten Distinguisher für Trivium basieren auf sogenannten Cube-Testern. Das sind Varianten des Cube-Angriffs aus Kapitel 4. Dabei suchen wir nicht nach algebraischen, sondern nach statistischen Eigenschaften von Polynomen.

In [KMNP11] wird nach der Neutralität von IV-Bits gesucht. Neutralität eines Bits bedeutet, dass das Bit mit einer Wahrscheinlichkeit, die von $\frac{1}{2}$ signifikant verschieden ist, keinen Einfluss auf die Ausgabe der Chiffre hat. Das erlaubt uns, zwei Instanzen zu generieren, die ein Ausgabebit mit hoher Wahrscheinlichkeit gemeinsam haben. Also können wir entscheiden, ob die Ausgabe zufällig oder von Trivium generiert worden ist. Die Autoren von [KMNP11] fanden die beiden öffentlichen Bits 72 und 78, die das erste Ausgabebit einer auf $R = 961$ Runden reduzierten Variante Triviums mit Wahrscheinlichkeit 0 beziehungsweise 1 verändern, wenn wir einen Cube vom Grad 25 betrachten. Der Angriff arbeitet auf einem reduzierten Schlüsselraum von 2^{26} Schlüsseln und benötigt für die Cube-Summation 2^{25} Trivium-Berechnungen. In dieser Arbeit berechnen wir stets den vollen Schlüssel, so dass wir uns nicht weiter mit solchen statistischen Angriffen auseinandersetzen.

Algebraische Angriffe auf Trivium sind in [SFP08, T$^+$13, SR12, Rad06] beschrieben. Die dort vorgestellten Techniken basieren auf zwei algebraischen Repräsentationen der Chiffre, die in [Rad06] vorgestellt wurden.

Im ersten Modell setzen wir den internen Zustand der Chiffre vor der Ausgabe auf symbolische Werte. Dann wird die Ausgabe

$$z_i := C_{i-65} + C_{i-110} + A_{i-65} + A_{i-92} + B_{i-68} + B_{i-83}$$

symbolisch gespeichert und zu der Ausgabe der numerischen Triviuminstanz addiert. In jeder Runde wird dabei die Aktualisierungsfunktion auch symbolisch durchlaufen. Dabei

Autor	Methode	R	Zeit	Daten
[FV13]	Cube-KR	799	2^{62}	2^{40}
[KMNP11]	Cube-D/WK (2^{26})	806/961	$2^{45}, 2^{25}$	$2^{45}, 2^{25}$
[Rad06]	Algebraisch	–	–	–
[KHK06]	Linearisierung	288	2^{72}	2^{62}

Tabelle 3.1: Angriffe auf Trivium mit R Runden, Zeit- und Datenkomplexität

steigt der Grad der Gleichungen im resultierenden System. So erhalten wir ein dichtes Gleichungssystem von hohem Grad. Dieses Gleichungssystem ist wegen des hohen Grades schwer bis gar nicht lösbar.

In der zweiten Modellierung setzen wir ebenso den internen Zustand der Chiffre vor der Ausgabe auf symbolische Werte. Darüber hinaus führen wir pro Runde drei Zwischenvariablen a_i,b_i und c_i für die Eingabe des jeweiligen Schieberegisters A_i, B_i und C_i ein. Damit beschränken wir den Grad des resultierenden Gleichungssystems auf 2 und erhalten ein dünn besetztes System mit $288+3\cdot n_o$ Variablen in $4\cdot n_o$ Gleichungen. Dabei ist n_o die Anzahl der Runden, in der Ausgabe produziert wird. Mit dieser Technik können wir die auf zwei Schieberegister reduzierten Varianten Bivium-A und Bivium-B brechen. Trivium oder auch rundenreduzierte Varianten von Trivium brechen wir hingegen nicht. Dies liegt daran, dass wir nur eine Triviuminstanz betrachten können und damit viele Ausgabebits benötigen, um ein bestimmtes oder überbestimmtes Gleichungssystem zu erhalten.

Andere Angriffe sind nicht so erfolgreich. So bricht ein linearer Angriff in [KHK06] eine auf 288 Runden reduzierte Variante von Trivium mit einer Wahrscheinlichkeit von 2^{-72}. Dabei stellen die Autoren die Ausgabe von Trivium rekursiv dar. Das heißt, wir setzen in die Ausgabefunktion die einzelnen Variablen für A_i, B_i und C_i wiederholt ein, bis wir Schlüsselvariablen einsetzen. Danach lassen die Autoren die quadratischen Monome weg. Die Wahrscheinlichkeit, dass ein quadratisches Monom Null ist, beträgt 3/4. Durch diese Technik erhalten die Autoren eine Wahrscheinlichkeit von 2^{-72}, dass der Angriff Erfolg hat.

In Tabelle 3.1 sind die wichtigsten Angriffe auf Trivium noch einmal aufgelistet. Dabei heißt KR, dass wir den vollen oder Teile des Schlüssels finden, D bezeichnet einen Distinguisher und Schwache Schlüssel - Weak Key (WK) bedeutet, dass der Schlüsselraum auf den in Klammern angegebenen Wert reduziert ist.

3.1.2 Ähnliche Variablen

Im vorherigen Abschnitt haben wir bereits bekannte algebraische Angriffe auf Trivium beschrieben. Diese scheiterten vor allem an zwei Schwächen des zu Grunde liegenden Modells. Einerseits ist der Informationsdurchsatz einer Triviuminstanz zu gering, um

das resultierende nichtlineare Gleichungssystem zu lösen. Informationsdurchsatz heißt hier, dass wir pro Ausgabegleichung, die das System definiert, drei quadratische Gleichungen bekommen, die jeweils eine Zwischenvariable einführen. Damit können wir das System nicht linearisieren. Andererseits erstellten die verschiedenen Entwickler algebraischer Angriffe die Modelle weitgehend unabhängig von den zur Verfügung stehenden Algorithmen zum Lösen von polynomiellen Gleichungssystemen.

Um diese Schwächen des früheren algebraischen Modells zu umgehen, stellen wir in Kapitel 6 einen Gleichungslöser für die aus unseren Modellen resultierenden Gleichungssysteme zusammen und entwickeln hier ein Modell, das mehrere Triviuminstanzen mit gleichem Schlüssel und unterschiedlichem initialem Wert verarbeitet.

Sei $I \subset V$ eine Untermenge der Menge aller Variablen des IV V. Wir betrachten Triviuminstanzen, die wir mit demselben unbekannten Schlüssel und verschiedenen, bekannten Startwerten definieren. Vor der Initialisierungsphase initialisieren wir alle Triviuminstanzen mit symbolischen Werten $k_0, \ldots, k_{79}$ für den Schlüssel. Die bekannten IVs setzen wir sofort in die internen Zustände ein. Jetzt aktualisieren wir die Triviuminstanzen, bis wir bei Rundenanzahl R angelangt sind. Währenddessen führen wir in jeder Runde drei Variablen $a_{i,t}, b_{i,t}$ und $c_{i,t}$ für die Rückmeldung der drei Register $A_{i,t}, B_{i,t}$ und $C_{i,t}$ ein. Hierbei kennzeichnet t die derzeit aktuelle Triviuminstanz. Durch dieses Vorgehen erhalten wir ein dünn besetztes quadratisches Gleichungssystem mit vielen Variablen. Wie viele Variablen wir für diese algebraische Repräsentation benötigen, zeigt das nächste Lemma.

Lemma 3.1.1. *Sei $R > 238$ und $n_o \leqslant 66$. Mit der oben dargestellten Modellierung brauchen wir genau $3 \cdot R - 522$ Zwischenvariablen, um eine Triviuminstanz zu beschreiben.*

Beweis: Wir zählen die benötigten Zwischenvariablen, um eine Triviuminstanz zu generieren. Dazu betrachten wir die Aktualisierungsfunktion

$$\begin{aligned} B_i &= A_{i-65} + A_{i-92} + A_{i-90}A_{i-91} + B_{i-77} \\ C_i &= B_{i-68} + B_{i-83} + B_{i-81}B_{i-82} + C_{i-86} \\ A_i &= C_{i-65} + C_{i-110} + C_{i-108}C_{i-109} + A_{i-68} \end{aligned}$$

von Trivium. Wir führen immer dann eine Zwischenvariable ein, wenn wir ansonsten eine Gleichung von Grad 3 oder höher erhalten würden. In diesem Beweis behandeln wir die Register A, B und C nacheinander.

Am Anfang der Initialisierung enthält das Register A die einzigen symbolischen Variablen. In der 13-ten Runde wird der erste quadratische Ausdruck $B_{13} = A_{78} \cdot A_{79} + \cdots = k_{79} \cdot k_{78} + \cdots$ berechnet. Auf Grund der Aktualisierungsfunktion dauert es 82 weitere Runden, bis obiger Ausdruck mit einem linearen Ausdruck in C_{95} multipliziert wird. Wir führen hier die erste Zwischenvariable c_{95} ein. Danach führen wir in jeder Runde im Register C eine neue Variable ein.

Die anderen Register behandeln wir analog. Als erstes betrachten wir das Register A. Den quadratischen Ausdruck B_{13} speichern wir im Register C in Runde $13 + 69 = 82$ durch $C_{80} = B_{12} + \ldots$. In Runde 191 wird dieser Ausdruck in $A_{189} = C_{81} \cdot C_{80} + \cdots$ mit

einem linearen Eintrag multipliziert. Ab dieser Runde führen wir auch im Register A in jeder Runde eine Zwischenvariable ein.

Im Register B passiert Folgendes: Wie bereits erwähnt, wird unser erster quadratischer Ausdruck B_{13} in Runde 82 im Register C gespeichert. Nach 66 weiteren Runden wird der Ausdruck in $A_{145} = C_{80} + \ldots$ gespeichert. Es dauert noch einmal 91 Runden, bis wir eine neue Zwischenvariable in $B_{236} = A_{146} \cdot A_{145} + \ldots$ benötigen. Von da an benötigen wir in jeder Runde in jedem Register eine neue Zwischenvariable.

Wenn wir keine weiteren Reduktionstechniken einsetzen, benötigen wir also

$$\begin{aligned}\nu &= (R-94) + (R-190) + (R-238)\\ &= 3R - 522\end{aligned}$$

Variablen für eine Triviuminstanz.

Wir fügen noch hinzu, dass wir für die ersten 66 Ausgabebits keine neuen Zwischenvariablen benötigen. Nach der Initialisierungsphase wollen wir nur noch Ausgabebits durch die Ausgabefunktion

$$z_i = C_{i-65} + C_{i-110} + A_{i-65} + A_{i-92} + B_{i-68} + B_{i-83}$$

von Trivium berechnen.

Diese Funktion ist linear und benutzt 66 alte Einträge des internen Zustands. Also müssen wir den Zustand nicht aktualisieren, falls wir $n_o \leqslant 66$ Ausgaberunden benötigen. Damit brauchen wir auch keine weiteren Zwischenvariablen. □

Korollar 3.1.2. *Für* $0 \leqslant n_o \leqslant 66$ *Ausgabebits benötigen wir* $R+n_o-66$ *Aktualisierungen von Trivium.*

Für den Rest dieses Abschnitts setzen wir $R > 238$ voraus. Wir wollen die Anzahl der Zwischenvariablen reduzieren. Dazu führen wir ähnliche Variablen ein. Dafür gehen wir zunächst von allgemeinen Gleichungssystemen aus. Wir kommen später jedoch wieder auf Trivium zurück.

Definition 3.1.3. *Sei* $P = \mathbb{F}_2[k_0, \ldots, k_{79}, y_0, y_1, \ldots] =: \mathbb{F}_2[K, Y]$ *der Boolesche Ring über den Schlüsselvariablen* K *und allen Zwischenvariablen* Y*.*

Wir nennen zwei Zwischenvariablen y_i *und* y_j *genau dann ähnlich, wenn* $y_i + y_j = p(K, Y\backslash\{y_i, y_j\})$ *wobei* $p(K, Y\backslash\{y_i, y_j\})$ *ein Polynom vom totalen Grad* $\deg(p) \leqslant 1$ *ist.*

Diese Definition wenden wir wie folgt an. Wenn wir eine neue Zwischenvariable einführen wollen, stellen wir zunächst fest, ob eine entsprechende ähnliche Variable existiert. Falls ja, führen wir keine neue Variable ein, sondern rechnen einfach mit $y_j + p(K, Y\backslash\{y_j\})$ weiter.

Wenn wir ein Gleichungssystem F über P behandeln, erweitern wir die Idee der ähnlichen Variablen wie folgt.

Definition 3.1.4. *Sei* $P = \mathbb{F}_2[k_0, \ldots, k_{79}, y_0, y_1, \ldots] =: \mathbb{F}_2[K, Y]$ *der Boolesche Ring über den Schlüsselvariablen* K *und allen Zwischenvariablen* Y *und* $F \subset P$ *ein Gleichungssystem.*

Wir nennen die Zwischenvariable y_i genau dann ähnlich *zum System F, wenn eine nichtleere Menge $Y_k \subset Y, Y_k \neq \emptyset$ für $1 \leqslant k < |Y|$ von bis zum Schritt k noch nicht eingeführten Zwischenvariablen existiert, so dass $y_i + \sum_{y \in Y_k} y = p(K, (Y \backslash Y_k) \backslash \{y_i\})$ mit einem Polynom $p(K, (Y \backslash Y_k) \backslash \{y_i\})$ vom Grad $\deg(p) \leqslant 1$.*

Das folgende Beispiel zeigt, wie wir mit ähnlichen Variablen arbeiten.

Beispiel 3.1.5. *Wir betrachten die Polynome*

$$f_0 = y_0 + k_{78}k_{79} + k_{53}$$
$$f_1 = y_1 + k_{77}k_{78} + k_{79} + k_{52}$$

in P. Diese Polynome definieren die Zwischenvariablen y_0 und y_1 und bilden das Gleichungssystem F.

Nehmen wir an, wir wollten die neue Zwischenvariable y_2 durch

$$f_2 = y_2 + k_{79}k_{78} + k_{78}k_{77} + k_5 + k_{61}$$

einführen. Es gilt $y_2 = y_1 + y_0 + k_{53} + k_{79} + k_{52} + k_5 + k_{61}$. Also brauchen wir y_2 nicht einzuführen und können mit y_0 und y_1 in f_2 weiter rechnen. Dieses Vorgehen spart uns nicht nur eine Variable, sondern es ermöglicht uns, mit einer zusätzlichen linearen anstatt einer quadratischen Gleichung weiter zu rechnen.

Also können wir eine Menge Zwischenvariablen und quadratische Gleichungen durch ähnliche Variablen sparen. Dieses Vorgehen ist insbesondere dann interessant, wenn wir mit mehreren Triviuminstanzen arbeiten.

Abbildung 3.2 zeigt die Anzahl der benötigten Variablen für $T = 32$ Instanzen mit $n_o = 66$ Ausgabebits mit und ohne Benutzung von ähnlichen Variablen. Wir verringern die Anzahl der benötigten Variablen um ein Vielfaches. Ohne ähnliche Variablen benötigen wir $1278 \cdot 32 = 40896$ Variablen, um 32 Instanzen mit 600 Runden zu generieren. Damit produzieren wir $66 \cdot 32 = 2112$ Ausgabebits. Das Modell in [Rad06] benötigt zum Vergleich $288 + 3 \cdot 2112 = 6624$ Variablen, um diese Menge an Ausgabebits zu produzieren. Mit ähnlichen Variablen benötigen wir dafür nur 6557 Variablen und unser Vorteil wird größer, je mehr Instanzen wir generieren.

Darüber hinaus stellen wir eine Sättigung der Anzahl an Variablen und Monomen in unserem Modell fest. Das heißt, ab einem bestimmten Punkt führen wir keine neuen Variablen und auch keine neuen Monome mehr ein, um eine Triviuminstanz zu generieren. Im Folgenden behandeln wir diese Beobachtung noch ausführlicher.

Gleichungssysteme dieser Größe lösen moderne Gröbnerbasis-Algorithmen wie PolyBoRi aus [BD09] nicht. Die Anzahl der Variablen ist zu hoch. Wie es dennoch möglich ist solche Gleichungssysteme zu lösen, lernen wir in Kapitel 6.

Bevor wir zu den Sättigungseffekten in unserem Modell kommen, widmen wir uns dem Algorithmus 3.1. Dieser generiert das Gleichungssystem für unser oben beschriebenes Modell. Dabei bezieht sich der Algorithmus auf Trivium, kann jedoch leicht auf

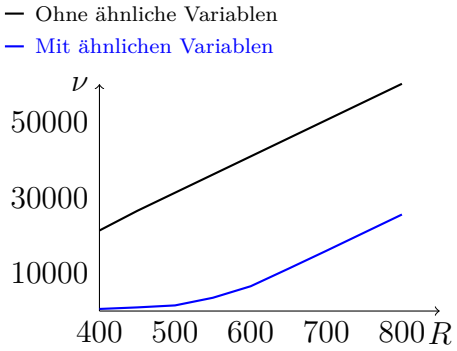


Abbildung 3.2: Gesamtzahl der Variablen für $T = 32$ Triviuminstanzen mit und ohne ähnliche Variablen

andere Kryptosysteme übertragen werden. Algorithmus 3.1 kann direkt angewandt werden, wenn die Chiffre eine quadratische Aktualisierungsfunktion besitzt. Andernfalls müssen wir ihn so abändern, dass alle Gleichungen Grad zwei haben.

Algorithmus 3.1 liefert uns ein Gleichungssystem, das $T \in \mathbb{N}$ Triviuminstanzen mit je R Runden und $n_o \leqslant 66$ Ausgabebits beschreibt. Dabei benutzen alle Instanzen denselben Schlüssel $k_0, \ldots, k_{79}$. Die Variablen der IVs $v_{t,0}, \ldots, v_{t,79}$, $t \in \{0, \ldots, T-1\}$ initialisieren wir hingegen mit verschieden Werten für verschiedene Triviuminstanzen.

In der Funktion `representation` initialisieren wir für die T Instanzen die drei Schieberegister und generieren danach die Ausgabe der Funktion `updateState`. Nachdem wir das System F aufgestellt haben, fassen wir F als Macaulay-Matrix auf. Mit der Funktion `echelonize` berechnen wir die Zeilenstufenform dieser Koeffizientenmatrix.

Das Gleichungssystem zur Zeilenstufenform der Macaulay-Matrix besteht aus zwei Mengen von Gleichungen. Der Menge der quadratischen Gleichungen Q und der Menge der linearen Gleichungen L. Mit Hilfe der linearen Gleichungen L können wir Q vereinfachen. Wir ersetzen die Leitterme $\mathrm{LT}(L)$ von L in Q durch die zugehörigen Gleichungen in L. Das ermöglicht es uns direkt, ähnliche Variablen zu benutzen, wenn wir unsere Ordnung entsprechend wählen.

Wir setzen die Schlüsselvariablen und die Ausgabe in Verbindung, um den Schlüssel berechnen zu können. Dazu nutzen wir wie auch bei Gröbnerbasen die *degrevlex*-Ordnung. Bei Gröbnerbasen ging es dabei ausschließlich um Geschwindigkeit. In unserem Modell wollen wir keine Gleichungen mit Grad größer als zwei erzeugen. Aus diesem Grund wählen wir eine gradierte Monomordnung, da wir ansonsten auch lineare Leitterme in Q erhalten. Wenn wir durch die dazugehörenden Gleichungen die Leitvariable in Q eliminieren, würde sich der Grad unserer Gleichungen erhöhen.

Die Variablen, die wir berechnen wollen, ordnen wir wie bei einer Gröbnerbasenberechnung klein ein. Die Informationen, die das Gleichungssystem definieren, finden wir in der Ausgabe. Also ordnen wir die Ausgabevariablen beziehungsweise die Zwischenvariablen aus hohen Runden groß ein. Da die Aktualisierungsfunktionen der einzelnen Register gleich aufgebaut sind, ordnen wir Zwischenvariablen aus der gleichen Runde ähnlich ein. Insgesamt erhalten wir die *degrevlex*-Ordnung mit

$$k_0 < \ldots < k_{79} < a_{0,0} < b_{0,0} < c_{0,0} < a_{1,0} < b_{1,0} < c_{1,0} < \ldots < a_{t,R-1} < b_{t,R-1} < c_{t,R-1}\,.$$

Wir benutzen diese Ordnung, weil wir eine Stromchiffre betrachten. Am Anfang, also „klein“, initialisieren wir die Chiffre mit dem geheimen Schlüssel. Danach initialisieren wir die Zwischenvariablen, die alle auf den am Anfang initialisierten Schlüsselvariablen basieren, aufsteigend nach Runde. Zum Schluss, also ganz „oben“, initialisieren wir die Ausgabe. Da wir den Schlüssel berechnen wollen und das System in der Ausgabe definieren, müssen wir die Zwischenvariablen berechnen, um eine Verbindung zu generieren.

Mit der gewählten Ordnung ist wiederholtes Anwenden von `echelonize` und Einsetzen der zu $\mathrm{LT}(L)$ gehörenden Gleichung gleichbedeutend zum Gebrauch von ähnlichen Variablen. Durch die Zeilenstufenform der Macaulay-Matrix erhalten wir lineare Zusammenhänge zwischen Variablen, die wir danach einsetzen. Wenn dadurch neue lineare Zusammenhänge entstehen, wiederholen wir das Einsetzen. Mit der gewählten Monomordnung ersetzen wir die Zwischenvariablen höherer Runden durch solche aus kleineren Runden. Wir verwenden alle linearen Gleichungen, um mehr ähnliche Variablen zu generieren. In der Funktion `insertLinVars` setzen wir diese linearen Gleichungen in den Rest des Gleichungssystems F ein.

Wir widmen uns jetzt der Laufzeit von Algorithmus 3.1. Dazu sehen wir uns zunächst die verschiedenen Teile des Algorithmus im Einzelnen an.

Als erstes betrachten wir die Generierung einer Triviuminstanz. Der interne Zustand wird $R + n_o$ mal aktualisiert. In jeder Aktualisierung führen wir drei Multiplikationen von linearen Polynomen durch. Also haben wir eine Zeitkomplexität von $\mathcal{O}((R+n_o)\cdot N_l^2)$, wobei N_l die maximale Anzahl von Termen in einem linearen Polynom ist.

Die zu F gehörende Matrix ist eine $m \times \mu-$Matrix, wobei μ die Anzahl der Monome und m die Anzahl der Gleichungen im Gleichungssystem F ist. In der Aktualisierungsfunktion fügen wir höchstens 3 neue Gleichungen pro Runde hinzu. Nach 111 Runden bestehen diese Gleichungen aus jeweils 4 Monomen. Da $R > 238$ ist, haben wir demnach mehr Monome als Gleichungen. Also approximieren wir die Laufzeit des zweiten Teils mit $\mathcal{O}(\mu^3)$. Je mehr Instanzen wir generieren, desto größer wird hierbei μ. Wie wir gleich sehen, wird sich die Anzahl der Monome und Variablen sättigen.

Die Funktion `insertLinVars` hat eine Laufzeit von $\mathcal{O}(\mu_p^2 \cdot m)$. Wir setzen in höchstens jede der m Gleichungen die maximale Anzahl der Monome μ_p in *einer* Gleichung aus F ein.

Alles in allem erhalten wir eine Laufzeit von $\mathcal{O}(T\mu^3)$, da der Gauß-Algorithmus den meisten Aufwand benötigt und wir ihn höchstens T mal ausführen. Wie oben beschrieben, können wir μ nicht genau abschätzen. Im nächsten Abschnitt sehen wir, dass sich die Anzahl der Monome und Variablen abhängig von der Anzahl der betrachteten Instanzen T sättigt.

Die Laufzeit des Algorithmus können wir noch verbessern. Zum Beispiel können wir den Strassen-Algorithmus nutzen. Dies allein verringert die Laufzeit zu $\mathcal{O}(T\mu^{2.7})$ oder $\mathcal{O}(T\mu^2)$, wenn das System F dünn besetzt ist.

Wir gehen im Folgenden auf die Experimente ein, die wir mit Gröbnerbasen aus-

Algorithmus 3.1 Berechnung des Gleichungssystems F für Trivium mit Hilfe ähnlicher Variablen

function updateState(k, out):
if out = **true then**
F.insert($C[65] + C[110] + A[65] + A[92] + B[68] + B[83]$)
return
end if
F.insert($a_{t,k} + A[0]$); F.insert($b_{t,k} + B[0]$); F.insert($c_{t,k} + C[0]$)
$A[0] \leftarrow a_{t,k}$; $B[0] \leftarrow b_{t,k}$; $C[0] \leftarrow c_{t,k}$
for $j \leftarrow 0$ **to** 91 **do**
A[j+1] = A[j]
end for
$A[0] = C[65] + C[110] + C[108]C[109] + A[68]$
for $j \leftarrow 0$ **to** 82 **do**
B[j+1] = B[j]
end for
$B[0] = A[65] + A[92] + A[90]A[91] + B[77]$
for $j \leftarrow 0$ **to** 109 **do**
C[j+1] = C[j]
end for
$C[0] = B[68] + B[83] + B[81]B[82] + C[86]$
return

function representation(R, n_o, t):
global $F \leftarrow \emptyset$
for $n \leftarrow 1$ **to** t **do**
global $A \leftarrow (k_0, \ldots, k_{79}, 0, \ldots, 0)$
global $B \leftarrow (v_{t,0}, \ldots, v_{t,79}, 0, \ldots, 0)$
global $C \leftarrow (0, \ldots, 0, 1, 1, 1)$
for $k \leftarrow 0$ **to** $R - 1$ **do**
updateState($k, out =$ **false**)
end for
for $k \leftarrow 1$ **to** n_o **do**
updateState($k, out =$ **true**)
end for
echelonize(F)
insertLinVars(F)
end for
return F

geführt haben. Dabei wird auch klar, welchen Nutzen ähnliche Variablen und die gewählte Modellierung haben.

Trivium und die Technik ähnlicher Variablen wurde in dem Computer-Algebra-System SAGE aus [S+14] mit etwa 1700 Zeilen implementiert.

Um zu überprüfen, wie gut konditioniert unser System ist, haben wir PolyBoRi aus [BD09] benutzt, um eine Gröbnerbasis vom Gesamtsystem mit ähnlichen Variablen zu berechnen. PolyBoRi ist speziell für boolesche Polynomringe entwickelt und benutzt eine Variante des F_4-Algorithmus, um eine Gröbnerbasis zu berechnen. Dabei wollen wir die Rundenanzahl von Trivium maximieren, für die wir eine Gröbnerbasis berechnen können. In Abschnitt 6.2 führen wir mit einem anderen Solver eine vollständige Parameteranalyse durch.

Unsere Abbruchbedingungen waren 16Gb Speicheranforderung und 24 Stunden Zeit auf einem Server-Knoten mit einem AMD-Opteron-6276@2.3GHz. Auch ist PolyBoRi sehr oft reproduzierbar abgestürzt, so dass wir das Experiment als abgebrochen werten mussten.

Ohne ähnliche Variablen löst PolyBoRi das Gleichungssystem, das nur durch $R = 150$ Runden Trivium generiert wird. Darüber hinaus lösen wir kein Gleichungssystem.

Wir betrachten verschiedene Ordnungen für unterschiedliche Rundenanzahlen R von Trivium. Mit einer nicht gradierten Ordnung lösen wir das Gleichungssystem nicht, das durch $R = 200$ Runden Trivium erzeugt wird.

Wir verwenden stets die *degrevlex*-Ordnung. Wenn wir diese durch die *deglex*-Ordnung austauschen, lösen wir das Gleichungssystem für $R = 250$ Runden Trivium, jedoch kein Gleichungssystem mit einer höheren Rundenanzahl.

Wenn wir die Variablen umsortieren, hat das großen Einfluss auf den Speicherverbrauch und die Laufzeit von PolyBoRi. Bei einer anderen Sortierung der Variablen haben wir nie vergleichbare Ergebnisse produziert beziehungsweise konnten nicht lösen. Getestet haben wir das mit dem Gleichungssystem, das $R = 300$ Runden Trivium beschreibt.

Insgesamt war es mit PolyBoRi und ähnlichen Variablen möglich, eine Gröbnerbasis für ein Gleichungssystem zu berechnen, das $R = 450$ Runden Trivium repräsentiert. Das Ausgangssystem hatte etwa 1300 Variablen in 1500 Gleichungen mit 3900 Monomen. Die Gröbnerbasis wurde in 13 Stunden berechnet. Dabei benötigte PolyBoRi 11Gb Speicher.

Der Speicherverbrauch und die verwendete Datenstruktur waren hier das Problem. PolyBoRi verwendet binäre Bäume, um dicht besetzte Polynome effizient darzustellen. Leider stellt PolyBoRi immer den vollen Baum dar, wenn es ein Polynom speichert. In Kapitel 6 lernen wir einen Solver kennen, der keine Speicherprobleme hat, weil er keine S-Polynome erzeugt und eine angepasste Datenstruktur für dünn besetzte Systeme benutzt. Mit Hilfe dieses Solvers haben wir die Abbildungen in diesem Kapitel dargestellt.

3.1.3 Sättigung von Monomen und Variablen im Modell

Wie wir bereits im letzten Abschnitt gesehen haben, vermögen ähnliche Variablen die Anzahl von Zwischenvariablen signifikant zu senken. In der Praxis sehen wir darüber hinaus den in Abbildung 3.3 und Abbildung 3.4 gezeigten Sättigungseffekt. Das heißt, wir generieren mit steigendem Hamming-Gewicht (HW) des IVs weniger Variablen und Monome für das Modell von weiteren Triviuminstanzen. Dabei benötigen wir mit steigender Rundenanzahl R mehr Triviuminstanzen, damit sich die Sättigung einstellt.

In unseren Experimenten verwenden wir i IV-Bits und generieren 2^i Triviuminstanzen, die zu allen möglichen Vektoren im $\mathbb{F}_2^i$ gehören. Die restlichen $(80-i)$ IV-Bits setzen

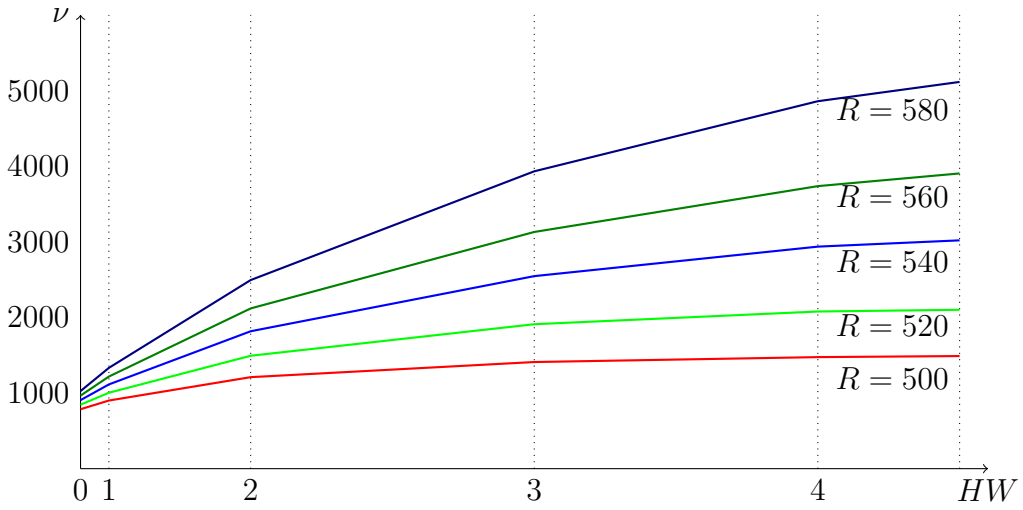


Abbildung 3.3: Sättigung der Variablen für Triviuminstanzen mit hohem Hamming-Gewicht des Startwerts

wir auf Null. Wenn wir jetzt eine Triviuminstanz mit hohem HW erstellen, benötigen wir weniger Variablen und Monome als für eine Triviuminstanz mit geringem HW. Dies gilt nur dann, wenn wir schon die Instanzen mit geringem HW generiert haben. Also versuchen wir die Anzahl i der benutzten IV-Bits klein zu halten, damit wir die Variablen und Monome der Instanzen mit geringem HW wiederverwenden. Mit steigender Rundenanzahl von Trivium werden die einzelnen Instanzen jedoch unabhängiger und wir benötigen mehr Instanzen, um eine Sättigung zu erreichen. In den Abbildungen 3.3 und 3.4 generierten wir 32 Instanzen und zählten die Anzahl der Variablen und quadratischen Monome. Dabei haben wir, um besser mit dem Modell von Katan vergleichen zu können, die Monomordnung so sortiert, dass die Variablen und Monome der ersten Instanzen mit einem IV von geringem HW genutzt werden, um die restlichen Instanzen zu beschreiben.

Abbildung 3.3 zeigt das HW des für die Triviuminstanz benutzten IV gegen die Anzahl der Variablen ν. Dabei verwenden wir auf $R = 500, 520, 540, 560, 580$ Runden reduzierte Varianten von Trivium. Instanzen mit demselben HW benötigen dieselbe Anzahl von Variablen. Diese Anzahl verringert sich mit höherem Hamming-Gewicht.

Die Sättigung der Monome benötigt ein geringeres HW des IVs, wie wir in Abbildung 3.4 sehen können. Hier fangen wir bei Runde $R = 510$ an, die Graphen zu zeichnen, und sind in 20er Schritten bis Runde $R = 590$ hochgegangen. Die Abbildung zeigt das Hamming-Gewicht der zu den Instanzen gehörenden IVs gegen die Anzahl der quadratischen Monome μ in F. Die Sättigung der quadratischen Monome setzt früher ein und die Kurven wirken dadurch ebener. Die Variablen, die noch nicht gesättigt sind, finden sich in linearen Termen.

Insgesamt bekommen wir ab einem Punkt mehr Ausgabegleichungen, die das System definieren, als neue Variablen und Monome. Also wird unser System bestimmt oder gar

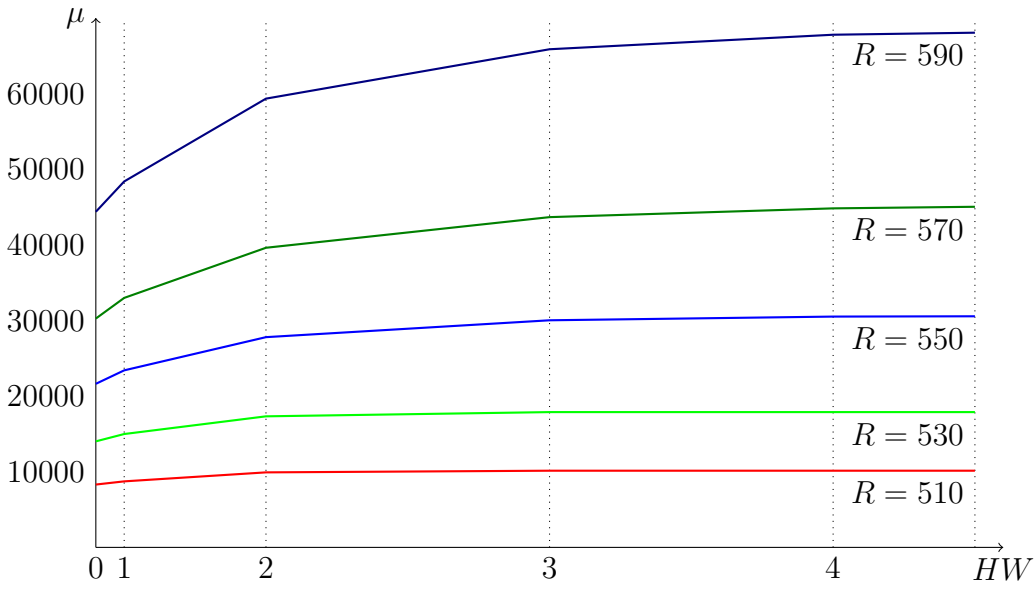


Abbildung 3.4: Sättigung der Monome für Triviuminstanzen mit hohem Hamming-Gewicht des Startwerts

überbestimmt, wenn wir genügend Triviuminstanzen berechnen. Die Bilder erstellten wir, ohne die Ausgabe von Trivium zu berücksichtigen. Die Ausgabe von Trivium ändert jedoch nichts an der Variablenanzahl. Allerdings führen wir neue Monome in das System ein.

