Das CrypTool-Buch:

Kryptographie lernen und anwenden mit CrypTool und SageMath

Prof. Bernhard Esslinger
und das Entwickler-Team
des Open-Source-Projektes CrypTool

12. Auflage (2018)

Das CrypTool-Buch:

Kryptographie lernen und anwenden mit CrypTool und SageMath

Hintergrundmaterial und Zusatzinformationen
zum freien E-Learning Krypto-Programm CrypTool
(Kryptographie, Mathematik und mehr)

12. Auflage – **Draft-Version** (01:37:02)

Prof. Bernhard Esslinger (Mitautor und Herausgeber)
und das CrypTool-Team, 1998-2018

www.cryptool.org

17. Mai 2018

Zum Referenzieren per bibtex:

```
@book{Esslinger:ctb_2018,
  editor    = {Bernhard Esslinger},
  title     = {{D}as {C}ryp{T}ool-{B}uch: {K}ryptographie lernen
               und anwenden mit {C}ryp{T}ool und {S}age{M}ath},
  publisher = {CrypTool-Projekt},
  edition   = {12},
  year      = {2018}
}
```

Quelle Coverfoto: `www.photocase.com`, Andre Günther
Schriftsatz-Software: LaTeX
Versionsverwaltungs-Software: Subversion

Überblick über den Inhalt des CrypTool-Buchs

Der Erfolg des Internets hat zu einer verstärkten Forschung der damit verbundenen Technologien geführt, was auch im Bereich Kryptographie viele neue Erkenntnisse schaffte.

In diesem *Buch zu den CrypTool-Programmen* finden Sie eher mathematisch orientierte Informationen zum Einsatz von kryptographischen Verfahren. Zu einigen Verfahren gibt es Beispielcode, geschrieben für das Computer-Algebra-System **SageMath** (siehe Anhang A.7). Die Hauptkapitel sind von verschiedenen **Autoren** verfasst (siehe Anhang A.8) und in sich abgeschlossen. Am Ende der meisten Kapitel finden Sie Literaturangaben und Web-Links. Die Kapitel wurden reichlich mit *Fußnoten* versehen, in denen auch darauf verwiesen wird, wie man die beschriebenen Funktionen in den verschiedenen CrypTool-Programmen aufruft.

Das erste Kapitel beschreibt die Prinzipien der symmetrischen und asymmetrischen **Verschlüsselung** und gibt Definitionen für deren Widerstandsfähigkeit.

Im zweiten Kapitel wird – aus didaktischen Gründen – eine ausführliche Übersicht über **Papier- und Bleistiftverfahren** gegeben.

Ein großer Teil des Buchs ist dem faszinierenden Thema der **Primzahlen** (Kapitel 3) gewidmet. Anhand vieler Beispiele wird in die **modulare Arithmetik** und die **elementare Zahlentheorie** (Kapitel 4) eingeführt. Hier bilden die Eigenschaften des **RSA-Verfahrens** einen Schwerpunkt.

Danach erhalten Sie Einblicke in die mathematischen Konzepte und Ideen hinter der **modernen Kryptographie** (Kapitel 5).

Kapitel 6 gibt einen Überblick zum Stand der Attacken gegen moderne **Hashalgorithmen** und widmet sich dann kurz den **digitalen Signaturen** — sie sind unverzichtbarer Bestandteil von E-Business-Anwendungen.

Kapitel 7 stellt **Elliptische Kurven** vor: Sie sind eine Alternative zu RSA und für die Implementierung auf Chipkarten besonders gut geeignet.

Kapitel 8 führt in die **Boolesche Algebra** ein. Boolesche Algebra ist Grundlage der meisten modernen, symmetrischen Verschlüsselungsverfahren, da diese auf Bitströmen und Bitblöcken operieren. Prinzipielle Konstruktionsmethoden dieser Verfahren werden beschrieben und in SageMath implementiert.

Kapitel 9 stellt **Homomorphe Kryptofunktionen** vor: Sie sind ein modernes Forschungsgebiet, das insbesondere im Zuge des Cloud-Computing an Bedeutung gewann.

Kapitel 10 beschreibt **Aktuelle Resultate zum Lösen diskreter Logarithmen und zur Faktorisierung**. Es gibt einen breiten Überblick und Vergleich über die zur Zeit besten Algorithmen für (a) das Berechnen diskreter Logarithmen in verschiedenen Gruppen, für (b) das Faktorisierungsproblem und für (c) Elliptische Kurven. Dieser Überblick wurde zusammenge-

stellt, nachdem ein provozierender Vortrag auf der Black Hat-Konferenz 2013 für Verunsicherung sorgte, weil er die Fortschritte bei endlichen Körpern mit kleiner Charakteristik fälschlicherweise auf Körper extrapolierte, die in der Realität verwendet werden.

Das letzte Kapitel **Krypto 2020** diskutiert Bedrohungen für bestehende kryptographische Verfahren und stellt alternative Forschungsansätze (Post-Quantum-Kryptographie) für eine langfristige kryptographische Sicherheit vor.

Während die CrypTool-*eLearning-Programme* eher den praktischen Umgang motivieren und vermitteln, dient das *Buch* dazu, dem an Kryptographie Interessierten ein tieferes Verständnis für die implementierten mathematischen Algorithmen zu vermitteln – und das didaktisch möglichst gut nachvollziehbar.

Die **Anhänge** A.1, A.2, A.3 und A.4 erlauben einen schnellen Überblick über die Funktionen in den verschiedenen CrypTool-Varianten via:

- der Funktionsliste und dem Menübaum von CrypTool 1 (CT1),
- der Funktionsliste und den Vorlagen in CrypTool 2 (CT2),
- der Funktionsliste von JCrypTool (JCT), und
- der Funktionsliste von CrypTool-Online (CTO).

Die Autoren möchten sich an dieser Stelle bedanken bei den Kollegen in der jeweiligen Firma und an den Universitäten Bochum, Darmstadt, Frankfurt, Gießen, Karlsruhe, Lausanne, Paris und Siegen.

Wie auch bei dem E-Learning-Programm CrypTool wächst die Qualität des Buchs mit den Anregungen und Verbesserungsvorschlägen von Ihnen als Leser. Wir freuen uns über Ihre Rückmeldung.

Kurzinhaltsverzeichnis

Inhaltsverzeichnis

Vorwort zur 12. Auflage des CrypTool-Buchs

Das CrypTool-Buch versucht, einzelne Themen aus der Mathematik der Kryptologie genau und trotzdem möglichst verständlich zu erläutern.

Dieses Buch wurde ab dem Jahr 2000 – zusammen mit dem CrypTool-1-Paket (CT1) in Version 1.2.01 – ausgeliefert. Seitdem ist das Buch mit fast jeder neuen Version von CT1 und CT2 ebenfalls erweitert und aktualisiert worden.

Themen aus Mathematik und Kryptographie wurden sinnvoll unterteilt und dafür wurden jeweils eigenständig lesbare Kapitel geschrieben, damit Entwickler/Autoren unabhängig voneinander mitarbeiten können. Natürlich gäbe es viel mehr Themen aus der Kryptographie, die man vertiefen könnte – deshalb ist diese Auswahl auch nur eine von vielen möglichen.

In der anschließenden redaktionellen Arbeit wurden in LaTeX Querverweise ergänzt, Fußnoten hinzugefügt, Index-Einträge vereinheitlicht und Korrekturen vorgenommen.

Im Vergleich zu Ausgabe 11 des Buchs wurden in dieser Ausgabe die TeX-Sourcen des Dokuments komplett überarbeitet (bspw. eine einzige bibtex-Datei für alle Kapitel und beide Sprachen), und etliche Themen ergänzt, korrigiert und auf den aktuellen Stand gebracht, z.B.:

- die größten Primzahlen (Kap. 3.4),
- die Auflistung, in welchen Filmen und Romanen Kryptographie eine wesentliche Rolle spielt (siehe Anhang A.5),
- die Funktionsübersichten zu CrypTool 2 (CT2), zu JCrypTool (JCT) und zu CrypTool-Online (CTO) (siehe Anhang),
- weitere SageMath-Skripte zu Kryptoverfahren, und die Einführung in das Computer-Algebra-System (CAS) SageMath (siehe Anhang A.7),
- der Abschnitt über die Goldbach-Vermutung (siehe 3.8.3) und über Primzahl-Zwillinge (siehe 3.8.4),
- der Abschnitt über gemeinsame Primzahlen in real verwendeten RSA-Modulen (siehe 4.11.5.4),
- die „Einführung in die Bitblock- und Bitstrom-Verschlüsselung“ ist völlig neu (siehe Kapitel 8),
- die Studie „Resultate zur Widerstandskraft diskreter Logarithmen und zur Faktorisierung“ ist völlig neu (siehe Kapitel 10). Das ist ein phantastischer und eingehender Überblick über die Grenzen der entsprechenden aktuellen kryptoanalytischen Methoden.

Dank

An dieser Stelle möchte ich explizit folgenden Personen danken, die bisher in ganz besonderer Weise zum CrypTool-Projekt beigetragen haben. Ohne ihre besonderen Fähigkeiten und ihr großes Engagement wäre CrypTool nicht, was es heute ist:

- Hr. Henrik Koy
- Hr. Jörg-Cornelius Schneider
- Hr. Florian Marchal
- Dr. Peer Wichmann
- Hr. Dominik Schadow
- Mitarbeiter in den Teams von Prof. Johannes Buchmann, Prof. Claudia Eckert, Prof. Alexander May, Prof. Torben Weis und insbesondere Prof. Arno Wacker.

Auch allen hier nicht namentlich Genannten ganz herzlichen Dank für das (meist in der Freizeit) geleistete Engagement.

Danke auch an die Leser, die uns Feedback sandten. Und ein ganz besonderer Dank für das konstruktive Gegenlesen dieser Version durch Helmut Witten und Prof. Ralph-Hardo Schulz.

Ich hoffe, dass viele Leser mit diesem Buch mehr Interesse an und Verständnis für dieses moderne und zugleich uralte Thema finden.

Bernhard Esslinger

Heilbronn/Siegen, August 2016 + August 2017 + Mai 2018

PS:
Wir würden uns freuen, wenn sich weitere Autoren finden, die vorhandene Kapitel verbessern oder fundierte Kapitel z.B. zu einem der folgenden Themen ergänzen könnten:

- Riemannsche Zeta-Funktion,
- Hashverfahren und Passwort-Knacken,
- Gitter-basierte Kryptographie,
- Zufallszahlen,
- Format-erhaltende Verschlüsselung, Privacy-preserving Kryptographie,
- Diskussion der Wirkung verschiedener Blockmodi auf die Sicherheit,
- Design/Angriff auf Krypto-Protokolle (wie SSL).

PPS:
Ausstehende Todos für Edition 12 dieses Buches (bis dahin nennen wir es weiterhin Draft):

- Updaten aller Informationen zu SageMath (Kap. 7.9.2 und Appendix) und Testen des Codes gegen das neueste SageMath (Version 8.x), sowohl von der Kommandozeile als auch mit dem SageMathCloud-Notebook. SageMath 8 ist auch für Windows verfügbar.
- Updaten der Funktionslisten zu den vier CT-Versionen (im Appendix).

Einführung – Zusammenspiel von Buch und Programmen

Das CrypTool-Buch

Dieses Buch wird zusammen mit den Open-Source-Programmen des CrypTool-Projektes ausgeliefert. Es kann auch direkt auf der Webseite des CT-Portals herunter geladen werden (`https://www.cryptool.org/de/ctp-dokumentation`).

Die Kapitel dieses Buchs sind weitgehend in sich abgeschlossen und können auch unabhängig von den CrypTool-Programmen gelesen werden.

Für das Verständnis der meisten Kapitel reicht Abiturwissen aus. Die Kapitel 5 („Moderne Kryptografie“), 7 („Elliptische Kurven“), 8 („Bitblock- und Bitstrom-Verschlüsselung“), 9 („Homomorphe Chiffren“) und 10 („Resultate für das Lösen diskreter Logarithmen und zur Faktorisierung“) erfordern tiefere mathematische Kenntnisse.

Die Autoren haben sich bemüht, Kryptographie für eine möglichst breite Leserschaft darzustellen – ohne mathematisch unkorrekt zu werden. Sie wollen die Awareness für die IT-Sicherheit und den Einsatz standardisierter, moderner Kryptographie fördern.

Die Programme CrypTool 1, CrypTool 2 und JCrypTool

CrypTool 1 (CT1) ist ein Lernprogramm, mit dem Sie unter einer einheitlichen Oberfläche kryptographische Verfahren anwenden und analysieren können. Die umfangreicher Onlinehilfe in CT1 enthält nicht nur Anleitungen zur Bedienung des Programms, sondern auch Informationen zu den Verfahren selbst (aber weniger ausführlich und anders strukturiert als im CT-Buch).

CrypTool 1 und die Nachfolgeversionen CrypTool 2 (CT2) und JCrypTool (JCT) werden weltweit in Schule, Lehre, Aus- und Fortbildung eingesetzt.

CrypTool-Online

Die Webseite CrypTool-Online (CTO) (`http://www.cryptool-online.org`), auf der man im Browser oder vom Smartphone aus kryptographische Verfahren ausprobieren und anwenden kann, gehört ebenfalls zum CT-Projekt. Der Umfang von CTO ist bei weitem nicht so groß wie der der Standalone-Programme CT1, CT2 und JCT. Jedoch wird CTO mehr und mehr als Erstkontakt genutzt, weshalb wir Backbone und Frontend momentan mit moderner Webtechnologie neu designen, um ein schnelles, konsistentes und responsives Look&Feel anzubieten.

MTC3

Der internationale Kryptographie-Wettbewerb MysteryTwister C3 (MTC3) (`http://www.mysterytwisterc3.org`) wird ebenfalls vom CT-Projekt getragen. Hier findet man krypto-

graphische Rätsel in vier verschiedenen Kategorien, eine High-Score-Liste und ein moderiertes Forum. Stand 2016-06-16 sind über 7000 Teilnehmer dabei, und es gibt über 200 Aufgaben, von denen 162 von zumindest einem Teilnehmer gelöst wurden.

Das Computer-Algebra-Programm SageMath

SageMath ist Open-Source und ein umfangreiches Computer-Algebra-System (CAS)-Paket, mit dem sich die in diesem Buch erläuterten mathematischen Verfahren leicht Schritt-für-Schritt programmieren lassen. Eine Besonderheit dieses CAS ist, dass als Skript-Sprache Python (z.Zt. Version 2.x) benutzt wird. Dadurch stehen einem in Sage-Skripten nach einem import-Befehl auch alle Funktionen der Sprache Python zur Verfügung.
SageMath wird mehr und mehr zum Standard-CAS an Hochschulen.

Die Schüler-Krypto-Kurse

Diese Initiative bietet Ein- und Zwei-Tages-Kurse in Kryptologie für Schüler und Lehrer, um zu zeigen, wie attraktiv MINT-Fächer wie Mathematik, Informatik und insbesondere Kryptologie sind. Die Kursidee ist eine virtuelle Geheimagenten-Ausbildung.
Inzwischen finden diese Kurse seit mehreren Jahren in Deutschland in unterschiedlichen Städten statt.
Alle Kursunterlagen sind frei erhältlich auf `http://www.cryptool.org/schuelerkrypto/`.
Alle eingesetzte Software ist ebenfalls frei (meist wird CT1 und CT2 eingesetzt).
Wir würden uns freuen, wenn jemand die Kursunterlagen übersetzt und einen entsprechenden Kurs in Englisch anbieten würde.

Dank

Herzlichen Dank an alle, die mit ihrem großem Einsatz zum Erfolg und zur weiten Verbreitung dieses Projekts beigetragen haben.

Bernhard Esslinger

Heilbronn/Siegen, August 2017

Kapitel 1

Sicherheits-Definitionen und Verschlüsselungsverfahren

(Bernhard Esslinger, Jörg-Cornelius Schneider, Mai 1999; Updates: Dez. 2001, Feb. 2003, Juni 2005, Juli 2007, Jan. 2010, März 2013, Aug. 2016)

Dieses Kapitel soll einen eher beschreibenden Einstieg bieten und die Grundlagen ohne Verwendung von allzu viel Mathematik vermitteln.

Sinn der Verschlüsselung ist es, Daten so zu verändern, dass nur ein autorisierter Empfänger in der Lage ist, den Klartext zu rekonstruieren. Das hat den Vorteil, dass verschlüsselte Daten offen übertragen werden können und trotzdem keine Gefahr besteht, dass ein Angreifer die Daten unberechtigterweise lesen kann. Der autorisierte Empfänger ist im Besitz einer geheimen Information, des sogenannten Schlüssels, die es ihm erlaubt, die Daten zu entschlüsseln, während sie jedem anderen verborgen bleiben.[1]

Zur Erläuterung verwenden wir im Folgenden die Begriffe aus der Abbildung 1.1:

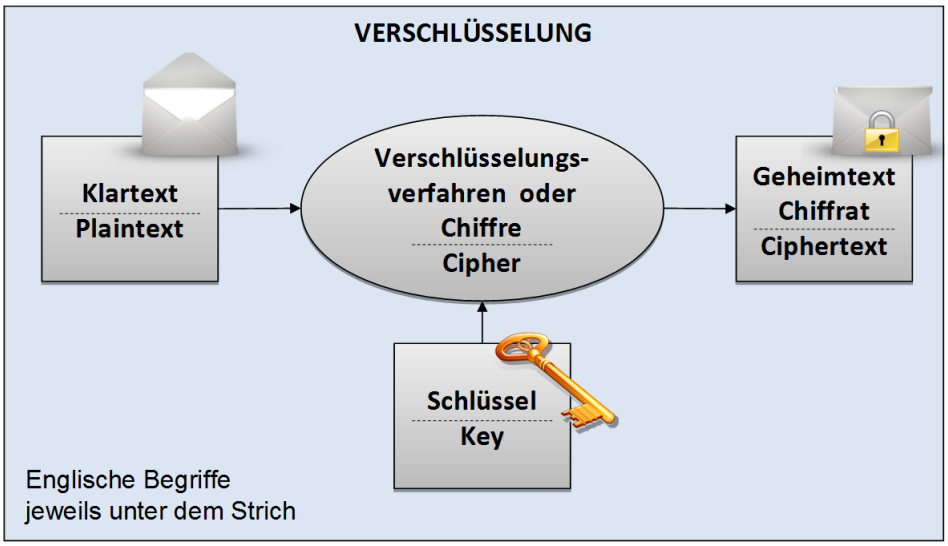


Abbildung 1.1: Übliche Bezeichnungen bei der Verwendung von Verschlüsselungsverfahren

[1] Natürlich kann ein Angreifer trotzdem die Verbindung stören oder Metadaten (wie wer mit wem kommuniziert) abgreifen.

Erkläre es mir, ich werde es vergessen.
Zeige es mir, ich werde es vielleicht behalten.
Lass es mich tun, und ich werde es können.

Zitat 1: Indisches Sprichwort

1.1 Sicherheits-Definitionen und Bedeutung der Kryptologie

Zuerst erklären wir, wie die Sicherheit von Kryptosystemen definiert wird.

Moderne Kryptographie basiert vor allem auf mathematischer Theorie und Computer-Praxis. Beim Design kryptographischer Algorithmen werden Annahmen zur Schwierigkeit von Berechnungen so gemacht, so dass sich solche Verfahren in der Praxis von einem Angreifer nur schwer brechen lassen.

Die zwei Hauptnotationen in der Literatur definieren Sicherheit in Abhängigkeit von den Möglichkeiten des Angreifers (vgl. z.B. *Contemporary Cryptography* [Opp11]):

- **Berechenbare, bedingte oder praktische Sicherheit**
  Ein Verschlüsselungsverfahren ist *berechenbar* sicher, wenn es (obwohl es theoretisch möglich ist, es zu brechen) selbst mit den besten bekannten Verfahren nicht gebrochen werden kann. Theoretische Fortschritte (z.B. Verbesserungen bei den Algorithmen zur Faktorisierung) und schnellere Computer erfordern, dass dies ständig angepasst wird.

  Selbst wenn man den besten bekannten Algorithmus zum Brechen benutzt, wird man so viele Ressourcen brauchen (z.B. 1.000.000 Jahre), dass das Kryptosystem sicher ist.

  Daher basiert dieses Konzept auf Annahmen über die begrenzte Rechenkraft des Angreifers und auf dem aktuellen Stand der Wissenschaft.

- **Informations-theoretische oder unbedingte Sicherheit**
  Ein Verschlüsselungsverfahren wird als *unbedingt* sicher bezeichnet, wenn seine Sicherheit gewährleistet ist, völlig unabhängig davon, wieviele Ressourcen (Zeit, Speicher) der Angreifer hat – also auch in dem Fall, wenn der Angreifer unbegrenzt viele Ressourcen hat, um das Verfahren zu brechen. Auch mit unbegrenzt vielen Ressourcen kann der Angreifer aus dem Chiffrat keine sinnvollen Informationen gewinnen.

  Es gibt Informations-theoretisch sichere Verfahren, die beweisbar nicht gebrochen werden können, auch nicht mit unendlich viel Rechenkraft – ein Beispiel dafür ist das *One-Time-Pad* (OTP).

  Da das OTP ein Informations-theoretisch sicheres Verschlüsselungsverfahren ist, leitet sich seine Sicherheit schon allein aus der Informationstheorie ab – und ist sicher, auch wenn der Angreifer unbegrenzte Rechenkapazitäten hat. Das OTP weist allerdings einige praktische Nachteile auf (der verwendete Schlüssel darf nur einmal verwendet werden, muss zufällig gewählt werden und mindestens so lang sein wie die zu schützende Nachricht), so dass es außer in geschlossenen Umgebungen, zum Beispiel beim heißen Draht zwischen Moskau und Washington, kaum eine Rolle spielt.

Manchmal werden auch zwei weitere Konzepte verwendet:

- **Beweisbare Sicherheit** Dies bedeutet, dass das Brechen eines Kryptosystems mindestens so schwierig ist wie die Lösung eines bestimmten schwierigen Problems, z.B. die

Berechnung des diskreten Logarithmus, die diskrete Quadratwurzel-Berechnung oder die Faktorisierung sehr großer Zahlen.

Beispiel: Aktuell wissen wir, dass RSA höchstens so schwierig ist wie die Faktorisierung, aber wir können nicht beweisen, dass es genauso schwierig ist. Deshalb hat RSA keine beweisbare Mindest-Sicherheit. Oder in anderen Worten: Wir können nicht beweisen, dass wenn das Kryptosystem RSA gebrochen ist, dass dann auch die Faktorisierung (ein schwieriges mathematisches Problem) gelöst werden kann.

Das Rabin-Kryptosystem war das erste Kryptosystem, für das sich beweisen ließ, dass es berechenbar äquivalent zu einem harten mathematischen Problem ist.

- **Ad-hoc-Sicherheit** Ein kryptographisches System hat diese Sicherheit, wenn es sich nicht lohnt, es zu versuchen es zu brechen, weil der Aufwand dafür teurer ist als der Wert der Daten, die man durch das Brechen erhalten würde. Z.B. weil ein Angriff nicht in einer ausreichend kurzen Zeit erfolgen kann (vgl. *Handbook of Applied Cryptography* [MvOV01]).

  Dies kann z.B. zutreffen, wenn Börsen-relevante Daten sowieso am nächsten Tag veröffentlicht werden und man für das Brechen ein Jahr brauchen würde.

Bei den heutzutage verwendeten guten Verfahren ist der Zeitaufwand zum Brechen so hoch, dass sie praktisch nicht gebrochen werden können. Deshalb kann man diese Verfahren als (praktisch) sicher ansehen – aus einer rein auf den Algorithmus bezogenen Sichtweise.[2]

Grundsätzlich unterscheidet man zwischen symmetrischen (siehe Kapitel 1.3) und asymmetrischen (siehe Kapitel 1.4) Verfahren zur Verschlüsselung. Einen sehr guten Überblick über die verschiedenen Verschlüsselungsverfahren bieten auch die Bücher von Bruce Schneier [Sch96b] und Klaus Schmeh [Sch16a].[3]

Bevor Verschlüsselungstechnologien mit dem Aufkommen des Internets und der drahtlosen Kommunikation für jedermann zur Verfügung standen, waren sie schon seit Jahrhunderten im Gebrauch von Regierungen, Militärs und Diplomaten. Welche Seite diese Technologie besser beherrschte konnte mit Hilfen der Geheimdienste großen Einfluss auf die **Politik** und den Kriegsverlauf nehmen. Auf die Historie geht dieses Buch nur insofern ein, als in Kapitel 2 auch die früher benutzten Verfahren vorgestellt werden. Einen Eindruck, welch entscheidende Bedeutung Kryptologie für die Mächtigen hatte und hat, kann man anhand der beiden folgenden Beispiele bekommen: dem Lehrfilm „Krieg der Buchstaben“[4] und der Debatte um die sogenannten Krypto-Wars[5].

[2] Insbesondere seit den Informationen von Edward Snowden gab es viele Diskussionen, ob Verschlüsselung sicher ist. In [ESS14] wird das Ergebnis einer Evaluierung vorgestellt, auf welche Kryptographie man sich verlassen kann – nach dem heutigen Kenntnisstand. Der Artikel untersucht: Welche Krypto-Verfahren können im Lichte der NSA-Enthüllungen noch als sicher gelten? Wo wurden Systeme gezielt geschwächt? Wie können wir die kryptographische Zukunft sicher gestalten? Wie unterscheiden sich Mathematik und Implementierung?

[3] Einen kompakten Überblick, der erklärt, was wozu verwendet wird, welche der Verfahren sicher sind, wo mit Problemen zu rechnen ist und wo die wichtigen Baustellen für die absehbare Zukunft liegen incl. des Procederes bei den Standardisierungen finden Sie in [Sch16b].

[4] Der Film schildert vor dem Hintergrund der Weltpolitik von 1900-1945 die Entwicklung der Kryptologie und ihre Bedeutung für den Kriegsverlauf im ersten (Zimmermann-Depesche) und zweiten Weltkrieg. Ausführlich wird – aus anglo-amerikanischer Sicht – darauf eingegangen, wie bedeutsam die Kryptoanalyse (auf dem Atlantik gegen die Enigma und auf dem Pazifik gegen Purple) für den Verlauf des zweiten Weltkriegs war.
Siehe `http://bscw.schule.de/pub/bscw.cgi/d1269787/Krieg_der_Buchstaben.pdf`.

[5] Siehe `https://de.wikipedia.org/wiki/Crypto_Wars`.

„Man kann nicht nicht kommunizieren!“

Zitat 2: Paul Watzlawick[6]

1.2 Einflüsse auf Verschlüsselungsverfahren

Hier sollen kurz zwei Aspekte von Kryptoverfahren erwähnt werden, auf die oft nicht früh genug eingegangen wird:

- **Zufallsbasiert**
  Man kann Algorithmen aufteilen in deterministische und heuristische Verfahren. Meistens lernten Studenten nur deterministische Verfahren kennen, bei denen die Ausgabe eindeutig durch die Eingabe vorgegeben wird. Bei heuristischen Verfahren werden Entscheidungen aufgrund zufälliger Werte getroffen. Moderne Verfahren des maschinellen Lernens gehören ebenfalls hierzu.

  In Kryptoverfahren spielt Zufall eine große Rolle. Immer müssen die Schlüssel zufällig gewählt werden, so dass zumindest bei der Schlüsselgenerierung „Zufall“ erzeugt werden muss. Zusätzlich sind manche Verfahren (vor allem aus der Kryptoanalyse) heuristisch.

- **Konstanten-basiert**
  Viele moderne Verfahren (insbesondere Hashverfahren und symmetrische Verschlüsselungsverfahren) benutzen numerische Konstanten. Diese sollten nachvollziehbar sein und keine Hintertüren ermöglichen. Zahlen, die das erfüllen, nennt man im Englischen „Nichts-im-Ärmel“-Zahlen: Nothing-up-my-sleeve number.[7]

[6] Paul Watzlawick, Janet H. Beavin und Don D. Jackson, „Menschliche Kommunikation. Formen, Störungen, Paradoxien“, Huber, (c) 2007, Das erste der fünf pragmatischen Axiome ihrer Kommunikationstheorie.

[7] `http://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number`

Die folgende Abbildung 1.2 soll eine Idee davon vermitteln, dass es nicht möglich ist, bei einem OTP den Klartext zu bestimmen (sofern das OTP-Verfahren richtig angewendet wird und alle Schlüssel gleich wahrscheinlich sind).

In dem Beispiel in der Abbildung ist als Geheimtext ein 8 Zeichen langes Wort gegeben: `11 1B 1E 18 00 04 0A 15`. Es gibt viele sinnvolle Worte aus 8 Buchstaben und zu jedem einen passenden Schlüssel. Ein Angreifer kann damit allein nicht bestimmen, welches der richtige Schlüssel bzw. welches das richtige Klartext-Wort ist.

Vergleiche auch Abbildung 8.19 in Kapitel 8.3.2, wo ein entsprechendes Text-Beispiel mit SageMath erstellt wird.

Schlüssel (hex)	Gefundener Klartext (hex)	Gefundener Klartext (txt)
52 49 47 48 54 4B 45 59	43 52 59 50 54 4F 4F 4C	CRYPTOOL
41 5A 50 57 52 45 47 54	50 41 4E 4F 52 41 4D 41	PANORAMA
54 77 7B 68 68 65 64 61	45 4C 45 50 48 41 4E 54	ELEPHANT
50 55 5B 55 4F 4A 4F 46	41 4E 45 4D 4F 4E 45 53	ANEMONES
52 53 5F 55 50 4D 45 5B	43 48 41 4D 50 49 4F 4E	CHAMPION
42 58 5B 56 41 56 43 5A	53 43 45 4E 41 52 49 4F	SCENARIO

Abbildung 1.2: Illustration für die Informations-theoretische Sicherheit des OTP[8]

[8] Bildquelle: Kostenlose Bilder von `https://pixabay.com/`

„Transparenz. Das ist das Höchste, was man sich in einer technologisch hoch entwickelten Gesellschaft erhoffen kann ... sonst wird man einfach nur manipuliert.“

Zitat 3: Daniel Suarez[9]

1.3 Symmetrische Verschlüsselung[10]

Bei der *symmetrischen* Verschlüsselung müssen Sender und Empfänger über einen gemeinsamen (geheimen) Schlüssel verfügen, den sie vor Beginn der eigentlichen Kommunikation ausgetauscht haben. Der Sender benutzt diesen Schlüssel, um die Nachricht zu verschlüsseln und der Empfänger, um diese zu entschlüsseln.

Dies veranschaulicht Abbildung 1.3:

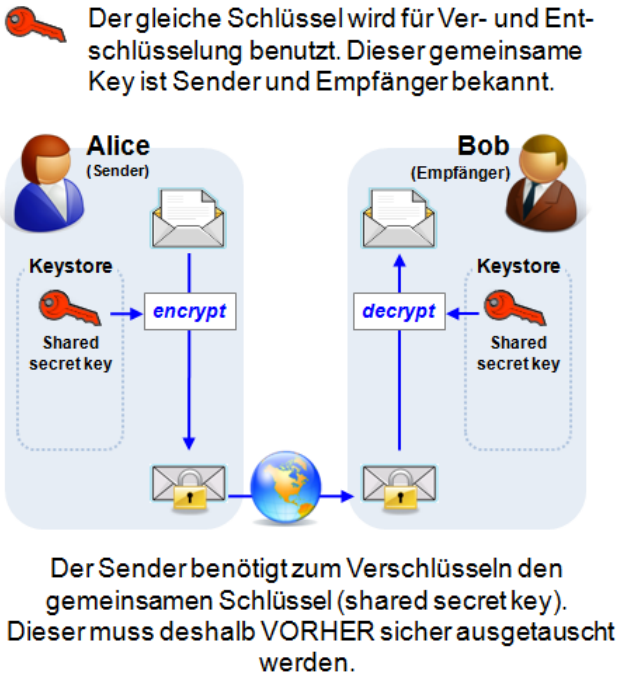


Abbildung 1.3: Symmetrische oder Secret-Key-Verschlüsselung

Alle klassischen Chiffren sind vom Typ symmetrisch. Beispiele dazu finden Sie in den CT-Programmen, im Kapitel 2 („Papier- und Bleistift-Verschlüsselungsverfahren“) in diesem Skript

[9] Daniel Suarez, „Darknet“, rororo, (c) 2011, Kapitel 5, „Einsichten“, S. 69, Price.

[10] Mit CrypTool 1 (**CT1**) können Sie über das Menü **Ver-/Entschlüsseln \ Symmetrisch (modern)** folgende modernen symmetrischen Verschlüsselungsverfahren ausführen:
IDEA, RC2, RC4, DES (ECB), DES (CBC), Triple-DES (ECB), Triple-DES (CBC), MARS (AES-Kandidat), RC6 (AES-Kandidat), Serpent (AES-Kandidat), Twofish (AES-Kandidat), Rijndael (offizielles AES-Verfahren).
Mit CrypTool 2 (**CT2**) können Sie im Startcenter über **Vorlagen \ Kryptographie \ Modern \ Symmetrisch** folgende modernen symmetrischen Verschlüsselungsverfahren ausführen:
AES, DES, PRESENT, RC2, RC4, SDES, TEA, Triple-DES, Twofish.
In JCrypTool (**JCT**) stehen Ihnen die folgenden modernen symmetrischen Verschlüsselungsverfahren zur Verfügung:
AES, Rijndael, Camellia, DES, Dragon, IDEA, LFSR, MARS, Misty1, RC2, RC5, RC6, SAFER+, SAFER++, Serpent, Shacal, Shacal2, Twofish.

oder in [Nic96]. In diesem Unterkapitel wollen wir jedoch nur die moderneren symmetrischen Verfahren betrachten.

Vorteile von symmetrischen Algorithmen sind die hohe Geschwindigkeit, mit denen Daten ver- und entschlüsselt werden. Ein Nachteil ist das Schlüsselmanagement. Um miteinander vertraulich kommunizieren zu können, müssen Sender und Empfänger vor Beginn der eigentlichen Kommunikation über einen sicheren Kanal einen Schlüssel ausgetauscht haben. Spontane Kommunikation zwischen Personen, die sich vorher noch nie begegnet sind, scheint so nahezu unmöglich. Soll in einem Netz mit n Teilnehmern jeder mit jedem zu jeder Zeit spontan kommunizieren können, so muss jeder Teilnehmer vorher mit jedem anderen der $n-1$ Teilnehmer einen Schlüssel ausgetauscht haben. Insgesamt müssen also $n(n-1)/2$ Schlüssel ausgetauscht werden.

1.3.1 AES (Advanced Encryption Standard)[11]

Vor AES war der DES-Algorithmus das bekannteste moderne, symmetrische Verschlüsselungsverfahren. Der DES-Algorithmus war eine Entwicklung von IBM in Zusammenarbeit mit der National Security Agency (NSA). Er wurde 1975 als Standard veröffentlicht. Trotz seines relativ hohen Alters ist bis heute kein „effektiver“ Angriff auf ihn gefunden worden. Der effektivste Angriff besteht aus dem Durchprobieren (fast) aller möglichen Schlüssel, bis der richtige gefunden wird (*Brute-Force-Angriff*). Aufgrund der relativ kurzen Schlüssellänge von effektiv 56 Bits (64 Bits, die allerdings 8 Paritätsbits enthalten), sind in der Vergangenheit schon mehrfach mit dem DES verschlüsselte Nachrichten gebrochen worden, so dass er heute nicht mehr als sicher anzusehen ist. Alternativen zum DES sind zum Beispiel die Algorithmen IDEA, Triple-DES (TDES) und vor allem AES.

Standard unter den symmetrischen Verfahren ist heute AES: Der dazu gehörende Rijndael-Algorithmus wurde am 2. Oktober 2000 zum Gewinner der AES-Ausschreibung erklärt und ist damit Nachfolger des DES-Verfahrens.

Einen Einstieg und weitere Verweise zum AES-Algorithmus und den AES-Kandidaten der letzten Runde finden Sie z.B. in der Online-Hilfe von CrypTool[12] oder in Wikipedia[13].

[11] In CT1 finden Sie 3 Visualisierungen dieses Verfahrens über das Menü **Einzelverfahren \ Visualisierung von Algorithmen \ AES**.
In CT2 können Sie im Startcenter mit dem Suchstring „AES“ eine Vorlage (Template) finden, die den Algorithmus Schritt-für-Schritt visualisiert.

[12] Online-Hilfe von CT1: Das Stichwort **AES** im Index führt auf die drei Hilfeseiten: **AES-Kandidaten**, **Der AES-Gewinner Rijndael** und **Der AES-Algorithmus Rijndael**.
Eine ausführliche Beschreibung von AES mit C-Code findet sich in [Haa08].

[13] `https://de.wikipedia.org/wiki/Advanced_Encryption_Standard`

Die beiden Screenshots 1.4 und 1.5 sind aus einer der 3 AES-Visualisierungen in CT1.

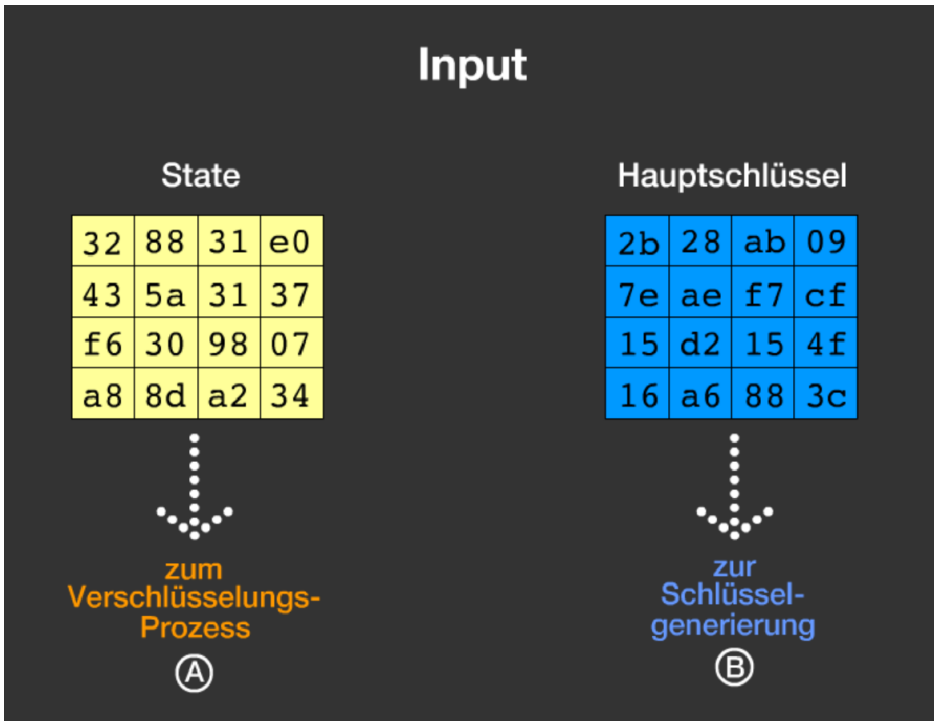


Abbildung 1.4: AES-Visualisierung von Enrique Zabala aus CT1 (Teil 1)

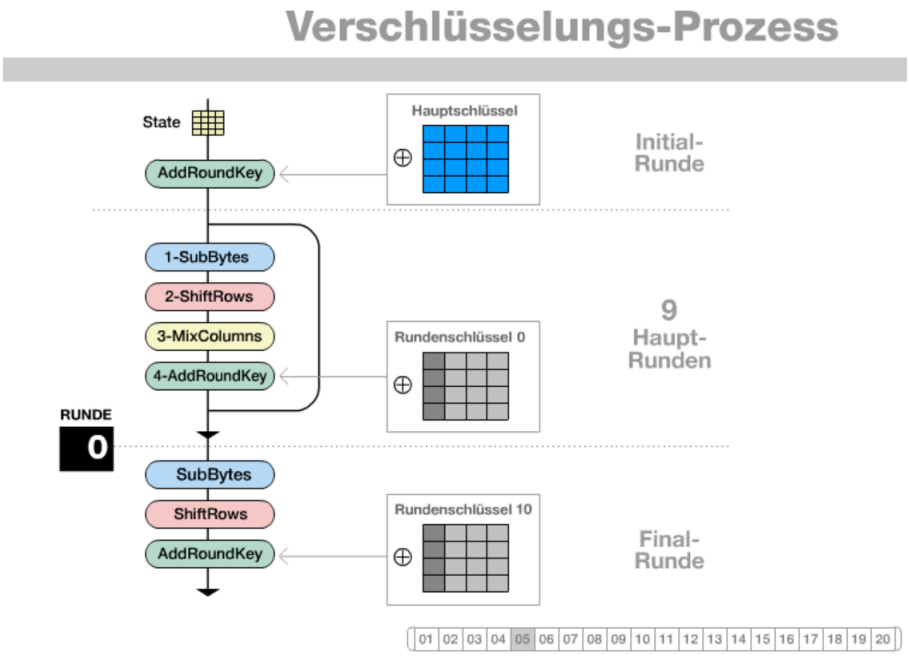


Abbildung 1.5: AES-Visualisierung von Enrique Zabala aus CT1 (Teil 2)

Im Folgenden wollen wir einen 128-bit-Block Klartext mit einem 128-bit Schlüssel mit AES im CBC-Modus verschlüsseln. Vom erhaltenen Geheimtext interessiert uns nur der 1. Block (wenn mehr vorhanden ist, ist das Padding – hier Null-Padding). Zur Veranschaulichung machen wir das einmal mit CT2 und einmal mit OpenSSL.

Abbildung 1.6 zeigt die Verschlüsselung eines Blocks in **CT2**.

Der Klartext „AESTEST1USINGCT2“ wird nach Hex (41 45 53 54 45 53 54 31 55 53 49 4E 47 43 54 32) konvertiert. Damit und mit dem Schlüssel 3243F6A8885A308D313198A2E0370734 erzeugt die AES-Komponente dann den Geheimtext. Dieser lautet in Hex:
B1 13 D6 47 DB 75 C6 D8 47 FD 8B 92 9A 29 DE 08

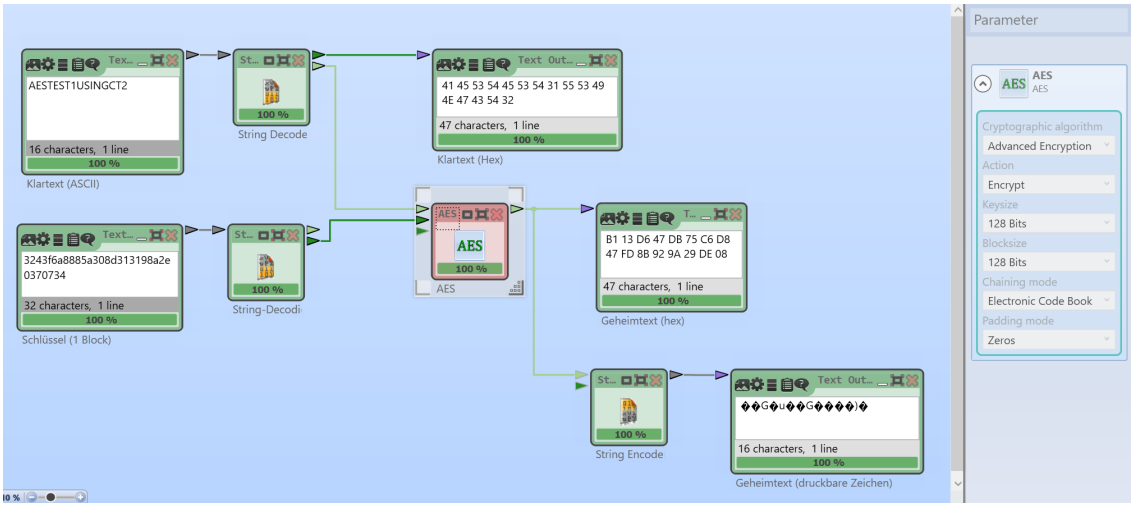


Abbildung 1.6: AES-Verschlüsselung (genau 1 Block ohne Padding) in CT2

Dasselbe Ergebnis kann man mit **OpenSSL**[14] auf der Kommandozeile auf ff. Weise erzielen:

OpenSSL-Beispiel 1.1 AES-Verschlüsselung (von genau einem Block ohne Padding) in OpenSSL

```
>openssl enc -e -aes-128-cbc
        -K 3243F6A8885A308D313198A2E0370734
        -iv 00000000000000000000000000000000
        -in klartext-1.hex  -out klartext-1.hex.enc
>dir
06.07.2016  12:43                16 key.hex
20.07.2016  20:19                16 klartext-1.hex
20.07.2016  20:37                32 klartext-1.hex.enc
```

[14] OpenSSL ist eine sehr verbreitete freie Open-Source Kryptobibliothek, die von vielen Anwendungen benutzt wird, bspw. um das TLS-Protokoll zu implementieren. Zu OpenSSL gehört auch das Kommandozeilentool openssl, mit dem man die Funktionalität auf vielen Betriebssystemen direkt ausprobieren und Zertifikate beantragen, erzeugen und verwalten kann.
Im Gegensatz zum ebenfalls sehr verbreiteten und sehr guten Kommandozeilentool gpg von GNU Privacy Guard (`https://de.wikipedia.org/wiki/GNU_Privacy_Guard`) erlaubt openssl auch Aufrufe, die sehr ins Detail gehen. Bei gpg liegt der Schwerpunkt auf den im praktischen Einsatz befindlichen Ciphersuites. Einfach genau einen Block ohne Padding zu verschlüsseln geht unseres Wissens mit dem Kommandozeilentool gpg nicht.
Siehe auch `https://de.wikipedia.org/wiki/OpenSSL`.

1.3.1.1 Ergebnisse zur theoretischen Kryptoanalyse von AES

Im Anschluss finden Sie einige Informationen, die das AES-Verfahren in letzter Zeit in Zweifel zogen – unserer Ansicht nach aber (noch) unbegründet. Die folgenden Informationen beruhen vor allem auf den unten angegebenen Originalarbeiten und auf [Wob02] und [LW02].

Der AES bietet mit einer Mindestschlüssellänge von 128 Bit gegen Brute-Force-Angriffe auch auf längere Sicht genügend Sicherheit – es sei denn, es stünden entsprechend leistungsfähige Quantencomputer zur Verfügung. Der AES war immun gegen alle bis dahin bekannten Kryptoanalyse-Verfahren, die vor allem auf statistischen Überlegungen beruhen und auf DES angewandt wurden: man konstruiert aus Geheim- und Klartextpaaren Ausdrücke, die sich nicht rein zufällig verhalten, sondern Rückschlüsse auf den verwendeten Schlüssel zulassen. Dazu waren aber unrealistische Mengen an abgehörten Daten nötig.

Bei Kryptoverfahren bezeichnet man solche Kryptoanalyseverfahren als „akademischen Erfolg“ oder als „kryptoanalytischen Angriff“, da sie theoretisch schneller sind als das Durchprobieren aller Schlüssel, das beim Brute-Force-Angriff verwendet wird. Im Fall des AES mit maximaler Schlüssellänge (256 Bit) braucht die erschöpfende Schlüsselsuche im Durchschnitt 2^{255} Verschlüsselungsoperationen. Ein kryptoanalytischer Angriff muss diesen Wert unterbieten. Als aktuell gerade noch praktikabel (z.B. für einen Geheimdienst) gilt ein Aufwand von 2^{75} bis 2^{90} Verschlüsselungsoperationen.

Eine neue Vorgehensweise wurde in der Arbeit von Ferguson, Schroeppel und Whiting im Jahre 2001 [FSW01] beschrieben: Sie stellten AES als geschlossene Formel (in der Form einer Art Kettenbruch) dar, was aufgrund seiner „relativ“ übersichtlichen Struktur gelang. Da diese Formel aus rund 1 Billiarde Summanden besteht, taugt sie zunächst nicht für die Kryptoanalyse. Dennoch war die wissenschaftliche Neugier geweckt. Bereits bekannt war, dass sich der 128-Bit-AES als ein überbestimmtes System von rund 8000 quadratischen Gleichungen (über algebraischen Zahlkörpern) mit rund 1600 Variablen (einige Unbekannte sind die Schlüsselbits) darstellen lässt – solch große Gleichungssysteme sind praktisch nicht lösbar. Dieses Gleichungssystem ist relativ dünn besetzt („sparse“), d.h. von den insgesamt etwa 1.280.000 möglichen quadratischen Termen tauchen nur relativ wenige überhaupt im Gleichungssystem auf.

Die Mathematiker Courtois und Pieprzyk [CP02] veröffentlichten 2002 eine Arbeit, die in der Krypto-Szene stark diskutiert wird: Sie entwickelten das auf der Eurocrypt 2000 von Shamir et al. vorgestellte XL-Verfahren (eXtended Linearization) weiter zum XSL-Verfahren (eXtended Sparse Linearization). Das XL-Verfahren ist eine heuristische Technik, mit der es manchmal gelingt, große nicht-lineare Gleichungssysteme zu lösen und die bei der Analyse eines asymmetrischen Algorithmus (HFE) angewandt wurde. Die Innovation von Courtois und Pieprzyk war, das XL-Verfahren auf symmetrische Verfahren anzuwenden: Das XSL-Verfahren kann auf spezielle Gleichungssysteme angewandt werden. Damit könnte ein 256-Bit-AES-Verfahren in rund 2^{230} Schritten geknackt werden. Dies ist zwar immer noch ein rein akademischer Angriff, aber er ist richtungsweisend für eine ganze Klasse von Blockchiffren. Das generelle Problem mit diesem Angriff besteht darin, dass man bisher nicht angeben kann, unter welchen Umständen er zum Erfolg führt: die Autoren geben in ihrer Arbeit notwendige Bedingungen an; es ist nicht bekannt, welche Bedingungen hinreichend sind. Neu an diesem Angriff war erstens, dass dieser Angriff nicht auf Statistik, sondern auf Algebra beruhte. Dadurch erscheinen Angriffe möglich, die nur geringe Mengen von Geheimtext brauchen. Zweitens steigt damit die Sicherheit eines Produktalgorithmus[15] nicht mehr exponentiell mit der Rundenzahl.

[15] Ein Geheimtext kann selbst wieder Eingabe für eine Chiffrierung sein. Eine Mehrfachverschlüsselung (cascade cipher) entsteht aus einer Komposition von mehreren Verschlüsselungstransformationen. Die Gesamtchiffrierung wird Produktalgorithmus oder Kaskadenalgorithmus genannt (manchmal ist die Namensgebung abhängig davon,

Aktuell wird sehr intensiv auf diesem Gebiet geforscht: z.B. stellten Murphy und Robshaw auf der Crypto 2002 ein Papier vor [MR02b], das die Kryptoanalyse drastisch verbessern könnte: Der Aufwand für 128-Bit-Schlüssel wurde auf 2^{100} geschätzt, indem sie AES als Spezialfall eines Algorithmus BES (Big Encryption System) darstellten, der eine besonders „runde“ Struktur hat. Aber auch 2^{100} Rechenschritte liegen jenseits dessen, was praktisch in absehbarer Zeit realisierbar ist. Bei 256 Bit Schlüssellänge schätzen die Autoren den Aufwand für den XSL-Angriff auf 2^{200} Operationen.

Weitere Details finden Sie unter den Weblinks bei „Das AES-/Rijndael-Kryptosystem“.

Für AES-256 wäre der Angriff ebenfalls viel besser als Brute-Force, aber noch weit außerhalb der Reichweite realisierbarer Rechenpower.

Die Diskussion war zeitweise sehr kontrovers: Don Coppersmith (einer der DES-Erfinder) z.B. bezweifelte die generelle Durchführbarkeit des Angriffs: XLS liefere für AES gar keine Lösung [Cop02]. Dann würde auch die Optimierung von Murphy und Robshaw [MR02a] nicht greifen.

2009 veröffentlichten Biryukov und Khovratovich [BK09] einen weiteren theoretischen Angriff auf AES, der sich anderer Techniken bedient, als die oben vorgestellten. Sie verwenden Methoden aus der Kryptoanalyse von Hashfunktionen (lokale Kollisionen und das Boomerang-Verfahren) und konstruierten daraus einen Related-Key-Angriff auf AES-256. D. h. der Angreifer muss nicht nur in der Lage sein, beliebige Daten zu verschlüsseln (chosen plain text), sondern auch den ihm unbekannten Schlüssel manipulieren können (related key).

Unter diesen Angriffs-Voraussetzungen reduziert der Angriff den Aufwand, einen AES-256-Schlüssel zu ermitteln, auf eine (asymmetrische) Komplexität von 2^{119} Zeit und 2^{77} Speicher. Bei AES-192 ist der Angriff noch weniger praktikabel, für AES-128 geben die Autoren keinen Angriff an.

1.3.2 Algebraische oder algorithmische Kryptoanalyse symmetrischer Verfahren

Es gibt unterschiedliche moderne Angriffsverfahren, die die Strukturen eines Problems direkt oder nach einer Transformation des Problems angreifen. Eines dieser Angriffsverfahren beruht auf dem Erfüllbarkeitsproblem der Aussagenlogik (SAT, von englisch satisfiability)[16].

Beschreibung SAT-Solver

Ein sehr altes und gut studiertes Problem in der Informatik ist das sogenannte SAT-Problem. Hier gilt es, für eine gegebene Boolesche Formel herauszufinden, ob es eine Belegung der Variablen gibt, so dass die Auswertung der Formel den Wert 1 ergibt.

Beispiel: Die Formel „A UND B“ wird zu 1 ausgewertet, wenn A=B=1 gilt. Für die Formel „A UND NICHT(A)“ gibt es keine Belegung für A, so dass sie zu 1 ausgewertet wird.

Für größere Formeln ist es nicht so einfach herauszufinden, ob eine Formel zu 1 ausgewer-

---

ob die verwendeten Schlüssel statistisch abhängig oder unabhängig sind).
Nicht immer wird die Sicherheit eines Verfahrens durch Produktbildung erhöht.
Dieses Vorgehen wird auch *innerhalb* moderner Algorithmen angewandt: Sie kombinieren in der Regel einfache und, für sich genommen, kryptologisch relativ unsichere Einzelschritte in mehreren Runden zu einem leistungsfähigen Gesamtverfahren. Die meisten modernen Blockchiffrierungen (z.B. DES, IDEA) sind Produktalgorithmen. Als Mehrfachverschlüsselung wird auch das Hintereinanderschalten desselben Verfahrens mit verschiedenen Schlüsseln wie bei Triple-DES bezeichnet.

[16] `http://de.wikipedia.org/wiki/Erf%C3%BCllbarkeitsproblem_der_Aussagenlogik`

tet werden kann (dieses Problem gehört zu den NP-vollständigen Problemen). Daher wurden spezielle Tools entwickelt, um dieses Problem für generelle Boolesche Formeln zu lösen, sogenannte SAT-Solver[17]. Wie sich herausgestellt hat, können SAT-Solver auch eingesetzt werden, um Krypto-Verfahren anzugreifen.

SAT-Solver basierte Kryptoanalyse

Der generelle Ansatz, um SAT-Solver in der Kryptoanalyse einzusetzen, ist sehr einfach: Zunächst wird das kryptographische Problem, etwa das Finden des symmetrischen Schlüssels oder die Umkehrung einer Hashfunktion, in ein SAT-Problem übersetzt. Dann wird ein SAT-Solver verwendet, um eine Lösung für das SAT-Problem zu finden. Die Lösung des SAT-Problems löst dann auch das ursprüngliche kryptographische Problem. Der Artikel von Massacci [MM00] beschreibt die erste bekannte Verwendung eines SAT-Solvers in diesem Kontext. Leider stellte sich sehr bald heraus, dass ein solch genereller Ansatz in der Praxis nicht effizient eingesetzt werden kann. Die kryptographischen SAT-Probleme sind sehr komplex und die Laufzeit eines SAT-Solvers wächst exponentiell mit der Problemgröße. Daher werden in modernen Ansätzen SAT-Solver nur für das Lösen von Teilproblemen der Kryptoanalyse verwendet. Ein gutes Beispiel zeigt der Artikel von Mironov und Zhang [MZ06]. Hier wird anhand eines Angriffs auf Hashfunktionen gezeigt, zur Lösung welcher Teilprobleme SAT-Solver effizient eingesetzt werden können.

1.3.3 Aktueller Stand der Brute-Force-Angriffe auf symmetrische Verfahren

Anhand der Blockchiffre RC5 kann der aktuelle Stand von Brute-Force-Angriffen auf symmetrische Verschlüsselungsverfahren gut erläutert werden.

Als Brute-Force-Angriff (exhaustive search, trial-and-error) bezeichnet man das vollständige Durchsuchen des Schlüsselraums: Dazu müssen keine besonderen Analysetechniken eingesetzt werden. Stattdessen wird der Geheimtext mit allen möglichen Schlüsseln des Schlüsselraums entschlüsselt[18] und geprüft, ob der resultierende Text einen sinnvollen Klartext ergibt[19]. Bei einer Schlüssellänge von 64 Bit sind dies maximal $2^{64} = 18.446.744.073.709.551.616$ oder rund 18 Trillionen zu überprüfende Schlüssel.

Um zu zeigen, welche Sicherheit bekannte symmetrische Verfahren wie DES, Triple-DES oder RC5 haben, veranstaltete z.B. RSA Security sogenannte Cipher-Challenges.[20] Unter kontrollierten Bedingungen wurden Preise ausgelobt, um Geheimtexte (verschlüsselt mit verschie-

[17] Mit **CT2** können Sie im Startcenter über **Vorlagen \ Mathematisch \ SAT-Solver (Texteingabe)** und **SAT-Solver (Dateieingabe)** einen SAT-Solver aufrufen.

[18] Mit CT1 können Sie über das Menü **Analyse \ Symmetrische Verschlüsselung (modern)** ebenfalls Brute-Force-Analysen von modernen symmetrischen Verfahren durchführen (unter der schwächsten aller möglichen Annahmen: Der Angreifer kennt nur den Geheimtext, er führt also einen Ciphertext-only-Angriff durch).
Mit CT2 können Sie bei den Vorlagen unter **Kryptoanalyse \ Modern** ebenfalls Brute-Force-Analysen durchführen. Besonders mächtig ist die KeySearcher-Komponente, die die Ausführung auch auf mehrere Rechner verteilen kann.

[19] Ist der Klartext natürlich-sprachlich und wenigstens 100 B lang, kann diese Prüfung ebenfalls automatisiert durchgeführt werden.
Um in einer sinnvollen Zeit auf einem Einzel-PC ein Ergebnis zu erhalten, sollten Sie nicht mehr als 24 Bit des Schlüssels als unbekannt kennzeichnen.

[20] `https://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-laboratories-secret-key-challenge.htm`
Im Mai 2007 meldete RSA Inc leider, dass sie die Korrektheit der bis dahin nicht gelösten RC5-72 Challenge nicht bestätigen werden.
Cipher-Challenges gibt es auch für asymmetrische Verfahren (siehe Kapitel 4.11.4).
Ein weites Spektrum einfacher und komplexer, symmetrischer und asymmetrischer Rätsel-Aufgaben bietet der internationale Krypto-Wettbewerb **MysteryTwister C3**: `http://www.mysterytwisterc3.org`.

denen Verfahren und verschiedenen Schlüssellängen) zu entschlüsseln und den symmetrischen Schlüssel zu ermitteln. Damit werden theoretische Resultate praktisch bestätigt.

Dass das „alte“ Standard-Verfahren DES mit der fixen Schlüssellänge von 56 Bit nicht mehr sicher ist, wurde schon im Januar 1999 von der Electronic Frontier Foundation (EFF) demonstriert, als sie mit Deep Crack eine DES-verschlüsselte Nachricht in weniger als einem Tag knackten.[21]

Der aktuelle Rekord für starke symmetrische Verfahren liegt bei 64 Bit langen Schlüsseln. Dazu wurde das Verfahren RC5 benutzt, eine Blockchiffre mit variabler Schlüssellänge.

Die RC5-64 Challenge wurde im Juli 2002 nach 5 Jahren vom distributed.net-Team gelöst.[22] Insgesamt arbeiteten 331.252 Personen gemeinsam über das Internet zusammen.[23] Getestet wurden rund 15 Trillionen Schlüssel, bis der richtige Schlüssel gefunden wurde.[24]

Somit sind symmetrische Verfahren (auch wenn sie keinerlei kryptographische Schwächen haben) mit 64 Bit langen Schlüsseln keine geeigneten Verfahren mehr sind, um sensible Daten länger geheim zu halten.

[21] `https://www.emc.com/emc-plus/rsa-labs/historical/des-challenge-iii.htm`

[22] `http://www.distributed.net/Pressroom_press-rc5-64`
`http://www.distributed.net/images/9/92/20020925_-_PR_-_64_bit_solved.pdf`

[23] Eine Übersicht über die aktuellen Projekte zum verteilten Rechnen finden Sie unter:
`http://distributedcomputing.info/`

[24] CT2 hat begonnen, mit einer allgemeinen Infrastruktur für verteiltes Rechnen zu experimentieren (CrypCloud, die sowohl Peer-to-Peer als auch zentralisiert eingesetzt werden kann). Damit wird CT2 in der Zukunft in die Lage versetzt, Berechnungen auf viele Computer zu verteilen. Was man erreichen kann, wenn die Komponenten für die Parallelisierung eingerichtet sind, zeigte ein Cluster zur verteilten Kryptoanalyse von DES und AES: Stand 21. März 2016 funktionierte ein Brute-force-Angriff (verteilte Schlüsselsuche) gegen AES auf 50 i5-PCs, jeder mit 4 virtuelles CPU-Kernen. Diese 200 virtuellen „Worker Threads“ konnten ca. 350 Millionen AES-Schlüssel/sec testen. Die „Cloud“ verarbeitete dabei insgesamt ca. 20 GB/sec an Daten. CrypCloud ist eine Volunteering-Cloud, so dass CT2-Nutzer sich freiwillig bei verteilten Jobs anschließen können.

1.4 Asymmetrische Verschlüsselung[25]

Bei der *asymmetrischen* Verschlüsselung hat jeder Teilnehmer ein persönliches Schlüsselpaar, das aus einem *geheimen* und einem *öffentlichen* Schlüssel besteht. Der öffentliche Schlüssel wird, der Name deutet es an, öffentlich bekanntgemacht – zum Beispiel in einem Schlüsselverzeichnis im Internet (diese Art von „Schwarzem Brett“ wird auch Directory oder öffentlicher Schlüsselring genannt) oder in einem sogenannten Zertifikat.

Die asymmetrische Verschlüsselung veranschaulicht Abbildung 1.7:

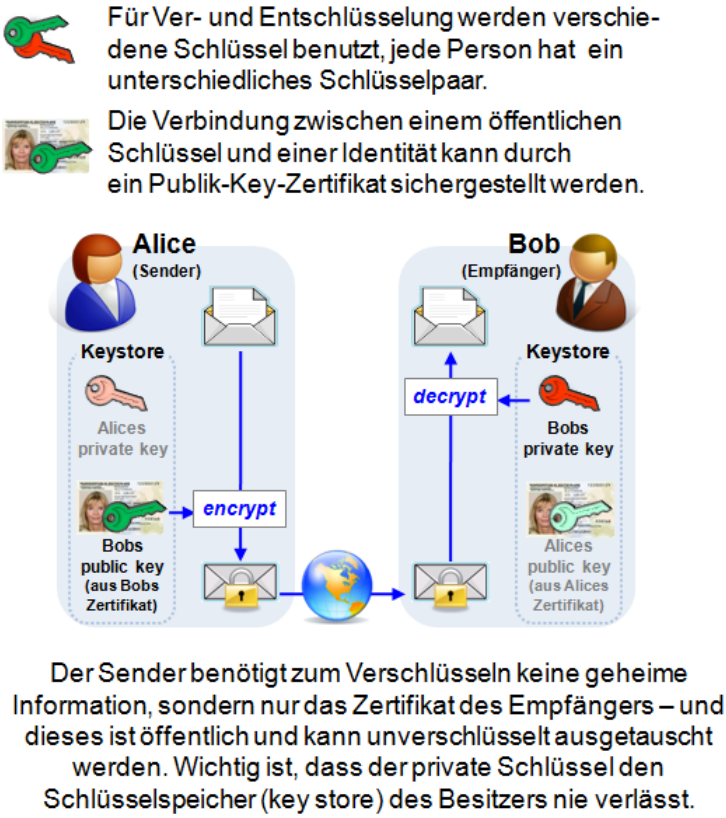


Abbildung 1.7: Asymmetrische oder Public-Key-Verschlüsselung

Möchte Alice[26] mit Bob kommunizieren, so sucht sie Bobs öffentlichen Schlüssel und benutzt ihn, um ihre Nachricht an ihn zu verschlüsseln. Diesen verschlüsselten Text schickt sie dann an

[25] Das RSA-Kryptosystem kann mit CT1 über das Menü **Einzelverfahren** \ **RSA-Kryptosystem** \ **RSA-Demo** in vielen Variationen nachvollzogen werden.
Mit CT1 können Sie über das Menü **Ver-/Entschlüsseln** \ **Asymmetrisch** mit RSA ver- und entschlüsseln. In beiden Fällen müssen Sie ein RSA-Schlüsselpaar auswählen. Nur bei der Entschlüsselung wird der geheime RSA-Schlüssel benötigt – deshalb wird nur hier die PIN abgefragt.
Mit CT2 können Sie bei den Vorlagen unter **Kryptographie** \ **Modern** ebenfalls asymmetrische Verfahren durchführen.
JCT bietet asymmetrische Verfahren wie RSA sowohl im Menü **Visualisierungen** der Standard-Perspektive als auch in der Algorithmen-Perspektive.

[26] Zur Beschreibung kryptographischer Protokolle werden den Teilnehmern oft Namen gegeben (vergleiche [Sch96b, S. 23]). Alice und Bob führen alle allgemeinen 2-Personen-Protokolle durch, wobei Alice dies initiiert und Bob antwortet. Die Angreifer werden als Eve (eavesdropper = passiver Lauscher) und Mallory (malicious active attacker = böswilliger, aktiver Abgreifer) bezeichnet.

Bob, der mit Hilfe seines geheimen Schlüssels den Text wieder entschlüsseln kann. Da einzig Bob Kenntnis von seinem geheimen Schlüssel hat, ist auch nur er in der Lage, an ihn adressierte Nachrichten zu entschlüsseln. Selbst Alice als Absenderin der Nachricht kann aus der von ihr versandten (verschlüsselten) Nachricht den Klartext nicht wieder herstellen. Natürlich muss sichergestellt sein, dass man aus dem öffentlichen Schlüssel nicht auf den geheimen Schlüssel schließen kann.

Veranschaulichen kann man sich ein solches Verfahren mit einer Reihe von einbruchssicheren Briefkästen. Wenn ich eine Nachricht verfasst habe, so suche ich den Briefkasten mit dem Namensschild des Empfängers und werfe den Brief dort ein. Danach kann ich die Nachricht selbst nicht mehr lesen oder verändern, da nur der legitime Empfänger im Besitz des Schlüssels für den Briefkasten ist.

Vorteil von asymmetrischen Verfahren ist das einfachere Schlüsselmanagement. Betrachten wir wieder ein Netz mit n Teilnehmern. Um sicherzustellen, dass jeder Teilnehmer jederzeit eine verschlüsselte Verbindung zu jedem anderen Teilnehmer aufbauen kann, muss jeder Teilnehmer ein Schlüsselpaar besitzen. Man braucht also $2n$ Schlüssel oder n Schlüsselpaare. Ferner ist im Vorfeld einer Übertragung kein sicherer Kanal notwendig, da alle Informationen, die zur Aufnahme einer vertraulichen Kommunikation notwendig sind, offen übertragen werden können. Hier ist lediglich[27] auf die Unverfälschtheit (Integrität und Authentizität) des öffentlichen Schlüssels zu achten. Nachteil: Im Vergleich zu symmetrischen Verfahren sind reine asymmetrische Verfahren jedoch um ein Vielfaches langsamer.

Das bekannteste asymmetrische Verfahren ist der RSA-Algorithmus[28], der nach seinen Entwicklern Ronald Rivest, Adi Shamir und Leonard Adleman benannt wurde. Der RSA-Algorithmus wurde 1978 veröffentlicht.[29] Das Konzept der asymmetrischen Verschlüsselung wurde erstmals von Whitfield Diffie und Martin Hellman in Jahre 1976 vorgestellt. Heute spielen auch die Verfahren nach ElGamal eine bedeutende Rolle, vor allem die Schnorr-Varianten im DSA (Digital Signature Algorithm).

Angriffe gegen asymmetrische Verfahren werden behandelt in
- Kapitel 4: Elementare Zahlentheorie,
- Kapitel 5: Moderne Kryptographie,
- Kapitel 7: Elliptische Kurven und
- Kapitel 10: Aktuelle Resultate zum Lösen diskreter Logarithmen und zur Faktorisierung.

[27] Dass auch dies nicht trivial ist, wird z.B. in Kapitel 4.11.5.4 erläutert. Neben den Anforderungen bei der Schlüsselgenerierung ist zu beachten, dass inzwischen auch die (Public-Key-)Infrastrukturen selbst Ziel von Cyber-Angriffen sind.

[28] RSA wird in diesem Skript ab Kapitel 4.10 ausführlich beschrieben. Die aktuellen Forschungsergebnisse im Umfeld von RSA werden in Kapitel 4.11 beschrieben.

[29] Hinweise zur Geschichte von RSA und seiner Veröffentlichung, die nicht im Sinne der NSA war, finden sich in der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*. Siehe [WS06], S. 55 ff („Penible Lämmergeier“).

1.5 Hybridverfahren[30]

Um die Vorteile von symmetrischen und asymmetrischen Techniken gemeinsam nutzen zu können, werden (zur Verschlüsselung) in der Praxis meist Hybridverfahren verwendet.

Hier werden die Mengen-Daten mittels symmetrischer Verfahren verschlüsselt: Der Schlüssel ist ein vom Absender zufällig[31] generierter geheimer Sitzungsschlüssel (session key), der nur für diese Nachricht verwendet wird.

Anschließend wird dieser Sitzungsschlüssel mit Hilfe des asymmetrischen Verfahrens verschlüsselt und zusammen mit der Nachricht an den Empfänger übertragen.

Der Empfänger kann den Sitzungsschlüssel mit Hilfe seines geheimen Schlüssels bestimmen und mit diesem dann die Nachricht entschlüsseln.

Auf diese Weise profitiert man von dem bequemen Schlüsselmanagement asymmetrischer Verfahren (mit öffentlichem und privatem Schlüssel), und man profitiert von der Schnelligkeit symmetrischer Verfahren, um große Datenmengen zu verschlüsseln (mit den geheimen Schlüsseln).

[30] In CT1 finden Sie dieses Verfahren über das Menü **Ver-/Entschlüsseln \ Hybrid**: Dabei können Sie die einzelnen Schritte und ihre Abhängigkeiten mit konkreten Zahlen nachvollziehen. Die Variante mit RSA als asymmetrischem Verfahren ist graphisch visualisiert; die Variante mit ECC nutzt die Standard-Dialoge. In beiden Fällen wird AES als symmetrisches Verfahren eingesetzt.
JCT bietet Hybrid-Verfahren (z.B. ECIES) in der Algorithmen-Perspektive an unter **Algorithms \ Hybrid Ciphers**.

[31] Die Erzeugung zufälliger Zahlen ist ein wichtiger Bestandteil kryptographisch sicherer Verfahren.
- Mit CT1 können Sie über das Menü **Einzelverfahren \ Zufallsdaten erzeugen** verschiedene Zufallszahlengeneratoren (PRNGs) ausprobieren. Über das Menü **Analyse \ Zufallsanalyse** können Sie verschiedene Testverfahren für Zufallsdaten auf binäre Dokumente anwenden.
- In CT2 können Sie im Startcenter mit dem Suchstring „Zufall“ Vorlagen (Templates) finden, die Zufallsgeneratoren (PRNGs) nutzen. Die PRNGs nutzen intern bspw. Keccak oder den Linear Congruential Generator (LCG). Sie werden dann bspw. für Schlüsselgenerierung oder Dezimalisierung genutzt.
- JCT bietet **Pseudo**zufallszahlengeneratoren sowohl im Menü **Algorithms \ Random Number Generator** der Standard-Perspektive als auch in der Algorithmen-Perspektive an.

Ein altes Sprichwort, angeblich geprägt von der US National Security Agency (NSA), sagt:
„Angriffe werden immer besser, niemals schlechter.“
“Attacks always get better; they never get worse.”

Zitat 4: IETF[32]

1.6 Kryptoanalyse und symmetrische Chiffren für Lehrzwecke[33]

Verglichen mit den auf der Zahlentheorie beruhenden Public-Key-Verschlüsselungsverfahren wie RSA, ist die Struktur von AES und den meisten anderen modernen symmetrischen Verschlüsselungsverfahren (wie DES, IDEA oder Present) sehr komplex und kann nicht so einfach wie RSA erklärt werden.

Deshalb wurden zu Lehrzwecken vereinfachte Varianten moderner symmetrischer Verfahren entwickelt, um Einsteigern die Möglichkeit zu geben, Ver- und Entschlüsselung von Hand zu lernen und ein besseres Verständnis zu gewinnen, wie die Algorithmen im Detail funktionieren. Diese vereinfachten Varianten helfen auch, die entsprechenden Kryptoanalyse-Methoden zu verstehen und anzuwenden.

Die bekanntesten Varianten sind SDES (Simplified DES)[34] und S-AES (Simplified-AES) von Prof. Ed Schaefer und seinen Studenten[35], und Mini-AES (siehe das Kapitel 1.8.1 „Mini-AES“):

- Edward F. Schaefer: *A Simplified Data Encryption Standard Algorithm* [Sch96a].
- Raphael Chung-Wei Phan: *Mini Advanced Encryption Standard (Mini-AES): A Testbed for Cryptanalysis Students* [Pha02].
- Raphael Chung-Wei Phan: *Impossible differential cryptanalysis of Mini-AES* [Pha03].
- Mohammad A. Musa, Edward F. Schaefer, Stephen Wedig: *A simplified AES algorithm and its linear and differential cryptanalyses* [MSW03].
- Nick Hoffman: *A SIMPLIFIED IDEA ALGORITHM* [Hof06].
- S. Davod. Mansoori, H. Khaleghei Bizaki: *On the vulnerability of Simplified AES Algorithm Against Linear Cryptanalysis* [MB07].

1.7 Weitere Informationsquellen

Neben den anderen Kapiteln in diesem Skript, der umfangreichen Fachliteratur und vielen Stellen im Internet enthält auch die Online-Hilfe aller CrypTool-Varianten sehr viele weitere Informationen zu den einzelnen symmetrischen und asymmetrischen Verschlüsselungsverfahren.

[32] `http://tools.ietf.org/html/rfc4270`

[33] Ein sehr guter Start in die Kryptoanalyse ist das Buch von Mark Stamp [SL07]. Ebenfalls gut, aber sehr high-level und nur bezogen auf die Analyse symmetrischer Blockchiffren ist der Artikel von Bruce Schneier [Sch00].
Einige der Cipher-Challenges bei „MysteryTwister C3“ (`http://www.mysterytwisterc3.org`) sind ebenfalls gut für Lehrzwecke einsetzbar.

[34] Visualisierung: Macht man in CT2 einen Doppelklick auf den Titel der SDES-Komponente, ist in der Fullscreen-Ansicht zu sehen, wie die Bits der eingegebenen Daten durch den Algorithmus fließen. Ein Screenshot dazu: `https://www.facebook.com/CrypTool2/photos/a.505204806238612.1073741827.243959195696509/597354423690316`

[35] Siehe auch den Artikel „Devising a Better Way to Teach and Learn the Advanced Encryption Standard“ unter `http://math.scu.edu/~eschaefe/getfile.pdf`

1.8 Anhang: Beispiele mit SageMath

Der folgende Abschnitt enthält SageMath Source-Code, der sich auf den Inhalt des Kapitels 1.6 („Kryptoanalyse und symmetrische Chiffren für Lehrzwecke“) bezieht.

Weitere Details zu in SageMath enthaltenen Kryptoverfahren (z.B. zum Simplified Data Encryption Standard SDES) finden sich z.B. in der Diplomarbeit von Minh Van Nguyen [Ngu09a].

1.8.1 Mini-AES

Das SageMath-Modul `crypto/block_cipher/miniaes.py` enthält den Mini-AES, mit dem Studenten die Funktionsweise moderner Blockchiffren untersuchen können.

Mini-AES, ursprünglich vorgestellt von [Pha02], ist eine vereinfachte Variante des Advanced Encryption Standard (AES) für Ausbildungszwecke.

Wie man Mini-AES benutzt, ist ausführlich in der SageMath Reference-Page beschrieben:

`http://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/block_cipher/miniaes.html`.

Das folgende SageMath-Code-Beispiel 1.1 ist aus den Release-Notes von SageMath 4.1[36] und ruft diese Implementierung des Mini-AES auf.

[36] Siehe `http://mvngu.wordpress.com/2009/07/12/sage-4-1-released/`.
Weiterer Beispiel-Code zum Mini-AES findet sich in [Ngu09b, Kap. 6.5 und Anhang D].