3.2 Katan

In [CDK09] wird von Cannière et al. eine Familie von kleinen und effizienten Blockchiffren vorgestellt. Sie heißt die Katan/Ktantan Familie von Blockchiffren. Dabei unterscheiden sich Katan und Ktantan nur durch den Algorithmus zur Aufstellung des Rundenschlüssels. Wir beschränken uns hier auf Katan, weil Ktantan auf Grund der Aufstellung Schwachstellen aufweist, die in [BR10] aufgezeigt werden.

Es gibt 3 Varianten von Katan. Der Hauptunterschied ist die Blocklänge der einzelnen Chiffren. Es gibt Katan mit $32, 48$ und 64 Bit Blocklänge. Die Schlüssellänge beträgt immer 80 Bit. Bei den Varianten mit 48 und 64 Bit Blocklänge wird die Aktualisierungsfunktion darüber hinaus zwei beziehungsweise drei Mal pro Runde mit demselben Rundenschlüssel ausgeführt. Wir beschäftigen uns im Folgenden mit Katan32 und bezeichnen es der Einfachheit halber als Katan.

Anders als bei Stromchiffren addieren wir den Klartext nicht zu einem Schlüsselstrom, sondern binden den Klartext in die Herstellung des Geheimtexts ein.

Auch bei Katan benutzen wir Addition und Multiplikation über $\mathbb{F}_2$. Wir betrachten zwei Schieberegister A und B mit $A := (A_i, \ldots, A_{i-12})$ und $B := (B_i, \ldots, B_{i-18})$. Diese

initialisieren wir mit

$$A = (p_{19}, \ldots, p_{31})$$
$$B = (p_0, \ldots, p_{18}),$$

wobei $P = (p_0, \ldots, p_{31})$ der aktuelle Klartextblock ist.

Danach aktualisieren wir die beiden Register Katans R Runden lang mit der Aktualisierungsfunktion

$$A_i = B_{i-18} + B_{i-7} + B_{i-12} \cdot B_{i-10} + B_{i-8} \cdot B_{i-3} + K_{2r+1}$$
$$B_i = A_{i-12} + A_{i-7} + A_{i-8} \cdot A_{i-5} + A_{i-3} \cdot f_r + K_{2r}$$

für die Runde r. Dabei bezeichnet K_x die Ausgabe eines linearen Schieberegisters, das den aktuellen Rundenschlüssel ausgibt, und f bezeichnet ein weiteres lineares Schieberegister. Volles Katan nutzt $R = 254$ Runden.

Katan benutzt das Schieberegister f, um ein quadratisches Monom irregulär in die Aktualisierungsfunktion hinzuzufügen. Dabei ist f ein mit Einsen initialisiertes Schieberegister von 8 Bit Länge, das durch

$$f_i = f_{i-1} + f_{i-3} + f_{i-5} + f_{i-7}$$

aktualisiert wird. Nach 254 Runden ist das Schieberegister wieder mit Einsen besetzt und die Werte wiederholen sich. Da wir an der Eingabe des Schieberegisters nichts ändern, können wir einfach alle Werte einmal berechnen und speichern.

Zur Aufstellung des Rundenschlüssels initialisieren wir ein 80 Bit langes lineares Schieberegister K mit dem geheimen Schlüssel $k_0, \ldots, k_{79}$. Danach aktualisieren wir es zwei Mal pro Runde mittels

$$K_i = K_{i-79} + K_{i-60} + K_{i-49} + K_{i-12}\,.$$

Der Geheimtextblock $Z = (z_0, \ldots, z_{31})$ besteht dann aus der Konkatenation $Z = B||A$ der Register.

Abbildung 3.5 zeigt eine Runde der beschriebenen Konstruktion Katans als Schaltbild.

Bei Katan ist die Aktualisierungsfunktion unser Substitutionsschritt der Blockchiffre aus Abschnitt 2.1.1 und das Schieberegister fungiert als Permutation der Werte in einer Runde. Auch wenn die Substitution und die Permutation pro Runde sehr einfach sind, sehen wir in den Abschnitten 3.2.1 und 6.3, dass damit eine sichere Chiffre konstruiert wurde. Insgesamt ist Katan eine Feistel-Chiffre, die wir in Abschnitt 2.1.1 kennenlernten.

Umgekehrt müssen wir eine Nachricht auch entschlüsseln können. Während wir bei Trivium einfach den gleichen Schlüsselstrom produzieren und ihn zum Geheimtext über $\mathbb{F}_2$ addieren, müssen wir bei Katan den Klartext aus dem Geheimtext ableiten, indem wir die Chiffre rückwärts durchlaufen.

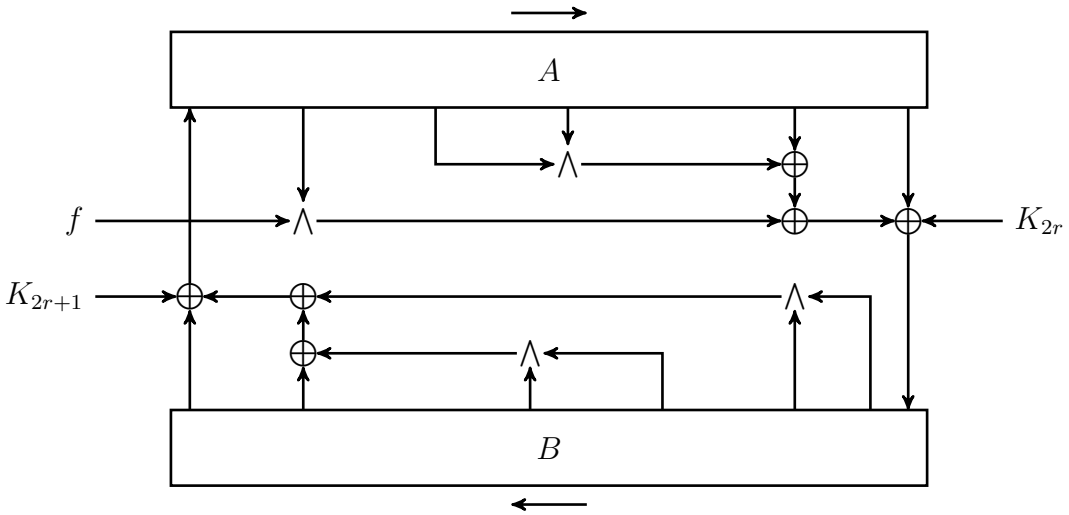


Abbildung 3.5: Schaltbild einer Runde der Verschlüsselung Katans

Dazu speichern wir einen Geheimtextblock in die Register A und B

$$A := (c_{19}, \ldots, c_{31})$$
$$B := (c_0, \ldots, c_{18})$$

und speichern alle Rundenschlüssel, während wir die Aufstellung der Rundenschlüssel einmal ganz ausführen. Die Liste der irregulären Werte aus dem linearen Schieberegister f speichern wir ebenso. Danach führen wir die inverse Aktualisierungsfunktion

$$A_{i-12} = s_B + A_{i-7} + A_{i-8} \cdot A_{i-5} + A_{i-3} \cdot f_r + K_{2r}$$
$$B_{i-18} = s_A + B_{i-7} + B_{i-12} \cdot B_{i-10} + B_{i-8} \cdot B_{i-3} + K_{2r+1}$$

rückwärts aus. Das heißt, wir speichern die ersten Bits $s_A = A_i$ und $s_B = B_i$ und schieben die Register nach links. Also wird das erste Bit, das wir gerade gespeichert haben, überschrieben. Danach wenden wir die inverse Aktualisierungsfunktion an und speichern das Ergebnis als letzten Eintrag des jeweiligen Schieberegisters.

Auf Katan führen wir genau wie auf Trivium einen Angriff im „Gewählter Klartext"-Szenario aus, in dem wir den Schlüssel finden wollen. Das heißt, wir haben Zugriff auf ein Orakel, dem wir den Klartext geben. Das Orakel führt wieder R Runden von Katan aus und schickt uns den Geheimtext zurück. Das Sicherheitsspiel dazu haben wir in Abschnitt 2.1.2 kennen gelernt.

3.2.1 Angriffe auf Katan32

Ausgenommen von einem Angriff auf Ktantan in [BR10] ist die Katan-Familie ungebrochen. Der Angriff in [BR10] überträgt sich nicht auf die restliche Katan-Familie.

Auf Katan wurden hauptsächlich differentielle, Meet-in-the-Middle, algebraische und Seitenkanal-Angriffe durchgeführt. Auf Seitenkanal-Angriffe, insbesondere Fehler-Angriffe, gehen wir in Kapitel 5 ein.

Differentielle Kryptoanalyse auf Katan wird in [AL12] und [KMNP11] durchgeführt. In [KMNP11] wird ein Angriff mit verwandten Schlüsseln ausgeführt. Dabei benutzen die Autoren mehrere Katan-Instanzen mit einem Unterschied im Klartext und im Schlüssel. Das ist ähnlich zum Cube-Angriff aus Kapitel 4, nur dass wir auch Schlüsselbits verändern. Das klingt erst einmal merkwürdig, da wir in vielen Applikationen den Schlüssel nicht ändern können, jedoch wird der Schlüssel von manchen Geräten regelmäßig geändert, damit derselbe Schlüssel nicht zu oft verwendet wird. Die Autoren von [KMNP11] haben eine differentielle Charakteristik nach 120 Runden mit einer Verzerrung von 2^{-21} gefunden. Damit können sie 10 Schlüsselvariablen verifizieren. Für jede der 2^{10} Möglichkeiten für diese Schlüsselbits müssen sie also 2^{21} Katan-Berechnungen durchführen. Insgesamt benötigt der Angriff damit 2^{31} Durchläufe von Katan. Dabei benutzen die Autoren 4 gewählte Schlüsselbits, die sie variieren, und 3 Klartextbits. Mit diesem Angriff waren die Autoren in der Lage 10 Schlüsselbits aus frühen Runden zu berechnen. Das heißt, wenn wir sie kennen, können wir das Verfahren mit einer größeren Rundenanzahl wiederholen und so den ganzen Schlüssel berechnen.

In [AL12] benutzten die Autoren keine verwandten Schlüssel. Stattdessen verwenden die Autoren wieder das „Gewählte Klartext“-Szenario mit mehreren Instanzen, die denselben Schlüssel benutzen, um Katan anzugreifen. Mit Hilfe des Computers werden viele, möglicherweise alle Differentiale gefunden. Die Autoren erhalten so zum Beispiel 16 Differentiale für $R = 95$ Runden. Insgesamt ist es damit möglich, 115 Runden von Katan zu brechen. Diese Charakteristik hat eine Verzerrung 2^{-32} und erlaubt so einen Angriff in 2^{32} Durchläufen von Katan.

Meet-in-the-Middle-Angriffe berechnen vom Klartext ausgehend einen Teil der Register in einer Runde r durch Raten von einigen Schlüsselbits. Parallel werden die Registereinträge der Runde r vom Geheimtext aus auch durch Raten von Schlüsselbits berechnet. Wenn wir dabei dieselben Registereinträge herausbekommen, so haben wir richtig geraten. Durch diese Methode bekommen wir also einen Filter, der falsche Schlüssel aussortiert. Damit können wir einen Brute-Force-Angriff schneller durchführen, weil wir nur einen Teil des Schlüssels raten. In [FM14] wird ein Teil des Registers aus Runde r wieder durch Raten vom Klartext aus berechnet. Vom Geheimtext ausgehend wird jetzt ein Teil des Registers aus Runde $r + \delta$ für $\delta \in \mathbb{N}$ berechnet. Die restlichen δ Runden werden überprüft, indem sich die Autoren überlegen, ob aus dem Teil des Registers aus Runde r der Teil aus Runde $r + \delta$ folgen kann. Das verschlechtert den Filter und braucht sehr viel Speicher für den Abgleich, aber ist genug um 153 Runden von Katan zu brechen. Dabei speichern sie 2^{76} Werte und benötigen $2^{78.5}$ Katan-Berechnungen. Es handelt sich also um einen theoretischen Angriff, der mit heutigen Mitteln nicht praktikabel ist.

Ein Boomerang-Angriff mit verwandten Schlüsseln wird in [ISC13] durchgeführt. Auch hier finden wir differentielle Charakteristiken, um einen Distinguisher zu bauen. In einem Boomerang-Angriff nutzen wir im Gegensatz zu einem normalen Meet-in-the-Middle-Angriff zwei beziehungsweise mehrere Charakteristiken, die mit hoher Wahrscheinlichkeit erfüllt sind, aber nur wenige Runden abdecken. Damit vermögen die Au-

Autor	Methode	R	Zeit	Speicher	Daten
[KMNP11]	Differential (RK)	120	2^{31}	–	2^{31}
[AL12]	Differential	115	2^{32}	–	2^{32}
[FM14]	Meet-in-the-Middle	153	$2^{78.5}$	2^{76}	$2^{78.5}$
[ISC13]	Meet-in-the-Middle (RK)	174	$2^{78.5}$	$2^{26.6}$	$2^{27.6}$
[BCJ+10]	Algebraisch	79	15 min + 2^{45}	–	2^5

Tabelle 3.2: Angriffe auf Katan mit R Runden, Zeit- und Datenkomplexität

toren $R = 174$ Runden in $2^{78.5}$ Katanschritten Zeit und mit Hilfe von $2^{27.6}$ Klartext-Geheimtext-Paaren zu brechen.

Algebraische Angriffe auf Katan werden mit Hilfe von SAT-Solvern, d.h. Algorithmen, die Systeme logischer Klauseln lösen können, durchgeführt. Dabei wird in [BCJ+10] das native quadratische algebraische System, das wir in 3.2.2 beschreiben, in logische Klauseln umgeschrieben. Danach wird die sogenannte Konjunktive Normalform (CNF) von einem SAT-Solver gelöst. Wenn 45 Schlüsselvariablen fest gehalten und 20 Instanzen von Katan betrachtet werden, ist es damit möglich, 79 Runden zu brechen.

Laut [CDK] ist das selbsternannte Sicherheitsziel der Katan-Familie, dass der durch die Aktualisierungsfunktion entstehende Grad des Gleichungssystems so hoch ist, dass die Chiffre auch für $R = 127$ Runden algebraischen Angriffen widersteht.

Seitenkanal- und differentielle Fehler-Angriffe auf Katan beschreiben wir in Kapitel 5. Dabei handelt es sich um physikalische Angriffe. Wir benötigen also ein Gerät, auf dem Katan zur Verschlüsselung benutzt wird, um einen Angriff auszuführen.

In Tabelle 3.2 sind die wichtigsten Angriffe auf Katan noch einmal aufgelistet.

3.2.2 Katan und Trivium

In diesem Abschnitt modellieren wir Katan mit Hilfe ähnlicher Variablen und vergleichen die Ergebnisse dabei mit denen in Abschnitt 3.1.2.

Bei der Modellierung von Katan gehen wir zunächst wie in Abschnitt 3.1.2 vor. Wir wollen mehrere Kataninstanzen mit gleichem Schlüssel und unterschiedlichen Klartexten modellieren. Daraus leiten wir dann einen Angriff im „Gewählter Klartext“-Szenario ab. Da wir aber auch aus der Rückrichtung Informationen ableiten können, drücken wir beide Richtungen algebraisch aus.

Sei $I \subset P$ eine Untermenge der Menge aller Variablen des Klartextes P. Wir initialisieren das Schieberegister für die Aufstellung des Schlüssels mit einem symbolischen Schlüssel $k_0, \ldots, k_{79} \in P := \mathbb{F}_2[k_0, \ldots, k_{79}, a_{0,0}, \ldots, a_{i,t}, \ldots, b_{0,0}, \ldots, b_{i,t}, \ldots]$ aus dem Booleschen Polynomring mit Schlüssel- und Zwischenvariablen und setzen den bekannten Klartext beziehungsweise Geheimtext in die Register ein. Danach aktualisieren wir die Register R Runden. Währenddessen führen wir in jeder Runde zwei Variablen

$a_{i,t} \in P$ und $b_{i,t} \in P$ in den beiden Registern A_t und B_t ein. Hierbei kennzeichnet t die aktuelle Kataninstanz. Auch hier bekommen wir ein dünn besetztes quadratisches Gleichungssystem mit vielen Variablen. An dieser Stelle bemerken wir den ersten Unterschied in den Gleichungssystemen von Katan und Trivium. Der symbolische Schlüssel ist am Anfang nicht vorhanden. Durch die Rundenschlüssel wird der Schlüssel erst nach und nach in das Gleichungssystem eingespeist. Dadurch haben wir nicht mehr die native Ordnung, die wir bei Trivium haben. Aufgrund der Aufstellung der Rundenschlüssel sind alle Schlüsselbits erst nach Runde 40 im System. Aber bereits nach spätestens $R = 62$ Runden hängt jeder Eintrag der Register quadratisch vom Schlüssel ab, wie das nächste Lemma zeigt. Bei Trivium war das erst nach der Hälfte aller Runden der Fall.

Lemma 3.2.1. *Mit der oben beschriebenen Modellierung ist in jedem Eintrag der Schieberegister A und B spätestens nach* 61 *Runden der Schlüssel in quadratischen Monomen vorhanden.*

Beweis: Wir betrachten die Aktualisierungsfunktion

$$A_i = B_{i-18} + B_{i-7} + B_{i-12} \cdot B_{i-10} + B_{i-8} \cdot B_{i-3} + K_{2r+1}$$
$$B_i = A_{i-12} + A_{i-7} + A_{i-8} \cdot A_{i-5} + A_{i-3} \cdot f_r + K_{2r}$$

für Runden $r < R$. Das Register A wird bereits nach 3 Runden mit quadratischen Einträgen aus B gespeist. Umgekehrt enthält B nach 5 Runden Einträge, die quadratisch aus A berechnet werden. Das heißt, spätestens ab Runde 43 beziehungsweise 45 wird der Schlüssel quadratisch durch $B_{i-8} \cdot B_{i-3}$ in A beziehungsweise $A_{i-8} \cdot A_{i-5}$ in B eingebunden. Nach Runde 43 passiert das in A jede Runde. Da das Schieberegister A insgesamt 19 Bit lang ist, sind die Schlüsselvariablen nach Runde 61 in quadratischen Monomen in jedem Eintrag von A enthalten. Das Schieberegister B ist 13 Bit lang. Also sind die Schlüsselvariablen nach spätestens Runde 58 in quadratischen Monomen von jedem Eintrag in B enthalten. □

Lemma 3.2.2. *Sei $R > 58$. Mit der oben beschriebenen Modellierung gilt für die Anzahl der Zwischenvariablen ν*

$$\nu = 4 \cdot R - 45$$

in einer auf R Runden reduzierten Variante Katans.

Beweis: Wir zählen die benötigten Zwischenvariablen, um eine Kataninstanz zu generieren. Dazu betrachten wir die Aktualisierungsfunktion

$$A_i = B_{i-18} + B_{i-7} + B_{i-12} \cdot B_{i-10} + B_{i-8} \cdot B_{i-3} + K_{2r+1}$$
$$B_i = A_{i-12} + A_{i-7} + A_{i-8} \cdot A_{i-5} + A_{i-3} \cdot f_r + K_{2r}\,.$$

von Katan. Wir führen immer dann eine Zwischenvariable ein, wenn wir ansonsten eine Gleichung vom Grad 3 oder höher erhalten würden. Wir gehen analog zum Beweis zu Trivium vor.

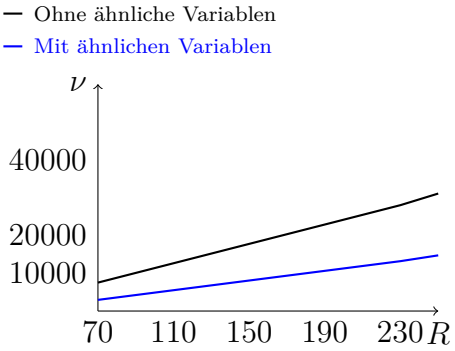


Abbildung 3.6: Gesamtzahl der Variablen für $T = 32$ Kataninstanzen mit und ohne ähnliche Variablen

Aufgrund der Definition des Rundenschlüssels haben wir bereits in der ersten Runde ein Schlüsselbit in den Registern A und B. Es braucht dann 8 Runden, bis die ersten quadratischen Monome durch $B_{i-8} \cdot B_{i-3}$ und $A_{i-8} \cdot A_{i-5}$ erzeugt werden. Nach 3 weiteren Runden muss wegen $A_i = B_{i-8} \cdot B_{i-3} + \ldots$ eine Zwischenvariable eingeführt werden. Danach führen wir in jeder Runde eine Zwischenvariable im Register A ein.

Im Schieberegister B wird wegen $B_i = A_{i-8} \cdot A_{i-5} + \ldots$ erst nach weiteren 5 Runden eine Zwischenvariable nötig. Nach Runde 5 brauchen wir auch im Register B eine Zwischenvariable in jeder Runde.

Betrachten wir die Rückrichtung analog, fangen wir in Runde $R - 10$ an, Zwischenvariablen in Register A einzuführen. Ferner führen wir ab Runde $R-13$ auch in Register B Zwischenvariablen ein.

Damit folgt $\nu = R - 10 + R - 12 + R - 10 + R - 13 = 4 \cdot R - 45$. □

Auch hier sparen wir viele Variablen durch die Einführung ähnlicher Variablen. Abbildung 3.6 zeigt experimentelle Ergebnisse für 32 Kataninstanzen. Im Vergleich zu Abbildung 3.2 befinden wir uns bei Katan direkt im linearen Anstieg der Variablen. Die Phase, in der wir fast keine neuen Variablen bekommen, ist bei Katan sehr schnell vorbei. Das liegt vor allem daran, dass die Register von Katan sehr viel kleiner sind als die Register von Trivium. Wie wir in Lemma 3.2.1 beschrieben haben, hängen die Einträge der Register von Katan ab Runde 61 quadratisch vom Schlüssel ab. Das ist nicht einmal ein Viertel der vollen Chiffre. Trivium braucht dafür die Hälfte 576 aller Runden.

Darüber hinaus ist die Steigung im Vergleich zu der normalen Steigung der Variablen geringer als bei Trivium. Das heißt, der Effekt von ähnlichen Variablen spart uns mit steigender Rundenanzahl mehr Variablen, als das bei Trivium der Fall war.

Analog zu Abschnitt 3.1.2 geben wir den Algorithmus an, der uns ein Gleichungssystem für $T \in \mathbb{N}$ Kataninstanzen mit R Runden ausgibt. Dabei beschreiben wir nur die Verschlüsselung. Der Algorithmus für die Entschlüsselung ist analog. Im Algorithmus wird das Schieberegister K als Liste ausgelegt, das heißt, wir speichern jeden Wert des Schieberegisters in eine Liste der Länge $80 + 2R$. Die Funktion `representation` stellt zunächst die symbolischen Rundenschlüssel und dann für jede Instanz das Glei-

chungssystem auf. Danach wird die Macaulay-Matrix auf Zeilenstufenform gebracht. Die Funktion `insertLinVars` setzt, wie in Abschnitt 3.1.2 beschrieben, Variablen aus linearen Gleichungen in die quadratischen Gleichungen des Systems ein.

An dieser Stelle haben wir aber nicht die Möglichkeit, die Ordnung so natürlich wie bei Trivium zu wählen. Das liegt daran, dass wir die Chiffre nicht mit dem Schlüssel initialisieren. Dadurch ist der Schlüssel nicht mehr nur in Runde 0 vorhanden, sondern wird in jeder Runde genutzt.

Um eine Ordnung festzulegen, merken wir zunächst an, dass die Aktualisierungsfunktionen der beiden Schieberegister unterschiedlich viele quadratische Monome produzieren. Aus diesem Grund ordnen wir die Zwischenvariablen $a_{t,i} < b_{t,i}$, um zunächst mit den „leichten“ Zwischenvariablen zu rechnen. Des Weiteren ordnen wir unsere Variablen nach Instanzen, um möglichst viele Informationen aus wenigen Instanzen zu erhalten. Da wir den Schlüssel berechnen möchten, sortieren wir die zugehörigen Variablen genau wie bei Gröbnerbasenberechnungen klein. Insgesamt nutzen wir die *degrevlex*-Monomordnung mit

$$k_0 < \ldots < k_{79} < a_{0,0} < \ldots < a_{0,R-1} < b_{0,0} < \ldots < b_{0,R-1} < \ldots < b_{t,R-1}\,.$$

Hier führten wir Experimente durch, um zu dieser Sortierung der Variablen zu gelangen. Insgesamt ist nicht sicher, ob diese Sortierung optimal ist. Leider ist das einzige, das wir darüber sagen können, dass wir keine bessere Sortierung der Variablen gefunden haben. Dieses Problem ist von Gröbnerbasen-Algorithmen bereits bekannt. Unsere Experimente beschreiben wir in Abschnitt 6.3.

Analog zu Abschnitt 3.1.2 ist wiederholtes Anwenden von `echelonize` und `insertLinVars` gleichbedeutend mit dem Gebrauch von ähnlichen Variablen. Dabei benutzen wir bei Katan aber nicht rundenbasierte ähnliche Variablen. Das ist ein gutes Beispiel für die Flexibilität unseres Modells. Von dieser Flexibilität unseres Modells wird auch unser Algorithmus zum Lösen quadratischer Gleichungssysteme in Kapitel 6 profitieren.

Die Laufzeit von Algorithmus 3.2 berechnet sich analog zu Algorithmus 3.1 zu $\mathcal{O}(T\mu^{2.7})$ oder $\mathcal{O}(T\mu^2)$, wenn das System F dünn besetzt ist. Dabei bezeichnet T die Anzahl der Instanzen, für die wir das System erzeugen wollen und μ die maximale Anzahl der quadratischen Monome im System während der Berechnung. Wie bei Trivium können wir keine exakte Monomanzahl μ angeben.
Im Folgenden vergleichen wir noch die Sättigung in den Modellen der beiden Chiffren. Abbildung 3.7 zeigt die Sättigung der Variablen in der algebraischen Repräsentation von Katan. Dabei ist die Anzahl der Variablen ν auf der y-Achse und das HW der gewählten Klartexte auf der x-Achse abgetragen.

Katan benötigt deutlich mehr Instanzen als Trivium, um zu einer Sättigung zu kommen. Bei $R = 70$ Runden können wir noch eine Sättigung bei 32 Instanzen ausmachen. Jedoch sind bei $R = 94$ Runden schon deutlich mehr Instanzen nötig. Das ist ein erstes Anzeichen dafür, dass Katan widerstandsfähiger gegen algebraische Angriffe allgemein und im Besonderen gegen unseren Angriff ist.

Algorithmus 3.2 Berechnung des Gleichungssystems F für Katan mit Hilfe ähnlicher Variablen

function `updateState`(k, out):
F.insert$(a_{t,k} + A[0])$; F.insert$(b_{t,k} + B[0])$
$A[0] \leftarrow a_{t,k}$; $B[0] \leftarrow b_{t,k}$
for $j \leftarrow 0$ **to** 19 **do**
  A[j+1] = A[j]
end for
$A[0] = B[18] + B[7] + B[12]B[10] + B[8]B[3] + K[2k+1]$
for $j \leftarrow 0$ **to** 82 **do**
  B[j+1] = B[j]
end for
$B[0] = A[12] + A[7] + A[8]A[5] + A[3]f(k) + K[2k]$
return

function `representation`(R, n_o, t):
global $F \leftarrow \emptyset$
$K \leftarrow (k_0, \ldots, k_{79}, 0, \ldots, 0)$
for $n \leftarrow 0$ **to** $R-1$ **do**
  $K[80+n] \leftarrow K[n] + K[19+n] + K[30+n] + K[67+n]$
end for
for $n \leftarrow 1$ **to** t **do**
  global $A \leftarrow (p_0, \ldots, p_{18})$; $B \leftarrow (p_{19}, \ldots, p_{31})$
  for $k \leftarrow 0$ **to** $R-1$ **do**
    `updateState`$(k, out =$ **false**$)$
  end for
  `echelonize`(F)
  `insertLinVars`(F)
end for
return F

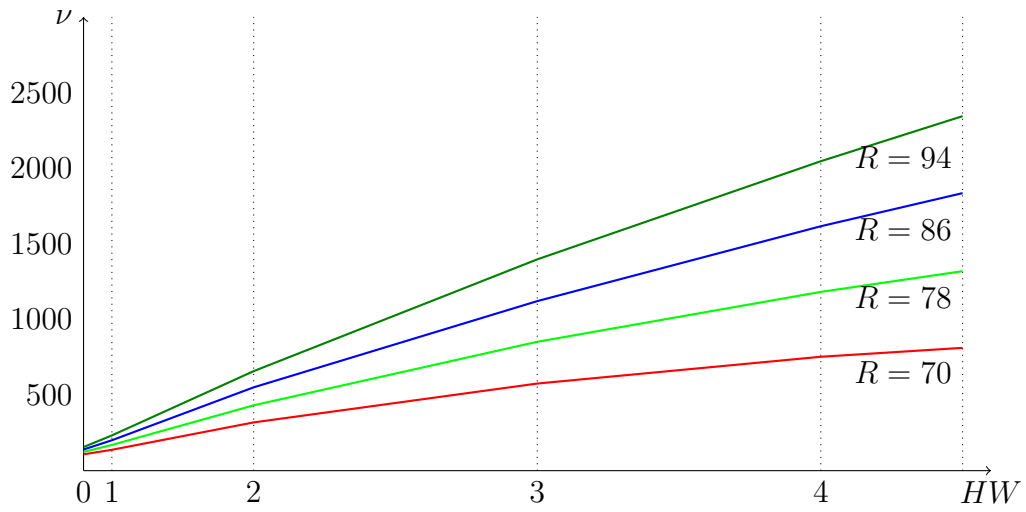


Abbildung 3.7: Sättigung der Variablen für Kataninstanzen mit hohem Hamming-Gewicht des Klartextblocks

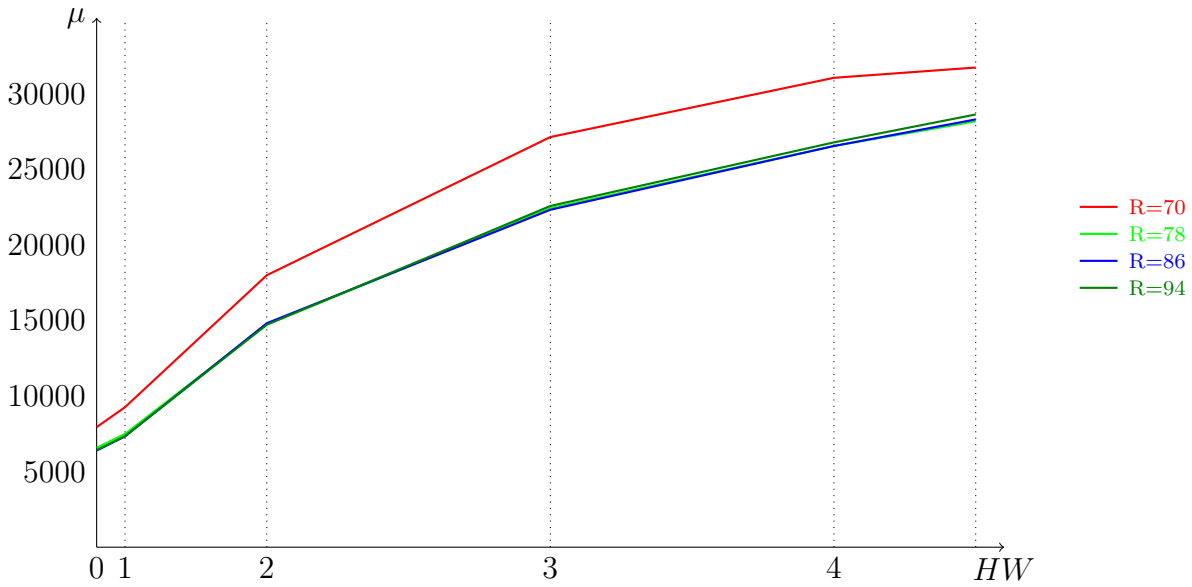


Abbildung 3.8: Sättigung der Monome für Kataninstanzen mit hohem Hamming-Gewicht des Klartextblocks

In Abbildung 3.8 bestätigt sich das weiter. Hier ist die Anzahl der Monome μ gegen das HW des Klartexts abgetragen. Bei $R = 70$ Runden sind mehr Monome im Modell als bei $R \geqslant 78$. Die Anzahl der Monome nimmt aber mit der Anzahl der Instanzen stark ab. Wir sehen also dieselbe Sättigung, die wir auch bei Trivium gesehen haben.

Bei $R \geqslant 78$ haben wir weniger Monome als bei $R = 70$ und die Sättigung ist nicht mehr so stark. Das ist ein schlechtes Zeichen für uns. Wir können weniger Variablen ersetzen, weil die Instanzen zu unabhängig sind. Dadurch erhalten wir nur die Monome,

die die Aktualisierungsfunktion liefert und sind tatsächlich weiter davon entfernt das Gleichungssystem zu lösen.

Wie viele Runden wir von Katan wirklich lösen können, sehen wir in Kapitel 6. Auf Katan führen wir darüber hinaus noch einen Fehler-Angriff durch. Dafür führen wir im Kapitel 5 das Standard-Modell für Fehler-Analysen und einige Verbesserungen vor. Auch führen wir dort unseren ersten Fehler-Angriff durch.

Kapitel 4

Algebraische Eigenschaften des Cube-Angriffs

In diesem Kapitel betrachten wir den Cube-Angriff aus [DS09] von einem algebraischen Standpunkt. Dabei folgen wir den Ausführungen aus [QW14b]. Der Cube-Angriff und die Cube-Tester, eine statistische Variante des Cube-Angriffs, liefern sehr gute Angriffe auf Trivium beziehungsweise Stromchiffren. So wurde die Chiffre Grain-128 mit einem dynamischen Cube-Angriff in [DAS11] und eine auf 799 Runden reduzierte Variante von Trivium gebrochen.

In der Literatur gibt es keine algebraische Definition des Cube-Angriffs. So gestaltet es sich schwer, unterschiedliche Ergebnisse zu vergleichen. Aus diesem Grund definieren wir in diesem Kapitel *Cubes* und leiten daraus resultierende sinnvolle Varianten ab.

Dabei geben wir im Abschnitt 4.1 eine Definition des Cube-Angriff aus [DS09] an und klassifizieren anschließend in Abschnitt 4.2 verschiedene Arten von Cubes. Dabei verallgemeinern wir die Idee des Cube-Angriffs unter verschiedenen Gesichtspunkten. Danach stellen wir in Abschnitt 4.3 Cubes vor, die, anstatt in der Ausgabe einer Chiffre Gleichungen zu liefern, im internen Zustand arbeiten. So vermögen wir Verbindungen zwischen verschieden initialisierten Instanzen einer Chiffre herzustellen. Am Beispiel von Trivium demonstrieren wir diese Konzepte.

Dieses Kapitel ist unabhängig vom Rest der Arbeit. Mit den Techniken aus Kapitel 3 und Abschnitt 6.1 bringen Cubes wenige Vorteile. Trotzdem sind die in diesem Kapitel enthaltenen Ideen aus kryptoanalytischer Sicht sehr nützlich. Darüber hinaus liefert das Kapitel eine exakte Notation und eine algebraische Charakterisierung von unterschiedlichen Cubes für weitere Arbeiten.

Dieses Kapitel entstand in Zusammenarbeit mit Christopher Wolf und Enrico Thomae.
Wir beginnen dieses Kapitel mit einem einführenden Beispiel aus [DS09].

Beispiel 4.0.3. *Wir betrachten das dicht besetzte Polynom* $\texttt{enc} \in \mathbb{F}_2[v_1, v_2, v_3, x_1, x_2, x_3]/\langle v_1^2 + v_1, v_2^2 + v_2, v_3^2 + v_3, x_1^2 + x_1, x_2^2 + x_2, x_3^2 + x_3\rangle$ *mit*

$$\begin{aligned}\texttt{enc}(v_1, v_2, v_3, x_1, x_2, x_3) = {} & v_1v_2v_3 + v_1v_2x_1 + v_1v_3x_1 + v_2v_3x_1 + v_1v_2x_3 + v_1v_3x_2 + v_2v_3x_2 \\ & + v_1v_3x_3 + v_1x_1x_3 + v_3x_2x_3 + x_1x_2x_3 + v_1v_2 + v_1x_3 + v_3x_1 \\ & + x_1x_2 + x_2x_3 + x_2 + v_1 + v_3 + 1\end{aligned}$$

und totalen Grad 3 *in den öffentlichen Variablen* v_1, v_2, v_3 *und den geheimen Variablen* x_1, x_2, x_3*, das die Verschlüsselung einer Chiffre beschreibt.*

Wenn wir in die öffentlichen Variablen alle möglichen 0/1*-Kombinationen einsetzen, vermögen wir* 8 *Polynome von totalem Grad* 3 *in den Schlüsselvariablen zu erzeugen. Diese Polynome sind immer noch dicht besetzt und können nicht ohne Aufwand gelöst werden.*

Wenn wir aber die 4 *Polynome mit* $v_1 = 0$ *und allen* 0/1*-Kombination eingesetzt in* v_2, v_3 *aufaddieren, erhalten wir das lineare Polynome* x_1+x_2*. Bilden wir die Summe über alle Kombinationen und setzen* $v_2 = 0$*, führt das zu dem linearen Polynom* $x_1 + x_2 + x_3$*. Zuletzt halten wir* $v_3 = 0$ *fest und erhalten* $x_1 + x_3$*.*

Durch diese 3 *linearen Polynome sind nun die geheimen Variablen eindeutig bestimmt. Wir müssen nur die Ausgabewerte entsprechend den öffentlichen Werten zu den linearen Gleichungen addieren und erhalten ein lineares Gleichungssystem für den Schlüssel.*

4.1 Der Cube-Angriff

In [DS09] wurde der Cube-Angriff vorgeschlagen. In diesem Abschnitt betrachten wir die Ausführungen aus einem mehr algebraischen Kontext. Wir folgen dabei dennoch den Ausführungen in [DS09].

Sei $n := n_v + n_k$ die Anzahl der Variablen, wobei n_v die Anzahl der öffentlichen Variablen und n_k die Anzahl der geheimen Schlüsselvariablen ist. Weiter seien $V := \{v_1, \ldots, v_{n_v}\}$ die Menge aller öffentlichen IV-Variablen und $K := \{k_1, \ldots, k_{n_k}\}$ die Menge aller geheimen Schlüsselvariablen. Wir definieren eine Boolesche Funktion

$$f : \mathbb{F}_2^n \to \mathbb{F}_2.$$

Zu dieser Funktion gibt es eine eindeutige algebraische Normalform (ANF) über $K \cup V$. Sei $M \subset \{\mu \subset V \cup K\}$ eine Menge von Variablenmengen. Wir schreiben $x_\mu := \prod_{a \in \mu} a$ für das Monom, das durch das Produkt von Variablen aus der Menge μ entsteht. Hierbei setzen wir $\prod_\emptyset := x_\emptyset := 1$. Damit können wir die ANF von f als

$$f(V, K) = \sum_{\mu \in M} \prod_{x \in \mu} x$$

schreiben. Weil wir dadurch Monome als Mengen von Variablen sehen, schreiben wir auch $\mu \in f$ anstatt $\mu \in M$ und $u \in x_\mu$ für Variablen $u \in \mu$.

Definition 4.1.1. *Sei nun $C \subset V$ eine Untermenge der IV-Variablen und $x_C := \prod_{a \in C} a$ ein Monom. Dann können wir f zerlegen in*

$$f(V, K) = x_C p(K) + r(V, K)\,,$$

wobei $p(K)$ ein Polynom in den Schlüsselvariablen K und r ein Restpolynom in $V \cup K$ ist.

Wenn $\nexists \mu \in r : x_C | \mu$ nennen wir C einen Cube *und x_C das zugehörige* Cube-Monom*. Abgekürzt bezeichnen wir auch x_C als Cube. Weiterhin bezeichnen wir $p(K)$ als* Superpoly *des Cubes C.*

Der Grad *eines Cubes ist der totale Grad deg(p) des Superpolys. Der Grad des Cubes entspricht dem Grad des Cube-Monoms.*

Die Dimension *eines Cubes ist die Anzahl $c := |C|$ der Elemente im Cube C.*

Durch das Superpoly $p(K)$ erhalten wir Informationen über die Schlüsselvariablen K der Funktion f.

Für einen k-dimensionalen Cube C sei $I_C \subset \mathcal{P}(V)$ die Untermenge aller 2^k Vektoren aus $\mathbb{F}_2^{|C|}$. Im Cube-Angriff weisen wir den Variablen aus C alle möglichen 0/1-Vektoren aus $\mathbb{F}_2^c$ zu.

In [DS09] wird aus der obigen Konstruktion die folgende Beobachtung abgeleitet.

Satz 4.1.2. *Für ein Polynom p und einen Cube C gilt*

$$p(K) = \sum_{v \in I_C} f(v, K).$$

Beweis: Wir betrachten die Zerlegung $f(V, K) = x_C p(K) + r(V, K)$. Die beliebigen Terme $\mu \in r(V, K)$ addieren sich in der Summe $\sum_{v \in I_C} f(v, K)$ zu Null. Da C ein Cube ist, gilt $x_C \nmid \mu$. Also existiert mindestens eine IV-Variable in C, die nicht in μ auftritt. In obiger Summe addieren wir dadurch μ geradzahlig oft. Da wir über $\mathbb{F}_2$ arbeiten, ist diese Summe Null.

Wenden wir uns dem Polynom $x_C p(K)$ zu. Alle Vektoren $v \in I_C$ außer dem Eins-Vektor $(1, \ldots, 1)$ lassen den Term x_C verschwinden. Also tritt $p(K)$ genau einmal in der Summe $\sum_{v \in I_C} f(v, K)$ auf. □

Die Summe $\sum_{v \in I_C} f(v, K)$ bezeichnen wir als Cube-Summe. Nach Satz 4.1.2 können wir zur Berechnung von Schlüsselbits die Gleichung $p(K)$ benutzen, indem wir die Cube-Summe von C bilden. Dadurch arbeiten wir nur auf einem Teil der Zerlegung von f.

So können wir ein System von Gleichungen aufstellen und Informationen über den Schlüssel erhalten. Der Cube-Angriff verläuft in zwei Phasen:

In der ersten Phase suchen wir Cubes und die zugehörigen Superpolys. Diese Phase ist für eine feste Chiffre unabhängig vom Schlüssel und muss nur einmal ausgeführt werden. Aus diesem Grund nennen wir sie die *Offline-Phase*. Hier werden in der Literatur probabilistische Algorithmen benutzt, die viele IV-Mengen ausprobieren und diese erweitern oder verkleinern, um Cubes zu finden.

Die in der ersten Phase gefundenen Gleichungen können wir in der *Online-Phase* des Cube-Angriffs mit der tatsächlichen Ausgabe einer Instanz der Chiffre gleichsetzen.

Aus dem resultierenden System gewinnen wir dann Teile des Schlüssels oder den ganzen Schlüssel.

Selbst wenn wir nur einen Teil des Schlüssels durch den Cube-Angriff finden, haben wir einen validen Angriff auf das Kryptosystem durchgeführt, wenn wir den Rest des Schlüssels mittels einer ausgiebigen Suche erlangen können. Das gilt nur dann, wenn der Cube-Angriff und die ausgiebige Suche auf die restlichen Schlüsselbits nicht so lange dauern wie die ausgiebige Suche für den ganzen Schlüssel.

Das folgende Beispiel gibt Aufschluss über die Vorgehensweise, mit der wir Cubes finden.

Beispiel 4.1.3. *Sei $K := \{a, b, c\}$ die Menge der Schlüsselvariablen und $V := \{\alpha, \beta, \gamma\}$ die Menge der IV-Variablen. Wir betrachten die folgende Funktion $f : K \cup V \to \mathbb{F}_2$ in algebraischer Normalform:*

$$f(K, V) := ab\alpha + a\beta\gamma + b\beta\gamma + bc\alpha + bc\beta + abc\alpha\gamma + bc + a + b + \beta\gamma + 1$$

Wenn wir f zerlegen, indem wir α aus allen möglichen Termen ausklammern, erhalten wir

$$f(K, V) = \alpha(ab + bc + abc\gamma) + a\beta\gamma + b\beta\gamma + bc\beta + bc + a + b + \beta\gamma + 1.$$

Da wir f mit α auf allen 0/1-Kombinationen und $\beta = \gamma = 0$ auswerten, erhalten wir das Superpoly $ab + bc$ zu α. Eine Liste aller Superpolys lautet wie folgt.

IV Menge	*Superpoly*	*Linear*
$\emptyset$	$bc + a + b + 1$	-
α	$ab + bc$	-
β	bc	-
$\alpha\gamma$	abc	-
$\beta\gamma$	$a + b + 1$	$\star$

Hier ist nur das Superpoly des Cubes $\beta\gamma$ linear und kann damit direkt für einen Angriff benutzt werden. Dennoch geben uns die anderen Superpolys Informationen über den Schlüssel von f.

Cube-Tester. Der Vollständigkeit halber betrachten wir hier kurz sogenannte *Cube-Tester*, die in [ADMS09] vorgestellt worden sind. Bei diesen Cubes sind wir nicht an algebraischen Eigenschaften von Cubes interessiert. Statt dessen suchen wir nach statistischen Eigenschaften, die uns helfen einen Distinguisher zu konstruieren. Zum Beispiel können wir untersuchen, ob ein Superpoly $p(K)$ ausgeglichen ist. Das heißt, ob es mit einer Wahrscheinlichkeit nicht signifikant von $1/2$ verschieden den Wert Null (oder Eins) ausgibt. Dafür benötigen wir die algebraische Normalform des Superpolys nicht, was ein großer Vorteil sein kann. Da wir in dieser Arbeit an algebraischen Angriffen und damit an algebraischen Eigenschaften des Cube-Angriffs interessiert sind, werden wir hier nicht weiter auf Cube-Tester eingehen und verweisen den interessierten Leser auf [ADMS09].

4.2 Klassifizierung von Cubes

In diesem Abschnitt verallgemeinern wir den Cube-Angriff unter verschiedenen Gesichtspunkten. Dabei variieren wir im Abschnitt 4.2.1 den Grad des Superpolys und halten im nächsten Abschnitt 4.2.2 verschiedene Variablen fest. Im Anschluss stellen wir *dynamische Cubes* vor. Diese wurden erstmals in [DAS11] eingeführt. Durch das Setzen von *dynamischen* Variablen wird die Aktualisierungsfunktion der Chiffre vereinfacht und die Chiffre so angreifbar gemacht. In [DAS11] wurde die Stromchiffre Grain-128 durch *dynamische Cubes* gebrochen. Auch können wir nicht nur Schlüsselvariablen im *Superpoly* zulassen, sondern auch öffentliche Variablen. Diese gemischten Cubes stellen wir in Abschnitt 4.2.4 vor. In Abschnitt 4.2.5 reduzieren wir den Grad von Cubes, indem wir die Summe aus Cubes von hohem Grad bilden.

4.2.1 Cubes nach Grad

In diesem Abschnitt sei d der Grad eines Cubes. Wir unterscheiden folgende Fälle: $d = 1$: *lineare Cubes*, $1 \leq d \leq 2$: *quadratische Cubes*, $1 \leq d \leq 3$: *kubische Cubes*, und $d \geq 2$: *Cubes von hohem Grad*. Weiter nennen wir Cubes mit konstantem Superpoly konstante Cubes und unterscheiden zwischen Eins-Cubes und Null-Cubes mit Superpoly $p(K) = 1$ beziehungsweise $p(K) = 0$.

Beispiel 4.2.1. *Wie in Beispiel 4.1.3 sei $K := \{a, b, c\}$ die Menge der Schlüsselvariablen und $V := \{\alpha, \beta, \gamma\}$ die Menge der IV-Variablen. Wir betrachten die Funktion $f : K \cup V \to \mathbb{F}_2$ mit*

$$f(K,V) := ab\alpha + a\alpha\beta + a\beta + a\beta\gamma + b\beta\gamma + bc\alpha + b\beta\gamma + abc\alpha\beta + abc\alpha\gamma + bc + a + 1 + \beta\gamma + \gamma$$

Dies führt zu folgenden Cubes:

Cube	*Superpoly*	*Grad*	*Klassifikation*
$\emptyset$	$bc + a + 1$	*2*	*Quadratisch*
α	$ab + bc$	*2*	*Quadratisch*
β	$a + b$	*1*	***Linear***
γ	*1*	*0*	**Konstant** *oder* **Eins**
$\alpha\beta$	$abc + a$	*3*	Kubisch
$\alpha\gamma$	abc	*3*	Kubisch
$\beta\gamma$	$a + b + 1$	*1*	***Linear***
$\alpha\beta\gamma$	*0*	*0*	**Konstant** *oder* **Null**

Wenn wir nur lineare Cubes verwenden, können wir die Schlüsselvariablen a, b und c nicht bestimmen. Dafür müssen wir ein System aus den resultierenden Gleichungen von den linearen Cubes $\{\beta\gamma, \beta\}$ und zumindest einem der Cubes $\{\alpha, \alpha\beta, \alpha\gamma\}$ entwickeln.

Streng genommen ist $\emptyset$ kein Cube, da $x_C = 1$ für $C = \emptyset$ und daraus $x_C|\mu \ \forall \mu \in r$. Dennoch können wir die resultierende Gleichung verwenden, so dass wir die Gleichung hier angeben.

Konstante Cubes helfen uns nicht bestimmte Schlüsselbits zu finden. Allerdings zeigen sie uns Invarianten oder Nullsummen in der Chiffre an. So vermögen die aus konstanten Cubes gewonnenen Gleichungen uns beim Lösen von Gleichungssystemen aus algebraischen Modellen von Chiffren zu helfen. Auch kann man den statistischen Angriff aus [KMNP11], den wir in Abschnitt 3.1.1 vorgestellt haben, als zwei konstante Cubes in Runde 961 auffassen.

Das Lösen von nichtlinearen Gleichungssystemen gilt als schwierig. In Kapitel 6 versuchen wir deshalb solche Gleichungssysteme zu linearisieren. Dabei betrachten wir jedes Monom einzeln. Im Allgemeinen steigt die Anzahl der Monome im Gleichungssystem mit dem totalen Grad der Gleichungen im System. Aus diesem Grund beschränken wir uns auf quadratische Gleichungssysteme. Also sind Cubes vom Grad größer als zwei für unsere Arbeit nicht weiter interessant.

4.2.2 Cubes durch Setzen von Variablen

Nach der Definition eines Cubes aus Abschnitt 4.1 arbeiten wir auf einem Polynom über allen Variablen aus $K \cup V$. In der Realität können wir mit der Funktion f nicht arbeiten, da die algebraische Beschreibung für eine explizite Darstellung viel zu groß ist. Aus diesem Grund setzen wir *alle* IV-Variablen aus $V \backslash C$ auf beliebige Werte. In der Praxis verwenden wir die Konstante Null. Die nächste Definition beschreibt diese Vorgehensweise.

Definition 4.2.2. *Sei $S \subset V$ die Menge der gesetzten Variablen und $F \subset V$ die Menge der freien, nicht gesetzten Variablen, so dass $S \cup F = V \backslash C, S \cap F = \emptyset$, weswegen $|S| + |F| = |V \backslash C|$. Hierdurch decken wir auch $S = \emptyset$ und $F = \emptyset$ ab.*

Wir nennen $\hat{f} := |F|$ den Freiheitsgrad eines Cubes *C. Weiter sei $\alpha : S \to \mathbb{F}_2$ eine* Belegung *der gesetzten Variablen. Wenn eine Ordnung der IV-Variablen vorliegt, können wir die Belegung α durch einen Vektor $a \in \mathbb{F}_2^{|S|}$ ersetzen.*

Jetzt schränken wir die Funktion f bezüglich der Belegung α ein und erhalten so

$$f(\alpha, V, K) = f(a, V, K) = f(F, K) = x_C p(K) + r(F, K)\,.$$

Dadurch, dass wir die Variablen v_i Null setzen, treten also die Monome $\mu \in f$ mit $v_i|\mu$ in der eingeschränkten Funktion $f(F, K)$ nicht auf.

Basierend auf obiger Definition 4.2.2 unterscheiden wir folgende Fälle.

Gesättigte Cubes: Der Freiheitsgrad ist Null. Das heißt, wir haben $F = \emptyset$, $S = V \backslash C$ und $\hat{f} = |F| = 0$.

Null-Cubes: wie *gesättigte Cubes*, in denen alle Variablen zu Null gesetzt werden. Die Belegung ist also die Nullfunktion $\alpha : I \to 0$.

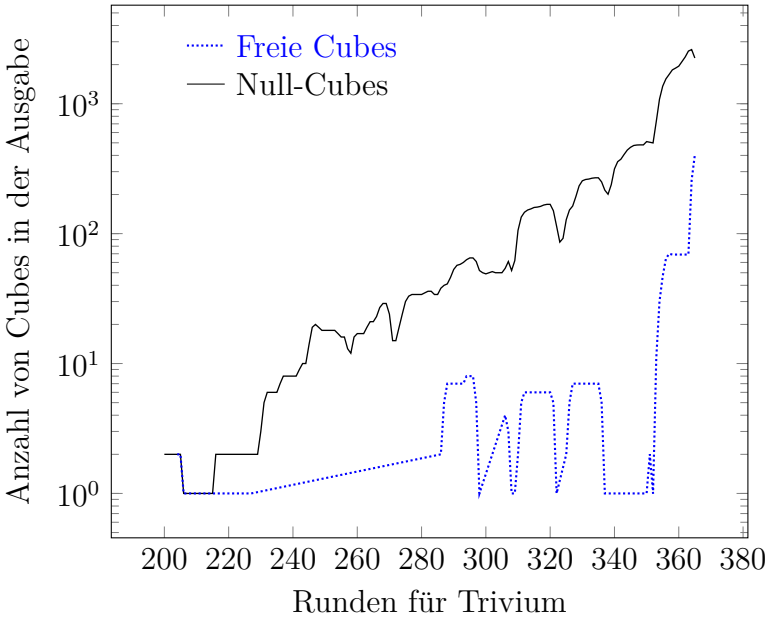


Abbildung 4.1: Anzahl an freien und Null-Cubes in der Ausgabe von rundenreduzierten Varianten von Trivium (Runden 200–365).

Freie Cubes: Hier setzen wir keine Variablen und arbeiten auf der vollen Funktion. Es ist $F = V \backslash C$, $S = \emptyset$ und $f = |V \backslash S|$.

Flexible Cubes: Der Freiheitsgrad ist $1 \leq f < |V \backslash S|$. In diesem Mischfall wird nur ein Teil aller Variablen gesetzt.

Beispiel 4.2.3. *Wie in Beispiel 4.2.1 seien* $K := \{a, b, c\}, V := \{\alpha, \beta, \gamma\}, F := \{\beta, \gamma\}$ *und* $f : K \cup V \to \mathbb{F}_2$. *Weiter sei* $\tau : \{\alpha\} \to 0$ *eine Belegung.*

Wir schränken die Funktion f *aus Beispiel 4.2.1 in der Variablen* α *ein und erhalten so*

$$f(K, F) = a\beta + a\beta\gamma + b\beta\gamma + bc + a + b + \beta\gamma + 1.$$

Dies führt zu den Cubes:

Cube	*Superpoly*	*Grad*	*Klassifizierung*
$\emptyset$	$bc + a + b + 1$	*2*	*quadratisch*
β	a	*1*	***linear***
$\beta\gamma$	$a + b + 1$	*1*	***linear***

Dies sind weniger Cubes als in den Beispielen 4.1.3–4.2.1, weil wir die Funktion f *eingeschränkt haben.*

Setzen von Variablen: Hier beschreiben wir den Effekt auf das Restpolynom $r(V, K)$, den das oben beschriebene Setzen von Variablen hat. Sei dazu μ ein Monom in r mit $|\mu| \geqslant 2$ und $v_i \in \mu$ Variablen des Monoms.

Durch das Setzen der Variablen $v_i = 0$ verschwindet das Monom μ. Darüber hinaus wird ein neuer Cube C möglich, wenn μ das einzige Monom mit $x_C|\mu$ in r ist. Setzen wir andererseits $v_i = 1$, führt das zu neuen Monomen $\mu' = \mu\backslash\{v_i\}$. Falls μ' auch vor dem Einsetzen schon in $r(V, K)$ enthalten ist, verschwindet das Monom μ'. Dieses Verhalten begünstigt auch die Möglichkeit auf einen neuen Cube. Also können beide Fälle hilfreich sein, um neue Cubes für eine Funktion f zu generieren.

Flexible Cubes sind dabei weniger allgemein als freie Cubes, da wir die Anzahl der Variablen verringern, die wir beliebig setzen können, ohne die Cube-Eigenschaft zu zerstören. Jeder freie Cube ist damit auch ein gesättigter Cube für jede Belegung α über $V\backslash C$. Genauso können wir mit einem flexiblen Cube starten und bis zu $|F|$ Variablen setzen, bis der Cube gesättigt ist. Dies können wir in Abbildung 4.1 sehen, da die Anzahl der gesättigten Cubes stets größer gleich der Anzahl der freien Cubes ist. Dabei wird Gleichheit nur sehr selten erreicht.

Kryptoanalytiker bevorzugen gesättigte beziehungsweise Null-Cubes, weil sie häufiger vorkommen als freie Cubes. Also haben wir eine höhere Wahrscheinlichkeit, Null-Cubes zu finden. Wenn wir aber den Cube-Angriff mit anderen Angriffen verbinden wollen, sind freie und flexible Cubes interessanter, weil sie nur eine begrenzte Anzahl an Variablen brauchen. Den Rest können wir für andere Angriffe, wie in Abschnitt 4.2.3 geschehen, verwenden.

4.2.3 Dynamische Cubes

Dynamische Cubes wurden in [DAS11] eingeführt und dazu benutzt Grain-128 zu brechen. Dabei können wir dynamische Cubes als einen Spezialfall von freien Cubes behandeln. Sei $S \subset V$ die Menge aller gesetzten Variablen im naiven Cube-Angriff. Wir betrachten eine Untermenge $D \subset S$, die wir die Menge der dynamischen Variablen nennen. Mit Hilfe dieser Variablen definieren wir eine Funktion $d_i(V, K_i)$ in den Startwert- und ein Teil der Schlüsselvariablen. Jede der Funktionen d_i wählen wir so, dass die Aktualisierungsfunktion einfacher wird. Das erreichen wir, indem wir Einträge des internen Zustands zu Null wählen. Darüber hinaus benutzen wir die Funktionen d_i, um Informationen über die Chiffre zu erhalten, die wir mit *Cube-Testern* nutzen können.

Um solche Funktionen d_i zu finden, ist eine ausgiebige Analyse der internen Struktur der betrachteten Chiffre nötig. Das folgende Beispiel aus [DAS11] illustriert die Idee von dynamischen Cubes.

Beispiel 4.2.4. *Wir betrachten das Polynom P, das wir aufteilen in $P = P_1P_2 + P_3$, wobei P_1, P_2 und P_3 Polynome über den Variablen $k_1, \ldots, k_5$ und $v_1, \ldots, v_5$ mit*

$$P_1 := v_2v_3k_1k_2k_3 + v_3v_4k_1k_3 + v_2k_1 + v_5k_1 + v_1 + v_2 + k_2 + k_3 + k_4 + k_5 + 1$$

$$P_3 := v_1v_4k_3k_4 + v_2k_2k_3 + v_3k_1k_4 + v_4k_2k_4 + v_5k_3k_5k_1k_2k_4 + v_1 + k_2 + k_4$$

und P_2 ist ein beliebiges dichtes Polynom in denselben Variablen.

Wenn wir nun öffentliche Variablen so setzen können, dass das Polynom P_1 gleich Null wird, verschwindet auch das Produkt $P_1 \cdot P_2$ und wir erhalten $P = P_3$. In diesem Beispiel ist P_1 ein einfaches Polynom. Dies ist bei Kryptosystemen häufiger der Fall, da die Aktualisierungsfunktion einfach gehalten wird, um Hardware-Kosten und Zeit zu sparen. Sicherheit wird dann durch eine hohe Rundenanzahl erreicht. Also wird wie in diesem Beispiel ein zufälliges dichtes Polynom mit einem einfachen multipliziert.

Als erstes setzen wir $v_4 = 0$ und benutzen die Linearität von v_1 in P_1, indem wir $v_1 = v_2v_3k_1k_2k_3 + v_2k_1 + v_5k_1 + v_2 + k_2 + k_3 + k_4 + k_5 + 1$ setzen. Dies reicht schon für $P_1 = 0$. Der Wert für v_1 wird während der Cube-Summe geändert. Das unterscheidet den dynamischen Cube-Angriff von der ursprünglichen Definition von Cubes in Abschnitt 4.1.

Die privaten Variablen müssen wir raten um v_1 zu berechnen. Wir benötigen $k_1, k_1k_2k_3$ und $k_2 + k_3 + k_4 + k_5 + 1$. Setzen wir nun v_1 und v_4 in P ein, erhalten wir

$$P = v_2v_3k_1k_2k_3 + v_2k_2k_3 + v_3k_1k_4 + v_5k_3k_5 + k_1k_2k_4 + v_2k_1 + v_2 + k_3 + k_5 + 1\,.$$

Dies ist ein einfacheres Polynom vom Grad 2 in den IV-Variablen und Grad 3 in den Schlüsselvariablen, das wir mit Cubes angreifen können.

4.2.4 Gemischte Cubes

Bis jetzt haben wir uns nicht weit von unserer Definition in Abschnitt 4.1 entfernt. In der nächsten Definition geben wir eine alternative Definition an.

Definition 4.2.5. *Sei $O \subset V$ die Menge der Orakel-Variablen mit $O \cap S = \emptyset$ und f eine Funktion mit*

$$f(V, K) = x_C p(K, O) + r(F, K)\,.$$

Wenn $\forall \mu \in r : x_C \not| \mu$, nennen wir C einen gemischten Cube.

Diese Definition ist auf den ersten Blick von keinem weiteren Nutzen, da wir IV-Variablen in unser Superpoly mischen. Damit können wir allerdings die Dimension des Cubes C reduzieren. So lange $x_C \nmid \mu$ für alle $\mu \in r$ gilt, können wir die Menge C verkleinern, indem wir O vergrößern. Dies verkleinert die Dimension des Cubes. Des Weiteren haben wir die Quelle für (nichtlineare) Gleichungen in den Schlüsselvariablen vergrößert, da wir die Variablen aus O frei wählen können und so unterschiedliche (nicht notwendigerweise linear unabhängige) Varianten des Superpolys $p(K, O)$ erhalten. Durch unterschiedliches Setzen der Orakel-Variable können wir so unterschiedliche Monome in $p(K, O)$ hinzufügen oder wegnehmen. Ferner ist diese Definition weniger restriktiv. Wir können also mehr mögliche Cubes für größere Runden erwarten.

Um mit dieser neuen Definition zu arbeiten, müssen wir die Definition des *Schlüssel-Grades* ändern. Es wird zu der maximalen Anzahl der Schlüsselvariablen in Monomen des Superpolys $p(K, O)$. Also ist

$$\text{key-degree}(f) := \max_{\alpha: O \to \mathbb{B}} \{\deg(p(K, O)_{|\alpha})\}$$

Beispiel 4.2.6. *Wir betrachten die Mengen* $K := \{a, b, c\}$ *und* $V := \{\alpha, \beta, \gamma\}$ *der geheimen beziehungsweise öffentlichen Variablen. Ferner sei* $O = \{\alpha, \beta\}$ *die Menge der Orakel-Variablen und* $f : K \cup V \to \mathbb{F}_2$ *mit*

$$f(K, V) := 1 + a + c + ab + \alpha a + +\gamma a + \gamma b + \alpha\gamma + \alpha\gamma a + \beta\gamma b\,.$$

Dies liefert folgende gemischte Cubes:

Cube	*Superpoly*
$\emptyset$	$ab + a + c + 1 + \alpha a$
γ	$a + b + \alpha + \alpha a + \beta b$

Der einzige Cube nach Definition 4.1.1 ist hier γ*. Ferner beziehen sich die Superpolys auf unterschiedliche Gleichungen, wenn wir unterschiedliche Werte für die IV-Variablen* α, β *einsetzen. Die folgende Tabelle zeigt die einzelnen Möglichkeiten.*

(β, α)	*Superpoly*
$(0, 0)$	$a + b$
$(0, 1)$	$b + 1$
$(1, 0)$	a
$(1, 1)$	1

Mit Hilfe der Superpolys $p(K, (1, 0))$ *und* $p(K, (0, 1))$ *kennen wir den Wert der Schlüsselvariablen* a *und* b*.*

Die Summation der Cubes benötigt keine $2^3 = 8$ *Werte der Funktion* f*, sondern nur* $3 \cdot 2 = 6$*.*

4.2.5 Polycubes

Bis jetzt haben wir Cubes immer mit einem *Cube-Monom* benutzt. In diesem Abschnitt wollen wir dies zu *Cube-Polynomen* erweitern.

Sei dazu $\mathcal{C} = \{C_1, \ldots, C_k\}$ eine Menge von k Cubes mit zugehörigen Superpolys $p_1(K), \ldots, p_k(K)$. Wir betrachten die Summe der Cubes

$$s(\mathcal{C}) := \sum_{i=1}^{k} p_i(K)\,.$$

Hier nennen wir $s(\mathcal{C})$ ein *Polycube*. Für den totalen Grad $\deg(s) = 1$ haben wir einen *linearen* Polycube.

Beispiel 4.2.7. *Wir übernehmen die Notation aus dem vorherigen Beispiel und betrachten die Funktion*

$$f(K, V) := ab\alpha + ab\beta + a\beta + a\beta\gamma + b\beta\gamma + bc\alpha + bc\beta\gamma + \beta\gamma$$

Für die Funktion f *gibt es unter anderem folgende Cubes:*

Cube	*Superpoly*
α	$ab + bc$
β	$ab + a$
$\beta\gamma$	$bc + a + b + 1$

Betrachten wir den Polycube $\mathcal{C} = \{\alpha, \beta, \beta\gamma\}$*, erhalten wir das Superpoly* $s(\mathcal{C}) = b + 1$*. Da der Grad* $\deg s(\mathcal{C}) = 1$*, ist der Cube* $\mathcal{C}$ *für kryptoanalytische Zwecke nützlicher als die einzelnen Cubes* $\{\beta\gamma, \beta\}$ *und* $\{\alpha\}$*.*

Polycubes sind allgemeiner als die bisher betrachteten Cubes. Allerdings sind Polycubes auch schwerer zu finden. Wir können leider keinen effizienten Algorithmus angeben, der Polycubes für ein Kryptosystem berechnet.

4.3 Cubes im internen Zustand einer Chiffre

Cubes sind auf die Ausgabe einer Chiffre beschränkt. Wenn wir in der Ausgabe keine Cubes finden, weil das zu Grunde liegende Polynom nicht geeignet zerlegt werden kann, können wir die zugehörige Chiffre nicht angreifen. Viele Chiffren wenden eine Aktualisierungsfunktion an, bevor die Ausgabe produziert wird. Wir haben in Kapitel 3 zwei solche Chiffren vorgestellt. Unter der Annahme, dass wir die interne Struktur der Chiffre kennen, können wir Cubes im internen Zustand der Chiffre finden. Idealerweise können wir Cubes verwenden, um in der Initialisierungsphase den internen Zustand der Chiffre zu vereinfachen.

Sei $\mathcal{V}$ die Menge aller möglichen öffentlichen Werte, $\mathcal{K}$ die Schlüsselmenge und $\mathcal{S}$ die Menge der möglichen Zustände der Chiffre. Wir bezeichnen die Aktualisierungsfunktion der Chiffre als $u : \mathcal{V} \times \mathcal{K} \times \mathcal{S} \to \mathcal{S}, (I, K, S) \mapsto S'$. Ein Cube in u oder einem Teil davon nennen wir einen *Zustands-Cube*.

Um Cubes im internen Zustand einer Chiffre zu benutzen, brauchen wir eine algebraische Repräsentation, die es uns erlaubt, viele Instanzen zur selben Zeit zu generieren und auf einzelne Bits des internen Zustands symbolisch zuzugreifen. Dann können wir die benötigten Cube-Summen symbolisch berechnen.

Beispiel 4.3.1. *Mit den Modellierungstechniken aus Kapitel 3 können wir sowohl viele Instanzen gleichzeitig behandeln als auch auf einzelne Bits im internen Zustand der Chiffre zugreifen. Also haben wir mit unserem Modell schon alle Voraussetzungen erfüllt, um mit Zustands-Cubes den internen Zustand zu vereinfachen.*

In dieser Arbeit beschäftigen wir uns nicht mit dem Auffinden von Cubes. Dazu verweisen wir auf [DS09] oder [QW14b].

Wir können Cubes im internen Zustand der Chiffre verwenden. Das ist ein großer Vorteil im Vergleich zu den Cubes, die wir bisher kennen gelernt haben. Allerdings führen uns Zustands-Cubes nicht direkt zu Schlüsselbits. Stattdessen liefern sie uns Gleichungen, die verschiedene Instanzen verbinden, wie die Technik ähnlicher Variablen. Im Falle von Trivium haben wir bis Runde 699 *für die ersten* 12 *IV-Variablen in etwa* 31000

Gleichungen gefunden. Das hört sich nach viel an. Jedoch finden wir mit der Technik ähnlicher Variablen aus Abschnitt 3.1 schon die meisten dieser Verbindungen. Dennoch sind Zustands-Cubes interessant. Gerade dann, wenn man nicht mit den Techniken aus Abschnitt 3.1 arbeitet, verbinden Zustands-Cubes die Zwischenvariablen aus unterschiedlichen Instanzen Triviums.