SageMath-Beispiel 1.1 Ver- und Entschlüsselung mit dem Mini-AES

```
# We can encrypt a plaintext using Mini-AES as follows:
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([K("x^3 + x"), K("x^2 + 1"), K("x^2 + x"), K("x^3 + x^2")]); P

[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: key = MS([K("x^3 + x^2"), K("x^3 + x"), K("x^3 + x^2 + x"), K("x^2 + x + 1")]); key

[    x^3 + x^2       x^3 + x]
[x^3 + x^2 + x   x^2 + x + 1]
sage: C = maes.encrypt(P, key); C

[            x       x^2 + x]
[x^3 + x^2 + x       x^3 + x]

# Here is the decryption process:
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True

# We can also work directly with binary strings:
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: key = bin.encoding("KE"); key
0100101101000101
sage: P = bin.encoding("Encrypt this secret message!")
sage: C = maes(P, key, algorithm="encrypt")
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True

# Or work with integers n such that 0 <= n <= 15:
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: P = [n for n in xrange(16)]; P
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: key = [2, 3, 11, 0]; key
[2, 3, 11, 0]
sage: P = maes.integer_to_binary(P)
sage: key = maes.integer_to_binary(key)
sage: C = maes(P, key, algorithm="encrypt")
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

1.8.2 Weitere symmetrische Krypto-Algorithmen in SageMath

Die Referenz zu SageMath v7.2 führt als kryptographische Funktionen u.a. auf:[37]

- Linear feedback shift register (LFSR),
- Blum-Blum-Shub (BBS): Pseudo-Zufallsbit-Generator (zu finden bei Streams),
- Gitter-basierte Funktionen.

[37] Siehe `http://doc.sagemath.org/html/en/reference/sage/crypto/index.html`, `http://doc.sagemath.org/html/en/reference/cryptography/index.html` und `http://combinat.sagemath.org/doc/reference/cryptography/sage/crypto/stream.html`

Literaturverzeichnis (CryptoMeth)

[BK09] Biryukov, Alex und Dmitry Khovratovich: *Related-key Cryptanalysis of the Full AES-192 and AES-256*. Cryptology ePrint Archive, 2009. http://eprint.iacr.org/2009/317.

[Cop02] Coppersmith, Don: *Re: Impact of Courtois and Pieprzyk results*. Journal unknown, 2002.
http://csrc.nist.gov/archive/aes/ Former link from the AES Discussion Groups.

[CP02] Courtois, Nicolas und Josef Pieprzyk: *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations*. Cryptology ePrint Archive, 2002.
A different, so called compact version of the first XSL attack, was published in the proceedings for Asiacrypt Dec 2002. http://eprint.iacr.org/2002/044.

[ESS14] Esslinger, B., J. Schneider und V. Simon: *Krypto + NSA = ? – Kryptografische Folgerungen aus der NSA-Affäre*. KES Zeitschrift für Informationssicherheit, 2014(1):70–77, März 2014.
https://www.cryptool.org/images/ctp/documents/krypto_nsa.pdf.

[FSW01] Ferguson, Niels, Richard Schroeppel und Doug Whiting: *A simple algebraic representation of Rijndael*, 2001.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.4921.

[Haa08] Haan, Kristian Laurent: *Advanced Encryption Standard (AES)*, 2008. http://www.codeplanet.eu/tutorials/cpp/51-advanced-encryption-standard.html.

[Hof06] Hoffman, Nick: *A SIMPLIFIED IDEA ALGORITHM*, 2006. http://www.nku.edu/~christensen/simplified%20IDEA%20algorithm.pdf.

[LW02] Lucks, Stefan und Rüdiger Weis: *Neue Ergebnisse zur Sicherheit des Verschlüsselungsstandards AES*. DuD, Dezember 2002.

[MB07] Mansoori, S. Davod und H. Khaleghei Bizaki: *On the vulnerability of Simplified AES Algorithm Against Linear Cryptanalysis*. IJCSNS International Journal of Computer Science and Network Security, 7(7):257–263, 2007.
http://paper.ijcsns.org/07_book/200707/20070735.pdf.

[MM00] Massacci, Fabio und Laura Marraro: *Logical Cryptanalysis as a SAT Problem: Encoding and Analysis*. Journal of Automated Reasoning Security, 24:165–203, 2000.

[MR02a] Murphy, S. P. und M. J. B. Robshaw: *Comments on the Security of the AES and the XSL Technique*, September 2002. http://crypto.rd.francetelecom.com/people/Robshaw/rijndael/rijndael.html.

[MR02b] Murphy, S. P. und M. J. B. Robshaw: *Essential Algebraic Structure within the AES.* Technischer Bericht, Crypto 2002, 2002. http://crypto.rd.francetelecom.com/people/Robshaw/rijndael/rijndael.html.

[MSW03] Musa, Mohammad A., Edward F. Schaefer und Stephen Wedig: *A simplified AES algorithm and its linear and differential cryptanalyses.* Cryptologia, 17(2):148–177, April 2003.
http://www.rose-hulman.edu/~holden/Preprints/s-aes.pdf,
http://math.scu.edu/eschaefe/ Ed Schaefer's homepage.

[MvOV01] Menezes, Alfred J., Paul C. van Oorschot und Scott A. Vanstone: *Handbook of Applied Cryptography.* Series on Discrete Mathematics and Its Application. CRC Press, 5. Auflage, 2001, ISBN 0-8493-8523-7. (Errata last update Jan 22, 2014).
http://cacr.uwaterloo.ca/hac/,
http://www.cacr.math.uwaterloo.ca/hac/.

[MZ06] Mironov, Ilya und Lintao Zhang: *Applications of SAT Solvers to Cryptanalysis of Hash Functions.* Springer, 2006.

[Ngu09a] Nguyen, Minh Van: *Exploring Cryptography Using the Sage Computer Algebra System.* Diplomarbeit, Victoria University, 2009.
http://www.sagemath.org/files/thesis/nguyen-thesis-2009.pdf,
http://www.sagemath.org/library-publications.html.

[Ngu09b] Nguyen, Minh Van: *Number Theory and the RSA Public Key Cryptosystem – An introductory tutorial on using SageMath to study elementary number theory and public key cryptography*, 2009. http://faculty.washington.edu/moishe/hanoiex/Number%20Theory%20Applications/numtheory-crypto.pdf.

[Nic96] Nichols, Randall K.: *Classical Cryptography Course, Volume 1 and 2.* Technischer Bericht, Aegean Park Press 1996, 1996. 12 lessons.
www.apprendre-en-ligne.net/crypto/bibliotheque/lanaki/lesson1.htm.

[Opp11] Oppliger, Rolf: *Contemporary Cryptography, Second Edition.* Artech House, 2. Auflage, 2011. http://books.esecurity.ch/cryptography2e.html.

[Pha02] Phan, Raphael Chung Wei: *Mini Advanced Encryption Standard (Mini-AES): A Testbed for Cryptanalysis Students.* Cryptologia, 26(4):283–306, 2002.

[Pha03] Phan, Raphael Chung Wei: *Impossible differential cryptanalysis of Mini-AES.* Cryptologia, 2003.
http://www.tandfonline.com/doi/abs/10.1080/0161-110391891964.

[Sch96a] Schaefer, Edward F.: *A Simplified Data Encryption Standard Algorithm.* Cryptologia, 20(1):77–84, 1996.

[Sch96b] Schneier, Bruce: *Applied Cryptography, Protocols, Algorithms, and Source Code in C.* Wiley, 2. Auflage, 1996.

[Sch00] Schneier, Bruce: *A Self-Study Course in Block-Cipher Cryptanalysis.* Cryptologia, 24:18–34, 2000. www.schneier.com/paper-self-study.pdf.

[Sch16a] Schmeh, Klaus: *Kryptographie – Verfahren, Protokolle, Infrastrukturen.* dpunkt.verlag, 6. Auflage, 2016. Sehr gut lesbares, aktuelles und umfangreiches Buch über Kryptographie. Geht auch auf praktische Probleme (wie Standardisierung oder real existierende Software) ein.

[Sch16b] Schmidt, Jürgen: *Kryptographie in der IT – Empfehlungen zu Verschlüsselung und Verfahren.* c't, 2016(1), 2016.
Dieser Artikel erschien ursprünglich in c't 01/2016, Seite 174. Danach veröffentlicht am 17.06.2016 in:
`http://www.heise.de/security/artikel/Kryptographie-in-der-IT-Empfehlungen-zu-Verschluesselung-und-Verfahren-3221002.html`.

[SL07] Stamp, Mark und Richard M. Low: *Applied Cryptanalysis: Breaking Ciphers in the Real World.* Wiley-IEEE Press, 2007.
`http://cs.sjsu.edu/faculty/stamp/crypto/`.

[Wob02] Wobst, Reinhard: *Angekratzt – Kryptoanalyse von AES schreitet voran.* iX, Dezember 2002. (Und der Leserbrief dazu von Johannes Merkle in der iX 2/2003).

[WS06] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 2: RSA für große Zahlen.* LOG IN, 2006(143):50–58, 2006.
`http://bscw.schule.de/pub/bscw.cgi/d404410/RSA_u_Co_NF2.pdf`.

Alle Links wurden am 10.07.2016 überprüft.

Web-Links

1. „AES Discussion Groups“ beim NIST (archive page provided for historical purposes, last update on Feb 28th, 2001)
   `http://csrc.nist.gov/archive/aes/`
2. Das AES-/Rijndael-Kryptosystem (page maintained by Nicolas T. Courtois, last update on Aug 24th, 2007)
   `http://www.cryptosystem.net/aes`
3. distributed.net: „RC5-64 has been solved“
   `http://www.distributed.net/Pressroom_press-rc5-64`
4. RSA Labs (ehemals RSA Security): „The RSA Secret Key Challenge“
   `https://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-laboratories-secret-key-challenge.htm`
5. RSA Labs (ehemals RSA Security): „DES Challenge“
   `https://www.emc.com/emc-plus/rsa-labs/historical/des-challenge-iii.htm`
6. Weiterführende Links auf der CrypTool-Homepage
   `http://www.cryptool.org`

Alle Links wurden am 10.07.2016 überprüft.

Kapitel 2

Papier- und Bleistift-Verschlüsselungsverfahren

(Christine Stötzel, Apr. 2004; Updates B.+C. Esslinger, Juni 2005; Updates: Minh Van Nguyen und Bernhard Esslinger, Nov. 2009, Juni 2010; Bernhard Esslinger, Mai 2013, Aug. 2016)

> Nur Wenige kann man glauben machen, dass es keine leichte Sache ist, eine Geheimschrift zu erdenken, die sich der Untersuchung widersetzt. Dennoch kann man rundheraus annehmen, dass menschlicher Scharfsinn keine Chiffre erdenken kann, die menschlicher Scharfsinn nicht lösen kann.

Zitat 5: Edgar Allan Poe: A Few Words on Secret Writing, 1841

Das folgende Kapitel bietet einen recht vollständigen Überblick über Papier- und Bleistiftverfahren.[1] Unter diesem Begriff lassen sich alle Verfahren zusammenfassen, die Menschen von Hand anwenden können, um Nachrichten zu ver- und entschlüsseln. Besonders populär waren diese Verfahren für Geheimdienste (und sind es immer noch), da ein Schreibblock und ein Stift – im Gegensatz zu elektronischen Hilfsmitteln – vollkommen unverdächtig sind.

Die ersten Papier- und Bleistiftverfahren entstanden bereits vor rund 3000 Jahren, aber auch während des vergangenen Jahrhunderts kamen noch zahlreiche neue Methoden hinzu. Bei allen Papier- und Bleistiftverfahren handelt es sich um symmetrische Verfahren. Selbst in den

[1] Die Fußnoten dieses Kapitel zeigen, wie man diese Verfahren mit den Offline-Programmen CrypTool 1 (**CT1**), CrypTool 2 (**CT2**) und JCrypTool (**JCT**) ausführen kann. Vergleiche dazu die Anhänge A.1, A.2 und A.3.

Viele der Verfahren lassen sich auch online im Browser durchführen, z.B. auf der Seite von CrypTool-Online (**CTO**) (`http://www.cryptool-online.org`). Vergleiche dazu den Anhang A.4 in diesem Buch.

Während die CrypTool-Webseiten und -Programme sowohl klassische und moderne Chiffren anbieten, gibt es auch mehrere Seiten, die sich mit großer Detailtiefe auf klassische Chiffren fokussieren – diese stehen oft in Verbindung mit der American Cryptogram Association (ACA) (`http://www.cryptogram.org/`).
Zum Beispiel `https://sites.google.com/site/bionspot/` und `https://encode-decode.appspot.com/` von Bion.
Eine attraktive neue Seite, um klassiche Chiffren auszuführen, ist von Phil Pilcrow (`www.cryptoprograms.com`). Diese Seite enthält auch Beschreibungen und Beispiele für jeden Verfahrenstyp. Sein FAQ vom 23. Juli 2016 sagt: „The site is designed for creating classical cipher types, not machine based or modern ones but if you want another cipher type added let me know."

Außerdem enthält das letzte Unterkapitel (2.5) dieses Kapitels zu einigen Verfahren Beispielcode für das Computer-Algebra-System **SageMath**.

ältesten Verschlüsselungsmethoden steckten schon die grundsätzlichen Konstruktionsprinzipien wie Transposition, Substitution, Blockbildung und deren Kombination. Daher lohnt es sich vor allem aus didaktischen Gesichtspunkten, diese „alten“ Verfahren genauer zu betrachten.

Erfolgreiche bzw. verbreiteter eingesetzte Verfahren mussten die gleichen Merkmale erfüllen wie moderne Verfahren:

- Vollständige Beschreibung, klare Regeln, ja fast Standardisierung (inkl. der Sonderfälle, dem Padding, etc.).
- Gute Balance zwischen Sicherheit und Benutzbarkeit (denn zu kompliziert zu bedienende Verfahren waren fehlerträchtig oder unangemessen langsam).

2.1 Transpositionsverfahren

Bei der Verschlüsselung durch Transposition bleiben die ursprünglichen Zeichen der Nachricht erhalten, nur ihre Anordnung wird geändert (Transposition = Vertauschung).[2]

2.1.1 Einführungs-Beispiele unterschiedlicher Transpositionsverfahren

- **Gartenzaun-Chiffre**[3] [Sin01]: Die Buchstaben des Klartextes werden abwechselnd in zwei (oder mehr) Zeilen geschrieben, so dass ein Zickzack-Muster entsteht. Dann werden die Zeichen zeilenweise nacheinander gelesen.
  Dieses Verfahren ist eher eine Kinderverschlüsselung.

  Siehe Tabelle 2.1.

  Klartext[4]: ein beispiel zur transposition

i	b	i	p	e	z	r	r	n	p	s	t	o		
e	n	e	s	i	l	u	t	a	s	o	i	i	n	

Tabelle 2.1: Gartenzaun-Verschlüsselung

  Geheimtext[5]: IBIPE ZRRNP STOEN ESILU TASOI IN

- **Skytale von Sparta**[6] [Sin01]: Dieses Verfahren wurde wahrscheinlich das erste Mal um 600 v.Chr. benutzt und es wurde von dem griechischen Schriftsteller und Philosophen Plutarch (50-120 v.Chr.) zuerst beschrieben.
  Um einen Holzstab wird ein Streifen Papier o.ä. gewickelt. Dann wird darauf zeilenweise der Klartext geschrieben. Nach dem Abwickeln steht auf dem Streifen der Geheimtext. Zum Entschlüsseln braucht der Empfänger einen zuvor verabredeten gleich großen Holzstab mit gleich vielen Kanten.

- **Schablonen-Chiffre** [Goe14]: Sender und Empfänger benutzen die gleiche Schablone. In deren Löcher werden zeilenweise die Klartextzeichen geschrieben, die dann spaltenweise

[2] Manchmal wird auch der Begriff Permutation verwendet, um zu beschreiben, wie Buchstaben, Buchstabengruppen oder Spalten des Klartextes vertauscht werden, z.B. in der Form $(1, 2, 3, 4, 5) \Leftrightarrow (3, 4, 2, 1, 5)$.

[3] Dieses Verfahren kann direkt in CT1 mit dem Menüpunkt **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Skytale / Gartenzaun** abgebildet werden. Man kann diese Methode auch simulieren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Permutation**: Für einen Gartenzaun mit 2 Zeilen gibt man als Schlüssel „B,A“ ein und lässt ansonsten die Standardeinstellungen (nur 1 Permutation, in der man zeilenweise ein- und spaltenweise ausliest). Mit dem Schlüssel „A,B“ würde man das Zick-zack-Muster unten so beginnen, dass der erste Buchstabe in der ersten statt in der zweiten Zeile steht.

[4] Konvention: Wenn das Alphabet nur die 26 Buchstaben verwendet, schreiben wir im Folgenden den Klartext in Kleinbuchstaben und den Geheimtext in Großbuchstaben.

[5] Die Buchstaben des Klartextes sind hier – wie historisch üblich – in 5-er Blöcken gruppiert. Man könnte o.E.d.A. auch eine andere (konstante) Blocklänge oder gar keine Trennung durch Leerzeichen wählen.

[6] Dieses Verfahren kann direkt in CT1 mit dem Menüpunkt **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Skytale / Gartenzaun** abgebildet werden. Da dieses Verschlüsselungsverfahrens ein Spezialfall der einfachen Spaltentransposition ist, kann man es in CT1 auch über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Permutation** simulieren: Für die Skytale braucht man in der Dialogbox nur die erste Permutation. Darin gibt man bei z.B. 4 Kanten als Schlüssel „1,2,3,4“ ein. Dies wäre so, als würde man den Text in 4-er Blöcken waagrecht in eine Tabelle schreiben und senkrecht auslesen. Weil der Schlüssel aufsteigend geordnet ist, bezeichnet man die Skytale auch als identische Permutation. Und weil das Schreiben und Auslesen nur einmal durchgeführt wird, als einfache (und nicht als doppelte) Permutation.
In CT2 findet sich die Skytale bei den Vorlagen unter **Kryptographie \ Klassisch**.

ausgelesen werden. Bleibt Klartext übrig, wird der Vorgang wiederholt, unter Umständen mit einer anderen Ausrichtung der Schablone.[7]

- **Fleißner-Schablone** [Sav99]: Die Fleißner-Schablone wurde im Ersten Weltkrieg von deutschen Soldaten benutzt.[8] Ein quadratisches Gitter dient als Schablone, wobei ein Viertel der Felder Löcher hat. Der erste Teil des Klartextes wird zeichenweise durch die Löcher auf ein Blatt Papier geschrieben, dann wird die Schablone um 90 Grad gedreht, und der zweite Teil des Textes wird auf das Papier geschrieben, usw. Die Kunst besteht in der richtigen Wahl der Löcher: Kein Feld auf dem Papier darf frei bleiben, es darf aber auch keines doppelt beschriftet werden. Der Geheimtext wird zeilenweise ausgelesen.

  In die Beispiel-Fleißner-Schablone in Tabelle 2.2 können 4 Mal je 16 Zeichen des Klartextes auf ein Blatt geschrieben werden:

O	-	-	-	-	O	-	-
-	-	-	O	O	-	-	O
-	-	-	O	-	-	O	-
-	-	O	-	-	-	-	-
-	-	-	-	O	-	-	-
O	-	O	-	-	-	O	-
-	O	-	-	-	-	-	O
-	-	-	O	O	-	-	-

Tabelle 2.2: 8x8-Fleißner-Schablone

2.1.2 Spalten- und Zeilentranspositionsverfahren[9]

- **Einfache Spaltentransposition** [Sav99]: Zunächst wird ein Schlüsselwort bestimmt, das über die Spalten eines Gitters geschrieben wird. Dann schreibt man den zu verschlüsselnden Text zeilenweise in dieses Gitter. Die Spalten werden entsprechend des Auftretens der Buchstaben des Schlüsselwortes im Alphabet durchnummeriert. In dieser Reihenfolge werden nun auch die Spalten ausgelesen und so der Geheimtext gebildet.[10]

  Siehe Tabelle 2.3.

  Klartext: ein beispiel zur transposition

  Transpositionsschlüssel: K=2; E=1; Y=3.
  Geheimtext: IEPLR APIOE BSEUR SSINI IZTNO TN

[7] Dieses Verfahren kann man nicht durch eine einfache Spaltentransposition darstellen.
[8] Erfunden wurde die Fleißner-Schablone bereits 1881 von Eduard Fleißner von Wostrowitz.
Eine gute Visualisierung findet sich unter www.turning-grille.com.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Visualisierungen** \ **Grille**.
[9] Die meisten der folgenden Verfahren können in CT1 mit dem Menüpunkt **Ver-/Entschlüsseln** \ **Symmetrisch (klassisch)** \ **Permutation** abgebildet werden.
[10] Darstellung mit CT1: Eingabe eines Schlüssels für die erste Permutation, zeilenweise einlesen, spaltenweise permutieren und auslesen.
In CT2 findet sich die Transposition bei den Vorlagen unter **Kryptographie** \ **Klassisch**. Diese Komponente visualisiert auch, wie der Text in die Matrix ein- und ausgelesen wird und wie die Spalten vertauscht werden.

K	E	Y
e	i	n
b	e	i
s	p	i
e	l	z
u	r	t
r	a	n
s	p	o
s	i	t
i	o	n

Tabelle 2.3: Einfache Spaltentransposition

- **AMSCO-Chiffre** [ACA02]: Die Klartextzeichen werden abwechselnd in Einer- und Zweiergruppen in ein Gitter geschrieben. Dann erfolgt eine Vertauschung der Spalten, anschließend das Auslesen.
- **Doppelte Spaltentransposition (DST) / „Doppelwürfel“** [Sav99]: Die doppelte Spaltentransposition wurde häufig im Zweiten Weltkrieg und zu Zeiten des Kalten Krieges angewendet. Dabei werden zwei Spaltentranspositionen nacheinander durchgeführt, für die zweite Transposition wird ein neuer Schlüssel benutzt.[11]
  Werden zwei unterschiedliche und genügend lange Schlüssel (mindestens je 20 Zeichen) verwendet, dann ist das auch für heutige Computer noch eine Herausforderung.[12]
- **Spaltentransposition, General Luigi Sacco** [Sav99]: Die Spalten eines Gitters werden den Buchstaben des Schlüsselwortes entsprechend nummeriert. Der Klartext wird dann zeilenweise eingetragen, in der ersten Zeile bis zur Spalte mit der Nummer 1, in der zweiten Zeile bis zur Spalte mit der Nummer 2 usw. Das Auslesen erfolgt wiederum spaltenweise.

  Siehe Tabelle 2.4.

  Klartext: ein beispiel zur transposition

  Geheimtext: ESIUR OTIPE TSINL RIOBZ ANENI SP

- **Spaltentransposition, Französische Armee im Ersten Weltkrieg** [Sav99]: Nach Durchführung einer Spaltentransposition werden diagonale Reihen ausgelesen.
- **Zeilentransposition** [Sav99]: Der Klartext wird in gleich lange Blöcke zerlegt, was auch mit Hilfe eines Gitters erfolgen kann. Dann wird die Reihenfolge der Buchstaben bzw. der Spalten vertauscht. Da das Auslesen zeilenweise erfolgt, wird nur jeweils innerhalb der Blöcke permutiert.[13]

[11] Darstellung mit CT1: Eingabe eines Schlüssels für die 1. Permutation, zeilenweise einlesen, spaltenweise permutieren und auslesen. Eingabe eines (neuen) Schlüssels für die 2. Permutation, Ergebnis der 1. Permutation zeilenweise einlesen, spaltenweise permutieren und auslesen.

[12] In MTC3 finden sich dazu Challenges, bspw.
`http://www.mysterytwisterc3.org/de/challenges/level-x-kryptographie-challenges/doppelwuerfel`
und
`https://www.mysterytwisterc3.org/de/challenges/level-3-kryptographie-challenges/doppelwuerfel-reloaded-teil-1`

[13] Darstellung mit CT1: Eingabe eines Schlüssels für die 1. Permutation, zeilenweise einlesen, spaltenweise permutieren und zeilenweise auslesen.

G 4	E 2	N 6	E 3	R 7	A 1	L 5
e s i u r o t	i p e t s i	n l r i o	b z a n	e n 	i s 	 p

Tabelle 2.4: Spaltentransposition nach General Luigi Sacco

2.1.3 Weitere Transpositionsverfahren

- **Geometrische Figuren** [Goe14]: Einem bestimmten Muster folgend wird der Klartext in ein Gitter geschrieben (Schnecke, Rösselsprung o.ä.), einem zweiten Muster folgend wird der Geheimtext ausgelesen.
- **Union Route Cipher** [Goe14]: Die Union Route Cipher hat ihren Ursprung im Amerikanischen Bürgerkrieg. Nicht die Buchstaben einer Nachricht werden umsortiert, sondern die Wörter. Für besonders prägnante Namen und Bezeichnungen gibt es Codewörter, die zusammen mit den Routen in einem Codebuch festgehalten werden. Eine Route bestimmt die Größe des Gitters, in das der Klartext eingetragen wird und das Muster, nach dem der Geheimtext ausgelesen wird. Zusätzlich gibt es eine Anzahl von Füllwörtern.
- **Nihilist-Transposition** [ACA02]: Der Klartext wird in eine quadratische Matrix eingetragen, an deren Seiten jeweils der gleiche Schlüssel steht. Anhand dieses Schlüssels werden sowohl die Zeilen als auch die Spalten alphabetisch geordnet und der Inhalt der Matrix dementsprechend umsortiert. Der Geheimtext wird zeilenweise ausgelesen.

  Siehe Tabelle 2.5.

  Klartext: ein beispiel zur transposition

	K	A	T	Z	E		A	E	K	T	Z
K	e	i	n	b	e	A	s	e	i	p	i
A	i	s	p	i	e	E	s	i	o	i	t
T	l	z	u	r	t	K	i	e	e	n	b
Z	r	a	n	s	p	T	z	t	l	u	r
E	o	s	i	t	i	Z	a	p	r	n	s

Tabelle 2.5: Nihilist-Transposition[14]

Geheimtext: SEIPI SIOIT IEENB ZTLUR APRNS

In CT2 findet sich die Transposition bei den Vorlagen unter **Kryptographie \ Klassisch**. Diese Komponente visualisiert auch die zeilenweise Transposition.

[14] Der linke Block ist das Resultat nach dem Einlesen. Der rechte Block ist das Resultat nach dem Vertauschen von Zeilen und Spalten.

- **Cadenus-Chiffre** [ACA02]: Hierbei handelt es sich um eine Spaltentransposition, die zwei Schlüsselworte benutzt.
  Das erste Schlüsselwort wird benutzt, um die Spalten zu vertauschen.
  Das zweite Schlüsselwort wird benutzt, um das Startzeichen jeder Spalte festzulegen: dieses zweite Schlüsselwort ist eine beliebige Permutation des benutzten Alphabets. Diese schreibt man links vor die erste Spalte. Jede Spalte wird dann vertikal so verschoben (wrap-around), dass sie mit dem Buchstaben beginnt, der in derjenigen Zeile, wo der Schlüsselbuchstabe des ersten Schlüsselwortes in dem zweiten Schlüsselwort zu finden ist. Der Geheimtext wird zeilenweise ausgelesen.

  Siehe Tabelle 2.6.

  Klartext: ein laengeres beispiel zur transposition mit cadenus

	K	**E**	Y	**E**	K	Y	**E**	K	Y
A	e	i	n	i	e	n	**p**	r	n
D	l	a	e	a	l	e	i	b	o
X	n	g	e	g	n	e	o	s	t
K	r	e	s	e	**r**	s	i	e	n
C	b	e	i	e	b	i	a	u	t
W	s	p	i	p	s	i	n	r	d
N	e	l	z	l	e	z	i	s	u
S	u	r	t	r	u	t	a	s	n
Y	r	a	n	a	r	**n**	g	i	e
E	s	**p**	o	**p**	s	o	e	m	e
D	s	i	t	i	s	t	e	c	s
T	i	o	n	o	i	n	p	e	i
U	m	i	t	i	m	t	l	s	i
B	c	a	d	a	c	d	r	e	z
R	e	n	u	n	e	u	a	l	t
G	s	-	-	-	s	-	-	n	-

Tabelle 2.6: Cadenus-Chiffre[15]

Geheimtext: PRNIB OOSTI ENAUT NRDIS UASNG IEEME ECSPE ILSIR EZALT N

[15] In dem zweiten Dreierblock sind diejenigen Zeichen fett, die nach der Anwendung des zweiten Schlüsselwortes oben im dritten Dreierblock stehen.

2.2 Substitutionsverfahren

2.2.1 Monoalphabetische Substitutionsverfahren

Monoalphabetische Substitutionsverfahren ordnen jedem Klartextzeichen ein Geheimtextzeichen fest zu, d.h. diese Zuordnung ist während des ganzen Verschlüsselungsprozesses dieselbe.

- **Allgemeine monoalphabetische Substitution / Zufällige Buchstabenzuordnung**[16] [Sin01]: Die Substitution erfolgt aufgrund einer festgelegten Zuordnung der Einzelbuchstaben.
- **Atbash-Chiffre**[17] [Sin01]: Der erste Buchstabe des Alphabets wird durch den letzten Buchstaben des Alphabets ersetzt, der zweite durch den vorletzten, usw.
- **Verschiebe-Chiffre, z.B. Caesar**[18] [Sin01]: Klartext- und Geheimtextalphabet werden um eine bestimmte Anzahl von Zeichen gegeneinander verschoben.

  Klartext:
  `bei der caesarchiffre wird um drei stellen verschoben`

  Geheimtext:
  `EHL GHU FDHVDUFKLIIUH ZLUG XP GUHL VWHOOHQ YHUVFKREHQ`
- **Affine Chiffre**[19]: Dies ist eine Verallgemeinerung der Verschiebechiffre. Jeder Klartext-Buchstabe wird erst durch einen anderen ersetzt und das Ergebnis wird dann mit der Verschiebechiffre verschlüsselt. Die Bezeichnung „affine Chiffre“ kommt daher, dass man die Ver- und Entschlüsselung als affine bzw. lineare Funktion beschreiben kann.
- **Substitution mit Symbolen, z.B. Freimaurerchiffre** [Sin01]: Ein Buchstabe wird durch ein Symbol ersetzt.
- **Varianten**: Füller, absichtliche Fehler [Sin01].
- **Nihilist-Substitution**[20] [ACA02]: Das Alphabet wird in eine 5x5-Matrix eingetragen und bei der Verarbeitung wird jedem Klartextbuchstaben die aus Zeilen- und Spaltennummer gebildete Zahl zugeordnet. Dann wird ein Schlüsselwort gewählt und über eine zweite Tabelle geschrieben. In diese wird der Klartext zeilenweise eingetragen. Die Geheimtextzeichen sind die Summen aus den Zahlen des Klartextes und den Zahlen des Schlüsselwortes. Bei Zahlen zwischen 100 und 110 wird die führende „1“ ignoriert, so dass jeder Buchstabe durch eine zweistellige Zahl repräsentiert wird.

[16] Dieses Verfahren kann in CT1 mit dem Menüpunkt **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Substitution / Atbash** abgebildet werden.
In CT2 finden sich diese Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch**. Entsprechende Analyzer finden sich bei den Vorlagen unter **Kryptoanalyse \ Klassisch**.

[17] Dieses Verfahren kann in CT1 mit dem Menüpunkt **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Substitution / Atbash** abgebildet werden.

[18] In CT1 kann man dieses Verfahren an drei verschiedenen Stellen im Menü finden:
- **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Caesar / ROT13**
- **Analyse \ Symmetrische Verschlüsselung (klassisch) \ Ciphertext only \ Caesar**
- **Einzelverfahren \ Visualisierung von Algorithmen \ Caesar**.

[19] Mit SageMath sind dazu unter 2.5.2.3 einige Beispiele implementiert.

[20] Eine Animation zu diesem Nihilist-Verfahren findet sich in CT1 unter dem Menüpunkt **Einzelverfahren \ Visualisierung von Algorithmen \ Nihilist**.
In CT2 findet sich Nihilist bei den Vorlagen unter **Kryptographie \ Klassisch**.

Siehe Tabelle 2.7.

Klartext: ein beispiel zur substitution

Matrix

	1	2	3	4	5
1	S	C	H	L	U
2	E	A	B	D	F
3	G	I	K	M	N
4	O	P	Q	R	T
5	V	W	X	Y	Z

Tabelle

K (33)	E (21)	Y (54)
e (54)	i (53)	n (89)
b (56)	e (42)	i (86)
s (44)	p (63)	i (86)
e (54)	l (35)	z (109)
u (48)	r (65)	s (65)
u (48)	b (44)	s (65)
t (78)	i (53)	t (99)
u (48)	t (66)	i (86)
o (74)	n (56)	

Tabelle 2.7: Nihilist-Substitution

Geheimtext: 54 53 89 56 42 86 44 63 86 54 35 09 48 65 65 48 44 65 78 53 99 48 66 86 74 56

- **Codes** [Sin01]: Im Laufe der Geschichte wurden immer wieder Codebücher verwendet. In diesen Büchern wird jedem möglichen **Wort** eines Klartextes ein Codewort, ein Symbol oder eine Zahl zugeordnet. Voraussetzung für eine erfolgreiche geheime Kommunikation ist, dass Sender und Empfänger exakt das gleiche Codebuch besitzen und die Zuordnung der Codewörter zu den Klartextwörtern nicht offengelegt wird.
- **Nomenklator** [Sin01]: Als Nomenklator werden Techniken bezeichnet, die ein Verschlüsselungsverfahren mit einem Codebuch kombinieren. Meist basiert das Verschlüsselungssystem auf einem Geheimtextalphabet, mit dem ein Großteil der Nachricht chiffriert (via Substitution) wird. Für besonders häufig auftretende oder geheim zu haltende Wörter existieren eine begrenzte Anzahl von Codewörtern.
- **Landkarten-Chiffre** [Thi99]: Diese Methode stellt eine Kombination aus Substitution

und Steganographie[21] dar. Klartextzeichen werden durch Symbole ersetzt, diese werden nach bestimmten Regeln in Landkarten angeordnet.

- **Straddling Checkerboard** [Goe14]: Eine 3x10-Matrix wird mit den Buchstaben des Alphabets und zwei beliebigen Sonderzeichen oder Zahlen gefüllt, indem zunächst die voneinander verschiedenen Zeichen eines Schlüsselwortes und anschließend die restlichen Buchstaben des Alphabetes eingefügt werden. Die Spalten der Matrix werden mit den Ziffern 0 bis 9, die zweite und dritte Zeile der Matrix mit den Ziffern 1 und 2 nummeriert. Jedes Zeichen des Geheimtextes wird durch die entsprechende Ziffer bzw. das entsprechende Ziffernpaar ersetzt. Da die 1 und die 2 die ersten Ziffern der möglichen Ziffernkombinationen sind, werden sie nicht als einzelne Ziffern verwendet.

  Siehe Tabelle 2.8.

  Klartext: substitution bedeutet ersetzung

	0	1	2	3	4	5	6	7	8	9
	S	-	-	C	H	L	U	E	A	B
1	D	F	G	I	J	K	M	N	O	P
2	Q	R	T	V	W	X	Y	Z	.	/

Tabelle 2.8: Straddling Checkerboard mit Passwort „Schluessel“

  Geheimtext: 06902 21322 23221 31817 97107 62272 27210 72227 61712

  Auffällig ist die Häufigkeit der Ziffern 1 und 2, dies wird jedoch durch die folgende Variante behoben.

- **Straddling Checkerboard, Variante** [Goe14]: Diese Form des Straddling Checkerboards wurde von sowjetischen Spionen im Zweiten Weltkrieg entwickelt. Angeblich haben auch Ernesto (Ché) Guevara und Fidel Castro diese Chiffre zur geheimen Kommunikation benutzt.[22] Das Alphabet wird in ein Gitter eingetragen (Spaltenanzahl = Länge des Schlüsselwortes), und es werden zwei beliebige Ziffern als „reserviert“ festgelegt, die später die zweite und dritte Zeile einer 3x10-Matrix bezeichnen (in unserem Bsp. 3 und 7). Nun wird das Gitter mit dem erzeugten Alphabet spaltenweise durchlaufen und Buchstaben zeilenweise in die Matrix übertragen: Die acht häufigsten Buchstaben (ENIRSATD für die deutsche Sprache) bekommen zur schnelleren Chiffrierung die Ziffern 0 bis 9 zugewiesen, dabei werden die reservierten Ziffern nicht vergeben. Die übrigen Buchstaben werden der Reihe nach in die Matrix eingetragen. Gegebenenfalls wird als zweite Stufe der Verschlüsselung zum Geheimtext noch eine beliebige Ziffernfolge addiert.

  Siehe Tabelle 2.9.

  Klartext: substitution bedeutet ersetzung

  Geheimtext: 07434 05957 45976 63484 87458 58208 53674 675

[21] Statt eine Nachricht zu verschlüsseln, versucht man bei der reinen Steganographie, die Existenz der Nachricht zu verbergen.

[22] Zusätzlich benutzte Ché Guevara für seine Kommunikation mit Fidel Castro auch ein One-Time-Pad.
Siehe Teil 3 aus der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*: [WLS99], S. 52.

Gitter

S	C	H	L	U	**E**
A	B	**D**	F	G	**I**
J	K	M	**N**	O	P
Q	**R**	**T**	V	W	X
Y	Z	.	/		

Matrix

	0	1	2	3	4	5	6	7	8	9
	S	**A**	**R**	-	**D**	**T**	**N**	-	**E**	**I**
3	J	Q	Y	C	B	K	Z	H	M	.
7	L	F	V	/	U	G	O	W	P	X

Tabelle 2.9: Variante des Straddling Checkerboards

  - **Ché Guevara-Chiffre**: Einen Spezialfall dieser Variante benutzte Ché Guevara (mit einem zusätzlichen Substitutionsschritt und einem leicht modifizierten Checkerboard):
    * Die sieben häufigsten Buchstaben im Spanischen werden auf die erste Zeile verteilt.
    * Es werden vier statt drei Zeilen benutzt.
    * Damit konnte man $10 * 4 - 4 = 36$ verschiedene Zeichen verschlüsseln.

- **Tri-Digital-Chiffre** [ACA02]: Aus einem Schlüsselwort der Länge 10 wird ein numerischer Schlüssel gebildet, indem die Buchstaben entsprechend ihres Auftretens im Alphabet durchnummeriert werden. Dieser Schlüssel wird über ein Gitter mit zehn Spalten geschrieben. In dieses Gitter wird unter Verwendung eines Schlüsselwortes zeilenweise das Alphabet eingetragen, wobei die letzte Spalte frei bleibt. Die Klartextzeichen werden durch die Zahl über der entsprechenden Spalte substituiert, die Zahl über der freien Spalte dient als Trennzeichen zwischen den einzelnen Wörtern.

- **Baconian-Chiffre** [ACA02]: Jedem Buchstaben des Alphabets und 6 Zahlen oder Sonderzeichen wird ein fünfstelliger Binärcode zugeordnet (zum Beispiel 00000 = A, 00001 = B, usw.). Die Zeichen der Nachricht werden entsprechend ersetzt. Nun benutzt man eine zweite, unverdächtige Nachricht, um darin den Geheimtext zu verbergen. Dies kann zum Beispiel durch Klein- und Großschreibung oder kursiv gesetzte Buchstaben geschehen: man schreibt z.B. alle Buchstaben in der unverdächtigen Nachricht groß, die unter einer „1“ stehen. Insgesamt fällt das sicher auf.

  Siehe Tabelle 2.10.

Nachricht	H	I	L	F	E
Geheimtext	`00111`	`01000`	`01011`	`00101`	`00100`
Unverdächtige Nachricht	`esist`	`warmu`	`nddie`	`sonne`	`scheint`
Baconian Cipher	`esIST`	`wArmu`	`nDdIE`	`soNnE`	`scHeint`

Tabelle 2.10: Baconian-Chiffre

2.2.2 Homophone Substitutionsverfahren

Homophone Verfahren stellen eine Sonderform der monoalphabetischen Substitution dar. Jedem Klartextzeichen werden mehrere Geheimtextzeichen zugeordnet.

- **Homophone monoalphabetische Substitution**[23] [Sin01]: Um die typische Häufigkeitsverteilung der Buchstaben einer natürlichen Sprache zu verschleiern, werden einem Klartextbuchstaben mehrere Geheimtextzeichen fest zugeordnet. Die Anzahl der zugeordneten Zeichen richtet sich gewöhnlich nach der Häufigkeit des zu verschlüsselnden Buchstabens.
- **Beale-Chiffre** [Sin01]: Die Beale-Chiffre ist eine Buchchiffre, bei der die Wörter eines Schlüsseltextes durchnummeriert werden. Diese Zahlen ersetzen die Buchstaben des Klartextes durch die Anfangsbuchstaben der Wörter.
- **Grandpré-Chiffre** [Sav99]: Eine 10x10-Matrix (auch andere Größen sind möglich) wird mit zehn Wörter mit je zehn Buchstaben gefüllt, so dass die Anfangsbuchstaben ein elftes Wort ergeben. Da die Spalten und Zeilen mit den Ziffern 0 bis 9 durchnummeriert werden, lässt sich jeder Buchstabe durch ein Ziffernpaar darstellen. Es ist offensichtlich, dass bei einhundert Feldern die meisten Buchstaben durch mehrere Ziffernpaare ersetzt werden können. Wichtig ist, dass die zehn Wörter möglichst alle Buchstaben des Alphabets enthalten.
- **Buch-Chiffre**: Die Wörter eines Klartextes werden durch Zahlentripel der Form „Seite-Zeile-Position“ ersetzt. Diese Methode setzt eine genaue Absprache des verwendeten Buches voraus, so muss es sich insbesondere um die gleiche Ausgabe handeln (Layout, Fehlerkorrekturen, etc.).

2.2.3 Polygraphische Substitutionsverfahren

Bei der polygraphische Substitution werden keine einzelne Buchstaben ersetzt, sondern Buchstabengruppen. Dabei kann es sich um Digramme, Trigramme, Silben, etc. handeln.

- **Große Chiffre** [Sin01]: Die Große Chiffre wurde von Ludwig XIV. verwendet und erst kurz vor Beginn des 20. Jahrhunderts entschlüsselt. Die Kryptogramme enthielten 587 verschieden Zahlen, jede Zahl repräsentierte eine Silbe. Die Erfinder dieser Chiffre (Rossignol, Vater und Sohn) hatten zusätzliche Fallen eingebaut, um die Sicherheit der Methode zu erhöhen. Eine Zahl konnte beispielsweise die vorangehende löschen oder ihr eine andere Bedeutung zuweisen.
- **Playfair-Chiffre**[24] [Sin01]: Eine 5x5-Matrix wird mit dem Alphabet gefüllt, z.B. erst mit den verschiedenen Zeichen eines Schlüsselworts, dann mit den restlichen Buchstaben des Alphabets. Der Klartext wird in Digramme unterteilt, die nach den folgenden Regeln verschlüsselt werden:

[23] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Homophone** aufrufen.

[24] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Playfair** aufrufen.
In CT2 findet sich Playfair bei den Vorlagen unter **Kryptographie \ Klassisch**.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Algorithmen \ Klassisch \ Playfair**.

1. Befinden sich die Buchstaben in derselben Spalte, werden sie durch die Buchstaben ersetzt, die direkt darunter stehen.
2. Befinden sich die Buchstaben in derselben Zeile, nimmt man jeweils den Buchstaben rechts vom Klartextzeichen.
3. Befinden sich die Buchstaben in unterschiedlichen Spalten und Zeilen, nimmt man jeweils den Buchstaben, der zwar in derselben Zeile, aber in der Spalte des anderen Buchstabens steht.
4. Für Doppelbuchstaben (falls sie in einem Digramm vorkommen) gelten Sonderregelungen, wie zum Beispiel die Trennung durch einen Füller.

Siehe Tabelle 2.11.

Unformatierter Klartext: buchstaben werden paarweise verschluesselt
Formatierter Klartext: bu ch st ab en we rd en pa ar we is ev er sc hl ue ss elt
Formatierter Klartext: bu ch st ab en we rd en pa ar we is ev er sc hl ue sx se lt

S	C	H	L	U
E	A	B	D	F
G	I	K	M	N
O	P	Q	R	T
V	W	X	Y	Z

Tabelle 2.11: 5x5-Playfair-Matrix mit dem Passwort „Schluessel“

Geheimtext: FHHLU OBDFG VAYMF GWIDP VAGCG SDOCH LUSFH VEGUR

- **Playfair für Trigramme** [Sav99]: Zunächst füllt man eine 5x5-Matrix mit dem Alphabet und teilt den Klartext in Trigramme auf. Für die Verschlüsselung gelten folgende Regeln:
  1. Drei gleiche Zeichen werden durch drei gleiche Zeichen ersetzt, es wird der Buchstabe verwendet, der rechts unter dem ursprünglichen Buchstaben steht (Beispiel anhand von Tabelle 11: BBB ⇒ MMM).
  2. Bei zwei gleichen Buchstaben in einem Trigramm gelten die Regeln der Original-Playfair-Verschlüsselung.
  3. Bei drei unterschiedlichen Buchstaben kommen relativ komplizierte Regeln zur Anwendung, mehr dazu unter [Sav99].
- **Ersetzung von Digrammen durch Symbole** [Sav99]: Giovanni Battista della Porta, 15. Jahrhundert. Er benutzte eine 20x20-Matrix, in die er für jede mögliche Buchstabenkombination (das verwendete Alphabet bestand aus nur zwanzig Zeichen) ein Symbol eintrug.
- **Four Square-Chiffre** [Sav99]: Diese Methode ähnelt Playfair, denn es handelt sich um ein Koordinatensystem, dessen vier Quadranten jeweils mit dem Alphabet gefüllt werden, wobei die Anordnung des Alphabets von Quadrant zu Quadrant unterschiedlich sein kann. Um eine Botschaft zu verschlüsseln, geht man wie folgt vor: Man sucht den ersten Klartextbuchstaben im ersten Quadranten und den zweiten Klartextbuchstaben im dritten Quadranten. Denkt man sich ein Rechteck mit den beiden Klartextbuchstaben als gegenüberliegende Eckpunkte, erhält man im zweiten und vierten Quadranten die zugehörigen Geheimtextzeichen.

Siehe Tabelle 2.12.

Klartext: buchstaben werden paarweise verschluesselt

d	w	x	y	m	E	P	T	O	L
r	q	e	k	i	C	V	I	Q	Z
u	v	h	p	s	R	M	A	G	U
a	l	**b**	z	n	F	W	**Y**	H	S
g	c	o	f	t	B	N	D	X	K
Q	T	B	L	E	v	q	i	p	g
Z	H	**N**	D	X	s	t	**u**	o	h
P	M	I	Y	C	n	r	d	x	y
V	S	K	W	O	b	l	w	m	f
U	A	F	R	G	c	z	k	a	e

Tabelle 2.12: Four Square Cipher

Geheimtext: YNKHM XFVCI LAIPC IGRWP LACXC BVIRG MKUUR XVKT

- **Two Square-Chiffre** [Sav99]: Die Vorgehensweise gleicht der der Four Square Cipher, allerdings enthält die Matrix nur zwei Quadranten. Befinden sich die beiden zu ersetzenden Buchstaben in der gleichen Reihe, werden sie nur vertauscht. Andernfalls werden die beiden Klartextzeichen als gegenüberliegende Eckpunkte eines Rechtecks betrachtet und durch die anderen Eckpunkte ersetzt. Die Anordnung der beiden Quadranten ist horizontal und vertikal möglich.
- **Tri Square-Chiffre** [ACA02]: Drei Quadranten werden jeweils mit dem Alphabet gefüllt. Der erste Klartextbuchstabe wird im ersten Quadranten gesucht und kann mit jedem Zeichen der selben Spalte verschlüsselt werden. Der zweite Klartextbuchstabe wird im zweiten Quadranten (diagonal gegenüberliegend) gesucht und kann mit jedem Buchstaben derselben Zeile verschlüsselt werden. Zwischen diese Geheimtextzeichen wird der Buchstabe des Schnittpunktes gesetzt.
- **Dockyard-Chiffre / Werftschlüssel** [Sav99]: Angewendet von der Deutschen Marine im Zweiten Weltkrieg.

2.2.4 Polyalphabetische Substitutionsverfahren

Bei der polyalphabetischen Substitution ist die Zuordnung Klartext-/Geheimtextzeichen nicht fest, sondern variabel (meist abhängig vom Schlüssel).

- **Vigenère**[25] [Sin01]: Entsprechend den Zeichen eines Schlüsselwortes wird jedes Klartextzeichen mit einem anderen Geheimtextalphabet verschlüsselt (als Hilfsmittel dient das

[25] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Vigenère** aufrufen.
In CT2 findet sich dieses Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch**.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Algorithmen \ Klassisch \ Vigenère**.

sog. Vigenère-Tableau). Ist der Klartext länger als der Schlüssel, wird dieser wiederholt. Siehe Tabelle 2.13.

Klartext:	das	alphabet	wechselt	staendig
Schlüssel:	KEY	KEYKEYKE	YKEYKEYK	EYKEYKEY
Geheimtext:	NEQ	KPNREZOX	UOGFCIJD	WRKILNME

-	A	B	C	**D**	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	**N**	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
...	...	...																								

Tabelle 2.13: Vigenère-Tableau

- **Unterbrochener Schlüssel**: Der Schlüssel wird nicht fortlaufend wiederholt, sondern beginnt mit jedem neuen Klartextwort von vorne.
- **Autokey-Variante**[26] [Sav99]: Nachdem der vereinbarte Schlüssel abgearbeitet wurde, geht man dazu über, die Zeichen der Nachricht als Schlüssel zu benutzen. Siehe Tabelle 2.14.

Klartext:	das	alphabet	wechselt	staendig
Schlüssel:	KEY	DASALPHA	BETWECHS	ELTSTAEN
Geheimtext:	NEQ	DLHHLQLA	XIVDWGSL	WETWGDMT

Tabelle 2.14: Autokey-Variante von Vigenère

- **Progressive-Key-Variante** [Sav99]: Der Schlüssel ändert sich im Laufe der Chiffrierung, indem er das Alphabet durchläuft. So wird aus KEY LFZ.
- **Gronsfeld** [Sav99]: Vigenère-Variante, die einen Zahlenschlüssel verwendet.
- **Beaufort** [Sav99]: Vigenère-Variante, keine Verschlüsselung durch Addition, sondern durch Subtraktion. Auch mit rückwärts geschriebenem Alphabet.
- **Porta** [ACA02]: Vigenère-Variante, die nur 13 Alphabete verwendet. Das bedeutet, dass jeweils zwei Schlüsselbuchstaben dasselbe Geheimtextalphabet zugeordnet wird, und die erste und zweite Hälfte des Alphabets reziprok sind.

[26] In CT2 findet sich dieses Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch**.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Algorithmen \ Klassisch \ Autokey-Vigenère**.

  - **Slidefair** [ACA02]: Kann als Vigenère-, Gronsfeld- oder Beaufort-Variante verwendet werden. Dieses Verfahren verschlüsselt Digramme. Den ersten Buchstaben sucht man im Klartextalphabet über dem Tableau, den zweiten in der Zeile, die dem Schlüsselbuchstaben entspricht. Diese beiden Punkte bilden gegenüberliegende Punkte eines gedachten Rechtecks, die verbleibenden Ecken bilden die Geheimtextzeichen.

- **One-Time-Pad (OTP)**[27,28]: Dies ist eine fundamentale Idee: Eine Folge von Bytes wird Byte-für-Byte mit dem Klartext XOR-verknüpft. Dies ist eine Verallgemeinerung des Vigenère-Verfahrens, und es war das erste Informations-theoretisch sichere Schema (vgl. Kapitel 1.1 „Sicherheits-Definitionen und Bedeutung der Kryptologie“).

  Für diese Anforderung muss das Pad zufällig sein und es darf nur einmal benutzt werden (um jede Ähnlichkeit von Patterns aus dem Geheimtext zu eliminieren).
  Begründung: Gegeben sei Geheimtext C, Klartext P, Pad K, und zwei Klartexte, die jeweils mit dem gleichen Schlüssel verschlüsselt sind: $C1 = P1 \oplus K$; $C2 = P2 \oplus K$;
  Damit ergibt sich $C1 \oplus C2 = (P1 \oplus K) \oplus (P2 \oplus K) = P1 \oplus P2$.
  Diese Kombination kann die Klartexte „durchsickern“ lassen.[29]

- **Superposition** (spezielle Varianten des OTP)
  - **Running-Key-Chiffre**: Addition eines Schlüsseltextes (z.B. aus einem Buch) zum Klartext.
  - **Überlagerung mit einer Zahlenfolge**: Eine Möglichkeit sind mathematische Folgen wie die Fibonacci-Zahlen.

- **Phillips-Chiffre** [ACA02]: Das Alphabet wird in eine 5x5-Matrix eingetragen. Dann werden 7 weitere Matrizen erzeugt, indem zunächst immer die erste, dann die zweite Zeile um eine Position nach unten verschoben wird. Der Klartext wird in Blöcke der Länge 5 unterteilt, die jeweils mit Hilfe einer Matrix verschlüsselt werden. Dazu wird jeweils der Buchstabe rechts unterhalb des Klartextzeichens verwendet.

- **Ragbaby-Chiffre** [ACA02]: Zuerst wird ein Alphabet mit 24 Zeichen konstruiert. Die Zeichen des Klartextes werden durchnummeriert, wobei die Nummerierung der Zeichen des ersten Wortes mit 1 beginnt, die des zweiten Wortes mit 2 usw. Die Zahl 25 entspricht wieder der Zahl 1. Ein Buchstabe der Nachricht wird chiffriert, indem man im Alphabet entsprechend viele Buchstaben nach rechts geht.

  Siehe Tabelle 2.15.

  Alphabet: SCHLUEABDFGIKMNOPQRTVWXZ

[27] In größerem Umfang wurden OTPs von Amerikanern und Briten im Rahmen des „Venona“-Projektes erfolgreich dechiffriert – aufgrund von Chiffrierfehlern der sowjetischen Spione. Siehe `https://de.wikipedia.org/wiki/VENONA-Projekt`.

[28] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Vernam / OTP** aufrufen.
In CT2 findet sich dieses Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch**.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Algorithmen \ Klassisch \ XOR**.
In Kapitel 8.3.1 finden Sie eine ausführliche Darstellung von OTP als Bitstrom-Chiffre und die Implementierung in SageMath.

[29] In JCT kann man über den Menüeintrag **Visualisierungen \ Viterbi** mit der automatischen Kryptoanalyse von Running-Key-Geheimtexten unter Verwendung der Viterbi-Analyse spielen. Und sehen wie erstaunlich es ist, wenn sich aus den geXOR-ten Chiffraten oder den geXOR-ten Klartexten nach und nach beide ursprünglichen Klartexte gewinnen lassen.

Klartext:	d a s	a l p h a b e t	w e c h s e l t	s t a e n d i g
Nummerierung:	1 2 3	2 3 4 5 6 7 8 9	3 4 5 6 7 8 9 10	4 5 6 7 8 9 10 11
Geheimtext:	F D L	D A V B K N M U	S F A D B M K E	U S K K X Q W W

Tabelle 2.15: Ragbaby-Chiffre

2.3 Kombination aus Substitution und Transposition

In der Geschichte der Kryptographie sind häufig Kombinationen der oben angeführten Verfahrensklassen anzutreffen. Diese haben sich - im Durchschnitt - als sicherer erwiesen als Verfahren, die nur auf einem der Prinzipien Transposition oder Substitution beruhen.

- **ADFG(V)X**[30] [Sin01]: Die ADFG(V)X-Verschlüsselung wurde in Deutschland im ersten Weltkrieg entwickelt. Eine 5x5- oder 6x6-Matrix wird mit dem Alphabet gefüllt, die Spalten und Zeilen werden mit den Buchstaben ADFG(V)X versehen. Jedes Klartextzeichen wird durch das entsprechende Buchstabenpaar ersetzt. Abschließend wird auf dem so entstandenen Text eine (Zeilen-)Transposition durchgeführt.
- **Zerlegung von Buchstaben, auch Fractionation genannt** [Sav99]: Sammelbegriff für die Verfahren, die erst ein Klartextzeichen durch mehrere Geheimtextzeichen verschlüsseln und auf diese Verschlüsselung dann eine Transposition anwenden, so dass die ursprünglich zusammengehörenden Geheimtextzeichen voneinander getrennt werden.
  - **Bifid/Polybius square/Checkerboard** [Goe14]: Bei der Grundform dieser Verschlüsselungsmethode wird eine 5x5-Matrix mit den Buchstaben des Alphabets gefüllt (siehe Playfair). Die Spalten und Zeilen dieser Matrix müssen durchnummeriert sein, damit jedes Zeichen des Klartextes durch ein Ziffernpaar (Zeile/Spalte) ersetzt werden kann. Die Zahlen können jede beliebige Permutation von (1,2,3,4,5,) sein. Dies ist ein „Schlüssel“ oder Konfigurations-Parameter dieses Verfahrens. Eine mögliche Variante besteht darin, Schlüsselwörter statt der Zahlen 1 bis 5 zu verwenden. Meist wird der Klartext vorher in Blöcke gleicher Länge zerlegt. Die Blocklänge (hier 5) ist ein weiterer Konfigurations-Parameter diese Verfahrens. Um den Geheimtext zu erhalten, werden zunächst alle Zeilennummern, dann alle Spaltennummern eines Blocks ausgelesen. Anschließend werden die Ziffern paarweise in Buchstaben umgewandelt. Siehe Tabelle 2.16.

	2	4	**5**	1	3
1	S	C	H	L	U
4	E	A	B	D	F
2	G	I	**K**	M	N
3	O	P	Q	R	T
5	V	W	X	Y	Z

Klartext:	**k**ombi	**n**atio	**n**enme	**h**rere	**r**verf	**a**hren
Zeilen:	**2**3242	24323	24224	13434	35434	41342
Spalten:	**5**2154	34342	32312	51212	12213	45123

Tabelle 2.16: Bifid-Chiffre

23242 52154 24323 34342 24224 32312 13434 51212 35434 12213 41342 45123

Geheimtext: NIKMW IOTFE IGFNS UFBSS QFDGU DPIYN

[30] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ ADFGVX** aufrufen.
In CT2 findet sich dieses Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch**.

– **Trifid** [Sav99]: 27 Zeichen (Alphabet + 1 Sonderzeichen) können durch Tripel aus den Ziffern 1 bis 3 repräsentiert werden. Die zu verschlüsselnde Botschaft wird in Blöcke der Länge 3 zerlegt und unter jeden Buchstaben wird das ihm entsprechende Zahlentripel geschrieben. Die Zahlen unter den Blöcken werden wiederum als Tripel zeilenweise ausgelesen und durch entsprechende Zeichen ersetzt.

- **Bazeries-Chiffre** [ACA02]: Eine 5x5-Matrix wird spaltenweise mit dem Alphabet gefüllt, eine zweite Matrix wird zeilenweise mit dem Schlüssel (einer ausgeschriebene Zahl kleiner als 1.000.000) und den übrigen Buchstaben des Alphabets gefüllt. Der Text wird in beliebige Blöcke unterteilt, die Reihenfolge dieser Zeichen wird jeweils umgekehrt und zu jedem Zeichen entsprechend seiner Position in der ersten Matrix sein Gegenstück in der Schlüsselmatrix gesucht.

  Siehe Tabelle 2.17.

  Klartext: kombinationen mehrerer verfahren
  Schlüsselwort: 900.004 (neunhunderttausendundvier)

a	f	l	q	v	N	E	U	H	D
b	g	**m**	r	w	R	T	**A**	S	V
c	h	n	s	x	I	B	C	F	G
d	i	o	t	y	K	L	M	O	P
e	k	p	u	z	Q	W	X	Y	Z

kom	bi	nation	enm	ehr	ere	rverf	ahr	en
mok	ib	noitan	mne	rhe	ere	frevr	rha	ne
AMW	LR	CMLONC	ACQ	SBQ	QSQ	ESQDS	SBN	CQ

Tabelle 2.17: Bazeries-Chiffre

- **Digrafid-Chiffre** [ACA02]: Zur Substitution der Digramme wird die nachfolgende Matrix benutzt (der Einfachheit halber wird das Alphabet hier in seiner ursprünglichen Reihenfolge verwendet). Der erste Buchstabe eines Digramms wird im waagerechten Alphabet gesucht, notiert wird die Nummer der Spalte. Der zweite Buchstabe wird im senkrechten Alphabet gesucht, notiert wird die Nummer der Zeile. Zwischen diese beiden Ziffern wird die Ziffer des Schnittpunktes gesetzt. Die Tripel werden vertikal unter die Digramme, die in Dreierblöcken angeordnet sind, geschrieben. Dann werden die horizontal entstandenen dreistelligen Zahlen ausgelesen und in Buchstaben umgewandelt.

  Bemerkung:
  Da das Verfahren immer ganze Dreiergruppen benötigt, ist eine vollständige Verfahrensbeschreibung zwischen Sender und Empfänger notwendig, die auch den Umgang mit Texten erläutert, die im letzten Block nur 1-5 Klartextbuchstaben enthalten. Vom Weglassen bis Padding mit zufällig oder fix vorher festgelegten Buchstaben ist alles möglich.

  Siehe Tabelle 2.18.

- **Nicodemus-Chiffre** [ACA02]: Zunächst wird eine einfache Spaltentransposition durchgeführt. Noch vor dem Auslesen erfolgt eine Vigenère-Verschlüsselung (die Buchstaben einer Spalte werden mit dem entsprechenden Zeichen des Schlüsselwortes chiffriert). Das Auslesen erfolgt in vertikalen Blöcken.

  Siehe Tabelle 2.19.

1	**2**	3	4	5	6	7	8	9				
A	B	C	D	E	F	G	H	I	1	2	3	
J	**K**	L	M	N	O	P	Q	R	4	**5**	6	
S	T	U	V	W	X	Y	Z	.	7	8	9	
									A	J	S	1
									B	K	T	2
									C	L	U	3
									D	M	V	4
									E	N	W	5
									F	**O**	X	**6**
									G	P	Y	7
									H	Q	Z	8
									I	R	.	9

ko	mb	in	at	io	ne	nm	eh	re	re	rv	er	fa	hr	en
2	4	9	1	9	5	5	5	9	9	9	5	6	8	5
5	4	2	3	2	4	5	1	4	4	6	2	1	2	2
6	2	5	2	6	5	4	8	5	5	4	9	1	9	5
KI	NB	FN	SW	CM	KW	NR	ED	VN	.W	MT	NI	XN	AK	SW

Tabelle 2.18: Digrafid-Chiffre

Klartext: kombinationen mehrerer verfahren

K	E	Y	E	K	Y	E	K	Y
k	o	m	o	k	m	S	U	K
b	i	n	i	b	n	M	L	L
a	t	i	t	a	i	X	K	G
o	n	e	n	o	e	R	Y	C
n	m	e	m	n	e	Q	X	C
h	r	e	r	h	e	V	R	C
r	e	r	e	r	r	I	B	P
v	e	r	e	v	r	I	F	P
f	a	h	a	f	h	E	P	F
r	e	n	e	r	n	I	B	L

Tabelle 2.19: Nicodemus-Chiffre

Geheimtext: SMXRQ ULKYX KLGCC VIIEI RBFPB CPPFL

- **Doppelte Spaltentransposition (DST) / „Granit E160“** [Dro15]:
  Granit ist ein 2-stufiges Verfahren. Der zweite Schritt ist der Doppelwürfel; zuvor wird im ersten Schritt der Klartext m. H. eines Codebuchs und einer Matrix (einer Variante des Polybios-Quadrats) in eine Ziffernfolge verwandelt.

  Unter anderem nutzte der Spion Günter Guillaume das Granit-Verfahren bis etwa 1960 für seine Kommunikation mit dem Ministerium für Staatssicherheit der ehemaligen DDR.[31]

[31] In MTC3 finden sich dazu auch Challenges. Wenn Sie unter

2.4 Andere Verfahren

- **Nadelstich-Verschlüsselung** [Sin01]: Dieses simple Verfahren wurde aus den unterschiedlichsten Gründen über viele Jahrhunderte hinweg praktiziert (eigentlich ein Verfahren der Steganografie). So markierten zum Beispiel im Viktorianischen Zeitalter kleine Löcher unter Buchstaben in Zeitungsartikeln die Zeichen des Klartextes, da das Versenden einer Zeitung sehr viel billiger war als das Porto für einen Brief.

- **Lochschablone**: Die Lochschablone ist auch unter der Bezeichnung Kardinal-Richelieu-Schlüssel bekannt. Eine Schablone wird über einen vorher vereinbarten Text gelegt und die Buchstaben, die sichtbar bleiben, bilden den Geheimtext.

- **Kartenspiele** [Sav99]: Der Schlüssel wird mit Hilfe eines Kartenspiels und vorher festgelegter Regeln erzeugt. Alle im Folgenden genannten Verfahren sind als Papier- und Bleistiftverfahren ausgelegt, also ohne elektronische Hilfsmittel durchführbar. Ein Kartenspiel ist für Außenstehende unverdächtig, das Mischen der Karten bietet ein gewisses Maß an Zufälligkeit, die Werte der Karten lassen sich leicht in Buchstaben umwandeln und Transpositionen lassen sich ohne weitere Hilfsmittel (sei es schriftlich oder elektronisch) durchführen.

  - **Solitaire-Chiffre (Bruce Schneier)**[32] [Sch99]: Sender und Empfänger der Botschaft müssen jeweils ein Kartenspiel besitzen, bei dem alle Karten in der gleichen Ausgangsanordnung im Stapel liegen. Mit den Karten wird ein Schlüsselstrom erzeugt, der ebenso viele Zeichen besitzen muss wie der zu verschlüsselnde Klartext.
    Als Basis zur Erzeugung des Schlüssels wird ein gemischtes Bridge-Kartenspiel mit 54 Karten (Ass, 2 - 10, Bube, Dame, König in vier Farben + 2 Joker) benutzt. Der Kartenstapel wird dazu offen in die Hand genommen:
    1. Der erste Joker wandert um eine Position nach hinten.
    2. Der zweite Joker wandert um zwei Positionen nach hinten.
    3. Die Karten über dem ersten Joker werden mit den Karten unter dem zweiten Joker vertauscht.
    4. Der Wert der untersten Karte (1 bis 53; Kreuz, Karo, Herz, Pik; Joker = 53) wird notiert. Genau so viele Karten werden von oben abgezählt und mit den übrigen Karten vertauscht, wobei die unterste Karte des Stapels liegen bleibt.
    5. Der Wert der obersten Karte wird notiert. Entsprechend viele Karten werden von oben abgezählt.
    6. Der Wert der darauf folgenden Karte ist das erste Schlüsselzeichen, Kreuz und Herz = 1 bis 13, Karo und Pik = 14 bis 26. Handelt es sich um einen Joker, wird wieder bei Schritt 1 begonnen.

    Diese 6 Schritte werden für jedes neue Schlüsselzeichen abgearbeitet. Dieser Vorgang dauert – von Hand – relativ lange (4 h für 300 Zeichen, je nach Übung) und erfordert hohe Konzentration.

`https://www.mysterytwisterc3.org/de/challenges/die-vier-kryptographie-challenge-level?showAll=1` im Browser nach „Granit“ suchen, finden Sie 6 Challenges dazu.
Eine 20-seitige ausführliche Beschreibung des Granit-Verfahrens finden Sie unter:
`https://www.mysterytwisterc3.org/images/challenges/mtc3-drobick-01-doppelwuerfel-01-de.pdf`

[32] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Solitaire** aufrufen.
In CT2 findet sich dieses Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch** und unter **Kryptoanalyse \ Klassisch**.

Die Verschlüsselung erfolgt dann durch Addition mod 26. Sie geht im Vergleich zur Schlüsselstromerzeugung relativ rasch.

Dieses P&B-Verfahren erzeugt einen so guten Schlüsselstrom, dass es auch heutzutage nur schwer zu knacken ist, wenn man das ursprünglich sortierte Kartenspiel nicht kennt (Ciphertext-only-Angriff).

- **Mirdek-Chiffre (Paul Crowley)** [Cro00]: Hierbei handelt es sich um ein relativ kompliziertes Verfahren, der Autor liefert aber anhand eines Beispiels eine sehr anschauliche Erklärung.

- **Playing Card-Chiffre (John Savard)** [Sav99]: Dieses Verfahren verwendet ein bereits gemischtes Kartenspiel ohne Joker, wobei das Mischen gesondert geregelt ist. Ein Schlüssel wird erzeugt ,indem:

  1. Der Stapel liegt verdeckt vor dem Anwender, der solange Karten aufdeckt und in einer Reihe ablegt, bis die Summe der Werte größer oder gleich 8 ist.
  2. Ist die letzte Karte ein Bube, eine Dame oder ein König, wird der Wert dieser Karte notiert, andernfalls notiert man die Summe der Werte der ausgelegten Karten (eine Zahl zwischen 8 und 17). In einer zweiten Reihe wird nun die notierte Anzahl von Karten ausgelegt.
  3. Die dann noch verbleibenden Karten werden reihenweise abgelegt und zwar in der ersten Reihe bis zur Position der niedrigsten Karte aus 2., in der zweiten Reihe bis zur Position der zweitniedrigsten Karte aus 2. usw. Sind 2 Karten gleich, ist rot niedriger zu bewerten als schwarz.
  4. Die in 3. ausgeteilten Karten werden spaltenweise eingesammelt und auf einem Stapel offen abgelegt (Spaltentransposition). Dabei wird mit der Spalte unter der niedrigsten Karte begonnen.(aufgedeckt)
  5. Die in 1. und 2. ausgeteilten Karten werden eingesammelt (die letzte Karte wird zuerst entfernt).
  6. Der Stapel wird umgedreht, so dass die Karten verdeckt liegen. Im Anschluss werden die Schritte 1 bis 6 noch zweimal ausgeführt.

  Um ein Schlüsselzeichen zu erzeugen, wird der Wert der ersten Karte, die nicht Bube, Dame oder König ist, notiert und die entsprechende Anzahl Karten abgezählt (auch der Wert dieser Karte muss zwischen 1 und 10 liegen). Die gleichen Schritte werden ausgehend von der letzten Karte angewendet. Die Werte der beiden ausgewählten Karten werden addiert und die letzte Stelle dieser Summe ist das Schlüsselzeichen.

- **VIC-Chiffre** [Sav99]: Dies ist ein extrem aufwändiges, aber verhältnismäßig sicheres Papier- und Bleistiftverfahren. Es wurde von sowjetischen Spionen eingesetzt. Unter anderem musste der Anwender aus einem Datum, einem Satzanfang und einer beliebigen fünfstelligen Zahl nach bestimmten Regeln zehn Pseudozufallszahlen erzeugen. Bei der Verschlüsselung findet unter anderem auch ein Straddling Checkerboard Verwendung. Eine genaue Beschreibung der Vorgehensweise findet sich unter [Sav99].

2.5 Anhang: Beispiele mit SageMath

Im Folgenden werden einige der klassischen Verfahren mit dem Open-Source Computer-Algebra-System `SageMath`[33] implementiert. Der Code ist lauffähig und getestet mit SageMath Version 5.3. Alle Verfahren sind im Kapitel 2 („Papier- und Bleistift-Verschlüsselungsverfahren“) erklärt.

Um die Beispielprogramme[34] zu den Verschlüsselungsverfahren leichter lesbar zu machen, hielten wir uns an die in der folgenden Abbildung 2.1 aufgeführten Bezeichnungen und Abläufe:

- Die Verschlüsselung besteht aus den Schritten codieren und verschlüsseln.
  - Beim Codieren werden im Klartext P die Zeichen auf Groß-/Kleinschreibung angepasst (je nach gegebenem Alphabet), und alle Nichtalphabetzeichen werden entfernt.
  - Durch die Verschlüsselung wird das Chiffrat C erzeugt.
- Die Entschlüsselung besteht ebenfalls aus den zwei Schritten entschlüsseln und decodieren.
  - Ein abschließender Decodier-Schritt ist nur nötig, wenn die benutzten Symbole im Alphabet nicht die ASCII-Zeichen sind.

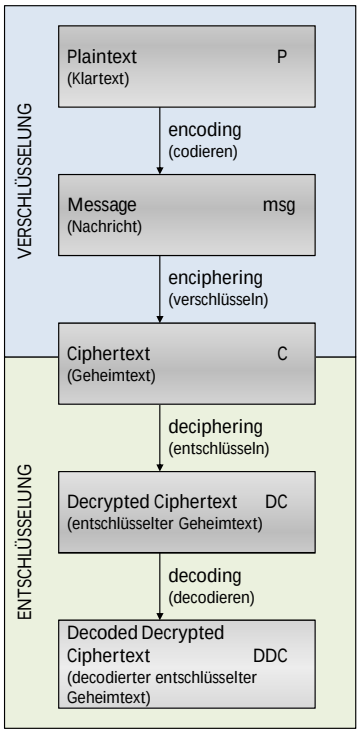


Abbildung 2.1: Namenskonventionen in den SageMath-Programmbeispielen

[33] Eine erste Einführung in das CAS SageMath finden Sie im Anhang A.7.

[34] Weitere Programmbeispiele in SageMath zu klassischen Kryptoverfahren finden sich z.B.:
- als PDF in `http://doc.sagemath.org/pdf/en/reference/cryptography/cryptography.pdf`
- als HTML unter `http://doc.sagemath.org/html/en/reference/cryptography/index.html`
- auf `http://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/classical.html`
- in der Diplomarbeit von Minh Van Nguyen [Ngu09]

2.5.1 Transpositions-Chiffren

Tranpositions-Chiffren sind implementiert in der SageMath-Klasse

`sage.crypto.classical.TranspositionCryptosystem`

Bevor man mit einer SageMath Transpositions-Chiffre arbeiten kann, muss man sie konstruieren: Dazu legt man das Alphabet fest, das sie Symbole (Zeichen) enthält, aus denen Klartext und Geheimtext bestehen können. Typischerweise besteht das Alphabet aus den normalen lateinischen Großbuchstaben. Dieses Alphabet legt man mit der folgenden Funktion fest

`sage.monoids.string_monoid.AlphabeticStrings`

Danach muss man die Blocklänge der Permutation festlegen, d.h. die Länge des Zeilenvektors der einfachen Spaltentransposition. Dieser Zeilenvektor ist der Schlüssel, mit dem die Buchstaben des Klartextes permutiert werden.

Im folgenden ersten Beispiel der Transpositions-Chiffren ist die Blocklänge 14, und der Schlüssel ist so gebaut, dass jeder Buchstabe im Klartext um zwei Zeichen nach rechts geshiftet wird (und wrap-around am Ende des Blocks). Das ist der Verschlüsselungsvorgang. Der Entschlüsselungsvorgang besteht im Shiften jedes Zeichens des Geheimtextes um $14 - 2 = 12$ Zeichen nach links.

SageMath-Beispiel 2.1 Einfache Transposition durch Shiften (die Schlüssel sind explizit gegeben)

```
sage: # transposition cipher using a block length of 14
sage: T = TranspositionCryptosystem(AlphabeticStrings(), 14)
sage: # given plaintext
sage: P   = "a b c d e f g h i j k l m n"
sage: # encryption key
sage: key = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1, 2]
sage:
sage: # encode plaintext (get rid of non-alphabet chars, convert lower-case to upper-case)
sage: msg = T.encoding(P)
sage: # encrypt plaintext by shifting to the left by 2 letters (do it in two steps)
sage: E   = T(key)
sage: C   = E(msg); C
CDEFGHIJKLMNAB
sage:
sage: # decrypt ciphertext by shifting to the left by 12 letters
sage: keyInv = [13, 14, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: D   = T(keyInv)
sage: D(C)
ABCDEFGHIJKLMN
sage:
sage: # Representation of key and inverse key as permutations
sage: E
(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)
sage: D
(1,13,11,9,7,5,3)(2,14,12,10,8,6,4)
```

Das zweite Beispiel der Transpositions-Chiffren ist ebenfalls eine einfache shiftende Spaltentransposition. Aber hier ist der Programmcode ein wenig mehr automatisiert: Die Schlüssel werden anhand des Shift-Wertes generiert.

SageMath-Beispiel 2.2 Einfache Transposition durch Shiften (die Schlüssel werden mit „range“ konstruiert)

```
sage: # transposition cipher using a block length of 14, code more variable
sage: keylen = 14
sage: shift = 2
sage: A = AlphabeticStrings()
sage: T = TranspositionCryptosystem(A, keylen)
sage:
sage: # construct the plaintext string from the first 14 letters of the alphabet plus blanks
sage: # plaintext   = "A B C D E F G H I J K L M N"
sage: A.gens()
(A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z)
sage: P=''
sage: for i in range(keylen): P=P + " " + str(A.gen(i))
....:
sage: P
' A B C D E F G H I J K L M N'
sage:
sage: # encryption key
sage: # key = [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1, 2]
sage: key = [(i+shift).mod(keylen) + 1 for i in range(keylen)]; key
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1, 2]
sage:
sage: # encode plaintext (get rid of non-alphabet chars)
sage: msg = T.encoding(P)
sage: # encrypt plaintext by shifting to the left by 2 letters (do it in one step)
sage: C   = T.enciphering(key, msg); C
CDEFGHIJKLMNAB
sage:
sage: # decrypt ciphertext by shifting to the left by 12 letters
sage: # keyInv = [13, 14, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: shiftInv=keylen-shift;
sage: keyInv = [(i+shiftInv).mod(keylen) + 1 for i in range(keylen)]; keyInv
[13, 14, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: DC   = T.enciphering(keyInv, C); DC
ABCDEFGHIJKLMN
sage:
sage: # decryption using the "deciphering method with key" instead of "enciphering with keyInv"
sage: # using the deciphering method requires to change the type of the variable key
sage: DC  = T.deciphering(T(key).key(), C); DC
ABCDEFGHIJKLMN
sage:
sage: # representation of key and inverse key as permutations
sage: T(key)
(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)
sage: T(key).key()
(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)
sage: T(keyInv)
(1,13,11,9,7,5,3)(2,14,12,10,8,6,4)
```

Im dritten Beispiel der Transpositions-Chiffren wird eine beliebige Permutation als Schlüssel für die Ver- und Entschlüsselung gewählt, um die Zeichen in jedem Block zu verwürfeln (Blocklänge = Anzahl Spalten in der einfachen Spaltentransposition). Ist die Blocklänge n, dann ist der Schlüssel eine Permutation über n Zeichen. Das folgende Beispiel nutzt die Methode `random_key()` der Klasse `TranspositionCryptosystem`. Jeder Aufruf von `random_key()` erzeugt einen anderen Schlüssel. Deshalb werden bei Ihrem Aufruf voraussichtlich andere Ergebnisse (Schlüssel und Geheimtext) auftreten.

SageMath-Beispiel 2.3 Einfache Spalten-Transposition mit zufällig erzeugtem Schlüssel

```
sage: # Remark: Enciphering here requires, that the length of msg is a multiple of keylen
sage: keylen = 14   # length of key
sage: A = AlphabeticStrings()
sage: T = TranspositionCryptosystem(A, keylen); T
Transposition cryptosystem on Free alphabetic string monoid on A-Z of block length 14
sage:
sage: P = "a b c d e f g h i j k l m n o p q r s t u v w x y z a b"
sage: key = T.random_key(); key
(1,2,3,13,6,5,4,12,7)(11,14)
sage: msg = T.encoding(P); msg
ABCDEFGHIJKLMNOPQRSTUVWXYZAB
sage: C   = T.enciphering(key, msg); C
BCMLDEAHIJNGFKPQAZRSOVWXBUTY
sage: # decryption using the "deciphering method with key" instead of "enciphering with keyInv"
ssage: DC  = T.deciphering(key, C); DC
ABCDEFGHIJKLMNOPQRSTUVWXYZAB
sage:
sage: # Just another way of decryption: Using "enciphering" with the inverse key
sage: keyInv = T.inverse_key(key); keyInv
(1,7,12,4,5,6,13,3,2)(11,14)
sage: DC     = T.enciphering(keyInv, C); DC
ABCDEFGHIJKLMNOPQRSTUVWXYZAB
sage:
sage: # Test correctness of decryption
sage: msg == DC
True
```

Das vierte Beispiel der Transpositions-Chiffren gibt zusätzlich die Größe des Schlüsselraums einer einfachen Spaltentransposition aus.

SageMath-Beispiel 2.4 Einfache Spalten-Transposition (mit Ausgabe der Größe des Schlüsselraumes)

```
sage: keylen = 14   # length of key
sage: A = AlphabeticStrings()
sage: T = TranspositionCryptosystem(A, keylen); T
Transposition cryptosystem on Free alphabetic string monoid on A-Z of block length 14
sage: T.key_space()
Symmetric group of order 14! as a permutation group
sage: # Remark: The key space is not quite correct as also permutations shorter than keylen are counted.
sage:
sage: P = "a b c d e f g h i j k l m n o p q r s t u v w x y z a b"
sage: key = T.random_key(); key
(1,2,7)(3,9)(4,5,10,12,8,13,11)(6,14)
sage: msg = T.encoding(P); msg
ABCDEFGHIJKLMNOPQRSTUVWXYZAB
sage:
sage: # enciphering in one and in two steps
sage: C   = T.enciphering(key, msg); C
BGIEJNAMCLDHKFPUWSXBOAQZRVYT
sage:
sage: enc = T(key); enc.key()
(1,2,7)(3,9)(4,5,10,12,8,13,11)(6,14)
sage: C = enc(msg); C
BGIEJNAMCLDHKFPUWSXBOAQZRVYT
sage:
sage: # deciphering
sage: DC  = T.deciphering(key, C); DC
ABCDEFGHIJKLMNOPQRSTUVWXYZAB
```

2.5.2 Substitutions-Chiffren

Substitutions-Verschlüsselungen sind in SageMath in der folgenden Klasse implementiert

`sage.crypto.classical.SubstitutionCryptosystem`

Das folgende Programmbeispiel lässt SageMath eine Substitutions-Chiffre mit einem zufälligen Schlüssel erstellen. Den zufälligen Schlüssel kann man mit der Methode `random_key()` der Klasse `SubstitutionCryptosystem` erzeugen. Unterschiedliche Schlüssel erzeugen eine unterschiedliche Substitutions-Chiffre. Deshalb erhält man mit jedem Aufruf von `random_key()` voraussichtlich ein anderes Ergebnis.

SageMath-Beispiel 2.5 Monoalphabetische Substitution mit zufällig erzeugtem Schlüssel

```
sage: # plaintext/ciphertext alphabet
sage: A   = AlphabeticStrings()
sage: S   = SubstitutionCryptosystem(A)
sage:
sage: P   = "Substitute this with something else better."
sage: key = S.random_key(); key
INZDHFUXJPATQOYLKSWGVECMRB
sage:
sage: # method encoding can be called from A or from T
sage: msg = A.encoding(P); msg
SUBSTITUTETHISWITHSOMETHINGELSEBETTER
sage: C   = S.enciphering(key, msg); C
WVNWGJGVGHGXJWCJGXWYQHGXJOUHTWHNHGGHS
sage:
sage: # We now decrypt the ciphertext to recover our plaintext.
sage:
sage: DC  = S.deciphering(key, C); DC
SUBSTITUTETHISWITHSOMETHINGELSEBETTER
sage: msg == DC
True
```

2.5.2.1 Caesar-Chiffre

Das folgende Programmbeispiel erzeugt eine Caesar-Chiffre.

SageMath-Beispiel 2.6 Caesar (Substitution durch Shiften des Alphabets; Schlüssel explizit gegeben; Schritt-für-Schritt-Ansatz)

```
sage: # plaintext/ciphertext alphabet
sage: A = AlphabeticStrings()
sage: P = "Shift the alphabet three positions to the right."
sage:
sage: # construct Caesar cipher
sage: S = SubstitutionCryptosystem(A)
sage: key = A([3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, \
....:          20, 21, 22, 23, 24, 25, 0, 1, 2])
sage:
sage: # encrypt message
sage: msg     = A.encoding(P); msg
SHIFTTHEALPHABETTHREEPOSITIONSTOTHERIGHT
sage: encrypt = S(key); encrypt
DEFGHIJKLMNOPQRSTUVWXYZABC
sage: C       = encrypt(msg); C
VKLIWWKHDOSKDEHWWKUHHSRVLWLRQVWRWKHULJKW
sage:
sage: # Next, we recover the plaintext.
sage: # decrypt message
sage: keyInv = A([23, 24, 25, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \
....:             14, 15, 16, 17, 18, 19, 20, 21, 22])
sage: decrypt = S(keyInv); decrypt
XYZABCDEFGHIJKLMNOPQRSTUVW
sage: DC      = decrypt(C); DC
SHIFTTHEALPHABETTHREEPOSITIONSTOTHERIGHT
sage: msg == DC
True
```

Das zweite Caesar-Beispiel macht dasselbe, aber der Programmcode ist variabler.

SageMath-Beispiel 2.7 Caesar (Substitution durch Shiften des Alphabets; Substitutions-Schlüssel wird berechnet)

```
sage: # plaintext/ciphertext alphabet
sage: A = AlphabeticStrings()
sage: keylen = len(A.gens()); keylen
26
sage: shift  = 3
sage: P = "Shift the alphabet three positions to the right."
sage:
sage: # construct Caesar cipher
sage: S = SubstitutionCryptosystem(A)
sage: S
Substitution cryptosystem on Free alphabetic string monoid on A-Z
sage: # key = A([3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, \
sage: #          20, 21, 22, 23, 24, 25, 0, 1, 2])
sage: key = [(i+shift).mod(keylen) for i in range(keylen)];
sage: key = A(key); key
DEFGHIJKLMNOPQRSTUVWXYZABC
sage: len(key)
26
sage:
sage: # encrypt message
sage: msg     = A.encoding(P); msg
SHIFTTHEALPHABETTHREEPOSITIONSTOTHERIGHT
sage: C       = S.enciphering(key, msg); C
VKLIWWKHDOSKDEHWWKUHHSRVLWLRQVWRWKHULJKW
sage:
sage: # Next, we recover the plaintext.
sage: # decrypt message
sage: # keyInv = A([23, 24, 25, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \
sage: #             14, 15, 16, 17, 18, 19, 20, 21, 22])
sage: shiftInv=keylen-shift;
sage: keyInv = [(i+shiftInv).mod(keylen) for i in range(keylen)];
sage: keyInv = A(keyInv); keyInv
XYZABCDEFGHIJKLMNOPQRSTUVW
sage: DC     = S.enciphering(keyInv, C); DC
SHIFTTHEALPHABETTHREEPOSITIONSTOTHERIGHT
sage:
sage: # Just another way of decryption: Using "deciphering" with the key
sage: DC     = S.deciphering(key, C); DC
SHIFTTHEALPHABETTHREEPOSITIONSTOTHERIGHT
sage:
sage: msg == DC
True
```

2.5.2.2 Verschiebe-Chiffre

Die Verschiebe-Chiffre kann man sich auch als Verallgemeinerung der ursprünglichen Caesar-Chiffre denken: Caesar shiftete das Alphabet immer genau um 3 Positionen. Die Shift-Chiffre kann um eine beliebige Anzahl Positionen entlang des Alphabets shiften. Shiften ist eine spezielle Form der Substitution.

In den obigen Beispielen wurde das `SubstitutionCryptosystem` angewandt, und Caesar wurde als Sonderfall der Substitution implementiert. Andererseits kann man Caesar auch Spezialfall der Shift-Chiffre betrachten.

Die Shift-Chiffre ist direkt implementiert in der SageMath-Klasse

`sage.crypto.classical.ShiftCryptosystem`

Im folgenden Beispiel konstruieren wir eine Shift-Chiffre über den Großbuchstaben des lateinischen Alphabets. Dann verschlüsseln wir den Klartext P durch Shiften um 12 Positionen. Schließlich entschlüsseln wir wieder das Chiffrat C und überprüfen, dass das Ergebnis (DC) identisch mit dem ursprünglichen Klartext ist.

SageMath-Beispiel 2.8 Verschiebe-Chiffre (über dem Großbuchstabenalphabet)

```
sage: # construct Shift cipher directly
sage: shiftcipher = ShiftCryptosystem(AlphabeticStrings()); shiftcipher
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: P = shiftcipher.encoding("Shift me any number of positions."); P
SHIFTMEANYNUMBEROFPOSITIONS
sage: key = 12  # shift can be any integer number
sage: # shift the plaintext by 12 positions to get the ciphertext
sage: C = shiftcipher.enciphering(key, P); C
ETURFYQMZKZGYNQDARBAEUFUAZE
sage:
sage: # decrypt the ciphertext and ensure that it is the original plaintext
sage: DC = shiftcipher.deciphering(key, C); DC
SHIFTMEANYNUMBEROFPOSITIONS
sage: DC == P
True
```

Die ursprüngliche Caesar-Chiffre ist eine Shift-Chiffre mit festem Schlüssel 3. Im nächsten Beispiel wird die Caesar-Chiffre über den Großbuchstaben des lateinischen Alphabets gebaut.

SageMath-Beispiel 2.9 Caesar-Verschlüsselung mit der Verschiebe-Chiffre

```
sage: caesarcipher = ShiftCryptosystem(AlphabeticStrings())
sage: P = caesarcipher.encoding("Shift the alphabet by three positions to the right."); P
SHIFTTHEALPHABETBYTHREEPOSITIONSTOTHERIGHT
sage:
sage: key = 3  # shift the plaintext by exactly 3 positions
sage: C = caesarcipher.enciphering(key, P); C
VKLIWWKHDOSKDEHWEBWKUHHSRVLWLRQVWRWKHULJKW
sage:
sage: # decrypt the ciphertext and ensure that it is the original plaintext
sage: DC = caesarcipher.deciphering(key, C); DC
SHIFTTHEALPHABETBYTHREEPOSITIONSTOTHERIGHT
sage: DC == P
True
```

2.5.2.3 Affine Chiffren

Die affine Chiffre ist implementiert in der SageMath-Klasse

`sage.crypto.classical.AffineCryptosystem`

Im folgenden Beispiel konstruieren wir eine affine Chiffre $c_i = b * p_i + a$ mit dem Schlüssel $(3, 13)$ und benutzen sie, um einen gegebenen Klartext $P = (p_1, p_2, ..., p_n)$ zu verschlüsseln. Der Klartext wird dann wieder entschlüsselt und das Ergebnis DC mit dem ursprünglichen Klartext verglichen.

SageMath-Beispiel 2.10 Affine Chiffre mit dem Schlüssel $(3, 13)$

```
sage: # create an affine cipher
sage: affineCipher = AffineCryptosystem(AlphabeticStrings()); affineCipher
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = affineCipher.encoding("The affine cryptosystem.")
sage: P
THEAFFINECRYPTOSYSTEM
sage:
sage: # encrypt the plaintext using the key (3, 13)
sage: a, b = (3, 13)
sage: C = affineCipher.enciphering(a, b, P)
sage: C
SIZNCCLAZTMHGSDPHPSZX
sage:
sage: # decrypt the ciphertext and make sure that it is equivalent to the original plaintext
sage: DC = affineCipher.deciphering(a, b, C)
sage: DC
THEAFFINECRYPTOSYSTEM
sage: DC == P
True
```

Man kann die Shift-Chiffre auch als Sonderfall der affinen Chiffre sehen. Dazu muss man den Schlüssel der affinen Chiffre auf die folgende Form $(1, b)$ einschränken, wobei b jede ganze, nicht-negative Zahl sein kann. Das SageMath-Beispiel 2.8 auf Seite 55 kann man wie folgt darstellen:

SageMath-Beispiel 2.11 Verschiebe-Chiffre (als Sonderfall der affinen Chiffre)

```
sage: # construct a shift cipher
sage: shiftcipher = AffineCryptosystem(AlphabeticStrings()); shiftcipher
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = shiftcipher.encoding("Shift me any number of positions.")
sage: P
SHIFTMEANYNUMBEROFPOSITIONS
sage:
sage: # shift the plaintext by 12 positions to get the ciphertext
sage: a, b = (1, 12)
sage: C = shiftcipher.enciphering(a, b, P)
sage: C
ETURFYQMZKZGYNQDARBAEUFUAZE
sage:
sage: # decrypt the ciphertext and ensure that it is the original plaintext
sage: DC = shiftcipher.deciphering(a, b, C); P
SHIFTMEANYNUMBEROFPOSITIONS
sage: DC == P
True
```

Man kann mit der affinen Chiffre auch eine Caesar-Chiffre bauen. Dazu muss der Ver-/Entschlüsselungsschlüssel den Wert $(1, 3)$ haben. Das SageMath-Beispiel 2.9 auf Seite 55 kann mit der affinen Chiffre folgendermaßen dargestellt werden.

SageMath-Beispiel 2.12 Caesar-Chiffre (als Sonderfall der affinen Chiffre)

```
sage: # create a Caesar cipher
sage: caesarcipher = AffineCryptosystem(AlphabeticStrings())
sage: P = caesarcipher.encoding("Shift the alphabet by three positions to the right.")
sage: P
SHIFTTHEALPHABETBYTHREEPOSITIONSTOTHERIGHT
sage:
sage: # shift the plaintext by 3 positions
sage: a, b = (1, 3)
sage: C = caesarcipher.enciphering(a, b, P)
sage: C
VKLIWWKHDOSKDEHWEBWKUHHSRVLWLRQVWRWKHULJKW
sage:
sage: # decrypt the ciphertext and ensure that it is the original plaintext
sage: DC = caesarcipher.deciphering(a, b, C)
sage: DC
SHIFTTHEALPHABETBYTHREEPOSITIONSTOTHERIGHT
sage: DC == P
True
```

2.5.2.4 Substitutions-Chiffre mit Symbolen

Im folgenden SageMath-Programmbeispiel sind die Alphabetzeichen aus dem Binärsystem. Eine monoalphabetische Substitution über einem Binäralphabet bietet nur wenig Sicherheit: Weil das Klartext-/Geheimtext-Alphabet nur aus den 2 Elementen 0 und 1 besteht, gibt es nur zwei mögliche Schlüssel: (0 1) und (1 0).
Anmerkung: Im Schlüssel einer allgemeinen Substitutions-Chiffre müssen alle Alphabetzeichen genau einmal auftreten.

SageMath-Beispiel 2.13 Monoalphabetische Substitution über dem Binär-Alphabet

```
sage: # the plaintext/ciphertext alphabet
sage: B = BinaryStrings()
sage: # substitution cipher over the alphabet B; no keylen argument possible
sage: S = SubstitutionCryptosystem(B); S
Substitution cryptosystem on Free binary string monoid
sage: # To get a substitute for each symbol, key has always the length of the alphabet
sage: key = S.random_key(); key
10
sage: len(key)
2
sage: P = "Working with binary numbers."
sage: # encryption
sage: msg = B.encoding(P); msg
010101110110111101110010011010110110100101101110011001110010000001110111011101\
001011101000110100000100000011000100110100101101110011000010111001001111001000\
100000011011100111010101101101101100010011001010111001001110011001001011110
sage: C   = S.enciphering(key, msg); C
101010001001000010001101100101001001011010010001100110001101111110001000100100\
110100010111001011111011111110011101100101101001000110011111010001101100001101\
011111100100011000101010010010100111011001101010001101100011001101000 1
sage: # decryption
sage: DC  = S.deciphering(key, C); DC
010101110110111101110010011010110110100101101110011001110010000001110111011101\
001011101000110100000100000011000100110100101101110011000010111001001111001000\
100000011011100111010101101101101100010011001010111001001110011001001011110
sage: msg == DC
True
```

Anmerkung: Im Moment hat `S` kein Attribut `key`, und ich fand keine Möglichkeit, den Binärstring `DC` wieder in ASCII zurück zu verwandeln.

Das zweite Beispiel einer monoalphabetischen Substitution mit Symbolen benutzt ein größeres Alphabet für den Klartext-/Geheimtext-Zeichenraum als das erste Beispiel. Hier wird nun das hexadezimale Zahlensystem als Substitutions-Alphabet verwendet.

SageMath-Beispiel 2.14 Monoalphabetische Substitution über dem Hexadezimal-Alphabet (Dekodieren in Python)

```
sage: A = HexadecimalStrings()
sage: S = SubstitutionCryptosystem(A)
sage: key = S.random_key(); key
2b56a4e701c98df3
sage: len(key)
16
sage: # Number of possible keys
sage: factorial(len(key))
20922789888000
sage: P   = "Working with a larger alphabet."
sage:
sage: msg = A.encoding(P); msg
576f726b696e67207769746820612061206c617267657220616c7068616265742e
sage: C   = S.enciphering(key, msg); C
47e375e9e1efe75277e17ae052eb52e8eb75e7e47552ebe872e0ebe5e47a5f
sage: DC  = S.deciphering(key, C); DC
576f726b696e67207769746820612061206c617267657220616c7068616265742e
sage: msg == DC
True
sage:
sage: # Conversion hex back to ASCII:
sage: # - AlphabeticStrings() and HexadecimalStrings() don't have according methods.
sage: # - So we used Python directly.
sage: import binascii
sage: DDC = binascii.a2b_hex(repr(DC)); DDC
'Working with a larger alphabet.'
sage:
sage: P == DDC
True
```

2.5.2.5 Vigenère-Verschlüsselung

Die Vigenère-Verschlüsselung ist in der folgenden SageMath-Klasse implementiert

`sage.crypto.classical.VigenereCryptosystem`

Als Klartext-/Geheimtext-Zeichenraum kann man die lateinischen Großbuchstaben, das Binärsystem, das Oktalsystem oder Hexadezimalsystem wählen. Hier ist ein Beispiel, das mit der Klasse `AlphabeticStrings` die Großbuchstaben nutzt.

SageMath-Beispiel 2.15 Vigenère-Verschlüsselung

```
sage: # construct Vigenere cipher
sage: keylen = 14
sage: A = AlphabeticStrings()
sage: V = VigenereCryptosystem(A, keylen); V
Vigenere cryptosystem on Free alphabetic string monoid on A-Z of period 14
sage:
sage: # alternative could be a given key: key = A('ABCDEFGHIJKLMN'); key
sage: key = V.random_key(); key
WSSSEEGVVAARUD
sage: len(key)
14
sage:
sage: # encoding
sage: P = "The Vigenere cipher is polyalphabetic."
sage: len(P)
38
sage: msg = V.encoding(P); msg     # alternative: msg = A.encoding(P); msg
THEVIGENERECIPHERISPOLYALPHABETIC
sage:
sage: # encryption [2 alternative ways (in two steps or in one): both work]
sage: # encrypt = V(key); encrypt
sage: # C = encrypt(msg); C
sage: C   = V.enciphering(key, msg); C
PZWNMKKIZRETCSDWJAWTUGTALGBDXWLAG
sage:
sage: # decryption
sage: DC  = V.deciphering(key, C); DC
THEVIGENERECIPHERISPOLYALPHABETIC
sage: msg == DC
True
```

2.5.3 Hill-Verschlüsselung

Die Hill [Hil29, Hil31]- oder Matrix-Verschlüsselung[35] ist mathematisch anspruchsvoller als die Verfahren in den bisherigen Programmbeispielen dieses Kapitels. Der Schlüssel dieses Verfahrens ist eine invertierbare quadratische Matrix (hier key genannt). Klartext und Geheimtext sind Vektoren (P und C). Die Ver- und Entschlüsselungsprozesse nutzen Matrizen-Operationen modulo 26, hier also $C = P * key \ (mod\ 26)$.

Die Hill-Verschlüsselung ist implementiert in der SageMath-Klasse

`sage.crypto.classical.HillCryptosystem`

Im folgenden Beispiel besteht der Klar-/Geheimtext-Zeichenraum wieder aus den Großbuchstaben des lateinischen Alphabets. Im Hill-Verfahren wird jedem Alphabet-Zeichen eine eindeutige natürliche Zahl modulo 26 zugewiesen. Die Größe der Schlüssel-Matrix (auch Matrix-Dimension genannt) ist vom Hill-Verfahren selbst nicht beschränkt.

Bemerkung: Vergleich der Hill-Implementierung in CrypTool v1.4.31 und in SageMath Version 5.3:

- SageMath bietet schnelle Operationen auf der Kommandozeile; CT1 stellt seine Funktionalität nur in einer GUI zur Verfügung.
- SageMath bietet für die Schlüssel-Matrix jede Matrix-Dimension an; CT1 ist beschränkt auf die Werte 1 bis 10.
- SageMath erlaubt auch negative Zahlen in der Schlüssel-Matrix und konvertiert diese selbstständig in nicht-negative Werte; CT1 erlaubt keine negativen Zahlen in der Schlüssel-Matrix.
- SageMath weist dem ersten Alphabet-Zeichen immer den Wert 0 zu;
  SageMath lässt hierbei nur die 26 Großbuchstaben als Alphabet zu;
  und es nutzt nur die Multiplikationsvariante Klartext-Zeilenvektor * Schlüsselmatrix:
  $C = P * key$.
- CT1 erlaubt die Wahl zwischen 0 und 1 für das erste Alphabet-Zeichen; man kann sein Alphabet im Textoptionen-Dialog zusammen stellen; und es bietet auch die umgekehrte Multiplikationsvariante: $C = key * P$

[35] In CT1 kann man dieses Verfahren über den Menüeintrag **Ver-/Entschlüsseln \ Symmetrisch (klassisch) \ Hill** aufrufen.
In CT2 findet sich dieses Verfahren bei den Vorlagen unter **Kryptographie \ Klassisch** und unter **Kryptoanalyse \ Klassisch**.

SageMath-Beispiel 2.16 Hill-Verschlüsselung mit einer zufällig generierten Schlüsselmatrix

```
sage: # construct a Hill cipher
sage: keylen = 19    # An Alternative could be: Use a non-random small key (e.g. keylen = 3)
sage: A = AlphabeticStrings(); H = HillCryptosystem(A, keylen); H
Hill cryptosystem on Free alphabetic string monoid on A-Z of block length 19
sage:
sage: # Alternative: Here, HKS is necessary in addition [H.key_space() isn't enough].
sage: # HKS = H.key_space(); key = HKS([[1,0,1],[0,1,1],[2,2,3]]); key
sage:
sage: # Random key creation
sage: key = H.random_key(); key
[10  7  5  2  0  6 10 23 15  7 17 19 18  2  9 12  0 10 11]
[23  1  1 10  4  9 21  1 25 22 19  8 17 22 15  8 12 25 22]
[ 4 12 16 15  1 12 24  5  9 13  5 15  8 21 23 24 22 20  6]
[ 5 11  6  7  3 12  8  9 21 20  9  4 16 18 10  3  2 23 18]
[ 8 22 14 14 20 13 21 19  3 13  2 11 13 23  9 25 25  6  8]
[24 25  8 24  7 18  3 20  6 11 25  5  6 19  7 24  2  4 10]
[15 25 11  1  4  7 11 24 20  2 18  4  9  8 12 19 24  0 12]
[14  6  2  9 11 20 13  4 10 11  4 23 14 22 14 16  9 12 18]
[12 10 21  5 21 15 16 17 19 20  1  1 15  5  0  2 23  4 14]
[21 15 15 16 15 20  4 10 25  7 15  4  7 12 24  9 19 10  6]
[25 15  2  3 17 23 21 16  8 18 23  4 22 11 15 19  6  0 15]
[14 23  9  3 18 15 10 18  7  5 12 23 11  9 22 21 20  4 14]
[ 3  6  8 13 20 16 11  1 13 10  4 21 25 15 12  3  0 11 18]
[21 25 14  6 11  3 21  0 19 17  5  8  5  4  9  2 23 19 15]
[ 8 11  9 11 20 15  6  1  3 18 18 22 16 17  6  3 15 11  2]
[21 15  5 22  2  9  0  4 22 10  2 10 19 19 17 19  1 21  4]
[ 7 17  9  2 15  5 14  3  6  9 12 12 22 15  8  4 21 14 19]
[19 14 24 19  7  5 22 22 13 14  7 18 17 19 25  2  1 23  6]
[ 2  6 14 22 17  7 23  6 22  7 13 20  0 14 23 17  6  1 12]
sage:
sage: # encoding and encryption
sage: P = "Hill or matrix cipher uses matrix operations."; len(P)
45
sage: # implementation requires: Length of msg is a multiple of matrix dimension (block_length)
sage: msg = H.encoding(P); msg; len(msg)
HILLORMATRIXCIPHERUSESMATRIXOPERATIONS
38
sage:
sage: # encryption (the length of msg must be a multiple of keylen).
sage: C  = H.enciphering(key, msg); C
CRWCKPRVYXNBRZTNZCTQWFWSDWBCHABGMNEHVP
sage:
sage: # decryption
sage: DC  = H.deciphering(key, C); DC; msg == DC
HILLORMATRIXCIPHERUSESMATRIXOPERATIONS
True
sage:
sage: # alternative decryption using inverse matrix
sage: keyInv = H.inverse_key(key); keyInv
[ 6 23  1 23  3 12 17 22  6 16 22 14 18  3  1 10 21 16 20]
[18 23 15 25 24 23  7  4 10  7 21  7  9  0 13 22  5  5 23]
...
[10 11 12  6 11 17 13  9 19 16 14 24  4  8  5 16 18 20  1]
[19 16 16 21  1 19  7 12  3 18  1 17  7 10 24 21  7 16 11]
sage: DC     = H.enciphering(keyInv, C); DC
HILLORMATRIXCIPHERUSESMATRIXOPERATIONS
```

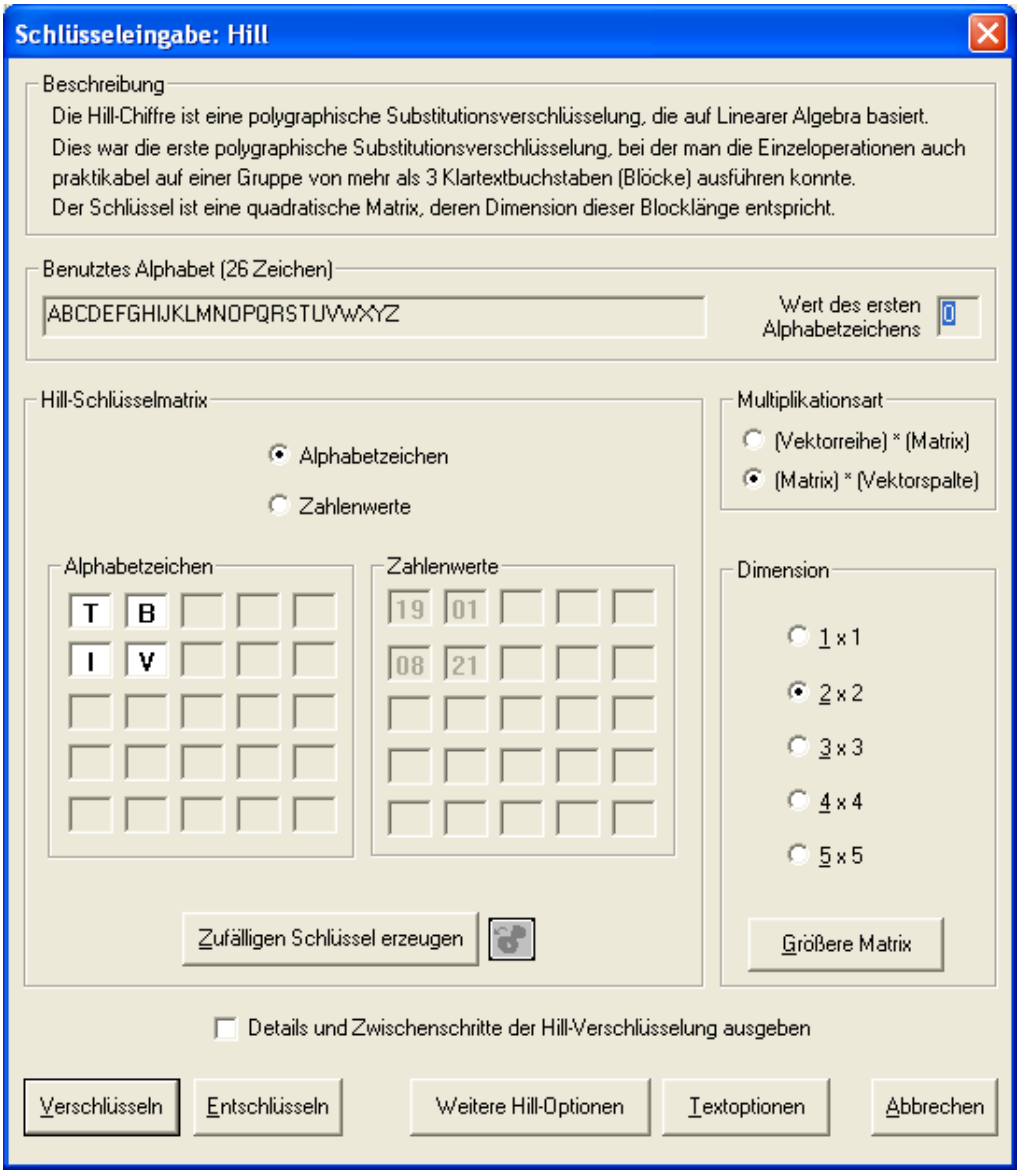


Abbildung 2.2: Hill-Dialog in CT1 mit den verfügbaren Operationen und Optionen

Literaturverzeichnis (Kap. PaP)

[ACA02] ACA: *Length and Standards for all ACA Ciphers.* Technischer Bericht, American Cryptogram Association, 2002.
http://www.cryptogram.org/cdb/aca.info/aca.and.you/chap08.html#,
http://www.und.edu/org/crypto/crypto/.chap08.html.

[Cro00] Crowley, Paul: *Mirdek: A card cipher inspired by "Solitaire"*, 2000. http://www.ciphergoth.org/crypto/mirdek/.

[Dro15] Drobick, Jörg: *Abriss DDR-Chiffriergeschichte: SAS- und Chiffrierdienst*, 2015. http://scz.bplaced.net/m.html#dwa.

[Goe14] Goebel, Greg: *Codes, Ciphers and Codebreaking*, 2014. Version 2.3.2. http://www.vectorsite.net/ttcode.html.

[Hil29] Hill, Lester S.: *Cryptography in an Algebraic Alphabet.* The American Mathematical Monthly, 36(6):306–312, 1929.

[Hil31] Hill, Lester S.: *Concerning Certain Linear Transformation Apparatus of Cryptography.* The American Mathematical Monthly, 38(3):135–154, 1931.

[Ngu09] Nguyen, Minh Van: *Exploring Cryptography Using the Sage Computer Algebra System.* Diplomarbeit, Victoria University, 2009.
http://www.sagemath.org/files/thesis/nguyen-thesis-2009.pdf,
http://www.sagemath.org/library-publications.html.

[Sav99] Savard, John J. G.: *A Cryptographic Compendium*, 1999.
http://www.quadibloc.com/crypto/jscrypt.htm.

[Sch99] Schneier, Bruce: *The Solitaire Encryption Algorithm*, 1999. v. 1.2.
https://www.schneier.com/academic/solitaire/.

[Sin01] Singh, Simon: *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet.* dtv, 2001.

[Thi99] ThinkQuest Team 27158: *Data Encryption*, 1999.

[WLS99] Witten, Helmut, Irmgard Letzner und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. Teil 3: Flusschiffren, perfekte Sicherheit und Zufall per Computer.* LOG IN, 1999(2):50–57, 1999.
http://bscw.schule.de/pub/nj_bscw.cgi/d637156/RSA_u_Co_T3.pdf.

Alle Links wurden am 11.07.2016 überprüft.

Kapitel 3

Primzahlen

(Bernhard Esslinger, Mai 1999; Updates: Nov. 2000, Dez. 2001, Juni 2003, Mai 2005, März 2006, Juni 2007, Jan. 2010, Aug. 2013, Juli 2016, Apr. 2018)

Der Fortschritt lebt vom Austausch des Wissens.

Zitat 6: Albert Einstein[1]

3.1 Was sind Primzahlen?

Primzahlen sind ganze, positive Zahlen größer gleich 2, die man nur durch 1 und durch sich selbst teilen kann. Alle anderen natürlichen Zahlen größer gleich 4 sind zusammengesetzte Zahlen und lassen sich durch Multiplikation von Primzahlen bilden.

Somit bestehen die *natürlichen* Zahlen $\mathbb{N} = \{1, 2, 3, 4, \cdots\}$ aus

- der Zahl 1 (dem Einheitswert)
- den Primzahlen (primes) und
- den zusammengesetzten Zahlen (composite numbers).

Primzahlen haben aus drei Gründen besondere Bedeutung erlangt:

- Sie werden in der Zahlentheorie als die Grundbausteine der natürlichen Zahlen betrachtet, anhand derer eine Menge genialer mathematischer Überlegungen geführt wurden.
- Sie haben in der modernen Kryptographie (Public-Key-Kryptographie) große praktische Bedeutung erlangt. Das verbreiteteste Public-Key-Verfahren ist die Ende der siebziger Jahre erfundene RSA-Verschlüsselung. Nur die Verwendung (großer) Primzahlen für bestimmte Parameter garantiert die Sicherheit des Algorithmus sowohl beim RSA-Verfahren als auch bei noch moderneren Verfahren (z.B. Elliptische Kurven).
- Die Suche nach den größten bekannten Primzahlen hat wohl bisher keine praktische Verwendung, erfordert aber die besten Rechner, gilt als hervorragender Benchmark (Möglichkeit zur Leistungsbestimmung von Computern) und führt zu neuen Formen der Berechnungen auf mehreren Computern (siehe auch: `http://www.mersenne.org/prime.htm`).

[1] Albert Einstein, deutscher Physiker und Nobelpreisträger, 14.03.1879−14.04.1955.

Von Primzahlen ließen sich im Laufe der letzten zwei Jahrtausende sehr viele Menschen faszinieren. Der Ehrgeiz, zu neuen Erkenntnissen über Primzahlen zu gelangen, führte dabei oft zu genialen Ideen und Schlussfolgerungen. Im folgenden wird in einer leicht verständlichen Art in die mathematischen Grundlagen der Primzahlen eingeführt. Dabei klären wir auch, was über die Verteilung (Dichte, Anzahl von Primzahl in einem bestimmten Intervall) der Primzahlen bekannt ist oder wie Primzahltests funktionieren.

3.2 Primzahlen in der Mathematik

Jede ganze Zahl hat Teiler. Die Zahl 1 hat nur einen, nämlich sich selbst. Die Zahl 12 hat die sechs Teiler $1, 2, 3, 4, 6, 12$. Viele Zahlen sind nur durch sich selbst und durch 1 teilbar. Bezüglich der Multiplikation sind dies die „Atome“ im Bereich der Zahlen. Diese Zahlen nennt man Primzahlen.

In der Mathematik ist eine etwas andere (aber äquivalente) Definition üblich.

Definition 3.2.1. *Eine ganze Zahl $p \in \mathbf{N}$ heißt Primzahl , wenn $p > 1$ und p nur die trivialen Teiler ± 1 und $\pm p$ besitzt.*

Per definitionem ist die Zahl 1 keine Primzahl. Im weiteren bezeichnet der Buchstabe p stets eine Primzahl.

Die Primzahlenfolge startet mit

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ 53,\ 59,\ 61,\ 67,\ 71,\ 73,\ 79,\ 83,\ 89,\ 97, \cdots .$$

Unter den ersten 100 Zahlen gibt es genau 25 Primzahlen. Danach nimmt ihr prozentualer Anteil stets ab. Primzahlen können nur auf eine einzige *triviale* Weise zerlegt werden:

$$5 = 1 \cdot 5, \quad 17 = 1 \cdot 17, \quad 1013 = 1 \cdot 1013, \quad 1.296.409 = 1 \cdot 1.296.409.$$

Alle Zahlen, die 2 und mehr von 1 verschiedene Faktoren haben, nennt man *zusammengesetzte* Zahlen. Dazu gehören

$$4 = 2 \cdot 2, \quad 6 = 2 \cdot 3$$

aber auch Zahlen, die *wie Primzahlen aussehen*, aber doch keine sind:

$$91 = 7 \cdot 13, \quad 161 = 7 \cdot 23, \quad 767 = 13 \cdot 59.$$

Die folgende Tabelle vermittelt einen Eindruck, wie Primzahlen in den natürlichen Zahlen verteilt sind. Es gibt auch viele grafische Darstellungsformen (am bekanntesten ist die Ulam-Spirale), diese brachten bisher keinen konkreten Erkenntnisgewinn, doch sie erweckten bei manchen den Eindruck, dass es in der zufälligen Verteilung zumindest lokale Muster gäbe.

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	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270
271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330
331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360
361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390

Abbildung 3.1: Primzahlen unter den ersten 390 natürlichen Zahlen – farblich markiert[2]

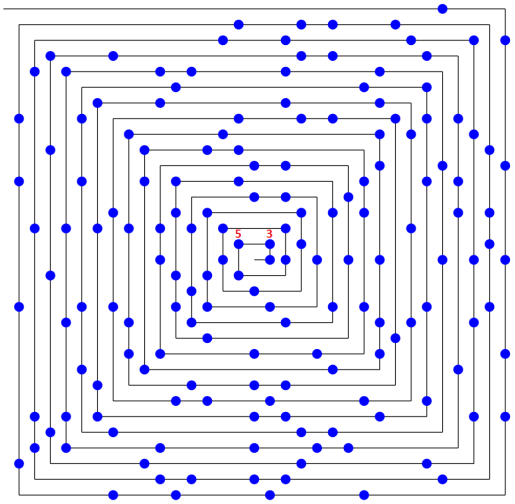


Abbildung 3.2: Primzahlen unter den ersten 999 natürlichen Zahlen – als Ulam-Spirale[3]

[2]Grafik von `http://mathforum.org/mathimages/index.php/Image:Irisprime.jpg`, 30*13-Rechteck
[3]Grafik von CT2, Menü Kryptotutorien, Die Welt der Primzahlen, Verteilung der Primzahlen, Ulam-Spirale; 32*32 Punkte.
[4]Grafik von `http://mathforum.org/mathimages/index.php/Image:Ulam_spiral.png`, 200*200 Ulam-Spirale

Satz 3.2.1. *Jede ganze Zahl m größer als 1 besitzt einen kleinsten Teiler größer als 1. Dieser ist eine Primzahl p. Sofern m nicht selbst eine Primzahl ist, gilt: p ist kleiner oder gleich der Quadratwurzel aus m.*

Aus den Primzahlen lassen sich alle ganzen Zahlen größer als 1 zusammensetzen — und das sogar in einer eindeutigen Weise. Dies besagt der **1. Hauptsatz der Zahlentheorie** (= Hauptsatz der elementaren Zahlentheorie = fundamental theorem of arithmetic = fundamental building block of all positive integers).

Satz 3.2.2. *Jedes Element n größer als 1 der natürlichen Zahlen lässt sich als Produkt $n = p_1 \cdot p_2 \cdot \cdots \cdot p_m$ von Primzahlen schreiben. Sind zwei solche Zerlegungen*

$$n = p_1 \cdot p_2 \cdots p_m = p'_1 \cdot p'_2 \cdots p'_{m'}$$

gegeben, dann gilt nach eventuellem Umsortieren $m = m'$ und für alle i: $p_i = p'_i$.
($p_1, p_2, \ldots, p_m$ nennt man die Primfaktoren von n).

In anderen Worten: Jede natürliche Zahl außer der 1 lässt sich auf genau eine Weise als Produkt von Primzahlen schreiben, wenn man von der Reihenfolge der Faktoren absieht. Die Faktoren sind also eindeutig (die *Expansion in Faktoren* ist eindeutig)! Zum Beispiel ist

$$60 = 2 \cdot 2 \cdot 3 \cdot 5 = 2^2 \cdot 3^1 \cdot 5^1$$

Und das ist — bis auf eine veränderte Reihenfolge der Faktoren — die einzige Möglichkeit, die Zahl 60 in Primfaktoren zu zerlegen. Wenn man nicht nur Primzahlen als Faktoren zulässt, gibt es mehrere Möglichkeiten der Zerlegung in Faktoren und die **Eindeutigkeit** (uniqueness) geht verloren:

$$60 = 1 \cdot 60 = 2 \cdot 30 = 4 \cdot 15 = 5 \cdot 12 = 6 \cdot 10 = 2 \cdot 3 \cdot 10 = 2 \cdot 5 \cdot 6 = 3 \cdot 4 \cdot 5 = \cdots .$$

Der folgende Absatz wendet sich eher an die mit der mathematischen Logik vertrauteren Menschen: Der 1. Hauptsatz ist nur scheinbar selbstverständlich. Man kann viele andere Zahlenmengen (ungleich der positiven ganzen Zahlen größer als 1) konstruieren, bei denen selbst eine Zerlegung in die Primfaktoren dieser Mengen nicht eindeutig ist: In der Menge $M = \{1, 5, 10, 15, 20, \cdots\}$ gibt es unter der Multiplikation kein Analogon zum Hauptsatz. Die ersten fünf Primzahlen dieser Folge sind $5, 10, 15, 20, 30$ (beachte: 10 ist prim, da innerhalb dieser Menge 5 kein Teiler von 10 ist — das Ergebnis 2 ist kein Element der gegebenen Grundmenge M). Da in M gilt:

$$100 = 5 \cdot 20 = 10 \cdot 10$$

und sowohl $5, 10, 20$ Primzahlen dieser Menge sind, ist hier die Zerlegung in Primfaktoren nicht eindeutig.

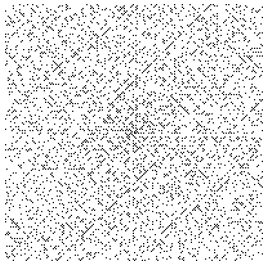


Abbildung 3.3: Primzahlen unter den ersten 4000 natürlichen Zahlen – als Ulam-Spirale[4]

3.3 Wie viele Primzahlen gibt es?

Für die natürlichen Zahlen sind die Primzahlen vergleichbar mit den Elementen in der Chemie oder den Elementarteilchen in der Physik (vgl. [Blu99, S. 22]).

Während es nur 92 natürliche chemische Elemente gibt, ist die Anzahl der Primzahlen unbegrenzt. Das wusste schon der Grieche Euklid[5] im dritten vorchristlichen Jahrhundert.

Satz 3.3.1 (Euklid[6])**.** *Die Folge der Primzahlen bricht nicht ab, es gibt also unendlich viele Primzahlen.*

Sein Beweis, dass es unendlich viele Primzahlen gibt, gilt bis heute als ein Glanzstück mathematischer Überlegung und Schlussfolgerung (**Widerspruchsbeweis**). Er nahm an, es gebe nur endlich viele Primzahlen und damit eine größte Primzahl. Daraus zog er solange logische Schlüsse, bis er auf einen offensichtlichen Widerspruch stieß. Damit musste etwas falsch sein. Da sich in die Schlusskette kein Lapsus eingeschlichen hatte, konnte es nur die Annahme sein. Demnach musste es unendlich viele Primzahlen geben!

Beweis nach Euklid (Widerspruchsbeweis)

Annahme: Es gibt *endlich* viele Primzahlen.

Schluss: Dann lassen sie sich auflisten $p_1 < p_2 < p_3 < \cdots < p_n$, wobei n für die (endliche) Anzahl der Primzahlen steht. p_n wäre also die größte Primzahl. Nun betrachtet Euklid die Zahl $a = p_1 \cdot p_2 \cdots p_n + 1$. Diese Zahl kann keine Primzahl sein, da sie in unserer Primzahlenliste nicht auftaucht. Also muss sie durch eine Primzahl teilbar sein. D.h. es gibt eine natürliche Zahl i zwischen 1 und n, so dass p_i die Zahl a teilt. Natürlich teilt p_i auch das Produkt $a-1 = p_1 \cdot p_2 \cdots p_n$, da p_i ja ein Faktor von $a-1$ ist. Da p_i die Zahlen a und $a-1$ teilt, teilt sie auch die Differenz dieser Zahlen. Daraus folgt: p_i teilt $a-(a-1) = 1$. p_i müsste also 1 teilen und das ist unmöglich.

Widerspruch: Unsere Annahme war falsch.

Also gibt es *unendlich* viele Primzahlen (siehe Übersicht unter 3.9 über die Anzahl von Primzahlen in verschiedenen Intervallen). □

Wir erwähnen hier auch noch eine andere, auf den ersten Blick überraschende Tatsache, dass nämlich in der Folge aller Primzahlen $p_1, p_2, \cdots$ Lücken von beliebig großer Länge n auftreten. Unter den n aufeinanderfolgenden natürlichen Zahlen

$$(n+1)! + 2, \cdots, (n+1)! + (n+1),$$

ist keine eine Primzahl, da ja in ihnen der Reihe nach die Zahlen $2, \cdots, n+1$ als echte Teiler enthalten sind (Dabei bedeutet $n!$ das Produkt der ersten n natürlichen Zahlen, also $n! = n * (n-1) * \cdots * 3 * 2 * 1$).

[5] Euklid, griechischer Mathematiker des 4./3. Jahrhunderts vor Christus. Wirkte an der Akademie in Alexandria und verfasste mit den „Elementen“ das bekannteste systematische Lehrbuch der griechischen Mathematik.

[6] Die üblich gewordene Benennung bedeutet nicht unbedingt, dass Euklid der Entdecker des Satzes ist, da dieser nicht bekannt ist. Der Satz wird bereits in Euklids „Elementen“ (Buch IX, Satz 20) formuliert und bewiesen. Die dortige Formulierung ist insofern bemerkenswert, als sie das Wort „unendlich“ nicht verwendet; sie lautet

Οί πρῶτοι ἀριθμοὶ πλείους εἰσὶ παντὸς τοῦ προτεθέντος πλήθ ους πρώτων ἀριθμῶν,

zu deutsch: Die Primzahlen sind mehr als jede vorgegebene Menge von Primzahlen.

3.4 Die Suche nach sehr großen Primzahlen

Die größten heute bekannten Primzahlen haben mehrere Millionen Stellen.[7] Das ist unvorstellbar groß. Die Anzahl der Elementarteilchen im Universum wird auf eine „nur“ 80-stellige Dezimalzahl geschätzt (siehe Übersicht unter 3.11 über verschiedene Größenordnungen / Dimensionen).

3.4.1 Die 30+ größten bekannten Primzahlen (Stand Jan. 2018)

In der folgenden Tabelle 3.1 sind die größten, derzeit bekannten Primzahlen und eine Beschreibung des jeweiligen Zahlentyps aufgeführt.[8]

[7] In CT1 können Sie über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Mersenne-Zahlen berechnen** alle Stellen einer solch großen Zahl sehr schnell berechnen.

[8] Eine jeweils aktuelle Fassung findet sich im Internet unter `https://primes.utm.edu/primes/search.php?Number=1000`, unter `http://primes.utm.edu/mersenne/index.html`, und unter `http://www.mersenne.org/primes/`.

[9] Diese Zahl wurde am 26.3.2007 im verteilt rechnenden Internet-Projekt „Seventeen or Bust“ (SoB) (`https://de.wikipedia.org/wiki/Seventeen_or_Bust`) gefunden. Anders als das bekannte GIMPS-Projekt (Kapitel 3.4.2), das immer größere der unendlich vielen Primzahlen aufspürt, könnte SoB aber irgendwann mal die gesetzte Aufgabe vollständig erledigt haben.

Das SoB-Projekt versucht rechnerisch zu beweisen, dass die Zahl $k = 78.557$ die kleinste Sierpinski-Zahl ist (John Selfridge bewies 1962, dass 78.557 eine Sierpinski-Zahl ist).

Der berühmte polnische Mathematiker Waclaw Sierpinski (1882 bis 1969) hatte im Jahre 1960 nachgewiesen, dass es unendlich viele ungerade natürliche Zahlen k gibt, die folgende Eigenschaft erfüllen: Für jede Sierpinski-Zahl k gilt: Sämtliche Zahlen $N = k \cdot 2^n + 1$ sind für alle natürlichen $n >= 1$ zusammengesetzte Zahlen (Sierpinski's Composite Number Theorem, `http://mathworld.wolfram.com/SierpinskisCompositeNumberTheorem.html`).

Am Projektanfang im Jahre 2002 gab es noch 17 mögliche Kandidaten < 78557 (daher der Name des Projekts „Seventeen or Bust“). Es reicht, wenn man ein einziges Gegenbeispiel findet, um einen Kandidaten k auszuschließen, also ein einziges $n >= 1$ zu finden, so dass $N = k \cdot 2^n + 1$ prim ist. Dass bei dieser Suche sehr große Primzahlen gefunden werden, ist also eigentlich nur ein „Nebenprodukt“ der Aufgabenstellung.

Vergleiche auch die Meldung vom 10.05.2007: `http://www.heise.de/newsticker/meldung/89582`.

Seit Mitte April 2016 ist der SoB-Server nicht mehr erreichbar und damit die Zukunft des Grundprojektes ungewiss.

[10] Verallgemeinerte Fermatzahl: $1.372.930^{131.072} + 1 = 1.372.930^{(2^{17})} + 1$

	Definition	Dezimalstellen	Wann	Beschreibung
1	$2^{77.232.917} - 1$	23.249.425	2018	Mersenne, 50. bekannte
2	$2^{74.207.281} - 1$	22.338.618	2016	Mersenne, 49. bekannte
3	$2^{57.885.161} - 1$	17.425.170	2013	Mersenne, 48. bekannte
4	$2^{43.112.609} - 1$	12.978.189	2008	Mersenne, M-47
5	$2^{42.643.801} - 1$	12.837.064	2009	Mersenne, M-46
6	$2^{37.156.667} - 1$	11.185.272	2008	Mersenne, M-45
7	$2^{32.582.657} - 1$	9.808.358	2006	Mersenne, M-44
8	$2^{30.402.457} - 1$	9.152.052	2005	Mersenne, M-43
9	$2^{25.964.951} - 1$	7.816.230	2005	Mersenne, M-42
10	$2^{24.036.583} - 1$	7.235.733	2004	Mersenne, M-41
11	$2^{20.996.011} - 1$	6.320.430	2003	Mersenne, M-40
12	$2^{13.466.917} - 1$	4.053.946	2001	Mersenne, M-39
13	$19.249 \cdot 2^{13.018.586} + 1$	3.918.990	2007	Verallgem. Mersenne[9]
14	$3 \cdot 2^{11.895.718} - 1$	3.580.969	2015	Verallgem. Mersenne
15	$3 \cdot 2^{11.731.850} - 1$	3.531.640	2015	Verallgem. Mersenne
16	$3 \cdot 2^{11.484.018} - 1$	3.457.035	2014	Verallgem. Mersenne
17	$3 \cdot 2^{10.829.346} + 1$	3.259.959	2014	Verallgem. Mersenne
18	$475.856^{524.288} + 1$	2.976.633	2012	Verallgem. Fermat
19	$356.926^{524.288} + 1$	2.911.151	2012	Verallgem. Fermat
20	$341.112^{524.288} + 1$	2.900.832	2012	Verallgem. Fermat
21	$27.653 \cdot 2^{9.167.433} + 1$	2.759.677	2005	Verallgem. Mersenne
22	$90.527 \cdot 2^{9.162.167} + 1$	2.758.093	2010	Verallgem. Mersenne
23	$2.038 \cdot 366^{1.028.507} - 1$	2.636.562	2016	Verallgem. Fermat
24	$75.898^{524.288} + 1$	2.558.647	2011	Verallgem. Fermat
25	$28.433 \cdot 2^{7.830.457} + 1$	2.357.207	2004	Verallgem. Mersenne
26	$502.573 \cdot 2^{7.181.987} - 1$	2.162.000	2014	Verallgem. Mersenne
27	$402.539 \cdot 2^{7.173.024} - 1$	2.159.301	2014	Verallgem. Mersenne
28	$161.041 \cdot 2^{7.107.964} + 1$	2.139.716	2015	Verallgem. Mersenne
29	$3 \cdot 2^{7.033.641} + 1$	2.117.338	2011	Verallgem. Mersenne
30	$33.661 \cdot 2^{7.031.232} + 1$	2.116.617	2007	Verallgem. Mersenne
31	$2^{6.972.593} - 1$	2.098.960	1999	Mersenne, M-38
...				
329	$1.372.930^{131.072} + 1$	804.474	2003	Verallgem. Fermat[10]
...				
343	$342.673 \cdot 2^{2.639.439} - 1$	794.556	2007	Verallgem. Mersenne

Tabelle 3.1: Die 30+ größten Primzahlen und ihr jeweiliger Zahlentyp (Stand Jan. 2018)

Die zeitliche Entwicklung wird in Abbildung 3.4 verdeutlicht. Sie startet mit M21701 (gefunden 1978). Bitte die logarithmische vertikale Skala beachten.

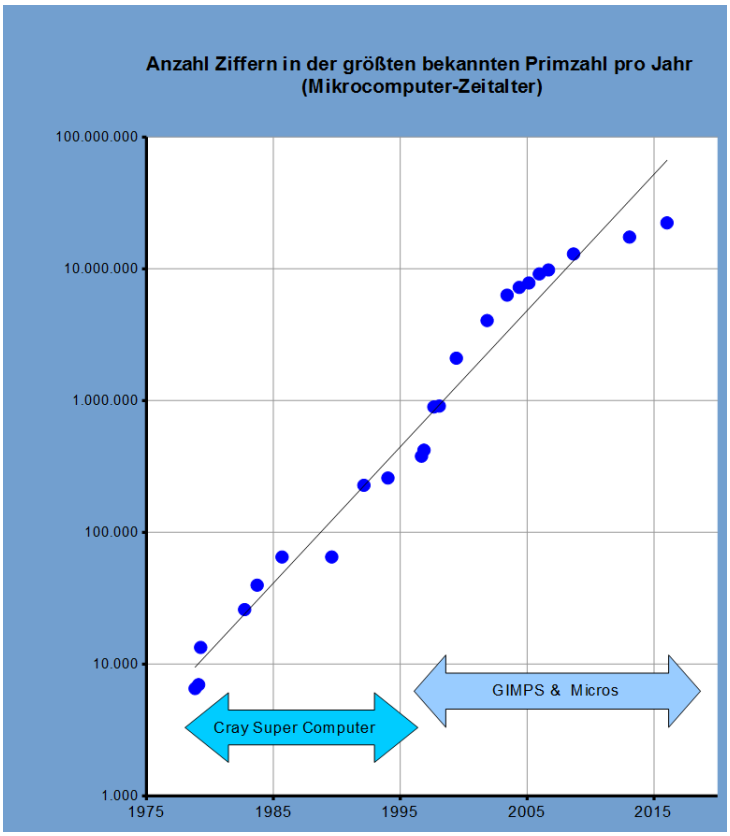


Abbildung 3.4: Anzahl Ziffern der größten bekannten Primzahl nach Jahren (Stand Juli 2016)[11]

Die größte, momentan bekannte Primzahl ist eine Mersenne-Primzahl. Diese wurde vom GIMPS-Projekt (Kapitel 3.4.2) gefunden. Unter den größten bekannten Primzahlen befinden sich außerdem Zahlen vom Typ verallgemeinerte Mersennezahl (Kapitel 3.6.2) und vom Typ verallgemeinerte Fermatzahl (Kapitel 3.6.5).

[11] Quelle: Chris Caldwell, `http://primes.utm.edu/notes/by_year.html`

3.4.2 Spezielle Zahlentypen – Mersennezahlen und Mersenne-Primzahlen

Nahezu alle bekannten riesig großen Primzahlen sind spezielle Kandidaten, sogenannte *Mersennezahlen*[12] der Form $2^p - 1$, wobei p eine Primzahl ist. Nicht alle Mersennezahlen sind prim:

$$\begin{aligned} 2^2 - 1 = 3 &\quad \Rightarrow \text{prim} \\ 2^3 - 1 = 7 &\quad \Rightarrow \text{prim} \\ 2^5 - 1 = 31 &\quad \Rightarrow \text{prim} \\ 2^7 - 1 = 127 &\quad \Rightarrow \text{prim} \\ 2^{11} - 1 = 2.047 = 23 \cdot 89 &\quad \Rightarrow \text{NICHT prim!} \end{aligned}$$

Dass Mersennezahlen nicht immer Primzahlen (Mersenne-Primzahlen) sind, wusste auch schon Mersenne (siehe Exponent $p = 11$). Eine Mersennezahl, die prim ist, wird Mersenne-Primzahl genannt.

Dennoch ist ihm der interessante Zusammenhang zu verdanken, dass eine Zahl der Form $2^n - 1$ keine Primzahl sein kann, wenn n eine zusammengesetzte Zahl ist:

Satz 3.4.1 (Mersenne)**.** *Wenn $2^n - 1$ eine Primzahl ist, dann folgt, n ist ebenfalls eine Primzahl (oder anders formuliert: $2^n - 1$ ist nur dann prim, wenn n prim ist).*

Beweis
Der Beweis des Satzes von Mersenne kann durch Widerspruch durchgeführt werden. Wir nehmen also an, dass es eine zusammengesetzte natürliche Zahl n mit echter Zerlegung $n = n_1 \cdot n_2$ gibt, mit der Eigenschaft, dass $2^n - 1$ eine Primzahl ist.

Wegen

$$\begin{aligned} (x^r - 1)((x^r)^{s-1} + (x^r)^{s-2} + \cdots + x^r + 1) &= ((x^r)^s + (x^r)^{s-1} + (x^r)^{s-2} + \cdots + x^r) \\ &\quad -((x^r)^{s-1} + (x^r)^{s-2} + \cdots + x^r + 1) \\ &= (x^r)^s - 1 = x^{rs} - 1, \end{aligned}$$

folgt

$$2^{n_1 n_2} - 1 = (2^{n_1} - 1)((2^{n_1})^{n_2 - 1} + (2^{n_1})^{n_2 - 2} + \cdots + 2^{n_1} + 1).$$

Da $2^n - 1$ eine Primzahl ist, muss einer der obigen beiden Faktoren auf der rechte Seite gleich 1 sein. Dies kann nur dann der Fall sein, wenn $n_1 = 1$ oder $n_2 = 1$ ist. Dies ist aber ein Widerspruch zu unserer Annahme. Deshalb ist unsere Annahme falsch. Also gibt es keine zusammengesetzte Zahl n, so dass $2^n - 1$ eine Primzahl ist. □

Leider gilt dieser Satz nur in einer Richtung (die Umkehrung gilt nicht, keine Äquivalenz): Das heißt, dass es prime Exponenten gibt, für die die zugehörige Mersennezahl **nicht** prim ist (siehe das obige Beispiel $2^{11} - 1$, wo 11 prim ist, aber $2^{11} - 1$ nicht).

Mersenne behauptete, dass $2^{67} - 1$ eine Primzahl ist. Auch zu dieser Behauptung gibt es eine interessante mathematische Historie: Zuerst dauerte es über 200 Jahre, bis Edouard Lucas (1842-1891) bewies, dass diese Zahl zusammengesetzt ist. Er argumentierte aber indirekt und kannte keinen der Faktoren. 1903 zeigte Cole[13], aus welchen Faktoren diese Primzahl besteht:

$$2^{67} - 1 = 147.573.952.589.676.412.927 = 193.707.721 \cdot 761.838.257.287.$$

[12] Marin Mersenne, französischer Priester und Mathematiker, 08.09.1588−01.09.1648.
[13] Frank Nelson Cole, amerikanischer Mathematiker, 20.09.1861−26.05.1926.

Er gestand, 20 Jahre an der **Faktorisierung** (Zerlegung in ein Produkt aus Primfaktoren)[14] dieser 21-stelligen Dezimalzahl gearbeitet zu haben!

Dadurch, dass man bei den Exponenten der Mersennezahlen nicht alle natürlichen Zahlen verwendet, sondern nur die Primzahlen, engt man den *Versuchsraum* deutlich ein. Die derzeit bekannten 50 Mersenne-Primzahlen gibt es für die folgenden Exponenten[15]

2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1.279, 2.203, 2.281, 3.217, 4.253, 4.423, 9.689, 9.941, 11.213, 19.937, 21.701, 23.207, 44.497, 86.243, 110.503, 132.049, 216.091, 756.839, 859.433, 1.257.787, 1.398.269, 2.976.221, 3.021.377, 6.972.593, 13.466.917, 20.996.011, 24.036.583, 25.964.951, 30.402.457, 32.582.657, 37.156.667, 42.643.801, 43.112.609, 57.885.161, 74.207.281, 77.232.917.

Die 19. Zahl mit dem Exponenten 4.253 war die erste mit mindestens 1.000 Stellen im Zehnersystem (der Mathematiker Samual Yates prägte dafür den Ausdruck *titanische* Primzahl; sie wurde 1961 von Hurwitz gefunden); die 27. Zahl mit dem Exponenten 44.497 war die erste mit mindestens 10.000 Stellen im Zehnersystem (Yates prägte dafür den Ausdruck *gigantische* Primzahl. Diese Bezeichnungen sind heute längst veraltet).

Für die ersten 47 Mersenne-Primzahlen weiß man inzwischen, dass diese Liste vollständig ist. Die Exponenten bis zur 50. bekannten Mersenne-Primzahl sind noch nicht vollständig geprüft.[16]

Inzwischen (Stand 2018-04-23) wurden alle primen Exponenten kleiner als 43.261.403 getestet und nochmal geprüft[17]: Somit können wir sicher sein, dass dies wirklich die 47. Mersenne-

---

[14] Mit CT1 können Sie Zahlen auf folgende Weise faktorisieren: Menü **Einzelverfahren \ RSA-Kryptosystem \ Faktorisieren einer Zahl**.
In sinnvoller Zeit zerlegt CT1 mit dem Quadratischen Sieb (QS) auf einem Einzel-PC Zahlen bis 250 Bit Länge. Zahlen größer als 1024 Bit werden zur Zeit von CT1 sowieso nicht angenommen.

CT2 hat die Komponente GeneralFactorizer (basierend auf YAFU). Diese ist schneller als die in CT1 implementierten Funktionen. Damit bietet CT2 die folgenden Faktorisierungsverfahren:

- Brute-force mit kleinen Primzahlen
- Fermat
- Shanks Square Forms Factorization (squfof)
- Pollard rho
- Pollard p-1
- Williams p+1
- Lenstra Elliptic Curve Method (ECM)
- Self-initializing Quadratic Sieve (SIQS)
- Multiple-polynomial Quadratic Qieve (MPQS)
- Special Number Field Sieve (SNFS)
- General Number Field Sieve (GNFS).

CT2 hat begonnen, mit einer allgemeinen Infrastruktur für verteiltes Rechnen zu experimentieren (CrypCloud, die sowohl Peer-to-Peer als auch zentralisiert eingesetzt werden kann). Damit wird CT2 in Zukunft in die Lage versetzt, Berechnungen auf viele Computer zu verteilen. Was man erreichen kann, wenn die Komponenten für die Parallelisierung eingerichtet sind, zeigte ein Cluster zur verteilten Kryptoanalyse von DES und AES: Stand 21. März 2016 funktionierte ein Brute-force-Angriff (verteilte Schlüsselsuche) gegen AES auf 50 i5-PCs, jeder mit 4 virtuelles CPU-Kernen. Diese 200 virtuellen „Worker Threads“ konnten ca. 350 Millionen AES-Schlüssel/sec testen. Die „Cloud“ verarbeitete dabei insgesamt ca. 20 GB/sec an Daten. CrypCloud ist eine Volunteering-Cloud, so dass CT2-Nutzer sich freiwillig bei verteilten Jobs anschließen können.
Die aktuellen Faktorisierungsrekorde finden Sie in Kapitel 4.11.4.

[15] Landon Curt Noll listet in einer Tabelle alle bekannten Mersenne-Primzahlen samt Entdeckungsdatum und Wert in Zahlen- und Wortform auf: `http://www.isthe.com/chongo/tech/math/prime/mersenne.html`
Siehe auch: `http://www.utm.edu/`.

[16] Den aktuellen Status der Prüfung findet man auf der Seite: `http://www.mersenne.org/primenet/`.
Hinweise, wie man Zahlen auf ihre Primalität prüfen kann, finden sich auch in Kapitel 3.5, Primzahltests.

[17] Siehe die Homepage des GIMPS-Projekts: `http://www.mersenne.org/report_milestones`.

Primzahl ist und dass keine kleineren unentdeckten Mersenne-Primzahlen existieren (es ist üblich, die Bezeichnung M-nn erst dann zu verwenden, wenn die nn. bekannte Mersenne-Primzahl auch bewiesenermaßen die nn. Mersenne-Primzahl ist).

Hier einige Beispiele mit ein paar mehr Details:

M-37 – Januar 1998

Die 37. Mersenne-Primzahl,

$$2^{3.021.377} - 1$$

wurde im Januar 1998 gefunden und hat 909.526 Stellen im Zehnersystem, was 33 Seiten in der FAZ entspricht!

M-38 – Juni 1999

Die 38. Mersenne-Primzahl, genannt M-38,

$$2^{6.972.593} - 1$$

wurde im Juni 1999 gefunden und hat 2.098.960 Stellen im Zehnersystem (das entspricht rund 77 Seiten in der FAZ).

M-39 – Dezember 2001

Die 39. Mersenne-Primzahl, genannt M-39,

$$2^{13.466.917} - 1$$

wurde am 6.12.2001 bekanntgegeben: genau genommen war am 6.12.2001 die Verifikation der am 14.11.2001 von dem kanadischen Studenten Michael Cameron gefundenen Primzahl abgeschlossen. Diese Zahl hat rund 4 Millionen Stellen (genau 4.053.946 Stellen). Allein zu ihrer Darstellung

$$924947738006701322247758 \cdots 1130073855470256259071$$

bräuchte man in der FAZ knapp 200 Seiten.

GIMPS

Das GIMPS-Projekt (Great Internet Mersenne-Prime Search) wurde 1996 von George Woltman gegründet, um neue größte Mersenne-Primzahlen zu finden (`http://www.mersenne.org`). Genauere Erläuterungen zu diesem Zahlentyp finden sich unter Mersennezahlen und Mersenne-Primzahlen.

Bisher hat das GIMPS-Projekt 16 größte Mersenne-Primzahlen entdeckt, inklusive der größten bekannten Primzahl überhaupt.

Tabelle 3.2 enthält diese Mersenne Rekord-Primzahlen.[18,19]

[18] Eine up-to-date gehaltene Version dieser Tabelle steht im Internet unter `http://www.mersenne.org/history.htm`.

[19] Bei jedem neuen Rekord, der gemeldet wird, beginnen in den einschlägigen Foren die immer gleichen, oft ironischen Diskussionen: Hat diese Forschung einen tieferen Sinn? Lassen sich diese Ergebnisse für irgendwas verwenden? Die Antwort ist, dass das noch unklar ist. Aber gerade das ist bei Grundlagenforschung normal, dass man nicht sofort sieht, ob und wie es die Menschheit voranbringt.

	Definition	Dezimalstellen	Wann	Wer
1	$2^{77.232.917} - 1$	23.249.425	26. Dez. 2017	Jonathan Pace
2	$2^{74.207.281} - 1$	22.338.618	7. Jan. 2016	Curtis Cooper
3	$2^{57.885.161} - 1$	17.425.170	25. Jan. 2013	Curtis Cooper
4	$2^{43.112.609} - 1$	12.978.189	23. Aug. 2008	Edson Smith
5	$2^{42.643.801} - 1$	12.837.064	12. Apr. 2009	Odd Magnar Strindmo
6	$2^{37.156.667} - 1$	11.185.272	6. Sep. 2008	Hans-Michael Elvenich
7	$2^{32.582.657} - 1$	9.808.358	4. Sep. 2006	Curtis Cooper/Steven Boone
8	$2^{30.402.457} - 1$	9.152.052	15. Dez. 2005	Curtis Cooper/Steven Boone
9	$2^{25.964.951} - 1$	7.816.230	18. Feb. 2005	Martin Nowak
10	$2^{24.036.583} - 1$	7.235.733	15. Mai 2004	Josh Findley
11	$2^{20.996.011} - 1$	6.320.430	17. Nov. 2003	Michael Shafer
12	$2^{13.466.917} - 1$	4.053.946	14. Nov. 2001	Michael Cameron
13	$2^{6.972.593} - 1$	2.098.960	1. Juni 1999	Nayan Hajratwala
14	$2^{3.021.377} - 1$	909.526	27. Jan. 1998	Roland Clarkson
15	$2^{2.976.221} - 1$	895.932	24. Aug. 1997	Gordon Spence
16	$2^{1.398.269} - 1$	420.921	November 1996	Joel Armengaud

Tabelle 3.2: Die größten vom GIMPS-Projekt gefundenen Primzahlen (Stand Jan. 2018)

Richard Crandall erfand den Transformations-Algorithmus, der im GIMPS-Programm benutzt wird. George Woltman implementierte Crandall's Algorithmus in Maschinensprache, wodurch das Primzahlenprogramm eine vorher nicht da gewesene Effizienz erhielt. Diese Arbeit führte zum GIMPS-Projekt.

Am 1. Juni 2003 wurde dem GIMPS-Server eine Zahl gemeldet, die evtl. die 40. Mersenne-Primzahl sein konnte. Diese wurde dann wie üblich überprüft, bevor sie veröffentlich werden sollte. Leider musste der Initiator und GIMPS-Projektleiter George Woltman Mitte Juni melden, dass diese Zahl zusammengesetzt war (dies war die erste falsche positive Rückmeldung eines Clients an den Server in 7 Jahren).

Am GIMPS-Projekt beteiligen sich z.Zt. rund 130.000 freiwillige Amateure und Experten, die ihre Rechner in das ursprünglich von der Firma Entropia organisierte „PrimeNet“ einbinden.

3.4.3 Wettbewerb der Electronic Frontier Foundation (EFF)

Angefacht wird diese Suche noch zusätzlich durch einen Wettbewerb, den die Nonprofit-Organisation EFF (Electronic Frontier Foundation) mit den Mitteln eines unbekannten Spenders gestartet hat. Den Teilnehmern winken Gewinne im Gesamtwert von 500.000 USD, wenn sie die längste Primzahl finden. Dabei sucht der unbekannte Spender nicht nach dem schnellsten Rechner, sondern er will auf die Möglichkeiten des *cooperative networking* aufmerksam machen: `http://www.eff.org/awards/coop`

Der Entdecker von M-38 erhielt für die Entdeckung der ersten Primzahl mit über 1 Million Dezimalstellen von der EFF eine Prämie von 50.000 USD.

Für die von 100.000 USD von der EFF für eine Primzahl mit mehr als 10 Millionen Dezimalstellen hat sich Edson Smith qualifiziert, der im GIMPS-Projekt $2^{43.112.609} - 1$ fand.

Nach den Preisregeln der EFF sind dann als nächste Stufe 150.000 US-Dollar für eine Prim-

zahl mit mehr als 100 Millionen Stellen ausgelobt.

Edouard Lucas (1842-1891) hielt über 70 Jahre den Rekord der größten bekannten Primzahl, indem er nachwies, dass $2^{127} - 1$ prim ist. So lange wird wohl kein neuer Rekord mehr Bestand haben.

3.5 Primzahltests[20,21]

Für die Anwendung sicherer Verschlüsselungsverfahren braucht man sehr große Primzahlen (Zahlen im Bereich von $2^{2.048}$ haben im Zehnersystem über 600 Stellen).

Sucht man nach den Primfaktoren, um zu entscheiden, ob eine Zahl prim ist, dauert die Suche zu lange, wenn schon der kleinste Primfaktor riesig ist. Die Zerlegung in Faktoren mittels rechnerischer systematischer Teilung oder mit dem Sieb des Eratosthenes ist mit heutigen Computern anwendbar für Zahlen mit bis zu circa 20 Stellen im Zehnersystem. Die größte Zahl, die bisher in ihre beiden annähernd gleich großen Primfaktoren zerlegt werden konnte, hatte 232 Stellen (vgl. RSA-768 in Kapitel 4.11.4).

Ist aber etwas über die *Bauart* (spezielle Struktur) der fraglichen Zahl bekannt, gibt es sehr hochentwickelte Verfahren, die deutlich schneller sind. Diese Verfahren beantworten nur die Primalitätseigenschaft einer Zahl, können aber nicht die Primfaktoren sehr großer zusammengesetzter Zahlen bestimmen.

Fermat[22] hatte im 17. Jahrhundert an Mersenne geschrieben, dass er vermute, dass alle Zahlen der Form

$$f(n) = 2^{2^n} + 1$$

für alle ganzen Zahlen $n \geq 0$ prim seien (siehe unten, Kapitel 3.6.4).

Schon im 19. Jahrhundert wusste man, dass die 29-stellige Zahl

$$f(7) = 2^{2^7} + 1$$

keine Primzahl ist. Aber erst 1970 fanden Morrison/Billhart ihre Zerlegung.

$$\begin{aligned} f(7) &= 340.282.366.920.938.463.463.374.607.431.768.211.457 \\ &= 59.649.589.127.497.217 \cdot 5.704.689.200.685.129.054.721 \end{aligned}$$

Auch wenn sich Fermat bei seiner Vermutung irrte, so stammt in diesem Zusammenhang von ihm doch ein sehr wichtiger Satz: Der (kleine) Fermatsche Satz, den Fermat im Jahr 1640 aufstellte, ist der Ausgangspunkt vieler schneller Primzahltests (siehe Kap. 4.8.3).

Satz 3.5.1 („kleiner" Fermat)**.** *Sei p eine Primzahl und a eine beliebige ganze Zahl, dann gilt für alle a*

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

[20] Ein didaktisch aufbereiteter Artikel zu den verschiedenen Primzahltests mit Schwerpunkt auf dem Miller-Rabin-Test findet sich in der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*. Siehe NF Teil 5 [WS10b].

[21] In dem Lernprogramm **ZT** können Sie den Fermat-Test und den Miller-Rabin-Test geführt Schritt für Schritt anwenden: Siehe die NT-Lern-Kapitel 3.2 und 3.3, Seiten 3-11/11.
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe auch Anhang A.6.
Eine Visualisierung dieser Verfahren ist in CT2 in dem Tutorial „**Die Welt der Primzahlen**" enthalten.

[22] Pierre de Fermat, französischer Mathematiker, 17.8.1601 – 12.1.1665.

Eine alternative Formulierung lautet:
Sei p eine Primzahl und a eine beliebige ganze Zahl, die kein Vielfaches von p ist (also $a \not\equiv 0 \bmod p$), dann gilt $a^{p-1} \equiv 1 \bmod p$.

Wer mit dem Rechnen mit Resten (Modulo-Rechnung) nicht so vertraut ist, möge den Satz einfach so hinnehmen oder erst Kapitel 4 „Einführung in die elementare Zahlentheorie mit Beispielen“ lesen. Wichtig ist, dass aus diesem Satz folgt, dass wenn diese Gleichheit für irgendein ganzes a nicht erfüllt ist, dann ist p keine Primzahl! Die Tests lassen sich (zum Beispiel für die erste Formulierung) leicht mit der *Testbasis* $a = 2$ durchführen.

Damit hat man ein Kriterium für Nicht-Primzahlen, also einen negativen Test, aber noch keinen Beweis, dass eine Zahl a prim ist. Leider gilt die Umkehrung zum Fermatschen Satz nicht, sonst hätten wir einen einfachen Beweis für die Primzahleigenschaft (man sagt auch, man hätte dann ein einfaches Primzahlkriterium).

Pseudoprimzahlen

Zahlen n, die die Eigenschaft

$$2^n \equiv 2 \bmod n$$

erfüllen, aber nicht prim sind, bezeichnet man als *Pseudoprimzahlen* (der Exponent n ist also keine Primzahl). Die erste Pseudoprimzahl ist

$$341 = 11 \cdot 31.$$

Carmichaelzahlen

Es gibt Pseudoprimzahlen n, die den Fermat-Test

$$a^{n-1} \equiv 1 \bmod n$$

mit allen Basen a, die teilerfremd zu n sind $[gcd(a, n) = 1]$, bestehen, obwohl die zu testenden Zahlen n nicht prim sind: Diese Zahlen heißen *Carmichaelzahlen*. Die erste ist

$$561 = 3 \cdot 11 \cdot 17.$$

Beispiel: Die zu testende Zahl sei 561.
Da $561 = 3 \cdot 11 \cdot 17$ ist, ergibt sich:
Die Testbedingung $a^{560} \bmod 561 = 1$
ist erfüllt für $a = 2, 4, 5, 7, \cdots$,
aber nicht für $a = 3, 6, 9, 11, 12, 15, 17, 18, 21, 22, \cdots$.
D.h. die Testbedingung muss nicht erfüllt sein, wenn die Basis ein Vielfaches von 3, von 11 oder von 17 ist.
Der Test angewandt auf $a = 3$ ergibt: $3^{560} \bmod 561 = 375$.
Der Test angewandt auf $a = 5$ ergibt: $5^{560} \bmod 561 = 1$.

Starke Pseudoprimzahlen

Ein stärkerer Test stammt von Miller/Rabin[23]: Dieser wird nur von sogenannten *starken Pseudoprimzahlen* bestanden. Wiederum gibt es starke Pseudoprimzahlen, die keine Primzahlen sind, aber das passiert deutlich seltener als bei den (einfachen) Pseudoprimzahlen oder bei den Carmichaelzahlen. Die kleinste starke Pseudoprimzahl zur Basis 2 ist

$$15.841 = 7 \cdot 31 \cdot 73.$$

Testet man alle 4 Basen $2, 3, 5$ und 7, so findet man bis $25 \cdot 10^9$ nur eine starke Pseudoprimzahl, also eine Zahl, die den Test besteht und doch keine Primzahl ist.

Weiterführende Mathematik hinter dem Rabin-Test gibt dann die Wahrscheinlichkeit an, mit der die untersuchte Zahl prim ist (solche Wahrscheinlichkeiten liegen heutzutage bei circa 10^{-60}).

Ausführliche Beschreibungen zu Tests, um herauszufinden, ob eine Zahl prim ist, finden sich zum Beispiel unter:

`http://www.utm.edu/research/primes/mersenne.shtml`
`http://www.utm.edu/research/primes/prove/index.html`

3.6 Spezial-Zahlentypen und die Suche nach einer Formel für Primzahlen

Derzeit sind keine brauchbaren, offenen (also nicht rekursiven) Formeln bekannt, die nur Primzahlen liefern (rekursiv bedeutet, dass zur Berechnung der Funktion auf dieselbe Funktion in Abhängigkeit einer kleineren Variablen zugegriffen wird). Die Mathematiker wären schon zufrieden, wenn sie eine Formel fänden, die wohl Lücken lässt (also nicht alle Primzahlen liefert), aber sonst keine zusammengesetzten Zahlen (Nicht-Primzahlen) liefert.

Optimal wäre, man würde für das Argument n sofort die n-te Primzahl bekommen, also für $f(8) = 19$ oder für $f(52) = 239$.

Ideen dazu finden sich in

`http://www.utm.edu/research/primes/notes/faq/p_n.html`.

Die Tabelle unter 3.10 enthält die exakten Werte für die n-ten Primzahlen für ausgewählte n.

Für „Primzahlformeln“ werden meist ganz spezielle Zahlentypen benutzt. Die folgende Aufzählung enthält die verbreitetesten Ansätze für „Primzahlformeln“, und welche Kenntnisse wir über sehr große Folgeglieder haben: Konnte die Primalität bewiesen werden? Wenn es zusammengesetzte Zahlen sind, konnten die Primfaktoren bestimmt werden?

[23] 1976 veröffentlichte Prof. Rabin einen effizienten probabilistischen Primzahltest, der auf einem zahlentheoretischen Ergebnis von Prof. Miller aus der Jahr davor basierte.
Prof. Rabin arbeitete an der Harvard und Hebrew Universität. Michael Oser Rabin wurde 1931 in Breslau geboren. Seine Familie emigrierte wegen der Nazi-Politik 1935 nach Palästina. Er ist ein weiteres Beispiel dafür, welch immensen intellektuellen Aderlass die Nazi-Rassenpolitik für Deutschland bewirkte.
Prof. Miller arbeitete an der Carnegie-Mellon Universität, School of Computer Science.
In [WS10b] wird die Funktionsweise des Miller-Rabin-Tests anhand eines Python-Programms Schritt für Schritt erklärt.

3.6.1 Mersennezahlen $f(n) = 2^n - 1$ für n prim

Wie oben gesehen, liefert diese Formel wohl relativ viele große Primzahlen, aber es kommt – wie für $n = 11$ $[f(n) = 2.047]$ – immer wieder vor, dass das Ergebnis auch bei primen Exponenten nicht prim ist.
Heute kennt man alle Mersenne-Primzahlen mit bis zu ca. 4.000.000 Dezimalstellen (M-39):

`http://yves.gallot.pagesperso-orange.fr/primes/index.html`

3.6.2 Verallgemeinerte Mersennezahlen $f(k, n) = k \cdot 2^n \pm 1$ für n prim und k kleine Primzahl / Proth-Zahlen[24]

Diese erste Verallgemeinerung der Mersennezahlen erzeugt die sogenannten Proth-Zahlen. Dafür gibt es (für kleine k) ebenfalls sehr schnelle Primzahltests (vgl. [Knu98]).
Praktisch ausführen lässt sich das zum Beispiel mit der Software Proths von Yves Gallot:

`http://www.prothsearch.net/index.html`.

3.6.3 Verallgemeinerte Mersennezahlen $f(b, n) = b^n \pm 1$ / Cunningham-Projekt

Dies ist eine zweite mögliche Verallgemeinerung der Mersennezahlen. Im **Cunningham-Projekt** werden die Faktoren aller zusammengesetzten Zahlen bestimmt, die sich in folgender Weise bilden:

$$f(b, n) = b^n \pm 1 \quad \text{für } b = 2, 3, 5, 6, 7, 10, 11, 12$$

(b ist ungleich der Vielfachen von schon benutzten Basen wie $4, 8, 9$).

Details hierzu finden sich unter:

`http://www.cerias.purdue.edu/homes/ssw/cun`

3.6.4 Fermatzahlen[25] $f(n) = 2^{2^n} + 1$

Wie oben in Kapitel 3.5 erwähnt, schrieb Fermat an Mersenne, dass er vermutet, dass alle Zahlen dieser Form prim seien. Diese Vermutung wurde jedoch von Euler (1732) widerlegt. Es gilt $641 | f(5)$.[26]

---

[24] Diese wurden nach dem französischen Landwirt François Proth (1852-1879) benannt. Berühmter noch als die Proth-Primzahlen dürfte das eng damit zusammenhängende Sierpinski-Problem sein, nämlich Zahlen k zu finden, so dass $k * 2^n + 1$ zusammengesetzt ist für alle $n > 0$. Siehe Tab. 3.1.

[25] Die Fermatschen Primzahlen spielen unter anderem eine wichtige Rolle in der Kreisteilung. Wie Gauss bewiesen hat, ist das reguläre p-Eck für eine Primzahl $p > 2$ dann und nur dann mit Zirkel und Lineal konstruierbar, wenn p eine Fermatsche Primzahl ist.

[26] Erstaunlicherweise kann man mit Hilfe des Satzes von Fermat diese Zahl leicht finden (siehe z.B. [Sch06, S. 176])

$$
\begin{array}{lll}
f(0) = 2^{2^0} + 1 = 2^1 + 1 & = 3 & \mapsto \text{ prim} \\
f(1) = 2^{2^1} + 1 = 2^2 + 1 & = 5 & \mapsto \text{ prim} \\
f(2) = 2^{2^2} + 1 = 2^4 + 1 & = 17 & \mapsto \text{ prim} \\
f(3) = 2^{2^3} + 1 = 2^8 + 1 & = 257 & \mapsto \text{ prim} \\
f(4) = 2^{2^4} + 1 = 2^{16} + 1 & = 65.537 & \mapsto \text{ prim} \\
f(5) = 2^{2^5} + 1 = 2^{32} + 1 & = 4.294.967.297 = 641 \cdot 6.700.417 & \mapsto \text{ NICHT prim!} \\
f(6) = 2^{2^6} + 1 = 2^{64} + 1 & = 18.446.744.073.709.551.617 & \\
 & = 274.177 \cdot 67.280.421.310.721 & \mapsto \text{ NICHT prim!} \\
f(7) = 2^{2^7} + 1 = 2^{128} + 1 & = \text{(siehe Seite 77)} & \mapsto \text{ NICHT prim!}
\end{array}
$$

Innerhalb des Projektes „Distributed Search for Fermat Number Dividers“, das von Leonid Durman angeboten wird, gibt es ebenfalls Fortschritte beim Finden von neuen Primzahl-Riesen (`http://www.fermatsearch.org/` – diese Webseite hat Verknüpfungen zu Seiten in russisch, italienisch und deutsch).

Die entdeckten Faktoren können sowohl zusammengesetzte natürliche als auch prime natürliche Zahlen sein.

Am 22. Februar 2003 entdeckte John Cosgrave

- die größte bis dahin bekannte zusammengesetzte Fermatzahl und
- die größte bis dahin bekannte prime nicht-einfache Mersennezahl mit 645.817 Dezimalstellen.

Die Fermatzahl

$$f(2.145.351) = 2^{(2^{2.145.351})} + 1$$

ist teilbar durch die Primzahl

$$p = 3 * 2^{2.145.353} + 1$$

Diese Primzahl p war damals die größte bis dahin bekannte prime verallgemeinerte Mersennezahl und die 5.-größte damals bekannte Primzahl überhaupt.

Zu diesem Erfolg trugen bei: NewPGen von Paul Jobling's, PRP von George Woltman's, Proth von Yves Gallot's Programm und die Proth-Gallot-Gruppe am St. Patrick's College, Dublin.

Weitere Details finden sich unter
`http://www.fermatsearch.org/history/cosgrave_record.htm`

3.6.5 Verallgemeinerte Fermatzahlen[27] $f(b, n) = b^{2^n} + 1$

Verallgemeinerte Fermatzahlen kommen häufiger vor als Mersennezahlen gleicher Größe, so dass wahrscheinlich noch viele gefunden werden können, die die großen Lücken zwischen den Mersenne-Primzahlen verkleinern. Fortschritte in der Zahlentheorie haben es ermöglicht, dass Zahlen, deren Repräsentation nicht auf eine Basis von 2 beschränkt ist, nun mit fast der gleichen Geschwindigkeit wie Mersennezahlen getestet werden können.

[27] Hier ist die Basis b nicht notwendigerweise 2. Noch allgemeiner wäre: $f(b, c, n) = b^{c^n} \pm 1$

Yves Gallot schrieb das Programm Proth.exe zur Untersuchung verallgemeinerter Fermatzahlen.

Mit diesem Programm fand Michael Angel am 16. Februar 2003 eine prime verallgemeinerte Fermatzahl mit 628.808 Dezimalstellen, die zum damaligen Zeitpunkt zur 5.-größten bis dahin bekannten Primzahl wurde:

$$b^{2^{17}} + 1 = 62.722^{131.072} + 1.$$

Weitere Details finden sich unter

`http://primes.utm.edu/top20/page.php?id=12`

3.6.6 Carmichaelzahlen

Wie oben in Kapitel 3.5 erwähnt, sind nicht alle Carmichaelzahlen prim.

3.6.7 Pseudoprimzahlen

Siehe oben in Kapitel 3.5.

3.6.8 Starke Pseudoprimzahlen

Siehe oben in Kapitel 3.5.

3.6.9 Idee aufgrund von Euklids Beweis $p_1 \cdot p_2 \cdots p_n + 1$

Diese Idee entstammt Euklids Beweis (siehe Kapitel 3.3), dass es unendlich viele Primzahlen gibt.

$$\begin{array}{lll}
2{\cdot}3 + 1 & = 7 & \mapsto \text{ prim} \\
2{\cdot}3{\cdot}5 + 1 & = 31 & \mapsto \text{ prim} \\
2{\cdot}3{\cdot}5{\cdot}7 + 1 & = 211 & \mapsto \text{ prim} \\
2{\cdot}3{\cdot}\cdots 11 + 1 & = 2311 & \mapsto \text{ prim} \\
2 \cdot 3 \cdots 13 + 1 & = 59 \cdot 509 & \mapsto \text{ NICHT prim!} \\
2 \cdot 3 \cdots 17 + 1 & = 19 \cdot 97 \cdot 277 & \mapsto \text{ NICHT prim!}
\end{array}$$

3.6.10 Wie zuvor, nur -1 statt $+1$: $p_1 \cdot p_2 \cdots p_n - 1$

$$\begin{array}{lll}
2 \cdot 3 - 1 & = 5 & \mapsto \text{ prim} \\
2 \cdot 3 \cdot 5 - 1 & = 29 & \mapsto \text{ prim} \\
2 \cdot 3 \cdots 7 - 1 & = 11 \cdot 19 & \mapsto \text{ NICHT prim!} \\
2 \cdot 3 \cdots 11 - 1 & = 2309 & \mapsto \text{ prim} \\
2 \cdot 3 \cdots 13 - 1 & = 30029 & \mapsto \text{ prim} \\
2 \cdot 3 \cdots 17 - 1 & = 61 \cdot 8369 & \mapsto \text{ NICHT prim!}
\end{array}$$

3.6.11 Euklidzahlen $e_n = e_0 \cdot e_1 \cdots e_{n-1} + 1$ mit $n \geq 1$ und $e_0 := 1$

e_{n-1} ist nicht die $(n-1)$-te Primzahl, sondern die zuvor hier gefundene Zahl. Diese Formel ist leider nicht offen, sondern rekursiv. Die Folge startet mit

$$\begin{array}{lll}
e_1 = 1 + 1 & = 2 & \mapsto \text{ prim} \\
e_2 = e_1 + 1 & = 3 & \mapsto \text{ prim} \\
e_3 = e_1 \cdot e_2 + 1 & = 7 & \mapsto \text{ prim} \\
e_4 = e_1 \cdot e_2 \cdot e_3 + 1 & = 43 & \mapsto \text{ prim} \\
e_5 = e_1 \cdot e_2 \cdots e_4 + 1 & = 13 \cdot 139 & \mapsto \text{ NICHT prim!} \\
e_6 = e_1 \cdot e_2 \cdots e_5 + 1 & = 3.263.443 & \mapsto \text{ prim} \\
e_7 = e_1 \cdot e_2 \cdots e_6 + 1 & = 547 \cdot 607 \cdot 1.033 \cdot 31.051 & \mapsto \text{ NICHT prim!} \\
e_8 = e_1 \cdot e_2 \cdots e_7 + 1 & = 29.881 \cdot 67.003 \cdot 9.119.521 \cdot 6.212.157.481 & \mapsto \text{ NICHT prim!}
\end{array}$$

Auch $e_9, \cdots, e_{17}$ sind zusammengesetzt, so dass dies auch keine brauchbare Primzahlformel ist.

Bemerkung:
Das Besondere an diesen Zahlen ist, dass sie jeweils paarweise keinen gemeinsamen Teiler außer 1 haben[28], sie sind also *relativ zueinander prim.*

3.6.12 $f(n) = n^2 + n + 41$

Diese Folge hat einen sehr *erfolgversprechenden* Anfang, aber das ist noch lange kein Beweis.

$$\begin{array}{lll}
f(0) = 41 & & \mapsto \text{ prim} \\
f(1) = 43 & & \mapsto \text{ prim} \\
f(2) = 47 & & \mapsto \text{ prim} \\
f(3) = 53 & & \mapsto \text{ prim} \\
f(4) = 61 & & \mapsto \text{ prim} \\
f(5) = 71 & & \mapsto \text{ prim} \\
f(6) = 83 & & \mapsto \text{ prim} \\
f(7) = 97 & & \mapsto \text{ prim} \\
\vdots & & \\
f(33) = 1.163 & & \mapsto \text{ prim} \\
f(34) = 1.231 & & \mapsto \text{ prim} \\
f(35) = 1.301 & & \mapsto \text{ prim} \\
f(36) = 1.373 & & \mapsto \text{ prim} \\
f(37) = 1.447 & & \mapsto \text{ prim} \\
f(38) = 1.523 & & \mapsto \text{ prim} \\
f(39) = 1.601 & & \mapsto \text{ prim} \\
f(40) = 1681 & = 41 \cdot 41 & \mapsto \text{ NICHT prim!} \\
f(41) = 1763 & = 41 \cdot 43 & \mapsto \text{ NICHT prim!}
\end{array}$$

Die ersten 40 Werte sind Primzahlen (diese haben die auffallende Regelmäßigkeit, dass ihr Abstand beginnend mit dem Abstand 2 jeweils um 2 wächst), aber der 41. und der 42. Wert sind keine Primzahlen. Dass $f(41)$ keine Primzahl sein kann, lässt sich leicht überlegen: $f(41) = 41^2 + 41 + 41 = 41(41 + 1 + 1) = 41 \cdot 43$.

---

[28] Dies kann leicht gezeigt werden mit Hilfe der Rechenregel für den *größten gemeinsamen Teiler ggT* mit $ggT(a, b) = ggT(b - \lfloor b/a \rfloor \cdot a, a)$.
Es gilt für $i < j$:
$ggT(e_i, e_j) \leq ggT(e_1 \cdots e_i \cdots e_{j-1}, e_j) = ggT(e_j - e_1 \cdots e_i \cdots e_{j-1}, e_1 \cdots e_i \cdots e_{j-1}) = ggT(1, e_1 \cdots e_i \cdots e_{j-1}) = 1$.
Siehe Seite 181.

3.6.13 $f(n) = n^2 - 79 \cdot n + 1.601$

Diese Funktion liefert für die Werte $n = 0$ bis $n = 79$ stets Primzahlwerte.[29] Leider ergibt $f(80) = 1.681 = 11 \cdot 151$ keine Primzahl. Bis heute kennt man keine Funktion, die mehr aufeinanderfolgende Primzahlen annimmt. Andererseits kommt jede Primzahl doppelt vor (erst in der absteigenden, dann in der aufsteigenden Folge), so dass sie insgesamt genau 40 verschiedene Primzahlwerte in Folge liefert (Es sind dieselben wie die, die die Funktion aus Kapitel 3.6.12 liefert)[30].

$f(0) = 1.601$	$\mapsto$	prim	$f(26) = 223$	$\mapsto$	prim
$f(1) = 1.523$	$\mapsto$	prim	$f(27) = 197$	$\mapsto$	prim
$f(2) = 1.447$	$\mapsto$	prim	$f(28) = 173$	$\mapsto$	prim
$f(3) = 1.373$	$\mapsto$	prim	$f(29) = 151$	$\mapsto$	prim
$f(4) = 1.301$	$\mapsto$	prim	$f(30) = 131$	$\mapsto$	prim
$f(5) = 1.231$	$\mapsto$	prim	$f(31) = 113$	$\mapsto$	prim
$f(6) = 1.163$	$\mapsto$	prim	$f(32) = 97$	$\mapsto$	prim
$f(7) = 1.097$	$\mapsto$	prim	$f(33) = 83$	$\mapsto$	prim
$f(8) = 1.033$	$\mapsto$	prim	$f(34) = 71$	$\mapsto$	prim
$f(9) = 971$	$\mapsto$	prim	$f(35) = 61$	$\mapsto$	prim
$f(10) = 911$	$\mapsto$	prim	$f(36) = 53$	$\mapsto$	prim
$f(11) = 853$	$\mapsto$	prim	$f(37) = 47$	$\mapsto$	prim
$f(12) = 797$	$\mapsto$	prim	$f(38) = 43$	$\mapsto$	prim
$f(13) = 743$	$\mapsto$	prim	$f(39) = 41$	$\mapsto$	prim
$f(14) = 691$	$\mapsto$	prim	$f(40) = 41$	$\mapsto$	prim
$f(15) = 641$	$\mapsto$	prim	$f(41) = 43$	$\mapsto$	prim
$f(16) = 593$	$\mapsto$	prim	$f(42) = 47$	$\mapsto$	prim
$f(17) = 547$	$\mapsto$	prim	$f(43) = 53$	$\mapsto$	prim
$f(18) = 503$	$\mapsto$	prim	$\cdots$		
$f(19) = 461$	$\mapsto$	prim	$f(77) = 1.447$	$\mapsto$	prim
$f(20) = 421$	$\mapsto$	prim	$f(78) = 1.523$	$\mapsto$	prim
$f(21) = 383$	$\mapsto$	prim	$f(79) = 1.601$	$\mapsto$	prim
$f(22) = 347$	$\mapsto$	prim	$f(80) = 41 \cdot 41$	$\mapsto$	NICHT prim!
$f(21) = 383$	$\mapsto$	prim	$f(81) = 41 \cdot 43$	$\mapsto$	NICHT prim!
$f(22) = 347$	$\mapsto$	prim	$f(82) = 1.847$	$\mapsto$	prim
$f(23) = 313$	$\mapsto$	prim	$f(83) = 1.933$	$\mapsto$	prim
$f(24) = 281$	$\mapsto$	prim	$f(84) = 43 \cdot 47$	$\mapsto$	NICHT prim!
$f(25) = 251$	$\mapsto$	prim			

3.6.14 Polynomfunktionen $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$ (a_i aus $\mathbb{Z}$, $n \geq 1$)

Es existiert kein solches Polynom, das für alle x aus $\mathbb{Z}$ ausschließlich Primzahlwerte annimmt. Zum Beweis sei auf [Pad96, S. 83 f.], verwiesen, wo sich auch weitere Details zu Primzahlformeln finden.

[29] In Kapitel 3.14, „Anhang: Beispiele mit SageMath“ finden Sie den Quellcode zur Berechnung der Tabelle mit SageMath.

[30] Ein weiteres quadratisches Polynom, das dieselben Primzahlen liefert, ist: $f(n) = n^2 - 9 \cdot n + 61$.
Unter den ersten 1000 Folgegliedern dieser Funktion sind über 50% prim (vgl. Kapitel 3.14, „Anhang: Beispiele mit SageMath“).

Damit ist es hoffnungslos, weiter nach Formeln (Funktionen) wie in Kapitel 3.6.12 oder Kapitel 3.6.13 zu suchen.

3.6.15 Catalans Mersenne-Vermutung[31]

Catalan äußerte die Vermutung[32], dass C_4 und jede weitere Zahl dieser Folge eine Primzahl ist:

$$\begin{aligned} C_0 &= 2, \\ C_1 &= 2^{C_0} - 1, \\ C_2 &= 2^{C_1} - 1, \\ C_3 &= 2^{C_2} - 1, \\ C_4 &= 2^{C_3} - 1, \cdots \end{aligned}$$

Diese Folge ist rekursiv definiert und wächst sehr schnell. Besteht sie nur aus Primzahlen?

$$\begin{array}{lll} C_0 = 2 & & \mapsto \text{ prim} \\ C_1 = 2^2 - 1 & = 3 & \mapsto \text{ prim} \\ C_2 = 2^3 - 1 & = 7 & \mapsto \text{ prim} \\ C_3 = 2^7 - 1 & = 127 & \mapsto \text{ prim} \\ C_4 = 2^{127} - 1 & = 170.141.183.460.469.231.731.687.303.715.884.105.727 & \mapsto \text{ prim} \end{array}$$

Ob $C_5 = 2^{C_4} - 1$ bzw. alle weiteren Elemente dieser Reihe prim sind, ist (noch) nicht bekannt. Bewiesen ist jedenfalls nicht, dass diese Formel nur Primzahlen liefert.

Es scheint sehr unwahrscheinlich, dass C_5 (und die weiteren Folgenglieder) prim sind.

Dies könnte ein weiteres Beispiel von Guys „Gesetz der kleinen Zahlen“[33] sein.

3.6.16 Doppelte Mersenne-Primzahlen

Die obigen Catalan-Mersenne-Zahlen sind ab dem Folgenglied C_2 eine Teilmenge der doppelten Mersenne-Primzahlen.[34] Doppelte Mersenne-Primzahlen haben die Form

$$M_{M_p} = 2^{2^p - 1} - 1$$

wobei p ein primer Mersenne-Exponent ist und M_p eine prime Mersenne-Zahl.

Die ersten Werte von p, für die M_p prim ist, sind p = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, ... (siehe oben).

M_{M_p} ist prim für p = 2, 3, 5, 7, und nimmt dabei ff. Werte an: 7, 127, 2.147.483.647, 170.141.183.460.469.231.731.687.303.715.884.105.727.[35]

---

[31] Eugene Charles Catalan, belgischer Mathematiker, 30.5.1814−14.2.1894.
Nach ihm sind auch die sogenannten *Catalanzahlen* $A(n) = (1/(n+1)) * (2n)!/(n!)^2$
$= 1, 2, 5, 14, 42, 132, 429, 1.430, 4.862, 16.796, 58.786, 208.012, 742.900, 2.674.440, 9.694.845, \ldots$ benannt.

[32] Siehe `http://primes.utm.edu/mersenne/index.html` unter „Conjectures and Unsolved Problems“.

[33] `http://primes.utm.edu/glossary/page.php?sort=LawOfSmall`

[34] `http://en.wikipedia.org/wiki/Catalan%E2%80%93Mersenne_number`

[35] Mit SageMath kann man sich das folgendermaßen ausgeben lassen:

```
sage: for p in (2,3,5,7): Mp=(2^p)-1; MMp=(2^Mp)-1; B=is_prime(MMp); print p, Mp, MMp, B;
....:
2  3    7  True
3  7    127  True
5  31   2147483647  True
7  127  170141183460469231731687303715884105727  True
```

Für p = 11, 13, 17, 19, und 31 sind die entsprechenden doppelten Mersenne-Zahlen nicht prim.

Der nächste Kandidat für die nächste doppelte Mersenne-Primzahl ist

$$M_{M_{61}} = 2^{2305843009213693951} - 1$$

Mit einer Größe von $1,695 * 10^{694.127.911.065.419.641}$ ist auch diese Zahl — ebenso wie C_5 — viel zu groß für alle derzeit bekannten Primzahltests.

3.7 Dichte und Verteilung der Primzahlen

Wie Euklid herausfand, gibt es unendlich viele Primzahlen. Einige unendliche Mengen sind aber *dichter* als andere.

Innerhalb der natürlichen Zahlen gibt es unendlich viele gerade, ungerade und quadratische Zahlen. Wie man die „Dichte“ zweier unendlicher Mengen vergleicht, soll kurz anhand der geraden und quadratischen Zahlen erläutert werden.

Nach folgenden Gesichtspunkten ist die Menge der geraden Zahlen dichter als die Menge der quadratischen Zahlen:[36]

- die Größe des n-ten Elements:
  Das n-te Element der geraden Zahlen ist $2n$; das n-te Element der Quadratzahlen ist n^2. Weil für alle $n > 2$ gilt: $2n < n^2$, kommt die n-te gerade Zahl viel früher als die n-te quadratische Zahl.
- die Anzahlen der Werte, die kleiner oder gleich einem bestimmten *Dachwert* x aus $\mathbb{R}$ sind:
  Es gibt $\lfloor x/2 \rfloor$ solcher gerader Zahlen und $\lfloor \sqrt{x} \rfloor$ Quadratzahlen. Da für alle $x > 6$ gilt, dass der Wert $\lfloor x/2 \rfloor$ größer ist als die größte ganze Zahl kleiner oder gleich der Quadratwurzel aus x, sind die geraden Zahlen dichter verteilt.

Der Wert der n-ten Primzahl $P(n)$

Satz 3.7.1. *Für große n gilt: Der Wert der n-ten Primzahl $P(n)$ ist asymptotisch zu $n \cdot ln(n)$, d.h. der Grenzwert des Verhältnisses $P(n)/(n \cdot \ln n)$ ist gleich 1, wenn n gegen unendlich geht.*

Es gilt für $n \geq 5$, dass $P(n)$ zwischen $2n$ und n^2 liegt. Es gibt also weniger Primzahlen als gerade natürliche Zahlen, aber es gibt mehr Primzahlen als Quadratzahlen.[37]

Die Anzahl der Primzahlen $PI(x)$

Ähnlich wird die Anzahl[38] $PI(x)$ definiert: Es ist die Anzahl aller Primzahlen, die den Dachwert x nicht übersteigen.

[36] Während in der Umgangssprache oft gesagt wird, es „gibt mehr“ gerade als quadratische Zahlen, sagen Mathematiker, dass es von beiden unendlich viele gibt, dass ihre Mengen äquivalent zu $\mathbb{N}$ sind (also beide unendlich und abzählbar, d.h. man kann für jede gerade Zahl und für jede quadratische Zahl eine natürliche Zahl angeben), dass aber die Menge der geraden Zahlen dichter ist als die der quadratischen Zahlen. Mathematiker haben für die Mächtigkeit von Mengen also sehr präzise Formulierungen gefunden.

[37] Vergleiche auch Tabelle 3.10.

[38] Oft wird statt PI(x) auch $\Pi(x)$ geschrieben.

Satz 3.7.2. *$PI(x)$ ist asymptotisch zu $x/ln(x)$.*

Dies ist der **Primzahlsatz** (prime number theorem). Er wurde von Legendre[39] und Gauss[40] aufgestellt und erst über 100 Jahre später bewiesen.[41]

Die Tabellen unter 3.9 zeigen die Anzahl von Primzahlen in verschiedenen Intervallen. Grafisch dargestellt ist die Verteilung in der Abbildung 3.9 auf Seite 108 im Anhang 3.13.

Primzahlsatz-Formeln gelten nur für n gegen unendlich. Die Formel von Gauss kann durch präzisere Formeln ersetzt werden. Für $x \geq 67$ gilt:

$$ln(x) - 1,5 < x/PI(x) < ln(x) - 0,5$$

Im Bewusstsein, dass $PI(x) = x/\ln x$ nur für sehr große x (x gegen unendlich) gilt, kann man folgende Übersicht erstellen:

x	$ln(x)$	$x/ln(x)$	$PI(x)$(gezählt)	$PI(x)/(x/ln(x))$
10^3	6,908	144	168	1,160
10^6	13,816	72.386	78.498	1,085
10^9	20,723	48.254.942	50.847.534	1,054

Für eine Binärzahl[42] x der Länge 250 Bit (2^{250} ist ungefähr $= 1,809251 * 10^{75}$) gilt:

$$PI(x) = 2^{250}/(250 \cdot \ln 2) \text{ ist ungefähr } = 2^{250}/173,28677 = 1,045810 \cdot 10^{73}.$$

Es ist also zu erwarten, dass sich innerhalb der Zahlen der Bitlänge kleiner als 250 ungefähr 10^{73} Primzahlen befinden (ein beruhigendes Ergebnis?!).

Man kann das auch so formulieren: Betrachtet man eine *zufällige* natürliche Zahl n, so sind die Chancen, dass diese Zahl prim ist, circa $1/\ln(n)$. Nehmen wir zum Beispiel Zahlen in der Gegend von 10^{16}, so müssen wir ungefähr (durchschnittlich) $16 \cdot \ln 10 = 36,8$ Zahlen betrachten, bis wir eine Primzahl finden. Ein genaue Untersuchung zeigt: Zwischen $10^{16} - 370$ und $10^{16} - 1$ gibt es 10 Primzahlen.

Unter der Überschrift *How Many Primes Are There* finden sich unter
`http://primes.utm.edu/howmany.html`
viele weitere Details.

$PI(x)$ lässt sich leicht per
`https://primes.utm.edu/nthprime/`
bestimmen.

Die **Verteilung** der Primzahlen[43] weist viele Unregelmäßigkeiten auf, für die bis heute kein „System“ gefunden wurde: Einerseits liegen viele eng benachbart wie 2 und 3, 11 und 13, 809 und 811, andererseits tauchen auch längere Primzahllücken auf. So liegen zum Beispiel zwischen 113 und 127, 293 und 307, 317 und 331, 523 und 541, 773 und 787, 839 und 853 sowie zwischen 887 und 907 keine Primzahlen.
Details siehe:

[39] Adrien-Marie Legendre, französischer Mathematiker, 18.9.1752−10.1.1833.
[40] Carl Friedrich Gauss, deutscher Mathematiker und Astronom, 30.4.1777−23.2.1855.
[41] Ein didaktisch aufbereiteter Artikel zum Primzahlsatz mit seiner Anwendung auf den RSA-Algorithmus findet sich in der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle.* Siehe NF Teil 4 [WS10a].
[42] Eine Zahl im Zweiersystem besteht nur aus den Ziffern 0 und 1.
[43] Einige Visualisierungen (Plots) zur Menge von Primzahlen in verschiedenen Zahlendimensionen finden Sie in Kapitel 3.13, „Anhang: Visualisierung der Menge der Primzahlen in hohen Bereichen“.

`http://primes.utm.edu/notes/gaps.html`

Ein Teil des Ehrgeizes der Mathematiker liegt gerade darin, die Geheimnisse dieser Unregelmäßigkeiten herauszufinden.

Sieb des Eratosthenes

Ein einfacher Weg, alle $PI(x)$ Primzahlen kleiner oder gleich x zu berechnen, ist das Sieb des Erathostenes. Er fand schon im 3. Jahrhundert vor Christus einen sehr einfach automatisierbaren Weg, das herauszufinden. Zuerst werden ab 2 alle Zahlen bis x aufgeschrieben, die 2 umkreist und dann streicht man alle Vielfachen von 2. Anschließend umkreist man die kleinste noch nicht umkreiste oder gestrichene Zahl (nun 3), streicht wieder alle ihre Vielfachen, usw.

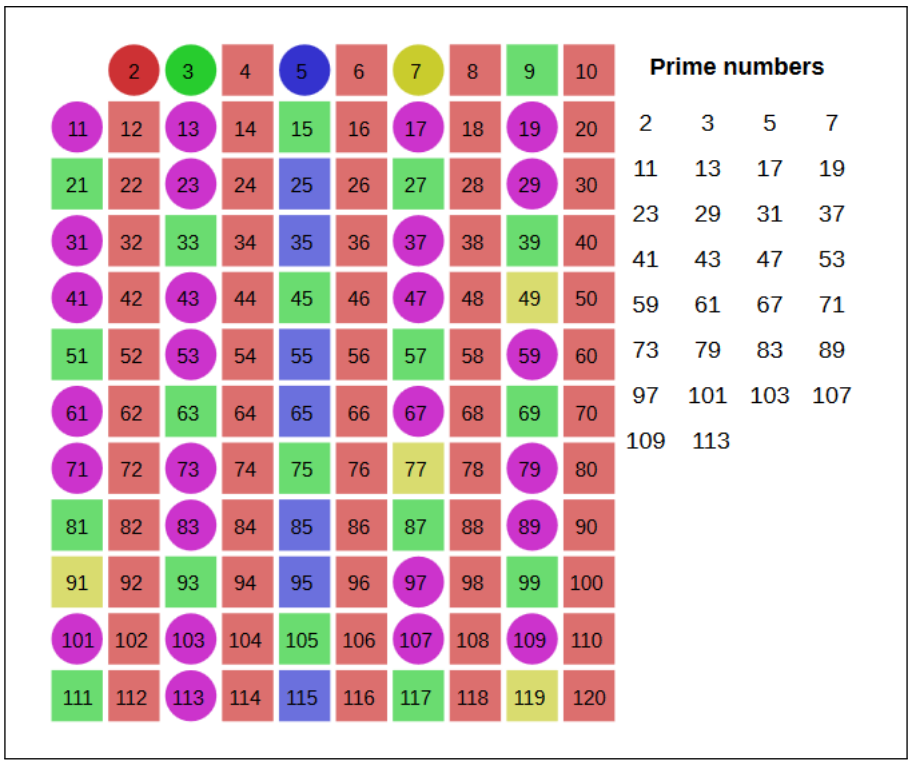


Abbildung 3.5: Das Sieb des Eratosthenes angewandt auf die ersten 120 Zahlen[44]

Durchzuführen braucht man das nur bis zu der größten Zahl, deren Quadrat kleiner oder gleich x ist (hier also bis 10, da 11^2 schon > 120).[45]

Abgesehen von 2 sind Primzahlen nie gerade. Abgesehen von 2 und 5 haben Primzahlen nie die Endziffern 2, 5 oder 0. Also braucht man sowieso nur Zahlen mit den Endziffern 1, 3, 7, 9 zu betrachten (es gibt unendlich viele Primzahlen mit jeder dieser letzten Ziffern; vergleiche [Tie73, Bd. 1, S. 137]).

[44] Grafik von `https://upload.wikimedia.org/wikipedia/commons/0/0b/Sieve_of_Eratosthenes_animation.svg`

[45] Mit dem Lernprogramm **ZT** können Sie für beliebige eigene Wertemengen das Sieb des Eratosthenes rechnergestützt und geführt Schritt für Schritt anwenden: Siehe ZT-Lern-Kapitel 1.2, Seite 6/21 und 7/21.
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.
Eine Visualisierung dieses Verfahrens ist in CT2 in dem Tutorial **„Die Welt der Primzahlen“** enthalten.

Inzwischen findet man im Internet große Datenbanken, die entweder viele Primzahlen oder die Zerlegung vieler zusammengesetzter Zahlen in ihre Primfaktoren enthalten.

Weitere interessante Themen rund um Primzahlen
In diesem Kapitel 3 wurden weitere, eher zahlentheoretische Themen wie Teilbarkeitsregeln, Modulo-Rechnung, modulare Inverse, modulare Potenzen und Wurzeln, chinesischer Restesatz, Eulersche Phi-Funktion und perfekte Zahlen nicht betrachtet. Auf einige dieser Themen geht das **nächste Kapitel** (Kapitel 4) ein.

3.8 Anmerkungen zu Primzahlen

Die folgenden Anmerkungen listen einzelne interessante Sätze, Vermutungen und Fragestellungen zu Primzahlen auf, aber auch Kurioses und Übersichten.

3.8.1 Bewiesene Aussagen / Sätze zu Primzahlen

- Zu jeder Zahl n aus $\mathbf{N}$ gibt es n aufeinanderfolgende natürliche Zahlen, die keine Primzahlen sind. Ein Beweis findet sich in [Pad96, S. 79].
- Paul Erdös[46] bewies: Zwischen jeder beliebigen Zahl ungleich 1 und ihrem Doppelten gibt es mindestens eine Primzahl. Er bewies das Theorem nicht als erster, aber auf einfachere Weise als andere vor ihm.
- Es existiert eine reelle Zahl a, so dass die Funktion $f : \mathbf{N} \to \mathbb{Z}$ mit $n \mapsto \lfloor a^{3^n} \rfloor$ für alle n nur Primzahlenwerte annimmt[47] (siehe [Pad96, S. 82]). Leider macht die Bestimmung von a Probleme (siehe Kapitel 3.8.2).[48]
- Es gibt arithmetische Primzahlfolgen beliebig großer Länge.[49,50]

  Die 1923 von dem berühmten englischen Mathematiker Hardy[51] aufgestellte Vermutung, dass es arithmetische Folgen beliebiger Länge gibt, die nur aus Primzahlen bestehen, wurde 2004 von zwei jungen amerikanischen Mathematikern bewiesen.

  Jedes Schulkind lernt im Mathematikunterricht irgendwann einmal die arithmetischen Zahlenfolgen kennen. Das sind Aneinanderreihungen von Zahlen, bei denen die Abstände zwischen je zwei aufeinander folgenden Gliedern gleich sind - etwa bei der Folge 5, 8, 11, 14, 17, 20. Der Abstand der Glieder beträgt hierbei jeweils 3 und die Folge hat 6 Folgenglieder. Eine arithmetische Folge muss mindestens 3 Folgenglieder haben, kann aber auch unendlich viele haben.

  Arithmetische Folgen sind seit Jahrtausenden bekannt und bergen eigentlich keine Geheimnisse mehr. Spannend wird es erst wieder, wenn die Glieder einer arithmetischen Folge noch zusätzliche Eigenschaften haben sollen, wie das bei Primzahlen der Fall ist.

  Primzahlen sind ganze Zahlen, die größer als 1 und nur durch 1 und sich selbst ohne Rest teilbar sind. Die zehn kleinsten Primzahlen sind 2, 3, 5, 7, 11, 13, 17, 19, 23 und 29.

  Eine arithmetische Primzahlfolge mit fünf Gliedern ist beispielsweise 5, 17, 29, 41, 53. Der Abstand der Zahlen beträgt jeweils 12.

---

[46] Paul Erdös, ungarischer Mathematiker, 26.03.1913−20.09.1996.

[47] Die Gaussklammer $\lfloor x \rfloor$ der reellwertigen Zahl x ist definiert als: $\lfloor x \rfloor$ ist die größte ganze Zahl kleiner oder gleich x.

[48] Wenn jemand weiß, wie man das beweist, würden wir uns sehr freuen, dies zu erfahren. Friedhelm Padberg sagte auf Nachfrage, er habe den Beweis nicht mehr.

[49] Quellen:
- `http://primes.utm.edu/glossary/page.php?sort=ArithmeticSequence` Original-Quelle
- `http://en.wikipedia.org/wiki/Primes_in_arithmetic_progression`
- `http://en.wikipedia.org/wiki/Problems_involving_arithmetic_progressions`
- `https://de.wikipedia.org/wiki/Cunningham-Kette`
- GEO 10 / 2004: „Experiment mit Folgen“
- `http://www.faz.net` „Hardys Vermutung – Primzahlen ohne Ende“ von Heinrich Hemme (06. Juli 2004)

[50] Arithmetische Folgen mit k Primzahlen werden auch Prime arithmetic progressions genannt und daher PAP-k bzw. AP-k abgekürzt.

[51] Godfrey Harold Hardy, britischer Mathematiker, 7.2.1877−1.12.1947.

Diese Folge lässt sich nicht verlängern, ohne ihre Eigenschaft einzubüßen, denn das nächste Glied müsste 65 sein, und diese Zahl ist das Produkt aus 5 und 13 und somit keine Primzahl.

Wie viele Glieder kann eine arithmetische Primzahlfolge haben? Mit dieser Frage haben sich schon um 1770 der Franzose Joseph-Louis Lagrange und der Engländer Edward Waring beschäftigt. Im Jahre 1923 vermuteten der berühmte britische Mathematiker Godfrey Harold Hardy und sein Kollege John Littlewood, dass es keine Obergrenze für die Zahl der Glieder gebe. Doch es gelang ihnen nicht, das zu beweisen. Im Jahr 1939 gab es jedoch einen anderen Fortschritt: Der holländische Mathematiker Johannes van der Corput konnte nachweisen, dass es unendlich viele arithmetische Primzahlfolgen mit genau drei Gliedern gibt. Zwei Beispiele hierfür sind 3, 5, 7 und 47, 53, 59.

Die längste Primzahlfolge, die man bisher kennt, hat 25 Glieder. In der Tabelle 3.3 sind die längsten bekannten arithmetischen Primzahlfolgen mit minimaler Distanz[52] aufgelistet.

Den beiden jungen[53] Mathematikern Ben Green and Terence Tao ist es im Jahre 2004 gelungen, die mehr als achtzig Jahre alte Hardysche Vermutung zu beweisen: Es gibt arithmetische Primzahlfolgen beliebiger Länge. Außerdem bewiesen sie, dass es zu jeder vorgegebenen Länge unendlich viele verschiedene solcher Folgen gibt.

Eigentlich hatten Green und Tao nur beweisen wollen, dass es unendlich viele arithmetische Primzahlfolgen mit vier Gliedern gibt. Dazu betrachteten sie Mengen, die neben Primzahlen auch Beinaheprimzahlen enthielten. Das sind Zahlen, die nur wenige Teiler haben - beispielsweise die Halbprimzahlen, die Produkte aus genau zwei Primzahlen sind. Dadurch konnten die beiden Mathematiker ihre Arbeit wesentlich erleichtern, denn über Beinaheprimzahlen gab es schon zahlreiche nützliche Theoreme. Schließlich erkannten sie, dass ihr Verfahren viel mächtiger ist, als sie selbst angenommen hatten, und sie bewiesen damit die Hardysche Vermutung.

Der Beweis von Green und Tao umfasst immerhin 49 Seiten. Tatsächlich beliebig lange arithmetische Primzahlfolgen kann man damit aber nicht finden. Der Beweis ist nicht konstruktiv, sondern ein so genannter Existenzbeweis. Das heißt, die beiden Mathematiker haben „nur“ gezeigt, dass beliebig lange Folgen existieren, aber nicht, wie man sie findet.

Das heißt, in der Menge der natürlichen Primzahlen gibt es zum Beispiel eine Folge von einer Milliarde Primzahlen, die alle den gleichen Abstand haben; und davon gibt es unendlich viele. Diese Folgen liegen aber sehr „weit draußen“.

[52] Dagegen sind in `http://en.wikipedia.org/wiki/Primes_in_arithmetic_progression` die „Largest known AP-k“ aufgelistet. Also ist dort das letzte Folgenelement eine möglichst große Primzahl.
Tabelle 3.3 listet jedoch die Folgen auf, die die kleinsten bekannten Differenzen haben – für eine gegebene Folgenlänge.

[53] In seinen Memoiren hat Hardy 1940 geschrieben, dass die Mathematik mehr als alle anderen Wissenschaften und Künste ein Spiel für junge Leute sei.
Der damals 27 Jahre alte Ben Green von der University of British Columbia in Vancouver und der damals 29 Jahre alte Terence Tao von der University of California in Los Angeles scheinen ihm recht zu geben.

Elemente	Startelement	Abstand	Wann Digits	Entdecker
3	3	2	1	
4	5	6	2	
5	5	6	2	
6	7	30	1909 3	G. Lenaire
7	7	150	1909 3	G. Lenaire
.....				
21	28.112.131.522.731.197.609	9.699.690 = 19#	2008 20	Jaroslaw Wroblewski
22	166.537.312.120.867	96.599.212.710 = 9.959·19#	2006 15	Markus Frind
23	403.185.216.600.637	2.124.513.401.010 = 9.523·23#	2006 15	Markus Frind,
24	515.486.946.529.943	30.526.020.494.970 = 136.831·23#	2008 16	Raanan Chermoni, Jaroslaw Wroblewski
25	6.171.054.912.832.631	81.737.658.082.080 = 366.384·23#	2008 16	Raanan Chermoni, Jaroslaw Wroblewski

Tabelle 3.3: Arithmetische Primzahlfolgen mit minimaler Distanz (Stand Aug. 2012)

Wer solche Folgen entdecken möchte, sollte folgendes berücksichtigen. Die Länge der Folge bestimmt den Mindestabstand zwischen den einzelnen Primzahlen. Bei einer Folge mit $k = 6$ Gliedern muss der Abstand 30 oder ein Vielfaches davon betragen. Die Zahl 30 ergibt sich als das Produkt aller Primzahlen, die kleiner als die Folgenlänge, also kleiner als 6, sind: $6\# = 5\# = 2 * 3 * 5 = 30$. Noch ein Beispiel: $10\# = 7\# = 2 * 3 * 5 * 7 = 210$. Sucht man Folgen mit der Länge 15, so muss der Abstand mindestens $15\# = 13\# = 2 * 3 * 5 * 7 * 11 * 13 = 30.030$ betragen.

Daraus ergibt sich, dass die Folgenlänge beliebig groß sein kann, aber der Abstand kann nicht jede beliebige Zahl annehmen: es kann keine arithmetische Primzahlfolge mit dem Abstand 100 geben, denn 100 ist nicht durch die Zahl 3 teilbar.

k	k#
2	2
3	6
5	30
7	210
11	2.310
13	30.030
17	510.510
19	9.699.690
23	223.092.870

Tabelle 3.4: Produkte der ersten Primzahlen <= k (genannt k Primorial oder $k\#$)

Weitere Restriktion an die arithmetische Folge:
Sucht man arithmetische Primzahlfolgen, die die *zusätzliche* Bedingung erfüllen, dass alle Primzahlen in der Folge auch *direkt hintereinander* liegen (consecutive primes sequence[54]), wird es noch etwas schwieriger. Auf der Webseite von Chris Caldwell[55] finden Sie weitere Informationen: Die längste bekannte arithmetische Folge, die nur aus direkt hintereinander liegenden Primzahlen besteht (Stand Aug. 2012), hat die Länge 10, der Abstand beträgt

$$10\# = 7\# = 2 * 3 * 5 * 7 = 210$$

und sie startet mit der 93-stelligen Primzahl
100 9969724697 1424763778 6655587969 8403295093 2468919004 1803603417 7589043417 0334888215 9067229719

[54]Sie werden auch consecutive prime arithmetic progressions genannt und daher CPAP-k bzw. CAP-k abgekürzt.
[55]`http://primes.utm.edu/glossary/page.php?sort=ArithmeticSequence`

3.8.2 Verschiedene unbewiesene Aussagen / Vermutungen / offene Fragestellungen zu Primzahlen[56]

- Goldbach[57] vermutete: Jede gerade natürliche Zahl größer 2 lässt sich als die Summe zweier Primzahlen darstellen.[58]
- Riemann[59] stellte eine wichtige, bisher unbewiesene Hypothese[60] über die Nullstellen der Riemannschen Zetafunktion auf, aus der auch eine verbesserte Restgliedabschätzung im Primzahlsatz (Verteilung von Primzahlen) folgt.
- Das Benfordsche Gesetz[61,62] gilt nicht für Primzahlen.

  Nach dem Benfordschen Gesetz sind die Ziffern in den Zahlen bestimmter empirischer Datensätze (z.B. über Einwohnerzahlen von Städten, Geldbeträge in der Buchhaltung, Naturkonstanten) ungleichmäßig verteilt: Z.B. ist die Ziffer `1` viel häufiger die erste Ziffer einer Zahl als jede andere.

  Welche Datensätze diesem Gesetz gehorchen ist noch nicht vollständig geklärt. Timo Eckhardt untersuchte in seiner Diplomarbeit 2008 ausführlich Eigenschaften von Primzahlen. Unter anderem wurden alle Primzahlen bis 7.052.046.499 mit verschiedenen Stellenwert-Basen dargestellt.

  Beim Vergleich der Basen 3 bis 10 ergab sich, dass die Abweichung von Benfords Gesetz bei der Basis 3 am geringsten ist. Für die Basis zehn besteht in etwa eine Gleichverteilung der ersten Ziffern. Bei der Untersuchung größerer Basen ergab sich, dass die Verteilung der ersten Ziffern von Basis zu Basis sehr starke Unterschiede aufweist.
- Der in Kapitel 3.8.1 erwähnte Beweis zu der Funktion $f : N \to Z$ mit $n \mapsto \lfloor a^{3^n} \rfloor$ garantiert nur die Existenz einer solchen Zahl a. Wie kann diese Zahl a bestimmt werden, und wird sie einen Wert haben, so dass die Funktion auch von praktischem Interesse ist?
- Gibt es unendlich viele Mersenne-Primzahlen?
- Gibt es unendlich viele Fermatsche Primzahlen?

---

[56] In seinem populärwissenschaftlichen Buch „Die Musik der Primzahlen“ beschreibt Marcus du Sautoy, Mathematikprofessor in Oxford, wie sich die brillantesten mathematischen Köpfe mit verschiedenen Aspekten der Primzahlen beschäftigten. Dabei stellt er diese Menschen vor (Gauß, Euler, Riemann, Ramanujan, Gödel, Connes, ...) und widmet sich insbesondere der „Riemannschen Vermutung“. Siehe [dS05].
Einfache Erläuterungen und Literaturhinweise zur Riemannschen Vermutung findet man auch in *RSA & Co. in der Schule*, NF Teil 4 [WS10a].

[57] Christian Goldbach, deutscher Mathematiker, 18.03.1690−20.11.1764.

[58] Vergleiche auch Kapitel 3.8.3.

[59] Bernhard Riemann, deutscher Mathematiker, 17.9.1826−20.7.1866.

[60] `http://de.wikipedia.org/wiki/Riemannsche_Vermutung`

[61] `http://de.wikipedia.org/wiki/Benfordsches_Gesetz`,
`http://www.spiegel.de/wissenschaft/mensch/0,1518,632541,00.html`,
`http://arxiv.org/PS_cache/arxiv/pdf/0906/0906.2789v1.pdf`.

[62] Didaktische Darstellungen zu Anwendungen von Benfords Gesetz finden sich unter:
- Rüdeger Baumann: „Ziffernanalyse zwecks Betrugsaufdeckung — Beispiel für kompetenzorientierten und kontextbezogenen Informatikunterricht“,
  in LOGIN, Informatische Bildung und Computer in der Schule, Nr. 154/155, 2008, S. 68-72
- Norbert Hungerbühler: „Benfords Gesetz über führende Ziffern“, März 2007,
  `https://www.ethz.ch/content/dam/ethz/special-interest/dual/educeth-dam/documents/Unterrichtsmaterialien/mathematik/Benfords%20Gesetz%20%C3%BCber%20f%C3%BChrende%20Ziffern%20(Artikel)/benford.pdf`

- Gibt es einen Polynomialzeit-Algorithmus zur Zerlegung einer Zahl in ihre Primfaktoren (vgl. [KW97, S. 167])? Diese Frage kann man auf die folgenden drei Fragestellungen aufsplitten:
  - Gibt es einen Polynomialzeit-Algorithmus, der entscheidet, ob eine Zahl prim ist? Diese Frage wurde durch den AKS-Algorithmus beantwortet (vgl. Kapitel 4.11.5.3, „PRIMES in P“: Testen auf Primalität ist polynominal).
  - Gibt es einen Polynomialzeit-Algorithmus, mit dem man feststellen kann, aus wievielen Primfaktoren eine zusammengesetzte Zahl besteht (ohne diese Primfaktoren zu berechnen)?
  - Gibt es einen Polynomialzeit-Algorithmus, mit dem sich für eine zusammengesetzte Zahl n ein nicht-trivialer (d.h. von 1 und von n verschiedener) Teiler von n berechnen lässt?[63]

  Am Ende von Kapitel 4.11.4, Abschnitt RSA-200 können Sie die Größenordnungen ersehen, für die heutige Algorithmen bei Primzahltests und bei der Faktorisierung gute Ergebnisse liefern.

3.8.3 Die Goldbach-Vermutung

Hier soll noch etwas tiefer auf die **Goldbach-Vermutung**[64] eingegangen werden.

3.8.3.1 Die schwache Goldbach-Vermutung[65]

Goldbach stellte 1742 in einem Brief an den Mathematiker Euler die Behauptung auf:

Jede **ungerade** natürliche Zahl größer als 5 ist als Summe von **genau drei** Primzahlen darstellbar.

Beispiele: $7 = 3 + 2 + 2$ oder $27 = 19 + 5 + 3$ oder $27 = 17 + 5 + 5$

Diese Behauptung ist immer noch (seit über 250 Jahren) unbewiesen.

Mit Computern ist die **schwache Goldbach-Vermutung** für alle ungeraden natürlichen Zahlen einfach bis $4*10^{18}$ (Stand April 2012) bzw. doppelt bis $4*10^{17}$ (Stand Mai 2013) verifiziert.[66]

Aus früheren Arbeiten ist bekannt, dass die schwache Goldbach-Vermutung für alle ungeraden Zahlen größer $e^{3100} \approx 2 \times 10^{1346}$ wahr ist.

Wie bei vielen berühmten Vermutungen in der Mathematik gibt es auch für die Goldbach-Vermutung eine Reihe angeblicher Beweise, die aber von der mathematischen Gemeinschaft (noch) nicht akzeptiert sind.[67]

---

[63] Vergleiche auch Kapitel 4.11.5.1 und Kapitel 4.11.4.

[64] `http://de.wikipedia.org/wiki/Goldbachsche_Vermutung`

[65] Sie wird auch **ungerade** oder **ternäre** Goldbach-Vermutung genannt.

[66] Siehe `http://sweet.ua.pt/tos/goldbach.html` von Tomás Oliveira e Silva

[67] Einer davon ist in dem Paper von Shan-Guang Tan, das am 16. Okt. 2011 (v1) veröffentlicht, und zuletzt am 20. Mai 2016 (v19) revidiert wurde. Es behauptet, sogar die starke Goldbach-Vermutung zu beweisen.
v1 hat den Titel „A proof of the Goldbach conjecture“.
v19 hat den Titel „On the representation of even numbers as the sum and difference of two primes and the representation of odd numbers as the sum of an odd prime and an even semiprime and the distribution of primes in short intervals“.
Siehe `http://arxiv.org/abs/1110.3465`.

Eine Vorarbeit für einen Beweis könnte die kürzlich[68] vorgestellte Arbeit von Terence Tao von der University of California sein. Er bewies, dass sich jede **ungerade** natürliche Zahl größer 1 als Summe von **höchstens fünf** Primzahlen darstellen lässt.

Inzwischen gab es erhebliche Arbeiten an der schwachen Goldbach-Vermutung. Sie kulminierten 2013 in der Behauptung von Harald Helfgott, die Vermutung komplett für alle natürlichen Zahlen größer 7 bewiesen zu haben.[69]

3.8.3.2 Die starke Goldbach-Vermutung[70]

Goldbachs starke Primzahlhypothese wurde von Euler nach einem Briefwechsel mit Goldbach formuliert, und wird meist einfach als **die** Goldbach-Vermutung bezeichnet:

Jede **gerade** natürliche Zahl größer als 2 ist als Summe von **genau zwei** Primzahlen darstellbar.

Beispiele für Goldach-Zerlegungen: $8 = 5 + 3$ oder $28 = 23 + 5$

Mit Computern ist die **Goldbach-Vermutung** für alle geraden Zahlen bis $4 * 10^{18}$ (Stand Mai 2013) einfach verifiziert[71], aber allgemein noch nicht bewiesen.[72,73,74]

---

[68] `http://arxiv.org/abs/1201.6656`, eingereicht am 31. Jan. 2012 (v1), letzte Änderung 3. Juli 2012 (v4)

[69] Helfgott, H.A. (2013): „Major arcs for Goldbach's theorem". `http://arxiv.org/abs/1305.2897`.
v4 seines 79-seitigen Papers (14. April 2014) besagt, dass der Beweis für alle natürlichen Zahlen ab 10^{29} (statt bisher 10^{30}) gilt. Und für alle Zahlen bis 10^{30} behauptet er, dass er und David Platt den Beweis durch Probieren am Computer erbrachten.
`http://truthiscool.com/prime-numbers-the-271-year-old-puzzle-resolved`.
`http://www.newscientist.com/article/dn23535-proof-that-an-infinite-number-of-primes-are-paired.html#.UgwhOpL0Ek0`.
`http://www.spiegel.de/wissenschaft/mensch/beweis-fuer-schwache-goldbachsche-vermutung-a-901111.html`.

[70] Sie wird auch **gerade** oder **binäre** Goldbach-Vermutung genannt.

[71] Dass die Goldbach-Vermutung wahr ist, d.h. für alle geraden natürlichen Zahlen größer als 2 gilt, wird heute allgemein nicht mehr angezweifelt. Der Mathematiker Jörg Richstein vom Institut für Informatik der Universität Gießen hat 1999 die geraden Zahlen bis 400 Billionen ($4 * 10^{14}$) untersucht ([Ric01]) und kein Gegenbeispiel gefunden.
Inzwischen wurden noch umfangreichere Überprüngen bis vorgenommem: Siehe
`http://sweet.ua.pt/tos/goldbach.html` von Tomás Oliveira e Silva,
`http://de.wikipedia.org/wiki/Goldbachsche_Vermutung`,
`http://primes.utm.edu/glossary/page.php/GoldbachConjecture.html`.
Trotzdem ist das kein allgemeiner Beweis.
Dass die Goldbach-Vermutung trotz aller Anstrengungen bis heute nicht bewiesen wurde, lässt Folgendes befürchten: Seit den bahnbrechenden Arbeiten des österreichischen Mathematikers Kurt Gödel ist bekannt, dass nicht jeder wahre Satz in der Mathematik auch beweisbar ist (siehe `https://www.mathematik.ch/mathematiker/goedel.php`). Möglicherweise hat Goldbach also Recht, und trotzdem wird nie ein Beweis gefunden werden. Das wiederum lässt sich aber vermutlich auch nicht beweisen.

[72] Der englische Verlag *Faber* und die amerikanische Verlagsgesellschaft *Bloomsbury* publizierten 2000 das 1992 erstmals veröffentlichte Buch „Onkel Petros und die Goldbachsche Vermutung" von Apostolos Doxiadis (deutsch bei Lübbe 2000 und bei BLT als Taschenbuch 2001). Es ist die Geschichte eines Mathematikprofessors, der daran scheitert, ein mehr als 250 Jahre altes Rätsel zu lösen.
Um die Verkaufszahlen zu fördern, schrieben die beiden Verlage einen Preis von 1 Million USD aus, wenn jemand die Vermutung beweist – veröffentlicht in einer angesehenen mathematischen Fachzeitschrift bis Ende 2004.
Erstaunlicherweise durften nur englische und amerikanische Mathematiker daran teilnehmen.

[73] Die Aussage, die der starken Goldbach-Vermutung bisher am nächsten kommt, wurde 1966 von Chen Jing-Run bewiesen – in einer schwer nachvollziehbaren Art und Weise: Jede gerade Zahl größer 2 ist die Summe einer Primzahl und des Produkts zweier Primzahlen. Z.B. $20 = 5 + 3 * 5$.
Die wichtigsten Forschungsergebnisse zur Goldbach-Vermutung sind zusammengefasst in dem von Wang Yuan herausgegebenen Band: „Goldbach Conjecture", 1984, World Scientific Series in Pure Maths, Vol. 4.

[74] Gerade diese Vermutung legt nahe, dass wir auch heute noch nicht in aller Tiefe den Zusammenhang zwischen der Addition und der Multiplikation der natürlichen Zahlen verstehen.

Je größer eine gerade Zahl ist, umso mehr solcher binärer Goldbach-Zerlegungen lassen sich im Durchschnitt finden: Für 4 gibt es nur eine Zerlegung 2+2; bei 16 sind es schon zwei, nämlich $3 + 13$ und $5 + 11$. Bei 100 sind es sechs $3 + 97$, $11 + 89$, $17 + 83$, $29 + 71$, $41 + 59$, $47 + 53$.[75]

3.8.3.3 Zusammenhang zwischen den beiden Goldbach-Vermutungen

Wenn die starke Goldbach-Vermutung gilt, gilt auch die schwache (d.h. die starke impliziert die schwache Vermutung).

Der Beweis dafür ist relativ einfach:
Voraussetzung: Sei u eine ungerade Zahl größer als 5.
Jede solche ungerade Zahl u kann als Summe $u = (u - 3) + 3$ geschrieben werden. Der erste Summand ist dann gerade und >= 4, erfüllt damit die Voraussetzung der starken Goldbach-Vermutung und kann damit als Summe zweier Primzahlen p_1 und p_2 geschrieben werden (wobei p_1 und p_2 nicht notwendigerweise verschieden sein müssen). Damit hat man eine Zerlegung von u in die drei Primzahlen p_1, p_2 und 3 gefunden. D.h. man kann sogar immer eine Summe finden, in der einer der drei primen Summanden die Zahl 3 ist.

Ähnlich einfach kann man zeigen, dass aus der schwachen Golfbach-Vermutung die oben erwähnte Behauptung von Terence Tao folgt (beide gelten für ungerade Zahlen):

- Für ungerade Zahlen $u > 5$ folgt direkt aus der schwachen Goldbach-Vermutung, dass die Summe aus höchstens fünf Primzahlen besteht.
- Für die restlichen ungeraden Zahlen 3 und 5 kann man es direkt einsetzen:
  $3 = 3$ (die „Summe“ hat nur einen und damit höchstens fünf prime Summanden);
  $5 = 2 + 3$ (die Summe hat zwei und damit höchstens fünf prime Summanden).

3.8.4 Offene Fragen zu Primzahlzwillingen und Primzahl-Cousins

Primzahlzwillinge sind Primzahlen, die genau den Abstand 2 voneinander haben, zum Beispiel 5 und 7, oder 101 und 103, oder $1.693.965 \cdot 2^{66.443} \pm 1$.

Das größte heutzutage bekannte Primzahlzwillingspaar

$$3.756.801.695.685 \cdot 2^{666.669} \pm 1$$

wurde im Dezember 2011 gefunden und hat 200.700 Dezimalstellen.[76]

Offen ist:

[75] Vergleiche `http://www.spiegel.de/wissenschaft/mensch/primzahlraetsel-loesung-der-goldbachschen-vermutung-rueckt-naeher-a-833216.html`. Dieser Artikel gehört in die Reihe der gut lesbaren Numerator-Kolumnen in Spiegel-Online von Holger Dambeck.

[76] `http://primes.utm.edu/primes`, `http://www.primegrid.com/download/twin-666669.pdf`

- Gibt es unendlich viele oder eine begrenzte Anzahl von Primzahlzwillingen?[77,78,79]
- Gibt es eine Formel für die Anzahl der Primzahlzwillinge pro Intervall?

Im Folgenden werden zwei größere Meilensteine erläutert, die dem Rätsel näher kommen.

3.8.4.1 GPY 2003

Einen großen Schritt zur Klärung der ersten Frage machten möglicherweise Dan Goldston, János Pintz und Cem Yildirim im Jahre 2003.[80] Die drei Mathematiker beschäftigten sich mit der Verteilung von Primzahlen. Sie konnten beweisen, dass

$$\liminf_{n\to\infty} \frac{p_{n+1} - p_n}{\log p_n} = 0,$$

wobei p_n die n-te Primzahl bezeichnet.
Dies bedeutet, dass der kleinste Häufungspunkt (lim inf) der Folge $\frac{p_{n+1}-p_n}{\log p_n}$ gleich Null ist.
Ein Punkt heißt Häufungspunkt einer Folge, wenn in jeder noch so kleinen Umgebung um diesen Punkt unendlich viele Folgeglieder liegen.
$\log p_n$ ist in etwa der erwartete Abstand zwischen der Primzahl p_n und der darauf folgenden Primzahl p_{n+1}.
Der obige Term besagt also, dass es unendlich viele aufeinander folgende Primzahlen gibt, deren Abstand im Verhältnis zum erwarteten durchschnittlichen Abstand beliebig nah an Null bzw. beliebig klein ist.

Außerdem konnte gezeigt werden, dass für unendlich viele Primzahlen gilt[81]:

$$p_{n+1} - p_n < (\log p_n)^{8/9}$$

3.8.4.2 Zhang 2013

Im Mai 2013 wurde die Arbeit von Yitang Zhang bekannt.[82] Zhang beweist, dass es unendlich viele „Primzahl-Cousins“ gibt, oder genauer, dass es unendlich viele Primzahlpaare gibt, die

---

[77] Bemerkung: Primzahl-**Drillinge** gibt es dagegen nur eines: 3, 5, 7. Bei allen anderen Dreierpacks aufeinanderfolgender ungerader Zahlen ist immer eine durch 3 teilbar und somit keine Primzahl.

[78] Die Vermutung, dass es unendlich viele Primzahlzwillinge gibt, ist nicht selbstverständlich. Man weiß, dass bei großen Zahlen im Durchschnitt der Abstand zwischen Primzahlen immer weiter wächst und ca. 2,3 mal so groß ist wie die Anzahl der Dezimalstellen. Beispielsweise beträgt bei 100-stelligen Dezimalzahlen der Abstand zwischen Primzahlen im Durchschnitt 230. Aber diese Aussage gilt nur für den Durchschnitt – oft ist der Abstand viel größer, oft viel kleiner.

[79] `http://de.wikipedia.org/wiki/Primzahlzwilling`

[80] D. A. Goldston: „Gaps Between Primes“
`http://www.math.sjsu.edu/~goldston/OberwolfachAbstract.pdf`
Siehe auch:
- D. A. Goldstone: „Are There Infinitely Many Twin Primes?“,
  `http://www.math.sjsu.edu/~goldston/twinprimes.pdf`
- K. Soundararajan: „Small Gaps Between Prime Numbers: The Work Of Goldston-Pintz-Yildirim“,
  `http://www.ams.org/bull/2007-44-01/S0273-0979-06-01142-6/S0273-0979-06-01142-6.pdf`

[81] c't 2003, Heft 8, Seite 54

[82] Erica Klarreich (19. Mai 2013): „Unheralded Mathematician Bridges the Prime Gap“
`https://www.simonsfoundation.org/quanta/20130519-unheralded-mathematician-bridges-the-prime-gap/`

einen Abstand von H haben und dass die Zahl H kleiner als 70 Millionen ist.[83,84,85]

Diese Aussagen könnten Grundlage sein um zu beweisen, dass es unendlich viele Primzahlzwillinge gibt.

[83] Während bei einem Primzahl-Zwilling der Abstand der beiden Primzahlen genau 2 ist, bezeichnen Primzahl-Cousins zwei Primzahlen, deren Abstand den Wert einer größeren geraden, aber endlichen Zahl H hat.

[84] Dies ist nahezu der Beweis für die Vermutung, die 1849 von dem französichen Mathematiker Alphonse de Polignac aufgestellt wurde, dass es unendlich viele Primzahl-Paare gibt zu jeder möglichen geraden und endlichen Lücke (nicht nur für die 2).

[85] In weiteren Arbeiten wurde dieser Mindestabstand H von 70 Millionen weiter verbessert. Diese Fortschritte werden im 8. Polymath-Projekt (massively collaborative online mathematical projects) dokumentiert: "Bounded gaps between primes". Der beste bisher bekannte Werte für H war 4680 (Stand August 2013) und ist 246 (Stand April 2014) – das ist ein guter Fortschritt gegenüber 70 Millionen, aber immer noch weit entfernt von 2.
Siehe `http://michaelnielsen.org/polymath1/index.php?title=Bounded_gaps_between_primes`

3.8.5 Kurioses und Interessantes zu Primzahlen[86]

Primzahlen sind nicht nur ein sehr aktives und ernstes mathematisches Forschungsgebiet. Mit ihnen beschäftigen sich Menschen auch hobbymäßig und außerhalb der wissenschaftlichen Forschung.

3.8.5.1 Mitarbeiterwerbung bei Google im Jahre 2004

Im Sommer 2004 benutzte die Firma Google die Zahl e, um Bewerber zu gewinnen.[87,88]

Auf einer hervorstechenden Reklamewand im kalifornischen Silicon Valley erschien am 12. Juli das folgende geheimnisvolle Rätsel:

(first 10-digit prime found in consecutive digits of e).com

In der Dezimaldarstellung von e sollte man also die erste 10-stellige Primzahl finden, die sich in den Nachkommastellen befindet. Mit verschiedenen Software-Tools kann man die Antwort finden:

7.427.466.391

Wenn man dann die Webseite *www*.7427466391.*com* besuchte, bekam man ein noch schwierigeres Rätsel gezeigt. Wenn man auch dieses zweite Rätsel löste, kam man zu einer weiteren Webseite, die darum bat, den Lebenslauf an Google zu senden. Diese Werbekampagne erzielte eine hohe Aufmerksamkeit.

Wahrscheinlich nahm Google an, dass man gut genug ist, für sie zu arbeiten, wenn man diese Rätsel lösen kann. Aber bald konnte jeder mit Hilfe der Google-Suche ohne Anstrengung die Antworten finden, weil viele die Lösung der Rätsel ins Netz gestellt hatten.[89]

3.8.5.2 Contact [Film, 1997] – Primzahlen zur Kontaktaufnahme

Der Film von Regiseur Robert Zemeckis entstand nach dem gleichnamigen Buch von Carl Sagan.

Die Astronomin Dr. Ellie Arroway (Jodie Foster) entdeckt nach jahrelanger vergeblicher Suche Signale vom 26 Lichtjahre entfernten Sonnensystem Wega. In diesen Signalen haben Außerirdische die Primzahlen lückenlos in der richtigen Reihenfolge verschlüsselt. Daran erkennt die Heldin, dass diese Nachricht anders ist als die ohnehin ständig auf der Erde eintreffenden

[86] Weitere kuriose und seltsame Dinge zu Primzahlen finden sich auch unter:
- `http://primes.utm.edu/curios/home.php`
- `http://www.primzahlen.de`.

[87] Die Basis des natürlichen Logarithmus e ist ungefähr 2,718 281 828 459. Dies ist eine der bedeutendsten Zahlen in der Mathematik. Sie wird gebraucht für komplexe Analysis, Finanzmathematik, Physik und Geometrie. Nun wurde sie – meines Wissens – das erste Mal für Marketing oder Personalbeschaffung verwendet.

[88] Die meisten Informationen für diesen Paragraphen stammen aus dem Artikel „e-number crunching“ von John Allen Paulos in TheGuardian vom 30.09.2004 und aus dem Internet:
- `https://mkaz.tech/google-billboard-problems.html`
- `http://epramono.blogspot.com/2004/10/7427466391.html`
- `http://mathworld.wolfram.com/news/2004-10-13/google/`.

[89] Im zweiten Level der Rätsel musste man das 5. Glied einer gegebenen Zahlenfolge finden: Dies hatte nichts mehr mit Primzahlen zu tun.

Radiowellen, die kosmischer und zufälliger Natur sind (von Radiogalaxien, Pulsaren oder Quasaren). In einer entlarvenden Szene fragt sie daraufhin ein Politiker, warum diese intelligenten Wesen nicht gleich Englisch sprechen ...

Eine Kommunikation mit absolut fremden und unbekannten Wesen aus dem All ist aus 2 Gründen sehr schwierig: Zunächst kann man sich bei der hier angenommenen Entfernung und dem damit verbundenen langen Übertragungsweg in einem durchschnittlichen Menschenleben nur einmal in jeder Richtungen nacheinander austauschen. Zum Zweiten muss man für den Erstkontakt darauf achten, dass eine möglichst hohe Chance besteht, dass der Empfänger der Radiowellen die Botschaft überhaupt bemerkt und dass er sie als Nachricht von intelligenten Wesen einstuft. Deshalb senden die Außerirdischen am Anfang ihrer Nachricht Zahlen, die als der einfachste Teil jeder höheren Sprache angesehen werden können und die nicht ganz trivial sind: die Folge der Primzahlen. Diese speziellen Zahlen spielen in der Mathematik eine so fundamentale Rolle, dass man annehmen kann, dass sie jeder Spezies vertraut sind, die das technische Know-how hat, Radiowellen zu empfangen.

Die Aliens schicken danach den Plan zum Bau einer mysteriösen Maschine ...

3.9 Anhang: Anzahl von Primzahlen in verschiedenen Intervallen

Zehnerintervall		Hunderterintervall		Tausenderintervall	
Intervall	Anzahl	Intervall	Anzahl	Intervall	Anzahl
1-10	4	1-100	25	1-1.000	168
11-20	4	101-200	21	1.001-2.000	135
21-30	2	201-300	16	2.001-3.000	127
31-40	2	301-400	16	3.001-4.000	120
41-50	3	401-500	17	4.001-5.000	119
51-60	2	501-600	14	5.001-6.000	114
61-70	2	601-700	16	6.001-7.000	117
71-80	3	701-800	14	7.001-8.000	107
81-90	2	801-900	15	8.001-9.000	110
91-100	1	901-1.000	14	9.001-10.000	112

Tabelle 3.5: Wieviele Primzahlen gibt es innerhalb der ersten Zehner-/Hunderter-/Tausender-Intervalle?

Dimension	Intervall	Anzahl	Durchschnittl. Anzahl pro 1000
4	1 - 10.000	1.229	122,900
5	1 - 100.000	9.592	95,920
6	1 - 1.000.000	78.498	78,498
7	1 - 10.000.000	664.579	66,458
8	1 - 100.000.000	5.761.455	57,615
9	1 - 1.000.000.000	50.847.534	50,848
10	1 - 10.000.000.000	455.052.512	45,505

Tabelle 3.6: Wieviele Primzahlen gibt es innerhalb der ersten Dimensionsintervalle?

Eine Visualisierung der Anzahl von Primzahlen in höheren Intervallen von Zehnerpotenzen finden Sie in Kapitel 3.13 auf Seite 108.

3.10 Anhang: Indizierung von Primzahlen (n-te Primzahl)

Index	Genauer Wert	Gerundeter Wert	Bemerkung
1	2	2	
2	3	3	
3	5	5	
4	7	7	
5	11	11	
6	13	13	
7	17	17	
8	19	19	
9	23	23	
10	29	29	
100	541	541	
1000	7917	7917	
664.559	9.999.991	9,99999E+06	Alle Primzahlen bis zu 1E+07 waren am Beginn des 20. Jahrhunderts bekannt.
1E+06	15.485.863	1,54859E+07	
6E+06	104.395.301	1,04395E+08	Diese Primzahl wurde 1959 entdeckt.
1E+07	179.424.673	1,79425E+08	
1E+09	22.801.763.489	2,28018E+10	
1E+12	29.996.224.275.833	2,99962E+13	

Tabelle 3.7: Liste selektierter n-ter Primzahlen P(n)

Bemerkung: Mit Lücke wurden früh sehr große Primzahlen entdeckt.

Web-Links (URLs):
`https://primes.utm.edu/nthprime/`
`https://primes.utm.edu/notes/by_year.html`.

3.11 Anhang: Größenordnungen / Dimensionen in der Realität

Bei der Beschreibung kryptographischer Protokolle und Algorithmen treten Zahlen auf, die so groß bzw. so klein sind, dass sie einem intuitiven Verständnis nicht zugänglich sind. Es kann daher nützlich sein, Vergleichszahlen aus der uns umgebenden realen Welt bereitzustellen, so dass man ein Gefühl für die Sicherheit kryptographischer Algorithmen entwickeln kann. Die angegebenen Werte stammen größtenteils aus [Sch96b] und [Sch96a, S.18].

Wahrscheinlichkeit, dass Sie auf ihrem nächsten Flug entführt werden	$5,5 \cdot 10^{-6}$
Jährliche Wahrscheinlichkeit, von einem Blitz getroffen zu werden	10^{-7}
Wahrscheinlichkeit für 6 Richtige im Lotto	$7,1 \cdot 10^{-8}$
Risiko, von einem Meteoriten erschlagen zu werden	$1,6 \cdot 10^{-12}$
Zeit bis zur nächsten Eiszeit (in Jahren)	$14.000 = (2^{14})$
Zeit bis die Sonne verglüht (in Jahren)	$10^9 = (2^{30})$
Alter der Erde (in Jahren)	$10^9 = (2^{30})$
Alter des Universums (in Jahren)	$10^{10} = (2^{34})$
Anzahl der Moleküle in einem Wassertropfen	$10^{20} = (2^{63})$
Anzahl der auf der Erde lebenden Bakterien	$10^{30,7} = (2^{102})$
Anzahl der Atome der Erde	$10^{51} = (2^{170})$
Anzahl der Atome der Sonne	$10^{57} = (2^{190})$
Anzahl der Atome im Universum (ohne dunkle Materie)	$10^{77} = (2^{265})$
Volumen des Universums (in cm^3)	$10^{84} = (2^{280})$

Tabelle 3.8: Wahrscheinlichkeiten und Größenordnungen aus Physik und Alltag

3.12 Anhang: Spezielle Werte des Zweier- und Zehnersystems

Diese Werte können dazu benutzt werden, aus einer Schlüssellänge in Bit die Anzahl möglicher Schlüssel und den Such-Aufwand abzuschätzen (wenn man z.B. annimmt, dass 1 Million Schlüssel in 1 sec durchprobiert werden können).

Dualsystem	Zehnersystem
2^{10}	1024
2^{40}	$1,09951 \cdot 10^{12}$
2^{56}	$7,20576 \cdot 10^{16}$
2^{64}	$1,84467 \cdot 10^{19}$
2^{80}	$1,20893 \cdot 10^{24}$
2^{90}	$1,23794 \cdot 10^{27}$
2^{112}	$5,19230 \cdot 10^{33}$
2^{128}	$3,40282 \cdot 10^{38}$
2^{150}	$1,42725 \cdot 10^{45}$
2^{160}	$1,46150 \cdot 10^{48}$
2^{192}	$6,27710 \cdot 10^{57}$
2^{250}	$1,80925 \cdot 10^{75}$
2^{256}	$1,15792 \cdot 10^{77}$
2^{320}	$2,13599 \cdot 10^{96}$
2^{512}	$1,34078 \cdot 10^{154}$
2^{768}	$1,55252 \cdot 10^{231}$
2^{1024}	$1,79769 \cdot 10^{308}$
2^{2048}	$3,23170 \cdot 10^{616}$

Tabelle 3.9: Spezielle Werte des Zweier- und Zehnersystems

Solche Tabellen lassen sich mit Computer-Algebra-Systemen einfach berechnen. Hier ein Codebeispiel für das CAS-System SageMath:

SageMath-Beispiel 3.1 Spezielle Werte des Zweier- und Zehnersystems

```
E = [10, 40, 56, 64, 80, 90, 112, 128, 150, 160, 192, 256, 1024, 2048]
for e in E:
    # print "2^" + str(e), "---", 1.0*(2^e)
    print "2^%4d" % e , " --- ", RR(2^e).n(24)
....:
2^  10  ---  1024.00
2^  40  ---  1.09951e12
2^  56  ---  7.20576e16
2^  64  ---  1.84467e19
2^  80  ---  1.20893e24
2^  90  ---  1.23794e27
2^ 112  ---  5.19230e33
2^ 128  ---  3.40282e38
2^ 150  ---  1.42725e45
2^ 160  ---  1.46150e48
2^ 192  ---  6.27710e57
2^ 256  ---  1.15792e77
2^1024  ---  1.79769e308
2^2048  ---  3.23170e616
```

3.13 Anhang: Visualisierung der Menge der Primzahlen in hohen Bereichen

Zur Verteilung von Primzahlen

Zwischen 1 und 10 gibt es 4 Primzahlen. Zwischen 10^3 und 10^4 sind es schon 1061. Im Intervall $[10^9, 10^{10}]$ liegen $404.204.977 \approx 4 \cdot 10^8$ Primzahlen, und in dem Intervall von 10^{19} bis 10^{20} sind es schon $1.986.761.935.284.574.233 \approx 1,9 \cdot 10^{18}$ Primzahlen.[90]

Warum unterscheidet sich die Anzahl der Primzahlen, die in den verschiedenen Intervallen liegen so stark, obwohl sich die Intervallgrenzen jeweils nur um den Wert 1 im Exponenten der 10-er Potenz unterscheiden?

Primzahlsatz

Die Anzahl der Primzahlen PI(x) bis zu einer gegebenen Zahl x kann durch eine Formel nach dem sogenannten Primzahlsatz näherungsweise bestimmt werden (siehe Kapitel 3.7). $PI(x)$ bezeichnet die Anzahl der Primzahlen kleiner oder gleich der Zahl x. Dann lautet die Formel

$$PI(x) \sim \frac{x}{\ln x}.$$

Es ist zu beachten, dass diese Formel die Anzahl der Primzahlen kleiner oder gleich x nur ungefähr angibt. Sie wird jedoch genauer je größer x wird. Im folgenden wollen wir den Primzahlsatz nutzen, um uns die Verteilung der Primzahlen anzusehen.

Um zu verstehen, warum die Anzahl der Primzahlen so schnell wächst, obwohl sich die Intervallgrenzen jeweils nur um einen Exponentenwert von 1 unterscheiden, werfen wir einen Blick auf beide Komponenten der rechten Seite der Formel: x und $\ln x$.

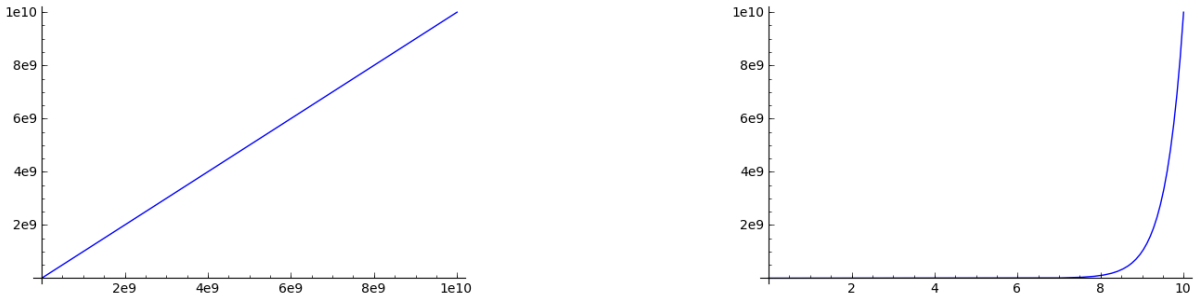
Die Funktionen x und 10^x

Die Funktion x ist eine Gerade. Sie ist in Abbildung 3.6a auf Seite 107 zu sehen.
Als nächstes tragen wir die Funktion der Intervallgrenzen in einem Graphen ab, der in Abbildung 3.6b auf Seite 107 zu finden ist. Um einen Eindruck zu bekommen, wie die Funktionen aussehen, wurde der Definitionsbereich von 0 bis 10^{10} bzw. entprechend von 0 bis 10 gewählt. Es ist zu sehen, dass mit steigendem Exponenten x die Zahlen immer größer werden.

Die Funktion $\ln x$

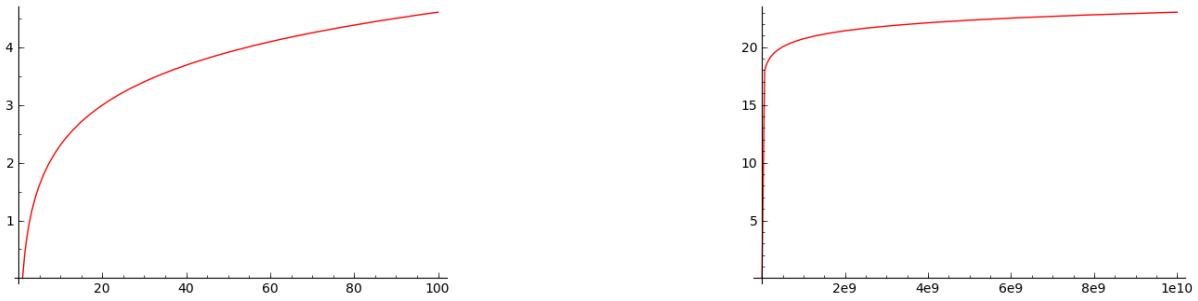
Im Vergleich dazu betrachten wir die Funktion $\ln x$. Auf dem linken Bild von Abbildung 3.7 auf Seite 107 wurde der Definitionsbereich von 1 bis 100 gewählt. Das rechte Bild zeigt die Werte der Funktion bis 10^{10}.
Es ist gut zu erkennen, dass die Werte der Funktion $\ln x$ im Vergleich zu x langsam wachsen. Dies verdeutlicht auch der Graph beider Funktionen in Abbildung 3.8 auf Seite 107. Zudem wurde in diesen Graphen auch die Funktion $\frac{x}{\ln x}$ eingezeichnet.

[90] `http://de.wikipedia.org/wiki/Primzahlsatz`



(a) x (b) 10^x

Abbildung 3.6: Graph der Funktionen x und 10^x



(a) (b)

Abbildung 3.7: Graph der Funktion $\ln x$ bis 100 und bis 10^{10}

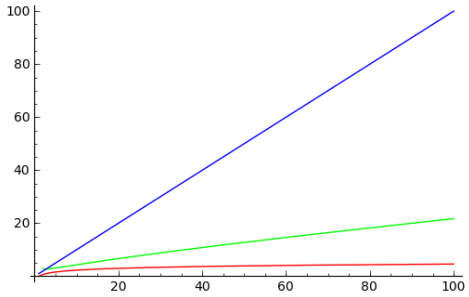


Abbildung 3.8: Die Funktionen x (blau), $\ln x$ (rot) und $\frac{x}{\ln x}$ (grün)

Die Funktion $PI(x) = \frac{x}{\ln x}$

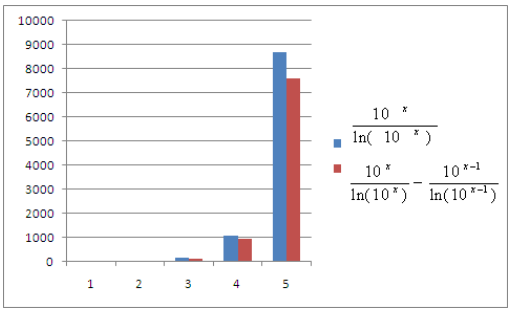
Die Funktion $\frac{x}{\ln x}$ setzt sich also zusammen aus der Funktion x im Zähler und der im Verhältnis dazu sehr langsam wachsenden Funktion $\ln x$ im Nenner. Die Anzahl der Primzahlen kleiner oder gleich einer Zahl x ist zwar verhältnismäßig klein im Vergleich zu x selbst. Dennoch ist $\frac{x}{\ln x}$ eine wachsende Funktion. Dies wird auch deutlich in Abbildung 3.8 auf Seite 107.

Die Anzahl der Primzahlen in verschiedenen Intervallen

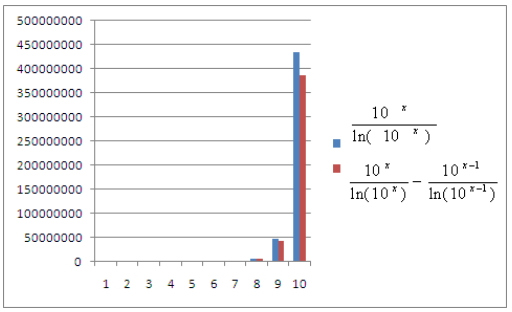
Abbildung 3.9 veranschaulicht, wie sich die Anzahl der Primzahlen in den Intervallen $[1, 10^x]$ und $[10^{x-1}, 10^x]$ entwickelt. Um es schneller berechnen zu können, wird nicht die exakte Anzahl (wie in den Tabellen in Kapitel 3.9) benutzt, sondern der Wert der Näherungsfunktion.

Dabei werden pro Zehner-Exponenten zwei Balken gezeichnet: $\frac{10^x}{\ln 10^x}$ im Vergleich zu $\frac{10^x}{\ln 10^x} - \frac{10^{x-1}}{\ln 10^{x-1}}$: Die linke Grafik für x von 1 bis 5, und die rechte für x von 1 bis 10, wobei x der Wert des Zehner-Exponenten ist.

Die blauen Balken geben an, wieviele Primzahlen es ingesamt bis 10^x gibt. Die roten Balken zeigen, wieviele Primzahlen jeweils im letzten Intervall $[10^{x-1}, 10^x]$ hinzukommen. Dies verdeutlicht, dass die Anzahl der Primzahlen in Intervallen mit höheren Exponenten immer noch stark steigt.



(a)



(b)

Abbildung 3.9: Anzahl der Primzahlen im Intervall $[1, 10^x]$ (blau) und im Intervall $[10^{x-1}, 10^x]$ (rot) (für verschiedene Exponenten x)

Eine Tabelle mit den Anzahlen von Primzahlen in einigen ausgewählten Intervallen finden Sie in Kapitel 3.9 auf Seite 102: Z.B. liegen im Intervall $[1, 10^4]$ 1229 Primzahlen; davon befinden sich im Intervall $[10^3, 10^4]$ 1229 - 168 = 1061 Primzahlen.

Theorie zum Primzahlsatz und zur Funktion PI(x) steht in Kapitel 3.7.

SageMath-Beispiel 3.2 Erzeugen der Graphen zu den drei Funktionen x, log(x) und x/log(x)

```
# Definition der Funktion f(x)=x und Plots für die Definitionsbereiche 0 bis 10^10 und 0 bis 100
sage: def f(x):return x
....:
sage: F=plot(f,(0,10^10))
sage: F.plot()

sage: F2=plot(f,(1,100))
sage: F2.plot()


# Definition der Funktion g(x)=10^x und Plot für den Definitionsbereich 0 bis 10
sage: def g(x): return 10^x
....:
sage: G=plot(g,(0,10))
sage: G.plot()


# Definition der Funktion h(x)=log(x) und Plots für die Definitionsbereiche 1 bis 100 und 1 bis 10^10
sage: def h(x): return log(x)
....:
sage: H=plot(h,(1,100),color="red")
sage: H.plot()

sage: H2=plot(h,(1,10^10),color="red")
sage: H2.plot()


# Definition der Funktion k(x)=x/log(x) und Plot für den Definitionsbereich 2 bis 100
sage: def k(x): return x/log(x)
....:
sage: K=plot(k,(2,100),color="green")
sage: K.plot()


# Plot der Funktionen f, k und h für den Definitionsbereich von 1 bzw. 2 bis 100
sage: F2+K+H


# Generieren der Daten für die Balkencharts ........................
# Bestimmen der Anzahl der Primzahlen im Intervall [1,10]
sage: pari(10).primepi()-pari(1).primepi()
4

# Bestimmen der Anzahl der Primzahlen im Intervall [10^3,10^4]
sage: pari(10**4).primepi()-pari(10**3).primepi()
1061

# Bestimmen der Anzahl der Primzahlen im Intervall [10^8,10^9]
sage: pari(10**9).primepi()-pari(10**8).primepi()
45086079

# (ab 10^10: OverflowError: long int too large to convert)
```

3.14 Anhang: Beispiele mit SageMath

Der folgende Abschnitt enthält SageMath Source-Code zu den Inhalten aus Kapitel 3 („Primzahlen“).

3.14.1 Einfache Funktionen zu Primzahlen mit SageMath

In diesem Teil des Anhangs finden Sie den Quellcode für SageMath, mit dem man einige einfache Berechnungen zu Primzahlen durchführen kann.[91]

SageMath-Beispiel 3.3 Einige einfache Funktionen zu Primzahlen

```
# Primzahlen (allgemeine Befehle)
# Die Menge der Primzahlen
sage: P=Primes(); P
Set of all prime numbers: 2, 3, 5, 7, ...

# Gibt die nächste Primzahl aus
sage: next_prime(5)
7

# Gibt die Anzahl der Primzahlen <=x aus
sage: pari(10).primepi()
4

# Gibt die ersten x Primzahlen aus
sage: primes_first_n(5)
[2, 3, 5, 7, 11]

# Gibt die Primzahlen in einem Interval aus
sage: list(primes(1,10))
[2, 3, 5, 7]
```

[91] Siehe die SageMath-Dokumentation über Elementare Zahlentheorie `http://doc.sagemath.org/html/en/constructions/number_theory.html`.

3.14.2 Primalitäts-Check der von einer quadratischen Funktion erzeugten Zahlen

In diesem Anhang finden Sie den Quellcode für SageMath, mit dem man z.B. prüfen kann, ob die von der Funktion $f(n) = n^2 - 9n + 61$ erzeugten Zahlen prim sind. Der Quellcode definiert eine Funktion namens `quadratic_prime_formula()`, die die folgenden drei Argumente hat:

- `start` — Ein natürliche Zahl, die die Untergrenze der Folge $\texttt{start}, \texttt{start} + 1, \texttt{start} + 2, \ldots, \texttt{end} - 1, \texttt{end}$ darstellt.
- `end` — Ein natürliche Zahl, die die Obergrenze der Folge $\texttt{start}, \texttt{start} + 1, \texttt{start} + 2, \ldots, \texttt{end} - 1, \texttt{end}$ darstellt.
- `verbose` — (Standardwert: `True`) Ein Schalter, der festlegt, ob für jede erzeugte Einzelzahl $f(n)$ eine Ausgabe zur Primalität erfolgen soll.

Eine sinnvolle Änderung dieses Codes besteht darin, eine andere Funktion anzugeben, deren Funktionswerte auf Primalität geprüft werden sollen.

SageMath-Beispiel 3.4 Testen der Primalität von Funktionswerten, erzeugt von einer quadratischen Funktion

```
def quadratic_prime_formula(start, end, verbose=True):
    print "N -- N^2 - 9*N + 61"
    P = 0 # the number of primes between start and end
    for n in xrange(start, end + 1):
        X = n^2 - 9*n + 61
        if is_prime(X):
            P += 1
            if verbose:
                 print str(n) + " -- " + str(X) + " is prime"
        else:
            if verbose:
                 print str(n) + " -- " + str(X) + " is NOT prime"
    print "Number of primes: " + str(P)
    print "Percentage of primes: " + str(float((P * 100) / (end - start + 1)))
```

Mit dem folgenden Funktionsaufruf berechnen wir die Werte von $f(n) = n^2 - 9n + 61$ für $n = 0, 1, 2, \ldots, 50$ und verifizieren die Primalität der erzeugten Funktionswerte:

```
sage: quadratic_prime_formula(0, 50)
 N -- N^2 - 9*N + 61
0 -- 61 is prime
1 -- 53 is prime
2 -- 47 is prime
3 -- 43 is prime
4 -- 41 is prime
5 -- 41 is prime
6 -- 43 is prime
7 -- 47 is prime
8 -- 53 is prime
9 -- 61 is prime
10 -- 71 is prime
11 -- 83 is prime
12 -- 97 is prime
13 -- 113 is prime
```

```
14 -- 131 is prime
15 -- 151 is prime
16 -- 173 is prime
17 -- 197 is prime
18 -- 223 is prime
19 -- 251 is prime
20 -- 281 is prime
21 -- 313 is prime
22 -- 347 is prime
23 -- 383 is prime
24 -- 421 is prime
25 -- 461 is prime
26 -- 503 is prime
27 -- 547 is prime
28 -- 593 is prime
29 -- 641 is prime
30 -- 691 is prime
31 -- 743 is prime
32 -- 797 is prime
33 -- 853 is prime
34 -- 911 is prime
35 -- 971 is prime
36 -- 1033 is prime
37 -- 1097 is prime
38 -- 1163 is prime
39 -- 1231 is prime
40 -- 1301 is prime
41 -- 1373 is prime
42 -- 1447 is prime
43 -- 1523 is prime
44 -- 1601 is prime
45 -- 1681 is NOT prime
46 -- 1763 is NOT prime
47 -- 1847 is prime
48 -- 1933 is prime
49 -- 2021 is NOT prime
50 -- 2111 is prime
Number of primes: 48
Percentage of primes: 94.1176470588
```

Die Statistik in den letzten beiden Zeilen der Ausgabe zeigt, dass $f(n)$ 48 Primzahlen erzeugt für $0 \leq n \leq 50$, was ca. 94% der von $f(n)$ erzeugten Werte darstellt.

Bei längeren Sequenzen ist die Einzelausgabe der erzeugten Zahl und ihrer Primalität ziemlich unpraktisch. Deshalb wird in der folgenden SageMath-Session nur die Statistik am Ende ausgegeben (indem man den Verbose-Parameter auf False setzt): die Gesamtzahl und der prozentuale Anteil der Primzahlen, die von $f(n)$ im Bereich $0 \leq n \leq 1000$ erzeugt wurde.