Zum Abschluss dieses Abschnitts hier noch ein praktisches Beispiel mit Trivium, um zu sehen, wie wir mit Zustands-Cubes arbeiten.

Beispiel 4.3.2. *Im internen Zustand von Trivium konnten wir beispielsweise folgende Zustands-Cubes finden.*

Runde	*Cube*	*Superpoly*
b_{302}	v_0	k_{65}
a_{350}	v_{11}	k_{15}
b_{546}	$v_0v_1v_3v_9$	$k_{55} + k_{61}$
c_{586}	$v_0v_1v_2v_3v_4v_5v_7v_8v_9v_{11}$	k_9
a_{601}	$v_0v_2v_3v_4v_5v_6v_7v_8v_9v_{11}$	k_{65}

Betrachten wir jetzt den Zustands-Cube zu b_{302} so erhalten wir die Cube-Summe

$$b_{0,302} + b_{1,302} + k_{65} = 0$$

In dieser Gleichung ersetzen wir $b_{0,302}$ und $b_{1,302}$ durch die definierenden Gleichungen in den Zwischen- und Schlüsselvariablen. Dies führt dann zu einer neuen quadratischen Gleichung für unser System F. Hier kann es passieren, dass $b_{0,302}$ und $b_{1,302}$ schon so weit reduziert sind, dass obige Gleichung trivial wird.

Kapitel 5

Statistische Fehler-Analyse von Katan

In diesem Kapitel lernen wir ein neues Angriffsszenario für kryptographische Primitive kennen. Wie wir in Abschnitt 2.1.2 beschrieben haben, nutzen wir im Fall von Blockchiffren das „Gewählte Klartext"-Szenario und im Fall von Stromchiffren das „Gewählte IV"-Szenario. An dieser Stelle führen wir das sogenannte Fehler-Modell ein. Dabei injizieren wir wiederholt einen Fehler in die Aktualisierungsfunktion der Chiffre und erhalten durch Beobachten der Ausgabeveränderungen Informationen über den Schlüssel der Chiffre. Damit vereinfachen wir die Chiffre aus algebraischer Sicht so weit, dass wir sie angreifen können. Da diese Angriffe voraussetzen, dass wir sie konkret ausführen, zum Beispiel in Hardware oder in Software, sind sie von den bisherigen Angriffen beziehungsweise Angriffsmodellen abgegrenzt, die von einer idealisierten Chiffre ausgehen.

Weiter stellen wir in Abschnitt 5.2 existierende Fehler-Analysen vor und führen in Abschnitt 5.3 unseren ersten Angriff auf eine Chiffre aus.

5.1 Grundlagen für Fehler-Angriffe

Katan und Trivium sind hardware-orientierte Chiffren. Das heißt, sie werden in Geräten wie Chipkarten für Handys, Smartcards oder RFID-Chips benutzt, um Geheimhaltung zu gewährleisten. Wenn wir Zugriff auf eine solche Karte oder ein solches Gerät haben, können wir davon ausgehen, dass wir messen können, wann verschlüsselt beziehungsweise entschlüsselt wird. Dann können wir, wie in [BDL97] und [BS97] beschrieben, einen sogenannten Fehler injizieren. Dabei beschäftigten sich die Autoren von [BDL97] mit asymmetrischen Kryptoverfahren. Diese Verfahren wurden in [BS97] auf symmetrische Primitive verallgemeinert.

Einen Fehler injizieren heißt, dass wir ein Bit im internen Zustand der Chiffre zu Null oder Eins setzen oder es flippen. Aus der Differenz der Ausgabe der fehlerfreien und der fehlerbehafteten Chiffre erhalten wir dann Informationen über den Schlüssel. Dieser Angriff ist bekannt als der DFA.

Für unsere Chiffren lautet das Fehler-Angriffs-Szenario formal wie folgt. Wir geben einem Orakel einen Klartext beziehungsweise einen Startwert und eine Rundenzahl R^*.

Das Orakel führt die Chiffre aus, flippt aber in der Runde R^* zufällige Bits und gibt den fehlerhaften Geheimtext beziehungsweise Schlüsselstrom zurück. Es gibt auch andere Modelle, die mehrere Bits verändern. Allerdings sind Trivium und Katan Bit-orientiert und das beschriebene Modell das Standard-Modell für unsere Chiffren. Auf eine Verbesserung dieses Modells werden wir im weiteren Verlauf dieses Abschnitts eingehen.

Danach setzen wir die Chiffre zurück. Das heißt, wir können dieselbe Eingabe nochmals an das Orakel geben, das zur Verschlüsselung denselben Schlüssel verwendet. Mit den fehlerbehafteten Ausgaben und der fehlerfreien Ausgabe und vor allem der Differenz $\Delta_i = Z + Z'_i$ der einzelnen Ausgaben können wir dann unsere Analyse starten.

Wir merken noch an, dass Fehler-Angriffe die Chiffre mit voller Rundenanzahl angreifen. Da wir unsere Informationen nicht aus dem internen Zustand der Chiffre beziehen, sondern direkt aus der Differenz der Ausgabe, macht es keinen Sinn die Rundenanzahl zu reduzieren.

In Abbildung 5.1 haben wir das Sicherheitsspiel zum generellen Ablauf des dargestellten Fehler-Angriffs noch einmal dargestellt. Der Herausforderer wählt dabei einen geheimen Schlüssel, den wir herausfinden wollen. Danach wählen wir einen Klartext pt_i und eine Runde R_i^*, in der ein Fehler injiziert werden soll. $\mathtt{enc}_K(.)$ ist die fehlerfreie Verschlüsselungsfunktion und $\mathtt{enc}'_K(., R_i^*)$ die Verschlüsselungsfunktion, bei der ein Fehler in Runde R_i^* auftrat.

Nachdem wir als Angreifer die Ausgabe der Chiffre erhalten, analysieren wir diese. Dabei gilt es erst einmal herauszufinden, an welcher Position der Fehler injiziert wurde. Dafür benötigen wir eine Charakteristik.

Definition 5.1.1. *Sei T_C eine Tabelle, in der für alle möglichen Positionen des Fehlers e die Ausgabedifferenz $\Delta = Z + Z'$ der Chiffre abgetragen ist.*

Wir nennen T_C eine Charakteristik, wenn die Tabelle eine Bestimmung der Fehler-Position auf Grund der Ausgabedifferenz zulässt. Die zu Grunde liegende Abbildung von e nach $\Delta = Z + Z'$ ist also injektiv.

Bei den meisten Fehler-Angriffen werden Charakteristiken mit Hilfe von statistischen Auswertungen gefunden. In Beispiel 5.1.2 gehen wir genauer auf Charakteristiken ein.

Danach starten wir mit der Analyse unserer Daten. Wenn die Analyse eines Fehlers abgeschlossen ist, entscheiden wir uns, ob wir noch einen weiteren Fehler injizieren. Wenn wir einen Schlüsselkandidaten haben, testen wir, ob unser Schlüsselkandidat K' gültig ist, indem wir die Chiffre damit ausführen. Falls er das ist, haben wir das Sicherheitsspiel gewonnen.

Bisher war die Beschreibung von Charakteristiken sehr theoretisch. Im folgenden Beispiel veranschaulichen wir daher, wie eine Charakteristik aussieht und wie sie entsteht. Dabei folgen wir den Darstellungen aus [HR08a] für die Erstellung der Charakteristik zu Trivium.

Beispiel 5.1.2. *Bei Trivium ist eine **Berechnung** der Charakteristik möglich. Dies liegt an der Ausgabefunktion*

$$z_i := C_{i-65} + C_{i-110} + A_{i-65} + A_{i-92} + B_{i-68} + B_{i-83}$$

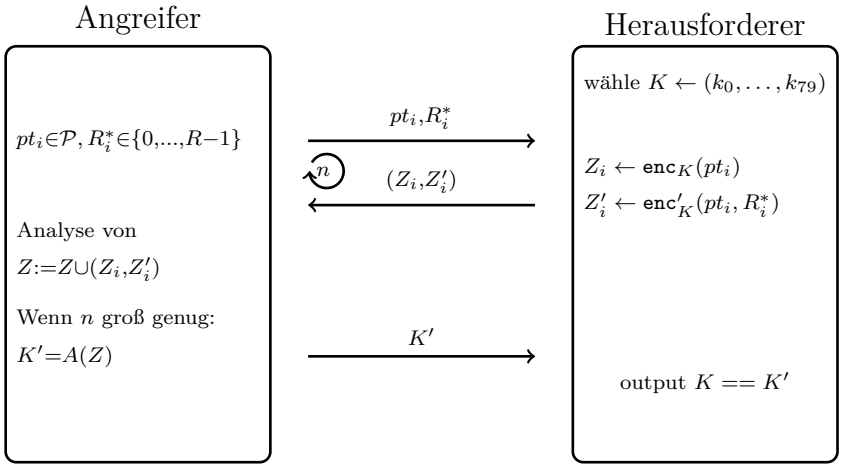


Abbildung 5.1: Sicherheitsspiel für Fehler-Angriffe auf Katan/Trivium

mit der unterschiedlichen Indexdifferenz für die einzelnen Register. Das erlaubt uns die Position des Fehlers zu finden, indem wir in der Differenz $\Delta = Z - Z'$ *der Ausgaben die Position der ersten Eins bestimmen. Im Register A ist die Differenz der Indizes* $92 - 65 = 27$*, im Register B ist die Differenz* $83 - 68 = 15$ *und im Register C ist die Differenz der Indizes* $110 - 65 = 45$*.*

Wir injizieren einen einzelnen Bit-Flip in den internen Zustand $(s_1, \ldots, s_{288})$ *der Chiffre nach der Initialisierung an einer unbekannten Position e. Also ist* $s'_e = s_e + 1$*.*

Der Einfachheit halber nehmen wir für dieses Beispiel an, dass $e \in \{1, \ldots, 66\} \cup \{94, \ldots, 162\} \cup \{178, \ldots, 243\}$*.*

Jetzt produzieren wir von dem fehlerhaften Zustand die Ausgabe $\{z'_i\}_{i \in \mathbb{N}}$ *und von dem fehlerfreien Zustand die Ausgabe* $\{z_i\}_{i \in \mathbb{N}}$*. Sei* δ_a *die erste von Null verschiedene Differenz in der Folge* $(\delta_i)_{i \in \mathbb{N}} = z'_i - z_i$ *der Differenz der Ausgaben und a die zugehörige Position. Also ist* $\delta_a = 1$ *und* $\delta_j = 0 \, \forall 1 \leqslant j < a$*.*

Wenn der Fehler im Register A ist, dann ist $\delta_{a+27} = 1$ *auf Grund der oben beschriebenen Differenz der Indizes. Wenn* $\delta_{a+15} = 1$ *und* $\delta_{a+27} = 0$*, wurde der Fehler im Register B injiziert. Nach derselben Logik ist bei einem Fehler im Register C* $\delta_{a+45} = 1$ *und* $\delta_{a+15} = \delta_{a+27} = 0$*.*

Wenn $e \neq \{1, \ldots, 66\} \cup \{94, \ldots, 162\} \cup \{178, \ldots, 243\}$*, betrachten wir noch die linearen Terme in der Aktualisierungsfunktion. Dann müssen wir an den ersten* 45 *Stellen noch einmal eine analoge Unterscheidung treffen.*

Damit können wir das Register bestimmen, in das der Fehler injiziert wurde. Mit ein wenig mehr Aufwand bestimmen wir auch die exakte Fehlerposition. Für die volle Charakteristik von Trivium verweisen wir auf [HR08a].

Das Beispiel 5.1.2 zeigt einen Weg, wie man durch die Struktur einer Chiffre eine Charakteristik finden kann. Die meisten Charakteristiken werden allerdings durch statistische Analyse gefunden. Das heißt, wir suchen nach Invarianten in der Ausgabe vieler Experimente. Ein weiteres Beispiel für eine Charakteristik geben wir in Anhang A an. Dort berechneten wir eine Charakteristik für Katan mit den Techniken aus Kapi-

tel 6. Darüber hinaus benutzen wir dort ein besseres Fehler-Modell, das wir im Folgenden einführen. In den Tabellen im Anhang listen wir die Charakteristik für die Fehler-Runde $R^* = 242$ auf. Wenn wir die Ausgabe nicht bestimmen können, steht dort ein $-$.

Wir berechnen also die Differenz der Ausgaben $z_i - z'_i$ und vergleichen das Ergebnis mit der Tabelle. Damit bestimmen wir die Position des Fehlers im Falle von Katan eindeutig.

5.2 Bestehende Fehler-Angriffe

Da Fehler-Angriffe auf Trivium bereits in [HR08a, HR08b] und [MBB11] ausführlich durchgeführt und verbessert wurden, konzentrieren wir unsere Ausführungen im weiteren Verlauf auf Katan. Die obigen Veröffentlichungen gehen dabei alle vom oben genannten Angriffs-Szenario mit genau einem geflippten Bit aus.

In [HR08a] wurden differentielle Fehler-Angriffe auf Stromchiffren mit nichtlinearer Aktualisierungsfunktion am Beispiel von Trivium eingeführt. Die erste Schwierigkeit besteht darin, die Charakteristik zu bestimmen. Dazu betrachten die Autoren die Differenz der ersten Ausgabebits $\delta_i = z_i + z'_i$. Da die Register Triviums groß sind, können die Autoren aus der Position a der ersten Eins von δ_i ableiten, wo exakt der Fehler aufgetreten ist. Wir haben dies in Beispiel 5.1.2 beschrieben. So erzeugen sie eine Charakteristik, mit der man die genaue Position des Fehlers finden kann. Danach nutzen Sie alle linearen und ausgewählte quadratische Gleichungen aus der symbolischen Darstellung (ohne Zwischenvariablen), die wir in Abschnitt 3.1 kennen gelernt haben. Für diesen Grundangriff brauchten die Autoren im Durchschnitt 43 Fehler in der letzten Runde von Trivium, um den gesamten Schlüssel zu finden. Die dafür benötigte Zeit ist vernachlässigbar.

Danach verringerten die Autoren von [HR08b] und [MBB11] die Anzahl der benötigten Fehler durch Benutzung einer anderen algebraischen Repräsentation für die Rechnung und eines SAT-Solvers weiter. Dabei gelang es den vollen Schlüssel mit nur 2 Fehlern und 420 Ausgabebits pro Fehler innerhalb von 2 Minuten auf einem Standard-Computer zu berechnen.

Die Zahl der benötigten Fehler ε ist wichtig, da in den angegriffenen Chips Gegenmaßnahmen ergriffen werden können. Zum Beispiel ändert ein RFID-Chip normalerweise den Schlüssel nicht, weil das Strom und Zeit kostet. Allerdings gibt es Wege zu überprüfen, ob ein fehlerhafter Geheimtext ausgegeben wurde, wie zum Beispiel in [JT12] und [MSGR10] aufgezeigt wird. Wenn das zu oft passiert, ändert der Chip den Schlüssel und der Angriff scheitert. Leider gibt es keine *genauen* Spezifikationen, nach wie vielen fehlerhaften Ausgaben bestimmte Chips den Schlüssel ändern. Siehe dazu auch die vom BSI vorgeschlagenen Spezifikationen für Smartcards in [Bun]. Also ist die Güte eines Fehler-Angriffs primär durch die Anzahl der dafür nötigen Fehler bemessen.

Fehler-Angriffe auf Katan haben bisher sehr viel weniger Erfolg als Fehler-Angriffe auf Trivium. In [ALRSS12] wird versucht, den Cube-Angriff aus Kapitel 4 zu benutzen, um Gleichungen für die differentielle Fehler-Analyse zu erhalten. Dabei werden auch Cubes der Dimension Eins genutzt, um eine Charakteristik für $R^* \geqslant 237$ aufzustellen. Im Falle Katans können wir nicht wie bei Trivium in der letzten Runde den Fehler injizieren, da wir dadurch nur triviale Gleichungen erzeugen würden. Das macht die

Autor	Chiffre	ε	τ
[HR08a]	Trivium	41	–
[HR08b]	Trivium	3.2	–
[MBB11]	Trivium	2	–
[ALRSS12]	Katan	115	2^{59}
[SH13]	Katan	140	–

Tabelle 5.1: Bisherige Fehler-Angriffe auf Katan und Trivium mit Anzahl der Fehler ε und Zeitkomplexität τ

differentielle Fehler-Analyse von Katan schwierig. Die Autoren von [ALRSS12] brechen Katan mit $\varepsilon = 115$ Fehlern und nach Raten von 59 Schlüsselbits.

Die Autoren von [SH13] berechnen aus den in [ALRSS12] bekannten Cubes mehr Rundenschlüsselbits durch rekursive Techniken. Außerdem benutzen sie SAT-Solver, um die resultierenden Gleichungssysteme zu lösen. Damit brechen sie Katan in vernachlässigbarer Zeit. Allerdings benötigen die Autoren $\varepsilon = 140$ Fehler für ihren Angriff.

Tabelle 5.1 zeigt noch einmal die wichtigsten Fehler-Angriffe auf unsere Chiffren. Dabei heißt $-$, dass wir nur vernachlässigbar viel Zeit auf einem normalen Computer benötigen.

5.3 Statistische Fehler-Analyse von Katan

In diesem Abschnitt gehen wir näher auf die Probleme bei einer Fehler-Analyse von Katan ein und führen ein realistischeres Fehler-Modell ein.

Darüber hinaus führen wir eine statistische Fehler-Analyse von Katan durch, die einen guten Filter zum Raten liefert und somit den vollen Schlüssel mit nur maximal $\varepsilon = 7,33$ Fehlern mit einer Zeitkomplexität von 2^{74} Katan-Berechnungen. Theoretisch ist damit sogar eine Analyse von Katan mit $\varepsilon = 4,33$ Fehlern und einer Zeitkomplexität von $2^{29.04}$ möglich.

Bei der Fehler-Analyse von Trivium wird ein Fehler direkt nach der Initialisierung der Chiffre injiziert. Das macht Sinn, weil wir durch die Aktualisierungsfunktion nichttriviale Gleichungen in der Differenz der Ausgabe erhalten. Bei Katan ist das nicht so. Wenn wir in der letzten Runde einen Fehler injizieren, so ist unsere Ausgabe außer an der Position des Fehlers dieselbe. Dort finden wir den Bit-Flip $s'_e = s_e + 1$. Daraus erhalten wir keine nützlichen Informationen über den Schlüssel. Also müssen wir den Fehler in früheren Runden injizieren. Bei steigendem Rundendelta $R_\Delta := R - R^*$ ist es schwerer eine Charakteristik anzugeben. In [ALRSS12] und [SH13] finden die Autoren mit Hilfe statischer Analysen Charakteristiken für $R^* \geqslant 237$. Trotzdem benutzen die Autoren von

[ALRSS12] Fehler in niedrigeren Runden. Dabei raten sie viele Variablen. [1]

Ein anderes Problem in der Fehler-Analyse ist das benutzte Modell. Einen Fehler injizieren wir in der Praxis in den Zustand, indem wir beispielsweise mit einem Laser auf den Schaltkreis schießen, in dem die Chiffre ausgeführt wird. Für die elektrotechnische Expertise in dem folgenden realistischeren Fehler-Modell wurde Falk Schellenberg aus dem Lehrstuhl Embedded Security der Ruhr-Universität Bochum zu Hilfe gezogen.

Wir starten bei dem oben beschriebenen Fehler-Modell. Die Zeit und damit die Fehler-Runde R^* können wir gut kontrollieren. Jedoch flippt der Laser nicht genau ein Bit. Es kann passieren, dass wir benachbarte Bits flippen. Auch kann es passieren, dass wir gar kein Bit flippen. Also wählen wir für unser Fehler-Modell eine zufällige Fehlerposition e im internen Zustand der Chiffre aus und bestimmen dann per Zufall einen Fehlervektor $\vec{e} \in \mathbb{F}_2^3$. Weiter gehen wir davon aus, dass wir den Laser nicht neu ausrichten, um weitere Fehler zu injizieren. Die Position e ist also fix und wir brauchen nicht bei jedem Fehler die Position innerhalb des internen Zustands zu bestimmen.

Insgesamt gehen wir wie folgt vor:

1. Bestimme die Fehler-Runde R^*.

2. Wähle zufällig und gleichverteilt eine Position $e \in \{0, \ldots, 31\}$ im internen Zustand der Chiffre und einen Fehler $\vec{e} \in \mathbb{F}_2^3$. Für $e \neq 0 \wedge e \neq 31$ erweitern wir den Fehlervektor $\vec{e}$ durch Nullen, so dass wir einen Vektor in $\mathbb{F}_2^{32}$ erhalten, der an der Position e den zweiten Eintrag von $\vec{e}$ enthält. Diesen neuen Vektor addieren wir zum internen Zustand der Chiffre. Für $e = 0 \vee e = 31$ verfahren wir ebenso, streichen aber den ersten beziehungsweise letzten Eintrag von $\vec{e}$.

3. Wenn die Analyse des aktuellen Fehlers abgeschlossen ist, beginne wieder neu, jedoch ohne eine neue Position zu wählen.

Damit ändert sich die Analyse. Wir können die ersten Fehler dazu verwenden, die exakte Position zu finden. Danach injizieren wir Fehler in Runden $R^* < 241$, die vorher nicht ohne Weiteres möglich waren. In Anhang A findet sich auch eine Charakteristik für $R^* = 242$. Insgesamt war es mit Hilfe des Solvers aus Kapitel 6 möglich, für jedes $R^* \geqslant 241$ eine Charakteristik zu bestimmen.

Hier gibt es aber ein weiteres Problem. Nur die Fehlervektoren $(1, 0, 1)$ und $(1, 1, 1)$ erlauben eine eindeutige Bestimmung der Position des Fehlers mit nur *einem* Fehler. Das heißt, wir haben eine Wahrscheinlichkeit $\mathcal{P}(e|\varepsilon = 1) = \frac{2}{7}$ die Position e mit einem Fehler zu finden, wenn wir eine Gleichverteilung des Fehlervektors annehmen. Wir nehmen eine Gleichverteilung des Fehlers an, weil der Laserpunkt die gleiche Energiedichte über die gesamte Breite des Strahls hat. Wir bemerken schnell, wenn kein Fehler injiziert wurde. Außerdem werden Gegenmaßnahmen nicht wirksam, wenn wir keinen Fehler injizieren. Aus diesem Grund zählen wir den Fehler $\vec{e} = (0, 0, 0)$ in unserer Analyse nicht mit. Also haben wir $2^3 - 1 = 7$ Möglichkeiten für den Fehler.

Nach $\varepsilon = 2$ Fehlern haben wir insgesamt 7^2 Möglichkeiten. Immer, wenn wir einen der Fehlervektoren $\vec{e} \in \{(1, 0, 1), (1, 1, 1)\}$ haben, kennen wir die exakte Position. Wenn

[1] Bei unseren Experimenten ist uns aufgefallen, dass ein Fehler in der Charakteristik von [ALRSS12] vorliegt. Genauer erhielten wir bei Fehlerposition $e = 0$ direkt an erster Stelle eine 1.

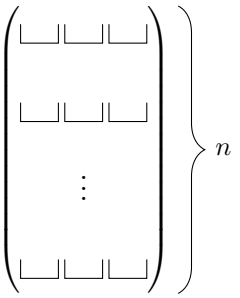
wir $\vec{e} = (1,1,0)$ haben, kann die Position auf dem zweiten Bit liegen oder aber auf dem ersten, weil wir nicht zwischen $(1,1,0)$ und $(0,1,1)$ unterscheiden können. Mit einer analogen Begründung schließen man die restlichen Vektoren aus. Das sind 14 Möglichkeiten bei der ersten Auswahl und $5+5=10$ Möglichkeiten die günstigen Vektoren beim zweiten Mal zu treffen. Darüber hinaus müssen die zwei Fehler für eine genaue Positionsbestimmung zusammen die Möglichkeiten $(1,0,1)$ oder $(1,1,1)$ ergeben. Das heißt, wenn wir $(1,0,0)$ und $(0,0,1)$ haben, ergibt sich daraus ein positives Ereignis. Wenn wir hingegen $(0,1,0)$ und $(0,1,1)$ ziehen, können wir die Position nicht exakt bestimmen. Insgesamt ergibt sich daraus die Wahrscheinlichkeit $\mathcal{P}(e|\varepsilon \leqslant 2) = \frac{32}{49} > \frac{1}{2}$, dass wir die Position nach 2 Fehlern bestimmen.

Nach dieser Einführung berechnen wir die erwartete Anzahl an Fehlern, die wir benötigen, um die Fehler-Position e zu bestimmen. Das folgende Lemma liefert uns die Wahrscheinlichkeit $\mathcal{P}(e|\varepsilon \leqslant n)$ für ein $n \in \mathbb{N}$.

Lemma 5.3.1. *Die Wahrscheinlichkeit die Fehler-Position nach maximal n Fehlern zu kennen, ist*

$$\mathcal{P}(e|\varepsilon \leqslant n) = \frac{1}{7^n} \cdot (7^n - 2 \cdot 3^n + 1)\,.$$

Beweis: Zunächst lassen wir den Fehler-Vektor $(0,0,0)$ zu. Dann haben wir insgesamt 2^3 Möglichkeiten pro Fehler-Vektor. Das macht insgesamt $(2^3)^n$ Möglichkeiten insgesamt. Dies können wir mit der folgenden Abbildung auch anders zählen.



Hier sind die n Fehler-Vektoren untereinander in einer Matrix abgebildet. Jede Stelle kann Null oder Eins sein. Anstatt Zeilenweise zu zählen, betrachten wir Spalten. Insgesamt haben wir dann $(2^n)^3$ Möglichkeiten. Die Wahrscheinlichkeit für die gewünschte spaltenübergreifende Form $(1,x,1), x \in \mathbb{F}_2$ können wir nicht direkt erhalten. Stattdessen berechnen wir die Möglichkeiten, dass diese Form nicht auftritt. Das passiert, wenn $(0,x,y)$ oder $(x,y,0), x,y \in \mathbb{F}_2$ für alle Zeilen der Matrix auftritt. Dafür erhalten wir $2 \cdot (2^n)^2$ Möglichkeiten, da wir eine Spalte komplett festhalten.

Jetzt ziehen wir von den $(2^3)^n$ Möglichkeiten all jene ab, die Nullzeilen enthalten. Es gibt insgesamt $8^n - 7^n$ Möglichkeiten für eine Nullzeile. Wenn wir die Differenz $(2^n)^3 - 2 \cdot (2^n)^2 - (8^n - 7^n) = 7^n - 2^{2n+1}$ bilden, ziehen wir die Möglichkeiten, den Nullvektor in $(0,x,y)$ oder $(x,y,0), x,y \in \mathbb{F}_2$ für alle Zeilen der Matrix zu erhalten, doppelt ab. Diese Anzahl müssen wir wieder addieren.

Im Fall $(0,x,y), x,y \in \mathbb{F}_2$ für alle Zeilen erhalten wir 4^{n-1} Möglichkeiten, wenn wir die erst Spalte und die erste Zeile festhalten. Wenn wir die zweite Zeile festsetzen, erhalten wir 4^{n-2} Möglichkeiten für die folgenden Zeilen, jedoch nur noch 3 Möglichkeiten für die erste, weil wir die Nullzeile dort bereits ausgenommen haben. Insgesamt addieren wir

$$4^{n-1} + 3 \cdot 4^{n-2} + \ldots + 3^{n-2} \cdot 4 + 3^{n-1} = \sum_{k=0}^{n-1} 3^k 4^{n-1-k} = 4^n - 3^n$$

zweimal auf Grund des analogen Falles $(x,y,0), x,y \in \mathbb{F}_2$ für alle Zeilen. Mit Berück-

sichtigung der kompletten Null-Belegung erhalten wir

$$7^n - 2^{2n+1} + 2 \cdot (4^n - 3^n) + 1 = 7^n - 3^n + 1$$

Möglichkeiten, die Fehler-Position nach maximal n Fehlern zu kennen. □

Im folgenden Satz berechnen wir die erwartete Anzahl an Fehlern, die wir benötigen, um die Fehler-Position e zu bestimmen.

Satz 5.3.2. *Der Erwartungswert E des Ereignisses $\{e|\varepsilon\}$, die Fehler-Position nach maximal ε Fehlern zu kennen, ist $E[e|\varepsilon] = 7/3$.*

Beweis: Sei $\omega_\varepsilon, \varepsilon \geqslant 1$ das Ereignis, die Fehler-Position mit *genau* ε Fehlern zu bestimmen. Wir definieren den Ereignisraum $\Omega = \cup_\varepsilon \{e|\varepsilon\}$. Also haben wir $\omega_\varepsilon \in \Omega$ und $\sum_{\omega_\varepsilon \in \Omega} \mathcal{P}(\omega_\varepsilon) = 1$.

Um die Wahrscheinlichkeit $\mathcal{P}(\omega_\varepsilon)$ zu berechnen, stellen wir zunächst fest, dass eine aufsteigende Kette von Wahrscheinlichkeiten $\mathcal{P}(e|\varepsilon = i) \leqslant \mathcal{P}(e|\varepsilon = i+1), i \geqslant 1$ vorliegt. Damit können wir $\mathcal{P}(\omega_\varepsilon)$ berechnen, indem wir für $\varepsilon \geqslant 2$ die Differenz

$$\mathcal{P}(\omega_\varepsilon) = \mathcal{P}(e|\varepsilon) - \sum_{i=1}^{\varepsilon-1} \mathcal{P}(\omega_i) = \mathcal{P}(e|\varepsilon) - \mathcal{P}(e|\varepsilon - 1)$$

berechnen. Es gilt nun $\mathcal{P}(\omega_1) = \frac{2}{7}$, $\mathcal{P}(\omega_2) = \frac{32}{49} - \frac{2}{7} = \frac{18}{49}$ und so weiter.

Der Erwartungswert $E[e|\varepsilon]$ ist gegeben durch

$$\begin{aligned}
E[e|\varepsilon] &= \sum_{\omega \in \Omega} \omega \mathcal{P}(\omega) = \sum_{k=1}^{\infty} \omega_k \mathcal{P}(\omega_k) \\
&= \sum_{k=1}^{\infty} k \cdot (\mathcal{P}(e|k) - \mathcal{P}(e|k-1)) \\
&= \sum_{k=1}^{\infty} k \cdot \left(\frac{1}{7^k} \cdot \left(7^k - 2 \cdot 3^k + 1\right) - \frac{1}{7^{k-1}} \cdot \left(7^{k-1} - 2 \cdot 3^{k-1} + 1\right) \right) \\
&= 2 \sum_{k=1}^{\infty} k \left(\frac{3}{7}\right)^{k-1} - 2 \sum_{k=1}^{\infty} k \left(\frac{3}{7}\right)^{k} + \sum_{k=1}^{\infty} k \left(\frac{1}{7}\right)^{k} - \sum_{k=1}^{\infty} k \left(\frac{1}{7}\right)^{k-1} \\
&= 2 \sum_{k=0}^{\infty} \left(\frac{3}{7}\right)^{k} - \sum_{k=0}^{\infty} \left(\frac{1}{7}\right)^{k} \\
&= \tfrac{14}{4} - \tfrac{7}{6} = \tfrac{7}{3}.
\end{aligned}$$

□

Also nehmen wir im Folgendem an, die Fehler-Position nach durchschnittlich $\varepsilon = 7/3 \approx 2,33$ Fehlern zu kennen.

Bevor wir unseren Filter einführen, rufen wir uns kurz die Dechiffrierung von Katan aus Abschnitt 3.2 ins Gedächtnis zurück. Dort benutzen wir pro Runde zwei Rundenschlüssel, um zu entschlüsseln. Diese Rundenschlüssel wurden durch ein lineares Schieberegister berechnet. Das heißt, wir können den Schlüssel berechnen, wenn wir 80 aufeinanderfolgende Rundenschlüsselbits kennen, indem wir das lineare Schieberegister invertieren.

Unser Filter `filter` in Algorithmus 5.1 übernimmt den Analyseteil des Sicherheitsspiels aus Abbildung 5.1. Die Funktion bekommt den von uns geratenen, benötigten

Algorithmus 5.1 Filter zum Raten der letzten Rundenschlüsselbits in Katan

function $\texttt{eval}_X(S_X, S'_X)$:
$S_\Delta \leftarrow S_X + S'_X$
$e' \leftarrow \emptyset$
for $i \leftarrow 0$ **to** 31 **do**
  if $S_\Delta[i] == 1$ **then**
    e'.append(i)
  end if
end for
if $|e'| > 3$ **then**
  return false
else if $|e'| == 3$ **then**
  return $e'[2] - e'[0] == 2$
else if $|e'| == 2$ **then**
  return $e'[1] - e'[0] \leqslant 2$
else
  return true
end if

function $\texttt{filter}(X, R, R^*, Z_i, Z'_i)$:
$R_\Delta \leftarrow R - R^*$
$S_X \leftarrow \texttt{dec}_X(R_\Delta, Z_i)$
$S'_X \leftarrow \texttt{dec}_X(R_\Delta, Z'_i)$
return $\texttt{eval}(S_X, S'_X)$

Teil des Rundenschlüssels, die Runde R, von der wir starten, die Fehler-Runde R^* und Z_i und Z'_i, die Ausgabe beziehungsweise den internen Zustand der Runde, von der wir starten. Der Algorithmus bekommt Z_i und Z'_i von einem Katan-Orakel und rechnet beide mit dem Algorithmus zum Entschlüsseln $\texttt{dec}_X(r, Z)$ von Katan r Runden mit dem geratenen Rundenschlüssel $X = (x_0, \ldots, x_{2r})$ zurück. Durch die Differenz der internen Zustände S_X und S'_X als Bit-Vektoren erhalten wir den potentiellen Fehlervektor. In der Funktion $\texttt{eval}_X(S_X, S'_X)$ testen wir, ob der Fehlervektor unserer gewünschten Form entspricht. Das heißt, wir prüfen, ob es maximal 3 Einträge gibt, die nicht Null sind. Diese Einträge müssen benachbart zu einer möglichen Fehlerpostion e sein.

Mit dem Algorithmus 5.1 testen wir zufällig gewählte Rundenschlüssel X. Diese müssen $|X| = 2R_\Delta$ lang sein. Damit filtern wir falsche Rundenschlüssel effizient aus, wie wir in Abbildung 5.2 sehen können.

Für die theoretische Analyse betrachten wir eine Differenz von zwei internen Zuständen. Diese hat 32 Einträge. Wir zählen nun die möglichen Fehlervektoren. Das sind Vektoren im $\mathbb{F}_2^3$. Das heißt, wir zählen, wie viele mögliche Vektoren in $\mathbb{F}_2^{32}$ von Hamming-Gewicht kleiner gleich 3 existieren, die benachbarte Einträge haben. Dabei zählen wir auch die 30 Möglichkeiten den Vektor $(1, 0, 1)$ in einen Vektor $\mathbb{F}_2^{32}$ abzubilden.