```
sage: quadratic_prime_formula(0, 1000, verbose=False)
N -- N^2 - 9*N + 61
Number of primes: 584
Percentage of primes: 58.3416583417
```

Literaturverzeichnis (Kap. Primes)

[Blu99] Blum, W.: *Die Grammatik der Logik*. dtv, 1999.

[dS05] Sautoy, Marcus du: *Die Musik der Primzahlen: Auf den Spuren des größten Rätsels der Mathematik*. Beck, 4. Auflage, 2005.

[Knu98] Knuth, Donald E.: *The Art of Computer Programming, vol 2: Seminumerical Algorithms*. Addison-Wesley, 3. Auflage, 1998.

[KW97] Klee, V. und S. Wagon: *Ungelöste Probleme in der Zahlentheorie und der Geometrie der Ebene*. Birkhäuser Verlag, 1997.

[Pad96] Padberg, Friedhelm: *Elementare Zahlentheorie*. Spektrum Akademischer Verlag, 2. Auflage, 1996.

[Ric01] Richstein, J.: *Verifying the Goldbach Conjecture up to* $4 * 10^{14}$. Mathematics of Computation, 70:1745–1749, 2001.

[Sch96a] Schneier, Bruce: *Applied Cryptography, Protocols, Algorithms, and Source Code in C*. Wiley, 2. Auflage, 1996.

[Sch96b] Schwenk, Jörg: *Conditional Access*. taschenbuch der telekom praxis. B. Seiler, Verlag Schiele und Schön, 1996.

[Sch06] Scheid, Harald: *Zahlentheorie*. Spektrum Akademischer Verlag, 4. Auflage, 2006.

[Tie73] Tietze, H.: *Gelöste und ungelöste mathematische Probleme*. C.H. Beck, 6. Auflage, 1973.

[WS10a] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 4: Gibt es genügend Primzahlen für RSA?* LOG IN, 2010(163):97–103, 2010.
`http://bscw.schule.de/pub/nj_bscw.cgi/d864891/RSA_u_Co_NF4.pdf`.

[WS10b] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 5: Der Miller-Rabin-Primzahltest oder: Falltüren für RSA mit Primzahlen aus Monte Carlo*. LOG IN, 2010(166/167):92–106, 2010.
`http://bscw.schule.de/pub/nj_bscw.cgi/d864895/RSA_u_Co_NF5.pdf`.

Alle Links wurden am 11.07.2016 überprüft.

Web-Links

1. GIMPS (Great Internet Mersenne Prime Search)
   www.mersenne.org ist die Homepage des GIMPS-Projekts,
   `http://www.mersenne.org/primes/`
2. Die Proth Search Page mit dem Windows-Programm von Yves Gallot
   `http://primes.utm.edu/programs/gallot/index.html`
3. Verallgemeinerte Fermat-Primzahlen-Suche
   `http://primes.utm.edu/top20/page.php?id=12`
4. Verteilte Suche nach Teilern von Fermatzahlen
   `http://www.fermatsearch.org/`
5. Die Universität von Tennessee hostet umfangreiche Forschungsergebnisse über Primzahlen.
   `http://www.utm.edu/`
6. Den besten Überblick zu Primzahlen (weil sehr aktuell und vollständig) bieten m.E. „The Prime Pages“ von Professor Chris Caldwell.
   `http://primes.utm.edu/`
7. Beschreibungen u.a. zu Primzahltests
   `http://primes.utm.edu/mersenne/index.html`
   `http://primes.utm.edu/prove/index.html`
8. Ausgabe der n-ten Primzahl P(n)
   `https://primes.utm.edu/notes/by_year.html`
   `https://primes.utm.edu/nthprime/`
9. Der Supercomputerhersteller SGI Cray Research beschäftigte nicht nur hervorragende Mathematiker, sondern benutzte die Primzahltests auch als Benchmarks für seine Maschinen.
   `http://www.isthe.com/chongo/tech/math/prime/prime_press.html`
10. Das Cunningham-Projekt
    `http://www.cerias.purdue.edu/homes/ssw/cun/`
11. EFF Cooperative Computing Awards
    `http://www.eff.org/awards/coop`
12. Goldbach conjecture verification project von Tomás Oliveira e Silva,
    `http://sweet.ua.pt/tos/goldbach.html`
13. Kurt Gödel
    `https://www.mathematik.ch/mathematiker/goedel.php`

Alle Links wurden am 11.07.2016 überprüft.

Dank

Für das sehr konstruktive Korrekturlesen der ersten Versionen dieses Artikels: Hr. Henrik Koy und Hr. Roger Oyono.

Kapitel 4

Einführung in die elementare Zahlentheorie mit Beispielen

(Bernhard Esslinger, Juli 2001; Updates: Nov. 2001, Juni 2002, Mai 2003, Mai 2005, März 2006, Juni 2007, Juli 2009, Jan. 2010, Aug. 2013, Juli 2016)

Diese „Einführung“ bietet einen Einstieg für mathematisch Interessierte. Erforderlich sind nicht mehr Vorkenntnisse als die, die im Grundkurs Mathematik am Gymnasium vermittelt werden.

Wir haben uns bewusst an „Einsteigern“ und „Interessenten“ orientiert, und nicht an den Gepflogenheiten mathematischer Lehrbücher, die auch dann „Einführung“ genannt werden, wenn sie schon auf der 5. Seite nicht mehr auf Anhieb zu verstehen sind und sie eigentlich den Zweck haben, dass man danach auch spezielle Monographien zu dem Thema lesen können soll.

4.1 Mathematik und Kryptographie

Ein großer Teil der modernen, asymmetrischen Kryptographie beruht auf mathematischen Erkenntnissen – auf den Eigenschaften („Gesetzen“) ganzer Zahlen, die in der elementaren Zahlentheorie untersucht werden. „Elementar“ bedeutet hier, dass die zahlentheoretischen Fragestellungen im wesentlichen in der Menge der natürlichen und der ganzen Zahlen durchgeführt werden.

Weitere mathematische Disziplinen, die heute in der Kryptographie Verwendung finden, sind (vgl. [Bau95, S. 2], [Bau00, Seite 3]) :

- Gruppentheorie
- Kombinatorik
- Komplexitätstheorie
- Stochastik (Ergodentheorie)
- Informationstheorie.

Die Zahlentheorie oder Arithmetik (hier wird mehr der Aspekt des Rechnens mit Zahlen betont) wurde von Gauss[1] als besondere mathematische Disziplin begründet. Zu ihren elementaren

[1] Carl Friedrich Gauss, deutscher Mathematiker und Astronom, 30.4.1777−23.2.1855.

Gegenständen gehören: größter gemeinsamer Teiler[2] (ggT), Kongruenzen (Restklassen), Faktorisierung, Satz von Euler-Fermat und primitive Wurzeln. Kernbegriff sind jedoch die Primzahlen und ihre multiplikative Verknüpfung.

Lange Zeit galt gerade die Zahlentheorie als Forschung pur, als Paradebeispiel für die Forschung im Elfenbeinturm. Sie erforschte die „geheimnisvollen Gesetze im Reich der Zahlen“ und gab Anlass zu philosophischen Erörterungen, ob sie beschreibt, was überall in der Natur schon da ist, oder ob sie ihre Elemente (Zahlen, Operatoren, Eigenschaften) nicht künstlich konstruiert.

Inzwischen weiß man, dass sich zahlentheoretische Muster überall in der Natur finden. Zum Beispiel verhalten sich die Anzahl der links- und der rechtsdrehenden Spiralen einer Sonnenblume zueinander wie zwei aufeinanderfolgende Fibonacci-Zahlen[3], also z.B. wie $21 : 34$.

Außerdem wurde spätestens mit den zahlentheoretischen Anwendungen der modernen Kryptographie klar, dass eine jahrhundertelang als theoretisch geltende Disziplin praktische Anwendung findet, nach deren Experten heute eine hohe Nachfrage auf dem Arbeitsmarkt besteht.

Anwendungen der (Computer-)Sicherheit bedienen sich heute der Kryptographie, weil Kryptographie als mathematische Disziplin einfach besser und beweisbarer ist als alle im Laufe der Jahrhunderte erfundenen „kreativen“ Verfahren der Substitution und besser als alle ausgefeilten physischen Techniken wie beispielsweise beim Banknotendruck [Beu96, S. 4].

In diesem Artikel werden in einer leicht verständlichen Art die grundlegenden Erkenntnisse der elementaren Zahlentheorie anhand vieler Beispiele vorgestellt – auf Beweise wird (fast) vollständig verzichtet (diese finden sich in den mathematischen Lehrbüchern).

Ziel ist nicht die umfassende Darstellung der zahlentheoretischen Erkenntnisse, sondern das Aufzeigen der wesentlichen Vorgehensweisen. Der Umfang des Stoffes orientiert sich daran, das RSA-Verfahren verstehen und anwenden zu können.

Dazu wird sowohl an Beispielen als auch in der Theorie erklärt, wie man in endlichen Mengen rechnet und wie dies in der Kryptographie Anwendung findet. Insbesondere wird auf die klassischen Public-Key-Verfahren Diffie-Hellman (DH) und RSA eingegangen.[4]

Es war mir wichtig, fundierte Aussagen zur Sicherheit des RSA-Verfahrens zu machen, und für möglichst alle Beispiele SageMath-Code anzugeben.

[2] Auf ggT, englisch gcd (greatest common divisor), geht dieser Artikel in Anhang 4.14 ein.

[3] Die Folge der Fibonacci-Zahlen $(a_i)_{i\in\mathbb{N}}$ ist definiert durch die „rekursive“ Vorschrift $a_1 := a_2 := 1$ und für alle Zahlen $n = 1, 2, 3, \cdots$ definiert man $a_{n+2} := a_{n+1} + a_n$. Zu dieser historischen Folge gibt es viele interessante Anwendungen in der Natur (siehe z.B. [GKP94, S. 290 ff] oder die Web-Seite von Ron Knott: Hier dreht sich alles um Fibonacci-Zahlen). Die Fibonacci-Folge ist gut verstanden und wird heute als wichtiges Werkzeug in der Mathematik benutzt.

[4] Das gleiche Ziele verfolgt auch die Artikelserie *RSA & Co. in der Schule: Neue Folge*. Links zu den Artikeln mit kurzer Inhaltsangabe finden sich z.B. unter `https://www.cryptoportal.org/`, Menü „Linksammlung“, Stichwort „rsa“.

Die Mathematik ist die Königin der Wissenschaften, die Zahlentheorie aber ist die Königin der Mathematik.

Zitat 7: Carl Friedrich Gauss

4.2 Einführung in die Zahlentheorie[5]

Die Zahlentheorie entstand aus Interesse an den positiven ganzen Zahlen $1, 2, 3, 4, \cdots$, die auch als die Menge der *natürlichen Zahlen* $\mathbb{N}$ bezeichnet werden. Sie sind die ersten mathematischen Konstrukte der menschlichen Zivilisation. Nach Kronecker[6] hat sie der liebe Gott geschaffen, nach Dedekind[7] der menschliche Geist. Das ist je nach Weltanschauung ein unlösbarer Widerspruch oder ein und dasselbe.

Im Altertum gab es keinen Unterschied zwischen Zahlentheorie und Numerologie, die einzelnen Zahlen mystische Bedeutung zumaß. So wie sich während der Renaissance (ab dem 14. Jahrhundert) die Astronomie allmählich von der Astrologie und die Chemie von der Alchemie löste, so ließ auch die Zahlentheorie die Numerologie hinter sich.

Die Zahlentheorie faszinierte schon immer Amateure wie auch professionelle Mathematiker. Im Unterschied zu anderen Teilgebieten der Mathematik können viele der Probleme und Sätze auch von Laien verstanden werden, andererseits widersetzten sich die Lösungen zu den Problemen und die Beweise zu den Sätzen oft sehr lange den Mathematikern. Es ist also leicht, gute Fragen zu stellen, aber es ist ganz etwas anderes, die Antwort zu finden. Ein Beispiel dafür ist der sogenannte letzte (oder große) Satz von Fermat.[8]

Bis zur Mitte des 20. Jahrhunderts wurde die Zahlentheorie als das reinste Teilgebiet der Mathematik angesehen – ohne Verwendung in der wirklichen Welt. Mit dem Aufkommen der Computer und der digitalen Kommunikation änderte sich das: die Zahlentheorie konnte einige unerwartete Antworten für reale Aufgabenstellungen liefern. Gleichzeitig halfen die Fortschritte in der EDV, dass die Zahlentheoretiker große Fortschritte machten im Faktorisieren großer Zahlen, in der Bestimmung neuer Primzahlen, im Testen von (alten) Vermutungen und beim Lösen bisher unlösbarer numerischer Probleme.

Die moderne Zahlentheorie besteht aus Teilgebieten wie

- Elementare Zahlentheorie
- Algebraische Zahlentheorie
- Analytische Zahlentheorie
- Geometrische Zahlentheorie

[5] Ein didaktisch sehr gut aufbereiteter Artikel zur elementaren Zahlentheorie mit historischen Bezügen findet sich in der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*. Siehe NF Teil 3 [WS08].

[6] Leopold Kronecker, deutscher Mathematiker, 7.12.1823−29.12.1891.

[7] Julius Wilhelm Richard Dedekind, deutscher Mathematiker, 06.10.1831−12.02.1916.

[8] In der Schul-Mathematik wird der Satz von Pythagoras behandelt: Den Schülern wird beigebracht, dass in einem ebenen, rechtwinkligen Dreieck gilt: $a^2 + b^2 = c^2$, wobei a, b die Schenkellängen sind und c die Länge der Hypothenuse ist.
Fermats berühmte Behauptung ist, dass für $a, b, c \in \mathbb{N}$ und für ganzzahlige Exponenten $n > 2$ immer die Ungleichheit $a^n + b^n \neq c^n$ gilt. Leider fand Fermat am Rand seiner Ausgabe des Buches von Diophant, wo er die Behauptung aufstellte, nicht genügend Platz, um den Satz zu beweisen. Der Satz konnte erst über 300 Jahre später bewiesen werden [Wil95, S. 433-551].

- Kombinatorische Zahlentheorie
- Numerische Zahlentheorie und
- Wahrscheinlichkeitstheorie.

Die verschiedenen Teilgebiete beschäftigen sich alle mit Fragestellungen zu den ganzen Zahlen (positive und negative ganze Zahlen und die Null), gehen diese jedoch mit verschiedenen Methoden an.

Dieser Artikel beschäftigt sich nur mit dem Teilgebiet der elementaren Zahlentheorie.

4.2.1 Konvention

Wird nichts anderes gesagt, gilt:

- Die Buchstaben $a, b, c, d, e, k, n, m, p, q$ stehen für ganze Zahlen.
- Die Buchstaben i und j stehen für natürliche Zahlen.
- Der Buchstabe p steht stets für eine Primzahl.
- Die Mengen $\mathbb{N} = \{1, 2, 3, \cdots\}$ und $\mathbb{Z} = \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}$ sind die *natürlichen* und die *ganzen* Zahlen.

Das ist nicht Zauberei, das ist Logik, ein Rätsel. Viele von den größten Zauberern haben keine Unze Logik im Kopf.

Zitat 8: Joanne K. Rowling[9]

4.3 Primzahlen und der erste Hauptsatz der elementaren Zahlentheorie

Viele der Fragestellungen in der elementaren Zahlentheorie beschäftigen sich mit Primzahlen (siehe Kapitel 3).

Jede ganze Zahl hat Teiler oder Faktoren. Die Zahl 1 hat nur einen, nämlich sich selbst. Die Zahl 12 hat die sechs Teiler 1, 2, 3, 4, 6 und 12.[10] Viele Zahlen sind nur teilbar durch sich selbst und durch 1. Bezüglich der Multiplikation sind dies die „Atome“ im Bereich der Zahlen.

Definition 4.3.1. Primzahlen *sind natürliche Zahlen größer als* 1*, die nur durch* 1 *und sich selbst teilbar sind.*

Per Definition ist 1 keine Primzahl.

Schreibt man die Primzahlen in aufsteigender Folge (Primzahlenfolge), so ergibt sich

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, \cdots .$$

Unter den ersten 100 Zahlen gibt es genau 25 Primzahlen. Danach nimmt ihr prozentualer Anteil ab, wird aber nie Null.

Primzahlen treten als ganze Zahlen nicht selten auf. Allein im letzten Jahrzehnt waren drei Jahre prim: 1993, 1997 und 1999. Wären sie selten, könnte die Kryptographie auch nicht so mit ihnen arbeiten, wie sie es tut.

Primzahlen können nur auf eine einzige („triviale“) Weise zerlegt werden:

$$\begin{aligned} 5 &= 1 * 5 \\ 17 &= 1 * 17 \\ 1.013 &= 1 * 1013 \\ 1.296.409 &= 1 * 1.296.409. \end{aligned}$$

Definition 4.3.2. *Natürliche Zahlen größer* 1*, die keine Primzahlen sind, heißen* **zusammengesetzte Zahlen***: Diese haben mindestens zwei von* 1 *verschiedene Faktoren.*

[9] Joanne K. Rowling, „Harry Potter und der Stein der Weisen“, Carlsen, (c) 1997, Kapitel „Durch die Falltür“, S. 310, Hermine.

[10] Aufgrund der großen Teilerzahl von 12 findet sich diese Zahl – und Vielfache dieser Zahl – oft im Alltag wieder: Die 12 Stunden-Skala der Uhr, die 60 Minuten einer Stunde, die 360 Grad-Skala der Winkelmessung, usw. Teilt man diese Skalen in Bruchteile auf, so ergeben in vielen Fällen die Brüche ganze Zahlen. Mit diesen kann man im Kopf einfacher rechnen als mit gebrochenen Zahlen.

Beispiel Primfaktorzerlegung solcher Zahlen:

$$\begin{aligned} 4 &= 2*2 \\ 6 &= 2*3 \\ 91 &= 7*13 \\ 161 &= 7*23 \\ 767 &= 13*59 \\ 1029 &= 3*7^3 \\ 5324 &= 22*11^3. \end{aligned}$$

Satz 4.3.1. *Jede zusammengesetzte Zahl a besitzt einen kleinsten Teiler größer als 1. Dieser Teiler ist eine Primzahl p und kleiner oder gleich der Quadratwurzel aus a.*

Aus den Primzahlen lassen sich alle ganzen Zahlen größer als 1 zusammensetzen – und das sogar in einer *eindeutigen* Weise.

Dies besagt der 1. *Hauptsatz der Zahlentheorie* (= Hauptsatz der elementaren Zahlentheorie = fundamental theorem of arithmetic = fundamental building block of all positive integers). Er wurde das erste Mal präzise von Carl Friedrich Gauss in seinen Disquisitiones Arithmeticae (1801) formuliert.

Satz 4.3.2. Gauss 1801 *Jede natürliche Zahl a größer als 1 lässt sich als Produkt von Primzahlen schreiben. Sind zwei solche Zerlegungen $a = p_1 * p_2 * \cdots * p_n = q_1 * q_2 * \cdots * q_m$ gegeben, dann gilt nach eventuellem Umsortieren $n = m$ und für alle i: $p_i = q_i$.*

In anderen Worten: Jede natürliche Zahl außer der 1 lässt sich auf genau eine Weise als Produkt von Primzahlen schreiben, wenn man von der Reihenfolge der Faktoren absieht. Die Faktoren sind also eindeutig (die „Expansion in Faktoren“ ist eindeutig)!

Zum Beispiel ist $60 = 2 * 2 * 3 * 5 = 2^2 * 3 * 5$. Und das ist — bis auf eine veränderte Reihenfolge der Faktoren — die einzige Möglichkeit, die Zahl 60 in Primfaktoren zu zerlegen.

Wenn man nicht nur Primzahlen als Faktoren zulässt, gibt es mehrere Möglichkeiten der Zerlegung in Faktoren und die *Eindeutigkeit* (uniqueness) geht verloren:

$$60 = 1 * 60 = 2 * 30 = 4 * 15 = 5 * 12 = 6 * 10 = 2 * 3 * 10 = 2 * 5 * 6 = 3 * 4 * 5 = \cdots$$

Der 1. Hauptsatz ist nur scheinbar selbstverständlich. Man kann viele andere Zahlenmengen[11] konstruieren, bei denen eine multiplikative Zerlegung in die Primfaktoren dieser Mengen *nicht* eindeutig ist.

Für eine mathematische Aussage ist es deshalb nicht nur wichtig, für welche Operation sie definiert wird, sondern auch auf welcher Grundmenge diese Operation definiert wird.

Weitere Details zu den Primzahlen (z.B. wie der „Kleine Satz von Fermat“ zum Testen von sehr großen Zahlen auf ihre Primzahleigenschaft benutzt werden kann) finden sich in diesem Skript in dem Artikel über Primzahlen, Kapitel 3.

[11] Diese Mengen werden speziell aus der Menge der natürlichen Zahlen gebildet. Ein Beispiel findet sich in diesem Skript auf Seite 69 am Ende von Kapitel 3.2.

4.4 Teilbarkeit, Modulus und Restklassen[12]

Werden ganze Zahlen addiert, subtrahiert oder multipliziert, ist das Ergebnis stets wieder eine ganze Zahl. Die Division zweier ganzer Zahlen ergibt nicht immer eine ganze Zahl. Wenn man z.B. 158 durch 10 teilt, ist das Ergebnis die Dezimalzahl $15,8$. Dies ist keine ganze Zahl!

Teilt man 158 dagegen durch 2, ist das Ergebnis 79 eine ganze Zahl. In der Zahlentheorie sagt man, 158 ist *teilbar* durch 2, aber nicht durch 10. Allgemein sagt man:

Definition 4.4.1. *Eine ganze Zahl n ist* **teilbar** *durch eine ganze Zahl d, wenn der Quotient n/d eine ganze Zahl c ist, so dass $n = c * d$.*

Die Zahl n wird *Vielfaches* von d genannt; d wird *Teiler, Divisor* oder *Faktor* von n genannt.

Mathematisch schreibt man das: $d|n$ (gelesen: „d teilt n“). Die Schreibweise $d \not| n$ bedeutet, dass d die Zahl n nicht teilt.

Also gilt in unserem obigen Beispiel: $10 \not| 158$, aber $2|158$.

4.4.1 Die Modulo-Operation – Rechnen mit Kongruenzen

Bei Teilbarkeitsuntersuchungen kommt es nur auf die Reste der Division an: Teilt man eine Zahl n durch m, so benutzt man oft die folgende Schreibweise:

$$\frac{n}{m} = c + \frac{r}{m},$$

wobei c eine ganze Zahl ist und r eine Zahl mit den Werten $0, 1, \cdots, m-1$. Diese Schreibweise heißt Division mit Rest. Dabei heißt c der ganzzahlige „Quotient“ und r der „Rest“ der Division.

Beispiel:

$$\frac{19}{7} = 2 + \frac{5}{7} \quad (m = 7, c = 2, r = 5)$$

Was haben die Zahlen $5, 12, 19, 26, \cdots$ bei der Division durch 7 gemeinsam? Es ergibt sich immer der Rest $r = 5$. Bei der Division durch 7 sind nur die folgenden Reste möglich:

$$r = 0, 1, 2, \cdots, 6$$

Wenn $r = 0$, dann gilt: $m|n$ („m teilt n“).

Wir fassen bei der Division durch 7 die Zahlen, die den gleichen Rest r ergeben, in die „Restklasse r modulo 7“ zusammen. Zwei Zahlen a und b, die zur gleichen Restklasse modulo 7 gehören, bezeichnen wir als „kongruent modulo 7“. Oder ganz allgemein:

Definition 4.4.2. *Als* **Restklasse** *r* **modulo** *m bezeichnet man alle ganzen Zahlen a, die bei der Division durch m denselben Rest r haben.*

[12] Mit dem Lernprogramm **ZT** können Sie das hier und im Folgekapitel vorgestellte Rechnen mit Kongruenzen spielerisch nachvollziehen (siehe ZT-Lern-Kapitel 2.1, Seiten 2-9/40).
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.

Beispiel von Restklassen:

Restklasse 0 modulo $4 = \{x | x = 4 * n;\ n \in \mathbb{Z}\} = \{\ldots, -16, -12, -8, -4, 0, 4, 8, 12, 16, \ldots\}$
Restklasse 3 modulo $4 = \{x | x = 4 * n + 3;\ n \in \mathbb{Z}\} = \{\ldots, -13, -9, -5, -1, 3, 7, 11, 15, \ldots\}$

Da modulo m nur die Reste $0, 1, 2, \cdots, m-1$ möglich sind, rechnet die modulare Arithmetik in endlichen Mengen. Zu jedem Modul m gibt es genau m Restklassen.

Das Ergebnis der Modulo-Operation lässt sich so ausdrücken: $a \bmod m \ = \ a - m * \lfloor a/m \rfloor$

Definition 4.4.3. *Zwei Zahlen $a, b \in \mathbb{N}$ heißen* **restgleich oder kongruent bezüglich** $m \in \mathbb{N}$ *genau dann, wenn beim Teilen durch m der gleiche Rest bleibt.*

Man schreibt: $a \equiv b \pmod m$. Und sagt: *a ist kongruent b modulo m*. Das bedeutet, dass a und b zur gleichen Restklasse gehören. Der Modul ist also der Teiler. Diese Schreibweise wurde von Gauss eingeführt. Gewöhnlich ist der Teiler positiv, aber a und b können auch beliebige ganze Zahlen sein.

Beispiel:

$19 \equiv 12 \pmod 7$, denn die Reste sind gleich: $19/7 = 2$ Rest 5 und $12/7 = 1$ Rest 5.
$23103 \equiv 0 \pmod{453}$, denn $23103/453 = 51$ Rest 0 und $0/453 = 0$ Rest 0.

Satz 4.4.1. *$a \equiv b \pmod m$ gilt genau dann, wenn die Differenz $(a-b)$ durch m teilbar ist, also wenn ein $q \in \mathbf{Z}$ existiert mit $(a-b) = q * m$.*[13]

In anderen Worten: $\quad a \equiv b \bmod m \iff m|(a-b) \iff (a-b) \equiv 0 \bmod m$

Daraus ergibt sich: Wenn m die Differenz teilt, gibt es eine ganze Zahl q, so dass gilt: $a = b + q * m$. Alternativ zur Kongruenzschreibweise kann man auch die Teilbarkeitsschreibweise verwenden: $m|(a-b)$.

Beispiel für äquivalente Aussagen:
$35 \equiv 11 \pmod 3 \iff 35 - 11 \equiv 0 \pmod 3$, wobei $35 - 11 = 24$ sich ohne Rest durch 3 teilen lässt, während $35 : 3$ und $11 : 3$ beide den Rest 2 ergeben.

Anwenden kann man die obige Äquivalenz von Satz 4.4.1, wenn man schnell und geschickt für große Zahlen entscheiden will, ob sie durch eine bestimmte Zahl teilbar sind.

Beispiel:
Ist 69.993 durch 7 teilbar?
Da die Zahl in eine Differenz zerlegt werden kann, bei der einfach zu ersehen ist, dass jeder Operand durch 7 teilbar ist, ist auch die Differenz durch 7 teilbar: $69.993 = 70.000 - 7$.

Diese Überlegungen und Definitionen mögen recht theoretisch erscheinen, sind uns im Alltag aber so vertraut, dass wir die formale Vorgehensweise gar nicht mehr wahrnehmen: Bei der Uhr werden die 24 h eines Tages durch die Zahlen $1, 2, \cdots, 12$ repräsentiert. Die Stunden nach 12:00 mittags erhält man als Reste einer Division durch 12. Wir wissen sofort, dass 2 Uhr nachmittags dasselbe wie 14:00 ist.

[13] Die obige Äquivalenz gilt nur für die Differenz $(a-b)$, nicht für die Summe $(a+b)$!
Beispiel:
$11 \equiv 2 \pmod 3$, also ist $11 - 2 = 9 \equiv 0 \pmod 3$; aber $11 + 2 = 13$ ist nicht durch 3 teilbar.
Für Summen gilt die Aussage von Satz 4.4.1 nicht einmal in eine Richtung. Richtig ist sie bei Summen nur für den Rest 0 und nur in der folgenden Richtung: Teilt ein Teiler beide Summanden ohne Rest, teilt er auch die Summe ohne Rest.

Die „modulare“, also auf die Divisionsreste bezogene Arithmetik ist die Basis der asymmetrischen Verschlüsselungsverfahren. Kryptographische Berechnungen spielen sich also nicht wie das Schulrechnen unter den reellen Zahlen ab, sondern unter Zeichenketten begrenzter Länge, das heißt unter positiven ganzen Zahlen, die einen gewissen Wert nicht überschreiten dürfen. Aus diesem und anderen Gründen wählt man sich eine große Zahl m und „rechnet modulo m“, das heißt, man ignoriert ganzzahlige Vielfache von m und rechnet statt mit einer Zahl nur mit dem Rest bei Division dieser Zahl durch m. Dadurch bleiben alle Ergebnisse im Bereich von 0 bis $m - 1$.

4.5 Rechnen in endlichen Mengen

4.5.1 Gesetze beim modularen Rechnen

Aus Sätzen der Algebra folgt, dass wesentliche Teile der üblichen Rechenregeln beim Übergang zum modularen Rechnen über der Grundmenge $\mathbb{Z}$ erhalten bleiben: Die Addition ist nach wie vor kommutativ. Gleiches gilt für die Multiplikation modulo m. Das Ergebnis einer Division[14] ist kein Bruch, sondern eine ganze Zahl zwischen 0 und $m-1$.

Es gelten die bekannten Gesetze:

1. **Assoziativgesetz:**
   $((a+b)+c) \pmod m \equiv (a+(b+c)) \pmod m$.
   $((a*b)*c) \pmod m \equiv (a*(b*c)) \pmod m$.

2. **Kommutativgesetz:**
   $(a+b) \pmod m \equiv (b+a) \pmod m$.
   $(a*b) \pmod m \equiv (b*a) \pmod m$.

Assoziativgesetz und Kommutativgesetz gelten sowohl für die Addition als auch für die Multiplikation.

3. **Distributivgesetz:**
   $(a*(b+c)) \pmod m \equiv (a*b+a*c) \pmod m$.

4. **Reduzierbarkeit:**
   $(a+b) \pmod m \equiv (a \pmod m + b \pmod m) \pmod m$.
   $(a*b) \pmod m \equiv (a \pmod m * b \pmod m) \pmod m$.
   Beim Addieren und Multiplizieren ist es gleichgültig, in welcher Reihenfolge die Modulo-Operation durchgeführt wird.

5. **Existenz einer Identität (neutrales Element):**
   $(a+0) \pmod m \equiv (0+a) \pmod m \equiv a \pmod m$.
   $(a*1) \pmod m \equiv (1*a) \pmod m \equiv a \pmod m$.

6. **Existenz des inversen Elements**[15]:
   - **Additive Inverse**
     Für jedes ganzzahlige a und m gibt es eine ganze Zahl $-a$, so dass gilt:
     $(a+(-a)) \pmod m \equiv 0 \pmod m$
   - **Multiplikative Inverse modulo einer Primzahl p**
     Für jedes ganzzahlige a (mit $a \not\equiv 0 \pmod p$ und p prim) gibt es eine ganze Zahl a^{-1}, so dass gilt: $(a*a^{-1}) \pmod p \equiv 1 \pmod p$.
   - **Multiplikative Inverse modulo einer zusammengesetzten Zahl m**[16]
     Für alle ganzzahligen a und m (mit $a \not\equiv 0 \pmod m$ und $ggT(a,m)=1$) gibt es eine ganze Zahl a^{-1}, so dass gilt: $(a*a^{-1}) \pmod m \equiv 1 \pmod m$

[14] Die Division modulo m ist nur für Zahlen, die teilerfremd zu m sind, definiert, da andere Zahlen die gleiche Eigenschaft wie Null haben, d.h. zu ihnen gibt es keine Inverse. Vergleiche das Gesetz Nummer 6 **Existenz des inversen Elements**. Vergleiche Fußnote 20 in Kapitel 4.6.1 und Tabelle 4.3 in Kapitel 4.6.2.

[15] Inverse sind nur dann Inverse, wenn sie bezüglich der gegebenen Operation eindeutig sind.

[16] Da $8 \equiv 3 \ mod\ 5$ und $3*2 \equiv 1 \ mod\ 5$, ist $2 = 3^{-1} = 8^{-1}$ eine (eindeutige) Inverse für 3 und 8.
Ein Vielfaches von p oder m hat keine Inverse mod p oder mod m: $5 \equiv 10 \equiv 0 \ mod\ 5$.

7. **Abgeschlossenheit**[17]:
   $a, b \in G \Longrightarrow (a + b) \in G$.
   $a, b \in G \Longrightarrow (a * b) \in G$.

8. **Transitivität:**
   $[a \equiv b \bmod m,\ b \equiv c \bmod m] \Longrightarrow [a \equiv c \bmod m]$.

4.5.2 Muster und Strukturen

Generell untersuchen die Mathematiker „Strukturen“. Sie fragen sich z.B. bei $a * x \equiv b \bmod m$, welche Werte x für gegebene Werte a, b, m annehmen kann.

Insbesondere wird dies untersucht für den Fall, dass das Ergebnis b der Operation das neutrale Element ist. Dann ist x die Inverse von a bezüglich dieser Operation.

[17] Diese Eigenschaft wird innerhalb einer Menge immer bezüglich einer Operation definiert. Siehe Kapitel 4.15 „Anhang: Abschlussbildung“.

Lang ist der Weg durch Lehren, kurz und wirksam durch Beispiele.

Zitat 9: Seneca[18]

4.6 Beispiele für modulares Rechnen

Wir haben bisher gesehen:

Für zwei natürliche Zahlen a und m bezeichnet a mod m den Rest, den man erhält, wenn man a durch m teilt. Daher ist $a \pmod m$ stets eine Zahl zwischen 0 und $m-1$.

Zum Beispiel gilt: $1 \equiv 6 \equiv 41 \pmod 5$, denn der Rest ist jeweils 1. Ein anderes Beispiel ist: $2000 \equiv 0 \pmod 4$, denn 4 geht in 2000 ohne Rest auf.

In der modularen Arithmetik gibt es nur eine eingeschränkte Menge nicht-negativer Zahlen. Deren Anzahl wird durch einen Modul m vorgegeben. Ist der Modul $m = 5$, werden nur die 5 Zahlen der Menge $\{0, 1, 2, 3, 4\}$ benutzt.

Ein Rechenergebnis größer als 4 wird dann „modulo“ 5 umgeformt, d.h. es ist der Rest, der sich bei der Division des Ergebnisses durch 5 ergibt. So ist etwa $2 * 4 \equiv 8 \equiv 3 \pmod 5$, da 3 der Rest ist, wenn man 8 durch 5 teilt.

4.6.1 Addition und Multiplikation

Im folgenden werden zwei Tabellen aufgestellt:

- die Additionstabelle[19] für mod 5 (Tabelle 4.1) und
- die Multiplikationstabellen[20] für mod 5 (Tabelle 4.2) und für mod 6 (Tabelle 4.3).

Beispiel Additionstabelle:
Das Ergebnis der Addition von 3 und 4 (mod 5) wird folgendermaßen bestimmt: Berechne $3 + 4 = 7$ und ziehe solange die 5 vom Ergebnis ab, bis sich ein Ergebnis kleiner als der Modul ergibt: $7 - 5 = 2$. Also ist: $3 + 4 \equiv 2 \pmod 5$.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Tabelle 4.1: Additionstabelle modulo 5

[18] Lucius Annaeus Seneca, philosophischer Schriftsteller und Dichter, 4 v. Chr. – 65 n. Chr.

[19] Bemerkung zur Subtraktion modulo 5:
$2 - 4 = -2 \equiv 3 \bmod 5$.
Es gilt modulo 5 also nicht, dass $-2 = 2$ (siehe Kapitel 4.16 „Anhang: Bemerkungen zur modulo Subtraktion“).

[20] Bemerkung zur modulo Division:
Aufgrund der besonderen Rolle der 0 als Identität bei der Addition, darf nicht durch die Null geteilt werden. Für alle a gilt $a * 0 = 0$, denn $a * 0 = a * (0 + 0) = a * 0 + a * 0$. Es ist offensichtlich, dass 0 keine Inverse bzgl. der Multiplikation besitzt, denn sonst müsste gelten: $0 = 0 * 0^{-1} = 1$. Vergleiche Fußnote 14 in Kapitel 4.5.1.

Beispiel Multiplikationstabelle:
Das Ergebnis der Multiplikation $4{*}4 \pmod 5$ wird folgendermaßen bestimmt: Berechne $4{*}4 = 16$ und ziehe solange die 5 ab, bis sich ein Ergebnis kleiner als der Modul ergibt.

$$16 - 5 = 11;\ 11 - 5 = 6;\ 6 - 5 = 1$$

Direkt ergibt es sich auch aus Tabelle 4.2: $4 * 4 \equiv 1 \pmod 5$, weil $16 : 5 = 3$ Rest 1.
Beachte: Die Multiplikation wird auf der Menge $\mathbb{Z}$ ohne 0 definiert (da $0 * x$ immer 0 ergibt und 0 keine Inverse hat).

*	1	2	3	4
1	1	2	3	4
2	2	**4**	**1**	3
3	3	**1**	**4**	2
4	4	3	2	1

Tabelle 4.2: Multiplikationstabelle modulo 5

4.6.2 Additive und multiplikative Inverse

Aus den Tabellen kann man zu jeder Zahl die Inversen bezüglich der Addition und der Multiplikation ablesen.

Die Inverse einer Zahl ist diejenige Zahl, die bei Addition der beiden Zahlen das Ergebnis 0 und bei der Multiplikation das Ergebnis 1 ergibt. So ist die Inverse von 4 für die Addition mod 5 die 1 und für die Multiplikation mod 5 die 4 selbst, denn

$$\begin{aligned} 4 + 1 &= \ \ 5 \equiv 0 \pmod 5; \\ 4 * 4 &= 16 \equiv 1 \pmod 5. \end{aligned}$$

Die Inverse von 1 bei der Multiplikation mod 5 ist 1; die Inverse modulo 5 von 2 ist 3 und weil die Multiplikation kommutativ ist, ist die Inverse von 3 wiederum die 2.

Wenn man zu einer beliebigen Zahl (hier 2) eine weitere beliebige Zahl (hier 4) addiert bzw. multipliziert und danach zum Zwischenergebnis (1 bzw. 3) die jeweilige Inverse der weiteren Zahl (1 bzw. 4) addiert[21] bzw. multipliziert, ist das Gesamtergebnis gleich dem Ausgangswert.

Beispiel:

$$\begin{aligned} 2 + 4 \equiv 6 \equiv 1 \pmod 5; &\quad 1 + 1 \equiv 2 \equiv 2 \pmod 5 \\ 2 * 4 \equiv 8 \equiv 3 \pmod 5; &\quad 3 * 4 \equiv 12 \equiv 2 \pmod 5 \end{aligned}$$

In der Menge $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$ für die Addition, und in der Menge $\mathbb{Z}_5 \setminus \{0\}$ für die Multiplikation haben alle Zahlen eine **eindeutige** Inverse bezüglich modulo 5.

Bei der modularen Addition ist das für jeden Modul (also nicht nur für 5) so.

Bei der modularen Multiplikation dagegen ist das nicht so (wichtiger Satz):

[21] Allgemein: $x + y + (-y) \equiv x \pmod m$ [$(-y)$ = additive Inverse zu $y \pmod m$]

Satz 4.6.1. *Für eine natürliche Zahl a aus der Menge* $\{1, \cdots, m-1\}$ *gibt es genau dann eine modulare multiplikative Inverse, wenn sie mit dem Modul m teilerfremd*[22] *ist, d.h. wenn a und m keine gemeinsamen Primfaktoren haben.*

Da $m = 5$ eine Primzahl ist, sind die Zahlen 1 bis 4 teilerfremd zu 5, und es gibt mod 5 zu **jeder** dieser Zahlen eine multiplikative Inverse.

Ein Gegenbeispiel zeigt die Multiplikationstabelle für mod 6 (da der Modul $m = 6$ nicht prim ist, sind nicht alle Elemente aus $\mathbb{Z}_6 \setminus \{0\}$ zu 6 teilerfremd):

*	1	2	3	4	5
1	1	2	3	4	5
2	2	**4**	**0**	**2**	4
3	3	**0**	**3**	**0**	3
4	4	**2**	**0**	**4**	2
5	5	4	3	2	1

Tabelle 4.3: Multiplikationstabelle modulo 6

Neben der 0 gibt es auch bei den Zahlen $2, 3$ und 4 keinen passenden Faktor, so dass das Produkt mod 6 die 1 ergibt. Man sagt auch, diese Zahlen haben **keine** Inverse.

Die Zahlen $2, 3$ und 4 haben mit dem Modul 6 den Faktor 2 oder 3 gemeinsam. Nur die zu 6 teilerfremden Zahlen 1 und 5 haben multiplikative Inverse, nämlich jeweils sich selbst.

Die Anzahl der zum Modul m teilerfremden Zahlen ist auch die Anzahl derjenigen Zahlen, die eine multiplikative Inverse haben (vgl. die Euler-Funktion $\phi(m)$ in Kapitel 4.8.2).

Für die beiden in den Multiplikationstabellen verwendeten Moduli 5 und 6 bedeutet dies: Der Modul 5 ist bereits eine Primzahl. Also gibt es in mod 5 genau $\phi(5) = 5 - 1 = 4$ mit dem Modul teilerfremde Zahlen, also alle von 1 bis 4.

Da 6 keine Primzahl ist, zerlegen wir 6 in seine Faktoren: $6 = 2 * 3$. Daher gibt es in mod 6 genau $\phi(6) = (2-1) * (3-1) = 1 * 2 = 2$ Zahlen, die eine multiplikative Inverse haben, nämlich die 1 und die 5.

Für große Moduli scheint es nicht einfach, die Tabelle der multiplikativen Inversen zu berechnen (das gilt nur für die in den oberen Multiplikationstabellen fett markierten Zahlen). Mit Hilfe des kleinen Satzes von Fermat kann man dafür einen einfachen Algorithmus aufstellen [Pfl97, S. 80]. Schnellere Algorithmen werden z.B. in [Knu98] beschrieben.[23]

Kryptographisch ist nicht nur die Eindeutigkeit der Inversen, sondern auch das Ausschöpfen des gesamten Wertebereiches eine wichtige Eigenschaft.

---

[22] Es gilt: Zwei ganze Zahlen a und b sind genau dann teilerfremd, wenn $\text{ggT}(a, b) = 1$.
Desweiteren gilt: Ist p prim und a eine beliebige ganze Zahl, die kein Vielfaches von p ist, so sind beide Zahlen teilerfremd.
Weitere Bezeichnungen zum Thema Teilerfremdheit (mit $a_i \in \mathbb{Z}, i = 1, \cdots, n$):

1. $a_1, a_2, \cdots, a_n$ heißen *relativ prim* , wenn $\text{ggT}(a_1, \cdots, a_n) = 1$.
2. Für mehr als 2 Zahlen ist eine noch stärkere Anforderung:
   $a_1, \cdots, a_n$ heißen *paarweise relativ prim*, wenn für alle $i = 1, \cdots, n$ und $j = 1, \cdots, n$ mit $i \neq j$ gilt: $\text{ggT}(a_i, a_j) = 1$.

Beispiel:
$2, 3, 6$ sind relativ prim, da $\text{ggT}(2, 3, 6) = 1$. Sie sind nicht paarweise prim, da $\text{ggT}(2, 6) = 2 > 1$.

[23] Mit dem erweiterten Satz von Euklid (erweiterter ggT) kann man die multiplikative Inverse berechnen und die Invertierbarkeit bestimmen (siehe Anhang 4.14). Alternativ kann auch die Primitivwurzel genutzt werden.

Satz 4.6.2. *Sei* $a, i \in \{1, \cdots, m-1\}$ *mit* $\text{ggT}(a, m) = 1$, *dann nimmt für eine bestimmte Zahl* a *das Produkt* $a * i \bmod m$ *alle Werte aus* $\{1, \cdots, m-1\}$ *an (erschöpfende Permutation der Länge* $m-1$*).*[24]

Die folgenden drei Beispiele[25] veranschaulichen Eigenschaften der multiplikativen Inversen (hier sind nicht mehr die vollständigen Multiplikationstabellen angegeben, sondern nur die Zeilen für die Faktoren 5 und 6).

Tabelle **4.4** (Multiplikationstabelle mod 17 wurde für $i = 1, 2, \cdots, 18$ berechnet:

$(5 * i)/17 = a$ Rest r und hervorgehoben $5 * i \equiv 1 \pmod{17}$,

$(6 * i)/17 = a$ Rest r und hervorgehoben $6 * i \equiv 1 \pmod{17}$.

Gesucht ist das i, für das der Produktrest $a * i$ modulo 17 mit $a = 5$ bzw. $a = 6$ den Wert 1 hat (also die multiplikative Inverse von $a * i$).

i ⇒	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$5 * i$	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
Rest	5	10	15	3	8	13	**1**	6	11	16	4	9	14	2	7	12	0	5
$6 * i$	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108
Rest	6	12	**1**	7	13	2	8	14	3	9	15	4	10	16	5	11	0	6

Tabelle 4.4: Multiplikationstabelle modulo 17 (für $a = 5$ und $a = 6$)

Da sowohl 5 als auch 6 jeweils teilerfremd zum Modul $m = 17$ sind, kommen zwischen $i = 1, \cdots, m$ für die Reste alle Werte zwischen $0, \cdots, m-1$ vor (vollständige m-Permutation).

Die multiplikative Inverse von 5 (mod 17) ist 7, die Inverse von 6 (mod 17) ist 3.

Tabelle **4.5** (Multiplikationstabelle mod 13 berechnet die Reste der Produkte $5 * i$ und $6 * i$:

i ⇒	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$5 * i$	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
Rest	5	10	2	7	12	4	9	**1**	6	11	3	8	0	5	10	2	7	12
$6 * i$	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108
Rest	6	12	5	11	4	10	3	9	2	8	**1**	7	0	6	12	5	11	4

Tabelle 4.5: Multiplikationstabelle modulo 13 (für $a = 5$ und $a = 6$)

Da sowohl 5 als auch 6 auch zum Modul $m = 13$ jeweils teilerfremd sind, kommen zwischen $i = 1, \cdots, m$ für die Reste alle Werte zwischen $0, \cdots, m-1$ vor.

Die multiplikative Inverse von 5 (mod 13) ist 8, die Inverse von 6 (mod 13) ist 11.

Tabelle **4.6** enthält ein Beispiel dafür, wo der Modul m und die Zahl ($a = 6$) *nicht* teilerfremd sind.

[24] Vergleiche auch Satz 4.9.1 in Kapitel 4.9, Multiplikative Ordnung und Primitivwurzel.

[25] In Kapitel 4.19.1 „Multiplikationstabellen modulo m“ finden Sie den Quellcode zur Berechnung der Tabellen mit SageMath.

i ⇒	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
5*i	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
Rest	5	10	3	8	**1**	6	11	4	9	2	7	0	5	10	3	8	1	6
6*i	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108
Rest	6	0	6	0	6	0	6	0	6	0	6	0	6	0	6	0	6	0

Tabelle 4.6: Multiplikationstabelle modulo 12 (für $a = 5$ und $a = 6$)

Berechnet wurde $(5 * i) \pmod{12}$ und $(6 * i) \pmod{12}$. Da 6 und der Modul $m = 12$ nicht teilerfremd sind, kommen zwischen $i = 1, \cdots, m$ nicht alle Werte zwischen $0, \cdots, m-1$ vor und 6 hat mod 12 auch keine Inverse!

Die multiplikative Inverse von 5 (mod 12) ist 5. Die Zahl 6 hat keine Inverse (mod 12).

4.6.3 Potenzieren

Das Potenzieren ist in der modularen Arithmetik definiert als wiederholtes Multiplizieren – wie üblich. Es gelten mit kleinen Einschränkungen die üblichen Rechenregeln wie

$$\begin{aligned} a^{b+c} &= a^b * a^c, \\ (a^b)^c &= a^{b*c} = a^{c*b} = (a^c)^b \end{aligned}$$

Analog der modularen Addition und der modularen Multiplikation funktioniert das modulare Potenzieren:

$$3^2 = 9 \equiv 4 \pmod{5}.$$

Auch aufeinanderfolgendes Potenzieren geht analog:

Beispiel 1:

$$(4^3)^2 = 64^2 \equiv 4096 \equiv 1 \pmod{5}.$$

(1) Reduziert man bereits **Zwischenergebnisse** modulo 5, kann man schneller[26] rechnen, muss aber auch aufpassen, da sich dann *nicht* immer alles wie in der gewöhnlichen Arithmetik verhält.

$$\begin{aligned} (4^3)^2 &\equiv (4^3 \pmod 5)^2 \pmod 5 \\ &\equiv (64 \pmod 5)^2 \pmod 5 \\ &\equiv 4^2 \pmod 5 \\ &\equiv 16 \equiv 1 \pmod 5. \end{aligned}$$

(2) Aufeinanderfolgende Potenzierungen lassen sich in der gewöhnlichen Arithmetik auf eine einfache Potenzierung zurückführen, indem man die Exponenten miteinander multipliziert:

$$(4^3)^2 = 4^{3*2} = 4^6 = 4096.$$

[26] Die Rechenzeit der Multiplikation zweier Zahlen hängt normalerweise von der Länge der Zahlen ab. Dies sieht man, wenn man nach der Schulmethode z.B. $474 * 228$ berechnet: Der Aufwand steigt quadratisch, da $3 * 3$ Ziffern multipliziert werden müssen. Durch die Reduktion der Zwischenergebnisse werden die Zahlen deutlich kleiner.

In der modularen Arithmetik geht das nicht ganz so einfach, denn man erhielte:

$$(4^3)^2 \equiv 4^{3*2 \pmod 5} \equiv 4^{6 \pmod 5} \equiv 4^1 \equiv 4 \pmod 5.$$

Wie wir oben sahen, ist das richtige Ergebnis aber 1 !

(3) Deshalb ist für das fortgesetzte Potenzieren in der modularen Arithmetik die Regel etwas anders: Man multipliziert die Exponenten nicht in (mod m), sondern in (mod $\phi(m)$).

Mit $\phi(5) = 4$ ergibt sich:

$$(4^3)^2 \equiv 4^{3*2 \pmod{\phi(5)}} \equiv 4^{6 \bmod 4} \equiv 4^2 \equiv 16 \equiv 1 \pmod 5.$$

Das liefert das richtige Ergebnis.

Satz 4.6.3. $(a^b)^c \equiv a^{b*c \pmod{\phi(m)}} \pmod m$.

Beispiel 2:

$$3^{28} = 3^{4*7} \equiv 3^{4*7 \pmod{10}} \equiv 3^8 \equiv 6561 \equiv 5 \pmod{11}.$$

4.6.4 Schnelles Berechnen hoher Potenzen

Bei RSA-Ver- und Entschlüsselungen[27] müssen hohe Potenzen modulo m berechnet werden. Schon die Berechnung (100^5) mod 3 sprengt den 32 Bit langen Long-Integer-Zahlenbereich, sofern man zur Berechnung von a^n getreu der Definition a tatsächlich n-mal mit sich selbst multipliziert. Bei sehr großen Zahlen wäre selbst ein schneller Computerchip mit einer einzigen Exponentiation länger beschäftigt als das Weltall existiert. Glücklicherweise gibt es für die Exponentiation (nicht aber für das Logarithmieren) eine sehr wirksame Abkürzung.

Wird der Ausdruck anhand der Rechenregeln der modularen Arithmetik anders aufgeteilt, sprengt die Berechnung nicht einmal den 16 Bit langen Short-Integer-Bereich:

$$(a^5) \equiv (((a^2 \pmod m)^2 \pmod m) * a) \pmod m.$$

Dies kann man verallgemeinern, indem man den Exponenten binär darstellt. Beispielsweise würde die Berechnung von a^n für $n = 37$ auf dem naiven Wege 36 Multiplikationen erfordern. Schreibt man jedoch n in der Binärdarstellung als $100101 = 1*2^5 + 1*2^2 + 1*2^0$, so kann man umformen: $a^{37} = a^{2^5+2^2+2^0} = a^{2^5} * a^{2^2} * a^1$.

Beispiel 3: $87^{43} \pmod{103}$.

Da $43 = 32 + 8 + 2 + 1$, 103 prim , $43 < \phi(103)$ ist und

die Quadrate (mod 103) vorab berechnet werden können

$$\begin{aligned} 87^2 &\equiv 50 \pmod{103}, \\ 87^4 &\equiv 50^2 \equiv 28 \pmod{103}, \\ 87^8 &\equiv 28^2 \equiv 63 \pmod{103}, \\ 87^{16} &\equiv 63^2 \equiv 55 \pmod{103}, \\ 87^{32} &\equiv 55^2 \equiv 38 \pmod{103}, \end{aligned}$$

[27] Siehe Kapitel 4.10 („Beweis des RSA-Verfahrens mit Euler-Fermat“) und Kapitel 4.13 („Das RSA-Verfahren mit konkreten Zahlen“).

gilt[28]:

$$\begin{aligned} 87^{43} &\equiv 87^{32+8+2+1} \pmod{103} \\ &\equiv 87^{32} * 87^{8} * 87^{2} * 87 \pmod{103} \\ &\equiv 38 * 63 * 50 * 87 \equiv 85 \pmod{103}. \end{aligned}$$

Die Potenzen $(a^2)^k$ sind durch fortgesetztes Quadrieren leicht zu bestimmen. Solange sich a nicht ändert, kann ein Computer sie vorab berechnen und – bei ausreichend Speicherplatz – abspeichern. Um dann im Einzelfall a^n zu finden, muss er nur noch genau diejenigen $(a^2)^k$ miteinander multiplizieren, für die an der k-ten Stelle der Binärdarstellung von n eine Eins steht. Der typische Aufwand für $n = 600$ sinkt dadurch von 2^{600} auf $2 * 600$ Multiplikationen! Dieser häufig verwendete Algorithmus heißt „Square and Multiply“.

4.6.5 Wurzeln und Logarithmen

Die Umkehrungen des Potenzierens modulo m sind ebenfalls definiert: Wurzeln und Logarithmen sind abermals ganze Zahlen, aber im Gegensatz zur üblichen Situation sind sie nicht nur mühsam, sondern bei sehr großen Zahlen in „erträglicher“ Zeit überhaupt nicht zu berechnen.

Gegeben sei die Gleichung: $a \equiv b^c \pmod{m}$.

a) Logarithmieren (Bestimmen von c) — Diskretes Logarithmusproblem[29]:

Wenn man von den drei Zahlen a, b, c, welche diese Gleichung erfüllen, a und b kennt, ist jede bekannte Methode, c zu finden, ungefähr so aufwendig wie das Durchprobieren aller m denkbaren Werte für c – bei einem typischen m in der Größenordnung von 10^{180} für 600-stellige Binärzahlen ein hoffnungsloses Unterfangen. Genauer ist für geeignet große Zahlen m der Aufwand nach heutigem Wissensstand proportional zu

$$\exp\left(C * (\log m [\log\log m]^2)^{1/3}\right)$$

mit einer Konstanten $C > 1$.

b) Wurzel-Berechnung (Bestimmen von b):

Ähnliches gilt, wenn b die Unbekannte ist und die Zahlen a und c bekannt sind.
Wenn die Euler-Funktion[30] $\phi(m)$ bekannt ist, kann man leicht[31] d finden mittels $c * d \equiv 1 \pmod{\phi(m)}$ und erhält mit Satz 4.6.3

$$a^d \equiv (b^c)^d \equiv b^{c*d} \equiv b^{c*d \ (mod\ \phi(m))} \equiv b^1 \equiv b \pmod{m}$$

die *c-te Wurzel* b von a.

Für den Fall, dass $\phi(m)$ in der Praxis nicht bestimmt werden kann[32], ist die Berechnung der c-ten Wurzel schwierig. Hierauf beruhen die Sicherheitsannahmen für das RSA-Kryptosystem (siehe Kapitel 4.10 oder Kapitel 5.3.1).

[28] In Kapitel 4.19.2 „Schnelles Berechnen hoher Potenzen“ finden Sie den Beispielcode zum Nachrechnen der Square-and-Multiply-Methode mit SageMath.
[29] Weitere Details zum Diskreten Logarithmusproblem finden sie in Kapitel 5.4.
[30] Siehe Kapitel 4.8.2, „Die Eulersche Phi-Funktion“.
[31] Siehe Kapitel 4.14, „Anhang: Der ggT und die beiden Algorithmen von Euklid“.
[32] Nach dem ersten Hauptsatz der Zahlentheorie und Satz 4.8.4 kann man $\phi(m)$ mit Hilfe der Primfaktorzerlegung von m bestimmen.

Dagegen ist der Aufwand für die Umkehrung von Addition und Multiplikation nur proportional zu $\log m$ beziehungsweise $(\log m)^2$. Potenzieren (zu einer Zahl x berechne x^a mit festem a) und Exponentiation (zu einer Zahl x berechne a^x mit festem a) sind also typische Einwegfunktionen (vergleiche Kapitel 5.1 und 4.12.1).

4.7 Gruppen und modulare Arithmetik über $\mathbb{Z}_n$ und $\mathbb{Z}_n^*$

In der Zahlentheorie und in der Kryptographie spielen mathematische „*Gruppen*“ eine entscheidende Rolle. Von Gruppen spricht man nur, wenn für eine definierte Menge und eine definierte Relation (eine Operation wie Addition oder Multiplikation) die folgenden Eigenschaften erfüllt sind:

- Abgeschlossenheit
- Existenz des neutralen Elements
- Existenz eines inversen Elements zu jedem Element und
- Gültigkeit des Assoziativgesetzes.

Die abgekürzte mathematische Schreibweise lautet: $(G, +)$ oder $(G, *)$.

Definition 4.7.1. $\mathbb{Z}_n$ *:*

$$\mathbb{Z}_n \textit{ umfasst alle ganzen Zahlen von } 0 \textit{ bis } n-1 : \mathbb{Z}_n = \{0, 1, 2, \cdots, n-2, n-1\}.$$

$\mathbb{Z}_n$ ist eine häufig verwendete endliche Gruppe aus den natürlichen Zahlen. Sie wird manchmal auch als Restmenge R modulo n bezeichnet.

Beispielsweise rechnen 32 Bit-Computer (übliche PCs) mit ganzen Zahlen direkt nur in einer endlichen Menge, nämlich in dem Wertebereich $0, 1, 2, \cdots, 2^{32} - 1$.

Dieser Zahlenbereich ist äquivalent zur Menge $\mathbb{Z}_{2^{32}}$.

4.7.1 Addition in einer Gruppe

Definiert man auf einer solchen Menge die Operation mod+ mit

$$a \text{ mod+ } b := (a + b) \pmod{n},$$

so ist die Menge $\mathbb{Z}_n$ zusammen mit der Relation mod+ eine Gruppe, denn es gelten die folgenden Eigenschaften einer Gruppe für alle Elemente von $\mathbb{Z}_n$:

- a mod+ b ist ein Element von $\mathbb{Z}_n$ (Abgeschlossenheit),
- $(a \text{ mod+ } b) \text{ mod+ } c \equiv a \text{ mod+ } (b \text{ mod+ } c)$ (mod+ ist assoziativ),
- das neutrale Element ist die 0.
- jedes Element $a \in \mathbb{Z}_n$ besitzt bezüglich dieser Operation ein Inverses, nämlich $n - a$ (denn es gilt: $a \text{ mod+ } (n - a) \equiv a + (n - a) \pmod{n} \equiv n \equiv 0 \pmod{n}$).

Da die Operation kommutativ ist, d.h. es gilt $(a \text{ mod+ } b) = (b \text{ mod+ } a)$, ist diese Struktur sogar eine „kommutative Gruppe“.

4.7.2 Multiplikation in einer Gruppe

Definiert man in der Menge $\mathbb{Z}_n$ die Operation mod* mit

$$a \bmod* b := (a * b) \pmod{n},$$

so ist $\mathbb{Z}_n$ zusammen mit dieser Operation **normalerweise keine** Gruppe, weil nicht für jedes n alle Eigenschaften erfüllt sind.

Beispiel:

a) In $\mathbb{Z}_{15}$ besitzt z.B. das Element 5 kein Inverses. Es gibt nämlich kein a mit $5 * a \equiv 1 \pmod{15}$. Jedes Modulo-Produkt mit 5 ergibt auf dieser Menge $5, 10$ oder 0.

b) In $\mathbb{Z}_{55} \setminus \{0\}$ besitzen z.B. die Elemente 5 und 11 keine multiplikativen Inversen. Es gibt nämlich kein a aus $\mathbb{Z}_{55}$ mit $5 * a \equiv 1 \pmod{55}$ und kein a mit $11 * a \equiv 1 \pmod{55}$. Das liegt daran, dass 5 und 11 nicht teilerfremd zu 55 sind. Jedes Modulo-Produkt mit 5 ergibt auf dieser Menge $5, 10, 15, \cdots, 50$ oder 0. Jedes Modulo-Produkt mit 11 ergibt auf dieser Menge $11, 22, 33, 44$ oder 0.

Dagegen gibt es Teilmengen von $\mathbb{Z}_n$, die bezüglich mod* eine Gruppe bilden. Wählt man sämtliche Elemente aus $\mathbb{Z}_n$ aus, die teilerfremd zu n sind, so ist diese Menge eine Gruppe bezüglich mod*. Diese Menge bezeichnet man mit $\mathbb{Z}_n^*$.

Definition 4.7.2. *$\mathbb{Z}_n^*$:*

$$\mathbb{Z}_n^* := \{a \in \mathbb{Z}_n \big| \mathrm{ggT}(a, n) = 1\}.$$

$\mathbb{Z}_n^*$ wird manchmal auch als *reduzierte Restmenge* R' modulo n bezeichnet.

Beispiel: Für $n = 10 = 2 * 5$ gilt:

vollständige Restmenge $R = \mathbb{Z}_n = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

reduzierte Restmenge $R' = \mathbb{Z}_n^* = \{1, 3, 7, 9\} \longrightarrow \phi(n) = 4$.

Bemerkung:
R' bzw. $\mathbb{Z}_n^*$ ist immer eine echte Teilmenge von R bzw. $\mathbb{Z}_n$, da 0 immer Element von $\mathbb{Z}_n$, aber nie Element von $\mathbb{Z}_n^*$ ist. Da 1 (per Definition) und $n - 1$ immer teilerfremd zu n sind, sind sie stets Elemente beider Mengen.

Wählt man irgendein Element aus $\mathbb{Z}_n^*$ und multipliziert es mit jedem anderen Element von $\mathbb{Z}_n^*$, so sind die Produkte[33] alle wieder in $\mathbb{Z}_n^*$ und außerdem sind die Ergebnisse eine eindeutige Permutation der Elemente von $\mathbb{Z}_n^*$. Da die 1 immer Element von $\mathbb{Z}_n^*$ ist, gibt es in dieser Menge eindeutig einen „Partner“, so dass das Produkt 1 ergibt. Mit anderen Worten:

Satz 4.7.1. *Jedes Element in $\mathbb{Z}_n^*$ hat eine multiplikative Inverse.*

Beispiel:
$a = 3$ modulo 10 mit $\mathbb{Z}_n^* = \{1, 3, 7, 9\}$ gilt $a^{-1} = 7$:

$$\begin{aligned} 3 &\equiv 3 * 1 \pmod{10}, \\ 9 &\equiv 3 * 3 \pmod{10}, \\ 1 &\equiv 3 * 7 \pmod{10}, \\ 7 &\equiv 3 * 9 \pmod{10}. \end{aligned}$$

[33] Dies ergibt sich aus der Abgeschlossenheit von $\mathbb{Z}_n^*$ bezüglich der Multiplikation und der ggT-Eigenschaft:
$[a, b \in \mathbb{Z}_n^*] \Rightarrow [((a * b) \pmod{n}) \in \mathbb{Z}_n^*]$, genauer:
$[a, b \in \mathbb{Z}_n^*] \Rightarrow [\mathrm{ggT}(a, n) = 1, \mathrm{ggT}(b, n) = 1] \Rightarrow [\mathrm{ggT}(a * b, n) = 1] \Rightarrow [((a * b) \pmod{n}) \in \mathbb{Z}_n^*]$.

Die eindeutige Invertierung (Umkehrbarkeit) ist eine notwendige Bedingung für die Kryptographie (siehe Kapitel 4.10: Beweis des RSA-Verfahrens mit Euler-Fermat).

Die mathematische Spielanalyse postuliert Spieler, die rational reagieren. Die transaktionale Spielanalyse dagegen befasst sich mit Spielen, die unrational, ja sogar **irrational und damit wirklichkeitsnäher** sind.

Zitat 10: Eric Berne[34]

4.8 Euler-Funktion, kleiner Satz von Fermat und Satz von Euler-Fermat

4.8.1 Muster und Strukturen

So wie Mathematiker die Struktur $a*x \equiv b \bmod m$ untersuchen (s. Kapitel 4.5.2), so interessiert sie auch die Struktur $x^a \equiv b \bmod m$.

Auch hierbei sind insbesondere die Fälle interessant, wenn $b = 1$ ist (also wenn b den Wert der multiplikativen Einheit annimmt) und wenn $b = x$ ist (also die Funktion $f(x) = x^a \bmod m$ einen Fixpunkt hat). Zu den RSA-Fixpunkten: siehe 4.19.7.

4.8.2 Die Eulersche Phi-Funktion

Bei vorgegebenem n ist die Anzahl der Zahlen aus der Menge $\{1, \cdots, n-1\}$, die zu n teilerfremd sind, gleich dem Wert der Euler[35]-Funktion $\phi(n)$.[36]

Definition 4.8.1. *Die Eulersche Phi-Funktion*[37] $\phi(n)$ *gibt die Anzahl der Elemente von* $\mathbb{Z}_n^*$ *an.*

$\phi(n)$ gibt auch an, wieviele ganze Zahlen in mod n multiplikative Inverse haben. $\phi(n)$ lässt sich ganz leicht berechnen, wenn man die Primfaktorzerlegung von n kennt.

Satz 4.8.1. *Für eine Primzahl gilt:* $\phi(p) = p - 1$.

Satz 4.8.2. *Ist* n *das Produkt zweier verschiedenen Primzahlen* p *und* q*, so gilt:*

$$\phi(p * q) = (p - 1) * (q - 1) \quad \text{oder} \quad \phi(p * q) = \phi(p) * \phi(q).$$

Dieser Fall ist für das RSA-Verfahren wichtig.

Satz 4.8.3. *Ist* $n = p_1 * p_2 * \cdots * p_k$*, wobei* p_1 *bis* p_k *verschiedene Primzahlen sind (d.h.* $p_i \neq p_j$ *für* $i \neq j$*), dann gilt (als Verallgemeinerung von Satz 4.8.2):*

$$\phi(n) = (p_1 - 1) * (p_2 - 1) * \cdots * (p_k - 1).$$

Satz 4.8.4. *Verallgemeinert gilt für jede Primzahl* p *und jedes* n *aus* $\mathbb{N}$*:*

1. $\phi(p^n) = p^{n-1} * (p - 1)$.
2. *Ist* $n = p_1^{e_1} * p_2^{e_2} * \cdots * p_k^{e_k}$*, wobei* p_1 *bis* p_k *verschiedene Primzahlen sind, dann gilt:*

$$\phi(n) = [(p_1^{e_1-1}) * (p_1 - 1)] * \cdots * [(p_k^{e_k-1}) * (p_k - 1)] = n * ([(p_1 - 1)/p_1] * \cdots * [(p_k - 1)/p_k]).$$

[34] Eric Berne, „Spiele der Erwachsenen“, rororo, (c) 1964, S. 235.
[35] Leonhard Euler, schweizer Mathematiker, 15.4.1707 – 18.9.1783
[36] Vergleiche auch die Erläuterungen zu der Eulerschen Phi-Funktion $\phi(n)$ in Kapitel 5.3.1 „Das RSA-Verfahren“.
[37] Die Eulersche Phi-Funktion wird auch $\Phi(n)$ oder $phi(n)$ geschrieben.

Beispiel:

- $n = 70 = 2 * 5 * 7 \Longrightarrow$ nach Satz 4.8.3: $\phi(n) = 1 \cdot 4 \cdot 6 = 24$.
- $n = 9 = 3^2 \Longrightarrow$ nach Satz 4.8.4: $\phi(n) = 3^1 \cdot 2 = 6$, weil $\mathbb{Z}_9^* = \{1, 2, 4, 5, 7, 8\}$.
- $n = 2.701.125 = 3^2 * 5^3 * 7^4 \Longrightarrow$ nach Satz 4.8.4: $\phi(n) = [3^1 * 2] * [5^2 * 4] * [7^3 * 6] = 1.234.800$.

Bemerkung: Zahlentheoretische Funktionen in CT2

Die Eulersche Phi-Funktion ist nur eine von mehreren verbreiteten zahlentheoretischen Funktionen bzw. Statistiken. Eine Übersicht und einen schnellen Vergleich für verschiedene Zahlen findet sich in CT2. In der folgenden Abbildung ist beispielsweise die Eulersche Phi-Funktion für die Zahl 24 hervorgehoben.

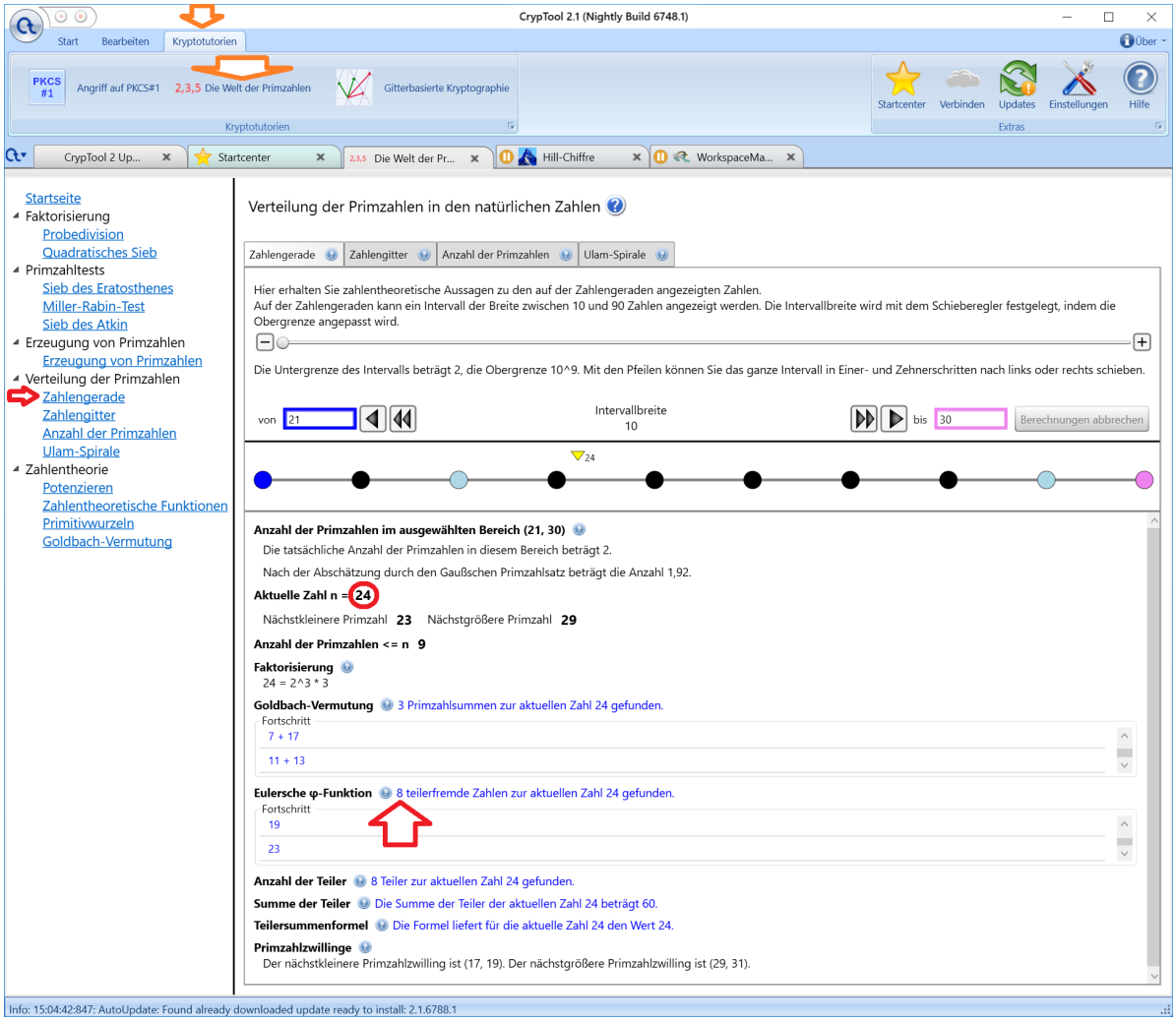


Abbildung 4.1: Zahlentheoretische Funktionen in CT2[38]

[38] Grafik von CT2, Menü Kryptotutorien, Die Welt der Primzahlen, Verteilung der Primzahlen, Zahlengerade.

4.8.3 Der Satz von Euler-Fermat

Für den Beweis des RSA-Verfahrens brauchen wir den Satz von Fermat und dessen Verallgemeinerung (Satz von Euler-Fermat) – vergleiche Kapitel 3.5.

Satz 4.8.5. Kleiner Satz von Fermat[39] *Sei p eine Primzahl und a eine beliebige ganze Zahl, dann gilt:*

$$a^p \equiv a \pmod{p}.$$

Eine alternative Formulierung des kleinen Satzes von Fermat lautet: Sei p eine Primzahl und a eine beliebige ganze Zahl, die teilerfremd zu p ist, dann gilt:

$$a^{p-1} \equiv 1 \pmod{p}.$$

Satz 4.8.6. Satz von Euler-Fermat (Verallgemeinerung des kleines Satzes von Fermat) *Für alle Elemente a aus der Gruppe $\mathbb{Z}_n^*$ gilt (d.h. a und n sind natürliche Zahlen, die teilerfremd zueinander sind):*

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

Dieser Satz besagt, dass wenn man ein Gruppenelement (hier a) mit der Ordnung der Gruppe (hier $\phi(n)$) potenziert, ergibt sich immer das neutrale Element der Multiplikation (die Zahl 1).

Die 2. Formulierung des kleinen Satzes von Fermat ergibt sich direkt aus dem Satz von Euler, wenn n eine Primzahl ist.

Falls n das Produkt zweier verschiedenen Primzahlen ist, kann man mit dem Satz von Euler in bestimmten Fällen sehr schnell das Ergebnis einer modularen Potenz berechnen. Es gilt: $a^{(p-1)*(q-1)} \equiv 1 \pmod{pq}$.

Beispiel Berechnung einer modularen Potenz:

- Was ist $5^2 \pmod 6$?
  Mit $2 = 1 * 2$ und $6 = 2 * 3$, wobei 2 und 3 jeweils prim; $\phi(6) = 2$, da nur 1 und 5 zu 6 teilerfremd sind, folgt $5^2 \equiv 5^{\phi(6)} \equiv 1 \pmod 6$, ohne dass man die Potenz berechnen musste.
- Was ist $31^{792} \pmod{851}$?
  Mit $792 = 22 * 36$ und $23 * 37 = 851$, wobei 23 und 37 jeweils prim sind, folgt für $31 \in \mathbb{Z}_{851}^*$ $31^{792} \equiv 31^{\phi(23*37)} \equiv 31^{\phi(851)} \equiv 1 \pmod{851}$.

4.8.4 Bestimmung der multiplikativen Inversen

Eine weitere interessante Anwendung ist ein Sonderfall der Bestimmung der multiplikativen Inverse mit Hilfe des Satzes von Euler-Fermat (multiplikative Inverse werden ansonsten mit dem erweiterten Euklid'schen Algorithmus ermittelt).

Beispiel:
Finde die multiplikative Inverse von 1579 modulo 7351.

Nach Euler-Fermat gilt: $a^{\phi(n)} = 1 \bmod n$ für alle a aus $\mathbb{Z}_n^*$. Teilt man beide Seiten durch a, ergibt sich: $a^{\phi(n)-1} \equiv a^{-1} \pmod n$. Für den Spezialfall, dass der Modul prim ist, gilt $\phi(n) = p-1$. Also gilt für die modulare Inverse a^{-1} von a:

$$a^{-1} \equiv a^{\phi(n)-1} \equiv a^{(p-1)-1} \equiv a^{p-2} \pmod{p}.$$

[39] Pierre de Fermat, französischer Mathematiker, 17.8.1601 – 12.1.1665.

Für unser Beispiel bedeutet das:

Da der Modul 7351 prim ist, ist $p - 2 = 7349$.
$1579^{-1} \equiv 1579^{7349} \pmod{p}$.

Durch geschicktes Zerlegen des Exponenten kann man diese Potenz relativ einfach berechnen[40]:

$7349 = 4096 + 2048 + 1024 + 128 + 32 + 16 + 4 + 1$

$1579^{-1} \equiv 4716 \pmod{7351}$

4.8.5 Wie viele private RSA-Schlüssel d gibt es modulo 26

Laut Satz 4.6.3 werden die arithmetischen Operationen von modularen Ausdrücken in den Exponenten modulo $\phi(n)$ und nicht modulo n durchgeführt.[41]

Wenn man in $a^{e*d} \equiv a^1 \pmod{n}$ die Inverse z.B. für den Faktor e im Exponenten bestimmen will, muss man modulo $\phi(n)$ rechnen.

Beispiel: (mit Bezug zum RSA-Algorithmus)
Wenn man modulo 26 rechnet, aus welcher Menge können e und d kommen?

Lösung: Es gilt $e * d \equiv 1 \pmod{\phi(26)}$.

Die reduzierte Restmenge $R' = \mathbb{Z}_{26}^* = \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}$ sind die Elemente in $\mathbb{Z}_{26}$, die eine multiplikative Inverse haben, also teilerfremd zu 26 sind (vgl. 4.7.2).

Die reduzierte Restmenge R'' enthält nur die Elemente aus R', die teilerfremd zu $\phi(26) = 12$ sind: $R'' = \{1, 5, 7, 11\}$.

Für jedes e aus R'' gibt es ein d aus R'', so dass $a \equiv (a^e)^d \pmod{n}$.

Somit gibt es also zu jedem e in R'' genau ein (nicht unbedingt von e verschiedenes) Element, so dass gilt: $e * d \equiv 1 \pmod{\phi(26)}$.

Der allgemeine Fall, wo n beliebige natürliche Zahlen annehmen kann (in dem Beispiel hier galt ja immer $n = 26$), wird in Kapitel 4.19.6 behandelt. Dort ist auch ein SageMath-Programm, das die Anzahl aller d liefert. Für alle e, die teilerfremd zu $\phi(n)$ sind, kann man das d nach dem Satz von Euler-Fermat folgendermaßen berechnen:

$$\begin{aligned} d &\equiv e^{-1} \pmod{\phi(n)} \\ &\equiv e^{\phi(\phi(n))-1} \pmod{\phi(n)}, \quad \text{denn } a^{\phi(n)} \equiv 1 \pmod{n} \quad \text{entspricht } a^{\phi(n)-1} \equiv a^{-1} \pmod{n}. \end{aligned}$$

Besteht die Zahl n aus zwei verschiedenen Primfaktoren (und sind noch ein paar weitere Kriterien erfüllt), so sind die Faktorisierung von n und das Finden von $\phi(n)$ ähnlich schwierig[42] (vergleiche Forderung 3 in Kapitel 4.10.1).

---

[40] Siehe Kapitel 4.6.4, „Schnelles Berechnen hoher Potenzen“.
[41] Für das folgende Beispiel wird der Modul wie beim RSA-Verfahren üblich mit „n“ statt mit „m“ bezeichnet.
[42] Für $n = pq$ mit $p \neq q$ gilt $\phi(n) = (p-1) * (q-1) = n - (p+q) + 1$. Ferner sind die Zahlen p und q Lösungen der quadratischen Gleichung $x^2 - (p+q)x + pq = 0$.
Sind nur n und $\phi(n)$ bekannt, so gilt $pq = n$ und $p + q = n + 1 - \phi(n)$. Man erhält somit die Faktoren p und q von n, indem man die folgende quadratische Gleichung löst:

$$x^2 + (\phi(n) - n - 1)x + n = 0$$

4.9 Multiplikative Ordnung und Primitivwurzel[43]

Die Multiplikative Ordnung (order) und die Primitivwurzel (primitive root) sind zwei nützliche Konstrukte (Konzepte) der elementaren Zahlentheorie.

Mathematiker stellen sich die Frage, unter welchen Bedingungen ergibt die wiederholte Anwendung einer Operation das neutrale Element (vergleiche Muster und Strukturen, Kapitel 4.8.1).

Für die i-fach aufeinander folgende modulare Multiplikation einer Zahl a für $i = 1, \ldots, m-1$ ergibt sich als Produkt das neutrale Element der Multiplikation nur dann, wenn a und m teilerfremd sind.

Definition 4.9.1. *Die* **multiplikative Ordnung** $ord_m(a)$ *einer ganzen Zahl* $a \pmod m$ *(wobei* a *und* m *teilerfremd sind) ist die kleinste ganze Zahl* i*, für die gilt:* $a^i \equiv 1 \pmod m$.

Die folgende Tabelle 4.7 zeigt, dass in einer multiplikativen Gruppe (hier $\mathbb{Z}_{11}^*$) nicht notwendig alle Zahlen die gleiche Ordnung haben: Die Ordnungen sind $1, 2, 5$ und 10. Es gilt:

1. Die Ordnungen sind alle Teiler von 10.
2. Die Zahlen $a = 2, 6, 7$ und 8 haben die Ordnung 10. Man sagt diese Zahlen haben in $\mathbb{Z}_{11}^*$ **maximale Ordnung**.

[43] Mit dem Lernprogramm **ZT** können Sie einige der hier besprochenen Verfahren vertiefen (siehe Lern-Kapitel 2.2, Seiten 10-14/40 und 24-40/40).
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.
Siehe auch die SageMath-Beispiele in 4.19.3.

Beispiel 1:
Die folgende Tabelle 4.7[44] zeigt die Werte a^i mod 11 für die Exponenten $i = 1, 2, \cdots, 10$ und für die Basen $a = 1, 2, \cdots, 10$, sowie den sich für jedes a daraus ergebenden Wert $ord_{11}(a)$.

	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	$ord_{11}(a)$
a=1	**1**	1	1	1	1	1	1	1	1	1	1
a=2	2	4	8	5	10	9	7	3	6	**1**	10
a=3	3	9	5	4	**1**	3	9	5	4	1	5
a=4	4	5	9	3	**1**	4	5	9	3	1	5
a=5	5	3	4	9	**1**	5	3	4	9	1	5
a=6	6	3	7	9	10	5	8	4	2	**1**	10
a=7	7	5	2	3	10	4	6	9	8	**1**	10
a=8	8	9	6	4	10	3	2	5	7	**1**	10
a=9	9	4	3	5	**1**	9	4	3	5	1	5
a=10	10	**1**	10	1	10	1	10	1	10	1	2

Tabelle 4.7: Werte von a^i mod $11, 1 \leq a, i < 11$ und zugehörige Ordnung von a modulo 11

Aus der Tabelle 4.7 kann man ersehen, dass z.B. die Ordnung von 3 modulo 11 den Wert 5 hat.

Definition 4.9.2. *Sind a und m teilerfremd und gilt $ord_m(a) = \phi(m)$, (d.h. a hat maximale Ordnung), dann nennt man a eine* **Primitivwurzel** *von m.*[45]

Nicht zu jedem Modul m gibt es eine Zahl a, die eine Primitivwurzel ist. In der Tabelle 4.7 sind nur $a = 2, 6, 7$ und 8 bezüglich mod 11 eine Primitivwurzel ($ord_m(a) = \phi(11) = 10$).

Mit Hilfe der Primitivwurzel kann man die Bedingungen klar herausarbeiten, wann Potenzen modulo m *eindeutig invertierbar* sind und die Berechnung in den Exponenten handhabbar ist.[46]

Die folgenden beiden Tabellen 4.8 und 4.9 zeigen multiplikative Ordnung und Primitivwurzel modulo 45 und modulo 46.

[44] Das SageMath-Beispiel 4.5 enthält den Quelltext zur Berechnung der Tabelle 4.7. Siehe Kapitel 4.19.3 „Multiplikative Ordnung“.
[45] In Kapitel 4.19.4, „Primitivwurzeln“ finden Sie SageMath-Programme zur Berechnung von Primitivwurzeln.
[46] Eine gute Übersicht zu Primitivwurzeln findet sich auch in Wikipedia: `https://de.wikipedia.org/wiki/Primitivwurzel` und vor allem `https://en.wikipedia.org/wiki/Primitive_root_modulo_n`.

Beispiel 2:
Die folgende Tabelle 4.8[47] zeigt die Werte a^i mod 45 für die Exponenten $i = 1, 2, \cdots, 12$ und für die Basen $a = 1, 2, \cdots, 12$ sowie den sich für jedes a daraus ergebenden Wert $ord_{45}(a)$.

$a \setminus i$	1	2	3	4	5	6	7	8	9	10	11	12	$ord_{45}(a)$	$\phi(45)$
1	1	1	1	1	1	1	1	1	1	1	1	1	1	24
2	2	4	8	16	32	19	38	31	17	34	23	1	12	24
3	3	9	27	36	18	9	27	36	18	9	27	36	—	24
4	4	16	19	31	34	1	4	16	19	31	34	1	6	24
5	5	25	35	40	20	10	5	25	35	40	20	10	—	24
6	6	36	36	36	36	36	36	36	36	36	36	36	—	24
7	7	4	28	16	22	19	43	31	37	34	13	1	12	24
8	8	19	17	1	8	19	17	1	8	19	17	1	4	24
9	9	36	9	36	9	36	9	36	9	36	9	36	—	24
10	10	10	10	10	10	10	10	10	10	10	10	10	—	24
11	11	31	26	16	41	1	11	31	26	16	41	1	6	24
12	12	9	18	36	27	9	18	36	27	9	18	36	—	24

Tabelle 4.8: Werte von a^i mod $45, 1 \leq a, i < 13$ und zugehörige Ordnung von a modulo 45

$\phi(45)$ berechnet sich nach Satz 4.8.4: $\phi(45) = \phi(3^2 * 5) = [3^1 * 2] * [1 * 4] = 24$.

Da 45 keine Primzahl ist, gibt es nicht für alle Werte von a eine „Multiplikative Ordnung“ (für alle nicht zu 45 teilerfremden Zahlen: $3, 5, 6, 9, 10, 12, \cdots$, da $45 = 3^2 * 5$).

Beispiel 3:
Hat 7 eine Primitivwurzel modulo 45?

Die notwendige, aber nicht hinreichende Voraussetzung/Bedingung $\text{ggT}(7, 45) = 1$ ist erfüllt. Aus der Tabelle 4.8 kann man ersehen, dass die Zahl $a = 7$ keine Primitivwurzel von 45 ist, denn $ord_{45}(7) = 12 \neq 24 = \phi(45)$.

[47] Das SageMath-Beispiel 4.6 enthält den Quelltext zur Berechnung der Tabelle 4.8. Siehe Kapitel 4.19.3, „Multiplikative Ordnung“.

Beispiel 4:
Die folgende Tabelle 4.9[48] beantwortet die Frage, ob die Zahl $a = 7$ eine Primitivwurzel von 46 ist.

Die notwendige, aber nicht hinreichende Voraussetzung/Bedingung $\text{ggT}(7, 46) = 1$ ist erfüllt. $\phi(46)$ berechnet sich nach Satz 4.8.2: $\phi(46) = \phi(2 * 23) = 1 * 22 = 22$. Die Zahl 7 ist eine Primitivwurzel von 46, denn $ord_{46}(7) = 22 = \phi(46)$.

$a \backslash i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	ord
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	4	8	16	32	18	36	26	6	12	24	2	4	8	16	32	18	36	26	6	12	24	2	–
3	3	9	27	35	13	39	25	29	41	31	1	3	9	27	35	13	39	25	29	41	31	1	3	11
4	4	16	18	26	12	2	8	32	36	6	24	4	16	18	26	12	2	8	32	36	6	24	4	–
5	5	25	33	27	43	31	17	39	11	9	45	41	21	13	19	3	15	29	7	35	37	1	5	22
6	6	36	32	8	2	12	26	18	16	4	24	6	36	32	8	2	12	26	18	16	4	24	6	–
7	7	3	21	9	17	27	5	35	15	13	45	39	43	25	37	29	19	41	11	31	33	**1**	7	22
8	8	18	6	2	16	36	12	4	32	26	24	8	18	6	2	16	36	12	4	32	26	24	8	–
9	9	35	39	29	31	3	27	13	25	41	1	9	35	39	29	31	3	27	13	25	41	1	9	11
10	10	8	34	18	42	6	14	2	20	16	22	36	38	12	28	4	40	32	44	26	30	24	10	–
11	11	29	43	13	5	9	7	31	19	25	45	35	17	3	33	41	37	39	15	27	21	1	11	22
12	12	6	26	36	18	32	16	8	4	2	24	12	6	26	36	18	32	16	8	4	2	24	12	–
13	13	31	35	41	27	29	9	25	3	39	1	13	31	35	41	27	29	9	25	3	39	1	13	11
14	14	12	30	6	38	26	42	36	44	18	22	32	34	16	40	8	20	4	10	2	28	24	14	–
15	15	41	17	25	7	13	11	27	37	3	45	31	5	29	21	39	33	35	19	9	43	1	15	22
16	16	26	2	32	6	4	18	12	8	36	24	16	26	2	32	6	4	18	12	8	36	24	16	–
17	17	13	37	31	21	35	43	41	7	27	45	29	33	9	15	25	11	3	5	39	19	1	17	22
18	18	2	36	4	26	8	6	16	12	32	24	18	2	36	4	26	8	6	16	12	32	24	18	–
19	19	39	5	3	11	25	15	9	33	29	45	27	7	41	43	35	21	31	37	13	17	1	19	22
20	20	32	42	12	10	16	44	6	28	8	22	26	14	4	34	36	30	2	40	18	38	24	20	–
21	21	27	15	39	37	41	33	3	17	35	45	25	19	31	7	9	5	13	43	29	11	1	21	22
22	22	24	22	24	22	24	22	24	22	24	22	24	22	24	22	24	22	24	22	24	22	24	22	–
23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	–

Tabelle 4.9: Werte von $a^i \bmod 46, 1 \leq a, i < 24$ und zugehörige Ordnung von a modulo 46

Satz 4.9.1. *Für einen Modul n und eine Zahl a, teilerfremd zu n gilt:*
Die Menge $\{a^i \ (mod\ n) |\ i = 1, \ldots, \phi(n)\}$ *ist gleich der multiplikativen Gruppe* Z_n^* *genau dann, wenn* $ord_n(a) = \phi(n)$.[49,50]

Die multiplikative Gruppe Z_n^* nimmt nur dann alle Werte von 1 bis $n - 1$ an, wenn n prim ist (vgl. Definition 4.7.2).

[48] Das SageMath-Beispiel 4.7 enthält den Quelltext zur Berechnung der Tabelle 4.9. Siehe Kapitel 4.19.3, „Multiplikative Ordnung“.

[49] Für Primmoduli p haben alle a mit $0 < a < p$ die Ordnung $\phi(p) = p - 1$. Vergleiche dazu Tabelle 4.8. In diesem Fall nimmt $a^i (\text{mod } n)$ alle Werte $1, \ldots, p-1$ an. Dieses Ausschöpfen des Wertebereiches ist eine wichtige kryptographische Eigenschaft (vergleiche Satz 4.6.2). Hiermit wird eine Permutation $\pi(p-1)$ festgelegt.

[50] Tabelle 4.9 zeigt, dass bei zusammengesetzten Moduli n nicht alle a die maximale Ordnung $\phi(n)$ haben. In diesem Beispiel haben nur $5, 7, 11, 15, 17, 19$ und 21 die Ordnung 22.

Beispiel 5: Zykluslängen

Die folgenden Tabellen 4.10 und 4.11[51] dienen als Beispiele, um Zykluslängen zu betrachten – ein Thema, das über die multiplikative Ordnung hinausführt.

Als Zyklus bezeichnet man hier eine Folge von Zahlen a^i mod n mit $1 \leq i < n$ für ein festes a, wobei sich die Folge wiederholt. Hier kommt innerhalb eines Zyklus jede Zahl nur einmal vor (aufgrund der Erzeugung als modulare Potenz). Die Zyklen hier müssen die 1 nicht enthalten – es sei denn, dieser Zyklus gehört zu einer multiplikativen Ordnung ≥ 1 (diese haben die 1 immer am Ende des Zyklus und an der Stelle a^{n-1} mod n).
Im Folgenden bezeichnet l die Zykluslänge.

Die maximale Zykluslänge l_{max} ist $\phi(n)$.

Für die folgenden Tabellen 4.10 und 4.11 gilt (nach Satz 4.8.4):
- $\phi(14) = \phi(2 * 7) = 1 * 6 = 6$.
- $\phi(22) = \phi(2 * 11) = 1 * 10 = 10$.

a) Falls die multiplikative Ordnung für a existiert, gilt (egal ob a prim ist): $ord_n(a) = l$.
Beispiele: Die maximale Länge l_{max}[52] wird beispielsweise erreicht für:
- $a = 3$ mit $l_{max} = ord_{14}(a) = 6$ in Tabelle 4.10, oder
- $a = 10$ mit $l_{max} = ord_{22}(a) = 10$ in Tabelle 4.11.

b) Auch wenn für a keine multiplikative Ordnung existiert, kann die Zykluslänge maximal sein.[53]
Beispiele:
- In Tabelle 4.10 ist $l_{max} = \phi(14) = 6$ für $a = 10, 12$.
- In Tabelle 4.11 ist $l_{max} = \phi(22) = 10$ für $a = 2, 6, 8, 18$.

[51] In Kapitel 4.19.3, „Multiplikative Ordnung“ finden Sie den Quelltext zur Berechnung der Tabellen 4.10 und 4.11 mit SageMath.

[52] Wir kennen keine Formel, für welche a die maximale Länge erreicht wird.

[53] Die Folgen hier werden gebildet per a^i mod n mit $1 \leq i < n$, und enthalten für zusammengesetzte n nie alle Zahlen $1, ..., n-1$.
Dies ist nicht zu verwechseln mit RSA, wo die „Folge“ anders gebildet wird, m^e mod n mit $0 \leq m < n$, und diese Folge dann alle Zahlen $0, ..., n-1$ durchläuft (Permutation).

$a \setminus i$	1	2	3	4	5	6	7	8	9	10	11	12	13	$ord_{14}(a)$	$\phi(14)$	l
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	6	1
2	2	4	8	2	4	8	2	4	8	2	4	8	2	0	6	3
3	3	9	13	11	5	1	3	9	13	11	5	1	3	6	6	6
4	4	2	8	4	2	8	4	2	8	4	2	8	4	0	6	3
5	5	11	13	9	3	1	5	11	13	9	3	1	5	6	6	6
6	6	8	6	8	6	8	6	8	6	8	6	8	6	0	6	2
7	7	7	7	7	7	7	7	7	7	7	7	7	7	0	6	1
8	8	8	8	8	8	8	8	8	8	8	8	8	8	0	6	1
9	9	11	1	9	11	1	9	11	1	9	11	1	9	3	6	3
10	10	2	6	4	12	8	10	2	6	4	12	8	10	0	6	6
11	11	9	1	11	9	1	11	9	1	11	9	1	11	3	6	3
12	12	4	6	2	10	8	12	4	6	2	10	8	12	0	6	6
13	13	1	13	1	13	1	13	1	13	1	13	1	13	2	6	2
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	1
15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	6	1
16	2	4	8	2	4	8	2	4	8	2	4	8	2	0	6	3

Tabelle 4.10: Werte von $a^i \bmod 14, 1 \leq a < 17, i < 14$

$a \backslash i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	$ord_{22}(a)$	l
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	4	8	16	10	20	18	14	6	12	2	4	8	16	10	20	18	14	6	12	2	0	10
3	3	9	5	15	1	3	9	5	15	1	3	9	5	15	1	3	9	5	15	1	3	5	5
4	4	16	20	14	12	4	16	20	14	12	4	16	20	14	12	4	16	20	14	12	4	0	5
5	5	3	15	9	1	5	3	15	9	1	5	3	15	9	1	5	3	15	9	1	5	5	5
6	6	14	18	20	10	16	8	4	2	12	6	14	18	20	10	16	8	4	2	12	6	0	10
7	7	5	13	3	21	15	17	9	19	1	7	5	13	3	21	15	17	9	19	1	7	10	10
8	8	20	6	4	10	14	2	16	18	12	8	20	6	4	10	14	2	16	18	12	8	0	10
9	9	15	3	5	1	9	15	3	5	1	9	15	3	5	1	9	15	3	5	1	9	5	5
10	10	12	10	12	10	12	10	12	10	12	10	12	10	12	10	12	10	12	10	12	10	0	2
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	0	1
12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	0	1
13	13	15	19	5	21	9	7	3	17	1	13	15	19	5	21	9	7	3	17	1	13	10	10
14	14	20	16	4	12	14	20	16	4	12	14	20	16	4	12	14	20	16	4	12	14	0	5
15	15	5	9	3	1	15	5	9	3	1	15	5	9	3	1	15	5	9	3	1	15	5	5
16	16	14	4	20	12	16	14	4	20	12	16	14	4	20	12	16	14	4	20	12	16	0	5
17	17	3	7	9	21	5	19	15	13	1	17	3	7	9	21	5	19	15	13	1	17	10	10
18	18	16	2	14	10	4	6	20	8	12	18	16	2	14	10	4	6	20	8	12	18	0	10
19	19	9	17	15	21	3	13	5	7	1	19	9	17	15	21	3	13	5	7	1	19	10	10
20	20	4	14	16	12	20	4	14	16	12	20	4	14	16	12	20	4	14	16	12	20	0	5
21	21	1	21	1	21	1	21	1	21	1	21	1	21	1	21	1	21	1	21	1	21	2	2
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
23	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
24	2	4	8	16	10	20	18	14	6	12	2	4	8	16	10	20	18	14	6	12	2	0	10
25	3	9	5	15	1	3	9	5	15	1	3	9	5	15	1	3	9	5	15	1	3	5	5

Tabelle 4.11: Werte von $a^i \bmod 22, 1 \leq a < 26, i < 22$

4.10 Beweis des RSA-Verfahrens mit Euler-Fermat

Mit dem Satz von Euler-Fermat kann man in der Gruppe $\mathbb{Z}_n^*$ das RSA[54]-Verfahren „beweisen“.

4.10.1 Grundidee der Public-Key-Kryptographie

Die Grundidee bei der Public-Key-Kryptographie besteht darin, dass alle Teilnehmer ein unterschiedliches Paar von Schlüsseln (P und S) besitzen und man für alle Empfänger die öffentlichen Schlüssel publiziert. So wie man die Telefonnummer einer Person aus dem Telefonbuch nachschlägt, kann man den öffentlichen Schlüssel P (public) des Empfängers aus einem Verzeichnis entnehmen. Außerdem hat jeder Empfänger einen geheimen Schlüssel S (secret), der zum Entschlüsseln benötigt wird und den niemand sonst kennt. Möchte der Sender eine Nachricht M (message) schicken, verschlüsselt er diese Nachricht mit dem öffentlichen Schlüssel P des Empfängers, bevor er sie abschickt:

Der Chiffretext C (ciphertext) ergibt sich mit $C = E(P; M)$, wobei E (encryption) die Verschlüsselungsvorschrift ist. Der Empfänger benutzt seinen privaten Schlüssel S, um die Nachricht wieder mit der Entschlüsselungsvorschrift D (decryption) zu entschlüsseln: $M = D(S; C)$.

Damit dieses System mit jeder Nachricht M funktioniert, müssen folgende 4 **Forderungen** erfüllt sein:

1. $D(S; E(P; M)) = M$ für jedes M (Umkehrbarkeit) und M nimmt „sehr viele“ verschiedene Werte an.
2. Alle (S, P)-Paare aller Teilnehmer sind verschieden.
3. Der Aufwand, S aus P herzuleiten, ist mindestens so hoch, wie das Entschlüsseln von M ohne Kenntnis von S.
4. Sowohl C als auch M lassen sich relativ einfach berechnen.

Die 1. Forderung ist eine generelle Bedingung für alle kryptographischen Verschlüsselungsalgorithmen.

Die Voraussetzung für die 2. Forderung kann leicht sichergestellt werden, weil es „sehr“ viele Primzahlen gibt[55]. Hinzu kommen muss noch, dass eine zentrale Stelle, die Zertifikate ausgibt, sicher stellt, dass das erfüllt ist (siehe Kapitel 4.11.5.4, S. 165).

Die letzte Forderung macht das Verfahren überhaupt erst anwendbar. Dies geht, weil die modulare Potenzierung in linearer Zeit möglich ist (da die Zahlen längenbeschränkt sind).

[54] Das RSA-Verfahren ist das verbreitetste asymmetrische Kryptoverfahren. Es wurde 1978 von Ronald Rivest, Adi Shamir und Leonard Adleman entwickelt und kann sowohl zum Signieren wie zum Verschlüsseln eingesetzt werden. Kryptographen assoziieren mit der Abkürzung „**RSA**“ immer dieses Verfahren – die folgende Anmerkung soll eher humorvoll zeigen, dass man jede Buchstabenkombination mehrfach belegen kann: Als Länderkürzel steht „RSA“ für die Republik Südafrika. In Deutschland gibt es sehr Interessen-lastige Diskussionen im Gesundheitswesen. Dabei wird mit „RSA“ der vom Gesundheitsministerium geregelte „**R**isiko**S**truktur**A**usgleich“ in der gesetzlichen Krankenversicherung bezeichnet: Im Jahr 2004 wurden durch den RSA ca. 16 Mrd. Euro zwischen den Krankenkassen umverteilt. Siehe `http://www.abbreviationfinder.org/de/acronyms/rsa.html`.

[55] Nach dem **Primzahlsatz** (Kapitel 3.7.2, S. 87) von Legendre und Gauss gibt es bis zur Zahl n asymptotisch $n/\ln(n)$ Primzahlen. Dies bedeutet beispielsweise: Es gibt $6,5*10^{74}$ Primzahlen unterhalb von $n = 2^{256}$ ($1,1*10^{77}$) und $3,2*10^{74}$ Primzahlen unterhalb von $n = 2^{255}$. Zwischen 2^{256} und 2^{255} gibt es also allein $3,3*10^{74}$ Primzahlen mit genau 256 Bits. Aufgrund dieser hohen Anzahl von Primzahlen kann man sie nicht einfach alle abspeichern – schon aus Gründen der Physik: vgl. die Anzahl der Atome im Universum in der Übersicht unter 3.11.

Während Whitfield Diffie und Martin Hellman schon 1976 das generelle Schema formulierten, fanden erst Rivest, Shamir und Adleman ein konkretes Verfahren, das alle vier Bedingungen erfüllte.

4.10.2 Funktionsweise des RSA-Verfahrens

Man kann die Einzelschritte zur Durchführung des RSA-Verfahren folgendermaßen beschreiben (siehe [Eck14, S. 213 ff] und [Sed90, S. 338 ff]). Schritt 1 bis 3 sind die Schlüsselerzeugung, Schritt 4 und 5 sind die Verschlüsselung, 6 und 7 die Entschlüsselung:

1. Wähle zufällig 2 verschiedene Primzahlen[56,57] p und q und berechne $n = p * q$.[58]
   Der Wert n wird als RSA-Modul bezeichnet.[59]

2. Wähle ein beliebiges $e \in \{2, \cdots, n-1\}$, so dass gilt[60]:
   e ist teilerfremd zu $\phi(n) = (p-1) * (q-1)$.
   Danach kann man p und q „wegwerfen“.[61]

3. Wähle $d \in \{1, \cdots, n-1\}$ mit $e * d = 1 \mod \phi(n)$, d.h. d ist die multiplikative Inverse zu e modulo $\phi(n)$.[62,63] Danach kann man $\phi(n)$ „wegwerfen“.

   → (n, e) ist der öffentliche Schlüssel P.

   → (n, d) ist der geheime Schlüssel S (es ist nur d geheim zu halten).

4. Zum Verschlüsseln wird die als (binäre) Zahl dargestellte Nachricht in Teile aufgebrochen, so dass jede Teilzahl kleiner als n ist.

---

[56] Compaq hatte in 2000 mit hohem Marketingaufwand das sogenannte Multi-prime-Verfahren eingeführt. n war dabei das Produkt von zwei großen und einer relativ dazu kleinen Primzahl: $n = o * p * q$. Nach Satz 4.8.3 ist dann $\phi(n) = (o-1) * (p-1) * (q-1)$. Das Verfahren hat sich nicht durchgesetzt.
Mit ein Grund dafür dürfte sein, dass Compaq ein Patent dafür angemeldet hat. Generell gibt es in Europa und in der Open-Source-Bewegung wenig Verständnis für Patente auf Algorithmen. Überhaupt kein Verständnis herrscht außerhalb der USA, dass man auf den Sonderfall (3 Faktoren) eines Algorithmus (RSA) ein Patent beantragen kann, obwohl das Patent für den allgemeinen Fall schon fast abgelaufen war.
JCT enthält das Multi-prime RSA-Verfahren sowohl im Menü **Visualisierungen** der Standard-Perspektive als auch in der Algorithmen-Perspektive.

[57] Für Primzahlen p und q mit $p = q$, und e, d mit $ed \equiv 1 \mod \phi(n)$ gilt i.a. nicht $(m^e)^d \equiv m \mod n$ für alle $m < n$. Sei z.B. $n = 5^2$, berechnet sich $\phi(n)$ nach Satz 4.8.4: $\phi(n) = 5 * 4 = 20$, $e = 3$, $d = 7$, $ed \equiv 21 \equiv 1 \mod \phi(n)$. Dann gilt $(5^3)^7 \equiv 0 \mod 25$.

[58] Das BSI (Bundesamt für Sicherheit in der Informationstechnik) empfiehlt, die Primfaktoren p und q ungefähr gleich groß zu wählen, aber nicht zu dicht beieinander, d.h. konkret etwa

$$0.5 < |\log_2(p) - \log_2(q)| < 30.$$

Dabei sollen die Primfaktoren unter Beachtung der genannten Nebenbedingung zufällig und unabhängig voneinander erzeugt werden (siehe [BSI16]).

[59] In CT1 und auch oftmals in der Literatur wird der RSA-Modul mit einem großen „N“ bezeichnet.

[60] Empfehlenswert aus kryptoanalytischen Gründen, aber nicht notwendig für das Funktionieren des Verfahrens ist es, e so zu wählen, dass gilt: $\max(p, q) < e < \phi(n) - 1$.

[61] Das Verfahren erlaubt es auch, d frei zu wählen und dann e zu berechnen. Dies hat aber praktische Nachteile. Normalerweise will man „schnell“ verschlüsseln können und wählt deshalb einen öffentlichen Exponenten e so, dass er im Vergleich zum Modul n sehr kleine Bitlängen und möglichst wenige binäre Einsen hat (z.B. $2^{16} + 1$). Damit ist eine schnelle Exponentiation bei der Verschlüsselung möglich. Als besonders praktisch haben sich hierfür die Primzahlen $3, 17$ und 65537 erwiesen. Am häufigsten verwendet wird die Zahl $65537 = 2^{16} + 1$, also binär: $10 \cdots 0 \cdots 01$ (diese Zahl ist prim und deshalb zu sehr vielen Zahlen teilerfremd).

[62] Aus Sicherheitsgründen darf d nicht zu klein sein.

[63] Je nach Implementierung wird zuerst d oder zuerst e bestimmt.

5. Verschlüsselung des Klartextes (bzw. seiner Teilstücke) $M \in \{1, \cdots, n-1\}$:

$$C = E((n,e); M) := M^e \bmod n.$$

6. Zum Entschlüsseln wird das binär als Zahl dargestellte Chiffrat in Teile aufgebrochen, so dass jede Teilzahl kleiner als n ist.

7. Entschlüsselung des Chiffretextes (bzw. seiner Teilstücke) $C \in \{1, \cdots, n-1\}$:

$$M = D((n,d); C) := C^d \bmod n.$$

Die Zahlen d, e, n sind normalerweise sehr groß (z.B. sind d und e 300 bit, und n 600 bit lang).

Bemerkung:
Die Sicherheit des RSA-Verfahrens hängt wie bei allen Public-Key-Verfahren davon ab, dass man den privaten Key d nicht aus dem öffentlichen Key (n, e) berechnen kann.

Beim RSA-Verfahren bedeutet dies,

1. dass $\phi(n)$ für große zusammengesetzte n schwer zu berechnen ist, und
2. dass n für große n nur schwer in seine Primfaktoren zerlegt werden kann (Faktorisierungsproblem).[64]

4.10.3 Beweis der Forderung 1 (Umkehrbarkeit)

Für Schlüsselpaare (n, e) und (n, d), die die in den Schritten 1 bis 3 des RSA-Verfahrens festgelegten Eigenschaften besitzen, muss für alle $M < n$ gelten:

$$M \equiv (M^e)^d \pmod n \quad \text{wobei} \quad (M^e)^d = M^{e*d}.$$

Das heißt, der oben angegebene Dechiffrieralgorithmus arbeitet korrekt.

Zu zeigen ist also:

$$M^{e*d} \equiv M \pmod n$$

Wir zeigen das in 3 Schritten mit Hilfe von Satz 4.8.5 (Kleiner Satz von Fermat)
(siehe [Beu96, S. 131ff]).

Schritt 1:

Im ersten Schritt zeigen wir: $M^{e*d} \equiv M \pmod p$

Da $n = p * q$ und $\phi(p * q) = (p-1) * (q-1)$ und da e und d so gewählt sind, dass $e * d \equiv 1 \pmod{\phi(n)}$, gibt es eine ganze Zahl k, so dass gilt: $e * d = 1 + k * (p-1) * (q-1)$.

$$\begin{aligned} M^{e*d} &\equiv M^{1+k*\phi(n)} \equiv M * M^{k*\phi(n)} \equiv M * M^{k*(p-1)*(q-1)} \pmod p \\ &\equiv M * (M^{p-1})^{k*(q-1)} \pmod p \quad \text{aufgrund des kleinen Fermat : } M^{p-1} \equiv 1 \pmod p \\ &\equiv M * (1)^{k*(q-1)} \pmod p \\ &\equiv M \pmod p. \end{aligned}$$

[64] Für die manchmal geäußerte Sorge, dass es nicht genug Primzahlen gäbe, besteht kein Grund: Dass es immer ausreichend Primzahlen gibt, wenn man die Dimension (Exponenten) des Moduls hochsetzt, wird visualisiert in Kapitel 3.13 „Anhang: Visualisierung der Menge der Primzahlen in hohen Bereichen“.

Die Voraussetzung für die Anwendung des vereinfachten Satzes von Euler-Fermat (Kleiner-Fermat Satz 4.8.5) war, dass M und p teilerfremd sind.

Da das im allgemeinen nicht gilt, müssen wir noch betrachten, was ist, wenn M und p nicht teilerfremd sind: Da p eine Primzahl ist, muss dann notwendigerweise p ein Teiler von M sein. Das heißt aber:

$$M \equiv 0 \pmod{p}$$

Wenn p die Zahl M teilt, so teilt p erst recht M^{e*d}. Also ist auch:

$$M^{e*d} \equiv 0 \pmod{p}.$$

Da p sowohl M als auch M^{e*d} teilt, teilt er auch ihre Differenz:

$$(M^{e*d} - M) \equiv 0 \pmod{p}.$$

Und damit gilt auch in diesem Spezialfall unsere zu beweisende Behauptung.

Schritt 2:

Völlig analog beweist man: $M^{e*d} \equiv M \pmod{q}$.

Schritt 3:

Nun führen wir die Behauptungen aus Schritt 1 und 2 zusammen für $n = p * q$, um zu zeigen:

$$M^{e*d} \equiv M \pmod{n} \text{ für alle } M < n.$$

Nach Schritt 1 und 2 gilt $(M^{e*d} - M) \equiv 0 \pmod{p}$ und $(M^{e*d} - M) \equiv 0 \pmod{q}$, also teilen p und q jeweils dieselbe Zahl $z = (M^{e*d} - M)$. Da p und q **verschiedene** Primzahlen sind, muss dann auch ihr Produkt diese Zahl z teilen. Also gilt:

$$(M^{e*d} - M) \equiv 0 \pmod{p*q} \quad \text{oder} \quad M^{e*d} \equiv M \pmod{p*q} \quad \text{oder} \quad M^{e*d} \equiv M \pmod{n}.$$

□

Bemerkung 1:
Man kann die 3 Schritte auch kürzer zusammenfassen, wenn man Satz 4.8.6 (Euler-Fermat) benutzt (also nicht den vereinfachten Satz, wo $n = p$ gilt):

$$(M^e)^d \equiv M^{e*d} \equiv M^{(p-1)(q-1)*k+1} \equiv (\underbrace{M^{(p-1)(q-1)}}_{\equiv M^{\phi(n)} \equiv 1 \pmod{n}})^k * M \equiv 1^k * M \equiv M \pmod{n}.$$

Bemerkung 2:
Beim Signieren werden die gleichen Operationen durchgeführt, aber zuerst mit dem geheimen Schlüssel d, und dann mit dem öffentlichen Schlüssel e. Das RSA-Verfahren ist auch für die Erstellung von digitalen Signaturen einsetzbar, weil gilt:

$$M \equiv (M^d)^e \pmod{n}.$$

4.11 Zur Sicherheit des RSA-Verfahrens[65]

Die Eignung des RSA-Verfahrens für digitale Signaturen und Verschlüsselung gerät immer wieder in die Diskussion, z.B. im Zusammenhang mit der Veröffentlichung aktueller Faktorisierungserfolge. Ungeachtet dessen ist das RSA-Verfahren seit seiner Veröffentlichung vor mehr als 20 Jahren unangefochtener De-facto-Standard (vgl. 7.1).

Die Sicherheit des RSA-Verfahrens basiert - wie die aller kryptographischen Verfahren - auf den folgenden 4 zentralen Säulen:

- der Komplexität des dem Problem zugrunde liegenden zahlentheoretischen Problems (hier der Faktorisierung großer Zahlen),
- der Wahl geeigneter Sicherheitsparameter (hier der Länge des Moduls n),
- der geeigneten Anwendung des Verfahrens sowie der Schlüsselerzeugung und
- der korrekten Implementierung des Algorithmus.

Die Anwendung und Schlüsselerzeugung wird heute gut beherrscht. Die Implementierung ist auf Basis einer Langzahlarithmetik sehr einfach.

In den folgenden Abschnitten werden die ersten beiden Punkte näher untersucht.

4.11.1 Komplexität

Ein erfolgreiches Entschlüsseln oder eine Signaturfälschung — ohne Kenntnis des geheimen Schlüssels — erfordert, die e-te Wurzel mod n zu ziehen. Der geheime Schlüssel, nämlich die multiplikative Inverse zu e mod $\phi(n)$, kann leicht bestimmt werden, wenn die Eulersche Funktion $\phi(n)$ bekannt ist. $\phi(n)$ wiederum lässt sich aus den Primfaktoren von n berechnen. Das Brechen des RSA-Verfahrens kann daher nicht schwieriger sein als das Faktorisieren des Moduls n.

Das beste heute bekannte Verfahren ist eine Weiterentwicklung des ursprünglich für Zahlen mit einer bestimmten Darstellung (z.B. Fermatzahlen) entwickelten General Number Field Sieve (GNFS). Die Lösungskomplexität des Faktorisierungsproblems liegt damit asymptotisch bei

$$O(l) = e^{c\cdot(l\cdot\ln 2)^{1/3}\cdot(\ln(l\cdot\ln(2))^{2/3}+o(l)}$$

Siehe: [LL93] und [Sil00]

Die Formel zeigt, dass das Faktorisierungsproblem zur Komplexitätsklasse der Probleme mit subexponentieller Berechnungskomplexität gehört (d.h. der Lösungsaufwand wächst asymptotisch nicht so stark wie e^l oder 2^l, sondern echt schwächer, z. B. wie $e^{\sqrt{l}}$). Diese Einordnung entspricht dem heutigen Kenntnisstand, sie bedeutet jedoch nicht, dass das Faktorisierungsproblem möglicherweise nicht auch mit (asymptotisch) polynominellem Aufwand gelöst werden kann (s. 4.11.5.1).

$O(l)$ gibt die Zahl der durchschnittlich erforderlichen Prozessor-Operationen abhängig von der Bitlänge l der zu faktorisierenden Zahl n an. Für den besten allgemeinen Faktorisierungsalgorithmus ist die Konstante $c = (64/9)^{1/173} = 1,923$.

[65] Große Teile des ersten Teils in Kapitel 4.11 sind angelehnt an den Artikel „Vorzüge und Grenzen des RSA-Verfahrens“ von F. Bourseau, D. Fox und C. Thiel [BFT02].

Die umgekehrte Aussage, dass das RSA-Verfahren ausschließlich durch eine Faktorisierung von n gebrochen werden kann, ist bis heute nicht bewiesen. Zahlentheoretiker halten das „RSA-Problem“ und das Faktorisierungsproblem für komplexitätstheoretisch äquivalent.

Siehe: *Handbook of Applied Cryptography* [MvOV01].

4.11.2 Sicherheitsparameter aufgrund neuer Algorithmen

Faktorisierungsalgorithmen[66]

Die Komplexität wird im wesentlichen von der Länge l des Moduls n bestimmt. Höhere Werte für diesen wesentlichen Parameter orientieren sich an den Möglichkeiten der aktuellen Faktorisierungsalgorithmen:

- 1994 wurde mit einer verteilten Implementierung des 1982 von Pomerance entwickelten Quadratic Sieve-Algorithmus (QS) nach knapp 8 Monaten der 1977 veröffentlichte 129-stellige RSA-Modul (428 bit) faktorisiert.
  Siehe:
  C. Pomerance: *The quadratic sieve factoring algorithm* [Pom84].

- 1999 wurde mit dem von Buhler, Lenstra und Pomerance entwickelten General Number Field Sieve-Algorithmus (GNFS), der ab ca. 116 Dezimalstellen effizienter ist als QS, nach knapp 5 Monaten ein 155-stelliger Modul (512 bit) faktorisiert.
  Siehe:
  J.P. Buhler, H.W. Lenstra, C. Pomerance: *Factoring integers with the number field sieve* [BLP93].

- Ende 2009, also rund 10 Jahre später, wurde von Kleinjung etc. nach rund 2 1/2 Jahren ein 232-stelliger Modul (768 bit) faktorisiert.
  Siehe:
  T. Kleinjung, et. al.: *Factorization of a 768-bit RSA modulus* [Kle10].

Damit ist auch praktisch klar geworden, dass eine Modullänge von 768 bit keinen Schutz mehr vor Angreifern darstellt.

Details zu den Fortschritten bei der Faktorisierung seit 1999: siehe Kapitel 4.11.4.

[66] Mit dem Lernprogramm **ZT** können Sie einige der gängigen Faktorisierungsverfahren (Fermat, Pollard-Rho, Pollard p-1, QS) vertiefen (siehe Lern-Kapitel 5.1-5.5, Seiten 1-15/15).
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.
Das Quadratische Sieb (QS) finden Sie in CT1 und CT2; GNFS, das modernste Faktorisierungsverfahren für Moduli größer 130 Dezimalstellen, ist (mittels YAFU und msieve) nur in CT2 enthalten.
CT2 hat die Komponente GeneralFactorizer (basierend auf YAFU). Diese ist schneller als die in CT1 implementierten Funktionen. Damit bietet CT2 die folgenden Faktorisierungsverfahren:
- Brute-force mit kleinen Primzahlen
- Fermat
- Shanks Square Forms Factorization (squfof)
- Pollard rho
- Pollard p-1
- Williams p+1
- Lenstra Elliptic Curve Method (ECM)
- Self-initializing Quadratic Sieve (SIQS)
- Multiple-polynomial Quadratic Qieve (MPQS)
- Special Number Field Sieve (SNFS)
- General Number Field Sieve (GNFS).

Algorithmen zur Gitterbasenreduktion

Die Modullänge l ist aber nicht der einzige Sicherheits-relevante Parameter. Neben Implementierungsanforderungen kommt es auch auf die Größen und die Größenverhältnisse der Parameter e, d und n an.

Entsprechende Angriffe, die auf Gitterbasenreduktion beruhen, stellen für einfache RSA-Implementierungen eine reale Bedrohung dar. Diese Angriffe lassen sich in die folgenden vier Kategorien unterteilen:

- Angriffe auf sehr kleine öffentliche Schlüssel e (z.B. $e = 3$).
- Angriffe auf relativ kurze geheime Exponenten d (z.B. $d < n^{0,5}$).
- Faktorisierung des Moduls n, wenn einer der Faktoren p oder q *teilweise* bekannt ist.
- Angriffe, die voraussetzen, dass ein *Teil* des geheimen Schlüssels d bekannt ist.

Eine gute Übersicht zu diesen Angriffen findet sich in der Diplomarbeit *Analyse der Sicherheit des RSA-Algorithmus. Mögliche Angriffe, deren Einfluss auf sichere Implementierungen und ökonomische Konsequenzen* von Matthias Schneider [Sch04].

4.11.3 Vorhersagen zur Faktorisierung großer Zahlen

Seit 1980 wurden erhebliche Fortschritte gemacht. Abschätzungen über die zukünftige Entwicklung der Sicherheit unterschiedlicher RSA-Modullängen differieren und hängen von verschiedenen Annahmen ab:

- Entwicklung der Rechnergeschwindigkeit (Gesetz von Moore: Verdopplung der Rechnerleistung alle 18 Monate) und des Grid-Computing.
- Entwicklung neuer Algorithmen.

Selbst ohne neue Algorithmen wurden in den letzten Jahren durchschnittlich ca. 10 Bit mehr pro Jahr berechenbar. Größere Zahlen erfordern mit den heute bekannten Verfahren einen immer größeren Arbeitsspeicher für die Lösungsmatrix. Dieser Speicherbedarf wächst mit der Wurzel des Rechenzeitbedarfs, also ebenfalls subexponentiell. Da in den letzten Jahren der verfügbare Hauptspeicher ebenso wie die Rechengeschwindigkeit exponentiell gewachsen ist, dürfte hierdurch kein zusätzlicher Engpass entstehen.

Lenstra/Verheul [LV01] trafen eine Abschätzung für die Entwicklung sicherer Schlüssellängen (vergleiche Abbildung 7.1 in Kapitel 7.1).

In dem Artikel [BFT02] veröffentlichte Dirk Fox[67] seine Prognose über einen annähernd linearen Verlauf der Faktorisierungserfolge unter Einbeziehung aller Einflussfaktoren: Pro Jahr kommen durchschnittlich 20 Bit dazu. Damit liegt er unter den optimistischeren Schätzungen von BSI und NIST.

[67] Seine Firma Secorvo GmbH gab eine Stellungnahme zur Schlüssellängenempfehlung des BSI für den Bundesanzeiger ab. Kapitel 2.3.1 dieser Stellungnahme geht kompetent und verständlich auf RSA-Sicherheit ein (existiert nur in Deutsch):
`https://www.secorvo.de/publikationen/stellungnahme-algorithmenempfehlung-020307.pdf`

Diese Prognose von Dirk Fox aus dem Jahr 2001 scheint sich anhand der neuesten Faktorisierungsrekorde von RSA-200 und RSA-768 (siehe Kapitel 4.11.4) zu bestätigen: Seine Schätzung für das Jahr 2005 mit 660 Bit hat die Bitlänge von RSA-200 nahezu auf den Punkt getroffen (vergleiche Abbildung 4.2).

Hält die Prognose, dann ist die Faktorisierung eines RSA-Moduls der Länge 1024 bit im Jahre 2020 zu erwarten.

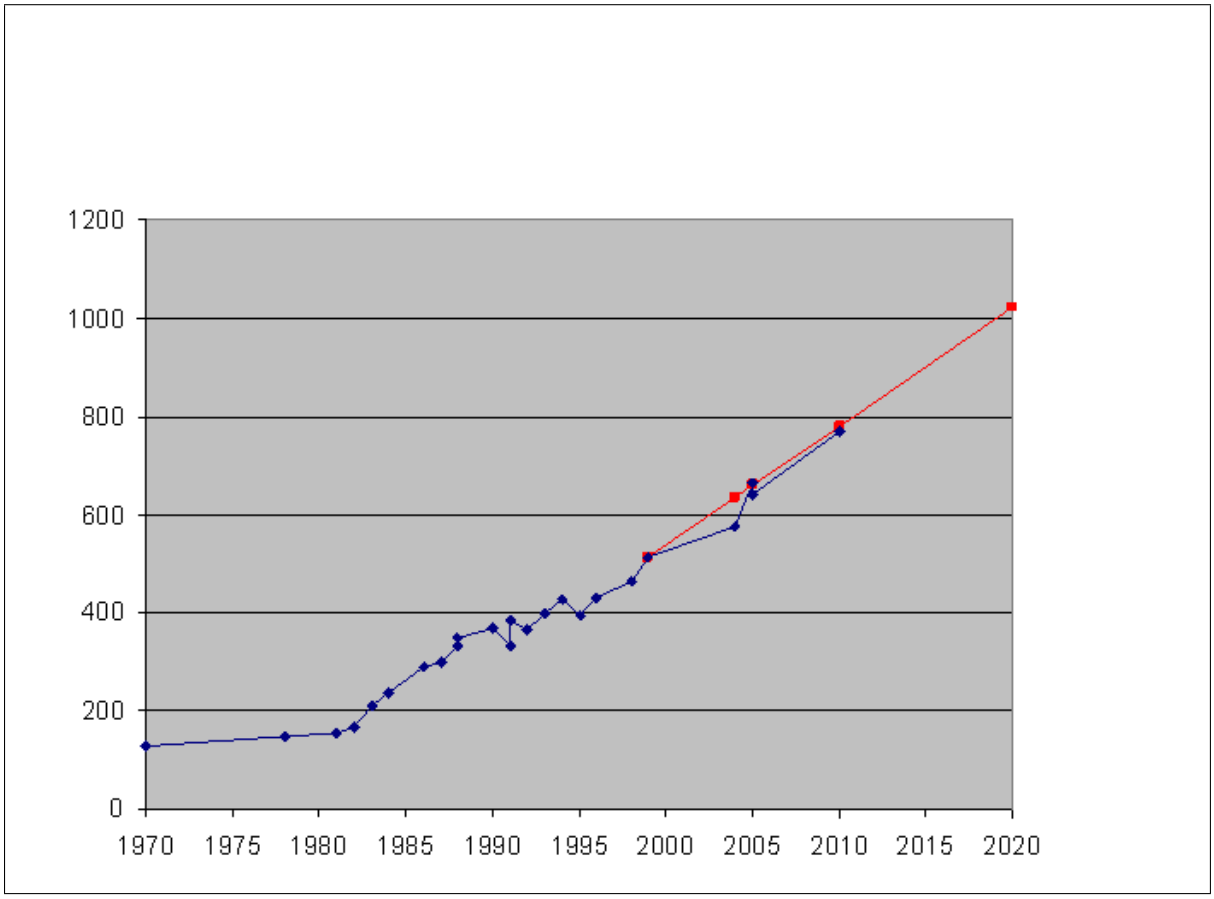


Abbildung 4.2: Vergleich der publizierten Faktorisierungserfolge (blau) mit der prognostizierten Entwicklung (rot) [Quelle Fox 2001; letzte Ergänzung 2011]

Damit das Mögliche entsteht, muss immer wieder das Unmögliche versucht werden.

Zitat 11: Hermann Hesse[68]

4.11.4 Status der Faktorisierung von konkreten großen Zahlen

Ausführliche Übersichten über die Rekorde im Faktorisieren zusammengesetzter Zahlen mit unterschiedlichen Methoden finden sich auf den folgenden Webseiten:

`http://primerecords.dk/consecutive_factorizations.htm`
`http://en.wikipedia.org/wiki/Integer_factorization_records`
`http://en.wikipedia.org/wiki/RSA_Factoring_Challenge`

Der aktuelle Rekord (Stand Nov. 2012) mit der GNFS-Methode (General Number Field Sieve) zerlegte eine allgemeine 232-stellige Dezimalzahl in ihre beiden Primfaktoren.

Die letzten Rekorde[69] mit Faktorisierungsverfahren für zusammengesetzte Zahlen sind in Tabelle 4.12 aufgeführt.

	Dezimalstellen	**Binärstellen**	**Faktorisiert am**	**Faktorisiert von**
RSA-768	232	768	Dez 2010	Thorsten Kleinjung et al.
RSA-200	200	663	Mai 2005	Jens Franke et al.
RSA-640[70]	193	640	Nov 2005	Jens Franke et al.
RSA-576	174	576	Dez 2003	Jens Franke et al.
RSA-160	160	530	Apr 2003	Jens Franke et al.
RSA-155	155	512	Aug 1999	Herman te Riele et al.
...				
C307	307	1017	Mai 2007	Jens Franke et al.
C176	176	583	Mai 2005	Kazumaro Aoki et al.
C158	158	523	Jan 2002	Jens Franke et al.

Tabelle 4.12: Die derzeitigen Faktorisierungsrekorde (Stand Nov. 2012)

[68] Hermann Hesse, deutsch-schweizerischer Schriftsteller und Nobelpreisträger, 02.07.1877−09.08.1962.

[69] Die „RSA-Zahlen“ sind große semiprime Zahlen (d.h. Zahlen, die genau aus 2 Primfaktoren bestehen). Sie wurden von der Firma RSA Security generiert und veröffentlicht: Im Wettbewerb „RSA Factoring Challenge“ wurden die Primfaktoren dieser Zahlen gesucht.
Siehe `http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm`.
Die RSA-Laboratorien schreiben ihre Challenges schon seit Anfang der 90er-Jahre aus. In der ersten RSA-Factoring-Challenge wurden die Zahlen, von RSA-100 bis RSA-500, gemäß der Anzahl ihrer Dezimalstellen benannt; die zweite RSA-Factoring-Challenge benannte die Zahlen anhand der Anzahl ihrer Binärstellen. Innerhalb des zweiten Wettbewerbs wurden Geldpreise für die erfolgreiche Faktorisierung von RSA-576 bis RSA-2048 ausgelobt (RSA-576, RSA-640 etc. in 64-er Schritten — Die Zahl RSA-617 bildet eine Ausnahme, da sie vor der Änderung des Namensschemas erzeugt wurde). Leider beendete RSA Inc. die Challenge vorzeitig und zog die Preise zurück. Alle noch ungelösten RSA-Challenges der RSA-Labs finden sich inzwischen auch auf der Webseite des Krypto-Wettbewerbs „MysteryTwister C3“ (`http://www.mysterytwisterc3.org`).
Die „C-Zahlen“ stammen aus dem Cunningham-Projekt: `http://homes.cerias.purdue.edu/~ssw/cun/`. Diese sind Faktoren großer Mersennezahlen, die eine ganz spezielle Struktur haben. Dies macht es um Größenordnungen einfacher, sie zu faktorisieren, als Moduli gleicher Längen, die man für RSA erstellt.

[70] Eine Arbeitsgruppe des BSI löste die mit 20.000 US-Dollar dotierte Challenge RSA-640 mit Hilfe der GNFS-Methode. Die Forscher benötigten für die Zerlegung der Zahl in ihre beiden 320 Bit langen Primfaktoren rund

Laufzeituntersuchungen zur Faktorisierung mit den Open-Source Software-Paketen Pari-GP, SageMath, CrypTool 1 und CrypTool 2 finden sich in „Zeitexperimente zur Faktorisierung“ (siehe [SW10]).

Im Folgenden werden die letzten Rekorde aus Tabelle 4.12 etwas ausführlicher erläutert[71]:

RSA-155

Am 22. August 1999 fanden niederländische Forscher die Lösung dieser RSA-Challenge. Sie zerlegten eine 155-stellige Zahl in ihre beiden 78-stelligen Primfaktoren (vergleiche Kapitel 4.11.2). Mit der 512 Bit-Zahl RSA-155 war eine *magische* Grenze erreicht.

C158

Am 18. Januar 2002 zerlegten Forscher der Universität Bonn[72] mit der GNFS-Methode (General Number Field Sieve) eine 158-stellige Dezimalzahl in ihre beiden Primfaktoren (diese haben 73 und 86 Dezimalstellen).

Dieser Rekord fand deutlich weniger Aufmerksamkeit in der Presse als die Lösung von RSA-155.

Die Aufgabe der Bonner Wissenschaftler entsprang auch nicht einer Challenge, sondern die Aufgabe war, die letzten Primfaktoren der Zahl $2^{953} + 1$ zu finden (siehe „Wanted List“ des Cunningham-Projekts[73]).

Die 6 kleineren, schon vorher gefundenen Primfaktoren dieser Zahl waren:

$$3, 1907, 425796183929,$$
$$1624700279478894385598779655842584377,$$
$$380230673854944132443213909127182812 1 \quad \text{und}$$
$$1280648868301666714448025761291158720600 27.$$

Die drei kleinsten Faktoren können leicht[74] bestimmt werden. Die nächsten drei Primfaktoren wurden von P. Zimmerman[75], T. Grandlund und R. Harley in den Jahren 1999 und 2000 mit der Methode der Elliptischen Kurven gefunden.

Als letzter Faktor blieb der sogenannte Teiler „C158“, von dem man bis dahin wusste, dass er zusammengesetzt ist, aber man kannte seine Primfaktoren nicht (die folgenden drei Zeilen

---

fünf Monate Rechenzeit.

Die Forscher um Prof. Jens Franke (von der Universität Bonn, dem BSI und dem CWI) waren nicht auf die Geldpreise aus, sondern wollten die Grenzen der Forschung ausdehnen. Dadurch werden Aussagen über notwendige Minimallängen für sichere RSA-Moduli fundierter.

[71] Die beiden dabei benutzten Methoden GNFS und SNFS werden kurz z.B. auf den ff. Webseiten dargestellt:

`http://en.wikipedia.org/wiki/Special_number_field_sieve`
`http://en.wikipedia.org/wiki/General_number_field_sieve`

[72] `https://members.loria.fr/PZimmermann/records/gnfs158`

[73] Cunningham-Projekt: `http://homes.cerias.purdue.edu/~ssw/cun/`

[74] Z.B. mit CT1 über das Menü **Einzelverfahren \ RSA-Kryptosystem \ Faktorisieren einer Zahl**. In sinnvoller Zeit zerlegt CT1 Zahlen bis 250 Bit Länge (Zahlen größer als 1024 Bit werden von CT1 nicht angenommen). CT2 kann auch Zahlen größer 250 Bit zerlegen.

[75] `http://homepages.loria.fr/PZimmermann/ecmnet/`

sind eine einzige Zahl):

$$395058745832651445264197678006144819960207764603049364541393760515793556265294506836097278424682195350935443058704902519956553357102097992264849779494429556033388495837466721394368393204672181522815830368604993048084929258405552811771165882340667125990314837655838327081813101225814639260043952099413134433416292453613 9.$$

Die Faktorisierung von C158 ergab die beiden 73- und 86-stelligen Primfaktoren:

$$33884958374667213943683932046721815228158303686049930480849258405552811771$$

und

$$11658823406671259903148376558383270818131012258146392600439520994131344334162924536139.$$

Damit wurde die Zahl $2^{953} + 1$ vollständig in ihre 8 Primfaktoren zerlegt.

Verweise:

- `https://members.loria.fr/PZimmermann/records/gnfs158`
- `https://web.archive.org/web/20170518021747/http://www.crypto-world.com:80/announcements/c158.txt`

RSA-160

Am 1. April 2003 zerlegten Forscher der Universität Bonn[76] mit der GNFS-Methode (General Number Field Sieve) eine 160-stellige Zahl in ihre beiden Primfaktoren (diese haben jeweils 80 Dezimalstellen).

Die Berechnungen dazu fanden auch im Bundesamt für Sicherheit in der Informationstechnik (BSI) in Bonn statt.[77]

Die 160-stellige Dezimalzahl stammt von der alten Challenge-Liste von RSADSI. Diese wurde nach der Faktorisierung von RSA-155 (RSA512) zurückgezogen. Die Primfaktoren von RSA-160 waren aber nicht bekannt. Deshalb ist dieser Rekord von Prof. Frankes Team immer noch die Lösung einer alten Challenge, für die es aber von RSADSI kein Geld gibt.

Die zusammengesetzte Zahl „RSA-160“ lautet (die folgenden drei Zeilen sind eine einzige Zahl):

$$215274110271888970189601520131282542925777358884567598017049767677813314521885913567301105977349105960249790711158521430207931466520284014061994699492757040775 3.$$

Die Faktorisierung von RSA-160 ergab die beiden Primfaktoren:

$$p = 45427892858481394071686190649738831656137145778469793250959984709250004157335359$$

[76] `https://members.loria.fr/PZimmermann/records/rsa160`
`https://members.loria.fr/PZimmermann/records/factor.html`
`https://web.archive.org/web/20170518021747/http://www.crypto-world.com:80/FactorWorld.html`

[77] Das BSI erstellt jedes Jahr ein Papier über die Eignung von Kryptoalgorithmen, mit denen Signaturen erzeugt werden können, die den Vorgaben des deutschen Signaturgesetzes genügen. Bei dieser Erstellung werden Experten aus Wirtschaft und Wissenschaft beteiligt. Um die Eignung von Signaturverfahren zu beurteilen, deren Schwierigkeit auf dem Faktorisierungsproblem beruht, kooperiert das BSI auch mit Forschern der Universität Bonn.

und

$$q = 473880906038320161966338323037889519732689229210409579447413546488120284939093679$$

Die Berechnungen erfolgten zwischen Dezember 2002 und April 2003.

RSA-200

Am 9. Mai 2005 meldete die Forschergruppe von Prof. Jens Franke der Universität Bonn[78], dass sie gemeinsam mit Kollegen des Amsterdam Centrum voor Wiskunde en Informatica einen neuen Weltrekord im Faktorisieren aufstellten.

Sie zerlegten mit der GNFS-Methode (General Number Field Sieve) eine 200-stellige Zahl in ihre beiden Primfaktoren (diese haben jeweils 100 Dezimalstellen).

Die zusammengesetzte Zahl „RSA-200“ lautet (die folgenden drei Zeilen sind eine einzige Zahl):

$$27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983$$

Die Faktorisierung von RSA-200 ergab die beiden Primfaktoren:

$$p = 35324619344027701212726049781984643686711974001976250236493034687761212536794232000585479565280883349$$

und

$$q = 79258699544783330333470858414800596877379758573642199607343303414557678728181521353814093047401854677$$

Die Berechnungen erfolgten zwischen Dezember 2003 und Mai 2005. Die Faktorisierung durch die Gruppe um Bahr, Böhm, Franke, Kleinjung, Montgomery und te Riele hatte also knapp 17 Monate gedauert. Der Rechenaufwand lag bei umgerechnet etwa 120.000 MIPS-Jahren[79].

RSA-768

Am 12. Dezember 2009 meldete die Forschergruppe um Prof. Thorsten Kleinjung[80], dass sie eine 232-stellige Zahl in ihre beiden Primfaktoren zerlegten (diese haben jeweils 116 Dezimalstellen). Sie benutzten dazu die GNFS-Methode (General Number Field Sieve) in einer Art, dass vor dem Matrix-Schritt auf mehreren hundert Rechnern „Oversieving“ betrieben wurde.

Die zusammengesetzte Zahl „RSA-768“ lautet (die folgenden drei Zeilen sind eine einzige Zahl):

$$12301866845301177551304949583849627207728535695953347921973224521517264005072636575187452021997864693899564749427740638459251925573263034537315482685079170261221429134616704292143116022212404792747377940806653514195974598569021434136$$

[78] `https://members.loria.fr/PZimmermann/records/rsa200`

[79] Ein MIPS-Jahr (MY) ist die Anzahl von Operationen, die eine Maschine, welche eine Million Integeroperationen pro Sekunde (MIPS) ausführt, in einem Jahr bewältigt. Zur Illustration: ein INTEL Pentium 100 Prozessor hat etwa 50 MIPS. Für die Zerlegung eines 2048-Bit-Moduls bräuchte man ca. $8,5 \cdot 10^{40}$ MY.

[80] `http://eprint.iacr.org/2010/006.pdf` [Kle10]

Die Faktorisierung von RSA-768 ergab die beiden Primfaktoren (je 384 bit):

$$p = 33478071698956898786044169848212690817704794983713768568912431388982883793878002287614711652531743087737814467999489$$

und

$$q = 36746043666799590428244633799627952632279158164343087642676032283815739666511279233373417143396810270092798736308917$$

Die Berechnungen dauerten 2 1/2 Jahre.[81]

C307 / M1039

Im Mai 2007 meldeten Prof. Franke, Prof. Kleinjung (von der Universität Bonn), das japanische Telekommunikationsunternehmen NTT und Prof. Arjen Lenstra von der Polytechnischen Hochschule in Lausanne, dass sie mit der SNFS-Methode (Special Number Field Sieve) innerhalb von 11 Monaten eine 307-stellige Dezimalzahl in ihre beiden Primfaktoren zerlegten (diese haben 80 und 227 Dezimalstellen).

Die Aufgabe der Wissenschaftler entsprang nicht einer Challenge, sondern die Aufgabe war, die letzten Primfaktoren der Mersenne-Zahl $2^{1039} + 1$ zu finden (siehe „Wanted List“ des Cunningham-Projekts[82]).

Die Zahl $2^{1039} - 1$ besteht aus 3 Primfaktoren: Der kleinste Faktor $p7 = 5080711$ war schon länger bekannt.[83]

Zur vollständigen Faktorisierung musste der zweite Faktor (Koteiler) „C307“ zerlegt werden: Bis dahin wusste man nur, dass er zusammengesetzt ist, aber man kannte weder die Anzahl seiner Primfaktoren, noch die Primfaktoren selbst. Die folgenden fünf Zeilen sind eine einzige

[81] Dies war ein „academic effort“ – Organisationen mit besserer Ressourcen-Ausstattung könnten es deutlich schneller durchführen.

[82] Cunningham-Projekt: `http://homes.cerias.purdue.edu/~ssw/cun/`
Cunningham-Tabelle: `http://homes.cerias.purdue.edu/~ssw/cun/pmain1215`
Die Zahlen in der Cunningham-Tabelle werden folgendermaßen geschrieben:
„(2,n)-“ bedeutet $2^n - 1$; „(2,n)+“ bedeutet $2^n + 1$.
Um die Größenordnung einer Zerlegung anzudeuten schreibt man $p < n >$ oder $c < n >$, wobei „n“ die Anzahl der Dezimalstellen ist und „p“ und „c“ bedeuten, dass die Zahl eine Primzahl oder eine zusammengesetzte Zahl ist.
$2^{1039} - 1 = p7 * c307 = p7 * p80 * p227$
Genauer erklärt wird dies auf der Seite zum Cunningham-Projekt folgendermaßen:
„2,651+ means $2^{651} + 1$ and the size (c209 means 209 decimal digits) of the number which was factored. Then come the new factor(s), the discoverer and the method used. Recently, only the multiple polynomial quadratic sieve (ppmpqs), the elliptic curve method (ecm) and the number field sieve (nfs) have been used. ‘hmpqs’ stands for hypercube multiple polynomial quadratic sieve. Under ‘new factors’, ‘p90’ means a 90-digit prime and ‘c201’ is a 201-digit composite number.“.

[83] Er kann mit CT1 über das Menü **Einzelverfahren \ RSA-Kryptosystem \ Faktorisieren einer Zahl** gefunden werden — mit den Algorithmen von Brent, Williams oder Lenstra, mit denen man gut „relativ“ kleine Faktoren abspalten kann. Analog geht es in CT2 mit der Komponente GeneralFactorizer.

Zahl:

$$C307 = 115942057407257306436980714887689464075389979170201772498686835353882248385996675660800060954080051794720539932612302048744028604353028619141014409345351233471273967988850226307575280937916602855510550042581077117617761009413797078797380618700843777718682868088984471282200293520180607475545154137071102381 7$$

Die Faktorisierung von C307 ergab die beiden 80- und 227-stelligen Primfaktoren:

$$p80 = 55853666619936291260749204658315944968646527018488637648010052346319853288374753$$

und

$$p227 = 20758181946442382764570481370359469516293970800739520988120838703792729090324679382343143884144834882534053344769112223028158327696525376091410189105241993899334109711624358962065972167481161749004803659735573409253205425523689.$$

Damit wurde die Zahl $2^{1039} - 1$ vollständig in ihre 3 Primfaktoren zerlegt.

Verweise:

- `http://www.loria.fr/~zimmerma/records/21039-`
- `https://web.archive.org/web/20170518021747/http://www.crypto-world.com:80/announcements/m1039.txt`
- `https://web.archive.org/web/20170518024506/http://www.crypto-world.com:80/FactorAnnouncements.html`

Größenordnung faktorisierter Zahlen im Vergleich zu auf Primalität getesteter Zahlen

Wie man sieht, sind die größten (aus 2 Primfaktoren) zusammengesetzten Zahlen, die man faktorisieren kann, deutlich kleiner als die Zahlen mit einer speziellen Struktur, für die Primzahltests in der Lage sind, Aussagen über ihre Primalität zu treffen (siehe Kapitel 3.4, 3.5 und 3.6).

Länge der derzeitigen Weltrekorde in Dezimaldarstellung:

$$\begin{aligned} [RSA{-}768{-}Zahl] &\longleftrightarrow [49.\ bekannte\ Mersenne\ Primzahl] \\ 232 &\longleftrightarrow 22.338.618 \\ [vgl.\ Tab.\ 4.12] &\longleftrightarrow [vgl.\ Tab.\ 3.1] \end{aligned}$$

4.11.5 Weitere Forschungsergebnisse zu Primzahlen und Faktorisierung

Primzahlen sind Teil vieler hochaktueller Forschungsgebiete der Zahlentheorie. Die Fortschritte bei der Faktorisierung sind größer als noch in 2005 geschätzt – sie gehen nicht nur auf das Konto schnellerer Rechner, sondern sie sind auch in neuen mathematischen Erkenntnissen begründet. Der aktuelle Stand der Forschung wird dikutiert in Kapitel 10.

Die Sicherheit des RSA-Algorithmus basiert auf der empirischen Beobachtung, dass die Faktorisierung großer ganzer Zahlen ein schwieriges Problem ist. Besteht wie beim RSA-Algorithmus der zugrunde liegende Modul n aus dem Produkt zweier großer Primzahlen p, q (typische Längen: p, q $500 - 600$ bit, n 1024 bit), so lässt sich $n = pq$ aus p und q leicht bestimmen, jedoch ist es mit den bisher bekannten Faktorisierungsalgorithmen nicht möglich, umgekehrt p, q aus n zu gewinnen. Um jedoch den privaten aus dem öffentlichen Schlüssel zu ermitteln, bräuchte man entweder Kenntnis von p und q oder vom Wert der Eulerschen Phi-Funktion $\phi(n)$.

Die Entdeckung eines Algorithmus zur effizienten Faktorisierung von Produkten $n = pq$ großer Primzahlen würde daher den RSA-Algorithmus wesentlich beeinträchtigen. Je nach Effizienz der Faktorisierung im Vergleich zur Erzeugung von p, q, n müsste der verwendete Modul n (z.Zt. 1024 bit) erheblich vergrößert oder — im Extremfall — auf den Einsatz des RSA ganz verzichtet werden.

4.11.5.1 Das Papier von Bernstein und seine Auswirkungen auf die Sicherheit des RSA-Algorithmus

Die im November 2001 veröffentlichte Arbeit „Circuits for integer factorization: a proposal“ von D.J. Bernstein [Ber01] behandelt das Problem der Faktorisierung großer Zahlen. Die Kernaussage des Papers besteht darin, dass es möglich ist, die Implementierung des General Number Field Sieve-Algorithmus (GNFS) so zu verbessern, dass mit gleichem Aufwand wie bisher Zahlen mit 3-mal größerer Stellenzahl (Bit-Länge) faktorisiert werden können.

Wesentlich bei der Interpretation des Resultats ist die Definition des Aufwandes: Als Aufwand wird das Produkt von benötigter Rechenzeit und Kosten der Maschine (insbesondere des verwendeten Speicherplatzes) angesetzt. Zentral für das Ergebnis des Papiers ist die Beobachtung, dass ein wesentlicher Teil der Faktorisierung auf Sortierung zurückgeführt werden kann und mit dem Schimmlerschen Sortierschema ein Algorithmus zur Verfügung steht, der sich besonders gut für den Einsatz von Parallelrechnern eignet. Am Ende des Abschnittes 3 gibt Bernstein konkret an, dass die Verwendung von m^2 Parallelrechnern mit jeweils gleicher Menge an Speicherplatz mit Kosten in der Größenordnung von m^2 einhergeht — genau so wie ein einzelner Rechner mit m^2 Speicherzellen. Der Parallelrechner bewältigt die Sortierung von m^2 Zahlen jedoch (unter Verwendung der o. g. Sortierverfahrens) in Zeit proportional zu m, wohingegen der Einprozessorrechner Zeit proportional m^2 benötigt. Verringert man daher den verwendeten Speicherplatz und erhöht — bei insgesamt gleich bleibenden Kosten — die Anzahl der Prozessoren entsprechend, verringert sich die benötigte Zeit um die Größenordnung $1/m$. In Abschnitt 5 wird ferner angeführt, dass der massive Einsatz der parallelisierten Elliptic Curve-Methode von Lenstra die Kosten der Faktorisierung ebenfalls um eine Größenordnung verringert (ein Suchalgorithmus hat dann quadratische statt kubische Kosten). Alle Ergebnisse von Bernstein gelten nur asymptotisch für große Zahlen n. Leider liegen keine Abschätzungen über den Fehlerterm, d.h. die Abweichung der tatsächlichen Zeit von dem asymptotischen Wert, vor — ein Mangel, den auch Bernstein in seinem Papier erwähnt. Daher kann zur Zeit keine Aussage getroffen werden, ob die Kosten (im Sinne der Bernsteinschen Definition) bei der Faktorisierung z. Zt. verwendeter RSA-Zahlen ($1024-2048$ bit) bereits signifikant sinken würden.

Zusammenfassend lässt sich sagen, dass der Ansatz von Bernstein durchaus innovativ ist. Da die Verbesserung der Rechenzeit unter gleichbleibenden Kosten durch einen massiven Einsatz von Parallelrechnern erkauft wird, stellt sich die Frage nach der praktischen Relevanz. Auch wenn formal der Einsatz von einem Rechner über 1 sec und 1.000.000 Rechnern für je 1/1.000.000 sec dieselben Kosten erzeugen mag, ist die Parallelschaltung von 1.000.000 Rechnern praktisch nicht (oder nur unter immensen Fixkosten, insbesondere für die Vernetzung der Prozessoren) zu realisieren. Solche Fixkosten werden aber nicht in Ansatz gebracht. Die Verwendung verteilter Ansätze (distributed computing) über ein großes Netzwerk könnte einen Ausweg bieten. Auch hier müssten Zeiten und Kosten für Datenübertragung einkalkuliert werden.

Solange noch keine (kostengünstige) Hardware oder verteilte Ansätze entwickelt wurden, die auf dem Bernsteinschen Prinzip basieren, besteht noch keine akute Gefährdung des RSA. Es bleibt zu klären, ab welchen Größenordnungen von n die Asymptotik greift.

Arjen Lenstra, Adi Shamir et. al. haben das Bernstein-Paper analysiert [LSTT02]. Als Ergebnis kommen Sie zu einer Bitlängen-Verbesserung der Faktorisierung um den Faktor 1.17 (anstatt Faktor 3 wie von Bernstein erwartet).

Die Zusammenfassung ihres Papers „Analysis of Bernstein's Factorization Circuit“ lautet:

„... Bernstein proposed a circuit-based implementation of the matrix step of the number field sieve factorization algorithm. We show that under the non-standard cost function used in [1], these circuits indeed offer an asymptotic improvement over other methods but to a lesser degree than previously claimed: for a given cost, the new method can factor integers that are 1.17 times larger (rather than 3.01). We also propose an improved circuit design based on a new mesh routing algorithm, and show that for factorization of 1024-bit integers the matrix step can, under an optimistic assumption about the matrix size, be completed within a day by a device that costs a few thousand dollars. We conclude that from a practical standpoint, the security of RSA relies exclusively on the hardness of the relation collection step of the number field sieve.“

Auch RSA Security kommt in ihrer Analyse der Bernstein-Arbeit [Sec02] vom 8. April 2002 erwartungsgemäß zu dem Ergebnis, dass RSA weiterhin als ungebrochen betrachtet werden kann.

Die Diskussion ist weiterhin im Gang.

Zum Zeitpunkt der Erstellung dieses Absatzes (Juni 2002) war nichts darüber bekannt, inwieweit die im Bernstein-Papier vorgeschlagenen theoretischen Ansätze realisiert wurden oder wieweit die Finanzierung seines Forschungsprojektes ist.

4.11.5.2 Das TWIRL-Device

Im Januar 2003 veröffentlichten Adi Shamir und Eran Tromer vom Weizmann Institute of Science den vorläufigen Draft *„Factoring Large Numbers with the TWIRL Device“*, in dem deutliche Bedenken gegen RSA-Schlüssellängen unter 1024 begründet werden [ST03a].

Das Abstract fasst ihre Ergebnisse folgendermaßen zusammen: „The security of the RSA cryptosystem depends on the difficulty of factoring large integers. The best current factoring algorithm is the Number Field Sieve (NFS), and its most difficult part is the sieving step. In 1999 a large distributed computation involving thousands of workstations working for many months managed to factor a 512-bit RSA key, but 1024-bit keys were believed to be safe for the next 15-20 years. In this paper we describe a new hardware implementation of the NFS sieving step ... which is 3-4 orders of magnitude more cost effective than the best previously published

designs ... . Based on a detailed analysis of all the critical components (but without an actual implementation), we believe that the NFS sieving step for 1024-bit RSA keys can be completed in less than a year by a $10M device, and that the NFS sieving step for 512-bit RSA keys can be completed in less than ten minutes by a $10K device. Coupled with recent results about the difficulty of the NFS matrix step ... this raises some concerns about the security of these key sizes."

Eine ausführliche Fassung findet sich auch in dem Artikel der beiden Autoren in den RSA Laboratories CryptoBytes [ST03b].

Eine sehr gute Erläuterung, wie der Angriff mit dem Generalized Number Field Sieve (GNFS) funktioniert und welche Fortschritte sich ergaben, findet sich in dem 3-seitigen Artikel in der DuD-Ausgabe Juni/2003 [WLB03]. Beim GNFS können 2 grundlegende Schritte unterschieden werden: der Siebungsschritt (Relationen sammeln) und die Matrix-Reduktion. Auch wenn der Siebungsschritt hochgradig parallelisierbar ist, so dominiert er doch den Gesamtrechenaufwand. Bisher haben Shamir und Tromer noch kein TWIRL-Device gebaut, jedoch ist der dafür geschätzte Aufwand von 10 bis 50 Millionen Euro, um eine 1024-Bit Zahl in einem Jahr zu faktorisieren für Geheimdienste oder große kriminelle Organisationen keineswegs prohibitiv, denn die „Kosten für einen einzigen Spionagesatelliten schätzt man z.B. auf mehrere Milliarden USD". Die Autoren empfehlen deshalb konkret, möglichst rasch sensible, bisher benutzte RSA-, Diffie-Hellman- oder ElGamal-Schlüssel von bis zu 1024 Bit zu wechseln und Schlüssel von mindestens 2048 Bit Länge einzusetzen. Auch für die geplante TCPA/Palladium-Hardware werden 2048-Bit RSA-Schlüssel verwendet!

Damit erscheinen die aktuellen Empfehlungen des BSI, auf längere RSA-Schlüssellängen umzustellen, mehr als gerechtfertigt.

4.11.5.3 „Primes in P": Testen auf Primalität ist polynominal

Im August 2002 veröffentlichten die drei indischen Forscher M. Agrawal, N. Kayal und N. Saxena ihr Paper *„PRIMES in P"* über einen neuen Primzahltest-Algorithmus, genannt AKS [AKS02]. Sie entdeckten einen polynominalen deterministischen Algorithmus, um zu entscheiden, ob eine gegebene Zahl prim ist oder nicht.

Die Bedeutung dieser Entdeckung liegt darin, dass sie Zahlentheoretiker mit neuen Einsichten und Möglichkeiten für die weitere Forschung versorgt. Viele Menschen haben im Lauf der Jahrhunderte nach einem polynominalen Primzahltest gesucht, so dass dieses Ergebnis einen theoretischen Durchbruch darstellt. Es zeigt sich immer wieder, dass aus schon lange bekannten Fakten neue Ergebnisse generiert werden können.

Aber selbst die Autoren merken an, dass andere bekannte Algorithmen (z.B. ECPP) schneller sein können. Der neue Algorithmus funktioniert für alle positiven ganzen Zahlen. Dagegen verwendet das GIMPS-Projekt den Lucas-Lehmer-Primzahltest, der besondere Eigenschaften der Mersennezahlen ausnutzt. Dadurch ist der Lucas-Lehmer-Test viel schneller und erlaubt, Zahlen mit Millionen von Stellen zu testen, während generische Algorithmen auf Zahlen mit einigen tausend Stellen beschränkt sind. Nachteil der bisherigen schnellen Verfahren ist, dass sie probabilistisch sind, also ihr Ergebnis höchstwahrscheinlich, aber nicht ganz sicher ist.

Aktuelle Forschungsergebnisse dazu finden sich z.B. auf:

`http://www.mersenne.org/`
`http://fatphil.org/maths/AKS/` Originaltext in Englisch
`http://ls2-www.cs.uni-dortmund.de/lehre/winter200203/kt/material/primes.ps`
Gute Erläuterung in Deutsch von Thomas Hofmeister.

4.11.5.4 „Shared Primes“: Module mit gleichen Primfaktoren

Der RSA-Algorithmus basiert auf der angenommenen Schwierigkeit, große semi-prime natürliche Zahlen (Module) zu faktorisieren (Faktorisierungsproblem). Wie Lenstra et al [LHA+12] beschrieben, ist es aber möglich, aus einer gegebenen Menge von Modulen einige zu faktorisieren, sofern sie gemeinsame Primfaktoren (shared primes) aufweisen. In diesem Fall kann das Faktorisierungsproblem umgangen werden, indem man die – relativ einfach zu berechnenden – größten gemeinsamen Teiler (ggT) bestimmt. Andererseits ist es keine triviale Aufgabe, alle Shared Primes (gemeinsame Primfaktoren) effizient zu bestimmen und die zugehörigen Module zu faktorisieren, wenn die gegebene Menge von Moduln sehr groß ist (mehrere Millionen).

Die ggTs lassen sich nur nutzen, wenn die RSA-Schlüssel nicht zufällig genug erzeugt wurden. Zieht man die Wichtigkeit starker kryptographischer Schlüssel in Betracht, sollte man verifizieren, dass alle Schlüssel (wirklich) zufällig erzeugt wurden [ESS12].

Als Lenstra et al ihr Paper [LHA+12] im Februar 2012 veröffentlichten, veröffentlichten sie nicht den zugehörigen Source-Code. Kurz danach wurde der Source-Code eines ähnlichen Programms auf der CrypTool-Website[84] in Python und C++ veröffentlicht, und noch etwas später auf der Seite, die von [HDWH12][85] benutzt wurde. Der schnellste bisher bekannte Code ist von [HDWH12].

Diese Programme finden alle eventuell existierenden gemeinsamen Primfaktoren aus einer gegebenen Menge von Modulen, auch wenn diese Menge Millionen von Modulen enthält. Mit diesen Programmen können dann System-Administratoren ihre eigenen RSA-Schlüssel testen.

Der einfachste Ansatz, alle Primfaktoren zu finden (indem man jeden Modul mit jedem anderen vergleicht), hat eine Komplexität, die quadratisch mit der Anzahl der Modulen wächst.

Eine sehr effiziente Methode, die Bäume zum Vergleich aller ggT-Paare benutzt, basiert auf einer Publikation von Dan Bernstein aus 2005 [Ber05]. Bernstein benutzt eine Vorberechnung, in der das Produkt aller Module erzeugt wird. Das ist ein weiteres Beispiel dafür, wie hilfreich Vorberechnungen sein können, um kryptographische Systeme zu brechen (ein anderes berühmtes Beispiel sind die Rainbow-Tabellen, um die Ursprungswerte von Hashwertes zu finden [Oec03]).

Das folgende SageMath-Beispiel zeigt die sehr unterschiedlichen Laufzeiten für die Berechnung eines ggT und einer Faktorisierung. In dem Abschnitt nach diesem Beispiel wird der Kern der Methode erklärt, der in [HDWH12] benutzt wird: Die Benutzung von zwei Bäumen beschleunigt die Berechnung deutlich.

Das SageMath-Beispiel 4.1 zeigt, dass die folgenden Operationen sehr schnell sind: Multiplikation von Faktoren, Dividieren eines Moduls durch einen bekannten Faktor und Berechnen des ggT. Im Gegensatz dazu steigt die Rechendauer zur Faktorisierung von längeren Moduli stark an. Selbst die relativ kleinen Module in diesem Beispiel zeigen das: Der kleinere Modul (69 Dezimalstellen, 228 bit) brauchte 76 Sekunden, während der größere (72 Dezimalstellen, 239 bit) schon fast 217 Sekunden benötigte.

Hinzu kommt, dass die Operationen Multiplikation, Division und ggT große Laufzeit-Unterschiede aufweisen, wenn die Operanden von sehr unterschiedlicher Größe sind.

[84] `http://www.cryptool.org/de/ctp-dokumentation-de/361-ctp-paper-rsa-moduli`
[85] `https://www.factorable.net/`

SageMath-Beispiel 4.1 Vergleich der Laufzeit bei der Berechnung eines ggT und einer Faktorisierung

```
# Multiplication
sage: 359387570449582375738819989426877315343 9 * 841157474490478814886355678 01
3023015411226397451703825301689038596254920570677809482933310608176 39

sage: 359387570449582375738819989426877315343 9 * 162259276829213363391578010288127
583139672825572068433667900695808357466165186436234672858047078770918753


# Division
sage: time 3023015411226397451703825301689038596254920570677809482933310608176 39 /
          359387570449582375738819989426877315343 9
Wall time: 0.00 s
841157474490478814886355678 01

sage: time 583139672825572068433667900695808357466165186436234672858047078770918753 /
          359387570449582375738819989426877315343 9
Wall time: 0.00 s
162259276829213363391578010288127


# Calculate gcd
sage: time gcd (583139672825572068433667900695808357466165186436234672858047078770918753,
               3023015411226397451703825301689038596254920570677809482933310608176 39)
Wall time: 0.00 s
359387570449582375738819989426877315343 9


# Factorize
sage: time factor (583139672825572068433667900695808357466165186436234672858047078770918753)
Wall time: 217.08 s
162259276829213363391578010288127 * 359387570449582375738819989426877315343 9

sage: time factor (3023015411226397451703825301689038596254920570677809482933310608176 39)
Wall time: 76.85 s
841157474490478814886355678 01 * 359387570449582375738819989426877315343 9
```

Effiziente Berechnung aller ggT-Paare und Erläuterung der benutzten Formel zur Bestimmung der Shared Primes

Der hervorragende Artikel „Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices“ [HDWH12] erklärt, wie der Algorithmus die ggTs von allen RSA-Moduln effizient berechnet.

Zuerst wird das Produkt P aller Moduln m_i mit Hilfe eines Produktbaumes berechnet, anschließend ein Restebaum modulo den Quadraten der Module. Dann werden alle ggTs aus einem Modul m_i und dem zugehörigen Rest z_i dividiert durch diesen Modul berechnet.

Dies wird in Abbildung 4.3 veranschaulicht – sie ist eine Kopie aus [HDWH12] (nur dass dort die Module N_i statt m_i genannt werden):

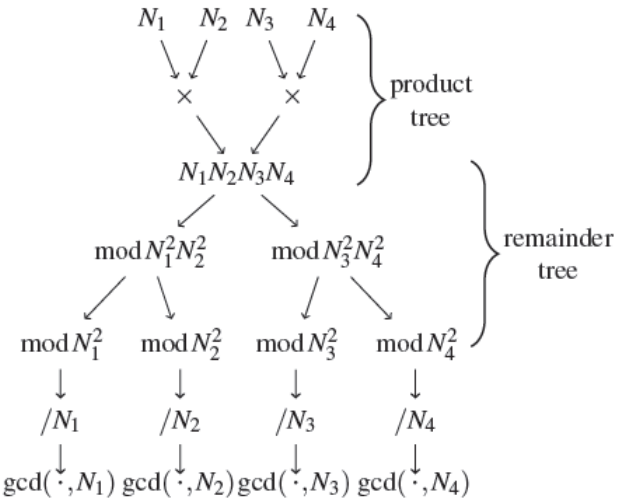


Figure 1: **Computing all-pairs GCDs efficiently**

Algorithm 1 Quasilinear GCD finding

Input: $N_1, \ldots, N_m$ RSA moduli

1: Compute $P = \prod N_i$ using a product tree.
2: Compute $z_i = (P \bmod N_i^2)$ for all i using a remainder tree.

Output: $\gcd(N_i, z_i/N_i)$ for all i.

Abbildung 4.3: Algorithmus und Abbildung, um alle ggTs effizient zu berechnen

Der Artikel [HDWH12] erklärt sehr gut *wie* der Algorithmus funktioniert, aber nicht so gut *warum*. Das Produkt P aller Module ist eine sehr große Zahl, sogar wenn man sie mit den einzelnen Modulen vergleicht. Ohne die Vereinfachungen durch den Restebaum würde man folgendermaßen vorgehen: Berechnen von $ggT_i = ggT(P/m_i, m_i)$ für alle i. Vergleichen jedes $ggT_i \neq 1$ mit allen anderen $ggT_j \neq 1$ mit $j \neq i$. Wenn zwei ggTs gleich sind, haben ihre Module einen gemeinsamen Faktor.[86]

[86] Eine Voraussetzung dafür, dass man nur Primfaktoren erhält, ist, dass doppelte Module entfernt werden, bevor

Weil diese Berechnungen bei den großen Längenunterschieden der Zahlen sehr langsam sind, wird der Restebaum benutzt. Obwohl es so aussieht, als benötige er mehr Schritte, bedeutet er eine große Vereinfachung.

Innerhalb des Restebaums erhält man – am Ende – $(P \bmod (m_i^2))/m_i$ für alle i.[87]

Die wesentliche offene Frage ist noch: Warum liefert $ggT((P \bmod m_i^2)/m_i, m_i)$ dasselbe Ergebnis wie $ggT(P/m_i, m_i)$? Wir beweisen, dass diese Identität korrekt ist.[88]

$$ggT((P \bmod m_i^2)/m_i, m_i) \quad \stackrel{!}{=} \quad ggT(P/m_i, m_i)$$

$\Longleftrightarrow$ [89]

$$ggT(((P \bmod m_i^2)/m_i) \bmod m_i, m_i) \quad \stackrel{!}{=} \quad ggT((P/m_i) \bmod m_i, m_i)$$

$\Longleftrightarrow$ [90]

$$((P \bmod m_i^2)/m_i) \bmod m_i \quad \stackrel{!}{=} \quad (P/m_i) \bmod m_i$$

$\Longleftrightarrow$ [91]

$$(P \bmod m_i^2)/m_i - P/m_i \equiv 0 \bmod m_i \quad \Leftrightarrow \quad m_i \mid ((P \bmod m_i^2)/m_i - P/m_i)$$

[92]

$$m_i \mid ((P - m_i^2 * \lfloor P/m_i^2 \rfloor - P))/m_i$$

[93]

$$m_i \mid (m_i * \lfloor P/m_i^2 \rfloor)$$

Weil dies offensichtlich wahr ist, können wir schließen, dass die zwei ggTs äquivalent sind.

---

man den Baum aufsetzt.

[87] Es würde keinen Sinn machen, beim linken ggT modulo m_i statt m_i^2 zu rechnen, denn das bedeutete, $(P \bmod m_i)/m_i$ zu benutzen: Denn $m_i|P$, so dass P/m_i immer eine ganze Zahl ist, so dass $(P \bmod m_i)$ immer $= 0$ ist.
Beispiel mit sehr kleinen Modulen:
$m_1 = 2 * 3 = 6; \quad m_2 = 2 * 7 = 14; \quad P = 6 * 14 = 84$
$P \bmod m_1 = 84 \bmod 6 = 0; \quad P \bmod m_1^2 = 84 \bmod 36 = 12$
$P \bmod m_2 = 84 \bmod 14 = 0; \quad P \bmod m_2^2 = 84 \bmod 196 = 84$
$ggT_1 = ggT(12/6, 6) = ggT(2, 6) = 2$
$ggT_2 = ggT(84/14, 14) = ggT(6, 14) = 2$
So wie der Baum strukturiert ist, würde es auch keinen Sinn machen, zuerst zu teilen und dann die Modulo-Operation vorzunehmen, denn das würde zu den gegebenen Modulen in umgekehrter Reihenfolge führen.
Ebenfalls keinen Sinn würde es machen, $(P \bmod (m_i^3))/m_i^2)$ zu berechnen, da dies nur zusätzlichen Aufwand ohne Verbesserung bedeutet.

[88] P ist das Produkt aller Module, und m_i ist irgendein Modul.

[89] Gemäß Euklids Algorithmus (erste Iteration) ist die folgende Identität wahr:
$ggT(a, b) = ggT(b, a \bmod b)$ wenn $b \neq 0$
Dies gilt, weil per Definition gilt: $ggT(a, 0) = a$
Angewandt auf unsere Problem bedeutet das:
$ggT((P \bmod m_i^2)/m_i, m_i) = ggT((P \bmod m_i^2)/m_i \bmod m_i, m_i)$
$ggT(P/m_i, m_i) = ggT(P/m_i \bmod m_i, m_i)$

[90] Die ggTs sind gleich, wenn ihre beiden ersten Argumente gleich sind.

[91] Die folgenden Umformungen sind Äquivalenzumformungen.

[92] Benutzt man die Modulo-Operation (Definition 4.4.2 von Seite 122) und eine Division, ergibt sich: $a \bmod b = a - b * \lfloor a/b \rfloor$. So kann $P \bmod m_i^2$ geschrieben werden als $P - m_i^2 \lfloor P/m_i^2 \rfloor$.

[93] P reduziert sich selbst, der Exponent von m_i im Zähler kürzt sich mit dem m_i im Nenner.

Viel mehr als unsere Fähigkeiten sind es unsere Entscheidungen ..., die zeigen, wer wir wirklich sind.

Zitat 12: Joanne K. Rowling[94]

4.12 Anwendungen asymmetrischer Kryptographie mit Zahlenbeispielen

In der modernen Kryptographie werden die Ergebnisse der modularen Arithmetik extensiv angewandt. Hier werden exemplarisch einige wenige Beispiele aus der Kryptographie mit kleinen[95] Zahlen vorgestellt.[96]

Die Chiffrierung eines Textes besteht darin, dass man aus einer Zeichenkette (Zahl) durch Anwenden einer Funktion (mathematische Operationen) eine andere Zahl erzeugt. Dechiffrieren heißt, diese Funktion umzukehren: aus dem Zerrbild, das die Funktion aus dem Klartext gemacht hat, das Urbild wiederherzustellen. Beispielsweise könnte der Absender einer vertraulichen Nachricht zum Klartext M eine geheimzuhaltende Zahl, den Schlüssel S, addieren und dadurch den Chiffretext C erhalten:

$$C = M + S.$$

Durch Umkehren dieser Operation, das heißt durch Subtrahieren von S, kann der Empfänger den Klartext rekonstruieren:

$$M = C - S.$$

Das Addieren von S macht den Klartext zuverlässig unkenntlich. Gleichwohl ist diese Verschlüsselung sehr schwach; denn wenn ein Abhörer auch nur ein zusammengehöriges Paar von Klar- und Chiffretext in die Hände bekommt, kann er den Schlüssel berechnen

$$S = C - M,$$

und alle folgenden mit S verschlüsselten Nachrichten mitlesen.
Der wesentliche Grund ist, dass Subtrahieren eine ebenso einfache Operation ist wie Addieren.

4.12.1 Einwegfunktionen

Wenn der Schlüssel auch bei gleichzeitiger Kenntnis von Klar- und Chiffretext nicht ermittelbar sein soll, braucht man eine Funktion, die einerseits relativ einfach berechenbar ist - man will ja chiffrieren können. Andererseits soll ihre Umkehrung zwar existieren (sonst würde beim Chiffrieren Information verlorengehen), aber de facto unberechenbar sein.

Was sind denkbare Kandidaten für eine solche **Einwegfunktion**? Man könnte an die Stelle der Addition die Multiplikation setzen; aber schon Grundschüler wissen, dass deren Umkehrung, die Division, nur geringfügig mühsamer ist als die Multiplikation selbst. Man muss noch eine Stufe höher in der Hierarchie der Rechenarten gehen: Potenzieren ist immer noch eine relativ

[94] Joanne K. Rowling, „Harry Potter und die Kammer des Schreckens“, Carlsen, 1998, letztes Kapitel „Dobbys Belohnung“, S. 343, Dumbledore.

[95] „Klein“ bedeutet beim RSA-Verfahren, dass die Bitlängen der Zahlen sehr viel kleiner sind als 1024 Bit (das sind 308 Dezimalstellen). 1024 Bit gilt im Moment in der Praxis als Mindestlänge für sichere RSA-Module.

[96] Didaktisch sehr gut aufbereitete Artikel mit Programmbeispielen in Python und SageMath finden sich in der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*. Siehe beispielsweise [WSE15].

einfache Operation; aber ihre beiden Umkehrungen *Wurzelziehen* (finde b in der Gleichung $a = b^c$, wenn a und c bekannt sind) und *Logarithmieren* (in derselben Gleichung finde c, wenn a und b bekannt sind) sind so kompliziert, dass ihre Ausführung in der Schule normalerweise nicht mehr gelehrt wird.

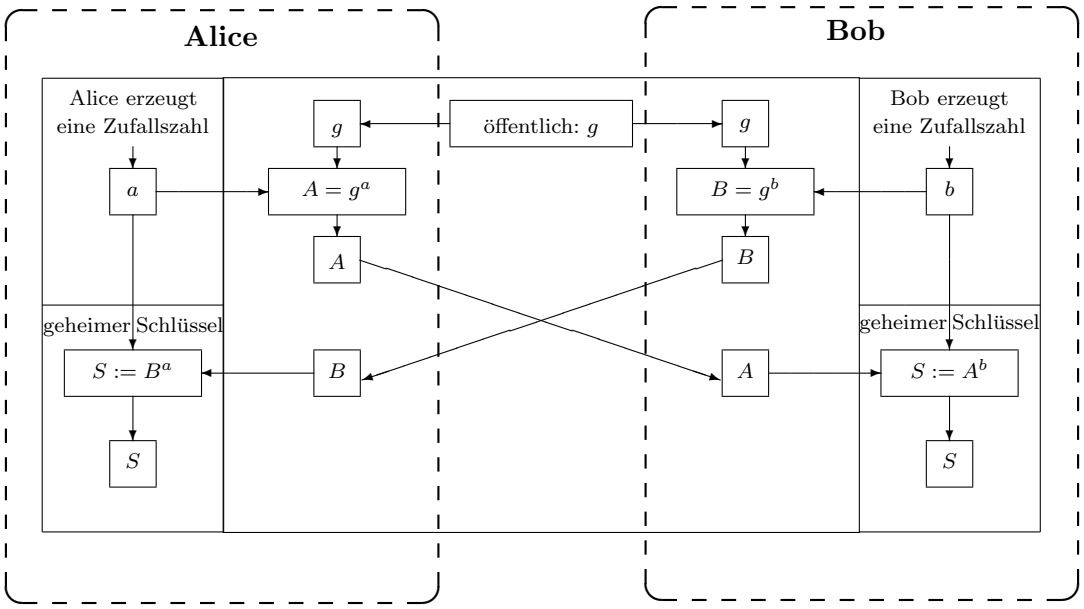
Während bei Addition und Multiplikation noch eine gewisse Struktur wiedererkennbar ist, wirbeln Potenzierung und Exponentiation alle Zahlen wild durcheinander: Wenn man einige wenige Funktionswerte kennt, weiß man (anders als bei Addition und Multiplikation) noch kaum etwas über die Funktion im ganzen.

4.12.2 Das Diffie-Hellman Schlüsselaustausch-Protokoll

Das DH-Schlüsselaustauschprotokoll (Key Exchange Protocol) wurde 1976 in Stanford von Whitfield Diffie, Martin E. Hellman und Ralph Merkle erdacht.[97]

Eine Einwegfunktion dient Alice und Bob[98] dazu, sich einen Schlüssel S, den Sessionkey, für die nachfolgende Verständigung zu verschaffen. Dieser ist dann ein Geheimnis, das nur diesen beiden bekannt ist. Alice wählt sich eine Zufallszahl a und hält sie geheim. Aus a berechnet sie mit der Einwegfunktion die Zahl $A = g^a$ und schickt sie an Bob. Der verfährt ebenso, indem er eine geheime Zufallszahl b wählt, daraus $B = g^b$ berechnet und an Alice schickt. Die Zahl g ist beliebig und darf öffentlich bekannt sein. Alice wendet die Einwegfunktion mit ihrer Geheimzahl a auf B an, Bob tut gleiches mit seiner Geheimzahl b und der empfangenen Zahl A.

Das Ergebnis S ist in beiden Fällen dasselbe, weil die Einwegfunktion kommutativ ist: $g^{a*b} = g^{b*a}$. Aber selbst Bob kann Alices Geheimnis a nicht aus den ihm vorliegenden Daten rekonstruieren, Alice wiederum Bobs Geheimnis b nicht ermitteln, und ein Lauscher, der g kennt und sowohl A als auch B mitgelesen hat, vermag daraus weder a noch b noch S zu berechnen.



[97] In CT1 ist dieses Austauschprotokoll visualisiert: Sie können die einzelnen Schritte mit konkreten Zahlen nachvollziehen per Menü **Einzelverfahren \ Protokolle \ Diffie-Hellman-Demo**.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Visualisierungen \ Diffie-Hellman Schlüsselaustausch (EC)**.

[98] Alice und Bob werden standardmäßig als die beiden berechtigten Teilnehmer eines Protokolls bezeichnet (siehe [Sch96, Seite 23]).

Ablauf:

Alice und Bob wollen also einen geheimen Sessionkey S über einen abhörbaren Kanal aushandeln.

1. Sie wählen eine Primzahl p und eine Zufallszahl g, und tauschen diese Information offen aus.
2. Alice wählt nun a, eine Zufallszahl kleiner p und hält diese geheim.

   Bob wählt ebenso b, eine Zufallszahl kleiner p und hält diese geheim.
3. Alice berechnet nun $A \equiv g^a \pmod{p}$.
   Bob berechnet $B \equiv g^b \pmod{p}$.
4. Alice sendet das Ergebnis A an Bob.
   Bob sendet das Ergebnis B an Alice.
5. Um den nun gemeinsam zu benutzenden Sessionkey zu bestimmen, potenzieren sie beide jeweils für sich das jeweils empfangene Ergebnis mit ihrer geheimen Zufallszahl modulo p. Das heißt:
   - Alice berechnet $S \equiv B^a \pmod{p}$, und
   - Bob berechnet $S \equiv A^b \pmod{p}$.

Auch wenn ein Spion g, p, und die Zwischenergebnisse A und B abhört, kann er den schließlich bestimmten Sessionkey nicht berechnen – wegen der Schwierigkeit, den diskreten Logarithmus[99] zu bestimmen.

Das Ganze soll an einem Beispiel mit (unrealistisch) kleinen Zahlen gezeigt werden.

Beispiel mit kleinen Zahlen:

1. Alice und Bob wählen $g = 11$, $p = 347$.
2. Alice wählt $a = 240$, Bob wählt $b = 39$ und behalten a und b geheim.
3. Alice berechnet $A \equiv g^a \equiv 11^{240} \equiv 49 \pmod{347}$.
   Bob berechnet $B \equiv g^b \equiv 11^{39} \equiv 285 \pmod{347}$.
4. Alice sendet Bob: $A = 49$,
   Bob sendet Alice: $B = 285$.
5. Alice berechnet $B^a \equiv 285^{240} \equiv 268 \pmod{347}$,
   Bob berechnet $A^b \equiv 49^{39} \equiv 268 \pmod{347}$.

Nun können Alice und Bob mit Hilfe ihres gemeinsamen Sessionkeys S sicher kommunizieren. Auch wenn ein Spion alles abhörte, was über die Leitung ging: $g = 11, p = 347, A = 49$ und $B = 285$; den geheimen Schlüssel S kann er nicht berechnen.

Dies ist aber nur für sehr große Zahlen so, weil der diskrete Logarithmus nur dann extrem schwierig gefunden werden kann (siehe Kapitel 10).

[99] Weitere Details zum Diskreten Logarithmusproblem finden Sie in den Kapiteln 5.4 und 10.

Bemerkung:

Für solch kleine Zahlen wie in dem obigen Beispiel ist das DH-Schlüsselaustausch-Protokoll angreifbar, da das diskrete Logarithmusproblem leicht berechnet werden kann, um die Exponenten a oder b zu finden.

Wenn man a oder b hat, kann man S genauso berechnen wie es Alice or Bob vorher machten.

Um die diskreten Logarithen zu erhalten, ist hier Folgendes zu berechnen:

a von Alice: $11^x \equiv 49 \pmod{347}$, also $\log_{11}(49) \pmod{347}$.

b von Bob: $11^y \equiv 285 \pmod{347}$, also $\log_{11}(285) \pmod{347}$.

Mit SageMath kann man den diskreten Logarithmus x, der die obige Gleichung (z.B. für Allice) löst:

SageMath-Beispiel 4.2 Beispiel mit kleinen Zahlen: Berechnen der diskreten Logs a und b, um DH anzugreifen

```
# Get the secret key of Alice:
### via numbers
sage: discrete_log(mod(49,347),mod(11,347))
67
### via variables in the ring of integers
sage: R=Integers(347)
sage: g=R(11)
sage: A=R(49)
sage: discrete_log(A,g)
67

# Get the secret key of Bob:
sage: B=R(285)
sage: discrete_log(B,g)
39
```

Da die SageMath-Funktion discrete_log als Argumente nur Elemente eines Rings erwartet (Integer zwischen 0 und einer Obergrenze), können wir diesen Typ einerseits erzwingen, indem wir die Zahlen direkt mit dem zugehörigen Modulo-Operator eingeben: `discrete_log(mod(49, 347), mod(11, 347))`.
Eine viel bessere Alternative ist es, die Variablen so wie in der obigen Formel zu benutzen und SageMath wissen zu lassen, dass es sich um Elements eines Rings handelt. Nach diesem „Mehraufwand“ zu Beginn für die Initialisierung, kann man die Formeln genauso niederschreiben, wie man es gewohnt ist: `discrete_log(A, g)`.[100,101]

[100] Solche zahlentheoretischen Aufgaben können auch mit anderen Tools wie PariGP, LiDIA, BC oder Mathematica (siehe Anhang Web-Links am Ende dieses Kapitel) gelöst werden: Hier die entsprechende Syntax, um den diskreten Log für Alice zu bekommen:

- Pari-GP: `znlog(Mod(49,347),Mod(11,347))`
- LiDIA: `dl(11,49,347)`
- Mathematica: `MultiplicativeOrder[11, 347, 49]`
  Die allgemeine Funktion „Solve“ liefert die „em tdep message“: The equations appear to involve the variables to be solved for in an essentially non-algebraic way.

Alle liefern das Ergebnis 67.

[101] Warum haben die Funktionen für den diskreten Logarithmus für Alice den Wert 67 geliefert und nicht den Wert 240, den Alice als Exponent a wählte?
Der diskrete Logarithmus ist der kleinste natürliche Exponent, der die Gleichung $11^x \equiv 49 \pmod{347}$ löst. Sowohl $x = 67$ als auch $x = 240$ (die im Beispiel gewählte Zahl) erfüllen die Gleichung und können damit zur Berechnung des Sessionkeys benutzt werden: $285^{240} \equiv 285^{67} \equiv 268 \pmod{347}$. Hätten Alice und Bob als Basis g eine Primitivwurzel modulo p gewählt, dann gibt es für jeden Rest aus der Menge $\{1, 2, \cdots, p-1\}$ genau einen

4.13 Das RSA-Verfahren mit konkreten Zahlen[102]

„Das Spiel ist eine Erfindung der Natur, um uns auf schwierige Realitäten vorzubereiten. Sind Sie jetzt endlich bereit, der Realität ins Auge zu sehen, Sergeant?“

Zitat 13: Daniel Suarez[103]

Nachdem oben die Funktionsweise des RSA-Verfahrens beschrieben wurde, sollen diese Schritte hier mit konkreten, aber kleinen Zahlen durchgeführt werden.

4.13.1 RSA mit kleinen Primzahlen und mit einer Zahl als Nachricht

Bevor wir RSA auf einen Text anwenden, wollen wir es erst direkt mit einer Zahl[104] zeigen.[105]

1. Die gewählten Primzahlen seien $p = 5$ und $q = 11$.
   Also ist $n = 55$ und $\phi(n) = (p-1) * (q-1) = 40$.
2. $e = 7$ (e sollte[106] zwischen 11 und 39 liegen, und muss teilerfremd zu 40 sein).
3. $d = 23$ (da $23 * 7 \equiv 161 \equiv 1 \pmod{40}$)

   $\rightarrow$ Public-Key des Empfängers: $(55, 7)$,
   $\rightarrow$ Private-Key des Empfängers: $(55, 23)$.
4. Nachricht sei nur die Zahl $M = 2$ (also ist kein Aufbrechen in Blöcke nötig).
5. Verschlüsseln: $C \equiv 2^7 \equiv 18 \pmod{55}$.
6. Chiffrat ist nur die Zahl $C = 18$ (also kein Aufbrechen in Blöcke nötig).

Exponenten aus der Menge $\{0, 1, \cdots, p-2\}$.

Info: Zum Modul 347 gibt es 172 verschiedene Primitivwurzeln, davon sind 32 prim (ist nicht notwendig). Da die im Beispiel für g gewählte Zahl 11 keine Primitivwurzel von 347 ist, nehmen die Reste nicht alle Werte aus der Menge $\{1, 2, \cdots, 346\}$ an. Somit kann es für einen bestimmten Rest mehr als einen oder auch gar keinen Exponenten aus der Menge $\{0, 1, \cdots, 345\}$ geben, der die Gleichung erfüllt.
Mit den entsprechenden SageMath-Funktionen findet man:
`is_prime(347)=True`, `euler_phi(347)=346`, `gcd(11,347)=1` und `multiplicative_order(mod(11, 347))=173`.

i	$11^i \bmod 347$	
0	1	
1	11	
2	121	
3	290	
67	49	gesuchter Exponent
172	284	
173	1	= Multiplikative Ordnung von 11^i (mod347)
174	11	
175	121	
176	290	
240	49	gesuchter Exponent

Weitere Details finden Sie in Kapitel 4.19.4 „Primitivwurzeln“.

[102] Nguyen schrieb einen kurzen und didaktisch sehr klaren Artikel zu einigen Grundlagen der Zahlentheorie und zur Benutzung von SageMath [Ngu09].

[103] Daniel Suarez, „Daemon“, rororo, (c) 2010, Kapitel 45, „Wiedereintritt“, S. 615, Sobol.

[104] In der Praxis wird RSA nie auf Texte angewandt, sondern nur auf große Zahlen.

[105] Mit CT1 können Sie dies per Menü **Einzelverfahren \ RSA-Kryptosystem \ RSA-Demo** lösen.

[106] Siehe Fußnote 60 auf Seite 149.

7. Entschlüsseln: $M \equiv 18^{23} \equiv 18^{(1+2+4+16)} \equiv 18 * 49 * 36 * 26 \equiv 2 \pmod{55}$.

Nun wollen wir RSA auf einen Text anwenden: zuerst mit dem Großbuchstabenalphabet (26 Zeichen), dann mit dem gesamten ASCII-Zeichensatz als Bausteine für die Nachrichten.

4.13.2 RSA mit etwas größeren Primzahlen und einem Text aus Großbuchstaben

Gegeben ist der Text „ATTACK AT DAWN“, und die Zeichen werden gemäß Tabelle 4.13 codiert.[107]

Zeichen	Zahlenwert	Zeichen	Zahlenwert
Blank	0	M	13
A	1	N	14
B	2	O	15
C	3	P	16
D	4	Q	17
E	5	R	18
F	6	S	19
G	7	T	20
H	8	U	21
I	9	V	22
J	10	W	23
K	11	X	24
L	12	Y	25
		Z	26

Tabelle 4.13: Großbuchstabenalphabet

Schlüsselerzeugung (Schritt 1 bis 3):
1. $p = 47, q = 79$ ($n = 3713$; $\phi(n) = (p-1) * (q-1) = 3588$).
2. $e = 37$ (e sollte[108] zwischen 79 und 3587 liegen, und muss teilerfremd zu 3588 sein).
3. $d = 97$ (denn $e * d = 1 \bmod \phi(n); 37 * 97 \equiv 3589 \equiv 1 \pmod{3588}$).[109]

4. Verschlüsselung:

Text:	A	T	T	A	C	K		A	T		D	A	W	N
Zahl:	01	20	20	01	03	11	00	01	20	00	04	01	23	14

Aufteilung dieser 28-stelligen Zahl in 4-stellige Teile (denn 2626 ist noch kleiner als $n = 3713$), d.h. dass die Blocklänge 2 beträgt.