Ohne weitere Voraussetzungen haben wir für einen Rundenschlüssel 32 Möglichkeiten einen Zustandsunterschied S_Δ von HW 1 zu finden, $31+30$ Möglichkeiten eine Differenz der Zustände von HW 2 zu finden und nochmals 30 Möglichkeiten für S_Δ von HW 3. Zusammen haben wir also $32+31+30+30=123$ mögliche Belegungen des Fehlers e', die unser Filter akzeptiert und 2^{32} mögliche Belegungen für die Differenz der Zustände insgesamt. Damit ist die theoretisch erreichbare Güte σ des Filters $\sigma = \frac{123}{2^{32}} < \frac{2^7}{2^{32}} = 2^{-25}$, wenn wir von einer Gleichverteilung ausgehen. Testen wir also beispielsweise alle möglichen 2^{80} Schlüsselbelegungen, so akzeptieren wir im Schnitt nur 2^{55} Schlüssel. Damit verkleinern wir also den (Runden-)Schlüsselraum. Wenn wir nicht den vollen (Runden-)Schlüsselraum durchlaufen, sondern nur einen kleinen Teil des Schlüsselraums, so können wir einen umsetzbaren Angriff konstruieren.

Im Algorithmus 5.1 ist der oben beschriebene Filter genau spezifiziert. Wenn wir eine passende Charakteristik haben, können wir unseren Filter weiter verbessern. Dabei ist die Kenntnis des exakten Fehlers weniger hilfreich als die Position. Der exakte Fehler hilft uns bei Fehler-Runden $R^* > 240$, da wir dann nur noch genau eine Möglichkeit für den Fehlervektor haben. Das Problem ist, dass wir bis Fehler-Runde $R^* = 241$ maximal 26 Rundenschlüsselbits benutzen können. Nachdem wir diese 26 Bits bestimmt haben, können wir allerdings einen weiteren Fehler mit einer kleineren Rundenanzahl injizieren und so den ganzen Schlüssel finden. Dabei hilft uns die Position deutlich mehr, weil wir damit die theoretische Güte auf $\sigma = \frac{2^3-1}{2^{32}} < \frac{2^3}{2^{32}} = 2^{-29}$ verbessern können.

Es ist klar, dass das Raten und Prüfen der Rundenschlüssel der zeitintensivste Schritt in diesem Angriff ist. Aus diesem Grund kommen wir erst nach der Erläuterung des Angriffs zur Laufzeit. Auch ist klar, dass Algorithmus 5.1 terminiert, weil wir pro Lauf von `filter` höchstens 4 Abfragen machen und eine endliche Schleife durchlaufen.

Bisher haben wir nur die theoretisch erreichbare Güte unseres Filters beschrieben. Abbildung 5.2 zeigt unsere Experimente mit zufällig gewählten Rundenschlüsseln. Für diese Experimente nutzen wir die Fehlerposition nicht und führen 1001 bis 10001 Experimente pro Rundendelta R_Δ durch. Wir führen nur dann 10001 Experimente durch, wenn die Wahrscheinlichkeit $\mathcal{P}(x \textit{ wird angenommen}) \leqslant 2/1001$ ist. In der Abbildung ist die Wahrscheinlichkeit, mit der ein zufällig ausgewählter Schlüssel angenommen wird, gegen das Rundendelta abgetragen.

Bei niedrigem Rundendelta $R_\Delta < 17$ haben wir eine hohe Wahrscheinlichkeit, dass wir falsche Schlüssel annehmen. Die oben beschriebene theoretische Analyse geht von einer Gleichverteilung der Werte im internen Zustand Katans aus. Allerdings haben wir bei niedrigem Rundendelta keine Gleichverteilung im internen Zustand der Chiffre, wie wir auch in den Charakteristiken in Anhang A sehen können. Der Fehlervektor verschiebt sich zuerst innerhalb des Zustandsvektors, bevor er durch den quadratischen Anteil der Aktualisierungsfunktion mit anderen Bits vermischt wird.

Wenn wir die Fehler-Position e kennen, wird der Graph in Abbildung 5.2 zur Null-Linie. Schon bei $R_\Delta = 14$ hat der Filter eine Güte von 2^{-14}. Um die theoretische Güte nachzuprüfen, wären 2^{32} Experimente pro Rundendelta nötig. Dabei können wir keine Parallelisierung anwenden und ein Experiment dauert etwa 8 Sekunden. Also insgesamt $2^{32} \cdot 28 \cdot 8 \approx 2^{40}$ Sekunden. Das sind in etwa 2^{23} Tage. Das können wir mit den uns zur Verfügung stehenden Mitteln nicht durchführen. Allein die 2^{14} Experimente haben

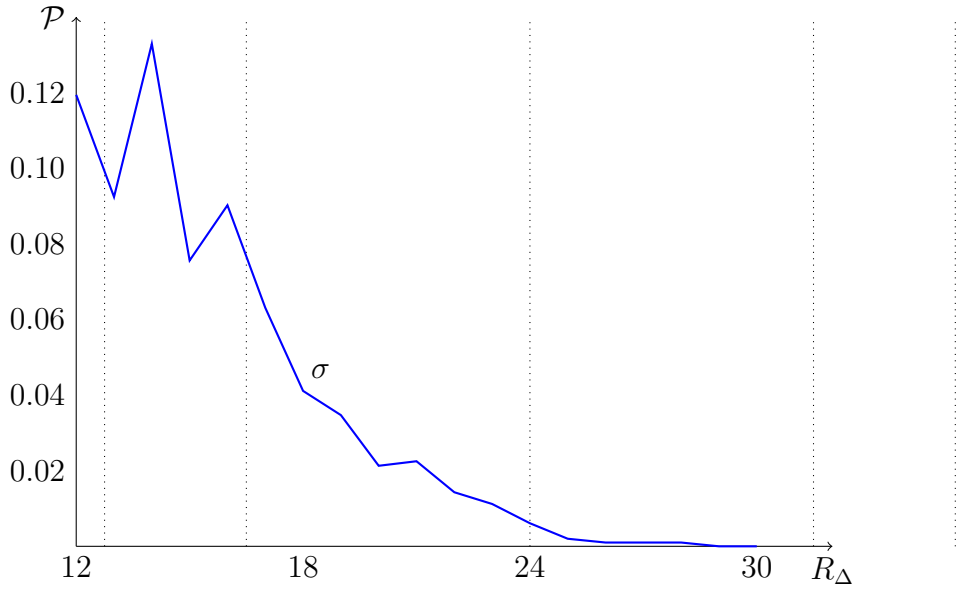


Abbildung 5.2: Empirische Filtergüte in Abhängigkeit von R_Δ; Rundendelta R_Δ gegen die Wahrscheinlichkeit $\mathcal{P}$ auf Annahme eines zufälligen Schlüssels

bereits einen Tag auf dem in Abschnitt 6.1.2 beschriebenen Cluster benötigt.

Der Angriff mit unserem Filter 5.1 und bekannter Position verläuft nun wie folgt. In diesem Angriff gehen wir nicht von der theoretischen Güte des Filters aus, sondern von den gemessenen Werten, die in Abbildung 5.2 gezeigt werden.

1. Injiziere Fehler in die Runde $R^* = 242$ und wähle zufällig Rundenschlüssel $X_1 = (x_1, \ldots, x_{24})$, bis die genaue Position von e fest steht. Da wir eine Filtergüte von $\sigma = 2^{-14}$ haben, erhalten wir dadurch 2^{10} mögliche Zustände der Chiffre.

2. Wähle im Folgenden $R_\Delta = 12$ und setze $R^* = 230$.

3. Rate den Rundenschlüssel $X_2 = (x_{25}, \ldots, x_{49})$ für alle übrigen 2^{10} möglichen Zustände der Runde 242.

4. Wiederhole dieses Verfahren, bis 80 Bits des Rundenschlüssels bekannt sind.

5. Ermittle den Schlüssel K mit Hilfe des Schieberegisters für den Rundenschlüssel.

Die Position zu ermitteln ist mit durchschnittlich $2,33$ Fehlern in Runde 242 möglich. Dafür müssen wir 24 Rundenschlüssel raten. Unsere Filtergüte beträgt allerdings nur 2^{-14}, so dass wir nur 2^{14} Schlüsselbelegungen eliminieren und 2^{10} mögliche interne Zustände in der Runde 242 erhalten.

Danach wählen wir $R_\Delta = 12$. Also raten wir 24 Variablen für die übrigen 2^{10} Zustände, die aus den Schlüsselbelegungen des ersten Schritts entstanden sind. Wir eliminieren von den $2^{10} \cdot 2^{24}$ möglichen Belegungen des internen Zustands 2^{14} Belegungen, so dass wir 2^{20} mögliche Zustände erhalten. Das führen wir höchstens 5 Mal aus und erhalten so den Schlüssel mit insgesamt $7,33$ Fehlern und benötigen dafür

Angriff	ε	τ
[ALRSS12]	115	2^{59}
[SH13]	140	–
Position bekannt	$7,33$	2^{74}
Position + theoretische Güte	$4,33$	$2^{29,04}$

Tabelle 5.2: Fehler-Angriffe auf Katan mit Anzahl der Fehler ε und Zeitkomplexität τ in Katan-Berechnungen

$2^{24} + 2^{24} \cdot (2^{10} + 2^{20} + 2^{30} + 2^{40} + 2^{50}) \approx 2^{74}$ Berechnungen für den Angriff. Dies ist nur ein theoretischer Angriff, den wir nicht auf heutigen Computern durchführen können.

Wir legen allerdings nicht die theoretisch erreichbare Güte zu Grunde. Wenn wir dies tun, erhalten wir statistisch betrachtet immer den richtigen Schlüssel für $R_\Delta \leqslant 14$, da unsere Filtergüte $\sigma = 2^{-29}$ beträgt. Mit $R_\Delta = 14$ raten wir in jedem Schritt 28 Variablen. Wir benötigen durchschnittlich $2,33$ Fehler, um die Position des Fehlers zu bestimmen und raten dabei 24 Rundenschlüssel. Danach wählen wir $R_\Delta = 14$ und raten 28 Rundenschlüsselbits pro Fehler, so dass wir nach 2 weiteren Fehlern den gesamten Schlüssel berechnen können. Dafür brauchen wir $2^{24} + 2 \cdot 2^{28} = 2^{29,04}$ Katan-Berechnungen.

Wir raten statistisch betrachtet immer den korrekten Schlüssel, weil die theoretische Güte $\sigma = 2^{-29}$ beträgt. Also erhalten wir nur eine Möglichkeit für den internen Zustand und verbessern so die Laufzeit des Angriffs gravierend.

Die Ergebnisse dieses Kapitels liegen irgendwo zwischen den empirisch gemessenen Werten und den theoretisch erreichbaren Werten. Das heißt, wir benötigen zwischen $4,33$ und $7,33$ Fehler und $2^{29,04}$ bis 2^{74} Katan-Berechnungen. Da die Laufzeit des Angriffs, dem die empirischen gemessenen Werte zu Grunde liegen, 2^{74} beträgt, können wir den schlimmsten Fall nicht durchführen.
In Tabelle 5.2 listen wir die Ergebnisse dieses Kapitels noch einmal auf.

In Abschnitt 6.3 lernen wir einen algebraischen Fehler-Angriff mit vergleichbaren Ergebnissen kennen. Dort raten wir aber keine Variablen raten und erhalten keine statistischen Ungenauigkeiten.

Kapitel 6

Ein Solver für die Modelle

In diesem Kapitel stellen wir den Solver vor, mit dem wir die Gleichungssysteme aus Kapitel 3 bearbeiten. Dort haben wir die Eigenschaften unserer Modelle kennen gelernt. Jetzt passen wir einen Solver auf diese Eigenschaften an, um erfolgreiche Angriffe nach Definition 2.1.19 zu liefern.

In Abschnitt 6.1 stellen wir die einzelnen Bausteine unseres Solvers vor. Als erstes beschreiben wir ElimLin aus [CSSV12] und die Verbesserungen, die wir durchgeführt haben. Dieser Algorithmus ist gut geeignet, um Gleichungssysteme mit hoher Anzahl von Variablen zu behandeln und setzt ähnliche Variablen auf natürliche Weise um. Ferner integrieren wir einen Algorithmus, genannt SL, für dünn besetzte Gleichungssysteme in unseren Solver.

In Abschnitt 6.1.2 beschreiben wir die Benutzung unseres Solvers in den einzelnen Angriffen. Unser Solver arbeitet in Echtzeit, die wir erst in Chiffre-Berechnungen umrechnen müssen. In Abschnitt 6.1.3 lernen wir diese Umrechnung von Sekunden in Trivium- oder Katan-Berechnungen kennen.

Danach verwenden wir in Abschnitt 6.2 unseren Solver, um einen algebraischen Angriff auf Trivium durchzuführen. Dieser braucht nur 2^{12} Ausgabebits und $2^{42.2}$ Trivium-Berechnungen für einen Angriff auf eine Variante Triviums mit $R = 625$ Initialisierungsrunden.

Zuletzt führen wir in Abschnitt 6.3 und 6.4 zwei Angriffe auf Katan durch. In dem ersten belegen wir erneut die Widerstandsfähigkeit Katans gegenüber algebraischen Angriffen. Wir brechen 80 Runden Katans in $2^{72,3}$ Katan-Berechnungen und raten dabei 40 Variablen. Der zweite Angriff benutzt das „Fehler“-Szenario mit ähnlichen Ergebnissen wie in Abschnitt 5.3. Allerdings nutzen wir das Modell aus Kapitel 3, um das Raten und die statistischen Ungenauigkeiten zu beseitigen. Dabei ist es uns möglich, Katan mit $7,47$ Fehlern in $2^{27,1}$ Berechnungen zu brechen.

6.1 Lösen dünn besetzter Gleichungssysteme

In diesem Abschnitt führen wir unseren Algorithmus zum Lösen der Gleichungssysteme ein, die wir in Kapitel 3 vorgestellt haben. Für unsere Modelle benötigen wir einen Solver für dünn besetzte, quadratische, polynomielle Gleichungssysteme über $\mathbb{F}_2$. Wir

beschreiben eine Implementierung, die ungefähr 10^6 Gleichungen in ebenso vielen Variablen bearbeiten kann. Die Implementierung besteht aus circa 2500 Zeilen *C++*-Code, die auf der beigefügten CD eingesehen werden können. Der *C++*-Code wurde zu einem großen Teil in erster Version von Dr. Christopher Wolf implementiert und von mir ausgetestet und korrigiert. Dieser Hauptteil wurde speziell für quadratische Polynome, ElimLin und SL über $\mathbb{F}_2$ geschrieben und ist in Abschnitt 6.1.1 beschrieben. Auf diesen Solver können wir über ein Python-Frontend zugreifen. Unsere algebraische Repräsentation der Chiffren aus Kapitel 3 ist in Python umgesetzt und wird dem Solver als String übergeben.

Danach beschreiben wir in Abschnitt 6.1.2, wie wir mit diesem Solver Gleichungssysteme lösen, und erklären danach die Zeitmessung, die wir benutzen, um die Zeit, die unser Solver benötigt, in Trivium- beziehungsweise Katan-Berechnungen umzurechnen.

6.1.1 Die Kernstücke des Solvers

Der Kern unseres Solvers ist ElimLin, das in [CB07] vorgestellt und in [CSSV12] weiter betrachtet wurde. ElimLin löst quadratische, polynomielle Gleichungssysteme durch Substitution von Variablen aus linearen Gleichungen. Mit Hilfe von ElimLin können laut [CB07] Gleichungssysteme mit 3056 Gleichungen in 2900 Variablen und insgesamt 4331 Monomen gelöst werden. Damit wurden die Kryptosysteme CTC2, LBlock, MIBS und DES angegriffen, wie in [CSSV12, CB07] zu lesen ist. Beispielsweise wurden 6 Runden der „Courtois Toy Cipher“ (CTC2) mit nur 64 gewählten Geheimtexten gebrochen. Dafür benötigten die Autoren aber 180 Stunden und rieten 210 Variablen. Auch wurden 5 von 16 Runden mit 3 bekannten Klartexten von DES in 173 Sekunden gebrochen. Auf der Hardware, auf dem dieser Angriff durchgeführt wurde, benötigt ein Brute-Force-Angriff ungefähr 540 Sekunden. Dazu wurden 23 von 64 Schlüsselvariablen geraten. Allerdings wurde DES bereits mehrere Male gebrochen, wie zum Beispiel in [BS93] beschrieben ist.

ElimLin aus [CB07] soll gut mit großen dünn besetzten Gleichungssystemen umgehen können, da es den Grad der Gleichungssysteme erhält und Heuristiken zur Erhaltung der Struktur benutzt.

In Algorithmus 6.1 ist ElimLin in der Version aus [CSSV12] dargestellt. Wir übergeben der Funktion `elimLin` das System F. Der Ausgangsalgorithmus benutzt keine spezifische Monomordnung, um das System darzustellen. Das System wird als Macaulay-Matrix und einer beliebigen gradierten Monomordnung interpretiert. Die Funktion `echelonize` bringt diese Matrix dann auf Zeilenstufenform. Das System zu dieser Zeilenstufenform wird in ein lineares Gleichungssystem L und ein quadratisches Gleichungssystem Q aufgeteilt. Die Funktion `split` liefert uns diese Gleichungssysteme aus der Matrix M_F. Danach wählen wir für jede lineare Gleichung $\ell \in L \subset \mathbb{F}_2[x_1, \ldots, x_n]$ eine Variable $\nu \in \ell$ und eliminieren sie aus Q, indem wir ν substituieren. Die `while`-Schleife terminiert, wenn das System F gelöst ist oder wir keine lineare Gleichung mehr in F haben.

Die Laufzeit des ElimLin-Algorithmus ist $\mathcal{O}(\nu_0^{d_F+1} m_0^2)$, wie in [CSSV12] beschrieben ist. Dabei ist ν_0 beziehungsweise m_0 die Anzahl der Variablen beziehungsweise Glei-

Algorithmus 6.1 ElimLin für quadratische, polynomielle Gleichungssysteme F

```
function elimLin(F):
L' ← ∅
flag ← true
while flag==true do
  flag ← false
  M_F = macaulay(F)
  M_F = echelonize(M_F)
  L, Q = split(M_F)
  for each ℓ ∈ L in an arbitrary order do
    if ℓ is trivial then
      if ℓ is unsolvable then
        return ∅
      end if
    else
      flag ← true
      Choose ν ∈ ℓ
      Substitute ν in all q ∈ Q
      Substitute ν in all p ∈ L
      L'.append(ℓ)
    end if
  end for
  F ← L' ∪ Q
end while
return F
```

chungen im anfänglichem System und $d_F = \max\{\deg(f)|f \in F\}$ der maximale totale Grad der Gleichungen im System F.

Um dieses Resultat zu zeigen, stellen wir zuerst fest, dass die Erzeugung der Zeilenstufenform in einem Schritt der `while`-Schleife die meiste Zeit fordert.

Die Macaulay-Matrix hat maximal m_0 Zeilen, da wir in einem Schritt von ElimLin die Anzahl der Gleichlungen nicht erhöhen. Weiter entspricht die Spaltenanzahl der Anzahl der Monome μ im aktuellen System. Da der Algorithmus den totalen Grad der Gleichungen nicht erhöht, haben wir maximal

$$\mu \leqslant \sum_{i=0}^{d_F} \binom{\nu_0}{i} = \mathcal{O}(\nu_0^{d_F})$$

Spalten im System.

Also benötigt die Erzeugung der Zeilenstufenform $\mathcal{O}(\mu \cdot m_0^2) = \mathcal{O}(\nu_0^{d_F} \cdot m_0^2)$. Die Anzahl der Iterationen der `while`-Schleife ist beschränkt durch die initiale Anzahl der Variablen im System, weil wir pro Schritt mindestens eine Variable eliminieren oder terminieren.

Also ist die Laufzeit beschränkt durch

$$\mathcal{O}(\nu_0^{2d_F+1} \cdot m_0).$$

Die richtige Auswahl der Variable $\nu \in \ell$ zu finden, ist ein schweres Problem, womit wir uns nach dem nächsten Beispiel beschäftigen.

Beispiel 6.1.1. *Sei* $R = \mathbb{F}_2[a,b,c]/\langle a^2+a, b^2+b, c^2+c\rangle$ *und* F *ein Gleichungssystem über* R *mit*

$$\begin{aligned} ab+bc+c+1 &= 0\\ ac+bc &= 0\\ a+b+c &= 0. \end{aligned}$$

Wir setzen eine Variable aus der linearen Gleichung $a+b+c=0$ *in die quadratischen Gleichungen ein.*

Wenn wir die Variable a *einsetzen, erhalten wir*

$$\begin{aligned} (b+c)b+bc+c+1 &= 0\\ (b+c)c+bc &= 0\\ \hline b+bc+bc+c+1 &= 0\\ bc+c+bc &= 0\\ \hline b+c+1 &= 0\\ c &= 0\\ \hline a+b+c &= 0. \end{aligned}$$

Dies ist ein lineares Gleichungssystem, das wir im nächsten Schritt mit $c=0, b=1, a=1$ *lösen.*

Wenn wir die Variable b *zuerst einsetzen, erhalten wir*

$$\begin{aligned} a(a+c)+(a+c)c+c+1 &= 0\\ ac+(a+c)c &= 0\\ \hline a+ac+ac+c+c+1 &= 0\\ ac+ac+c &= 0\\ \hline a+1 &= 0\\ c &= 0\\ \hline a+b+c &= 0. \end{aligned}$$

Das führt auch zu der Lösung $a=1, c=0, b=1$. *Allerdings ohne die zusätzliche Berechnung der Zeilenstufenform.*

Setzen wir hingegen die Variable c ein, vereinfachen wir das System zu

$$\begin{aligned} ab + b(a+b) + a + b + 1 &= 0 \\ a(a+b) + b(a+b) &= 0 \\ \hline ab + ab + b + a + b + 1 &= 0 \\ a + ab + ab + b &= 0 \\ \hline a + 1 &= 0 \\ a + b &= 0 \\ \hline a + b + c &= 0. \end{aligned}$$

Also müssen wir im Gegensatz zur Wahl von b wieder einen zusätzlichen Schritt machen. Das mag nicht nach viel Arbeit aussehen, allerdings ist dies nur ein kleines Beispiel. Die Wahl, wann wir welche Variable einsetzen, hat großen Einfluss auf den Algorithmus 6.1. Wenn wir die falsche Wahl treffen, können wir manche Systeme nur sehr langsam oder gar nicht lösen. Auch dieses Verhalten kennen wir von Algorithmen, die eine Gröbnerbasis berechnen.

ElimLin aus [CB07] benutzt für die Auswahl der Variablen in Zeile 16 des Algorithmus 6.1 eine Heuristik. Wenn wir ein lineares Polynom ℓ auswählen, zählen wir, wie oft jede Variable $\nu \in \ell$ im restlichen System vorkommt. Die Variable, die am wenigstens oft erscheint, ersetzen wir dann mit Hilfe der linearen Gleichung ℓ. Das soll das Gleichungssystem dünn besetzt halten, da jede Einsetzung in ein quadratisches Monom potentiell neue Monome erzeugt. In Beispiel 6.1.1 treten die Variablen b und c beide 3 mal auf. Die Variable a tritt nur 2 mal auf. Also würden wir die lineare Gleichung nutzen, um a zu eliminieren. Auf den ersten Blick scheint das eine gute Heuristik zu sein. In Systemen mit vielen Variablen haben wir lineare Gleichungen ℓ mit vielen Termen. Wenn wir jetzt eine Variable mit Hilfe einer solchen Gleichung ℓ ersetzen, bekommen wir potenziell viele neue quadratische Monome durch die vorhandenen quadratischen Monome, in die wir einsetzen.

Unsere erste Verbesserung von ElimLin ist die Einführung einer Monomordnung. Monomordnungen haben wir in Abschnitt 2.2.1 vorgestellt. Das erlaubt uns die Wahl der Variablen zu kontrollieren, wie wir es bei Gröbnerbasen in Abschnitt 2.2.2 beschrieben haben. Das ist besonders gut, wenn wir Gleichungssysteme mit viel Struktur haben, wie es in der Kryptographie beziehungsweise den Modellen aus Kapitel 3 der Fall ist.

Gerade bei Trivium haben wir dort beschrieben, wie einfach das Finden einer Ordnung ist, die wir mit ähnlichen Variablen verwenden können. Denn ElimLin mit einer Monomordnung anzuwenden, heißt explizit ähnliche Variablen zu verwenden. Ferner suchen wir nach der Modellierung der Chiffre nicht extra nach ähnlichen Variablen. Wir wenden einfach ElimLin mit einer passenden Monomordnung auf die Instanzen an und machen somit direkt Gebrauch von ähnlichen Variablen.

Trotzdem ist die Auswahl einer Monomordnung ein schwieriges Unterfangen und erfordert wie bei Gröbnerbasen eine sorgsame Untersuchung des Gleichungssystems.

Wie oben beschrieben können wir, abhängig von der Auswahl der Variablen, unter Umständen langsamer oder gar nicht lösen. In unseren Experimenten in Abschnitt 6.2 und 6.3 benutzen wir stets die *degrevlex*-Ordnung mit den Variablenordnungen, die wir in Kapitel 3 beschrieben haben.

ElimLin ist also ein guter Kandidat für sehr große dünn besetzte Systeme mit viel Struktur. Mit der auf multivariate, quadratische Gleichungen über $\mathbb{F}_2$ spezialisierten Implementierung können wir dünn besetzte Systeme mit 10^6 Variablen und Gleichungen behandeln.

Allerdings ist ElimLin nicht in der Lage, polynomielle Gleichungssysteme generell zu lösen wie zum Beispiel Gröbnerbasen-Algorithmen. Wenn wir zu Anfang keine linearen Polynome haben, löst ElimLin das System nicht. Um dem entgegenzuwirken, führen wir einen Algorithmus zur erweiterten Linearisierung für dünn besetzte Gleichungssysteme (SL) ein. Auch wenn wir damit nicht in der Lage sind, allgemeine Gleichungssysteme zu lösen, so werden wir wenigstens nicht aufhören müssen, wenn wir keine linearen Gleichungen haben.

In Abschnitt 2.2.3 haben wir den Algorithmus zur erweiterten Linearisierung von Gleichungssystemen (XL) und den Algorithmus zur erweiterten Linearisierung dünn besetzter Gleichungssysteme (XSL) kennen gelernt. In XL versuchen wir ein überbestimmtes System zu generieren, indem wir die Polynome im System mit allen möglichen Monomen multiplizieren, die wir mit den vorhanden Variablen erzeugen können. Das zerstört die Struktur unserer Systeme und erzeugt über den Substitutions-Schritt ein dicht besetztes System. Weiter erhalten wir viele algebraisch abhängige Gleichungen. Wenn wir das System linearisieren, ist das aber unwichtig, weil wir viele linear unabhängige Gleichungen erhalten. Wir benötigen allerdings mehr Zeit, um das System zu linearisieren. Wenn wir es schaffen, genügend viele linear unabhängige Gleichungen zu erzeugen, wird das System überbestimmt und wir können die Macaulay-Matrix des Gleichungssystems linearisieren, was uns zur Lösung des Systems führt. Wenn wir dabei den Grad D hoch genug wählen, gelingt das immer, wenn wir über endlichen Körpern arbeiten und ausreichend Speicher voraussetzen.

Auch XSL erzeugt neue Monome und erhöht den Grad der Polynome im System. In XSL werden die Polynome nicht mehr mit allen möglichen Monomen multipliziert, sondern nur mit Produkten aus Monomen, die bereits im System vorhanden sind. Dabei erzeugen wir nicht nur neue Gleichungen, sondern wieder neue Monome im System. Die Autoren von [CP02] versuchen, das System dünn besetzt zu halten.

In Algorithmus 6.2 gehen wir noch einen Schritt weiter. Dort beschreiben wir den SL-Algorithmus. Wir wollen sowohl den Grad des Systems als auch die Monomstruktur erhalten. Dazu multiplizieren wir jedes Polynom f mit jeder Variablen im System $\nu(F)$ einzeln. Falls der Grad dabei ansteigt, verwerfen wir das neu erzeugte Polynom νf. Danach prüfen wir, ob die Monome μ des neuen Polynoms νf schon im System vorkommen. Dabei bezeichnet $\mu(F)$ die Menge aller Monome im System F. Ist das der Fall, so fügen wir νf in das Gleichungssystem F ein. Danach versuchen wir das System zu linearisieren. Die Funktion `mat2sys` führt die Macaulay-Matrix wieder in ein Gleichungssystem über.

Das ist sehr restriktiv und wir erhalten nicht so viele Gleichungen wie bei XL bezie-

Algorithmus 6.2 SL für beliebige Gleichungssysteme F

```
function genericSL(F):
d_F = max{deg(f)|f ∈ F}
V ← ν(F)
M ← μ(F)
for each f ∈ F do
  for each ν ∈ V do
    f* = ν · f
    if deg(f*) ⩽ d_F then
      if μ(f*) ∈ M then
        F.append(f*)
      end if
    end if
  end for
end for
M_F = macaulay(F)
M_F = echelonize(M_F)
F = mat2sys(M_F)
return F
```

hungsweise XSL. Allerdings wollen wir das auch gar nicht. Später werden wir ElimLin durchführen, anstatt die Zeilenstufenform der Macaulay-Matrix zu berechnen. In Beispiel 6.1.2 werden wir sehen, welchen großen Vorteil wir dadurch erhalten. Bevor wir zu dem Beispiel kommen, analysieren wir die Laufzeit von Algorithmus 6.2.

Algorithmus 6.2 benötigt $\mathcal{O}(m_1^2 \cdot \mu_0)$ Schritte, um das Gleichungssystem zu modifizieren. Dabei ist $m_1 = m_0 + m_{SL}$ die Summe aus der Anzahl aller Gleichungen m_0 im anfänglichen System und m_{SL}, der Anzahl aller neu erzeugten linear unabhängigen Gleichungen. Die Anzahl aller Monome im System ist μ_0.

Wir generieren $m_0 \cdot \nu_0$ neue Polynome und machen pro neuem Polynom zwei Vergleiche. Dafür brauchen wir $\mathcal{O}(m_0 \cdot \nu_0)$ Schritte. Um die Matrix auf Zeilenstufenform zu bringen, brauchen wir $\mathcal{O}(m_1^2 \cdot \mu_0)$ Schritte, da unsere Macaulay-Matrix m_1 Zeilen und μ_0 Spalten besitzt. Dabei ist m_1 auf Grund unserer Beschränkungen für neue Polynome in derselben Größenordnung wie m_0.

Im folgenden Beispiel 6.1.2 beschreiben wir, wie wir SL im Zusammenspiel mit ElimLin benutzen.

Beispiel 6.1.2. *Wir betrachten das Gleichungssystem* $ab+a=1$ *in* $\mathbb{F}_2[a,b]/\langle a^2+a, b^2+b\rangle$. *Da die Gleichung nicht linear ist, kann ElimLin die Lösung* $a=1, b=0$ *nicht folgern.*

Wenn wir aber einen SL-Schritt ohne die Zeilenstufenform der Macaulay-Matrix anwenden, erhalten wir die Lösung. Dazu multiplizieren wir die Gleichung mit a und mit b und bekommen die Gleichungen

$$ab + a + a = 0 \Rightarrow ab = 0$$
$$ab + ab + b = 0 \Rightarrow \quad b = 0.$$

Hieraus folgt direkt $b = 0$ und damit $a = 1$ durch Einsetzen. Das können wir durch das Einsetzen in dem ElimLin-Algorithmus durchführen, ohne eine kostspielige Zeilenstufenform zu berechnen.

Wenn wir in SL auch neue Monome zulassen, war das in unseren Experimenten nicht gut. Die Gleichungssysteme haben zwar mehr linear unabhängige Gleichungen erhalten, aber die Monomanzahl stieg, so dass unsere Gleichungssysteme nicht mehr dünn besetzt waren. Insgesamt haben wir viel mehr Monome als linear unabhängige Gleichungen erzeugt.

Wir nutzen SL in unserem Solver genau so, wie wir in Beispiel 6.1.2 beschrieben haben. Da schon ElimLin die Matrix auf Zeilenstufenform bringt, brauchen wir das nicht in SL zu tun. Auch den Rest von SL können wir deutlich schneller durchführen, wenn wir nur quadratische Polynome betrachten. Wir nutzen pro quadratischem Polynom nur die beiden Variablen aus dem Leitmonom $\mathrm{LM}(f)$. Weiter erzeugen wir damit nur maximal zwei neue Gleichungen. Also brauchen wir insgesamt nur $\mathcal{O}(m_q)$ Schritte, wobei m_q die Anzahl der quadratischen Polynome im System F ist.

Algorithmus 6.3 beschreibt den Aufbau unseres Solvers. Wir führen zuerst die Funktion `elimLin` aus Algorithmus 6.1 durch. Falls das Gleichungssystem nicht gelöst wurde, wenden wir die Funktion `sl` an. Diesen Algorithmus haben wir in Algorithmus 6.2 für beliebige polynomielle Gleichungssysteme kennen gelernt und danach für quadratische Systeme spezialisiert. Da die Funktion `sl` sehr restriktiv ist, werden wir damit wahrscheinlich nicht lösen, sondern nur neue Gleichungen dem System hinzufügen. Wenn wir keine neuen oder nur linear abhängige Gleichungen hinzufügen, brechen wir ab, weil wir das System nicht direkt lösen können. Das prüft die Funktion `test`. Ansonsten führen wir wieder die Funktion `elimLin` durch.

Wenn wir ElimLin in einzelne Teile für Zeilenstufenform (Zeile 6-7 in Algorithmus 6.1) und Einsetzen (Zeile 9-21) spalten, können wir uns die Funktion `test` sparen, da wir bei dem Berechnen der Zeilenstufenform bemerken, ob wir neue linear unabhängige Gleichungen in das Gleichungssystem eingefügt haben.

Der Algorithmus besteht aus den einzelnen Anwendungen von `elimLin` und `sl`. Die Laufzeit beider Funktionen wird dominiert von der Laufzeit von `echelonize`, also $\mathcal{O}(\mu \cdot m^2)$ mit der Anzahl der Gleichungen m und der Anzahl der Monome μ im Gleichungssystem. Hierbei können wir die Anzahl der Gleichungen und der Iterationen nicht wie bei ElimLin abschätzen, weil wir pro Schritt nicht mehr eine Variable eliminieren oder terminieren. Wir führen die Funktion `sl` aus, wenn wir keine Variablen eliminieren. Dadurch bricht der Algorithmus nicht ab. Unser Algorithmus bricht erst dann ab, wenn ElimLin keine Variablen eliminiert und SL keine linear unabhängigen Gleichungen findet.

Unser Solver ist gut geeignet, um die Gleichungssysteme aus Kapitel 3 zu bearbeiten. Allerdings müssen wir im ElimLin-Algorithmus 6.1 zwischen der Macaulay-Matrix und dem eigentlichen Gleichungssystem hin und her wechseln, wie es auch bei modernen Gröbnerbasen-Algorithmen der Fall und in [Alb10] beschrieben ist. Mit Hilfe einer speziellen Datenstruktur ist es möglich, die benötigten Matrixoperationen direkt auf den Polynomen durchzuführen. Die Datenstruktur und ElimLin wurden dabei von Christo-

Algorithmus 6.3 Ausführung der Algorithmen SL und ElimLin

```
function solve(F):
done ← False
while done == False do
  F̄ = elimLin(F)
  if F̄ is not solved then
    F = sl(F̄)
    if test(F̄, F) then
      done = True
    end if
  else
    done = True
  end if
end while
return F̄
```

pher Wolf in erster Version implementiert und von mir ausgetestet und korrigiert.