```
0120 2001 0311 0001 2000 0401 2314
```

Verschlüsselung aller 7 Teile jeweils per: $C \equiv M^{37} \pmod{3713}$:[110]

```
1404 2932 3536 0001 3284 2280 2235
```

[107] Mit CT1 können Sie dies per Menü **Einzelverfahren \ RSA-Kryptosystem \ RSA-Demo** lösen. Dies ist auch im Tutorial/Szenario der Online-Hilfe zu CT1 beschrieben [Optionen, Alphabet vorgeben, Basissystem, Blocklänge 2 und Dezimaldarstellung].
[108] Siehe Fußnote 60 auf Seite 149.
[109] Wie man $d = 97$ mit Hilfe des erweiterten ggT berechnet, wird in Anhang 4.14 gezeigt.
[110] In Kapitel 4.19.5 „RSA-Beispiele mit SageMath“ finden Sie den Beispiel-Quelltext zur RSA-Verschlüsselung mit SageMath.

5. Entschlüsselung:
Chiffrat: `1404 2932 3536 0001 3284 2280 2235`

Aufteilung dieser 28-stelligen Zahl in 4-stellige Teile.

Entschlüsselung aller 7 Teile jeweils per: $M \equiv C^{97} \pmod{3713}$:
`0120 2001 0311 0001 2000 0401 2314`

Umwandeln von 2-stelligen Zahlen in Großbuchstaben und Blanks.

Bei den gewählten Werten ist es für einen Kryptoanalytiker einfach, aus den öffentlichen Parametern $n = 3713$ und $e = 37$ die geheimen Werte zu finden, indem er offenlegt, dass $3713 = 47 * 79$.

Wenn n eine 1024-Bit-Zahl ist, bestehen dafür – nach heutigen Kenntnissen – wenig Chancen.

4.13.3 RSA mit noch etwas größeren Primzahlen und mit einem Text aus ASCII-Zeichen

Real wird das ASCII-Alphabet benutzt, um die Einzelzeichen der Nachricht in 8-Bit lange Zahlen zu codieren.

Diese Aufgabe[111] ist angeregt durch das Beispiel aus [Eck14, S. 271].

Der Text „RSA works!“ bedeutet in Dezimalschreibweise codiert:

Text:	R	S	A		w	o	r	k	s	!
Zahl:	82	83	65	32	119	111	114	107	115	33

Das Beispiel wird in 2 Varianten durchgespielt. Gemeinsam für beide sind die Schritte 1 bis 3.

Schlüsselerzeugung (Schritt 1 bis 3):
1. $p = 509,\ q = 503 \quad (n = 256.027;\ \ \phi(n) = (p-1) * (q-1) = 255.016 = 2^3 * 127 * 251)$.[112]
2. $e = 65.537$ (e sollte[113] zwischen 509 und 255.015 liegen, u. muss[114] teilerfremd zu 255.016 sein).
3. $d = 231.953$
(denn $e \equiv d^{-1} \bmod \phi(n)$; $\quad 65.537 * 231.953 \equiv 15.201.503.761 \equiv 1 \pmod{255.016}$).[115]

Variante 1:
ASCII-Zeichen werden einzeln ver- und entschlüsselt (keine Blockbildung).

4. Verschlüsselung:

Text:	R	S	A		w	o	r	k	s	!
Zahl:	82	83	65	32	119	111	114	107	115	33

Keine Zusammenfassung der Buchstaben![116]

---

[111] Mit CT1 können Sie diese Aufgabe per Menü **Einzelverfahren \ RSA-Kryptosystem \ RSA-Demo** lösen. Mit JCT können Sie diese Aufgabe im Menü **Visualisierungen \ RSA-Kryptosystem** der Standard-Perspektive lösen.

[112] In Kapitel 4.19.5 „RSA-Beispiele mit SageMath“ finden Sie den Quelltext zur Faktorisierung von $\phi(n)$ mit SageMath.

[113] Siehe Fußnote 60 auf Seite 149.

[114] e darf also nicht $2, 127$ oder 251 sein $(65537 = 2^{16} + 1)\ (255,016 = 2^3 * 127 * 251)$.
Real wird $\phi(n)$ nicht faktorisiert, sondern für das gewählte e wird mit dem Euklidschen Algorithmus sichergestellt, dass $\text{ggT}(e, \phi(n)) = 1$.

[115] Andere mögliche Kombinationen von (e,d) sind z.B.: (3, 170.011), (5, 204.013), (7, 36.431).

[116] Für sichere Verfahren braucht man große Zahlen, die möglichst alle Werte bis $n-1$ annehmen. Wenn die mögliche Wertemenge der Zahlen in der Nachricht zu klein ist, nutzen auch große Primzahlen nichts für die Sicherheit. Ein ASCII-Zeichen ist durch 8 Bits repräsentiert. Will man größere Werte, muss man mehrere Zeichen zusammen-

Verschlüsselung pro Zeichen per: $C \equiv M^{65.537} \pmod{256.027}$:[117]

```
212984   025546   104529   031692   248407
100412   054196   100184   058179   227433
```

5. Entschlüsselung:

Chiffrat:

```
212984   025546   104529   031692   248407
100412   054196   100184   058179   227433
```

Entschlüsselung pro Zeichen per: $M \equiv C^{231.953} \bmod 256.027$:

```
82 83 65 32 119 111 114 107 115 33
```

Variante 2:
Jeweils zwei ASCII-Zeichen werden als Block ver- und entschlüsselt.

Bei der Variante 2 wird die Blockbildung in zwei verschiedenen Untervarianten 4./5. und 4'./5'. dargestellt.

Text:	R	S	A		w	o	r	k	s	!
Zahl:	82	83	65	32	119	111	114	107	115	33

4. Verschlüsselung:

Blockbildung[118] (je zwei ASCII-Zeichen werden als je 8-stellige Binärzahlen hintereinander geschrieben):

```
21075 16672 30575 29291 29473
```
[119]

Verschlüsselung pro Block per: $C \equiv M^{65.537} \pmod{256027}$:[120]

```
158721 137346 37358 240130 112898
```

5. Entschlüsselung:

Chiffrat:

```
158721 137346 37358 240130 112898
```

Entschlüsselung pro Block per: $M \equiv C^{231.953} \pmod{256.027}$:

```
21075 16672 30575 29291 29473
```

---

fassen. Zwei Zeichen benötigen 16 Bit, womit maximal der Wert 65.536 darstellbar ist; dann muss der Modul n größer sein als $2^{16} = 65.536$. Dies wird in Variante 2 angewandt. Beim Zusammenfassen bleiben in der Binär-Schreibweise die führenden Nullen erhalten (genauso wie wenn man oben in der Dezimalschreibweise alle Zahlen 3-stellig schreiben würde und dann die Folge `082 083, 065 032, 119 111, 114 107, 115 033` hätte).

[117] Kapitel 4.19.5 „RSA-Beispiele mit SageMath“ enthält den Quelltext zur RSA-Exponentiation mit SageMath.

[118] Blockbildung:

```
Einzelzeichen   Binärdarstellung    Dezimaldarstellung
01010010, 82    01010010 01010011 = 21075
01010011, 83
01000001, 65    01000001 00100000 = 16672
00100000, 32
01110111, 119   01110111 01101111 = 30575
01101111, 111
01110010, 114   01110010 01101011 = 29291
01101011, 107
01110011, 115   01110011 00100001 = 29473
00100001, 33:
```

[119] Mit CT1 können Sie dies per Menü **Einzelverfahren \ RSA-Kryptosystem \ RSA-Demo** mit den folgenden Optionen lösen: alle 256 Zeichen, b-adisch, Blocklänge 2, dezimale Darstellung.

[120] In Kapitel 4.19.5 „RSA-Beispiele mit SageMath“ finden Sie den Quelltext zur RSA-Exponentiation mit SageMath.

4'. Verschlüsselung:
Blockbildung (die ASCII-Zeichen werden als 3-stellige Dezimalzahlen hintereinander geschrieben):
`82083 65032 119111 114107 115033`[121]

Verschlüsselung pro Block per: $C \equiv M^{65537} \pmod{256.027}$:[122]
`198967 051405 254571 115318 014251`

5'. Entschlüsselung:
Chiffrat:
`198967 051405 254571 115318 014251`

Entschlüsselung pro Block per: $M \equiv C^{231.953} \pmod{256.027}$:
`82083 65032 119111 114107 115033`

4.13.4 Eine kleine RSA-Cipher-Challenge (1)

Die Aufgabe stammt aus [Sti06, Exercise 4.6]: Prof. Stinson hat dazu eine Lösung veröffentlicht.[123] Es geht aber nicht nur um das Ergebnis, sondern vor allem um die Einzelschritte der Lösung, also um die Darlegung der Kryptoanalyse.[124]

Hier die Aufgabe im Originaltext:

Two samples of RSA ciphertext are presented in Tables 4.14[125] and 4.15[126]. Your task is to decrypt them. The public parameters of the system are

$n = 18.923$ and $e = 1261$ (for Table 4.14) and
$n = 31.313$ and $e = 4913$ (for Table 4.15).

This can be accomplished as follows. First, factor n (which is easy because it is so small). Then compute the exponent d from $\phi(n)$, and, finally, decrypt the ciphertext. Use the square-and-multiply algorithm to exponentiate modulo n.

In order to translate the plaintext back into ordinary English text, you need to know how alphabetic characters are "encoded" as elements in $\mathbb{Z}_n$. Each element of $\mathbb{Z}_n$ represents three alphabetic characters as in the following examples:

$$\begin{array}{lcl} \texttt{DOG} & \mapsto & 3 * 26^2 + 14 * 26 + 6 = 2398 \\ \texttt{CAT} & \mapsto & 2 * 26^2 + 0 * 26 + 19 = 1371 \\ \texttt{ZZZ} & \mapsto & 25 * 26^2 + 25 * 26 + 25 = 17.575\,. \end{array}$$

You will have to invert this process as the final step in your program.

The first plaintext was taken from "The Diary of Samuel Marchbanks", by Robertson Davies, 1947, and the second was taken from "Lake Wobegon Days", by Garrison Keillor, 1985.

[121] Die RSA-Verschlüsselung mit dem Modul $n = 256.027$ ist bei dieser Einstellung korrekt, da die ASCII-Blöcke in Zahlen kleiner oder gleich 255.255 kodiert werden.
[122] In Kapitel 4.19.5 „RSA-Beispiele mit SageMath" finden Sie den Quelltext zur RSA-Exponentiation mit SageMath.

[123] `http://cacr.uwaterloo.ca/~dstinson/solns.html`
[124] Im Szenario der Online-Hilfe zu CT1 und in der Präsentation auf der CT-Webseite wird der Lösungsweg skizziert.

[125] Die Zahlen dieser Tabelle können Sie mit Copy und Paste weiter bearbeiten.
[126] Die Zahlen dieser Tabelle befinden sich auch in der Online-Hilfe zu CT1 im Kapitel „Szenario für die RSA-Demonstration".

12423	11524	7243	7459	14303	6127	10964	16399
9792	13629	14407	18817	18830	13556	3159	16647
5300	13951	81	8986	8007	13167	10022	17213
2264	961	17459	4101	2999	14569	17183	15827
12693	9553	18194	3830	2664	13998	12501	18873
12161	13071	16900	7233	8270	17086	9792	14266
13236	5300	13951	8850	12129	6091	18110	3332
15061	12347	7817	7946	11675	13924	13892	18031
2620	6276	8500	201	8850	11178	16477	10161
3533	13842	7537	12259	18110	44	2364	15570
3460	9886	8687	4481	11231	7547	11383	17910
12867	13203	5102	4742	5053	15407	2976	9330
12192	56	2471	15334	841	13995	17592	13297
2430	9741	11675	424	6686	738	13874	8168
7913	6246	14301	1144	9056	15967	7328	13203
796	195	9872	16979	15404	14130	9105	2001
9792	14251	1498	11296	1105	4502	16979	1105
56	4118	11302	5988	3363	15827	6928	4191
4277	10617	874	13211	11821	3090	18110	44
2364	15570	3460	9886	9988	3798	1158	9872
16979	15404	6127	9872	3652	14838	7437	2540
1367	2512	14407	5053	1521	297	10935	17137
2186	9433	13293	7555	13618	13000	6490	5310
18676	4782	11374	446	4165	11634	3846	14611
2364	6789	11634	4493	4063	4576	17955	7965
11748	14616	11453	17666	925	56	4118	18031
9522	14838	7437	3880	11476	8305	5102	2999
18628	14326	9175	9061	650	18110	8720	15404
2951	722	15334	841	15610	2443	11056	2186

Tabelle 4.14: RSA-Geheimtext A

6340	8309	14010	8936	27358	25023	16481	25809
23614	7135	24996	30590	27570	26486	30388	9395
27584	14999	4517	12146	29421	26439	1606	17881
25774	7647	23901	7372	25774	18436	12056	13547
7908	8635	2149	1908	22076	7372	8686	1304
4082	11803	5314	107	7359	22470	7372	22827
15698	30317	4685	14696	30388	8671	29956	15705
1417	26905	25809	28347	26277	7897	20240	21519
12437	1108	27106	18743	24144	10685	25234	30155
23005	8267	9917	7994	9694	2149	10042	27705
15930	29748	8635	23645	11738	24591	20240	27212
27486	9741	2149	29329	2149	5501	14015	30155
18154	22319	27705	20321	23254	13624	3249	5443
2149	16975	16087	14600	27705	19386	7325	26277
19554	23614	7553	4734	8091	23973	14015	107
3183	17347	25234	4595	21498	6360	19837	8463
6000	31280	29413	2066	369	23204	8425	7792
25973	4477	30989					

Tabelle 4.15: RSA-Geheimtext B

4.13.5 Eine kleine RSA-Cipher-Challenge (2)

Die folgende Aufgabe ist eine korrigierte Variante aus dem Buch von Prof. Yan [Yan00, Example 3.3.7, S. 318]. Es geht aber nicht nur um das Ergebnis, sondern vor allem um die Einzelschritte der Lösung, also um die Darlegung der Kryptoanalyse.[127]

Man kann sich drei völlig unterschiedlich schwierige Aufgaben vorstellen: Gegeben ist jeweils der Geheimtext und der öffentliche Schlüssel (e, n):

- Known-Plaintext: finde den geheimen Schlüssel d unter Benutzung der zusätzlich bekannten Ursprungsnachricht.
- Ciphertext-only: finde d und die Klartextnachricht.
- RSA-Modul knacken, d.h. faktorisieren (ohne Kenntnis der Nachrichten).

$n = 639784868795271438588314150 41,\ e = 17579$

Klartextnachricht[128]:

```
1401202118011200,
1421130205181900,
0118050013010405,
0002250007150400
```

Geheimtext:

```
45411667895024938209259253423,
16597091621432020076311552201,
46468979279750354732637631044,
32870167545903741339819671379
```

Bemerkung:
Die ursprüngliche Nachricht bestand aus einem Satz mit 31 Zeichen (codiert mit dem Großbuchstabenalphabet aus Abschnitt 4.13.2). ann wurden je 16 Dezimalziffern zu einer Zahl zusammengefasst (die letzte Zahl wurde mit Nullen aufgefüllt). Diese Zahlen wurden mit e potenziert.

Beim Entschlüsseln ist darauf zu achten, dass die berechneten Zahlen vorne mit Nullen aufzufüllen sind, um den Klartext zu erhalten.

Wir betonen das, weil in der Implementierung und Standardisierung die Art des Padding sehr wichtig ist für interoperable Algorithmen.

[127] Im Szenario der Online-Hilfe zu CT1 und in der CrypTool-Präsentation werden der Lösungsweg skizziert.
[128] Die Zahlen dieser Tabelle befinden sich auch in der Online-Hilfe zu CT1 im Kapitel „Szenario für die RSA-Demonstration“.

4.14 Anhang: Der größte gemeinsame Teiler (ggT) von ganzen Zahlen und die beiden Algorithmen von Euklid[129]

Der größte gemeinsame Teiler zweier natürlicher Zahlen a und b ist eine wichtige Größe, die sehr schnell berechnet werden kann. Wenn eine Zahl c die Zahlen a und b teilt (d.h. es gibt ein a' und ein b', so dass $a = a' * c$ und $b = b' * c$), dann teilt c auch den Rest r der Division von a durch b. Wir schreiben in aller Kürze: Aus c teilt a und b folgt: c teilt $r = a - \lfloor a/b \rfloor * b$.[130]

Weil die obige Aussage für alle gemeinsamen Teiler c von a und b gilt, folgt für den größten gemeinsamen Teiler von a und b ($\mathrm{ggT}(a, b)$) die Aussage

$$\mathrm{ggT}(a, b) = \mathrm{ggT}(a - \lfloor a/b \rfloor * b, b).$$

Mit dieser Erkenntnis lässt sich der Algorithmus zum Berechnen des ggT zweier Zahlen wie folgt (in Pseudocode) beschreiben:

```
INPUT: a,b != 0
1. if ( a < b ) then  x = a; a = b; b = x; // Vertausche a und b (a > b)
2. a = a - int(a/b) * b        // a wird kleiner b, der ggT(a, b)
                               // bleibt unverändert
3. if ( a != 0 ) then goto 1.  // nach jedem Schritt fällt a, und
                               // der Algorithmus endet, wenn a == 0.
OUTPUT "ggT(a,b) = " b         // b ist der ggT vom ursprünglichen a und b
```

Aus dem ggT lassen sich aber noch weitere Zusammenhänge bestimmen: Dazu betrachtet man für a und b das Gleichungssystem:

$$\begin{aligned} a &= 1 * a + 0 * b \\ b &= 0 * a + 1 * b, \end{aligned}$$

bzw. in Matrix-Schreibweise:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} a \\ b \end{pmatrix}.$$

Wir fassen diese Informationen in der erweiterten Matrix

$$\left(\begin{array}{c|cc} a & 1 & 0 \\ b & 0 & 1 \end{array}\right)$$

zusammen. Wendet man den ggT-Algorithmus auf diese Matrix an, so erhält man den *erweiterten Euklidschen Algorithmus*, mit dem die multiplikative Inverse bestimmt wird.[131]

---

[129] Mit dem Zahlentheorie-Lernprogramm **ZT** können Sie sehen,
a) wie Euklids Algorithmus den ggT berechnet (Lern-Kapitel 1.3, Seiten 14-19/21) und
b) wie Euklids erweiterter Algorithmus das multiplikative Inverse findet (Lern-Kapitel 2.2, Seite 13/40).
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.

CT2 enthält in dem Krypto-Tutorial „**Die Welt der Primzahlen** ⇒ Zahlentheorie ⇒ Zahlentheoretische Funktionen“ die Funktionen „Erweiterter euklidischer Algorithmus“ und „Modulare multiplikative Inverse“.

In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Visualisierungen \ Erweiterter Euklid / Wechselwegnahme**.

[130] Die untere Gaussklammer $\lfloor x \rfloor$ der reellwertigen Zahl x ist definiert als: $\lfloor x \rfloor$ ist die größte ganze Zahl kleiner oder gleich x. Vergleiche Fußnote 134 auf Seite 184.

[131] Einen eher intuitiven und generischen Gang vom einfachen zum erweiterten Euklidschen Algorithmus findet sich in der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*. Siehe NF Teil 2 [WS06].

INPUT: $a, b \neq 0$

0. $x_{1,1} := 1, x_{1,2} := 0, x_{2,1} := 0, x_{2,2} := 1$

1. $\left(\begin{array}{c|cc} a & x_{1,1} & x_{1,2} \\ b & x_{2,1} & x_{2,2} \end{array}\right) := \left(\begin{array}{cc} 0 & 1 \\ 1 & -\lfloor a/b \rfloor * b \end{array}\right) * \left(\begin{array}{c|cc} a & x_{1,1} & x_{1,2} \\ b & x_{2,1} & x_{2,2} \end{array}\right).$

2. `if (b != 0) then goto 1.`

OUTPUT: „ggT$(a, b) = a * x + b * y$: “, „ggT$(a, b) =$ “b, „$x =$“$x_{2,1}$, „$y =$“$x_{2,2}$

Da dieser Algorithmus nur lineare Transformationen durchführt, gelten immer die Gleichungen

$$\begin{aligned} a &= x_{1,1} * a + x_{1,2} * b \\ b &= x_{2,1} * a + x_{2,2} * b, \end{aligned}$$

Am Ende liefert der Algorithmus[132] die erweiterte ggT-Gleichung:

$$gcd(a, b) = a * x_{2,1} + b * x_{2,2}.$$

Beispiel:
Mit dem erweiterten ggT lässt sich für $e = 37$ die modulo 3588 multiplikativ inverse Zahl d bestimmen (d.h. $37 * d \equiv 1 \pmod{3588}$):

0. $\left(\begin{array}{c|cc} 3588 & 1 & 0 \\ 37 & 0 & 1 \end{array}\right)$

1. $\left(\begin{array}{c|cc} 37 & 1 & 0 \\ 36 & 0 & -96 \end{array}\right) = \left(\begin{array}{cc} 0 & 1 \\ 1 & -(\lfloor 3588/36 \rfloor = 96) * 37 \end{array}\right) * \left(\begin{array}{c|cc} 3588 & 1 & 0 \\ 37 & 0 & 1 \end{array}\right).$

2. $\left(\begin{array}{c|cc} 36 & 1 & -96 \\ 1 & -1 & 97 \end{array}\right) = \left(\begin{array}{cc} 0 & 1 \\ 1 & -(\lfloor 37/36 \rfloor = 1) * 36 \end{array}\right) * \left(\begin{array}{c|cc} 37 & 1 & 0 \\ 36 & 0 & -96 \end{array}\right).$

3. $\left(\begin{array}{c|cc} \mathbf{1} & \mathbf{-1} & \mathbf{97} \\ 0 & 37 & -3588 \end{array}\right) = \left(\begin{array}{cc} 0 & 1 \\ 1 & -(\lfloor 36/1 \rfloor = 36) * 1 \end{array}\right) * \left(\begin{array}{c|cc} 36 & 1 & -96 \\ 1 & -1 & 97 \end{array}\right).$

OUTPUT:
ggT$(37, 3588) = a * x + b * y$:
ggT$(37, 3588) = 1$, $x = -1$, $y = 97$.

Es folgt

1. 37 und 3588 sind teilerfremd (37 ist invertierbar modulo 3588).
2. $37 * 97 = (1 * 3588) + 1$ mit anderen Worten $37 * 97 \equiv 1 \pmod{3588}$
   und damit ist die Zahl 97 multiplikativ invers zu 37 modulo 3588.

[132] Wenn der ggT-Algorithmus endet, steht in den Programmvariablen a und b: $a = 0$ und $b = ggT(a, b)$. Bitte beachten Sie, dass die Programmvariablen zu den Zahlen a und b verschieden sind und ihre Gültigkeit nur im Rahmen des Algorithmus haben.

4.15 Anhang: Abschlussbildung

Die Eigenschaft der Abgeschlossenheit innerhalb einer Menge wird immer bezüglich einer Operation definiert. Im folgenden wird gezeigt, wie man für eine gegebene Ausgangsmenge G_0 die abgeschlossene Menge G bezüglich der Operation + (mod 8) konstruiert:

$G_0 = \{2,3\}$ $---$die Addition der Zahlen in G_0 bestimmt weitere Zahlen :

$2+3 \equiv 5 \pmod 8 = 5$

$2+2 \equiv 4 \pmod 8 = 4$

$3+3 \equiv 6 \pmod 8 = 6$

$G_1 = \{2,3,4,5,6\}$ $---$die Addition der Zahlen in G_1 bestimmt :

$3+4 \equiv 7 \pmod 8 = 7$

$3+5 \equiv 8 \pmod 8 = 0$

$3+6 \equiv 9 \pmod 8 = 1$

$G_2 = \{0,1,2,3,4,5,6,7\}$ $---$die Addition der Zahlen in G_2 *erweitert die Menge nicht*!

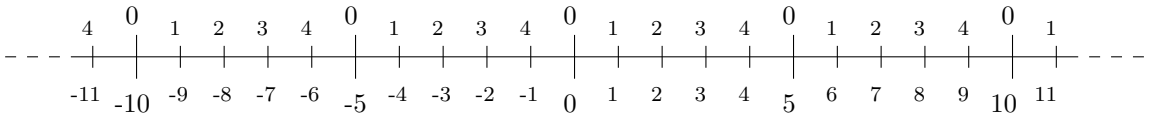
$G_3 = G_2$ $---$man sagt : G_2 ist abgeschlossen bezüglich der Addition (mod 8).

4.16 Anhang: Bemerkungen zur modulo Subtraktion

Beispielweise gilt für die Subtraktion modulo 5: $2-4=-2 \equiv 3 \bmod 5$.
Es gilt also **nicht**, dass: $-2 \equiv 2 \bmod 5$!

Dies gleichzusetzen ist ein häufig gemachter Fehler. Warum dies nicht gleich ist, kann man sich gut verdeutlichen, wenn man die Permutation $(0,1,2,3,4)$ aus $\mathbb{Z}_5$ von z.B. -11 bis $+11$ wiederholt über den Zahlenstrahl aus $\mathbb{Z}$ legt.

Zahlengerade modulo 5



Zahlengerade der ganzen Zahlen

4.17 Anhang: Basisdarstellung und -umwandlung von Zahlen, Abschätzung der Ziffernlänge

Betrachtet man eine Zahl z, so stellt sich die Frage, wie man sie darstellt. Üblich sind die Schreibweisen $z = 2374$ oder $z = \sqrt{2}$. Die zweite Zahl kann nicht in der ersten Form dargestellt werden, da sie unendlich viele Stellen hat. Man umgeht das Problem durch die symbolische Schreibweise. Muss man solche Zahlen in der Ziffernschreibweise darstellen, bleibt nichts anderes übrig als die Zahl zu runden.

Wir haben uns an die Dezimalschreibweise (Zahlen zur Basis 10) gewöhnt. Computer rechnen intern mit Zahlen im Binärformat, die nur bei der Ausgabe in Dezimalschreibweise oder auch manchmal in Hexadezimalschreibweise (Basis 16) dargestellt werden.

Dieser Anhang beschreibt, wie man die Basisdarstellung einer natürlichen Zahl umrechnet in die Darstellung derselben Zahl mit einer anderen Basis, und wie man mit Hilfe der Logarithmusfunktion die Ziffernlänge jeder natürlichen Zahl zu einer beliebigen Basis abschätzen kann.

b-adische Summendarstellung von natürlichen Zahlen

Zur *Basis* b kann man jede natürliche Zahl z darstellen als eine *b-adische Summe von Zahlen*.

$$z = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0,$$

wobei die natürlichen Zahlen $a_i,\ i = 0, \ldots, n$ aus dem Wertebereich $0, 1, 2, \ldots, b-1$ gewählt sind. Wir nennen diese Zahlen a_i *Ziffern*.

Für diese spezielle Summe gilt:
1) Für beliebige Ziffern $a_0, a_1, \ldots, a_n$ gilt: $b^{n+1} > a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$.
2) Es gibt auch Ziffern $a_0, a_1, \ldots, a_n$ (nämlich $a_i = b-1$ für $i = 0, \ldots, n$), so dass $b^{n+1} - 1 \leq a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$.
(Damit lässt sich leicht zeigen, dass sich jede natürliche Zahl als b-adische Summe darstellen lässt).

Wenn man diese Ziffern $a_n a_{n-1} \cdots a_1 a_0$ direkt hintereinander schreibt und die Gewichte b^i weglässt, so erhält man die gewohnte Schreibweise für Zahlen.

Beispiel:
Basis $b = 10$: $10278 = 1 \cdot 10^4 + 0 \cdot 10^3 + 2 \cdot 10^2 + 7 \cdot 10^1 + 8$
Basis $b = 16$: $FE70A = 15 \cdot 16^4 + 14 \cdot 16^3 + 7 \cdot 16^2 + 0 \cdot 16^1 + 10$.

Länge der Zifferndarstellung

Für eine natürliche Zahl z kann die Länge der Zahl in b-adischer Darstellung wie folgt bestimmt werden. Dazu geht man von der Abschätzung $b^{n+1} > z \geq b^n$ aus, wobei $n+1$ die gesuchte Stellenzahl ist. Nach dem Logarithmieren zur Basis b[133] bestimmt man die Ungleichung $n+1 > log_b z \geq n$. Es folgt, dass $n = \lfloor \log_b z \rfloor$.[134] Man bezeichnet mit $l_b(z)$ die Zahl der benötigten

[133] Nach den Logarithmengesetzen gilt für die Basen b und b' die Gleichung $\log_b z = \log_{b'} z / \log_{b'}(b)$. Es ist also einfach, ausgehend von Logarithmentafeln für z.B. die Basis $b' = 10$ den Logarithmus zur Basis $b = 2$ zu berechnen.

[134] Die Funktion $\lfloor x \rfloor$ bestimmt die nächste ganze Zahl kleiner x (wenn $x \geq 0$, dann werden die Nachkommastellen der Zahl einfach abgeschnitten). Man sagt auch *untere Gaußklammer von x*.
Vergleiche Fußnote 130 auf Seite 181.

Ziffern zur Darstellung der Zahl z in b-adischer Schreibweise. Es gilt die Gleichung

$$l_b(z) = \lfloor \log_b z \rfloor + 1.$$

Beispiel 1 (dezimal→hex):
Für die Zahl $z = 234$ in Dezimalschreibweise (EA in hex) bestimmt man die Länge in Hexadezimalschreibweise durch

$$l_{16}(z) = \lfloor \log_{16}(z) \rfloor + 1 = \lfloor \ln(z)/\ln(16) \rfloor + 1 = \lfloor 1,96... \rfloor + 1 = 1 + 1 = 2.$$

Beispiel 2 (dezimal→binär):
Für die Zahl $z = 234$ in Dezimalschreibweise (11101010 in binär) bestimmt man die Länge in Binärschreibweise durch

$$l_2(z) = \lfloor \log_2(z) \rfloor + 1 = \lfloor \ln(z)/\ln(2) \rfloor + 1 = \lfloor 7,87... \rfloor + 1 = 7 + 1 = 8.$$

Beispiel 3 (binär→dezimal):
Für die Zahl $z = 11101010$ in Binärschreibweise (234 dezimal) bestimmt man die Länge in Dezimalschreibweise durch

$$l_{10}(z) = \lfloor \log_{10}(z) \rfloor + 1 = \lfloor \ln(z)/\ln(10) \rfloor + 1 = \lfloor 2,36... \rfloor + 1 = 2 + 1 = 3.$$

Algorithmus zur Basisdarstellung von Zahlen

Ausgehend von der Zahl z bestimmt man die Zahlendarstellung zur Basis b durch den folgenden Algorithmus:

input: z, b
$n := 0, z' := z$
while $z' > 0$ **do**
  $a_n := z' \bmod b$,
  $z' := \lfloor z'/b \rfloor$
  $n := n + 1$
end do
output: $a_n a_{n-1} \cdots a_1 a_0$ Zahlendarstellung zur Basis b.

Beispiel 1 (dezimal→hex):
Die Zahl $z = 234$ in Dezimalschreibweise wird umgewandelt in Hexadezimalschreibweise:
$a_0 = 234 \bmod 16 = 10 = A$, $234/16 = 14 = E$,
$a_1 = 14 \bmod 16 = E$
Damit ergibt sich EA.

Beispiel 2 (binär→dezimal):
Die Binärzahl $z = 1000100101110101$ wird in die Dezimaldarstellung durch die folgenden Berechnungen umgewandelt:

$1000100101110101 = 1001 \pmod{1010} \Longrightarrow a_0 = 9, \quad 1000100101110101/1010 = 110110111110$
$110110111110 = 1000 \pmod{1010} \Longrightarrow a_1 = 8, 110110111110/1010 = 101011111$
$101011111 = 1 \pmod{1010} \Longrightarrow a_2 = 1, 10101111/1010 = 100011$
$100011 = 101 \pmod{1010} \Longrightarrow a_3 = 5, 100011/1010 = 1$
$11 = 11 \pmod{1010} \Longrightarrow a_4 = 3$
Damit ergibt sich $z = 35189$.

4.18 Anhang: Interaktive Präsentation zur RSA-Chiffre

Die folgende Präsentation (letzte Änderung Nov. 2010) zeigt in interaktiver Form die Grundlagen und Funktionsweisen der RSA-Chiffre.

Folgende drei Varianten der Präsentation sind vorhanden:

- Powerpoint 2007 (dynamisch, animiert)[135]
- PDF (statisch, keine Interaktion)[136]
- Flash (kann im Browser gestartet werden, erfordert JavaScript; zeitgesteuerte Wiedergabe)[137]

Abbildung 4.4: Screenshot RSA-Präsentation (PDF)

[135] http://www.cryptool.org/images/ct1/presentations/RSA/RSA-de.pptx
[136] http://www.cryptool.org/images/ct1/presentations/RSA/RSA-de(keine%20Interaktivitaet).pdf
[137] http://www.cryptool.org/images/ct1/presentations/RSA/RSA-Flash-de/player.html

4.19 Anhang: Beispiele mit SageMath

„Nie würde sie ihren Eltern ... von dieser ganzen Welt erzählen können. Nicht von ihrer Arbeit, die im Knacken von Codes bestand. Nicht vom Fiasko der Daemon-Taskforce ... Nicht von den schattenhaften Marionetten, die die Regierung nach ihrer Pfeife tanzen ließen.“

Zitat 14: Daniel Suarez[138]

In diesem Anhang finden Sie SageMath-Quellcode, mit dem Sie die Tabellen und Beispiele des Kapitels 4 („Einführung in die elementare Zahlentheorie mit Beispielen“) berechnen können.

4.19.1 Multiplikationstabellen modulo m

Die Multiplikationstabelle 4.4 (von Seite 130) für $a \times i \pmod m$, mit $m = 17$, $a = 5$ and $a = 6$, und i von 0 bis 16 kann mit folgenden SageMath-Befehlen berechnet werden:

SageMath-Beispiel 4.3 Multiplikationstabelle $a \times i \pmod m$ mit $m = 17$, $a = 5$ and $a = 6$

```
sage: m = 17; a = 5; b = 6
sage: [mod(a * i, m).lift() for i in xrange(m)]
[0, 5, 10, 15, 3, 8, 13, 1, 6, 11, 16, 4, 9, 14, 2, 7, 12]
sage: [mod(b * i, m).lift() for i in xrange(m)]
[0, 6, 12, 1, 7, 13, 2, 8, 14, 3, 9, 15, 4, 10, 16, 5, 11]
```

Die Funktion `mod()` gibt das Objekt zurück, das die natürlichen Zahlen modulo m (in unserem Fall $m = 17$) repräsentiert. Aus dem `Mod`-Objekt kann man die einzelnen Komponenten entweder mit der `component`- oder mit der `lift`-Funktion zurückgewinnen. Wir nutzen hier die Methode `lift()`, um das Objekt in eine Zahl umzuwandeln und auszugeben.

Die weiteren Beispiele der Multiplikationstabelle modulo 13 (Tabelle 4.5) und modulo 12 (Tabelle 4.6) auf Seite 130 kann man auf dieselbe Weise bestimmen, wenn man im Quelltext jeweils `m=17` durch den entsprechenden Zahlenwert (`m=13` bzw. `m=12`) ersetzt.

4.19.2 Schnelles Berechnen hoher Potenzen

Das schnelle Potenzieren modulo m kann mit der SageMath-Funktion `power_mod()` durchgeführt werden. Das Ergebnis dieser Funktion ist eine natürliche Zahl. Sie können mit Hilfe der folgenden Zeilen die Idee der Square-and-Multiply-Methode nachvollziehen, wie sie im Beispiel in Kapitel „Schnelles Berechnen hoher Potenzen“ auf Seite 133 dargestellt ist:

SageMath-Beispiel 4.4 Schnelles Berechnen hoher Potenzen mod $m = 103$

```
sage: a = 87; m = 103
sage: exp = [2, 4, 8, 16, 32, 43]
sage: [power_mod(a, e, m) for e in exp]
[50, 28, 63, 55, 38, 85]
```

[138] Daniel Suarez, „Darknet“, rororo, (c) 2011, Kapitel 19, „Scheideweg“, S. 229, Philips.

4.19.3 Multiplikative Ordnung

Die Ordnung $ord_m(a)$ einer Zahl a in der multiplikativen Gruppe Z_m^* ist die kleinste natürliche Zahl $i \geq 1$ für die gilt $a^i \equiv 1 \bmod m$ (siehe Kapitel 4.9, „Multiplikative Ordnung und Primitivwurzel“).

Um die Tabelle 4.7 auf Seite 142 zu berechnen, können wir alle Potenzen $a^i \pmod{11}$ wie folgt ausgeben:

SageMath-Beispiel 4.5 Tabelle mit allen Potenzen $a^i \pmod{m}$ für $m = 11$, $a = 1, ..., 10$

```
sage: m = 11
sage: for a in xrange(1, m):
....:     print [power_mod(a, i, m) for i in xrange(1, m)]
....:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[2, 4, 8, 5, 10, 9, 7, 3, 6, 1]
[3, 9, 5, 4, 1, 3, 9, 5, 4, 1]
[4, 5, 9, 3, 1, 4, 5, 9, 3, 1]
[5, 3, 4, 9, 1, 5, 3, 4, 9, 1]
[6, 3, 7, 9, 10, 5, 8, 4, 2, 1]
[7, 5, 2, 3, 10, 4, 6, 9, 8, 1]
[8, 9, 6, 4, 10, 3, 2, 5, 7, 1]
[9, 4, 3, 5, 1, 9, 4, 3, 5, 1]
[10, 1, 10, 1, 10, 1, 10, 1, 10, 1]

und die letzte Spalte ergänzt um die Ordnung des jeweiligen a mod (11)

sage: m = 11
sage: for a in xrange(1, m):
....:     lst= [power_mod(a, i, m) for i in xrange(1, m)]
....:     lst.append(multiplicative_order(mod(a,m)))
....:     print lst
....:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[2, 4, 8, 5, 10, 9, 7, 3, 6, 1, 10]
[3, 9, 5, 4, 1, 3, 9, 5, 4, 1, 5]
[4, 5, 9, 3, 1, 4, 5, 9, 3, 1, 5]
[5, 3, 4, 9, 1, 5, 3, 4, 9, 1, 5]
[6, 3, 7, 9, 10, 5, 8, 4, 2, 1, 10]
[7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 10]
[8, 9, 6, 4, 10, 3, 2, 5, 7, 1, 10]
[9, 4, 3, 5, 1, 9, 4, 3, 5, 1, 5]
[10, 1, 10, 1, 10, 1, 10, 1, 10, 1, 2]
```

Die Tabelle 4.8 auf Seite 143 enthält Beispiele für die Ordnung von a modulo 45 ($ord_{45}(a)$) und den Wert der Eulerfunktion von 45 ($\phi(45)$).

Der folgende SageMath-Code erzeugt eine analoge Tabelle.

SageMath-Beispiel 4.6 Tabelle mit allen Potenzen $a^i \pmod{45}$ für $a = 1, ..., 12$ plus der Ordnung von a

```
sage: m = 45
sage: for a in xrange(1, 13):
....:     lst = [power_mod(a, i, m) for i in xrange(1, 13)]
....:     try:
....:         lst.append(multiplicative_order(mod(a, m)))
....:     except:
....:         lst.append("None")
....:     lst.append(euler_phi(m))
....:     print lst
....:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 24]
[2, 4, 8, 16, 32, 19, 38, 31, 17, 34, 23, 1, 12, 24]
[3, 9, 27, 36, 18, 9, 27, 36, 18, 9, 27, 36, 'None', 24]
[4, 16, 19, 31, 34, 1, 4, 16, 19, 31, 34, 1, 6, 24]
[5, 25, 35, 40, 20, 10, 5, 25, 35, 40, 20, 10, 'None', 24]
[6, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 'None', 24]
[7, 4, 28, 16, 22, 19, 43, 31, 37, 34, 13, 1, 12, 24]
[8, 19, 17, 1, 8, 19, 17, 1, 8, 19, 17, 1, 4, 24]
[9, 36, 9, 36, 9, 36, 9, 36, 9, 36, 9, 36, 'None', 24]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 'None', 24]
[11, 31, 26, 16, 41, 1, 11, 31, 26, 16, 41, 1, 6, 24]
[12, 9, 18, 36, 27, 9, 18, 36, 27, 9, 18, 36, 'None', 24]
```

Die Ordnung $\text{ord}_m(a)$ kann nur berechnet werden, wenn a teilerfremd zu m ist. Das kann mit der Abfrage, ob `gcd(a, m)==1`, überprüft werden.

In unserem Codebeispiel haben wir stattdessen die Berechung der multiplikativen Ordnung in einem `try-except`-Block durchgeführt. Auf diese Weise kann SageMath jede Ausnahme und jeden Fehler abfangen, der von der Funktion `multiplicative_order()` geworfen wird. Wird eine Ausnahmen oder ein Fehler im `try`-Block geworfen, wissen wir, dass $\text{ord}_m(a)$ nicht existiert für den gegebenen Wert von a. Daher wird dann im `except`-Block ans Ende der Zeile der String `"None"` angehängt (die Zeile wird durch das Objekt `lst` repräsentiert).

Die Tabelle 4.9 auf Seite 144 enthält Beispiele für die Exponentiationen a^i mod 46 sowie die Ordnungen $ord_{46}(a)$

Der folgende SageMath-Code erzeugt eine analoge Tabelle.

SageMath-Beispiel 4.7 Tabelle mit allen Potenzen $a^i \pmod{46}$ für $a = 1, ..., 23$ plus die Ordnung von a

```
sage: m = 46
sage: euler_phi(m)
22
sage: for a in xrange(1, 24):
....:     lst = [power_mod(a, i, m) for i in xrange(1, 24)]
....:     try:
....:         lst.append(multiplicative_order(mod(a, m)))
....:     except:
....:         lst.append("None")
....:     print lst
....:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[2, 4, 8, 16, 32, 18, 36, 26, 6, 12, 24, 2, 4, 8, 16, 32, 18, 36, 26, 6, 12, 24, 2, 'None']
[3, 9, 27, 35, 13, 39, 25, 29, 41, 31, 1, 3, 9, 27, 35, 13, 39, 25, 29, 41, 31, 1, 3, 11]
[4, 16, 18, 26, 12, 2, 8, 32, 36, 6, 24, 4, 16, 18, 26, 12, 2, 8, 32, 36, 6, 24, 4, 'None']
[5, 25, 33, 27, 43, 31, 17, 39, 11, 9, 45, 41, 21, 13, 19, 3, 15, 29, 7, 35, 37, 1, 5, 22]
[6, 36, 32, 8, 2, 12, 26, 18, 16, 4, 24, 6, 36, 32, 8, 2, 12, 26, 18, 16, 4, 24, 6, 'None']
[7, 3, 21, 9, 17, 27, 5, 35, 15, 13, 45, 39, 43, 25, 37, 29, 19, 41, 11, 31, 33, 1, 7, 22]
[8, 18, 6, 2, 16, 36, 12, 4, 32, 26, 24, 8, 18, 6, 2, 16, 36, 12, 4, 32, 26, 24, 8, 'None']
[9, 35, 39, 29, 31, 3, 27, 13, 25, 41, 1, 9, 35, 39, 29, 31, 3, 27, 13, 25, 41, 1, 9, 11]
[10, 8, 34, 18, 42, 6, 14, 2, 20, 16, 22, 36, 38, 12, 28, 4, 40, 32, 44, 26, 30, 24, 10, 'None']
[11, 29, 43, 13, 5, 9, 7, 31, 19, 25, 45, 35, 17, 3, 33, 41, 37, 39, 15, 27, 21, 1, 11, 22]
[12, 6, 26, 36, 18, 32, 16, 8, 4, 2, 24, 12, 6, 26, 36, 18, 32, 16, 8, 4, 2, 24, 12, 'None']
[13, 31, 35, 41, 27, 29, 9, 25, 3, 39, 1, 13, 31, 35, 41, 27, 29, 9, 25, 3, 39, 1, 13, 11]
[14, 12, 30, 6, 38, 26, 42, 36, 44, 18, 22, 32, 34, 16, 40, 8, 20, 4, 10, 2, 28, 24, 14, 'None']
[15, 41, 17, 25, 7, 13, 11, 27, 37, 3, 45, 31, 5, 29, 21, 39, 33, 35, 19, 9, 43, 1, 15, 22]
[16, 26, 2, 32, 6, 4, 18, 12, 8, 36, 24, 16, 26, 2, 32, 6, 4, 18, 12, 8, 36, 24, 16, 'None']
[17, 13, 37, 31, 21, 35, 43, 41, 7, 27, 45, 29, 33, 9, 15, 25, 11, 3, 5, 39, 19, 1, 17, 22]
[18, 2, 36, 4, 26, 8, 6, 16, 12, 32, 24, 18, 2, 36, 4, 26, 8, 6, 16, 12, 32, 24, 18, 'None']
[19, 39, 5, 3, 11, 25, 15, 9, 33, 29, 45, 27, 7, 41, 43, 35, 21, 31, 37, 13, 17, 1, 19, 22]
[20, 32, 42, 12, 10, 16, 44, 6, 28, 8, 22, 26, 14, 4, 34, 36, 30, 2, 40, 18, 38, 24, 20, 'None']
[21, 27, 15, 39, 37, 41, 33, 3, 17, 35, 45, 25, 19, 31, 7, 9, 5, 13, 43, 29, 11, 1, 21, 22]
[22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 24, 22, 'None']
[23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 'None']
```

Der folgende Code für die Tabellen 4.10 und 4.11 auf Seite 146 f. gibt auch gleich das Ergebnis so aus, dass man es leicht in LaTeX weiter verarbeiten kann. Voraussetzung dafür ist, dass alle Inhalte vorher einem SageMath-Objekt (hier einer Matrix) zugewiesen werden.[139]

SageMath-Beispiel 4.8 Code für Tabellen mit allen Potenzen $a^i \pmod m$ für variable a und i plus Ordnung von a und Eulerphi von m

```
def power_mod_order_matrix(m, max_a, max_i):
    r = matrix(ZZ, max_a+1, max_i+3)
    for a in xrange(0, max_a+1):
        r[a, 0] = a
        for i in xrange(1, max_i+1):
            if a==0:
                r[a,i] = i
            else:
                r[a, i] = power_mod(a, i, m)
        try:
            r[a, max_i+1] = multiplicative_order(mod(a, m))
        except:
            r[a, max_i+1] = 0
        r[a, max_i+2] = euler_phi(m)
    return r

print "\n1: m=45;max_i=13;max_a=13";m=45;max_i=13;max_a=13
r = power_mod_order_matrix(m, max_a, max_i);print r;print latex(r)

print "\n2: m=46;max_i=25;max_a=25";m=46;max_i=25;max_a=25
r = power_mod_order_matrix(m, max_a, max_i);print r.str();print latex(r)

print "\n3: m=14;max_i=13;max_a=16";m=14;max_i=13;max_a=16
r = power_mod_order_matrix(m, max_a, max_i);print r;print latex(r)

print "\n4: m=22;max_i=21;max_a=25";m=22;max_i=21;max_a=25
r = power_mod_order_matrix(m, max_a, max_i);print r.str();print latex(r)
...
3: m=14;max_i=13;max_a=16
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13  0  6]
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  6]
[ 2  2  4  8  2  4  8  2  4  8  2  4  8  2  0  6]
[ 3  3  9 13 11  5  1  3  9 13 11  5  1  3  6  6]
...
\left(\begin{array}{rrrrrrrrrrrrrrr}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 0 & 6\\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 6\\
2 & 2 & 4 & 8 & 2 & 4 & 8 & 2 & 4 & 8 & 2 & 4 & 8 & 2 & 0 & 6\\
3 & 3 & 9 & 13 & 11 & 5 & 1 & 3 & 9 & 13 & 11 & 5 & 1 & 3 & 6 & 6\\
...
```

[139] Anmerkungen zu dem SageMath-Programm, insbesondere den SageMath-Indizes:

- for x in xrange(2, 5) liefert 2,3,4.
- m = matrix(ZZ, 2, 5) hat 2 Zeilen und 5 Spalten.
  Die Zellen haben die Bezeichner m(0,0) bis m(1,4).
- Alle Elemente der Matrix müssen numerisch sein, daher „0“statt „None“.
- Die Ausgabe von Matrizen kann man in SageMath so steuern:

```
sage: from sage.matrix.matrix0 import set_max_cols, set_max_rows
sage: set_max_cols(100)
sage: set_max_rows(100)
```

- Die Zykluslänge in der letzten Spalte in den Tabellen 4.10 und 4.11 wurde noch von Hand ermittelt.

4.19.4 Primitivwurzeln

Das Berechnen von Primitivwurzeln (siehe Kapitel 4.9, „Multiplikative Ordnung und Primitivwurzel“) geht in SageMath sehr einfach: Sei `n` eine natürliche Zahl, dann kann mit dem Befehl `primitive_root(n)` *eine* Primitivwurzel der multiplikativen Gruppe $(\mathbf{Z}/n\mathbf{Z})^*$ berechnet werden, sofern eine existiert. Ist n prim, dann ist das äquivalent zum Berechnen einer Primitivwurzel in $\mathbf{Z}/n\mathbf{Z}$.

Im folgenden berechnen wir die Primitivwurzeln einiger natürlicher Zahlen.

SageMath-Beispiel 4.9 Berechnen einer Primitivwurzel für eine gegebene Primzahl

```
sage: primitive_root(4)
3
sage: primitive_root(22)
13
sage: for p in primes(1, 50):
....:     print p, primitive_root(p)
....:
2 1
3 2
5 2
7 3
11 2
13 2
17 3
19 2
23 5
29 2
31 3
37 2
41 6
43 3
47 5
```

Ist p prim, dann hat $\mathbf{Z}/p\mathbf{Z}$ mindestens eine Primitivwurzel.

Will man `alle` Primitivwurzeln von $(\mathbf{Z}/n\mathbf{Z})^*$ berechnen, und nicht nur irgend eine einzige von $(\mathbf{Z}/n\mathbf{Z})^*$, dann kann man das mit der folgenden selbstgeschriebenen Funktion durchführen.[140]

SageMath-Beispiel 4.10 Funktion „enum_PrimitiveRoots_of_an_Integer“ zur Berechnung aller Primitivwurzeln für eine gegebene Zahl

```
def enum_PrimitiveRoots_of_an_Integer(M):
    r"""
    Return all the primitive roots of the integer M (if possible).
    """
    try:
        g = primitive_root(M)
    except:
        return None
    targetOrder = euler_phi(M)
    L=[]
    # Stepping through all odd integers from 1 up to M, not including
    # M. So this loop only considers values of i where 1 <= i < M.
    for i in xrange(1,M,2):
            testGen = Mod(g^i,M)
            if testGen.multiplicative_order() == targetOrder:
                L.append(testGen.lift())
    # removing duplicates
    return Set(L)

# AA_Start -- Testcases for enum_PrimitiveRoots_of_an_Integer(M)
print "AA_Start -- Testcases for enum_PrimitiveRoots_of_an_Integer(M)"
M=10; print "1-----------Testcase: M = %s" % M
LL = enum_PrimitiveRoots_of_an_Integer(M)
if LL==None:
    print M
else:
    print LL
M=8; print "2-----------Testcase: M = %s" % M
# M=8 hat keine primitive root mod m. Checke, ob per try - except abgefangen.
LL = enum_PrimitiveRoots_of_an_Integer(M)
if LL==None:
    print M
else:
    print LL
M=17; print "3-----------Testcase: M = %s" % M
LL = enum_PrimitiveRoots_of_an_Integer(M)
if LL==None:
    print M
else:
    print LL
# AA_End -- Testcases

OUTPUT:
AA_Start -- Testcases for enum_PrimitiveRoots_of_an_Integer(M)
1-----------Testcase: M = 10
{3, 7}
2-----------Testcase: M = 8
8
3-----------Testcase: M = 17
{3, 5, 6, 7, 10, 11, 12, 14}
```

[140] Der folgende Code wurde als SageMath-Skriptdatei erstellt und nicht-interaktiv ausgeführt. Deshalb gibt es in der Ausgabe keine Zeilen, die mit „sage:“ oder „....:“ anfangen wie in den SageMath-Programmbeispielen bisher.

Das folgende Beispiel listet alle Primitivwurzeln der Primzahl 541 auf.

SageMath-Beispiel 4.11 Tabelle mit allen Primitivwurzeln der vorgegebenen Primzahl 541

```
sage: L=enum_PrimitiveRoots_of_an_Integer(541); L
{2, 517, 10, 523, 13, 14, 527, 528, 18, 531, 24, 539, 30, 37, 40, 51,
54, 55, 59, 62, 65, 67, 68, 72, 73, 77, 83, 86, 87, 91, 94, 96, 98,
99, 107, 113, 114, 116, 117, 126, 127, 128, 131, 132, 138, 150, 152,
153, 156, 158, 163, 176, 181, 183, 184, 195, 197, 199, 206, 208,
210, 213, 218, 220, 223, 224, 244, 248, 250, 257, 258, 259, 260,
261, 263, 267, 269, 270, 271, 272, 274, 278, 280, 281, 282, 283,
284, 291, 293, 297, 317, 318, 321, 323, 328, 331, 333, 335, 342,
344, 346, 357, 358, 360, 365, 378, 383, 385, 388, 389, 391, 403,
409, 410, 413, 414, 415, 424, 425, 427, 428, 434, 442, 443, 445,
447, 450, 454, 455, 458, 464, 468, 469, 473, 474, 476, 479, 482,
486, 487, 490, 501, 504, 511}
sage: len(L)
144
```

Mit etwas Programmieren kann man zählen, wie viele Primitivwurzeln es gibt für alle natürlichen Zahlen in einem gegebenen Zahlenbereich. Wir können das für alle Zahlen oder nur für die Primzahlen in diesem Bereich berechnen.

SageMath-Beispiel 4.12 Funktion „count_PrimitiveRoots_of_an_IntegerRange“ zur Berechnung aller Primitivwurzeln für einen gegebenen Zahlenbereich

```
def count_PrimitiveRoots_of_an_IntegerRange(start, end, bPrimesOnly=True):
        r"""
        Compute all primitive roots of all numbers between start and end,
        inclusive, and count them.
        If the flag bPrimesOnly is True, it performs primality tests, so it
        allows us to count the number of primes from start to end, inclusive.
        If the flag bPrimesOnly is false, it additionally counts these even
        numbers which have no primitive root.
        """
        nCheckedNumb = 0
        nCheckedNumb_WithoutPrimitivRoots = 0
        nPrimitiveRoots = 0
        for n in xrange(start, end+1):
                if bPrimesOnly:
                        if is_prime(n):
                                nCheckedNumb += 1
                                L = enum_PrimitiveRoots_of_an_Integer(n)
                                nPrimitiveRoots += len(L)
                else:
                        nCheckedNumb += 1
                        L = enum_PrimitiveRoots_of_an_Integer(n)
                        if L==None:
                                nCheckedNumb_WithoutPrimitivRoots += 1
                        else:
                                nPrimitiveRoots += len(L)

        if bPrimesOnly:
                print "Found all %s" % nPrimitiveRoots + \
                      " primitive roots of %s primes." % nCheckedNumb
        else:
                if nCheckedNumb_WithoutPrimitivRoots == 0:
                        print "Found all %s " % nPrimitiveRoots + \
                              "primitive roots of %s numbers." % nCheckedNumb
                else:
                        print "Found all %s " % nPrimitiveRoots + \
                              "primitive roots of %s numbers." % \
                                  (nCheckedNumb - nCheckedNumb_WithoutPrimitivRoots)
                        print "(Total of numbers checked: %s, " % nCheckedNumb + \
                              "Amount of numbers without primitive roots: %s)" % \
                                  nCheckedNumb_WithoutPrimitivRoots
```

Um zu sehen, wie lange unser Computer für diese Berechnung braucht, kann man den SageMath-Befehl `time` verwenden.

SageMath-Beispiel 4.13 Funktion „count_PrimitiveRoots_of_an_IntegerRange“: Testfälle und Testausgaben

```
# BB_Start -- Testcases for count_PrimitiveRoots_of_an_IntegerRange(start, end, bPrimesOnly=True)
print "\n\nBB_Start -- Testcases for count_PrimitiveRoots_of_an_IntegerRange(start, end, True)"

print "\n1----------Testcase: (1, 500)"
time count_PrimitiveRoots_of_an_IntegerRange(1, 500)

print "\n2----------Testcase: (5, 6, False)"
time count_PrimitiveRoots_of_an_IntegerRange(5, 6, False)

print "\n3----------Testcase: (1, 500, False)"
time count_PrimitiveRoots_of_an_IntegerRange(1, 500, False)
# BB_End -- Testcases

OUTPUT:
BB_Start -- Testcases for count_PrimitiveRoots_of_an_IntegerRange(start, end, bPrimesOnly=True)

1----------Testcase: (1, 500)
Found all 8070 primitive roots of 95 primes.
Time: CPU 0.94 s, Wall: 0.97 s

2----------Testcase: (5, 6, False)
Found all 3 primitive roots of 2 numbers.
Time: CPU 0.00 s, Wall: 0.00 s

3----------Testcase: (1, 500, False)
Found all 11010 primitive roots of 170 numbers.
(Total of numbers checked: 500, Amount of numbers without primitive roots: 330)
Time: CPU 1.52 s, Wall: 1.59 s
```

Mit unserer selbst erstellten Funktion `enum_PrimitiveRoots_of_an_Integer` kann man alle Primitivwurzeln einer Primzahl p finden.

Die folgende Funktion zählt, wie viele Primitivwurzeln es innerhalb eines Primzahl-Bereiches gibt, und listet diese Primitivwurzeln jeweils auf.

Aus dieser Liste der Primitivwurzeln können wir jeweils die kleinste und größte Primitivwurzel pro $\mathbf{Z}/p\mathbf{Z}$ bestimmen, als auch die Anzahl von Primitivwurzeln pro $\mathbf{Z}/p\mathbf{Z}$ zählen.

SageMath-Beispiel 4.14 Funktion „count_PrimitiveRoots_of_a_PrimesRange“ zur Berechnung aller Primitivwurzeln für ein gegebenes Intervall von Primzahlen

```
def count_PrimitiveRoots_of_a_PrimesRange(start, end):
      r"""
      Compute all primitive roots of all primes between start and end,
      inclusive. This uses a primes iterator.
      """
      nPrimes = 0
      nPrimitiveRoots = 0
      for p in primes(start, end+1):
          L = enum_PrimitiveRoots_of_an_Integer(p)
          print p, len(L)
          nPrimes += 1
          nPrimitiveRoots += len(L)
      print "Found all %s" % nPrimitiveRoots + " primitive roots of %s primes." % nPrimes

# CC_Start -- Testcases for count_PrimitiveRoots_of_a_PrimesRange(start, end)
print "\n\nBB_Start -- Testcases for count_PrimitiveRoots_of_a_PrimesRange(start, end)"
print "-----------Testcase: (1, 1500)"
time count_PrimitiveRoots_of_a_PrimesRange(1, 1500)
# CC_End -- Testcases

OUTPUT:
CC_Start -- Testcases for count_PrimitiveRoots_of_a_PrimesRange(start, end)
-----------Testcase: (1, 1500)
2 1
3 1
5 2
7 2
11 4
13 4
17 8
19 6
23 10
29 12
31 8
37 12
...
1483 432
1487 742
1489 480
1493 744
1499 636
Found all 62044 primitive roots of 239 primes.
Time: CPU 7.55 s, Wall: 7.85 s
```

Die Funktion `count_PrimitiveRoots_of_a_PrimesRange` wurde von Minh Van Nguyen leicht geändert, um eine Liste (Datenbank) aller Primitivwurzeln für alle Primzahlen zwischen 1 und 100.000 zu erstellen.

SageMath-Beispiel 4.15 Code zur Erstellung einer Liste mit allen Primitivwurzeln für alle Primzahlen zwischen 1 und 100.000

```
start = 1
end = 10^5
fileName = "./primroots.dat"
file = open(fileName, "w")
for p in primes(start, end+1):
    L = enum_PrimitiveRoots_of_an_Integer(p)
    print p, len(L)
    # Output to a file. The format is:
    # (1) the prime number p under consideration
    # (2) the number of primitive roots of Z/pZ
    # (3) all the primitive roots of Z/pZ
    file.write(str(p) + " " + str(len(L)) + " " + str(L) + "\n")
    file.flush()
file.close()
```

Auch dieser Code und die Funktion `enum_PrimitiveRoots_of_an_Integer` wurden mit einer Sage-Skriptdatei nicht-interaktiv ausgeführt. Die Ausführung dauerte mit SageMath 7.2 auf einem modernen PC rund 6 h.

Im Bereich 1 bis 100.000 liegen 9.592 Primzahlen. Für sie wurden insgesamt mehr als 169 Millionen Primitivwurzeln berechnet. Für jede Primzahl $p > 3$ gilt, dass zwischen 20 % und knapp 50 % aller Zahlen zwischen 1 und p eine Primitivwurzel davon darstellen.

Die resultierende Datei „primroots_1-100000.dat“ enthält die Liste aller Primitivwurzeln für alle Primzahlen zwischen 1 und 100.000 inklusive. Dies ist eine große Datei (ca. 1,1 GB unkomprimiert, und 156 MB komprimiert mit 7Zip). Die komprimierte Datei kann abgerufen werden unter
`https://www.cryptool.org/images/ctp/documents/primroots_1-100000.7z`.

Ihr Inhalt sieht wie folgt aus:

```
2      1      {1}
3      1      {2}
5      2      {2, 3}
7      2      {3, 5}
11     4      {8, 2, 6, 7}
...
99989  42840  {2, 3, 8, 10, 11, 13, 14, ..., 99978, 99979, 99981, 99986, 99987}
99991  24000  {65539, 6, 65546, 11, 12, ..., 65518, 65520, 87379, 65526, 65528}
```

Das Sage-Skript 4.16 berechnet für jede Primzahl (bis „end“) alle Primitivwurzeln, und gibt jeweils die Anzahl unterschiedlicher Primitivwurzeln und die jeweils kleinste Primitivwurzel aus.

SageMath-Beispiel 4.16 Code zur Erstellung einer Liste der jeweils kleinsten Primitivwurzeln für alle Primzahlen zwischen 1 und 1.000.000

```
start = 1
end = 10^6
fileName = "./primroot-smallest_up-to-one-million.dat"
file = open(fileName, "w")
file.write(timestring + "\n")
file.flush()
for p in primes(start, end+1):
        L = enum_PrimitiveRoots_of_an_Integer(p)
        # Output to commandline only p and number of prim roots of Z_p
        print p, len(L)
        # Output more to a file. The format is:
        # (1) the prime number p under consideration
        # (2) the number of primitive roots of Z_p
        # (3) the smallest primitive roots of Z_p
        LL = sorted(L) # necessary as the smallest primroot is not always
                       # found first by the enum fct (see L of p=43)
        file.write(str(p) + " " + str(len(L)) + " " + str(LL[0]) + "\n")
        file.flush()
file.flush()
file.close()
```

Das Sage-Skript 4.16 wurde nach mehreren Wochen gestoppt (wo es auf einem modernen PC mit SageMath 7.2 lief), nachdem es alle Primzahlen bis zu einer halben Million auswertete. Das Ergebnis wurde in der Datei „primroot_number-of-and-smallest_up-to-prime-500107.dat“ gespeichert (sie ist unkomprimiert 617 kB groß, und 178 kB komprimiert mit 7Zip).

Die komprimierte Datei kann abgerufen werden unter `https://www.cryptool.org/images/ctp/documents/primroot_number-of-and-smallest_up-to-prime-500107.7z`.

Diese Datenbank enthält für jede einzelne Primzahl p zwischen 1 und 500.107 die Anzahl der zugehörigen Primitivwurzeln und die kleinste Primitivwurzel mod p. Es gilt, dass die Anzahl der Primitivwurzeln (für $p > 3$) immer gerade ist. Eine Formel, die die Anzahl der Primitivwurzeln zu einer Zahl angibt, kennen wir nicht. Diese Datenbank könnte für Zahlentheoretiker interessant sein.[141] Ihr Inhalt sieht wie folgt aus:

```
2       1       1
3       1       2
5       2       2
7       2       3
11      4       2
13      4       2
17      8       3
...
99989   42840   2
99991   24000   6
100003  28560   2
...
500069  250032  2
500083  151520  2
500107  156864  2
```

[141] Siehe auch Re: [sage-devel] What can we do with a database of primitive roots? `https://groups.google.com/forum/m/#!topic/sage-devel/TA5Nk2GdhOI`

Wenn man pro Primzahl nur die kleinste Primitivwurzel sucht, kann man dieses Skript drastisch beschleunigen, indem man anhand der Theorie die möglichen Kandidaten direkter sucht (anstatt mit `enum_PrimitiveRoots_of_an_Integer` erstmal alle Primroots zu generieren).

Die Datei „primroots_1-100000.dat“ wurde dann als Input benutzt, um mit dem folgenden Code (Sage-Skript 4.17) drei Grafiken zu erstellen.

SageMath-Beispiel 4.17 Code zur Erzeugung der Grafiken zur Verteilung der Primitivwurzeln

```
sage: # open a database file on primitive roots from 1 to 100,000
sage: file = open("/scratch/mvngu/primroots.dat", "r")
sage: plist = []    # list of all primes from 1 to 100,000
sage: nlist = []    # number of primitive roots modulo prime p
sage: minlist = []  # smallest primitive root modulo prime p
sage: maxlist = []  # largest primitive root modulo prime p
sage: for line in file:
....:     # get a line from the database file and tokenize it for processing
....:     line = line.strip().split(" ", 2)
....:     # extract the prime number p in question
....:     plist.append(Integer(line[0]))
....:     # extract the number of primitive roots modulo p
....:     nlist.append(Integer(line[1]))
....:     # extract the list of all primitive roots modulo p
....:     line = line[-1]
....:     line = line.replace("{", "")
....:     line = line.replace("}", "")
....:     line = line.split(", ")
....:     # sort the list in non-decreasing order
....:     line = [Integer(s) for s in line]
....:     line.sort()
....:     # get the smallest primitive root modulo p
....:     minlist.append(line[0])
....:     # get the largest primitive root modulo p
....:     maxlist.append(line[-1])
....:
sage: file.close()  # close the database file
sage: # plot of number of primitive roots modulo p
sage: nplot = point2d(zip(plist, nlist), pointsize=1)
sage: nplot.axes_labels(["x", "y"])
sage: nplot
sage: # plot of smallest primitive root modulo prime p
sage: minplot = point2d(zip(plist, minlist), pointsize=1)
sage: minplot.axes_labels(["x", "y"])
sage: minplot
sage: # plot of largest primitive root modulo prime p
sage: maxplot = point2d(zip(plist, maxlist), pointsize=1)
sage: maxplot.axes_labels(["x", "y"])
sage: maxplot
```

Abbildung 4.5 gibt die Anzahl der Primitivwurzeln für jede Primzahl zwischen 1 und 100.000 aus. Die x-Achse repräsentiert die Primzahlen 1 bis 100.000, die y-Achse gibt die Anzahl der Primitivwurzeln pro Primzahl aus.

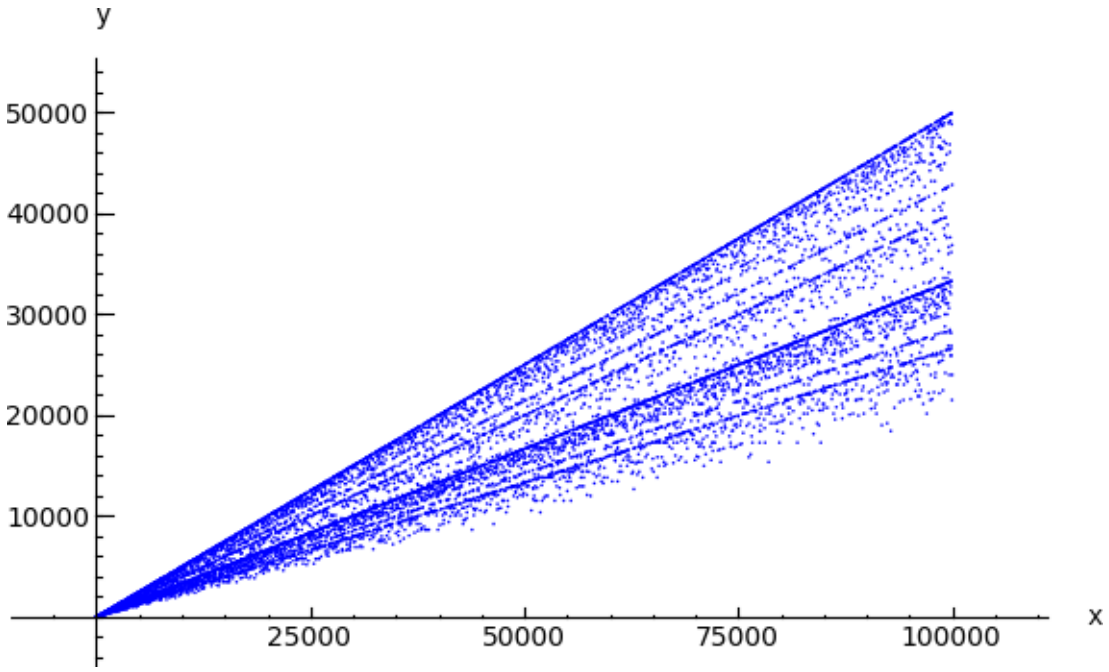


Abbildung 4.5: Die Anzahl der Primitivwurzeln für alle Primzahlen zwischen 1 und 100.000

Abbildung 4.6 gibt die kleinste Primitivwurzel von jeder Primzahl zwischen 1 und 100.000 aus. Die x-Achse repräsentiert die Primzahlen 1 bis 100.000, die y-Achse gibt die kleinste Primitivwurzel pro Primzahl aus.

Abbildung 4.7 zeigt die entsprechende Grafik mit der größten Primitivwurzel zu jeder Primzahl im obigen Intervall.

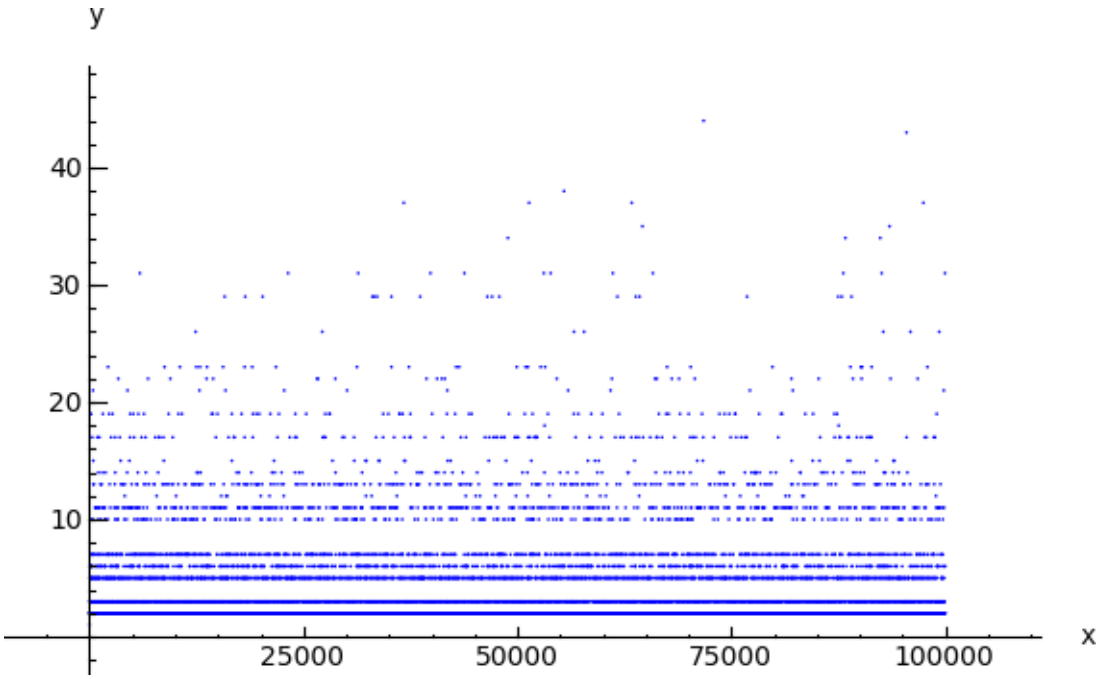


Abbildung 4.6: Die kleinste Primitivwurzel von jeder Primzahl zwischen 1 und 100.000

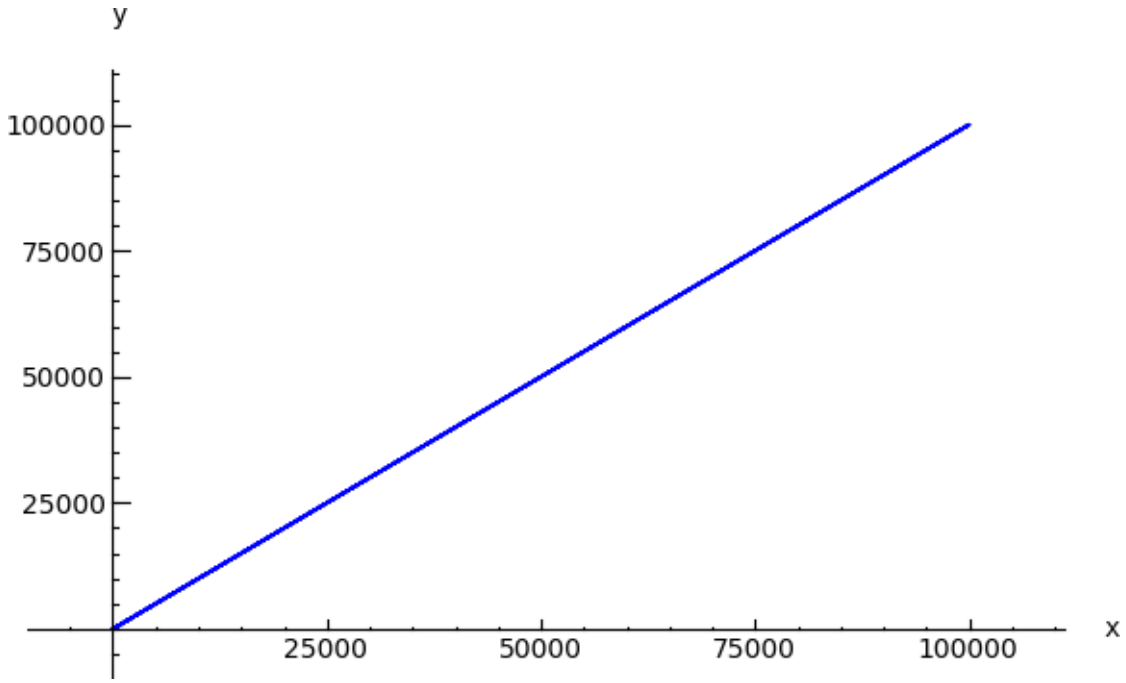


Abbildung 4.7: Die größte Primitivwurzel von jeder Primzahl zwischen 1 und 100.000

4.19.5 RSA-Beispiele mit SageMath

In diesem Abschnitt sind die SageMath-Quelltexte für die einfachen RSA-Beispiele im Kapitel 4.13 („Das RSA-Verfahren mit konkreten Zahlen“) angegeben.

Beispiel auf Seite 174:
Die RSA-Exponentiation M^{37} (mod 3713) für die Nachricht $M = 120$ kann mit SageMath folgendermaßen ausgeführt werden:

```
sage: power_mod(120, 37, 3713)
1404
```

Beispiel auf Seite 175:
Die Faktorisierung von $\phi(256.027) = 255.016 = 2^3 * 127 * 251$ kann mit SageMath folgendermaßen durchgeführt werden:

SageMath-Beispiel 4.18 Faktorisierung einer Zahl

```
sage: factor(255016)
2^3 * 127 * 251
```

Beispiel auf Seite 175:
RSA-Verschlüsselung mit SageMath:

SageMath-Beispiel 4.19 RSA-Verschlüsselung durch modulare Exponentiation einer Zahl (als Nachricht)

```
sage: A = [82, 83, 65, 32, 119, 111, 114, 107, 115, 33]
sage: e = 65537; m = 256027
sage: [power_mod(a, e, m) for a in A]
[212984, 25546, 104529, 31692, 248407, 100412, 54196, 100184, 58179, 227433]
```

Beispiel auf Seite 176:
RSA-Verschlüsselung mit SageMath:

```
sage: A = [21075, 16672, 30575, 29291, 29473]
sage: e = 65537; m = 256027
sage: [power_mod(a, e, m) for a in A]
[158721, 137346, 37358, 240130, 112898]
```

Beispiel auf Seite 177:
RSA-Verschlüsselung mit SageMath:

```
sage: A = [82083, 65032, 119111, 114107, 115033]
sage: e = 65537; m = 256027
sage: [power_mod(a, e, m) for a in A]
[198967, 51405, 254571, 115318, 14251]
```

4.19.6 Wie viele private RSA-Schlüssel d gibt es innerhalb eines Modulo-Bereiches?

Die RSA-Verschlüsselung wurde beschrieben in Abschnitt 4.10.2 („Funktionsweise des RSA-Verfahrens“). Schritt 1 bis 3 definieren die Schlüsselerzeugung, Schritt 4 und 5 stellen die eigentliche Verschlüsselung dar:

1. Wähle zwei unterschiedliche Primzahlen p and q und berechne $n = p * q$.
   Der Wert n wird RSA-Modul genannt.

2. Wähle ein zufälliges $e \in \{2, \cdots, n-1\}$, so dass gilt:
   e ist relativ prim zu $\phi(n) = (p-1) * (q-1)$.
   Danach kann man p und q „wegwerfen“.

3. Wähle $d \in \{1, \cdots, n-1\}$ mit $e * d \equiv 1 \pmod{\phi(n)}$,
   d.h. d ist die multiplikative Inverse von e modulo $\phi(n)$. Dann kann man $\phi(n)$ „wegwerfen“.

   $\rightarrow (n, e)$ ist der öffentliche Schlüssel P.
   $\rightarrow (n, d)$ ist der private Schlüssel S (nur d muss man geheim halten).

4. Zur Verschlüsselung wird die Nachricht als (binäre) Ziffernfolge geschrieben. Diese Ziffernfolge wird dann so in gleich lange Teilfolgen aufgeteilt, dass jede Teilfolge eine Zahl kleiner als n darstellt.

5. Vorgang der Verschlüsselung auf dem Klartext (bzw. auf Teilen davon) $M \in \{1, \cdots, n-1\}$:

$$C = E((n, e); M) := M^e \pmod{n}.$$

Standardmäßig versucht man, einen mit RSA verschlüsselten Geheimtext C dadurch zu knacken, dass man den öffentlichen Schlüssel des Empfängers betrachtet und versucht, n zu faktorisieren. Hat man das erreicht, geht man wieder durch die Schritte 2 und 3 und erzeugt den privaten Schlüssel e, den man zum Entschlüsseln des Geheimtextes braucht.

Gemäß dem „Primzahlsatz“[142] geht die Anzahl der Primzahlen $PI(x)$ asymptotisch gegen $x/ln(x)$. Zwischen 1 und einem gegebenen n gibt es also ca. $n/ln(n)$ unterschiedliche Primzahlen.

Will man nicht faktorisieren, sondern stellt sich eine Frage ähnlich wie bei den klassischen Verschlüsselungsverfahren, kann man herausfinden wollen: Wie viele verschiedene private Schlüssel (n, d) gibt es für einen bestimmten Bereich der Schlüsselgröße $n \in [a, b]$?[143]

[142] Siehe Abschnitt 3.7.2 „(Dichte und Verteilung der Primzahlen“).
[143] Kapitel 4.8.5 („Wie viele private RSA-Schlüssel d gibt es modulo 26“), S. 140 behandelt den Spezialfall $n = 26$.

Das SageMath-Programm 4.20 unten definiert die Funktion `count_Number_of_RSA_Keys`, die diese Frage konkret beantwortet (wenn der Modulus nicht zu groß ist).[144]

Weil es mehr private Schlüssel (n, d) innerhalb eines größeren Bereiches von Werten für n gibt, ist das Brute-Force-Faktorisieren viel effizienter als das Durchprobieren aller möglichen Schlüssel.

[144]
a) Der Aufruf `sage: count_Number_of_RSA_Keys(100, 1000)` bedeutet, dass man das Intervall $[100, 1000]$ für n betrachtet. n war definiert durch die beiden Primzahlen $p, q : n = p * q$.
Daher kann hier die eine Primzahl höchstens den Wert 500 annehmen, weil $2 * 500 = 1000$ (d.h. wenn die andere Primzahl den kleinst-möglichen Primzahlwert 2 annimmt).
Die Anzahl möglicher Primzahl-Kombinationen beträgt: $comb = 258$.
Die Anzahl der Primzahlen im gegebenen Bereich beträgt: 143.
Die Anzahl der privaten Schlüssel beträgt: 34.816.

b) Der Aufruf `sage: count_Number_of_RSA_Keys(100, 100, True)` hat die folgende Ausgabe:
- Number of private keys for modulus in a given range: 0
- Number of primes in a given range: 0
Der Grund dafür ist, dass mit diesem Aufruf nur $n = 100$ betrachtet wird und die Funktion nur semiprime n untersucht. Die Zahl 100 ist nicht semiprim, d.h. 100 ist nicht das Produkt von genau zwei Primzahlen.

SageMath-Beispiel 4.20 Wie viele private RSA-Schlüssel d gibt es, wenn man den Bereich für die Schlüsselgröße n kennt?