In Tabelle 6.1 stellen wir den Vorteil, den wir durch SL erhalten, dar. Dabei wenden wir ElimLin ohne und mit SL an, um die Gleichungssysteme von Varianten mit geringerer Rundenanzahl von Katan und Trivium aus Kapitel 3 zu lösen. Genaue Parameteranalysen für Katan und Trivium beschreiben wir in den Abschnitten 6.2 und 6.3. Hier führen wir für jede neue Instanz einer Chiffre eine Solver-Iteration aus. Auch nehmen wir $n_o = 6$ Ausgabebits für das Gleichungssystem zu Trivium mit $R = 500$ und $R = 525$ Runden. Die restlichen Parameter entnehmen wir aus den Abschnitten 6.2 und 6.3. Sollten wir das polynomielle Gleichungssystem nicht lösen können, so modellieren wir noch eine Instanz und fügen die entsprechenden Polynome dem bestehenden System hinzu bis wir lösen können. Pro Experiment haben wir 11 Tests gemacht. Dabei bezieht sich die Tabelle nur auf das Lösen von Gleichungssystemen für einen zufällig gewählten Schlüssel. Wir wollen hier noch keinen Angriff ausführen. In der Tabelle stehen jeweils die Durchschnittswerte der Ergebnisse. In diesem Fall verzichten wir auf Angabe der Varianz, weil sie nichts am Gesamtergebnis ändert. Die Zeit messen wir dabei in Minuten.

Wie wir in der Tabelle sehen, beschleunigt SL nicht nur ElimLin, sondern sorgt dafür, dass wir weniger Daten benötigen, um die Gleichungssysteme zu lösen. Für die Variante von Trivium mit $R = 525$ Runden liegen die Ergebnisse mit und ohne SL näher beieinander. Das liegt an der Länge der Ausgangsgleichungen, wenn wir SL starten. Je länger die Gleichung ist, desto unwahrscheinlicher ist es, dass wir durch SL neue Gleichungen erhalten. Bei den Gleichungssystemen zu Katan zeigt SL seine Stärke, weil wir nicht so viele Variablen einsetzen können. Dann bleiben die Gleichungen kurz und die Wahrscheinlichkeit, dass SL neue Gleichungen hinzufügt, steigt.

In SL verwenden wir im Gegensatz zu XSL nur noch im Gleichungssystem F enthaltende Monome und wählen den Parameter $D = \deg F$ entsprechend dem maximalen totalen Grad eines Polynoms im Gleichungssystem F. Das verstärkt die selektive Aus-

	mit SL		ohne SL	
Gleichungssystem	Instanzen	Zeit	Instanzen	Zeit
Katan (65)	$17,6$	$0,1$	27	$0,3$
Katan (70)	$57,3$	$7,1$	104	$40,4$
Trivium (500)	$58,4$	$1,9$	$108,4$	4
Trivium (525)	$48,1$	2	$52,8$	$2,5$

Tabelle 6.1: Lösen von ausgewählten polynomiellen Gleichungssystemen durch ElimLin mit und ohne SL

wahl von XSL in Bezug auf XL noch weiter. Weiter verwenden wir SL nicht mehr, um ein Gleichungssystem zu lösen. Im Gegensatz zum Algorithmus XSL ist SL kein Algorithmus zum Lösen eines Gleichungssystems. SL benutzen wir nur, um den Algorithmus ElimLin zu verstärken.

Beim Lösen unserer dünn besetzten Gleichungssysteme werden diese zunehmend dichter. Das können wir nicht verhindern, weil wir weniger Variablen beziehungsweise Monome haben, wenn wir uns der Lösung annähern. Wie oben beschrieben, arbeitet unser Solver, vor allem unser Algorithmus für die Linearisierung dünn besetzter Gleichungssysteme, nicht gut für dicht besetzte Gleichungssysteme. Hier benutzen wir das schnellste bekannte frei zugängliche Paket für lineare Algebra über $\mathbb{F}_2$ für dicht besetzte Systeme. Das ist die „Implementierung der „Methode der vier Russen“ (M4RI)“ aus [AAB+]. Die Idee hinter M4RI ist, die Matrix in kleinere quadratische $t \times t$ Matrizen für einen Parameter t zu unterteilen. Dann werden vorher gespeicherte Tabellen verwendet, um den dazugehörenden Teil der Zeilenstufenform schnell zu bestimmen.

Wir haben experimentell bestimmt, dass wir für Systeme mit weniger als 0.16% Koeffizienten, die verschieden von Null sind, unsere Strategie für dünn besetzte Systeme über $\mathbb{F}_2$ anwenden sollten und ansonsten M4RI.

Darüber hinaus benutzen wir in manchen Experimenten einen Algorithmus zum Raten von Variablen. In den meisten Angriffen auf Kryptosysteme werden Strategien vorgeschlagen, wie man einen Teil des Schlüssels oder Rundenschlüssels rät, um den vollen Schlüssel zu bestimmen. Das beschrieben wir auch in unserem ersten Fehler-Angriff in Abschnitt 5.3. Im Falle unseres Solvers ist das schwieriger. Wenn wir am Anfang eine nicht zu große Anzahl an Schlüsselvariablen raten, bringt uns das nicht viel. Gerade Trivium hat viele Runden, so dass der totale Grad des resultierenden Gleichungssystems dennoch groß wird und die geratenen Variablen dadurch nicht die gewünschte Wirkung erzielen, das System signifikant zu vereinfachen.

Bei Katan ist das System weit dichter und die übrigen Variablen werden jede Runde eingebracht. Durch die geratenen Schlüsselvariablen wird das resultierende Gleichungssystem dünn besetzter. Das ist gut für uns, liefert aber nicht so gute Ergebnisse wie folgende Heuristik.

Wir raten die Zwischenvariablen, die am häufigsten in einzelnen Gleichungen und

Algorithmus 6.4 Raten von Variablen im Solver

```
function guess(F):
for each f ∈ F do
  C ← hash
  G ← []
  for each ν ∈ ν(F) do
    C.append(ν : 0)
  end for
  C ← {a : 0}
  for each μ ∈ μ(F) do
    if deg(μ) == 2 then
      C[μ[0]] = C[μ[0]] + 1
      C[μ[1]] = C[μ[1]] + 1
    else if deg(μ) == 1 then
      C[μ[0]] = C[μ[0]] + 1
    end if
  end for
  c_ν = max{C[ν]|ν ∈ ν(f)}
  for each ν_C ∈ C.keys do
    if C[ν_C] == c_ν then
      G.append(ν_C)
    end if
  end for
end for
G = sortCount(G)
return G
```

im System vorkommen. In Algorithmus 6.4 stellen wir die Strategie zum Raten von Variablen dar. In dem Algorithmus initialisieren wir die Hash-Tabelle C, in der wir pro Polynom das Vorkommen der Variablen zählen. Hierbei ist $\mu[0]$ und $\mu[1]$ die erste beziehungsweise zweite Variable des Monoms μ. In der Liste G zählen wir die Variablen, die am häufigsten in den einzelnen Polynomen vorkommen. Dabei lassen wir mehrfaches Vorkommen von Variablen zu. Danach fasst die Funktion `sortCount` die Variablen zusammen und sortiert diese nach Anzahl des Vorkommens in G.

Im kommenden Abschnitt erklären wir, wie wir unseren Solver für die einzelnen Angriffe verwenden und wie wir die Zeit messen, so dass wir unsere Angriffe mit anderen vergleichen können.

6.1.2 Die Benutzung des Solvers

Zunächst versuchen wir mit unserem Solver dünn besetzte, quadratische, polynomielle Gleichungssysteme zu lösen. Das klingt zuerst nach einer Tautologie. Die Frage ist aber, in welcher Weise wir das zu lösende Gleichungssystem mit dem Solver bearbeiten. Mit

der Funktion `addEquations` unseres Solvers haben wir die Möglichkeit, ein Teilsystem zu berechnen, bis unser Solver das System nicht weiter vereinfachen kann. Danach können wir mehr Informationen beziehungsweise Gleichungen hinzufügen. Mit diesem iterativen Verfahren, das auch in modernen Gröbnerbasen-Algorithmen genutzt wird, können wir erst Teilsysteme reduzieren, bevor wir das gesamte System betrachten.

Von dieser Strategie machen wir ausgiebig Gebrauch. Wie schon in Kapitel 3 beschrieben, berechnen und reduzieren wir die Instanzen nacheinander. In den Frontends des Solvers ist das durch den Parameter `nKat` beziehungsweise `nTriv` realisiert. Dabei betrachten wir nicht jede Instanz einzeln, sondern eine bestimmte Anzahl gleichzeitig. Den richtigen Wert bestimmen wir durch Experimente. Wenn wir die Anzahl zu klein wählen, verlieren wir unnötig Zeit. Wählen wir den Parameter zu groß, können wir das System gar nicht mehr oder nur sehr langsam vereinfachen.

Bei Trivium werden wir darüber hinaus das Lösen des Gleichungssystems in zwei Phasen unterteilen. In der Offline-Phase reduzieren wir das System, bevor wir die Ausgabe der Triviuminstanzen bekommen. Das erlaubt mehr Zeit in das Auffinden ähnlicher Variablen zu investieren und mehr Instanzen zu betrachten. Auch bestimmen wir durch Algorithmus 6.4 vorher, welche Variablen wir raten. In der Online-Phase fügen wir die Ausgabebits und die geratenen Variablen hinzu. Danach führen wir eine Iteration unseres Solvers aus und lösen das System. Insgesamt haben wir für Trivium folgendes Vorgehen.

Offline:
1. Generieren des Systems für viele Triviuminstanzen.
2. Reduzieren des Systems mit Hilfe von ElimLin und SL.
3. Bestimmen der zu ratenden Variablen.

Online:
1. Raten der Werte.
2. Hinzufügen des reduzierten System, der Ausgabe für die Triviuminstanzen und Werte für die Variablen, die wir raten.
3. Lösen des Systems mit Hilfe von ElimLin und SL.

Das Frontend für unseren Angriff auf Trivium ist in ungefähr 2500 Zeilen Python beziehungsweise dem Computer-Algebra-System (CAS) SAGE geschrieben. SAGE selbst ist ein offen zugängliches CAS, das in [S+14] beschrieben wird.

Dieses Vorgehen liefert bei Katan keine guten Resultate. In der Offline-Phase reduzieren wir das Gleichungssystem nur geringfügig, so dass der Gewinn der Offline-Phase zu gering für den hierfür nötigen Aufwand ist. Tatsächlich benötigen wir mit der Trennung in Offline- und Online-Phase bei Katan mehr Instanzen und mehr Zeit in der Online-Phase, um den Schlüssel zu berechnen, weil wir das System in der Offline-Phase nicht in Bezug auf die Ausgabe hin reduzieren und uns Algorithmus 6.4 dadurch nicht die optimalen Variablen liefert, die wir raten sollten. Wir gehen darauf in Abschnitt 6.3 im Detail ein. Die Bestimmung der zu ratenden Variablen ist außerdem sehr ungenau. Es gibt einfach zu viele Variablen, die sofort eingesetzt werden, wenn wir die Ausgabebits hinzufügen. Also führen wir beim algebraischen Angriff auf Katan folgenden Algorithmus aus.

1. Generieren des Systems iterativ für eine vorher bestimmte Anzahl von Kataninstanzen mit dazugehöriger Ausgabe.
2. Reduzieren des Systems mit Hilfe von ElimLin und SL.
3. Bestimmen der zu ratenden Variablen und Raten dieser Werte.
4. Lösen des Systems mit Hilfe von ElimLin und SL.

Auch wenn dieses Vorgehen kürzer erscheint, hat das Frontend zu dem Katan-Angriff ungefähr 2300 Zeilen und ist ebenfalls in Python beziehungsweise SAGE geschrieben. Das liegt daran, dass wir wie bei Trivium alle Schritte des Angriffs durchführen müssen.

Beim algebraischen Fehler-Angriff auf Katan ist das System für unsere Verhältnisse klein und wir können es uns erlauben pro Fehler mehrere Iterationen unseres Solvers durchzuführen. Den genauen Ablauf beschreiben wir in Abschnitt 6.4.

Für unsere Experimente in Abschnitt 6.2 und 6.3 führten wir 101 Tests mit zufällig gewählten Schlüsseln und optimalen Parametern durch. Um die optimalen Parameter zu finden, analysieren wir die in Abschnitt 6.2 und 6.3 erklärten Parameter. In einer Parameteranalyse testen wir in einem ausgewählten Bereich zuerst die Grenzen und den mittleren Wert. Danach wählen wir den Bereich, der minimal nach Ressourcenverbrauch war. So grenzen wir den Bereich ein, bis wir den optimalen Wert finden. Die Ressourcen, die wir minimieren wollten, sind zur Lösung des polynomiellen Gleichungssystems gebrauchte Zeit und Daten. Nach Definition 2.1.19 haben wir einen validen Bruch, wenn wir mit einer Wahrscheinlichkeit größer 95% den Schlüssel finden und dabei weniger als $|\mathcal{K}|$ Chiffren-Berechnungen und Chiffren-Daten benötigen. Das heißt, wir müssen sowohl weniger als 2^{80} Berechnungen unserer Chiffren als auch weniger als 2^{80} Chiffren-Ausgabebits verwenden. Aus diesem Grund rechnen wir die gemessene Zeit mit der in Abschnitt 6.1.3 beschriebenen Methode in Chiffren-Berechnungen um. Damit wir besser mit dem Wert $|\mathcal{K}| = 2^{80}$ vergleichen können, bringen wir die Anzahl der benötigten Berechnungen und Ausgabebits in die Form 2^x für ein $x \in \mathbb{Q}$.

Für unsere Experimente benutzten wir einen Cluster aus AMD-Opteron-6276@2.3GHz Prozessoren mit 256 Knoten und 1 TB RAM. Nur wenige Schritte unseres Solvers sind parallelisierbar, so dass wir nur mit einem Knoten und maximal 256 GB RAM rechnen konnten. Des Weiteren führten wir die Online-Phase bei Trivium, den algebraischen Angriff auf Katan und den kompletten Fehler-Angriff auf einem Standardrechner mit 16 GB RAM durch.

6.1.3 Zeitmessung für den Solver

In allen algebraischen Angriffen ist Skalierbarkeit und der Vergleich zu anderen Angriffen beziehungsweise zu der Chiffre selbst eine große Frage. Das gilt vor allem dann, wenn wir Variablen raten.

Wir messen unseren Angriff in Sekunden, da die Zeit einfach messbar ist und sehr viele andere Angriffe auch dieses Maß verwenden. Andere Einheiten, wie ein Produkt aus Zeit und Fläche, Können und Ressourcen eines Angreifers, wie in [Ecr12] beschrieben,

oder das Produkt aus Zeit und Speicher werden bei den Angriffen, mit denen wir uns vergleichen wollen, nicht angewendet.

Um unsere Zeit nicht in Sekunden, sondern in Berechnungen der Chiffre zu messen, führen wir die Umrechnung

$$T_p := \frac{D((2^g - 1)T_f + T_s)}{R_C}$$

durch. Dabei ist D der Durchsatz der Chiffre in Ausgabebits pro Sekunde, R_C die Rundenanzahl, die wir für den Angriff benötigen plus die Anzahl der Berechnungen, die es braucht um im Durchschnitt zwei Ausgabebits zu produzieren. Wir müssen mit einer Chiffre durchschnittlich zwei Ausgabebits berechnen, um zu überprüfen, ob wir richtig geraten haben oder nicht. Im Falle von Trivium ist $R_C = R + 2$ und bei Katan ist $R_C = R$, weil wir 32 Ausgabebits nach R Runden zur Verfügung haben. Wir raten g Variablen. Wenn wir falsch raten, benötigen wir T_f Sekunden, um das herauszufinden. Es gibt insgesamt $2^g - 1$ Möglichkeiten falsch zu raten. Wenn wir richtig raten, benötigen wir T_s Sekunden für die Lösung des Systems. Damit brauchen wir maximal T_p Chiffre-Berechnungen für einen Angriff mit unserem Solver, wenn wir $2^g - 1$ mal falsch und einmal richtig raten. Genau betrachtet hat die Zeit T_p Ausgabebits als Einheit. Allerdings ist das auch die Zeit, die eine Chiffre braucht um eine Berechnung auszuführen. Also genau die Einheit, die wir haben wollen.

Wir benötigen statistisch betrachtet zwei Ausgabebits, weil wir mit dem ersten Ausgabebit im Durchschnitt die Hälfte aller Schlüssel als falsch bewerten können. Mit dem zweiten können wir die Hälfte aller übrigen eliminieren. Also haben wir nach zwei Ausgabebits eine Wahrscheinlichkeit größer als $1/2$, um einen falschen Schlüssel zu eliminieren.

Bevor wir genauer auf den Durchsatz D eingehen, merken wir an, dass T_f die Zeit ist, die möglichst gering sein muss, damit wir einen erfolgreichen Angriff erhalten. Wenn wir nicht raten, ist das unwichtig, aber wenn wir Variablen raten, ist es eine gute Eigenschaft, dass unser Solver falsche Lösungen schnell erkennt. In Algorithmus 6.1 ist dies durch die Abbruchbedingung in den Zeilen $10 - 13$ gegeben. Wir wissen, dass unsere Gleichungssysteme nur eine richtige Lösung haben, also darf es keine mehrfach bestimmten Variablen geben. Wir haben nur eine Lösung, weil unsere Gleichungssysteme von Kryptosystemen stammen. Ein solches Gleichungssystem ist immer voll durch den geheimen Schlüssel bestimmt. Dieser ist in einem Experiment immer zufällig, aber fest gewählt.

Um die Vergleichbarkeit zu der Chiffre selbst zu gewährleisten, messen wir nicht den Durchsatz unserer Chiffren, sondern nehmen die schnellste bekannte Implementierung. Für Katan liefert das nach [CDK] einen Durchsatz von $D_K = 1,0714 \times 10^9$ Bits pro Sekunde (bps) und für Trivium ist $D_T = 22,2996 \times 10^9$ bps nach [GB08].

In [BCJ+10] haben die Autoren leider vergessen einen solchen Vergleich durchzuführen. Das holen wir im folgenden Beispiel 6.1.3 nach.

Beispiel 6.1.3. *Die Autoren in [BCJ+10] benötigen für einen erfolgreichen Angriff* 15 *Minuten. Für unser Beispiel gehen wir davon aus, dass der dort benutzte Solver eine falsche Lösung in einem Viertel der Zeit identifiziert, also* 225 *Sekunden. Das ist großzügig bemessen, wenn man bedenkt, dass das normalerweise bei SAT-Solvern nicht*

so ist, da SAT-Solver selbst raten, um zur Lösung eines Gleichungssystems zu gelangen. Die Autoren selbst machen dazu keine Angaben, sagen aber, dass sie 45 *Variablen fixieren. Das bedeutet, sie engen den Schlüsselraum ein oder raten Variablen. Für dieses Beispiel nehmen wir an, sie raten die Variablen. Das liegt nahe, da die Autoren für ihren Angriff keinen schwachen Schlüsselraum ausgewiesen haben.*

Also benötigt der Angriff

$$\begin{aligned} T_p &= \tfrac{D_K}{79}(2^{45} \cdot 225 + 900s) \\ &= \tfrac{D_K}{79} 7,916 \cdot 10^{15} \\ &= 1,074 \cdot 10^{23} \\ &= 2^{76,5} \end{aligned}$$

Katan-Berechnungen. Das heißt, der Angriff in [BCJ+10] ist nicht praktikabel auf modernen Computern, da er zu viele Rateschritte enthält. Dennoch ist der Angriff erfolgreich nach Definition 2.1.19.

6.2 Algebraischer Angriff auf Trivium

In diesem Abschnitt führen wir unseren ersten algebraischen Angriff durch. Wir analysieren Trivium mit Hilfe unseres Solvers und dem Modell aus Abschnitt 3.1. Dazu gehen wir zuerst nochmals auf die Sättigung von Variablen und Monomen ein. Danach führen wir eine Parameteranalyse durch und schließen mit den Ergebnissen unseres Angriffs ab.

Die Sättigung der Variablen und Monome haben wir in Abschnitt 3.1 beschrieben. Unser Solver ist so gestaltet, dass wir direkt Gebrauch von ähnlichen Variablen machen. Wir stellen also dieselbe Sättigung fest, die wir in den Abbildungen 3.3–3.4 gesehen haben. Diese starke Sättigung bedeutet vor allem, dass wir mehr Gleichungen erhalten, die das System definieren, als Gleichungen, die neue Variablen einführen.

Unser Ziel ist es, die Rundenanzahl R zu maximieren, mit der wir die Triviuminstanzen initialisieren. Dazu maximieren wir in den jeweiligen Experimenten Geschwindigkeit und Datenverbrauch. Die Erfolgswahrscheinlichkeit unserer Angriffe war dabei stets $\mathcal{P}(\mathit{success}) = 1$. Weiter führten wir die Experimente ohne eine Aufteilung in Online- und Offline-Phase durch.

Dabei behandelten wir die Ordnung schon in Kapitel 3. Wir optimieren den benutzten Startwert. Den Angriff führen wir im „Gewählter IV“-Szenario durch. Aus diesem Grund können wir den Startwert beliebig wählen. Wir verwenden 2^i Triviuminstanzen. Dabei ist i die Anzahl an Bits im Startwert für Trivium, die wir variieren. Die restlichen Bits des Startwerts wählen wir zu Null. Wenn wir die Bits dabei zufällig wählen, ist der Angriff schlechter als bei einer ausgewählten Menge. Das haben wir schon bei den Versuchen zu Gröbnerbasen in Kapitel 3 gesehen. Dort benutzten wir zufällige Bits des IVs und konnten damit bis zu $R = 280$ Runden mit PolyBoRi brechen. Mit den Bits aus dem Cube-Angriff aus [DS09] war es möglich bis zu $R = 450$ Runden anzugreifen.

Mit dem Solver in Kapitel 6 erreichen wir Rundenanzahlen, bei denen wir nicht genügend Cubes in einer kleinen Überdeckungsmenge von Startwertbits (< 12 Bits)

abdecken können, ohne die Datenkomplexität des Cube-Angriffs zu erhalten. Also betrachten wir die Menge von Startwertbits aus einem anderen Blickwinkel.

Zufällige Startwerte führen zu weniger Struktur im System, weil die Aktualisierungsfunktion in frühen Runden „zufällige“ Monome generiert. Nehmen wir hingegen benachbarte Bits, so erzeugen wir mehr ähnliche Variablen, kommen früher in die oben beschriebene Sättigung und können das Gleichungssystem schneller lösen, wie in Tabelle 6.2 dargestellt ist. Der Parameter „Belegung“ kennzeichnet die benutzte Belegung des Startwerts. Ausgewählte Experimente stellen wir in der Tabelle 6.2 dar. Mit 0 bezeichnen wir die Variation der ersten i Bits. Bei den Belegungen $1, 2$ und 3 haben wir jedes dritte Bit variiert. Dabei fangen wir jeweils an der Stelle 0,1 oder 2 an. Für jedes Experiment führten wir 5 Testläufe mit zufälligen Schlüsseln durch. Die Tabelle zeigt die maximalen Werte. Dabei brachen wir ab, wenn die Tests länger als einen Tag oder mehr als 256 Triviuminstanzen benötigt haben. Für das Gleichungssystem mit $R = 500$ Runden nutzen wir 5 Ausgabebits pro Triviuminstanz. Für das System mit $R = 550$ erzeugten wir pro Triviuminstanz 10 Ausgabebits.

R	Belegung	Zeit in Sekunden	Anzahl Instanzen
500	0	110	66
500	1	120	103
500	2	140	140
500	3	155	145
550	0	ca. 3000	87
550	1	ca. 3600	123
550	2	ca. 5200	174
550	3	−−	−−

Tabelle 6.2: Zeit- und Datenkomplexität für unterschiedliche Mengen der Startwertbits

Also sind benachbarte Bits eine gute Wahl. Allerdings können wir nicht nur die ersten Bits des Startwerts wählen. Wenn wir dies nicht tun, lösen wir für $R = 550$ schon für das Startbit 14 innerhalb von 24 Stunden mit maximal 512 Triviuminstanzen nicht. Für $R = 600$ Runden lösen wir bereits für das Startbit 1 nicht mehr innerhalb von 24 Stunden. Wenn wir beim 0. Startbit anfangen, lösen wir das polynomielle Gleichungssystem, das Trivium mit $R = 600$ beschreibt, mit 2304 Instanzen in etwa 3 Stunden. Dabei generieren wir pro Instanz genau ein Ausgabebit. Der nächste Parameter, mit dem wir uns beschäftigen, ist die Anzahl der Ausgabebits pro Instanz.

In unserem Angriff wählen wir die ersten i Bits des öffentlichen Startwerts und setzen diese auf alle möglichen Vektoren aus $\mathbb{F}_2^i$, um unsere Triviuminstanzen zu erzeugen. Die restlichen $80-i$ IV Bits setzen wir auf Null. Wir erstellen das Gleichungssystem mit allen 2^i Instanzen in der Offline-Phase und erhalten damit ein gut konditioniertes System für

die Online-Phase. Dabei brauchen wir weniger als eine Woche für die Offline-Phase der hier vorgestellten Angriffe.

Die Ausgabe hilft uns ein überbestimmtes System zu erhalten und wir können bis zu 66 Ausgabebits pro Instanz erzeugen, ohne zusätzliche Variablen zu generieren, wie wir in Abschnitt 3.1 beschrieben haben. Allerdings erhalten wir durch die Ausgabe mehr quadratische Monome im Gleichungssystem. Wenn wir die Ausgabefunktion

$$Z_i := C_{i-65} + C_{i-110} + A_{i-65} + A_{i-92} + B_{i-68} + B_{i-83}$$

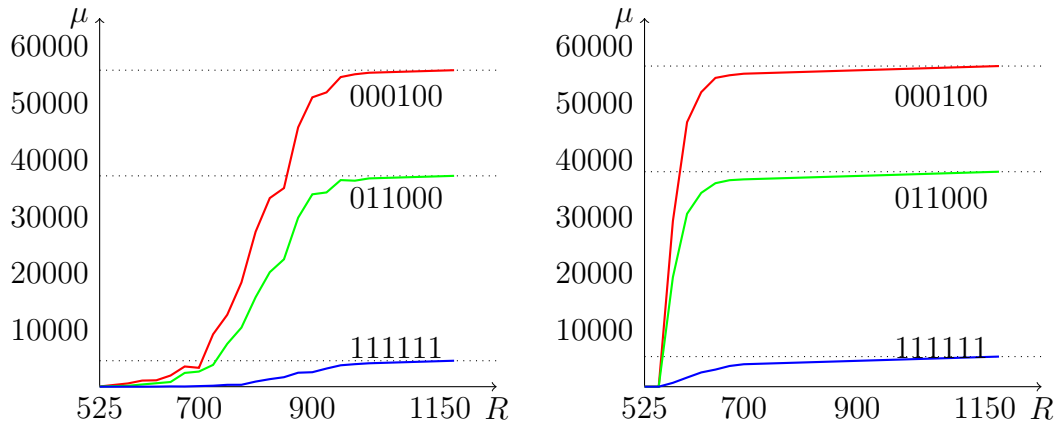
betrachten, sind dort sechs Bits vom internen Zustand aus unterschiedlichen Runden. Wenn wir diese einsetzen, erhalten wir fünf quadratische Monome mehr für jedes Auftreten der dazugehörigen Bits im internen Zustand. Das heißt, wir bekommen tatsächlich viel mehr Monome, wenn wir mehr Ausgabebits pro Instanz nehmen.

Abbildung 6.1 zeigt diesen Anstieg in der Anzahl der quadratischen Monome. Wir vergleichen darin die Anzahl der quadratischen Monome für verschiedene einzelne Instanzen und ansteigender Rundenanzahl. Die Instanzen stammen aus einem System mit insgesamt $T = 64$ Instanzen. Die Abbildungen wurden nach der Offline-Phase gemacht, so dass wir auch den Effekt unserer Sättigung nochmals sehen können. Im ersten Bild verwenden wir ein Ausgabebit $n_o = 1$ und im zweiten Bild $n_o = 66$ Ausgabebits. Auch wenn mehr Ausgabebits prinzipiell gut sind, zeigt sich hier, dass wir kein gut konditioniertes System für $n_o = 66$ Ausgabebits bekommen, weil wir zu viele quadratische Monome erhalten. Dieses Phänomen konnten wir schon bei $n_o = 2$ Ausgabebits ausmachen. Das System lässt sich schwerer lösen und hat fast mehr als doppelt so viele Monome als das System, das wir mit $n_o = 1$ erzeugen. Insgesamt verwenden wir in unserem Angriff also nur ein Ausgabebit pro Instanz.

In Abschnitt 3.2 beschreiben wir, dass weniger Monome bei Katan ein schlechtes Zeichen sind und hier schreiben wir, dass es bei Trivium ein gutes Zeichen ist. Das ist kein Widerspruch, weil die Umstände andere sind. Bei Katan entwickeln wir die Ausgangsgleichungen nicht genug, um Informationen über den Schlüssel zu erhalten. Bei Trivium mit vielen Ausgabebits erhalten wir viele Monome, weil wir diese jeweils einzeln auf die Anfangsgleichungen zurückführen müssen. Dabei sind die entstehenden Teile des Gleichungssystems weitgehend unabhängig, so dass wir jedes Teil einzeln lösen müssen.

Wenn wir $n_o = 66$ Ausgabebits pro Instanz benutzen, ist die Anzahl der quadratischen Monome im Gessamtsystem bei $R = 700$ Runden unwesentlich kleiner als die Anzahl der Monome, die wir für das volle Gleichungssystem mit $R = 1152$ Runden Triviums brauchen. Für $n_o = 1$ Ausgabebit tritt dieser Effekt für $R = 925$ Runden auch auf. Da unser Solver auf Linearisierung des Gleichungssystems basiert, stellen wir folgende Vermutung an:

Vermutung 6.2.1. *Die Komplexität unseres Angriffs auf Trivium wächst für $R > 925$ nur unwesentlich an, wenn wir ein Ausgabebit $n_o = 1$ verwenden. Für $n_o = 66$ wächst die Komplexität unseres Angriffs für $R > 700$ nicht wesentlich an. Wenn wir also $R = 925$ Runden mit einem Ausgabebit pro Instanz beziehungsweise $R = 700$ mit $n_o = 66$ Ausgabebits brechen können, sollten wir auch in der Lage sein, volles Trivium zu brechen.*



(a) Runden R gegen Anzahl der quadratischen Monome μ für $n_o = 1$

(b) Rundenanzahl R gegen Anzahl der quadratischen Monome μ für $n_o = 66$

Abbildung 6.1: Anzahl der Monome im gesamten Gleichungssystem für $n_o = 1$ und $n_o = 66$; Die Beschriftung kennzeichnet den signifikanten Teil des IV in binärer Schreibweise

Die Begründung unser Aussage ist einfach. Wenn sich die Anzahl der nichtlinearen Monome nicht wesentlich erhöht, ist es auch nicht wesentlich schwerer, das System zu brechen, weil wir dieselben Monome in den Systemen haben. Wir können die Monome der übrigen Runden einfach durch die Monome der ersten 700 beziehungsweise 925 Runden ausdrücken. Leider können wir dies nicht durch praktische Tests bestätigen, weil wir kein geeignetes System für Runde 700 oder 925 aufstellen können.

Um überflüssige Monome in unseren Gleichungssystemen zu sparen, ändern wir Algorithmus 3.1 noch ein wenig. Anstatt alle Instanzen vom Schlüssel ausgehend zu entwickeln, generieren wir von der Ausgabe ausgehend alle Bits des internen Zustands, die wir tatsächlich brauchen. Wenn wir ab einer Runde r alle Bits der drei Register brauchen, entwickeln wir diese vom Schlüssel ausgehend. Insgesamt gehen wir wie folgt vor.

1. Wir bestimmen eine Anzahl von Ausgabebits pro Instanz, die wir generieren wollen. Im Folgenden wählen wir $n_o = 1$.

2. Ausgehend von den Ausgabegleichungen berechnen wir die Einträge des internen Zustands, die wir benötigen, um die Ausgabe zu bestimmen, bis wir alle Einträge benötigen.

3. Ausgehend von der Initialisierung der Register entwickeln wir die restlichen Runden und verbinden die beiden Teilsysteme.

Danach gehen wir wie oben beschrieben vor. Wir übergeben das System, ohne die Ausgabe zu setzen, dem Solver und berechnen ähnliche Variablen, reduzieren das System entsprechend und berechnen die Variablen, die wir raten wollen. Daraufhin gehen wir in die Online-Phase.

In unseren Experimenten war es uns möglich, eine auf $R = 625$ Runden reduzierte Variante von Trivium zu brechen. In der Tabelle 6.3 sehen wir die Ergebnisse unserer

Angriffe. Die Anzahl der Monome und Variablen entspricht der Anzahl vor der Online-Phase unseres Angriffs. Sobald wir die Ausgabebits setzen, eliminiert unser Solver alle Variablen und Monome.

R	μ	ν	Anzahl geratener Variablen	Benötigte Daten	Laufzeit
625	**499741**	**15869**	23	2^{11}	$2^{59,7}$
625	1135858	32518	**0**	$\mathbf{2 \cdot 2^{11}}$	$\mathbf{2^{42,2}}$

Tabelle 6.3: Ergebnisse für den Angriff auf $R = 625$ Runden Trivium

Im ersten Angriff raten wir 23 Variablen. Wenn wir falsch raten, braucht der Solver durchschnittlich $2^{11,6}$ Sekunden, um zu erkennen, dass das System inkonsistent ist. Wenn wir richtig raten, lösen wir das System in durchschnittlich $2^{13,8}$ Sekunden. Diese Werte haben wir als Mittelwert in jeweils 101 Experimenten ermittelt. Wenn wir falsch raten, erkennen wir das in $2^{10,7}$ bis $2^{13,5}$ Sekunden. Dabei benötigen wir in 95% aller Experimente weniger als $2^{12,8}$ Sekunden, um falsch geratenen Werte zu finden. Die Varianz unserer Testläufe lag bei 1435 Sekunden. Wenn wir richtig raten, haben wir analoge Werte. Insgesamt liefert uns das nach Abschnitt 6.1.3 eine Laufzeit von $2^{59,7}$ Trivium-Berechnungen für unseren Angriff. Das kann leider auf modernen Computern nicht in praktikabler Laufzeit berechnet werden. Dazu raten wir immer noch zu viele Variablen. Allerdings ist der Angriff nach Definition 2.1.19 valide.

Ein weiteres Problem, wenn wir raten, ist der Zeitpunkt. Wir raten, nachdem wir das System aufgebaut haben. Schlau wäre es zu raten, bevor wir das Gleichungssystem aufbauen. Dann reduzieren wir sofort dementsprechend und unser System ist nicht so groß. Leider können wir das in der Praxis nicht tun, da wir im Vorhinein nicht wissen, welche Variablen wir raten sollen.

Im zweiten Angriff raten wir nicht. Dafür benötigen wir mehr Daten. Im Vergleich zu den Angriffen aus Abschnitt 3.1 haben wir dennoch die kleinste Datenkomplexität. Wir benötigen 2^{12} Ausgabebits, um $R = 625$ Runden von Trivium zu brechen gegen 2^{31} bis $2^{78,5}$ für die beschriebenen Angriffe. Dafür sinkt die Laufzeit unseres Angriffs signifikant, da wir nicht mehr raten.

Dennoch ist es schwer, ein volles symbolisches Gleichungssystem für 2^{12} Triviuminstanzen zu erstellen, weil wir zu viele Ressourcen in der Offline-Phase benötigen. Dafür stellen wir zwei Gleichungssysteme für jeweils 2^{11} Triviuminstanzen her. Die zwei Systeme haben keine Verbindung zueinander, so dass sie nicht von den ähnlichen Variablen aus dem anderen System profitieren. Aus diesem Grund haben wir mehr als doppelt so viele Monome und Variablen im System und brauchen damit länger, um das System zu lösen. Auch bei diesem Angriff haben wir 101 Testläufe durchgeführt. Im Durchschnitt benötigten wir dabei $2^{17,1}$ Sekunden. Das führt zu einer Laufzeit von $2^{42,2}$ Trivium-Berechnungen.

Wir konnten die benötigte Anzahl von Triviuminstanzen nicht bestimmen, die für einen Angriff auf eine auf R Runden reduzierte Variante Triviums benötigt wird. Das Problem ist, dass wir keine Formel angeben können, wann die Sättigung der Monome

Autor	Methode	R	Zeit	Daten
[FV13]	Cube-KR	799	2^{62}	2^{40}
[Rad06]	Algebraisch	–	–	–
hier	Algebraisch	625	$2^{42,2}$	2^{12}

Tabelle 6.4: Angriffe auf Trivium mit R Runden, Zeit- und Datenkomplexität

und Variablen einsetzt. Danach müssten wir bestimmen, wann wir ein überbestimmtes Gleichungssystem haben. Das garantiert aber noch kein lösbares System.

Wenn wir einmal ein System mit genügend Instanzen generiert haben, können wir es auch lösen. Das wirkliche Problem bei unserem Angriff auf Trivium ist die Generierung eines Systems in der Offline-Phase. Wie wir gesehen haben, beheben wir dieses Problem teilweise, indem wir Teilsysteme erstellen, die wir erst in der Online-Phase zusammenfügen. Das verschiebt unser Problem in die Online-Phase, in der wir nur begrenzte Zeit haben, ähnliche Variablen zu finden, die Systeme zu reduzieren und das System zu lösen. In Abbildung 6.1 sehen wir einen exponentiellen Anstieg der Monome. Das ist auch die Beschränkung unseres Angriffs. Es ist wahrscheinlich möglich, bis zu $R = 700$ Runden Triviums mit unserem Angriff zu brechen, aber danach benötigen wir zu viele Instanzen, um ein gut konditioniertes System zu generieren. Ohne weitere Techniken oder neue Ideen ist das nicht möglich.

An dieser Stelle könnten wir die Laufzeit, die wir für einen Angriff auf die volle Chiffre oder Varianten Triviums mit mehr Runden benötigen könnten, extrapolieren. Dies tun wir absichtlich nicht. In der algebraischen Kryptoanalyse wurde dies zum Beispiel in [CB07, CSSV12, CP02, CKPS00a, Cou02] getan. Leider wurde immer wieder festgestellt, das solche Extrapolationen falsch sind.

In Tabelle 6.4 vergleichen wir das Ergebnis unseres Angriffs mit relevanten Angriffen aus der Literatur. Im Bezug auf Runden R der angegriffenen Variante Triviums findet sich in [FV13] ein besserer Angriff. Allerdings ist der hier vorgestellte Angriff praktikabel in Zeit- und Datenkomplexität und der beste algebraische Angriff, die bis jetzt keinen Erfolg auf Trivium gehabt haben.

6.3 Algebraischer Angriff auf Katan

Als nächstes stellen wir den algebraischen Angriff auf Katan vor. Die Gleichungssysteme, die wir durch die Modellierung von Katan aus Abschnitt 3.2 erhalten, sind weit dichter als die Systeme, die wir bei Trivium kennen gelernt haben. Wie schon erwähnt, stellt das unseren Solver vor große Probleme. Unser Angriff bricht $R = 80$ Runden Katans. Bei diesem Angriff raten wir 40 Bits und benötigen $2^{72,3}$ Katan-Berechnungen und 2^7 Katan-Ausgaben. Eine Katan-Ausgabe enthält 32 Bits, so dass wir insgesamt 2^{12} Ausgabebits benötigen.

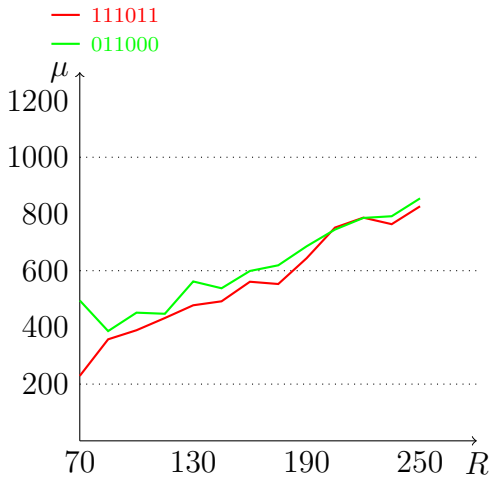


Abbildung 6.2: Runden R gegen Anzahl der quadratischen Monome μ für ein Gleichungssystem für $T = 32$ Kataninstanzen

Bereits das reduzierte Gleichungssystem für $R = 60$ Runden hat nach 4 Kataninstanzen 423 Zeilen, 820 Spalten und 6182 nichtverschwindende Einträge. Das heißt, es ist in unserem Sinne nicht mehr dünn besetzt ($1,7\% > 0,16\%$). Das liegt an den 32 Bit kleinen internen Zustand von Katan im Vergleich mit dem 288 Bit großen internen Zustand Triviums. Außerdem hat die Aktualisierungsfunktion des Registers B zwei quadratische Monome.

In unserer Parameteranalyse zu Katan betrachteten wir, analog zu der Analyse der Parameter von Trivium, zunächst die Bits des Klartextes, die wir für unseren Angriff in den unterschiedlichen Kataninstanzen variieren. Dabei führen wir unsere Experimente mit jeweils 10 Tests mit zufälligen Schlüsseln durch. In der Parameteranalyse sagen wir, ein Test schlug fehl, wenn wir mehr als 1 Stunde oder mehr als 256 Instanzen benötigten. In den durchgeführten Experimenten war die Erfolgswahrscheinlichkeit der Angriffe stets Eins, weil wir keine Abbruchbedingungen gesetzt haben.

Wie bei Trivium ist es keine gute Idee zufällige Bits des Klartexts zu wählen. Auch nicht benachbarte Bits sind nicht geeignet. Mit zufälligen Bits benötigen wir zu lange, um das polynomielle Gleichungssystem für $R = 60$ Runden zu lösen.

Insgesamt stellten wir fest, dass wir für ein gutes Resultat die letzten Bits des Klartextes wählen sollten. Dieselben Bits verwendeten die Autoren von [BCJ+10]. Deren Ergebnisse bestätigten wir in unseren Experimenten.

Danach erhalten wir leider nur sehr spärliche Informationen aus der Chiffre. Die Abbildung 6.2 der Monomanzahl über die Runden der Chiffre liefert leider nicht viele Informationen. Im Vergleich zu Abbildung 6.1 weist die Abbildung darauf hin, dass unserer Angriff nicht mehr als $R > 80$ Runden brechen kann.

Darüber hinaus ist die Trennung in zwei Phasen kontraproduktiv. Wir generieren in einer Offline-Phase 128 Instanzen, die mit $R = 80$ Runden Katans arbeiten. Wir raten darüber hinaus 40 Zwischenvariablen und führen 101 Experimente mit zufällig gewählten Schlüsseln aus. Dann benötigen wir etwa 25 Minuten für einen Durchlauf des Solvers

in der Online-Phase, wenn wir richtig raten. Dabei erhalten wir lediglich 20 Schlüsselvariablen, lösen das System also nicht vollständig. Dies führt zu einem erfolgreichen Angriff, jedoch können wir einen besseren Angriff ausführen.

Wenn wir dasselbe Experiment wieder 101 mal mit zufällig gewählten Schlüsseln in einer Phase durchführen, erhalten wir den vollen Schlüssel durchschnittlich in ungefähr 8 Minuten, wenn wir richtig raten. Das liegt am Rundenschlüssel. Da der Schlüssel in jeder Runde benutzt wird, hat er sehr viel größeren Einfluss auf das System. Unser Algorithmus zum Raten liefert in der Offline-Phase keine guten Ergebnisse, weil wir das System in der Offline-Phase nicht komplett reduzieren. Wenn wir die Ausgabe danach in der Online-Phase hinzufügen, sortieren wir das gesamte System um. Das ist der zeitintensive Teil in der Online-Phase. Also erstellen wir das System in unserem Angriff immer, wenn wir die Ausgabe hinzufügen.

Wenn wir richtig raten, benötigen wir durchschnittlich 645 Sekunden über 101 Experimente. Wenn wir falsch raten, erkennen wir das im Schnitt in 561 Sekunden. Wir benötigen also nach Abschnitt 6.1.3 insgesamt $2^{72,8}$ Berechnungsschritte. Die Zeitdifferenz in den Experimenten, in denen wir richtig und falsch raten, ist klein im Vergleich zu den Experimenten aus Abschnitt 6.2. Dies liegt daran, dass wir am Ende des Angriffs raten. Wir modellieren zuerst alle notwendigen Kataninstanzen und generieren das Gleichungssystem, bevor wir die mit Hilfe von Algorithmus 6.4 festgelegten Variablen raten. Das heißt, der Hauptteil des Angriffs wird durchlaufen, bevor wir Variablen raten.

In Tabelle 6.5 zeigen wir zwei Angriffe auf Katan. Im ersten Angriff raten wir keine Variablen und benötigen nur 80 Instanzen um eine Lösung zu erhalten. Wollen wir nur 10 Runden mehr angreifen, müssen wir schon 40 Variablen raten, benötigen 128 Instanzen und ungleich mehr Zeit.

In der Parameteranalyse zu SL aus Abschnitt 6.1.1 brauchen wir zur Lösung des Gleichungssystems für Katan mit $R = 70$ Runden nur etwa 57 Instanzen. Dort rechnen wir mit `nKat` $= 1$. Wenn wir diesen Parameter hoch setzen, um Zeit zu sparen, benötigen wir mehr Instanzen, um den Schlüssel zu finden. Es handelt sich also um einen Zeit-Daten-Austausch.

R	Anzahl geratener Variablen	Benötigte Instanzen	Laufzeit
70	**0**	**80**	$\mathbf{2^{29,8}}$
80	40	128	$2^{72,8}$

Tabelle 6.5: Ergebnisse für den algebraischen Angriff auf Katan

Für $R = 80$ Runden hat das System maximal ungefähr 170000 Monome. Für 81 Runden haben wir bereits über 900000 Monome und können das System auch mit 256 Instanzen nicht lösen. Das entspricht 2^{13} Ausgabebits.

Dieser Angriff ließe sich verbessern, wenn wir davor raten könnten. Leider kennen wir dazu keine (gute) Möglichkeit.

Auch hier erhalten wir leider nicht den besten Angriff. In dem verwendeten Szenario gibt es den Angriff aus [AL12], der $R = 115$ Runden in praktikabler Zeit- und

Autor	Methode	R	Zeit	Speicher	Daten
[AL12]	Differential	115	2^{32}	–	2^{32}
[BCJ+10]	Algebraisch	79	15 min + 2^{45}	–	2^{5}
hier	Algebraisch	80	2^{72}	–	2^{7}

Tabelle 6.6: Angriffe auf Katan mit R Runden, Zeit- und Datenkomplexität

Datenkomplexität zu brechen vermag. Jedoch ist unser Angriff auf Katan der beste algebraische Angriff. Wie wir in Abschnitt 6.1.3 gesehen haben, benötigt der Angriff in [BCJ+10] mindestens $2^{76,5}$ Katan-Berechnungen, um $R = 79$ Runden zu brechen. Unser Angriff bricht $R = 80$ Runden und benötigt dabei weniger Zeit. Die Ergebnisse sind in Tabelle 6.6 zusammengefasst.

6.4 Algebraische Fehler-Analyse von Katan

Unser letzter Angriff ist die algebraische Erweiterung des Fehler-Angriffs auf Katan aus Abschnitt 5.3. Diese Erweiterung nutzt die Fähigkeit unseres Solvers, schnell Fehler zu erkennen, wie wir in Abschnitt 6.2 schon gesehen haben. Auch bei Katan wäre das so, wenn wir nicht das Gleichungssystem jedes mal aufstellen müssten, bevor wir raten. Damit ist es uns möglich, den vollen Schlüssel Katans mit durchschnittlich $7,5$ Fehlern in $2^{27,1}$ Schritten zu berechnen. Wenn wir darüber hinaus 38 Variablen raten, können wir die Anzahl der benötigten Fehler im Mittel auf $4,85$ Fehler reduzieren. Dann brauchen wir aber $2^{64,8}$ Katan-Berechnungen, um unseren Angriff durchzuführen.

Für unseren Fehler-Angriff modellieren wir Katan, wie in Abschnitt 3.2 beschrieben, als dünn besetztes quadratisches Gleichungssystem. Da wir die Fehler-Runde R^* kennen, in der wir ein Bit flippen, können wir auch die Kataninstanzen mit Fehler wie dort beschrieben modellieren, wenn wir eine Charakteristik haben. Diese gibt uns den Fehler, so dass wir einfach in der Fehler-Runde den Fehlervektor an der richtigen Stelle addieren können.

Um eine Charakteristik für eine bestimmte Fehler-Runde R^* zu berechnen, injizieren wir alle möglichen Fehlervektoren in R^*. Danach berechnen wir die Differenz der Ausgaben $\Delta = Z + Z'$ der Instanzen mit und ohne Fehler. Wir reduzieren diese Differenz Δ so weit wie möglich und lesen die Charakteristik ab. Die Charakteristik für die Fehler-Runden $R^* = 242$ ist in Anhang A dargestellt. Insgesamt können wir eine Charakteristik für Runden $R^* \geqslant 241$ berechnen.

Mit Fehlern in den Runden $R^* \geqslant 241$ berechnen wir kein einziges Schlüsselbit. Das quadratische System ist für unseren Solver nicht lösbar. In unseren Experimenten benötigten wir mindestens einen Fehler in Runde $R^* = 198$, um den Schlüssel zu berechnen. Dort haben wir keine Charakteristik. Wie wir aber schon in Abschnitt 6.2 gesehen haben, findet Solver schnell Inkonsistenzen. Also bestimmen wir die Fehlerposition

durch Fehler in den Runden $R^* \geqslant 241$ und testen dann alle möglichen Fehlervektoren. Davon gibt es $2^3 - 1 = 7$, wenn wir den Fehlervektor Null nicht mitzählen. Die Wahrscheinlichkeit, dass ein falscher Fehlervektor nicht erkannt wird kennzeichnen wir mit $\mathcal{P}(\vec{e_w}\ \textit{wird angenommen})$. Tabelle 6.7 zeigt die Wahrscheinlichkeit $\mathcal{P}(\vec{e_w}\ \textit{wird angenommen})$ für mehrere Runden R^*. Dabei initialisieren wir das Gleichungssystem der vorherigen dort angegebenen Runden mit den richtigen Fehlern und übergeben danach einen falschen zufälligen Fehlervektor. Die folgende Tabelle basiert auf 1001 Experimenten mit zufällig gewählten Schlüsseln.

R_w^*	230	222	214	206	198	190
$\mathcal{P}(\vec{e_w}\ \textit{wird angenommen})$	0,01	0,01	0,01	0,03	0,01	0,02

Tabelle 6.7: Wahrscheinlichkeit, dass ein zufälliger Fehlervektor angenommen wird

In der Tabelle 6.7 sehen wir, dass unser Solver Fehler sicher erkennt. Das liegt auch daran, dass wir nur sehr wenige Instanzen im Vergleich zu den anderen Modellen haben. Injizieren wir 6 Fehler, so haben wir 7 Instanzen von Katan. Für diese Instanzen ist der Anfang für $r < R^*$ gleich.

In Algorithmus 6.5 beschreiben wir unsere Fehler-Analyse im Detail. Die Funktion `faultSolve` deklariert die maximal erlaubte Anzahl an Fehlern ε_{max}, alle möglichen Testvektoren T und den Speicher E für alle Daten, die wir aus den Fehlern erhalten. Die Funktion `faultPosition` ermittelt die Position des Fehlers in Runde 241 wie in Abschnitt 5.3 beschrieben. Danach ist alles fertig, um die Rekursion `recFaultSolve` zu starten. Der Algorithmus bricht ab, wenn wir die vorher festgelegte zulässige Anzahl an Fehlern überschreiten oder wenn wir den vollen Schlüssel finden. Sobald die Position geprüft ist, ändern wir die Fehler-Runde und erzeugen einen Fehler. Die Funktion `faultGenerator` prüft dabei, ob wir schon einen Fehler in die aktuelle Runde injiziert haben. Wenn ja, erhöhen wir die Anzahl der gemachten Fehler ε nicht und geben den alten Fehler zurück. Falls nicht, erzeugen wir einen neuen Fehler und setzen $\varepsilon = \varepsilon + 1$. Der Speicher ist nötig, wenn wir ein falsches System als richtig bewertet haben. Dann werden wir im nächsten Schritt, spätestens aber beim Wechsel der Runde auf $R^* = 245$ feststellen, dass wir einen falschen Fehlervektor als richtig bewertet haben, weil wir in Runde $R^* = 245$ wieder eine Charakteristik zur Verfügung haben. Warum wir gerade zu dieser Runde gehen, klären wir in der folgenden Parameteranalyse. Wir testen jeden Fehlervektor. Wenn unser System unlösbar ist, wissen wir, dass das System inkonsistent ist. Wenn wir G lösen können, sind wir fertig. Wenn beides nicht der Fall ist, ist das System mit der in Tabelle 6.7 beschrieben Wahrscheinlichkeit richtig und wir injizieren den nächsten Fehler. Falls die `for`-Schleife bis zum Ende läuft, wissen wir, das uns ein Fehler in der Vergangenheit unterlaufen ist, so dass wir zum letzten Fehler zurückkehren. Wenn die Rekursion ohne Lösung durchlaufen wird, haben wir Schlüsselvariablen falsch geraten.

Der Algorithmus terminiert spätestens, wenn wir alle 6 falschen Möglichkeiten $\varepsilon_{max} - 3$ mal durchgegangen sind. Genauer können wir nur 6 mal in einem Durchgang falsch

Algorithmus 6.5 Algebraische Fehler-Analyse von Katan

```
function recFaultSolve(F, R*, e):
if F solved ∨ε > ε_max then
  return F
end if
if R* > 190 then
  R* = R* − 8
else
  R* = 245
end if
ε, E = faultGenerator(E)
for each ē ∈ T do
  G ← F
  G.append(generateSystem(ē))
  G =solve(G)
  if G unsolvable then
    continue
  else if G solved then
    return G
  else
    G = recFaultSolve(G, R*, e)
    if G solved then
      return G
    end if
  end if
end for
if R* == 245 then
  R* = 190
else
  R* = R* + 8
end if
return F

function faultSolve():
global ε_max = 15
global T ← F_2^3
global ε = 2, 33
E = ∅
F, e ← faultPosition(R* = 241)
global R* = 231
return recFaultSolve(F, R*, e)
```

liegen. Für jedes dieser 6 Blätter eines Baumes können wir 7 mal eines der jetzt immer falschen Blätter wählen. Sollten wir jedes Mal falsch liegen, so schaltet der Algorithmus

bei $\varepsilon = \varepsilon_{max}$ ab. Damit können wir nur $\varepsilon_{max} - 4$ mal in ein falsches Gleichungssystem einsteigen, da wir beim ersten Aufruf der Funktion `recFaultSolve` schon $\varepsilon = 3$ haben. Ansonsten gehen wir nach einem vollen Durchlauf der `for`-Schleife wieder zurück und durchlaufen die nächste (falsche) Schleife. Insgesamt können wir maximal $6 \cdot 7^{\varepsilon_{max}-4}$ falsche Durchläufe machen. Mit den von uns gewählten Parametern sind das $2^{33,5}$ Solver-Iterationen.

Eine bessere Abschätzung der Solver-Iterationen liefert das folgende Lemma.

Lemma 6.4.1. *Wir benötigen im Durchschnitt*

$$2,33 + t + 42 \cdot \sum_{i=1}^{t} \mathcal{P}_i(\vec{e_w} \text{ wird angenommen})$$

Solver-Iterationen für einen erfolgreichen Angriff auf Katan.

Insbesondere benötigen wir durchschnittlich $12,11$ *Solver-Iterationen, falls wir Fehler maximal bis Fehler-Runde* $R^* = 190$ *injizieren und* $R_\Delta = 8, R_0^* = 246$ *wählen.*

Beweis: Die Baumstruktur unseres Problems ist in Abbildung 6.3 abgebildet. Sei t die Tiefe des Baums. Wir benötigen im Schnitt $2,33$ Fehler, um die Fehler-Position zu bestimmen. Für die Fehler in Runden $R^* \geqslant 190$ wählen wir mit einer Wahrscheinlichkeit von $\mathcal{P}_i(\vec{e_w} \text{ wird angenommen})$ für den i-ten Fehler das falsche Blatt. Insgesamt gibt es 6 falsche Blätter und ein richtiges Blatt pro Fehler. Wir müssen alle t richtigen Blätter durchlaufen, um erfolgreich zu sein. Wenn wir ein falsches Blatt beim $i-$ten Fehler wählen, werden wir mit vernachlässigbar kleiner Wahrscheinlichkeit ($< 2/1001$) nach 7 Solver-Iterationen merken, dass wir ein falsches Blatt ausgewählt haben. Das macht insgesamt

$$2,33 + t + 6 \cdot \sum_{i=1}^{t} \mathcal{P}_i(\vec{e_w} \text{ wird angenommen}) \cdot 7$$

Solver-Iterationen für einen erfolgreichen Angriff.

Setzen wir die Werte aus Tabelle 6.7 ein, folgt die Anzahl der Solver-Iterationen direkt. □

Den Sprung auf $R^* = 245$ behandeln wir einfach als weiteren Fehler. Dabei ist die Wahrscheinlichkeit Null, dass wir ein falsches Blatt wählen. Danach wählen wir wieder $R_\Delta = 8$ als Rundendelta. In Tabelle 6.7 stellen wir die Wahrscheinlichkeit für den Zweig ab $R^* = 245$ nicht dar, weil die Werte sehr ähnlich sind.

In unseren Experimenten für den Fehler-Angriff auf Katan übernehmen wir alle nötigen Parameter des algebraischen Angriffs auf Katan. Die neuen Parameter sind das Rundendelta R_Δ und die anfängliche Fehler-Runde R^*. Wir messen auch nicht mehr Zeit und Daten, sondern Zeit und Anzahl der benötigten Fehler ε. Die Anzahl der benötigten Fehler begrenzen wir auf 20, so dass wir nach 20 Fehlern einen Angriff als nicht erfolgreich bewerten. Auch die Zeit, die für unsere Angriffe sehr klein ist, beschränken wir auf 5 Minuten. Das mag nach sehr wenig klingen, soll jedoch nur vermeiden, dass wir alle Blätter des Baumes durchprobieren. Wir optimieren die Anzahl der benötigten Fehler ε.

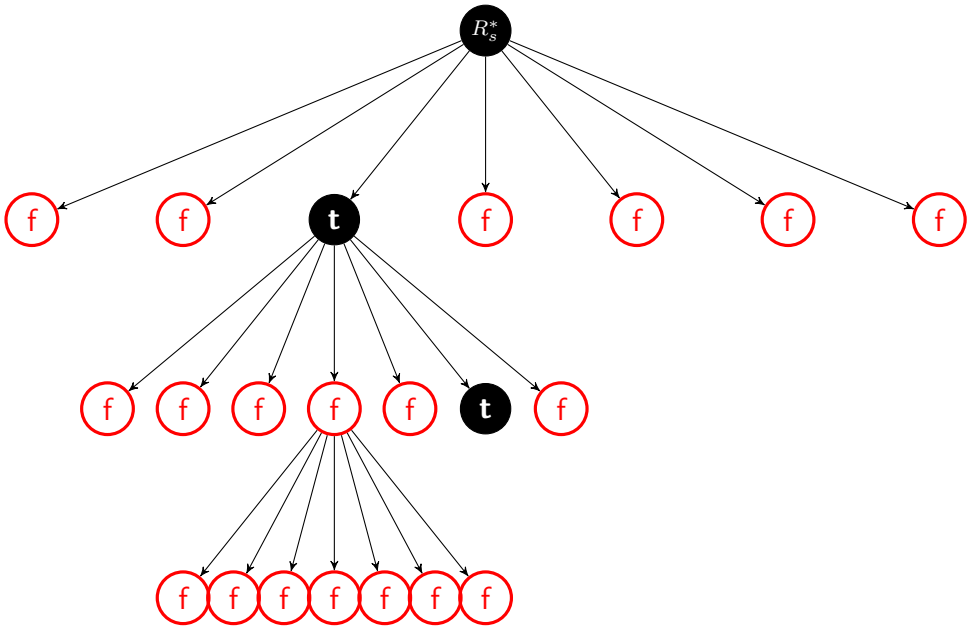


Abbildung 6.3: Baumstruktur des Fehler-Angriffs auf Katan, f kennzeichnet einen falschen und t den richtigen Fehlervektor

Der Bereich für die anfängliche Fehler-Runde R^* ist $R^* = 241$ bis $R^* = 250$. Dabei halten wir das Rundendelta $R_\Delta = 8$ fest, was sich nachher als sehr gut herausstellen soll. Für $R^* = 250$ brauchen wir im Durchschnitt $\varepsilon = 8,73$ Fehler bei einer Varianz von $\sigma = 0,9$. Das verbessert sich für $R^* = 241$ zu $\varepsilon = 7,53 \pm 0,5$ Fehlern. Interessant an diesen ersten Experimenten ist, dass das System für Fehler in Fehler-Runden $R^* < 190$ nicht mehr lösbar ist. Um dies zu umgehen, fangen wir wieder bei hohen Fehler-Runden an. Genauer setzen wir die zweite Startrunde $R_s^* = R_0^* + 4$ fest. Das wirkt sich vor allem auf die Varianz aus. Auch wenn wir die Wahrscheinlichkeit für einen erfolgreichen Angriff erhöhen, haben wir für $R^* = 246$ eine Varianz von $\sigma = 1,9$ durch den Bedarf eines zweiten Durchgangs. Die Erfolgswahrscheinlichkeit unseres Angriffs war immer größer als 99/101.

Für $R^* = 241$ erhalten wir das beste Ergebnis, so dass wir die anfängliche Fehler-Runde für die Analyse des Parameters R_Δ auf diesen Wert festhalten.

Für R_Δ prüfen wir in dem Bereich 4 bis 16. Die Experimente zu $R_\Delta = 16$ ergaben eine Erfolgswahrscheinlichkeit von $84/101 \approx 83\%$ bei durchschnittlich benötigten $8,93 \pm 2,9$ Fehlern. Das war das schlechteste Ergebnis verglichen zu den anderen Ergebnissen aus Tabelle 6.8.

R_Δ	4	5	6	7	**8**	**9**	10
$\varepsilon \pm \sigma$	$13,39 \pm 0,83$	$11,61 \pm 0,73$	$10,33 \pm 0,62$	$9,26 \pm 0,57$	$\mathbf{8,57 \pm 0,81}$	$\mathbf{8,12 \pm 1,04}$	$8,46 \pm 2,12$
$\mathcal{P}(success)$	100%	100%	100%	100%	**100%**	**96%**	83%

Tabelle 6.8: Ergebnisse der Parameteranalyse für R_Δ

Wir betrachten für $R_\Delta = 8$ und $R_\Delta = 9$ noch einen Parameter.

Für $R_\Delta > 8$ sinkt die Erfolgswahrscheinlichkeit. Dass $R_\Delta = 8, 9$ für das Rundendelta am geeignetsten sind, sehen wir auch an der Aktualisierungsfunktion von Katan

$$A_i = B_{i-18} + B_{i-7} + B_{i-12} \cdot B_{i-10} + B_{i-8} \cdot B_{i-3} + K_{2R+1}$$
$$B_i = A_{i-12} + A_{i-7} + A_{i-8} \cdot A_{i-5} + A_{i-3} \cdot f(r) + K_{2R}\,.$$

Grundsätzlich wollen wir so wenig Fehler wie möglich einsetzen. Wenn wir allerdings $R_\Delta > 12$ wählen, überspringen wir eine Aktualisierung pro Bit im Register B. Damit sind die erzeugten Gleichungssysteme für die jeweiligen Fehler unabhängig. Also ziehen wir keine Informationen mehr aus den Fehlern untereinander, sollte der Fehler zu einem Teil in A liegen. Der Solver findet auch keine Fehler im Gleichungssystem mehr. Dieses Phänomen beginnt schon bei $R_\Delta > 9$, so dass wir mit $R_\Delta = 8$ oder $R_\Delta = 9$ in dem Teil ohne Charakteristik rechnen. Bei $R_\Delta = 8$ oder $R_\Delta = 9$ überspringen wir die beiden quadratischen Monome $A_{i-8} \cdot A_{i-5}$ und $B_{i-8} \cdot B_{i-3}$. Dadurch haben wir nur noch ein quadratisches Monom, um die Sicherheit der Chiffre zu gewährleisten. Das ist sehr wenig.

In unseren bisherigen Experimenten benötigten wir stets Fehler in den Runden $R^* < 200$, um das System zu lösen. Leider können wir dort nicht direkt den Fehler injizieren, da uns keine Charakteristik für diese Runden zur Verfügung steht. Also wollen wir so wenig Fehler wie möglich in hohe Runden injizieren. Nach den durchschnittlich zwei Fehlern zur Bestimmung der Fehler-Position erhöhen wir das Rundendelta R_Δ für einen Fehler noch einmal. Wir testen den Bereich $R^1_\Delta = 8$ bis $R^1_\Delta = 24$. Für $R_\Delta = 8$ erhalten wir mit $R^1_\Delta = 24$ eine Erfolgswahrscheinlichkeit für den Angriff von nur 71%. Wie in Abschnitt 2.1.2 beschrieben, sagen wir, ein Angriff ist erfolgreich, wenn die Erfolgswahrscheinlichkeit über 95% liegt. Die Tabelle 6.9 liefert die Ergebnisse für die Werte um $R^1_\Delta = 16$.

R^1_Δ	14	15	16	17	**18**	19
$\varepsilon \pm \sigma$	$7,86 \pm 1,24$	$7,68 \pm 0,48$	$7,61 \pm 1,02$	$7,4 \pm 0,44$	$\mathbf{7,23 \pm 0,54}$	$7,79 \pm 2,37$
$\mathcal{P}(_{success})$	97%	99%	97%	95%	**97**%	94%

Tabelle 6.9: Ergebnisse der Parameteranalyse für R^1_Δ

Für $R_\Delta = 9$ steigt die Varianz der Ergebnisse stark an und die Erfolgswahrscheinlichkeit sinkt. Immer wenn die Varianz ansteigt, lösen wir oft nicht innerhalb der angegebenen Fehleranzahl, sondern fangen wieder „oben“ bei dem vorher beschriebenen Versatz an. Wenn wir das tun, injizieren wir immer so viele Fehler, bis wir wieder in kleinen Fehler-Runden $R^* < 200$ angelangt sind. Je höher also die Varianz, desto häufiger passiert dies.

Für die so gefundenen Parameter $R^* = 241, R_\Delta = 8$ und $R^1_\Delta = 18$ haben wir 1001 Experimente durchgeführt und die Ergebnisse sind in Tabelle 6.10 dargestellt. Dabei vergleichen wir die Ergebnisse unseres algebraischen Fehler-Angriffs mit unserem statistischen Fehler-Angriff. Wie wir in Abschnitt 5.3 geschrieben haben, benötigt dieser $4,33$ bis $7,33$ Fehler und $2^{29.04}$ bis 2^{74} Katan-Berechnungen.

Angriff	ε	Zeit
[ALRSS12]	115	2^{59}
[SH13]	140	–
Position bekannt	$7,33$	2^{74}
Position + theoretische Güte	$\mathbf{4,33}$	$2^{29.04}$
algebraisch	$7,5 \pm 1,47$	$\mathbf{2^{27,1}}$
algebraisch + Raten	$4,85 \pm 0,6$	$2^{64,8}$

Tabelle 6.10: Ergebnisse für den Fehler-Angriff auf Katan

Wir benötigen im Mittel $\varepsilon = 7,5$ Fehler mit einer Varianz $\sigma = 1,47$ und führen dabei durchschnittlich $2^{27,1}$ Katan-Berechnungen durch. Die Varianz von $1,47$ und das Mittel von $7,5$ sagen uns, dass wir selten den Sprung zu Fehler-Runde $R^* = 245$ durchführen. Insgesamt springen wir 15 mal in unseren Experimenten. Die Erfolgswahrscheinlichkeit unseres Angriffs ist $17/1001 \approx 98\%$. Also ist der Angriff erfolgreich.

Falls wir weniger injizierte Fehler für die Schlüsselsuche verwenden müssen, können wir Schlüsselvariablen raten. Dabei fanden wir in unseren Experimenten 38 geeignete Variablen. Diese tauchen in frühen Schritten des Angriffs, das heißt in den Gleichungssystemen zu hohen Fehler-Runden, in kurzen, quadratischen Gleichungen auf. Die Variablen finden wir durch die in Abschnitt 6.1.2 beschriebene Heuristik angewendet auf die kurzen Gleichungen im System, in denen nur Schlüsselvariablen vorkommen.

Damit können wir die Anzahl der benötigten Fehler auf durchschnittlich $\varepsilon = 4,85$ mit einer Varianz von $0,6$ reduzieren, wenn wir diese Variablen raten. Wir identifizieren mit Hilfe des Solvers falsche Fehlervektoren. Damit ist der Solver allerdings nicht mehr gut geeignet, geratene Variablen zu evaluieren. Wir identifizieren falsch geratene Variablen in durchschnittlich $28,1$ Sekunden. Allerdings brauchen wir nur 6 Sekunden, um das System zu lösen, wenn wir richtig geraten haben. Wir merken erst, dass wir falsch geraten haben, wenn wir die ersten 7 Blätter alle als falsch evaluiert haben.

Unser algebraischer Angriff ohne Raten ist besonders praktikabel und hat nicht die statistische Ungenauigkeit des statistischen Angriffs auf Katan. Dafür ist unsere statistische Fehler-Analyse auch in dem alten Fehler-Modell anwendbar. Beide Angriffe im „Fehler“-Szenario sind, wie bereits in Kapitel 5 erwähnt, eine signifikante Verbesserung der bisherigen Resultate.

Kapitel 7

Zusammenfassung und Fazit

In dieser Arbeit modellieren wir die Chiffren Trivium und Katan als dünn besetzte, quadratische, polynomielle Gleichungssysteme. Unsere anfängliche Modellierungstechnik von Trivium orientiert sich dabei am Cube-Angriff. In einem Cube-Angriff bilden wir die Summe von vielen Instanzen einer Chiffre, um Informationen über den Schlüssel zu erhalten. Die Schwäche von früheren Modellen algebraischer Angriffe auf Trivium ist die Unfähigkeit mehrere Instanzen zu benutzen. Um diese Schwäche zu beseitigen, führen wir viele Zwischenvariablen ein. Das macht unser Modell sehr schwer zu handhaben. Also verwenden wir lineare Beziehungen zwischen diesen Instanzen, sogenannte ähnliche Variablen, um so wenig Zwischenvariable und Monome wie möglich in den Gleichungssystemen zu verwenden. Unsere Techniken lösen eine Sättigung in der Anzahl der Zwischenvariablen und Monome aus, die es uns erlaubt, Instanzen samt der Ausgabe einer Chiffre zu generieren, ohne dafür Zwischenvariablen einzuführen.

Auch mit ähnlichen Variablen enthalten die Gleichungssysteme bis zu einer Million Monomen und 30000 Variablen. Solche Gleichungssysteme können auf Grund des hohen Speicherverbrauchs nicht von Gröbnerbasen-Algorithmen gelöst werden. Dafür erweitern wir den bekannten ElimLin-Algorithmus mit einer Monomordnung und SL, einer neuen Variante von XL für dünn besetzte Systeme. Danach haben wir versucht, die aus unseren Modellen resultierenden Gleichungssysteme mit diesem Solver zu lösen.

Wir greifen die beide Chiffren in dem „Gewählter Klartext“-Szenario an. Dabei erzielen wir jeweils den besten algebraischen Angriff. Statistische Angriffe vermögen jedoch mehr Runden der einzelnen Chiffren anzugreifen, sind bei Trivium jedoch nicht praktikabel und benötigen bei beiden betrachteten Chiffren viele Daten, um einen Angriff durchzuführen, wie in Tabelle 7.1 zu sehen ist, haben unsere Angriffe eine sehr niedrige Datenkomplexität.

Unser Modell ist sehr flexibel, so dass wir es schnell für verschiedene Angriffsmodelle und Chiffren anpassen können. Als Beispiel hierfür führen wir noch einen Angriff auf Katan im „Fehler“-Szenario durch. Als Vergleich entwickeln wir zunächst einen statistischen Angriff. Unsere erste Verbesserung ist das Modell, das für einen Angriff im „Fehler“-Szenario benutzt wird. Anstatt einen Fehler in genau einem Eintrag des internen Zustands zu injizieren, verwenden wir drei benachbarte Einträge, die verändert werden könnten. Für diese Modelle entwerfen wir einen statistischen Filter für falsch geratene Schlüsselbits. Wir rechnen die fehlerfreie und fehlerbehaftete Ausgabe mit einem

Chiffre	Szenario	Angriff	Runden	Daten bzw. Fehler	Zeit	Herkunft
Trivium	Gewählter IV	Cube + Raten (62)	799/1152	2^{40}	2^{62}	[FV13]
Trivium	Gewählter IV	algebraisch + Raten (23)	625/1152	2^{11}	$2^{59.7}$	hier
Trivium	Gewählter IV	algebraisch	625/1152	2^{12}	$2^{42.2}$	hier
Katan32	Gewählter Klartext	differentiell	115/254	2^{32}	2^{32}	[AL12]
Katan32	Gewählter Klartext	algebraisch	70/254	2^{7}	$2^{29.8}$	hier
Katan32	Gewählter Klartext	algebraisch + Raten (40)	80/254	2^{7}	$2^{72.8}$	hier
Katan32	Fehler	Cube	254	115	2^{59}	[ALRSS12]
Katan32	Fehler	Cube + SAT	254	140	-	[SH13]
Katan32	Fehler	Filter	254	$7,33$	2^{74}	hier
Katan32	Fehler	Filter + theoretische Güte	254	$4,33$	$2^{29.04}$	hier
Katan32	Fehler	algebraisch	254	$7,5 \pm 1,47$	$2^{27,1}$	hier
Katan32	Fehler	algebraisch + Raten (38)	254	$4,85 \pm 0,6$	$2^{64,8}$	hier

Tabelle 7.1: Experimentelle Ergebnisse dieser Arbeit im Vergleich zu ausgewählten anderen Angriffen

geratenen Schlüssel zurück und überprüfen, ob ein gültiger Fehler den Unterschied der Ausgabe hervorgerufen hat. Leider können wir die genaue Anzahl an benötigten Fehlern und benötigter Zeit nicht bestimmen. Dafür geben wir eine untere und obere Schranke dieser Zahlen an.

Mit Hilfe des algebraischen Modells von Katan und dem entwickelten Solver beseitigen wir die statistischen Ungenauigkeiten und machen den Angriff praktikabel. Dabei verwenden wir den Solver nicht nur, um ein Gleichungssystem zu lösen, sondern auch, um das richtige Gleichungssystem zu finden. Beide Angriffe sind viel besser als die bisherigen Analysen im „Fehler“-Szenario.

Tabelle 7.1 zeigt alle experimentellen Ergebnisse dieser Arbeit und aussagekräftige frühere Angriffe. Jedes Mal, wenn wir Schlüsselvariablen geraten haben, steht die Anzahl der geratenen Variablen in Klammern dahinter. Eine Laufzeit von „-“ bedeutet, dass der Angriff in vernachlässigbarer Zeit auf einem Standard-Computer ausgeführt werden kann.

Fazit

Insgesamt zeigt diese Arbeit, dass algebraische Kryptoanalyse sehr mächtig ist, wenn man die Chiffren angemessen modelliert und die benutzten Algorithmen zur Erstellung der algebraischen Repräsentation einer Chiffre und zum Lösen der entstehenden polynomiellen Gleichungssysteme aufeinander abstimmt. Gerade am Beispiel Triviums ist dies zu sehen, da vorherige rein algebraische Angriffe mit Standardtechniken wie Gröbnerbasen in [SFP08] keinen Erfolg hatten.

Die einzige wirkliche Grenze, die wir an dem entwickelten Solver feststellen konnten, ist die Dichte des polynomiellen Gleichungssystems. Der Solver ist für dünn besetzte Gleichungssysteme entwickelt. Auch wenn wir die schnellste, frei zugängliche Software benutzen, um die zu dem dicht besetzten System gehörende Matrix auf Zeilenstufenform zu bringen, braucht es zu viel Zeit, eine solche zu erzeugen.

Dabei ist es von entscheidender Bedeutung, wie wir die algebraische Repräsentation und den Solver einsetzen. Das sehen wir gut am Beispiel von Katan. Während der algebraische Angriff im „Gewählter Klartext"-Szenario nur mäßigen Erfolg hatte, konnten wir einen großen Erfolg im „Fehler"-Szenario verbuchen, indem wir nicht versucht haben, direkt ein System zu lösen, sondern unseren Solver benutzt haben, um das richtige Gleichungssystem zu finden.

Die betrachteten Chiffren haben wir in der Arbeit nur als Beispiel verwendet. Die Techniken sind auch auf andere Kryptosysteme übertragbar.

Eine spannende Richtung für weitere Forschung wäre, weiter mit polynomiellen Gleichungssystemen zu arbeiten, die nur mit einer gewissen Wahrscheinlichkeit erfüllt sind.

Ferner ist es vorstellbar, SL zu erweitern, so dass wir in Zwischenrechnungen Monome höheren Grades zulassen, die sich gegenseitig auslöschen. Dazu könnten wieder Techniken der linearen Algebra angewendet werden. Genauer könnten wir wie bei SL alle Gleichungen des polynomiellen Gleichungssystems F mit allen Variablen aus F multiplizieren. So erhalten wir das neue Gleichungssystem $\overline{F}$. Danach stellen wir die Macaulay-Matrix von $\overline{F}$ auf und bringen die Matrix auf Zeilenstufenform. Alle linearen und quadratischen Gleichungen der Zeilenstufenform, die nur Monome aus F verwenden, fügen wir zu F hinzu. So erhalten wir potenziell mehr Gleichungen in F. Dies würde aber signifikant mehr Zeit als SL benötigen.

Eine weitere Richtung für zukünftige Forschung ist die Weiterentwicklung von ElimLin beziehungsweise eine Anpassung von Gröbnerbasen für dünn besetzte Gleichungssysteme. Dazu könnten wir zum Beispiel nur die S-Polynome erzeugen, die keine neuen Monome erzeugen.

Glossar

$\mathbf{F_4}$: Gröbnerbasis-Algorithmus auf Basis linearer Algebra von Faugère 24

AND: Logischer Operator für ein *und* 39

Angriffsmethode: Die Methode/Das Ziel unseres Angriffs 20

Angriffszenario: Rahmenbedingungen für einen Angriff 20

Blockchiffre: Eine Klasse von Chiffren mit fester Länge der Ein- und Ausgabe 14

Blocklänge: Die Länge der Ein- und Ausgabe einer Blockchiffre 14

Charakteristik: Eine Tabelle zum Auffinden der Position eines Fehlers 74

Chiffre: Algorithmus zur Ver- und Entschlüsselung einer Nachricht 10

Cube: Eine Menge von IV Variablen, die eine Zerlegung des Polynoms einer Chiffre erlaubt 63

DFA: Ein Angriff, in dem wiederholt ein Fehler in den internen Zustand der Chiffre injiziert wird, um Informationen über den Schlüssel zu erhalten 73

Distinguisher: Algorithmus zum Unterscheiden der Chiffre vom Zufall 21

ElimLin: Algorithmus zum Lösen von Gleichungssystem mit Hilfe von Eliminierung linearer Variablen im Gleichungssystem 86

Fehler: Ein Fehler im internen Zustand einer Chiffre. Dieser besteht aus Fehlerposition und Fehlervektor 73

Fehlerposition: Zufällige, unbekannte Stelle, an der ein Fehlervektor oder ein Fehlerbit addiert beziehungsweise injiziert wird 78

Fehlervektor: Zufälliger, unbekannter Vektor aus $\mathbb{F}_2^3$, der zusammen mit der Fehlerposition einen Fehler beschreibt 78

Frontend: Bedienungsoberfläche für den Solver 97

Geheimtext: Verschlüsselte Nachricht; Ausgabe eines Kryptosystems 10

Geheimtextblock: Verschlüsselte Nachricht einer festgesetzten Länge; Ausgabe einer Blockchiffre 14

IV: Öffentlicher Startwert einer Chiffre, um die Ausgabe bei vielen, mit dem gleichen Schlüssel verschlüsselten Nachrichten zu verschleiern 13

Katan: Eine hardware-orientierte Blockchiffre, die mit Hilfe von zwei nichtlinearen Schieberegistern funktioniert 51

Klartext: Zu verschlüsselnde Nachricht; Eingabe eines Kryptosystems 10

Klartextblock: Zu verschlüsselnde Nachricht einer festen Länge; Eingabe einer Blockchiffre 14

Kryptosystem: Ein Algorithmus zum Ver- und Entschlüsseln von Nachrichten 10

M4RI: Schnellste bekannte, öffentlich zugängliche Implementierung, um eine dicht besetzte Matrix über $\mathbb{F}_2$ auf Zeilenstufenform zu bringen 95

Macaulay-Matrix: Matrix-Darstellung eines polynomiellen, nicht-linearen Gleichungssystems 34

Meet-in-the-Middle: Angriff, indem vom Klartext und Geheimtext ausgehend passende Schlüssel einer Chiffre berechnet werden 53

Monomordnung: Eine Ordnungsrelation für Monome in einem Polynom 25

Orakel: Ein Algorithmus, der bestimmte formalisierte Hinweise in einem Sicherheitsspiel liefert 19

Ordnung: Wenn nicht anders beschrieben, ist eine Monomordnung gemeint 25

RFID: Ein System zum Identifizieren und Lokalisieren mittels Radiowellen 73

Schieberegister: Ein rückgekoppeltes logisches Schaltwerk mit konstantem Speicher, mit dem wir Chiffren konstruieren 18

Schlüssel: Element des Schlüsselraums einer Chiffre, mit der eine Chiffre eine Nachricht verschlüsselt; geheime Eingabe einer Chiffre 10

Schlüsselraum: Menge aller Schlüssel einer Chiffre 10

Schlüsselstrom: Ausgabe einer Stromchiffre, mit der ein Klartext beliebiger Länge verschlüsselt werden kann 16

Sicherheitsspiel: Beschreibung von Angriffsszenarien und -modelle 19

SL: Komponente des vorgestellten Solvers; Verbesserung zu ElimLin, mit der rein quadratische Systeme mittels Multiplikation von Variablen gelöst werden; Variante von XL 91

Solver: Algorithmus zum Lösen eines polynomiellen, multivariaten Gleichungssystems über einem ausgewählten Körper 86

Stromchiffre: Eine Klasse von Chiffren, die einen Schlüsselstrom produziert, um eine Nachricht großer Länge zu verschlüsseln 16

Superpoly: Bei der Zerlegung des Polynoms für eine Chiffre durch einen Cube erzeugtes Polynom 63

Trivium: Eine hardware-orientierte Stromchiffre, die einen Schlüsselstrom mit Hilfe von drei nichtlinearen Schieberegistern erzeugt 39

XL-Algorithmus: Algorithmus zum Lösen eines quadratischen, polynomiellen Gleichungssystems durch Multiplizieren der Polynome mit allen Monomen bis zu einem bestimmten Grad 37

XOR: Logischer Operator für ein *ausschließendes oder* 39

Abkürzungsverzeichnis

ANF	Algebraische Normalform
bps	Bits pro Sekunde
CAS	Computer-Algebra-System
CNF	Konjunktive Normalform
$\deg(f)$	Totaler Grad des Polynoms f
DFA	Differentieller Fehler-Angriff
ElimLin	Elimination Linear
HW	Hamming-Gewicht
IV	Startwert - Initial Value
KR	Schlüsselsuche - Key Recovery
M4RI	Implementierung der „Methode der vier Russen“
RFID	Ein System zum Identifizieren
RK	Verwandte Schlüssel-Angriff - Related Key

SL	Sparse Linearization
WK	Schwache Schlüssel - Weak Key
XL	Erweiterte Linearisierung
XSL	Erweiterte Linearisierung für dünn besetzte Systeme

Symbolverzeichnis

A	Schieberegister
B	Schieberegister
D	Durchsatz einer Chiffre in Bits pro Sekunde
K	Menge aller geheimen Schlüsselvariablen
R	Anzahl der Initialisierungsrunden
R^*	Runde, in der ein Fehler injiziert wird
T_f	Benötigte Zeit eines fehlgeschlagenen Angriffs in Sekunden
T_p	Benötigte Zeit eines Angriffs in Berechnungen der Chiffren
T_s	Benötigte Zeit eines erfolgreichen Angriffs in Sekunden
V	Menge aller öffentlichen IV Variablen
Δ	Differenz der Ausgabe einer Instanz mit und ohne Fehler
$\mathbb{F}_2$	Körper mit 2 Elementen
$\mathcal{K}$	Schlüsselraum einer Chiffre
μ	Anzahl der Monome in einem System
$\mu(F)$	Menge aller Monome im Gleichungssystem F
ν	Anzahl der Variablen in einem System
$\nu(F)$	Menge aller Variablen im Gleichungssystem F
σ	Güte eines Filters
ε	Gesamtanzahl der benötigten Fehler
$\vec{e}$	Fehlervektor eines Fehlers
a_i	Zwischenvariable für den internen Zustand einer Chiffre in Runde i
b_i	Zwischenvariable für den internen Zustand einer Chiffre in Runde i
c_i	Zwischenvariable für den internen Zustand einer Chiffre in Runde i

e	Position des Fehlers im internen Zustand der Chiffre
n_o	Anzahl der Ausgaberunden von Trivium
$p(K)$	Superpoly eines Cubes
`dec`	Abbildung zum Verschlüsseln
`enc`	Abbildung zum Entschlüsseln

Literaturverzeichnis

[AAB+] Tim Abbott, Martin Albrecht, Gregory Bard, Marco Bodrato, Michael Brickenstein, Alexander Dreyer, Jean-Guillaume Dumas, William Hart, David Harvey, Jerry James, David Kirkby, Clément Pernet, Wael Said, and Carlo Wood. M4RI(e)—Linear Algebra over F_2 (and F_2^e). `http://m4ri.sagemath.org/`.

[ADMS09] Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In *FSE 2009, Fast Software Encryption*, pages 1–22. Springer, 2009.

[AFI+04] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and Gröbner Basis Algorithms. In *ASIACRYPT 2004, LECTURE*, pages 338–353. Springer-Verlag, 2004.

[AL12] Martin R. Albrecht and Gregor Leander. An All-In-One Approach to Differential Cryptanalysis for Small Block Ciphers. In *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, pages 1–15. Springer, 2012.

[Alb10] Martin Albrecht. *Algorithmic Algebraic Techniques and their Application to Block Cipher Cryptanalysis.* PhD thesis, Royal Holloway, University of London, 2010.

[ALRSS12] Shekh Faisal Abdul-Latip, Reza Reyhanitabar, Willy Susilo, and Jennifer Seberry. Fault Analysis of the KATAN Family of Block Ciphers. In *ISPEC*, pages 319–336. Springer, 2012.

[Arm06] Frederik Armknecht. *Algebraic Attacks on Certain Stream Ciphers.* PhD thesis, University of Mannheim, 2006.

[BCJ+10] Gregory V. Bard, Nicolas Courtois, Jorge Nakahara Jr., Pouyan Sepehrdad, and Bingsheng Zhang. Algebraic, AIDA/Cube and Side Channel Analysis of KATAN Family of Block Ciphers. In *INDOCRYPT*, pages 176–196. Springer, 2010.

[BD09] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Groebner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, pages 1326 – 1345, 2009. Effective Methods in Algebraic Geometry.

[BDL97] Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology – Eurocrypt '97*, pages 37–51. Springer-Verlag, 1997.

[BR10] Andrey Bogdanov and Christian Rechberger. A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN. In *Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers*, pages 229–240. Springer, 2010.

[BS91] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '90, pages 2–21, London, UK, UK, 1991. Springer-Verlag.

[BS93] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, London, UK, UK, 1993.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 513–525, London, UK, UK, 1997. Springer-Verlag.

[Buc79] Bruno Buchberger. A Criterion for Detecting Unnecessary Reductions in the Construction of Groebner Bases. In *Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, EUROSAM '79, pages 3–21, London, UK, UK, 1979. Springer-Verlag.

[Buc85] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. *Recent trends in multidimensional system theory*, 1985.

[Buc03] Johannes Buchmann. *Einführung in die Kryptographie.* Springer, Berlin, 2003.

[Bun] Bundesamt für Sicherheit und Informationstechnik. Minimum Requirements for Evaluating Side-Channel Attack Resistance of Elleptic Curve Implementations. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_ECCGuide_e_pdf.pdf?__blob=publicationFile`. Accessed: 30.09.2014.

[Cab10] Daniel Cabarcas. An Implentation of Faugère's F4 Algorithm for Computing Gröbner Bases. Master's thesis, University of Cincinnati, 2010.

[CB07] Nicolas T. Courtois and Gregory V. Bard. Algebraic Cryptanalysis of the Data Encryption Standard. In *Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169. Springer Berlin Heidelberg, 2007.

[CDK] Christophe Cannière, Orr Dunkelman, and Miroslav Knežević. The KATAN/KTANTAN Family of Block Ciphers. `http://www.cs.technion.ac.il/~orrd/KATAN/index.html`. Accessed: 18.09.2014.

[CDK09] Christophe Cannière, Orr Dunkelman, and Miroslav Knežević. KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 272–288, Berlin, Heidelberg, 2009. Springer-Verlag.

[CKM97] S. Collart, M. Kalkbrener, and D. Mall. Converting bases with the Gröbner walk. *Journal of Symbolic Computation*, pages 465–469, 1997.

[CKPS00a] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer Berlin Heidelberg, 2000.

[CKPS00b] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Bart Preneel, editor, Springer, 2000. Extended Version: `http://www.minrank.org/xlfull.pdf`.

[CLO07] David A. Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Cou02] Nicolas Courtois. Higher Order Correlation Attacks, XL Algorithm and Cryptanalysis of Toyocrypt. In *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, pages 182–199, 2002.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of equations. In *ASIACRYPT*, pages 267–287. Springer, 2002.

[CP08] C. De Cannière and B. Prenel. Trivium. In *New Stream Cipher Designs*, LNCS, pages 84–97. Springer, 2008.

[CSSV12] Nicolas T. Courtois, Pouyan Sepehrdad, Petr Susil, and Serge Vaudenay. Elimlin Algorithm Revisited. In *Fast Software Encryption—FSE 2012*, pages 306–325. Springer, 2012.

[DAS11] Itai Dinur and Adi Adi Shamir. Breaking Grain-128 with Dynamic Cube Attacks. In *18th International Workshop Fast Software Encryption, FSE 2011*, pages 167–187. Springer, 2011.

[Die04] Claus Diem. The XL-Algorithm and a Conjecture from Commutative Algebra. In *ASIACRYPT*, Lecture Notes in Computer Science, pages 323–337. Springer, 2004.

[DS09] I. Dinur and A. Shamir. Cube Attacks on tweakable black box polynomials. In *EUROCRYPT. Lecture Notes in Computer Science*, pages 278–299. Springer, 2009.

[Ecr12] ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012), rev. 1.0. `http://www.ecrypt.eu.org/documents/D.SPA.20.pdf`, Accessed: 2012-09-30 2012.

[Fau02] Jean-Charles Faugère. A New Efficient Algorithm for Computing Gröbner Bases (F4). In *IN: ISSAC '02: PROCEEDINGS OF THE 2002 INTERNATIONAL SYMPOSIUM ON SYMBOLIC AND ALGEBRAIC COMPUTATION*, pages 75–83. Springer, 2002.

[FGLM93] J. C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering. *J. Symb. Comput.*, pages 329–344, 1993.

[FJ03] Jean-Charles Faugère and Antoine Joux. Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases. In *In Advances in Cryptology — CRYPTO 2003*, pages 44–60. Springer, 2003.

[FM14] Thomas Fuhr and Brice Minaud. Match Box Meet in the Middle Attack against Katan. In *SAC–Selected Areas in Cryptography 2014*. Springer, 2014.

[FV13] P.A. Fouque and T. Vannet. Improving Key Recovery to 784 and 799 rounds of Trivium using Optimized Cube Attacks. FSE 2013, Fast Software Encryption, 2013.

[GB08] Tim Good and Mohammed Benaissa. Hardware performance of eStream phase-III stream cipher candidates. SASC 2008 - The State of the Art of Stream Ciphers, 2008. `http://www.ecrypt.eu.org/stvl/sasc2008/`, pages 163–173.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GM88] Rüdiger Gebauer and H. Michael Möller. On an Installation of Buchberger's Algorithm. *J. Symb. Comput.*, pages 275–286, 1988.

[HR08a] Michal Hojsík and Bohuslav Rudolf. Differential Fault Analysis of Trivium. In Kaisa Nyberg, editor, *Fast Software Encryption*, Lecture Notes in Computer Science, pages 158–172. Springer Berlin Heidelberg, 2008.

[HR08b] Michal Hojsík and Bohuslav Rudolf. Floating Fault Analysis of Trivium. In *Progress in Cryptology - INDOCRYPT 2008*, Lecture Notes in Computer Science, pages 239–250. Springer Berlin Heidelberg, 2008.

[HU07] Silke Hartlieb and Luise Unger. Mathematische Grundlagen der Kryptografie. Vorlesungsskript FernUniversität Hagen, 2007.

[ISC13] Takanori Isobe, Yu Sasaki, and Jiageng Chen. Related-Key Boomerang Attacks on KATAN32/48/64. In *ACISP'13*, pages 268–285. Springer, 2013.

[JT12] Marc Joye and Michael Tunstall. *Fault Analysis in Cryptography*. Springer Publishing Company, Incorporated, 2012.

[KHK06] Shahram Khazaei, Mahdi M. Hasanzadeh, and Mohammad S. Kiaei. Linear Sequential Circuit Approximation of Grain and Trivium Stream Ciphers. Cryptology ePrint Archive, Report 2006/141, 2006. `http://eprint.iacr.org/2006/141/`, Accessed: 13.09.2014.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[KMNP11] Simon Knellwolf, Willi Meier, and María Naya-Plasencia. Conditional Differential Cryptanalysis of Trivium and KATAN. In *Selected Areas in Cryptography*, pages 200–212. Springer, 2011.

[LA94] Philippe Loustaunau and William W. Adams. *An Introduction to Grobner Bases*. American Mathematical Society, 1994.

[LN86] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, New York, NY, USA, 1986.

[Mat94] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, EUROCRYPT '93, pages 386–397, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[MBB11] MohamedSaiedEmam Mohamed, Stanislav Bulygin, and Johannes Buchmann. Using SAT Solving to Improve Differential Fault Analysis of Trivium. In *Information Security and Assurance*, Communications in Computer and Information Science, pages 62–71. Springer Berlin Heidelberg, 2011.

[Moh11] Wael Said Abdelmageed Mohamed. *Improvements for the XL Algorithm with Applications to Algebraic Cryptanalysis*. PhD thesis, TU Darmstadt, Darmstadt, Germany, Juni 2011.

[MSGR10] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In DanielJ. Bernstein and Tanja Lange, editors, *Progress in Cryptology – AFRICACRYPT 2010*, Lecture Notes in Computer Science, pages 279–296. Springer Berlin Heidelberg, 2010.

[Mö88] H. Michael Möller. On the Construction of Gröbner Bases Using Syzygies. *J. Symb. Comput.*, pages 345–359, 1988.

[Que11] Frank Quedenfeld. Algorithmen zur Berechnung von Gröbnerbasen. Master's thesis, FernUniversität Hagen, 2011.

[Que14] Frank Quedenfeld. Algebraic Fault Analysis of Katan. Cryptology ePrint Archive, Report 2014/954, 2014. `http://eprint.iacr.org/2014/954/`.

[QW14a] Frank Quedenfeld and Christopher Wolf. Advanced Algebraic Attack on Trivium. Cryptology ePrint Archive, Report 2014/893, 2014. `http://eprint.iacr.org/`.

[QW14b] Frank Quedenfeld and Christopher Wolf. Algebraic Properties of the Cube Attack. Cryptology ePrint Archive, Report 2014/800, 2014. `http://eprint.iacr.org/2014/800/`.

[Rad06] H. Raddum. Cryptanalytic results on Trivium. http://www.ecrypt.eu.org/stream/triviump3.html, 2006. Accessed: 13.09.2014.

[S+14] W. A. Stein et al. *Sage Mathematics Software (Version 6.1)*. The Sage Development Team, 2014. `http://www.sagemath.org`.

[SFP08] Ilaria Simonetti, Jean-Charles Faugère, and Ludovic Perret. Algebraic Attack Against Trivium. In *First International Conference on Symbolic Computation and Cryptography, SCC 08*, pages 95–102, Beijing, China, April 2008.

[SH13] Ling Song and Lei Hu. Improved Algebraic and Differential Fault Attacks on the KATAN Block Cipher. In *ISPEC*, pages 372–386. Springer, 2013.

[SR12] T.E. Schilling and H. Raddum. Analysis of Trivium using compressed right hand side equations. In *Information Security and Cryptology. Lecture Notes in Computer Science*, pages 18–32. Springer, 2012.

[Sta10] Paul Stankovski. Greedy Distinguishers and Nonrandomness Detectors. In *INDOCRYPT*, pages 210–226. Springer, 2010.

[T+13] S. Teo et al. Algebraic analysis of Trivium-like ciphers. Cryptology ePrint Archive, Report 2013/240, 2013. http://www.eprint.iacr.org/2013/240.pdf, Accessed: 13.09.2014.

Anhang A

Katan-Charakteristik

In den folgenden Tabellen stellen wir die Charakteristik für Katan in dem Modell aus Kapitel 5 dar. Beispielhaft bilden wir hier die Charakteristiken für die Fehler-Runde 242 ab. Insgesamt können wir eine Charakteristik für Fehler-Runden $R^* \geqslant 241$ angeben.

In unseren Charakteristiken ist in den Zeilen jeweils die Ausgabe $Z = (z_0, \ldots, z_{31})$ für die jeweilige Position e des Fehlers. - kennzeichnet einen Wert, den wir nicht berechnen können beziehungsweise der nicht konstant ist.

Tabelle A.1: Charakteristik für einen einzelnen Fehler nach $R^* = 242$ Runden von Katan. Fehler-Position e bedeutet, dass das Bit an der Position e in Runde R^* geändert wurde.

$e \backslash z_i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	0	0	-	0	-	-	0	0	0	-	0	-	-	-	-	1	-	-	-	-	-	-	0	-	-	0	-	1	0	0
1	0	0	1	-	0	-	-	-	-	0	-	0	-	0	-	-	-	-	1	-	-	-	-	-	-	1	-	0	-	0	-	0
2	0	0	0	1	-	-	0	0	0	0	0	0	0	-	0	-	0	0	0	1	-	-	0	-	0	0	0	0	0	0	0	-
3	-	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
4	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-
6	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	-	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
19	0	-	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	-
20	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	-
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
29	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabelle A.2: Charakteristik für einen zweifachen Fehler nach $R^* = 242$ Runden von Katan. Fehler-Position e bedeutet, dass das Bit an der Position e und $e+1$ in Runde R^* geändert wurde.

$e \backslash z_i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	1	-	-	0	0
1	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	1	-	-	0
2	0	0	1	1	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	0
3	-	0	0	1	1	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	1
4	-	-	0	0	1	1	0	0	0	0	0	0	0	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	1	-	-	0
5	1	1	0	0	1	1	0	0	0	0	0	0	0	0	-	-	-	0	-	-	-	-	-	-	-	-	-	-	1	-	-	0
6	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	0
7	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	0
8	-	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	1	-	-	0
9	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	1	-	-	0
10	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	1	-	-	0
11	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	1	-	-	0
12	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	-	1	-	-	0
13	0	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	-	1	-	-	0
14	0	0	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	-	1	-	-	0
15	0	0	0	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	-	1	-	-	0
16	0	0	0	0	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	-	1	-	-	0
17	0	0	0	0	0	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	-	1	-	-	0
18	0	0	0	0	0	0	0	0	0	0	0	-	-	-	1	1	0	0	1	1	0	0	0	0	0	0	0	-	1	-	-	0
19	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	0
20	-	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	1	-	-	-	-	-	-	-	1	-	-	0
21	0	0	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	0	0	1	-	-	-	-	-	-	1	-	-	0
22	0	0	0	0	0	0	0	0	0	1	1	0	-	-	-	-	-	-	0	0	0	0	1	-	-	-	-	-	1	-	-	0
23	0	0	0	0	0	0	0	0	1	1	0	0	1	-	-	-	-	-	-	-	0	0	0	1	-	-	-	-	1	-	-	0
24	0	0	0	0	0	0	0	1	1	0	0	0	0	1	-	-	-	-	-	-	-	0	0	0	1	-	-	-	1	-	-	0
25	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	0	0	1	-	-	1	-	-	0
26	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	0	0	1	-	1	-	-	0
27	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	0	0	1	1	-	-	0
28	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	0	0	1	-	-	0
29	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	0	1	-	-	0
30	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	0	1	-	-	0
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-	-	-	-	-	-	-	1	-	-	0

Tabelle A.3: Charakteristik für einen zweifachen Fehler nach $R^* = 242$ Runden von Katan. Fehler-Position e bedeutet, dass das Bit an der Position e und $e + 2$ in Runde R^* geändert wurde.

$e \backslash z_i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	-	0	-	-	-	-	0	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-
1	1	-	1	-	-	-	-	-	-	0	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-
2	0	1	0	1	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	0	1	0	1	0	0	0	0	0	0	0	-	0	-	0	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-
4	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-
5	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-
6	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-
7	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-	-	-	-	-	-
8	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-	-	-	-	-
9	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-	-	-	-
10	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-	-	-
11	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-	-
12	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1	-
13	0	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-	1
14	0	0	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-	-
15	0	0	0	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-	-
16	0	0	0	0	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-	-
17	0	0	0	0	0	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-	-
18	0	0	0	0	0	0	0	0	0	0	0	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-	-
19	0	-	0	-	0	-	0	-	0	-	0	-	-	-	-	1	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-	-
20	-	0	0	0	-	0	-	0	-	0	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0	0	-	0	-	0	-
21	0	0	0	0	0	-	0	-	0	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0	0	-	0	-	0
22	0	0	0	0	0	0	-	0	-	0	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0	0	-	0	-
23	0	0	0	0	0	0	0	-	0	-	0	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0	0	-	0
24	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0	0	-
25	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0	1
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	0	-	1	-	1	-	1	-	-	-	-	0	-	0	1	0

Tabelle A.4: Charakteristik für einen dreifachen Fehler nach $R^* = 242$ Runden von Katan. Fehler-Position e bedeutet, dass das Bit an der Position e, $e+1$ und $e+2$ in Runde R^* geändert wurde.

$e \backslash z_i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	1	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	0	1	1	1	0	-	-	-	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	0	1	1	1	0	-	-	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	0	1	1	1	1	0	-	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	1	1	1	1	0	0	0	0	0	0	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	1	1	1	1	0	0	0	0	0	0	-	-	-	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	1	1	1	1	0	0	0	0	0	0	-	-	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	1	1	1	1	0	0	0	0	0	0	-	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-
9	0	-	-	-	-	-	-	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-
10	0	0	-	-	-	-	-	-	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-
11	0	0	0	-	-	-	-	-	-	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-
12	0	0	0	0	-	-	-	-	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-
13	0	0	0	0	0	-	-	-	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-
14	0	0	0	0	0	0	-	-	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-	-
15	0	0	0	0	0	0	0	-	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-	-
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-	-
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-
19	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	-	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	1	-	-	-	-	-	-	-	-	-	-	-
21	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	-	-	-	1	-	-	-	-	-	-	-	-	-	-
22	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	-	-	-	0	1	-	-	-	-	-	-	-	-	-
23	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	-	-	-	-	0	1	-	-	-	-	-	-	-	-
24	0	0	0	0	1	1	1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	-	-	-	-	-	0	-
25	0	0	0	1	1	1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	-	-	-	-	0	0
26	0	0	1	1	1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	-	-	-	0	0
27	0	1	1	1	0	0	0	-	-	-	-	-	-	-	-	1	0	-	-	-	-	-	-	-	-	-	0	1	-	-	0	0
28	1	1	1	0	0	0	-	-	-	-	-	-	-	-	-	0	1	-	-	-	-	-	-	-	-	-	-	0	1	-	0	0
29	1	1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	0	1	-	-	-	-	-	-	-	-	-	-	0	1	0	0
30	1	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	-	-	-	-	-	-	-	-	-	-	0	1	0
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbstständig, ohne unerlaubte Hilfe Dritter angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Dritte waren an der inhaltlich-materiellen Erstellung der Dissertation nicht beteiligt; insbesondere habe ich hierfür nicht die Hilfe eines Promotionsberaters in Anspruch genommen. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Ort, Datum, Unterschrift

Frank-Michael Quedenfeld