```
def count_Number_of_RSA_Keys(start, end, Verbose=False):
      r"""
      How many private RSA keys (n,d) exist, if only modulus N is given, and start <= N <= end?
        (prime_range(u,o) delivers all primes >=u und < o).
      """
      a = start
      b = end
      s = 0
      comb = 0
      for p in prime_range(1, b/2+1):
          for q in prime_range(p + 1, b/2+1):
              if a <= p * q and p * q <= b:
                  comb = comb +1
                  s = s + (euler_phi(euler_phi(p * q))-1)
                  if Verbose:
                      print "p=%s, " % p + "q=%s, " % q + "s=%s" % s
      print "Number of private keys d for modulus in a given range: %s" % s + " (comb=%s), " % comb

      # Just for comparison: How many primes are in this range?
      s = 0
      for p in prime_range(a, b+1):
          if Verbose:
              print "a=%s, " % a + "b=%s, " % b + "p=%s" % p
          s = s + 1
      print "Number of primes in a given range: %s" % s

print "\n\nDD_Start -- Testcases for count_Number_of_RSA_Keys(start, end)"
print "\n-----------Testcase: (100, 1000) [Should deliver 34.816]"
time count_Number_of_RSA_Keys(100, 1000)
print "\n-----------Testcase: (100, 107, True) [Should deliver 23]"
time count_Number_of_RSA_Keys(100, 107, True)
u = 10^3; o = 10^4;
print "\n-----------Testcase: (%s, " % u + "%s) [Should deliver 3.260.044]" % o
time count_Number_of_RSA_Keys(u, o)

OUTPUT:
DD_Start -- Testcases for count_Number_of_RSA_Keys(start, end)

-----------Testcase: (100, 1000) [Should deliver 34.816]
Number of private keys d for modulus in a given range: 34816 (comb=258),
Number of primes in a given range: 143
Time: CPU 0.03 s, Wall: 0.04 s

-----------Testcase: (100, 107, True) [Should deliver 23]
p=2, q=53, s=23
Number of private keys d for modulus in a given range: 23 (comb=1),
a=100, b=107, p=101
a=100, b=107, p=103
a=100, b=107, p=107
Number of primes in a given range: 3
Time: CPU 0.00 s, Wall: 0.00 s

-----------Testcase: (1000, 10000) [Should deliver 3.260.044]
Number of private keys d for modulus in a given range: 3260044 (comb=2312),
Number of primes in a given range: 1061
Time: CPU 0.63 s, Wall: 0.66 s
```

4.19.7 RSA-Fixpunkte $m^e = m \bmod n$ mit $m \in \{1, ..., n-1\}$

Auch Verschlüsselungsverfahren können Fixpunkte haben, also Texte, deren Chiffrat mit dem Original übereinstimmt. In der Mathematik nennt man Variablen, die von einem Verfahren (Funktion) auf sich selbst abgebildet werden, Fixpunkte. In der Kryptographie nennt man entsprechende Nachrichten „unconcealed messages“ („unconcealed“ = unverborgen, offen).

Generell gilt: Je mehr Fixpunkte ein Verschlüsselungsalgorithmus besitzt, desto einfacher ist es, ihn zu knacken.

Beim RSA-Verfahren gilt: $n = pq$ ist das Produkt zweier verschiedener Primzahlen, und es gibt ein e mit $ggT(e, (p-1)(q-1)) = 1$. Die Verschlüsselung erfolgt mit $c = m^e \mod n$. Ein Fixpunkt beim RSA-Verfahren ist eine Nachricht m, für die gilt: $m = m^e \mod n$. Das Ergebnis der Verschlüsselung ist wieder die gegebene Nachricht.

Die Wahrscheinlichkeit für das Auftreten von Fixpunkten ist bei RSA bei genügend großem n allerdings sehr gering – wie Abbildung 4.8 zeigt: Im Durchschnitt fanden wir nicht mehr als 40 Fixpunkte.

Studenten nehmen oft an, dass es viele Fixpunkte gibt, da sie beim Probieren mit relativ **kleinen** Primzahlen immer auf „relativ“ viele Fixpunkte-Beispiele stoßen, denn m = 0, 1 und n-1 sind immer auch Fixpunkte.

In der Praxis, also bei groß genug gewählten Primzahlen, haben Fixpunkte keine Bedeutung für die Sicherheit von RSA. Deshalb bezieht sich dieser Abschnitt eher auf mathematische Fragen.[145]

4.19.7.1 Die Anzahl der RSA-Fixpunkte

In diesem Kapitel zeigen wir, wie viele RSA-Fixpunkte es für $m \in \{1, ..., n-1\}$ gibt.

Satz 4.19.1. *Die Anzahl der Fixpunkte $m^e = m \bmod n$ mit $m \in \{1, ..., n-1\}$ ist $ggT(p-1, e-1) \cdot ggT(q-1, e-1)$.*

Beweis
Sei $m^e = m \bmod n$. Nach dem CRT[146] sind die beiden folgenden Aussagen äquivalent:

$$[m^e = m \bmod n] \quad \Leftrightarrow \quad [m^e = m \bmod p \quad \text{und} \quad m^e = m \bmod q]$$

Weiterhin ist die Zerlegung auf der rechten Seite äquivalent zu:

$$m^{e-1} = 1 \bmod p \quad \text{und} \quad m^{e-1} = 1 \bmod q.$$

Wir betrachten $m^{e-1} = 1 \bmod p$ und suchen alle $(e-1)$-ten Einheitswurzeln[147] in $\mathbb{Z}_p^*$.

---

[145] Dank geht an Taras Shevchenko, der Teile des Inhalts dieses Abschnitts zusammentrug, und an Volker Simon, der das SageMath-Programm 4.21 „Getfixpoints“ schrieb.

[146] CRT = Chinesischer Restsatz. `http://de.wikipedia.org/wiki/Chinesischer_Restsatz`

[147] - In der Algebra werden Zahlen x, deren n-te Potenz die Zahl 1 ergibt, n-te **Einheitswurzeln** genannt.
- Eine n-te Einheitswurzel x heißt **primitiv**, falls für x gilt:

$$x^n = 1 \text{ und } x^k \neq 1 \quad (k = 1, 2, 3, ..., n-1)$$

- Ist F ein endlicher Körper und n eine natürliche Zahl, dann ist eine n-te Einheitswurzel in F eine Lösung der Gleichung

$$x^n - 1 = 0 \text{ in } F$$

Es gilt: $\mathbb{Z}_p^*$ für p prim ist zyklisch. $\Rightarrow$ Es existiert ein Generator g, der $\mathbb{Z}_p^*$ erzeugt: $\mathbb{Z}_p^* =< g >$.
Der ff. Satz aus [Kat01, S. 69] charakterisiert alle $(e-1)$-ten Einheitswurzeln in $\mathbb{Z}_p^*$:

Satz 4.19.2. *g^α ist genau dann $(e-1)$-te Einheitswurzel in $\mathbb{Z}_p^*$, wenn $(e-1)\alpha = 0 \bmod p-1$. Davon gibt es $ggT(p-1, e-1)$ viele.*

Beweis
Die erste Behauptung ergibt sich direkt aus dem kleinen Satz von Fermat:

$$g^{\alpha(e-1)} = 1 \bmod p \quad \Rightarrow \quad \alpha(e-1) = 0 \bmod p-1$$

Sei $\delta = ggT(p-1, e-1)$. Aus $\alpha(e-1) = 0 \bmod p-1$ folgt, dass $\frac{\alpha(e-1)}{\delta} = 0 \bmod \frac{p-1}{\delta}$.
Da $\frac{e-1}{\delta}$ und $\frac{p-1}{\delta}$ teilerfremd sind (da jeweils durch den ggT ihrer Zähler gekürzt wurde), muss α ein Vielfaches von $\frac{p-1}{\delta}$ sein.

$$\alpha\frac{p-1}{\delta} \quad \text{mit} \quad \alpha = 1, ..., \delta$$

Diese δ verschiedenen Potenzen entsprechen dann den $(e-1)$-ten Einheitswurzeln $g^{\alpha\frac{p-1}{\delta}} \bmod p$ in $\mathbb{Z}_p^*$. □
Analog für q: Für $m^{e-1} = 1 \bmod q$ haben wir dann $ggT(q-1, e-1)$ viele $(e-1)$-te Einheitswurzeln.

Die Anzahl der Arten, die $(e-1)$-ten Einheitswurzeln in $\mathbb{Z}_p^*$ und $\mathbb{Z}_q^*$ zu kombinieren, ergibt die Gesamt-Anzahl der RSA-Fixpunkte $m^e = m \bmod n$ mit $m \in \{1, ..., n-1\}$:
$ggT(p-1, e-1) \cdot ggT(q-1, e-1)$

Nimmt man $m = 0$ hinzu, ergibt sich Satz 4.19.3:

Satz 4.19.3. *Wenn $m \in \{0, ..., n-1\}$ ist, dann ist die Anzahl der RSA-Fixpunkte:*

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

□

4.19.7.2 Untere Schranke für die Anzahl der RSA-Fixpunkte

Im folgenden Kapitel zeigen wir, dass eine untere Schranke für die Anzahl der RSA-Fixpunkte existiert. Diese untere Schranke 6 liegt vor, wenn die beiden unterschiedlichen RSA-Primzahlen die kleinstmöglichen sind (2 und 3).

Behauptung 1: Sei $p = 2, q = 3$.
Die Anzahl der RSA-Fixpunkte für $p = 2$ und $q = 3$ ist
$(\underbrace{ggT(p-1, e-1)}_{=1}+1) \cdot (\underbrace{ggT(q-1, e-1)}_{=2}+1) = 2 \cdot 3 = 6$

Behauptung 2: Sei $p \neq q; p > 2, q > 2$.
Die Anzahl der RSA-Fixpunkte für $p \neq q; p, q > 2$ ist ≥ 9.

Beweis (der 2. Behauptung)
Da p und q prim sind, sind $(p-1)$ und $(q-1)$ für $p, q > 2$ gerade.

Nach dem RSA-Verfahren ist e so zu wählen, dass $1 < e < \phi(n) = (p-1)(q-1)$ und $ggT(e,(p-1)(q-1)) = 1$
Da $(p-1)$ und $(q-1)$ gerade sind, ist e ungerade $\Rightarrow e-1$ ist gerade.
Da $(p-1)$ und $(e-1)$ gerade sind, gilt:
$ggT(p-1,e-1) \geq 2$
$\Rightarrow (ggT(p-1,e-1)+1) \geq 3$ und $(ggT(q-1,e-1)+1) \geq 3$
$\Rightarrow (ggT(p-1,e-1)+1) \cdot (ggT(q-1,e-1)+1) \geq 9$ □

Beispiele:
Für $(e,n) = (17,6)$ sind alle sechs möglichen Nachrichten {0,1,2,3,4,5} Fixpunkte (bei $n = 6$ ist das unabhängig vom Wert von e).
Für $(e,n) = (17,10)$ sind alle 10 möglichen Nachrichten Fixpunkte.
Für $(e,n) = (19,10)$ sind nur 6 der 10 möglichen Nachrichten Fixpunkte.

4.19.7.3 Ungeschickte Wahl von e

In diesem Kapitel zeigen wir, dass mit $e = 1+kgV(p-1,q-1)$ jede Verschlüsselung ein Fixpunkt ist (unabhängig von der Größe von p,q oder n, wird m auf sich selbst abgebildet); und erweitern das dann auf alle möglichen schlecht gewählten Werte für e.

Wenn $e = 1$, dann gilt für alle m: $c = m^e = m$. Das ist der Trivialfall.

Behauptung: Sei $p,q > 2$.
Wenn $e = 1 + kgV(p-1,q-1)$, dann gilt für alle $m \in \{1,...,n-1\}$: $m^e = m \bmod n$.

Beweis
Es gilt:
- $e \cdot d = 1 \mod \phi(n)$ oder $e \cdot d = 1 \mod kgV(p-1,q-1)$
- $m^x \mod n = m^{x \bmod \phi(n)} \mod n$

Verschlüsseln von Nachrichten:
$c = m^e \mod n$, wobei c der Geheimtext und m der Klartext ist.

Entschlüsseln von Nachrichten:
$m' = c^d \mod n$, wobei d die multiplikative Inverse von e ist.

Zu zeigen ist: $c = m \mod n$ für das gewählte e.

$c = m^e \mod n$
$c = m^{1+kgV(p-1,q-1)} \mod n$ # Umformung gilt aufgrund der Voraussetzung
$c = m^1 \cdot m^{k\cdot(p-1)\cdot(q-1)} \mod n$
$c = m^1 \cdot m^{[k\cdot\phi(n)] \bmod \phi(n)} \mod n$
$c = m^1 \cdot m^0 = m \mod n$ □

Beispiel: Fixpunkteigenschaft für alle m:
Gegeben sei $n = p \cdot q = 13 \cdot 37 = 481$
$\Rightarrow \phi(n) = (p-1)(q-1) = 12 \cdot 36 = 432$
$\Rightarrow e = kgV(p-1, q-1) + 1 = kgV(12, 36) + 1 = 36 + 1 = 37.$
Mit $m \in \{4, 6, 7, 480\}$ ergibt sich $m^e \mod n$ als:
$4^{37} \mod 481 = 4$
$6^{37} \mod 481 = 6$
$7^{37} \mod 481 = 7$
$480^{37} \mod 481 = 480$

Es gibt nicht nur das einzige e (siehe oben), so dass für alle $m \in \{1, ..., n-1\}$ die Fixpunkteigenschaft $m^e = m \bmod n$ gilt.[148]

Satz 4.19.4. *Die vollständige Fixpunkteigenschaft aller m gilt für jedes* $e = j \cdot kgV(p-1, q-1) + 1$, *wobei* $j = 0, 1, 2, 3, 4, ...$ *bis* $e \leq \phi(n)$.

Beispiel: Weitere Werte für e mit Fixpunkteigenschaft:
Betrachten wir wieder $n = p \cdot q = 13 \cdot 37 = 481$ mit $kgV(p-1, q-1) = kgV(12, 36) = 36$.
Dann kann e die ff. Werte annehmen: $e = j \cdot kgV(p-1, q-1) + 1$ für $j = 0, 1, 2, ..., 11$:
$\Rightarrow e \in \{1, 37, 73, 109, 145, 181, 217, 253, 289, 325, 361, 397\}$.

Ab $j = 12$ gilt: $e = 12 \cdot kgV(12, 36) + 1 = 432 + 1 = 433 > 432 = \phi(n)$.

Überprüfen wir z.B. wieder die obigen vier Werte für m mit $e = 217$, ergibt sich:
$4^{217} \mod 481 = 4$
$6^{217} \mod 481 = 6$
$7^{217} \mod 481 = 7$
$480^{217} \mod 481 = 480$

Satz 4.19.5. *Die Anzahl der möglichen Werte für e mit* $m^e = m \bmod n$ *lässt sich wie folgt berechnen:*

$$[Anzahl\ e] = \left\lfloor \frac{\phi(n)}{kgV(p-1, q-1) + 1} \right\rfloor + 1 = \frac{\phi(n)}{kgV(p-1, q-1)}$$

In unserem Beispiel ergibt dies $\frac{432}{kgV(12,36)} = 12$ verschiedene Werte für e, bei denen für alle m in $\mathbb{Z}_{481}$ gilt: $m^e = m \bmod n$.

[148] Man kann diese e, die jede Nachricht zu einem Fixpunkt machen, als „schwache Schlüssel" (e, n) des RSA-Verfahrens bezeichnen. Diese Bezeichnung unterscheidet sich jedoch von den „schwachen Schlüsseln" k bei DES, die **jede** Nachricht m bei doppelt hintereinander durchgeführter **Ver**schlüsselung auf sich selbst abbilden. Das RSA-Verfahren kennt m.W. für größere n keine schwachen Schlüssel dieser Art: $(m^e)^e = m$.
In JCT findet man schwache DES-Schlüssel in der Standard-Perspektive über den Menüeintrag **Visualisierungen \ Innere Zustände im Data Encryption Standard (DES)**.

4.19.7.4 Eine empirische Abschätzung der Anzahl der Fixpunkte für wachsende Moduli

In diesem Kapitel machen wir eine empirische Abschätzung der Anzahl der Fixpunkte für wachsende Moduli (und nicht schwache e).

Dabei haben wir p und q zufällig aus den sechs folgenden Bereichen gewählt (jeder gkennzeichnet durch seine untere und obere Schranke):
$(2^2, 2^{10}), (2^{10}, 2^{20}), (2^{20}, 2^{40}), (2^{40}, 2^{80}), (2^{80}, 2^{160}), (2^{160}, 2^{320})$.
In jedem Bereich haben wir 10 Versuche gemacht. Für den Exponenten e haben wir immer den Standardwert $e = 2^{16} + 1$ genommen. Die Anzahl der Fixpunkte wurde für alle 60 Versuche mit dem Programm 4.21 „Getfixpoints.sage“ berechnet.

Die folgenden fünf Mengen enthalten die zufällig gewählten Wertepaare (p,q) innerhalb der ersten fünf Größenbereiche.

$$\begin{aligned} Aus(2^2, 2^{10}) : (p, q) \ \in \{&(127, 947), (349, 809), (47, 461), (587, 151), (19, 23), \\ &(709, 509), (653, 11), (859, 523), (823, 811), (83, 331)\} \end{aligned}$$

$$\begin{aligned} Aus(2^{10}, 2^{20}) : (p, q) \ \in \{&(447401, 526283), (474223, 973757), (100829, 126757), (35803, 116933), \\ &(577751, 598783), (558121, 607337), (950233, 248167), (451103, 73009), \\ &(235787, 164429), (433267, 287939)\} \end{aligned}$$

$$\begin{aligned} Aus(2^{20}, 2^{40}) : (p, q) \ \in \{&(58569604997, 321367332149), (286573447351, 636576727223), \\ &(134703821971, 134220414529), (161234614601, 711682765579), \\ &(19367840881, 804790726361), (932891507377, 521129503333), \\ &(337186437739, 426034644493), (986529569219, 604515928397), \\ &(276825557171, 654134442649), (639276602353, 1069979301731)\} \end{aligned}$$

$$\begin{aligned} Aus(2^{40}, 2^{80}) : (p, q) \ \in \{&(667530919106151273090539, 287940270633610590682889), \\ &(437090557112369481760661, 590040807609821698387141), \\ &(113192118893748086305 4851, 813935599673320990215139) \\ &(874130181777177966406673, 632270193935624953596331), \\ &(599303355925474677078809, 717005631177936134003029), \\ &(752829320004631398659063, 714134510643836818718761), \\ &(104631331509274349291 7349, 835721729660755006973833), \\ &(877161707568112212806617, 428315033282611057936 49), \\ &(575464819450637793425803, 542583205115904343 3027), \\ &(321404337099945148592363, 992663778486687980443879)\} \end{aligned}$$

$$
\begin{aligned}
Aus(2^{80}, 2^{160}) : (p, q) \ \in \{&(838952969674957834783403492645269831354775774659,\\
&694309130163549038783972189350416942879771871411),\\
&(9819851072906295013741877488599617868043115646 43,\\
&178616495258601001174141825667078950281544628693),\\
&(614446632627716919862227545890890553330513965359,\\
&761232454374959264696945191327265643178491649141),\\
&(142175695272200809558594586396256042555470793633 7,\\
&986781711714138924140285492105143175328486228197),\\
&(862346475785474165539441761205023498091366178341,\\
&438589995804600940885415547506719456975478582911),\\
&(103408131889966934541660257403408124753805300153 3,\\
&120703277857143470461811129707277488474870622344 7),\\
&(308083812465705343620096534684980088954958466893,\\
&350597371862294596793629011464584694618569736021),\\
&(830376326124356299120963861338027196931951857769,\\
&924874232653136669722297184352059466357375363191),\\
&(856005811201545908101892375698207060066598292 31,\\
&297064381842806596646150718828138629443319259829),\\
&(135898449201351605205579012932458184759027590912 9,\\
&609402294805414245544586792657989060761523960427)\}
\end{aligned}
$$

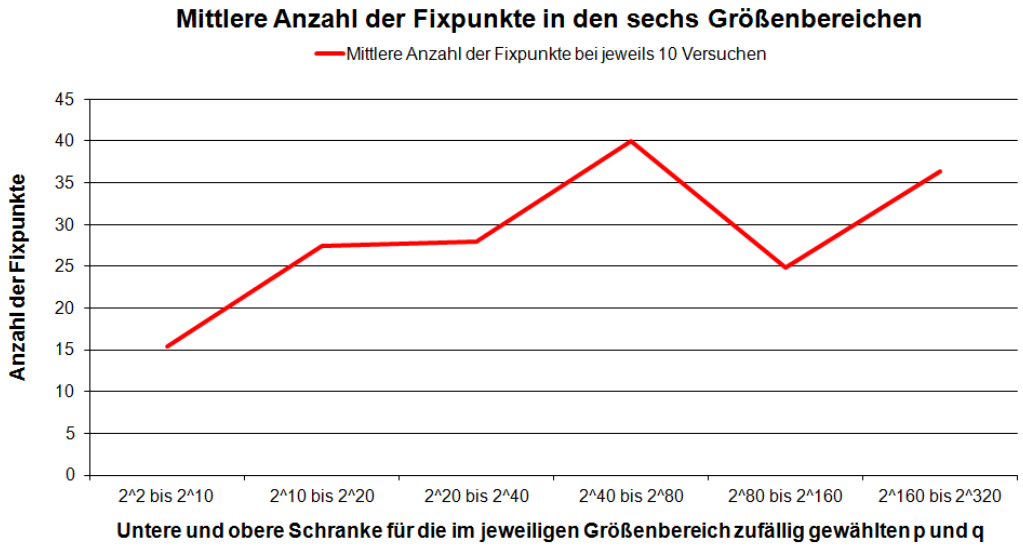


Abbildung 4.8: Eine empirische Abschätzung der Anzahl der Fixpunkte für wachsende Moduli

Abbildung 4.8 zeigt, dass innerhalb der sechs Größenbereiche die durchschnittliche Anzahl der Fixpunkte nicht größer als 40 ist.

4.19.7.5 Beispiel: Bestimmung aller Fixpunkte für einen bestimmten öffentlichen RSA-Schlüssel

Die Aufgabe besteht darin, alle Fixpunkte für (n, e) = (866959, 17) zu bestimmem.

Lösung:
Zuerst faktorisieren wir n: $866959 = 811 \cdot 1069$.

Die Anzahl der RSA-Fixpunkte ergibt sich nach Satz 4.19.3:
$(ggT(p-1, e-1)+1) \cdot (ggT(q-1, e-1)+1) = (ggT(811-1, 17-1)+1) \cdot (ggT(1069-1, 17-1)+1) = (2+1) \cdot (4+1) = 15$

Das SageMath-Programm 4.21 (Getfixpoints) liefert für $(n, e) = (866959, 17)$ die folgenden 15 Fixpunkte:

```
0       1       23518   23519   47037
188964  212482  236000  654477  843440
843441  630959  677995  819922  866958
```

Beispiel:
Beispielhaftes Validieren für 843441: $843441^{17} \mod 866959 = 843441$
Also ist $m = 843441$ ein Fixpunkt für das gegebene (n, e).

Bedeutung der Variablen im SageMath-Code 4.21:

```
- gen_f_p = r.multiplicative_generator()
  r ist ein Restklassen-Ring modulo p, und multiplicative_generator() gibt
  ein Generator-Element zurück, welches diesen Ring modulo p erzeugt.
- power_mod(gen_f_p,Integer(i*(p-1)/gcd_p),p)
  Die power_mod Funktion potenziert eine Zahl m mit e, und gibt das Ergebnis modulo n aus.
  Bsp.: power_mod(m, e, n) := m^e modulo n
- numpy.append(fp,power_mod(gen_f_p,Integer(i*(p-1)/gcd_p),p))
  Die append Funktion erweitert ein Array (fp) um ein weiteres Element.
- crt(Integer(r),Integer(s),Integer(p),Integer(q))
  CRT steht für Chinese Remainder Theorem. crt(r, s, p, q) löst die Kongruenzen
  x = r mod p und x = s mod q mit Hilfe des Chinesischen Restsatzes.
```

SageMath-Beispiel 4.21 Bestimmung aller Fixpunkt-Nachrichten für einen gegebenen öffentlichen RSA-Schlüssel

```
import numpy

print "--- Search for fixpoints in Textbook-RSA given p, q, e ---";
fp=numpy.array([0])
fq=numpy.array([0])

#Edit e,p,q here
###EDIT BEGIN###
e=17;
p=811;
q=1069;
###EDIT END###

n=p*q;
print "Prime p: ",p;
print "Prime q: ",q;
print "Modul n: ",n;
print "Public exponent e: ", e;

r=Integers(p)
gen_f_p = r.multiplicative_generator(); print "\nGenerator of f_p: ",gen_f_p;
s=Integers(q)
gen_f_q = s.multiplicative_generator(); print "Generator of f_q: ",gen_f_q;

gcd_p = gcd(e-1,p-1)
gcd_q = gcd(e-1,q-1)
print "\ngcd(e-1,p-1): ", gcd_p;
print "gcd(e-1,q-1): ", gcd_q;

print "\nNumber of fixpoints: ",(gcd_p+1)*(gcd_q+1);
#Calculating fixpoints modulo F_p
#run i from 0 until gcd(e-1,p-1):
#g^( i*(p-1) / (ggT(e-1,p-1)) ) mod p

print "\nFixpoints modulo p";
print "0 (trivial fixpoint added manually)";
i=0;
for i in range(gcd_p):
                fix_p = power_mod(gen_f_p,Integer(i*(p-1)/gcd_p),p); print fix_p;
                fp = numpy.append(fp,fix_p)

print "\nFixpoints modulo q";
print "0 (trivial fixpoint added manually)";
j=0;
for j in range(gcd_q):
                fix_q = power_mod(gen_f_q,Integer(j*(q-1)/gcd_q),q); print fix_q;
                fq = numpy.append(fq,fix_q);

print "\nFixpoints for the public RSA key (n,e) = (", n, ",", e, ")"
for r in fp:
       for s in fq:
               print crt(Integer(r),Integer(s),Integer(p),Integer(q))

print "\nRemark: You can verify each fixpoint with power_mod(m,e,n).";
```

4.20 Anhang: Liste der in diesem Kapitel formulierten Definitionen und Sätze

Literaturverzeichnis (Kap. NT)

[AKS02] Agrawal, M., N. Kayal und N. Saxena: *PRIMES in P*, August 2002. Corrected version.
`http://www.cse.iitk.ac.in/~manindra/algebra/primality_v6.pdf`,
`http://fatphil.org/maths/AKS/`.

[Bau95] Bauer, Friedrich L.: *Entzifferte Geheimnisse*. Springer, 1995.

[Bau00] Bauer, Friedrich L.: *Decrypted Secrets*. Springer, 2. Auflage, 2000.

[Ber01] Bernstein, Daniel J.: *Circuits for integer factorization: a proposal*.
`http://cr.yp.to/papers/nfscircuit.ps`,
`http://cr.yp.to/djb.html`, 2001.

[Ber05] Bernstein, Daniel J.: *Factoring into coprimes in essentially linear time*. Journal of Algorithms, 54, 2005. `http://cr.yp.to/lineartime/dcba-20040404.pdf`.

[Beu96] Beutelspacher, Albrecht: *Kryptologie*. Vieweg, 5. Auflage, 1996.

[BFT02] Bourseau, F., D. Fox und C. Thiel: *Vorzüge und Grenzen des RSA-Verfahrens*. Datenschutz und Datensicherheit (DuD), 26:84–89, 2002.
`http://www.secorvo.de/publikationen/rsa-grenzen-fox-2002.pdf`.

[BLP93] Buhler, J. P., H. W. Lenstra und C. Pomerance: *Factoring integers with the number field sieve*. In: Lenstra, K. und H.W. Lenstra (Herausgeber): *The Development of the Number Field Sieve, Lecture Notes in Mathematics, Vol. 1554*, Seiten 50–94. Springer, 1993.

[BSI16] BSI: *Angaben des BSI für die Algorithmenkataloge der Vorjahre, Empfehlungen zur Wahl der Schlüssellängen*. Technischer Bericht, BSI (Bundesamt für Sicherheit in der Informationstechnik), 2016.
`https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/ElektronischeSignatur/TechnischeRealisierung/Kryptoalgorithmen/kryptoalg.html`
- Vgl.: BNetzA (Bundesnetzagentur): Jährlich erscheinende Veröffentlichung zu Algorithmen und Parametern im Umfeld elektronischer Signaturen:
`http://www.bundesnetzagentur.de/cln_1411/DE/Service-Funktionen/ElektronischeVertrauensdienste/QES/WelcheAufgabenhatdieBundesnetzagentur/GeeigneteAlgorithmenfestlegen/geeignetealgorithmenfestlegen_node.html`
- Vgl.: Eine Stellungnahme zu diesen Empfehlungen:
`http://www.secorvo.de/publikationen/stellungnahme-algorithmenempfehlung-020307.pdf` .

[Eck14] Eckert, Claudia: *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. De Gruyter Oldenbourg, 9. Auflage, 2014. Paperback.

[ESS12] Esslinger, B., J. Schneider und V. Simon: *RSA – Sicherheit in der Praxis*. KES Zeitschrift für Informationssicherheit, 2012(2):22–27, April 2012. https://www.cryptool.org/images/ctp/documents/kes_2012_RSA_Sicherheit.pdf.

[GKP94] Graham, R. E., D. E. Knuth und O. Patashnik: *Concrete Mathemathics, a Foundation of Computer Science*. Addison Wesley, 6. Auflage, 1994.

[HDWH12] Heninger, Nadia, Zakir Durumeric, Eric Wustrow und J. Alex Halderman: *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices*. In: *Proceedings of the 21st USENIX Security Symposium*, August 2012. https://factorable.net/paper.html.

[Kat01] Katzenbeisser, Stefan: *Recent Advances in RSA Cryptography*. Springer, 2001.

[Kle10] Kleinjung, Thorsten et al.: *Factorization of a 768-bit RSA modulus, version 1.4*, 2010. http://eprint.iacr.org/2010/006.pdf.

[Knu98] Knuth, Donald E.: *The Art of Computer Programming, vol 2: Seminumerical Algorithms*. Addison-Wesley, 3. Auflage, 1998.

[LHA+12] Lenstra, Arjen K., James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung und Christophe Wachter: *Ron was wrong, Whit is right, A Sanity Check of Public Keys Collected on the Web*. Cryptology ePrint Archive, Februar 2012. http://eprint.iacr.org/2012/064.pdf.

[LL93] Lenstra, A. und H. Lenstra: *The development of the Number Field Sieve*. Lecture Notes in Mathematics 1554. Springer, 1993.

[LSTT02] Lenstra, Arjen K., Adi Shamir, Jim Tomlinson und Eran Tromer: *Analysis of Bernstein's Factorization Circuit*, 2002. http://tau.ac.il/~tromer/papers/meshc.pdf.

[LV01] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes (1999 + 2001)*. Journal of Cryptology, 14:255–293, 2001. http://www.cs.ru.nl/E.Verheul/papers/Joc2001/joc2001.pdf, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.69&rep=rep1&type=pdf.

[MvOV01] Menezes, Alfred J., Paul C. van Oorschot und Scott A. Vanstone: *Handbook of Applied Cryptography*. Series on Discrete Mathematics and Its Application. CRC Press, 5. Auflage, 2001, ISBN 0-8493-8523-7. (Errata last update Jan 22, 2014). http://cacr.uwaterloo.ca/hac/, http://www.cacr.math.uwaterloo.ca/hac/.

[Ngu09] Nguyen, Minh Van: *Number Theory and the RSA Public Key Cryptosystem – An introductory tutorial on using SageMath to study elementary number theory and public key cryptography*, 2009. http://faculty.washington.edu/moishe/hanoiex/Number%20Theory%20Applications/numtheory-crypto.pdf.

[Oec03] Oechslin, Philippe: *Making a Faster Cryptanalytic Time-Memory Trade-Off.* Technischer Bericht, Crypto 2003, 2003.
http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf.

[Pfl97] Pfleeger, Charles P.: *Security in Computing.* Prentice-Hall, 2. Auflage, 1997.

[Pom84] Pomerance, Carl: *The Quadratic Sieve Factoring Algorithm.* In: Blakley, G.R. und D. Chaum (Herausgeber): *Proceedings of Crypto '84, LNCS 196*, Seiten 169–182. Springer, 1984.

[Sch96] Schneier, Bruce: *Applied Cryptography, Protocols, Algorithms, and Source Code in C.* Wiley, 2. Auflage, 1996.

[Sch04] Schneider, Matthias: *Analyse der Sicherheit des RSA-Algorithmus. Mögliche Angriffe, deren Einfluss auf sichere Implementierungen und ökonomische Konsequenzen.* Diplomarbeit, Universität Siegen, 2004.

[Sec02] Security, RSA: *Has the RSA algorithm been compromised as a result of Bernstein's Paper?* Technischer Bericht, RSA Security, April 2002. http://www.emc.com/emc-plus/rsa-labs/historical/has-the-rsa-algorithm-been-compromised.htm.

[Sed90] Sedgewick, Robert: *Algorithms in C.* Addison-Wesley, 1990.

[Sil00] Silverman, Robert D.: *A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths.* RSA Laboratories Bulletin, 13:1–22, April 2000.

[ST03a] Shamir, Adi und Eran Tromer: *Factoring Large Numbers with the TWIRL Device*, 2003. http://www.tau.ac.il/~tromer/papers/twirl.pdf.

[ST03b] Shamir, Adi und Eran Tromer: *On the Cost of Factoring RSA-1024.* RSA Laboratories CryptoBytes, 6(2):11–20, 2003.
http://www.tau.ac.il/~tromer/papers/cbtwirl.pdf.

[Sti06] Stinson, Douglas R.: *Cryptography – Theory and Practice.* Chapman & Hall/CRC, 3. Auflage, 2006.

[SW10] Schulz, Ralph Hardo und Helmut Witten: *Zeitexperimente zur Faktorisierung. Ein Beitrag zur Didaktik der Kryptographie.* LOG IN, 166/167:113–120, 2010.
http://bscw.schule.de/pub/bscw.cgi/d864899/Schulz_Witten_Zeit-Experimente.pdf.

[Wil95] Wiles, Andrew: *Modular elliptic curves and fermat's last theorem.* Annals of Mathematics, 141, 1995.

[WLB03] Weis, Rüdiger, Stefan Lucks und Andreas Bogk: *Sicherheit von 1024 bit RSA-Schlüsseln gefährdet.* Datenschutz und Datensicherheit (DuD), 27(6):360–362, 2003.

[WS06] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 2: RSA für große Zahlen.* LOG IN, 2006(143):50–58, 2006.
http://bscw.schule.de/pub/bscw.cgi/d404410/RSA_u_Co_NF2.pdf.

[WS08] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 3: RSA und die elementare Zahlentheorie.* LOG IN, 2008(152):60–70, 2008.
http://bscw.schule.de/pub/nj_bscw.cgi/d533821/RSA_u_Co_NF3.pdf.

[WSE15] Witten, Helmut, Ralph Hardo Schulz und Bernhard Esslinger: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle, NF Teil 7: Alternativen zu RSA oder Diskreter Logarithmus statt Faktorisierung.* LOG IN, 2010(181-182):85–102, 2015.
Hierin werden u.a. DH und Elgamal in einem breiteren Kontext behandelt. Die Verfahren werden mit Codebeispielen in Python und SageMath erläutert.
`http://bscw.schule.de/pub/nj_bscw.cgi/d1024013/RSA_u_Co_NF7.pdf`,
`http://www.log-in-verlag.de/wp-content/uploads/2015/07/Internetquellen-LOG-IN-Heft-Nr.181-182.doc`.

[Yan00] Yan, Song Y.: *Number Theory for Computing.* Springer, 2000.

Alle Links wurden am 12.07.2016 überprüft.

Web-Links

1. Fibonacci-Seite von Ron Knott
   Hier dreht sich alles um Fibonacci-Zahlen.
   `http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fib.html`

2. CrypTool
   E-Learning-Freeware zur Veranschaulichung von Kryptographie und Kryptoanalyse
   `http://www.cryptool.de`,
   `http://www.cryptool.org`

3. Mathematica
   Kommerzielles Mathematik-Paket
   `http://www.wolfram.com`

4. LiDIA
   Umfangreiche Bibliothek mit zahlentheoretischen Funktionen und der Interpretersprache LC. LiDIA wird seit 2000 nicht weiter entwickelt.
   `http://cs.nyu.edu/exact/core/download/core_v1.3/core_v1.3/lidia/`

5. BC
   Interpreter mit zahlentheoretischen Funktionen. Wird leider seit 2006 nicht weiter entwickelt und funktioniert unter neueren Windows-Versionen auch nicht mehr korrekt.
   Von Keith Matthews gibt es für Unix aber regelmäßig keine Updates.
   `http://www.gnu.org/software/bc/bc.html`
   `http://www.numbertheory.org/gnubc/gnubc.html`

6. Pari-GP
   Schneller und freier Interpreter mit zahlentheoretischen Funktionen. Auch im Browser. Gepflegt von Karim Belabas.
   `http://pari.math.u-bordeaux.fr/`
   `http://pari.math.u-bordeaux.fr/gp.html`
   `http://en.wikipedia.org/wiki/PARI/GP`

7. SageMath
   Ausgezeichnetes Open-Source Computer-Algebra-System, basierend auf Python als Skript-Sprache. Damit sind die Code-Beispiele in diesem Kapitel erstellt. Vergleiche hierzu die Einführung in Kapitel A.7.
   `http://www.sagemath.org/`
   `https://de.wikipedia.org/wiki/Sage_(Software)`

8. Münchenbach
   Erst nach Vollendung der ersten Versionen dieses Artikels wurde mir die Web-Seite von Herrn Münchenbach (1999) bekannt, die interaktiv und didaktisch sehr ausgereift die grundlegenden Denkweisen der Mathematik anhand der elementaren Zahlentheorie nahebringt. Sie entstand für ein Unterrichtsprojekt der 11. Klasse des Technischen Gymnasiums und nutzt MuPAD (leider nur in Deutsch verfügbar).
   `http://www.hydrargyrum.de/kryptographie`

9. Wagner
   Seite des Beauftragten für die Lehrplanentwicklung des Fachs Informatik an der gymnasialen Oberstufe des Landes Saarland aus 2012. Hier befindet sich eine Sammlung von

Texten und Programmen (in Java), die aus didaktischen Überlegungen entstand (alles nur in Deutsch verfügbar).
`http://www.saar.de/~awa/kryptologie.html`
`http://www.saar.de/~awa/menu_kryptologie.html`

10. BSI
    Bundesamt für Sicherheit in der Informationstechnik
    `http://www.bsi.bund.de`

11. Faktorisierungsrekorde und Challenges
    `https://web.archive.org/web/20170704003418/http://www.crypto-world.com:80/`, letztes Update 2013
    `https://web.archive.org/web/20170518021747/http://www.crypto-world.com:80/FactorWorld.html`, Webseite von Scott Contini
    `http://www.loria.fr/~zimmerma/records/factor.html`
    `http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm`
    `https://www.uni-bonn.de/Pressemitteilungen/004-2010`, zu RSA-768
    `https://members.loria.fr/PZimmermann/records/gnfs158`, zu C158
    `http://www.loria.fr/~zimmerma/records/rsa160`

12. Das Cunningham-Projekt
    `http://www.cerias.purdue.edu/homes/ssw/cun/`, letzte Änderung 11.05.2016

13. Daniel J. Bernsteins Webseiten
    `http://cr.yp.to`

14. Post-Quantum-Kryptographie
    `http://pqcrypto.org/`

Alle Links wurden am 12.07.2016 überprüft.

Dank

Ich möchte hier die Gelegenheit nutzen, den folgenden Personen ganz herzlich zu danken:

- Hr. Henrik Koy für das anregende und sehr konstruktive Korrekturlesen und die vielen Verbesserungen der ersten Version dieses Artikels, und für die Hilfen bei TeX, ohne die dieser Artikel nie in dieser Form erschienen wäre.
- Jörg Cornelius Schneider für die engagierte Unterstützung bei TeX und die mannigfaltigen Hilfen bei allen Arten von Programmier- und Design-Problemen.
- Dr. Georg Illies für den Hinweis auf Pari-GP.
- Lars Fischer für seine Hilfe bei schnellem Pari-GP-Code für Primitivwurzeln.
- Minh Van Nguyen aus Australien für seine immer schnelle, kompetente und ausführliche Hilfe bei den ersten SageMath-Code-Beispielen in diesem Kapitel. Schade, dass er nicht mehr erreichbar ist ...

Kapitel 5

Die mathematischen Ideen hinter der modernen Kryptographie[1]

(Roger Oyono / Bernhard Esslinger / Jörg-Cornelius Schneider, Sep. 2000; Updates: Nov. 2000, Feb. 2003, Apr. 2007, März 2010, Jan. 2013)

> Ich weiß nicht, ob es besser wird, wenn wir es ändern,
> aber ich weiß, dass wir es ändern müssen, wenn es besser werden soll.
> *Anmerkung von Unbekannt (Radio) dazu:*
> Und Gott gebe den Akteuren bei der notwendigen Umsetzung
> das Wissen, die Weisheit und das Verantwortungsbewusstsein,
> zwischen Aktionismus, Selbstdarstellung und planvollem Handeln
> zu unterscheiden – und ihr Wissen auch anzuwenden.

Zitat 15: Georg Christoph Lichtenberg[2]

5.1 Einwegfunktionen mit Falltür und Komplexitätsklassen

Eine **Einwegfunktion** ist eine effizient zu berechnende Funktion, deren Umkehrung jedoch nur mit extrem hohem Rechenaufwand – jedoch praktisch unmöglich – zu berechnen ist.

Etwas genauer formuliert: Eine Einwegfunktion ist eine Abbildung f einer Menge X in eine Menge Y, so dass $f(x)$ für jedes Element x von X leicht zu berechnen ist, während es für (fast) jedes y aus Y praktisch unmöglich ist, ein Urbild x (d.h. ein x mit $f(x) = y$) zu finden.

Ein alltägliches Beispiel für eine Einwegfunktion ist ein Telefonbuch: die auszuführende Funktion ist die, einem Namen die entsprechende Telefonnummer zuzuordnen. Da die Namen alphabetisch geordnet sind, ist diese Zuordnung einfach auszuführen. Aber ihre Invertierung, also die Zuordnung eines Namens zu einer gegebenen Nummer, ist offensichtlich schwierig, wenn man nur ein Telefonbuch zur Verfügung hat.

[1] Mit dem Lernprogramm **ZT** können Sie spielerisch einige der hier besprochenen Verfahren (RSA, Rabin, DH, ElGamal) nachvollziehen (siehe Lern-Kapitel 4.2 und 4.3, Seite 9-17/17).
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.
Entsprechende Funktionen finden Sie auch in den Programmen CT1, CT2 und JCT: siehe die Liste der darin enthaltenen Funktionen in Anhang A.1, A.2 und A.3.

[2] Georg Christoph Lichtenberg, deutscher Schriftsteller und Physiker (1742-1799),
(siehe auch: `http://de.wikipedia.org/wiki/Georg_Christoph_Lichtenberg`)

Einwegfunktionen spielen in der Kryptographie eine entscheidende Rolle. Fast alle kryptographischen Begriffe kann man durch Verwendung des Begriffs Einwegfunktion umformulieren. Als Beispiel betrachten wir die Public-Key-Verschlüsselung (asymmetrische Kryptographie):

Jedem Teilnehmer T des Systems wird ein privater Schlüssel d_T und ein sogenannter öffentlicher Schlüssel e_T zugeordnet. Dabei muss die folgende Eigenschaft (Public-Key-Eigenschaft) gelten:
Für einen Gegner, der den öffentlichen Schlüssel e_T kennt, ist es praktisch unmöglich, den privaten Schlüssel d_T zu bestimmen.

Zur Konstruktion nützlicher Public-Key-Verfahren sucht man also eine Einwegfunktion, die in einer Richtung „einfach“ zu berechnen, die in der anderen Richtung jedoch „schwer“ (praktisch unmöglich) zu berechnen ist, solange eine bestimmte zusätzliche Information (Falltür) nicht zur Verfügung steht. Mit der zusätzlichen Information kann die Umkehrung effizient gelöst werden. Solche Funktionen nennt man **Einwegfunktionen mit Falltür** (trapdoor one-way function). Im obigen Fall ist die Einwegfunktion die Verschlüsselung durch Potenzieren mit dem öffentlichen Schlüssel e_T (als Exponent). Der geheime Schlüssel d_T ist die Falltür-Information.

Dabei bezeichnet man ein Problem als „einfach“, wenn es in polynomialer Zeit als Funktion der Länge der Eingabe lösbar ist, d.h. wenn es so gelöst werden kann, dass der Zeitaufwand sich als polynomiale Funktion in Abhängigkeit der Länge der Eingabe darstellen lässt. Wenn die Länge der Eingabe n Bits beträgt, so ist die Zeit der Berechnung der Funktion proportional zu n^a, wobei a eine Konstante ist. Man sagt, dass die Komplexität solcher Probleme $O(n^a)$ beträgt (Landau- oder Big-O-Notation).

Vergleicht man 2 Funktionen 2^n und n^a, wobei a eine Konstante ist, dann gibt es immer einen Wert für n, ab dem für alle weiteren n gilt: $n^a < 2^n$. Die Funktion n^a hat eine geringere Komplexität. Z.B. für $a = 5$ gilt: ab der Länge $n = 23$ ist $2^n > n^5$ und danach wächst 2^n auch deutlich schneller $[(2^{22} = 4.194.304,\ 22^5 = 5.153.632),\ (2^{23} = 8.388.608,\ 23^5 = 6.436.343),\ (2^{24} = 16.777.216,\ 24^5 = 7.962.624)]$.

Der Begriff „praktisch unmöglich“ ist etwas schwammiger. Allgemein kann man sagen, ein Problem ist nicht effizient lösbar, wenn der zu seiner Lösung benötigte Aufwand schneller wächst als die polynomiale Zeit als Funktion der Größe der Eingabe. Wenn beispielsweise die Länge der Eingabe n Bits beträgt und die Zeit zur Berechnung der Funktion proportional zu 2^n ist, so gilt gegenwärtig: die Funktion ist für $n > 80$ praktisch nicht zu berechnen.

Die Entwicklung eines praktisch einsetzbaren Public-Key-Verfahrens hängt daher von der Entdeckung einer geeigneten Einwegfunktion mit Falltür ab.

Um Ordnung in die verwirrende Vielfalt von möglichen Problemen und ihre Komplexitäten zu bringen, fasst man Probleme mit ähnlicher Komplexität zu Klassen zusammen.

Die wichtigsten Komplexitätsklassen sind die Klassen **P** und **NP**:

- Die Klasse **P**: Zu dieser Klasse gehören diejenigen Probleme, die mit polynomialem Zeitaufwand lösbar sind.
- Die Klasse **NP**: Bei der Definition dieser Klasse betrachten wir nicht den Aufwand zur Lösung eines Problems, sondern den Aufwand zur Verifizierung einer gegebenen Lösung. Die Klasse **NP** besteht aus denjenigen Problemen, bei denen die Verifizierung einer gegebenen Lösung mit polynomialem Zeitaufwand möglich ist. Dabei bedeutet der Begriff **NP** „nichtdeterministisch“ polynomial und bezieht sich auf ein Berechnungsmodell, d.h. auf einen nur in der Theorie existierenden Computer, der richtige Lösungen nichtdeterministisch „raten“ und dies dann in polynomialer Zeit verifizieren kann.

Die Klasse **P** ist in der Klasse **NP** enthalten. Ein berühmtes offenes Problem ist die Frage, ob $\mathbf{P} \neq \mathbf{NP}$ gilt oder nicht, d.h. ob **P** eine echte Teilmenge ist oder nicht. Eine wichtige Eigenschaft der Klasse **NP** ist, dass sie auch sogenannte **NP**-vollständige Probleme enthält. Dies sind Probleme, welche die Klasse **NP** in folgender Weise vollständig repräsentieren: Wenn es einen „guten" Algorithmus für ein solches Problem gibt, dann existieren für alle Probleme aus **NP** „gute" Algorithmen. Insbesondere gilt: Wenn auch nur ein vollständiges Problem in **P** läge, d.h. wenn es einen polynomialen Lösungsalgorithmus für dieses Problem gäbe, so wäre **P**=**NP**. In diesem Sinn sind die **NP**-vollständigen Probleme die schwierigsten Probleme in **NP**.

Viele kryptographische Protokolle sind so gemacht, dass die „guten" Teilnehmer nur Probleme aus **P** lösen müssen, während sich ein Angreifer vor Probleme aus **NP** gestellt sieht.

Man weiß leider bis heute nicht, ob es Einwegfunktionen überhaupt gibt. Man kann aber zeigen, dass Einwegfunktionen genau dann existieren, wenn $\mathbf{P} \neq \mathbf{NP}$ gilt [BDG98, S.63].

Immer wieder behauptete jemand, er habe die Äquivalenz bewiesen (siehe [Hes01]), aber bisher erwiesen sich diese Aussagen stets als falsch.

Es wurden eine Reihe von Algorithmen für Public-Key-Verfahren vorgeschlagen. Einige davon erwiesen sich, obwohl sie zunächst vielversprechend erschienen, als polynomial lösbar. Der berühmteste durchgefallene Bewerber ist der von Ralph Merkle [MH78] vorgeschlagene Knapsack mit Falltür.

5.2 Knapsackproblem als Basis für Public-Key-Verfahren

5.2.1 Knapsackproblem

Gegeben n Gegenstände $G_1, \ldots, G_n$ mit den Gewichten $g_1, \ldots g_n$ und den Werten $w_1, \cdots, w_n$. Man soll wertmäßig so viel wie möglich unter Beachtung einer oberen Gewichtsschranke g davontragen. Gesucht ist also eine Teilmenge von $\{G_1, \cdots, G_n\}$, etwa $\{G_{i_1}, \ldots, G_{i_k}\}$, so dass $w_{i_1} + \cdots + w_{i_k}$ maximal wird unter der Bedingung $g_{i_1} + \cdots + g_{i_k} \leq g$.

Derartige Fragen gehören zu den **NP**-vollständige Problemen (nicht deterministisch polynomial), die aufwendig zu berechnen sind.

Ein Spezialfall des Knapsackproblems ist:
Gegeben sind die natürlichen Zahlen $a_1, \ldots, a_n$ und g. Gesucht sind $x_1, \ldots, x_n \in \{0, 1\}$ mit $g = \sum_{i=1}^{n} x_i a_i$ (wo also $g_i = a_i = w_i$ gewählt ist). Dieses Problem heißt auch **0-1-Knapsackproblem** und wird mit $K(a_1, \ldots, a_n; g)$ bezeichnet.

Zwei 0-1-Knapsackprobleme $K(a_1, \ldots, a_n; g)$ und $K(a'_1, \ldots, a'_n; g')$ heißen kongruent, falls es zwei teilerfremde Zahlen w und m gibt, so dass

1. $m > \max\{\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} a'_i\}$,
2. $g \equiv wg' \mod m$,
3. $a_i \equiv wa'_i \mod m$ für alle $i = 1, \ldots, n$.

Bemerkung:
Kongruente 0-1-Knapsackprobleme haben dieselben Lösungen. Ein schneller Algorithmus zur Klärung der Frage, ob zwei 0-1-Knapsackprobleme kongruent sind, ist nicht bekannt.

Das Lösen eines 0-1-Knapsackproblems kann durch Probieren der 2^n Möglichkeiten für $x_1, \ldots, x_n$ erfolgen. Die beste Methode erfordert $O(2^{n/2})$ Operationen, was für $n = 100$ mit $2^{100} \approx 1,27 \cdot 10^{30}$ und $2^{n/2} \approx 1,13 \cdot 10^{15}$ für Computer eine unüberwindbare Hürde darstellt. Allerdings ist die Lösung für spezielle $a_1, \ldots, a_n$ recht einfach zu finden, etwa für $a_i = 2^{i-1}$. Die binäre Darstellung von g liefert unmittelbar $x_1, \ldots, x_n$. Allgemein ist die Lösung des 0-1-Knapsackproblems leicht zu finden, falls eine Permutation[3] π von $1, \ldots, n$ mit $a_{\pi(j)} > \sum_{i=1}^{j-1} a_{\pi(i)}$ mit $j = 1, \ldots, n$ existiert. Ist zusätzlich π die Identität, d.h. $\pi(i) = i$ für $i = 1, 2, \ldots, n$, so heißt die Folge $a_1, \ldots, a_n$ superwachsend. Das Verfahren 5.1 löst das Knapsackproblem mit superwachsender Folge im Zeitraum von $O(n)$.

Krypto-Verfahren 5.1 Lösen von Knapsackproblemen mit superwachsenden Gewichten

for $i = n$ **to** 1 **do**
  if $T \geq a_i$ **then**
    $T := T - s_i$
    $x_i := 1$
  else
    $x_i := 0$
if $T = 0$ **then**
  $X := (x_1, \ldots, x_n)$ ist die Lösung.
else
  Es gibt keine Lösung.

5.2.2 Merkle-Hellman Knapsack-Verschlüsselung

1978 gaben Merkle und Hellman [MH78] ein Public-Key-Verschlüsselungs-Verfahren an, das darauf beruht, das leichte 0-1-Knapsackproblem mit einer superwachsenden Folge in ein kongruentes mit einer nicht superwachsenden Folge zu „verfremden“. Es ist eine Blockchiffrierung, die bei jedem Durchgang einen n Bit langen Klartext chiffriert, siehe Krypto-Verfahren 5.2.

1982 gab Shamir [Sha82] einen Algorithmus zum Brechen des Systems in polynomialer Zeit an, ohne das allgemeine Knapsackproblem zu lösen. Len Adleman [Adl83] und Jeff Lagarias [Lag83] gaben einen Algorithmus zum Brechen des 2-fachen iterierten Merkle-Hellman Knapsack-Verschlüsselungsverfahrens in polynomialer Zeit an. Ernst Brickell [Bri85] gab schließlich einen Algorithmus zum Brechen des mehrfachen iterierten Merkle-Hellman Knapsack-Verschlüsselungsverfahren in polynomialer Zeit an. Damit war dieses Verfahren als Verschlüsselungsverfahren ungeeignet. Dieses Verfahren liefert also eine Einwegfunktion, deren Falltür-Information (Verfremden des 0-1-Knapsackproblems) durch einen Gegner entdeckt werden könnte.

[3] Eine Permutation π der Zahlen $1, \ldots, n$ ist die Vertauschung der Reihenfolge, in der diese Zahlen aufgezählt werden. Beispielsweise ist eine Permutation π von $(1, 2, 3)$ gleich $(3, 1, 2)$, also $\pi(1) = 3$, $\pi(2) = 1$ und $\pi(3) = 2$.

Krypto-Verfahren 5.2 Merkle-Hellman (auf Knapsackproblemen basierend)

Es sei $(a_1, \ldots, a_n)$ superwachsend. Seien m und w zwei teilerfremde Zahlen mit $m > \sum_{i=1}^{n} a_i$ und $1 \leq w \leq m-1$. Wähle $\bar{w}$ mit $w\bar{w} \equiv 1 \mod m$ die modulare Inverse von w und setze $b_i := wa_i \mod m$, $0 \leq b_i < m$ für $i = 1, \ldots, n$, und prüfe, ob die Folge $b_1, \ldots b_n$ nicht superwachsend ist. Danach wird eine Permutation $b_{\pi(1)}, \ldots, b_{\pi(n)}$ von $b_1, \ldots, b_n$ publiziert und insgeheim die zu π inverse Permutation μ festgehalten. Ein Sender schreibt seine Nachricht in Blöcke $(x_1^{(j)}, \ldots, x_n^{(j)})$ von Binärzahlen der Länge n und bildet

$$g^{(j)} := \sum_{i=1}^{n} x_i^{(j)} b_{\pi(i)}$$

und sendet $g^{(j)}, (j = 1, 2, \ldots)$.
Der Schlüsselinhaber bildet

$$G^{(j)} := \bar{w} g^{(j)} \mod m, \quad 0 \leq G^{(j)} < m$$

und verschafft sich die $x_{\mu(i)}^{(j)} \in \{0, 1\}$ (und somit auch die $x_i^{(j)}$) aus

$$\begin{aligned} G^{(j)} &\equiv \bar{w} g^{(j)} = \sum_{i=1}^{n} x_i^{(j)} b_{\pi(i)} \bar{w} \equiv \sum_{i=1}^{n} x_i^{(j)} a_{\pi(i)} \mod m \\ &= \sum_{i=1}^{n} x_{\mu(i)}^{(j)} a_{\pi(\mu(i))} = \sum_{i=1}^{n} x_{\mu(i)}^{(j)} a_i \mod m, \end{aligned}$$

indem er die leichten 0-1-Knapsackprobleme $K(a_1, \ldots, a_n; G^{(j)})$ mit superwachsender Folge $a_1, \ldots, a_n$ löst.

5.3 Primfaktorzerlegung als Basis für Public-Key-Verfahren

Primzahlen sind Grundlage für eine große Anzahl von Algorithmen für Public-Key-Verfahren.

5.3.1 Das RSA-Verfahren[4,5]

Bereits 1978 stellten R. Rivest, A. Shamir, L. Adleman [RSA78] das bis heute wichtigste asymmetrische Kryptographie-Verfahren vor (Krypto-Verfahren 5.3).

Krypto-Verfahren 5.3 RSA (auf dem Faktorisierungsproblem basierend)

Schlüsselgenerierung:
Seien p und q zwei verschiedene Primzahlen und $N = pq$. Sei e eine frei wählbare, zu $\phi(N)$ relativ prime Zahl , d.h. $\text{ggT}(e, \phi(N)) = 1$. Mit dem Euklidschen Algorithmus berechnet man die natürliche Zahl $d < \phi(N)$, so dass gilt

$$ed \equiv 1 \mod \phi(N).$$

Dabei ist ϕ die **Eulersche Phi-Funktion**.
Der Ausgangstext wird in Blöcke zerlegt und verschlüsselt, wobei jeder Block einen binären Wert $x^{(j)} \leq N$ hat.

Öffentlicher Schlüssel:

$$N, e.$$

Privater Schlüssel:

$$d.$$

Verschlüsselung:

$$y = e_T(x) = x^e \mod N.$$

Entschlüsselung:

$$d_T(y) = y^d \mod N$$

---

[4] Vergleiche auch die Kapitel 4.10, ff.
[5] Mit CT1 können Sie praktische Erfahrungen mit dem RSA-Verfahren sammeln: per Menü **Einzelverfahren \ RSA-Kryptosystem \ RSA-Demo**.
RSA finden Sie ebenfalls in CT2 und JCT.

Bemerkung: Eulersche Phi-Funktion

Die Eulersche Phi-Funktion ist definiert duch:

$\phi(n)$ ist die Anzahl der natürlichen Zahlen $x < n$, die mit n keinen gemeinsamen Faktor haben.

Kein gemeinsamer Faktor bedeutet: Zwei natürliche Zahlen a und b sind teilerfremd, falls $\text{ggT}(a, b) = 1$.

Für die Eulersche Phi-Funktion gilt:

$$\phi(1) = 1, \quad \phi(2) = 1, \quad \phi(3) = 2, \quad \phi(4) = 2, \quad \phi(6) = 2, \quad \phi(10) = 4, \quad \phi(15) = 8.$$

Zum Beispiel ist $\phi(24) = 8$, weil

$$|\{x < 24 : \text{ggT}(x, 24) = 1\}| = |\{1, 5, 7, 11, 13, 17, 19, 23\}|.$$

Tabelle 5.1 zeigt Werte von $\phi(n)$ bis 25.

n	$\phi(n)$	Die zu n teilerfremden natürlichen Zahlen kleiner n.
1	1	1
2	1	1
3	2	1, 2
4	2	1, 3
5	4	1, 2, 3, 4
6	2	1, 5
7	6	1, 2, 3, 4, 5, 6
8	4	1, 3, 5, 7
9	6	1, 2, 4, 5, 7, 8
10	4	1, 3, 7, 9
15	8	1, 2, 4, 7, 8, 11, 13, 14
20	8	1, 3, 7, 9, 11, 13, 17, 19
25	20	1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24

Tabelle 5.1: Eulersche Phi-Funktion

Ist p eine Primzahl, so gilt $\phi(p) = p - 1$.

Im Spezialfall $N = pq$ gilt:

$$\phi(N) = pq(1 - 1/p)(1 - 1/q) = p(1 - 1/p)q(1 - 1/q) = (p - 1)(q - 1).$$

Kennt man die verschiedenen Primfaktoren $p_1, \ldots, p_k$ von n, so ist

$$\phi(n) = n \cdot (1 - \frac{1}{p_1}) \cdots (1 - \frac{1}{p_k}).^{6}$$

[6] Weitere Formeln zur Eulerschen Phi-Funktion finden sich in Kapitel 4.8.2 „Die Eulersche Phi-Funktion“.

Die Funktion e_T ist eine Einwegfunktion, deren Falltür-Information die Primfaktorzerlegung von N ist.

Zur Zeit ist kein Algorithmus bekannt, der das Produkt zweier Primzahlen bei sehr großen Werten geeignet schnell zerlegen kann (z.B. bei mehreren hundert Dezimalstellen). Die heute schnellsten bekannten Algorithmen [Sti06] zerlegen eine zusammengesetzte ganze Zahl N in einem Zeitraum proportional zu $L(N) = e^{\sqrt{\ln(N)\ln(\ln(N))}}$. Einige Beispielwerte finden sich in Tabelle 5.2.

N	10^{50}	10^{100}	10^{150}	10^{200}	10^{250}	10^{300}
$L(N)$	$1,42 \cdot 10^{10}$	$2,34 \cdot 10^{15}$	$3,26 \cdot 10^{19}$	$1,20 \cdot 10^{23}$	$1,86 \cdot 10^{26}$	$1,53 \cdot 10^{29}$

Tabelle 5.2: Wertetabelle für $L(N)$ [Faktorisierungsaufwand bezogen auf die Modullänge]

Bewiesen ist bis heute nicht, dass das Problem, RSA zu brechen, äquivalent zum Faktorisierungsproblem ist. Es ist aber klar, dass wenn das Faktorisierungsproblem „gelöst“ ist, dass dann das RSA-Verfahren nicht mehr sicher ist.[7]

5.3.2 Rabin-Public-Key-Verfahren (1979)

Für dieses Verfahren (5.4) konnte gezeigt werden, dass es äquivalent zum Brechen des Faktorisierungsproblems ist. Leider ist dieses Verfahren anfällig gegen Chosen-Ciphertext-Angriffe.

Krypto-Verfahren 5.4 Rabin (auf dem Faktorisierungsproblem basierend)

Seien p und q zwei verschiedene Primzahlen mit $p, q \equiv 3 \mod 4$ und $n = pq$. Sei $0 \leq B \leq n-1$.

Öffentlicher Schlüssel:

$$e = (n, B).$$

Privater Schlüssel:

$$d = (p, q).$$

Verschlüsselung:

$$y = e_T(x) = x(x + B) \mod n.$$

Entschlüsselung:

$$d_T(y) = \sqrt{y + B^2/4} - B/2 \mod n.$$

Vorsicht: Wegen $p, q \equiv 3 \mod 4$ ist die Verschlüsselung (mit Kenntnis des Schlüssels) leicht zu berechnen. Dies ist nicht der Fall für $p \equiv 1 \mod 4$. Außerdem ist die Verschlüsselungsfunktion nicht injektiv: Es gibt genau vier verschiedene Quellcodes, die $e_T(x)$ als Urbild besitzen $x, -x - B, \omega(x + B/2) - B/2, -\omega(x + B/2) - B/2$, dabei ist ω eine der vier Einheitswurzeln. Es

[7] Im Jahre 2000 waren die Autoren der Ansicht, dass Werte der Größenordnung 100 bis 200 Dezimalstellen sicher sind. Sie schätzten, dass mit der aktuellen Computertechnik eine Zahl mit 100 Dezimalstellen bei vertretbaren Kosten in etwa zwei Wochen zerlegt werden könnte, dass mit einer teuren Konfiguration (z.B. im Bereich von 10 Millionen US-Dollar) eine Zahl mit 150 Dezimalstellen in etwa einem Jahr zerlegt werden könnte und dass eine 200-stellige Zahl noch für eine sehr lange Zeit unzerlegbar bleiben dürfte, falls es zu keinem mathematischen Durchbruch kommt. Dass es aber nicht doch schon morgen zu einem mathematischen Durchbruch kommt, kann man nie ausschließen.
Wie leicht man sich verschätzen kann, zeigt die Faktorisierung von RSA-200 (siehe Kapitel 4.11.4) – ganz ohne „mathematische Durchbrüche“.

muss also eine Redundanz der Quellcodes geben, damit die Entschlüsselung trotzdem eindeutig bleibt!

Hintertür-Information ist die Primfaktorzerlegung von $n = pq$.

5.4 Der diskrete Logarithmus als Basis für Public-Key-Verfahren[8]

Diskrete Logarithmen sind die Grundlage für eine große Anzahl von Algorithmen für Public-Key-Verfahren.

5.4.1 Der diskrete Logarithmus in $\mathbb{Z}_p^*$

Sei p eine Primzahl, und sei g ein Erzeuger der zyklischen multiplikativen Gruppe $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$. Dann ist die diskrete Exponentialfunktion zur Basis g definiert durch

$$e_g : k \longrightarrow y := g^k \mod p, \quad 1 \leq k \leq p-1.$$

Die Umkehrfunktion wird diskrete Logarithmusfunktion $\log_g$ genannt; es gilt

$$\log_g(g^k) = k.$$

Unter dem Problem des diskreten Logarithmus (in $\mathbb{Z}_p^*$) versteht man das folgende:

Gegeben p, g (ein Erzeuger der Gruppe $\mathbb{Z}_p^*$) und y, bestimme k so, dass $y = g^k \mod p$ gilt.

Die Berechnung des diskreten Logarithmus ist viel schwieriger als die Auswertung der diskreten Exponentialfunktion (siehe Kapitel 4.9). Tabelle 5.3 listet verschiedene Verfahren zur Berechnung des diskreten Logarithmus [Sti06] und ihre Komplexität.

Name	Komplexität
Babystep-Giantstep	$O(\sqrt{p})$
Silver-Pohlig-Hellman	polynomial in q, dem größten Primteiler von $p-1$.
Index-Calculus	$O(e^{(1+o(1))\sqrt{\ln(p)\ln(\ln(p))}})$

Tabelle 5.3: Verfahren zur Berechnung des diskreten Logarithmus in $\mathbb{Z}_p^*$

Der aktuelle Rekord (Stand April 2007) für die Berechnung des diskreten Logarithmus wurde im Februar 2007 von der Gruppe Kleinjung, Franke und Bahr an der Universität Bonn aufgestellt.[9] Kleinjung berechnete den diskreten Logarithmus modulo einer 160-stelligen Primzahl p

[8] In dem Lernprogramm **ZT** können Sie mit der Verteilung des diskreten Logarithmus experimentieren und Shanks Babystep-Giantstep-Methode anwenden: Siehe Lern-Kapitel 6.1-6.3, Seiten 1-6/6.
ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen. Siehe Anhang A.6.

[9] `http://www.nabble.com/Discrete-logarithms-in-GF(p)-----160-digits-t3175622.html`

und Erzeuger g:

$$\begin{aligned} p &= \lfloor 10^{159}\pi \rfloor + 119849 \\ &= 31415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284812372 99 \\ g &= 2 \end{aligned}$$

Konkret wurde der diskrete Logarithmus k von folgender Zahl y berechnet:[10]

$$\begin{aligned} y &= \lfloor 10^{159}e \rfloor \\ &= 27182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952605956307 38 \\ k &= \log_g(y) \mod p \\ &= 82989716465034897051864680264075784402496146932312647219853184518689598402644834266625285046612688143761738165394262430753767931963671156105352608242351366559 6 \end{aligned}$$

Die Suche wurde mit der GNFS-Methode (General Number Field Sieve, Index-Calculus) durchgeführt und benötigte ca. 17 CPU-Jahre auf 3.2 GHz Xeon Maschinen.

5.4.2 Diffie-Hellman-Schlüsselvereinbarung[11]

Die Mechanismen und Algorithmen der klassischen Kryptographie greifen erst dann, wenn die Teilnehmer bereits den geheimen Schlüssel ausgetauscht haben. Im Rahmen der klassischen Kryptographie führt kein Weg daran vorbei, dass Geheimnisse kryptographisch ungesichert ausgetauscht werden müssen. Die Sicherheit der Übertragung muss hier durch nicht-kryptographische Methoden erreicht werden. Man sagt dazu, dass man zum Austausch der Geheimnisse einen geheimen Kanal braucht; dieser kann physikalisch oder organisatorisch realisiert sein.
Das Revolutionäre der modernen Kryptographie ist unter anderem, dass man keine geheimen Kanäle mehr braucht: Man kann geheime Schlüssel über nicht-geheime, also öffentliche Kanäle vereinbaren.
Ein Protokoll, das dieses Problem löst, ist das von Diffie und Hellman (Krypto-Verfahren 5.5).

Krypto-Verfahren 5.5 Diffie-Hellman-Schlüsselvereinbarung

Zwei Teilnehmer A und B wollen einen gemeinsamen geheimen Schlüssel vereinbaren.
Sei p eine Primzahl und g eine natürliche Zahl. Diese beide Zahlen müssen nicht geheim sein.
Zunächst wählen sich die beiden Teilnehmer je eine geheime Zahl a bzw. b. Daraus bilden sie die Werte $\alpha = g^a \mod p$ und $\beta = g^b \mod p$. Dann werden die Zahlen α und β ausgetauscht. Schließlich potenziert jeder den erhaltenen Wert mit seiner geheimen Zahl und erhält $\beta^a \mod p$ bzw. $\alpha^b \mod p$.
Damit gilt

$$\beta^a \equiv (g^b)^a \equiv g^{ba} \equiv g^{ab} \equiv (g^a)^b \equiv \alpha^b \mod p$$

Die Sicherheit des **Diffie-Hellman-Protokolls** hängt eng mit der Berechnung des diskreten Logarithmus modulo p zusammen. Es wird sogar vermutet, dass diese Probleme äquivalent sind.

[10] Die Zahl y ergab sich aus den ersten 159 Stellen der Eulerschen Zahl e.
[11] In CT1 ist dieses Austauschprotokoll visualisiert: Sie können die einzelnen Schritte mit konkreten Zahlen nachvollziehen per Menü **Einzelverfahren \ Protokolle \ Diffie-Hellman-Demo**.
In JCT findet man es in der Standard-Perspektive über den Menüeintrag **Visualisierungen \ Diffie-Hellman-Schlüsselaustausch (EC)**.

5.4.3 ElGamal-Public-Key-Verschlüsselungsverfahren in $\mathbb{Z}_p^*$

Indem man das Diffie-Hellman Schlüsselvereinbarungsprotokoll leicht variiert, kann man einen asymmetrischen Verschlüsselungsalgorithmus erhalten (Krypto-Verfahren 5.6). Diese Beobachtung geht auf Taher ElGamal zurück.

Krypto-Verfahren 5.6 ElGamal (auf dem diskreten Logarithmusproblem basierend)

Sei p eine Primzahl, so dass der diskrete Logarithmus in $\mathbb{Z}_p$ schwierig zu berechnen ist. Sei $\alpha \in \mathbb{Z}_p^*$ ein primitives Element. Sei $a \in \mathbb{N}$ eine natürliche Zahl und $\beta = \alpha^a \mod p$.

Öffentlicher Schlüssel:

$$p, \alpha, \beta.$$

Privater Schlüssel:

$$a.$$

Sei $k \in \mathbb{Z}_{p-1}$ eine zufällige Zahl und $x \in \mathbb{Z}_p^*$ der Klartext.

Verschlüsselung:

$$e_T(x, k) = (y_1, y_2),$$

wobei

$$y_1 = \alpha^k \mod p,$$

und

$$y_2 = x\beta^k \mod p.$$

Entschlüsselung:

$$d_T(y_1, y_2) = y_2(y_1^a)^{-1} \mod p.$$

5.4.4 Verallgemeinertes ElGamal-Public-Key-Verschlüsselungsverfahren

Den diskreten Logarithmus kann man in beliebigen endlichen Gruppen $(G, \circ)$ verallgemeinern. Im Folgenden geben wir einige Eigenschaften über die Gruppe G an, die das diskrete Logarithmusproblem schwierig machen. Dabei schreiben wir statt $g \circ h$ oft nur gh.

Berechnung der diskreten Exponentialfunktion Sei G eine Gruppe mit der Operation $\circ$ und $g \in G$. Die (diskrete) Exponentialfunktion zur Basis g ist definiert durch

$$e_g : k \longmapsto g^k, \quad \text{für alle } k \in \mathbb{N}.$$

Dabei definiert man

$$g^k := \underbrace{g \circ \ldots \circ g}_{k \text{ mal}}.$$

Die Exponentialfunktion ist leicht zu berechnen:

Lemma

Die Potenz g^k kann in höchstens $2\log_2 k$ Gruppenoperationen berechnet werden.

Beweis

Sei $k = 2^n + k_{n-1}2^{n-1} + \cdots + k_1 2 + k_0$ die Binärdarstellung von k. Dann ist $n \leq \log_2(k)$, denn

$2^n \leq k < 2^{n+1}$. k kann in der Form $k = 2k' + k_0$ mit $k' = 2^{n-1} + k_{n-1}2^{n-2} + \cdots + k_1$ geschrieben werden. Es folgt

$$g^k = g^{2k'+k_0} = (g^{k'})^2 g^{k_0}.$$

Man erhält also g^k aus $g^{k'}$ indem man einmal quadriert und eventuell mit g multipliziert. Damit folgt die Behauptung durch Induktion nach n. □

Problem des diskreten Logarithmus

Sei G eine endliche Gruppe mit der Operation $\circ$. Sei $\alpha \in G$ und $\beta \in H = \{\alpha^i : i \geq 0\}$.
Gesucht ist das eindeutige $a \in \mathbb{N}$ mit $0 \leq a \leq |H| - 1$ und $\beta = \alpha^a$.
Wir bezeichnen a mit $\log_\alpha(\beta)$.

Berechung des diskreten Logarithmus Ein einfaches Verfahren zur Berechnung des diskreten Logarithmus eines Gruppenelements, das wesentlich effizienter ist als das bloße Durchprobieren aller möglichen Werte für k, ist der Babystep-Giantstep-Algorithmus.

Satz 5.4.1 (Babystep-Giantstep-Algorithmus)**.** *Sei G eine Gruppe und $g \in G$. Sei n die kleinste natürliche Zahl mit $|G| \leq n^2$. Dann kann der diskrete Logarithmus eines Elements $h \in G$ zur Basis g berechnet werden, indem man die folgenden zwei Listen mit jeweils n Elementen erzeugt und diese Listen vergleicht:*

$$\textit{Giantstep-Liste: } \{1, g^n, g^{2n}, \ldots, g^{n \cdot n}\},$$
$$\textit{Babystep-Liste: } \{hg^{-1}, hg^{-2}, \ldots, hg^{-n}\}.$$

Dabei kann man nach Feststellen eines gemeinsamen Elements die Berechnung abbrechen. Zur Berechnung dieser Listen braucht man $2n$ Gruppenoperationen.

Beweis
Falls $g^{jn} = hg^{-i}$, also $h = g^{i+jn}$, so ist das Problem gelöst. Falls die Listen disjunkt sind, so ist h nicht als $g^{i+jn}, i, j \leq n$, darstellbar. Da dadurch alle Potenzen von g erfasst werden, hat das Logarithmusproblem keine Lösung. □

Man kann sich mit Hilfe des Babystep-Giantstep-Algorithmus klar machen, dass die Berechnung des diskreten Logarithmus sehr viel schwieriger ist als die Berechnung der diskreten Exponentialfunktion. Wenn die auftretenden Zahlen etwa 1000 Bit Länge haben, so benötigt man zur Berechnung aller g^k nur etwa 2000 Multiplikationen (siehe Satz 5.4.1), zur Berechnung des diskreten Logarithmus mit dem Babystep-Giantstep-Algorithmus aber etwa $2^{500} \approx 10^{150}$ Operationen.
Neben dem Babystep-Giantstep-Algorithmus gibt es noch zahlreiche andere Verfahren zur Berechnung des diskreten Logarithmus [Sti06].

Der Satz von Silver-Pohlig-Hellman In endlichen abelschen Gruppen lässt sich das diskrete Logarithmusproblem in Gruppen kleinerer Ordnung reduzieren.

Satz 5.4.2 (Silver-Pohlig-Hellman)**.** *Sei G eine endliche abelsche Gruppe mit $|G| = p_1^{a_1} p_2^{a_2} \cdot \ldots \cdot p_s^{a_s}$. Dann lässt sich das diskrete Logarithmusproblem in G auf das Lösen von Logarithmenproblemen in Gruppen der Ordnung $p_1, \ldots, p_s$ zurückführen.*

Enthält $|G|$ einen „dominanten“ Primteiler p, so ist die Komplexität des Logarithmusproblems ungefähr

$$O(\sqrt{p}).$$

Wenn also das Logarithmusproblem schwer sein soll, so muss die Ordnung der verwendeten Gruppe G einen großen Primteiler haben. Insbesondere gilt, wenn die diskrete Exponentialfunktion in der Gruppe $\mathbb{Z}_p^*$ eine Einwegfunktion sein soll, so muss $p-1$ einen großen Primteiler haben. In diesem Fall kann man ein verallgemeinertes ElGamal-Verfahren definieren (Krypto-Verfahren 5.7).

Krypto-Verfahren 5.7 Verallgemeinertes ElGamal (auf dem diskreten Logarithmusproblem basierend)

Sei G eine endliche Gruppe mit Operation $\circ$, und sei $\alpha \in G$, so dass der diskrete Logarithmus in $H = \{\alpha^i : i \geq 0\}$ schwer zu berechnen ist. Sei a mit $0 \leq a \leq |H| - 1$ und sei $\beta = \alpha^a$.

Öffentlicher Schlüssel:

$$\alpha, \beta$$

Privater Schlüssel:

$$a$$

Sei $k \in \mathbb{Z}_{|H|}$ eine zufällige Zahl, $k \neq 0$, und $x \in G$ ein Klartext.

Verschlüsselung:

$$e_T(x, k) = (y_1, y_2),$$

wobei

$$y_1 = \alpha^k$$

und

$$y_2 = x \circ \beta^k.$$

Entschlüsselung:

$$d_T(y_1, y_2) = y_2 \circ (y_1^a)^{-1}.$$

Elliptische Kurven (siehe Kapitel 7) liefern nützliche Gruppen für Public-Key-Verschlüsselungsverfahren.

Literaturverzeichnis (Kap. ModernCrypto)

[Adl83] Adleman, L.: *On breaking the iterated Merkle-Hellman public-key Cryptosystem.* In: *Advances in Cryptologie, Proceedings of Crypto 82*, Seiten 303–308. Plenum Press, 1983.

[BDG98] Balcazar, J. L., J. Daaz und J. Gabarr: *Structural Complexity I.* Springer, 1998.

[Bri85] Brickell, E. F.: *Breaking Iterated Knapsacks.* In: *Advances in Cryptology: Proc. CRYPTO'84, Lecture Notes in Computer Science, vol. 196*, Seiten 342–358. Springer, 1985.

[Hes01] Hesselink, Wim H.: *The borderline between P and NP*, Februar 2001. `http://www.cs.rug.nl/~wim/pub/whh237.pdf`.

[Lag83] Lagarias, J. C.: *Knapsack public key Cryptosystems and diophantine Approximation.* In: *Advances in Cryptology, Proceedings of Crypto 83.* Plenum Press, 1983.

[MH78] Merkle, R. und M. Hellman: *Hiding information and signatures in trapdoor knapsacks.* IEEE Trans. Information Theory, IT-24, 24, 1978.

[RSA78] Rivest, Ron L., Adi Shamir und Leonard Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21(2):120–126, April 1978.

[Sha82] Shamir, A.: *A polynomial time algorithm for breaking the basic Merkle-Hellman Cryptosystem.* In: *Symposium on Foundations of Computer Science*, Seiten 145–152, 1982.

[Sti06] Stinson, Douglas R.: *Cryptography – Theory and Practice.* Chapman & Hall/CRC, 3. Auflage, 2006.

Alle Links wurden am 13.07.2016 überprüft.

Kapitel 6

Hashfunktionen und Digitale Signaturen

(Jörg-Cornelius Schneider / Bernhard Esslinger / Henrik Koy, Juni 2002; Updates: Feb. 2003, Juni 2005, Juli 2009, Nov. 2012)

> Wir können alles aus dieser Welt machen, nur nicht eine Welt, in der die Menschen in einigen zigtausend Jahren überlegen könnten: 'So, es ist nun genug. So soll es von nun an für immer bleiben. Verändern wir nichts, erfinden wir nichts, weil es besser nicht sein kann, und wenn doch, dann wollen wir es nicht.'

Zitat 16: Stanislaw Lem[1]

Ziel der digitalen Signatur ist es, folgende zwei Punkte zu gewährleisten:

- Benutzerauthentizität:
  Es kann überprüft werden, ob eine Nachricht tatsächlich von einer bestimmten Person stammt.
- Nachrichtenintegrität:
  Es kann überprüft werden, ob die Nachricht (unterwegs) verändert wurde.

Zum Einsatz kommt wieder eine asymmetrische Technik (siehe Verschlüsselungsverfahren). Ein Teilnehmer, der eine digitale Signatur für ein Dokument erzeugen will, muss ein Schlüsselpaar besitzen. Er benutzt seinen geheimen Schlüssel, um Signaturen zu erzeugen, und der Empfänger benutzt den öffentlichen Schlüssel des Absenders, um die Richtigkeit der Signatur zu überprüfen. Es darf wiederum nicht möglich sein, aus dem öffentlichen den geheimen Schlüssel abzuleiten.

Im Detail sieht ein *Signaturverfahren*[2] folgendermaßen aus:
Der Absender berechnet aus seiner Nachricht und seinem geheimen Schlüssel die digitale Signatur der Nachricht. Im Vergleich zur handschriftlichen Unterschrift hat die digitale Signatur den

[1] Antwort von Stanislaw Lem auf die Kritik an seinem philosophischen Hauptwerk „Summa Technologiae", 1964, in der er die evolutionäre Möglichkeit einer Entstehung der künstlichen Intelligenz ausführte.

[2] Mit CT1 können Sie ebenfalls digitale Signaturen erzeugen und prüfen:
entweder in den Untermenüs des Hauptmenüpunktes **Digitale Signaturen / PKI** oder
per **Einzelverfahren \ RSA-Kryptosystem \ Signaturdemo (Signaturerzeugung)**.
Ebenso können mit JCT (in der Standard- und der Algorithmen-Perspektive) unterschiedliche Arten elektronischer Signaturen erzeugt werden.

Vorteil, dass die Unterschrift auch vom unterschriebenen Dokument abhängt. Die Unterschriften ein und desselben Teilnehmers sind verschieden, sofern die unterzeichneten Dokumente nicht vollkommen übereinstimmen. Selbst das Einfügen eines Leerzeichens in den Text würde zu einer anderen Signatur führen. Eine Verletzung der Nachrichtenintegrität wird also vom Empfänger der Nachricht erkannt, da in diesem Falle die Signatur nicht mehr zum Dokument passt und sich bei der Überprüfung als unkorrekt erweist.

Das Dokument wird samt Signatur an den Empfänger verschickt. Dieser kann mit Hilfe des öffentlichen Schlüssels des Absenders, des Dokuments und der Signatur feststellen, ob die Signatur korrekt ist. Das gerade beschriebene Verfahren hätte in der Praxis jedoch einen entscheidenden Nachteil: Die Signatur wäre ungefähr genauso lang wie das eigentliche Dokument. Um den Datenverkehr nicht unnötig anwachsen zu lassen und aus Performance-Gründen wendet man – vor dem Signieren – auf das Dokument eine kryptographische Hashfunktion[3] an. Deren Output wird dann signiert.

6.1 Hashfunktionen

Eine *Hashfunktion*[4] bildet eine Nachricht beliebiger Länge auf eine Zeichenfolge mit konstanter Größe, den Hashwert, ab.

6.1.1 Anforderungen an Hashfunktionen

Kryptographisch sichere Hashfunktionen erfüllen folgende drei Anforderungen (Reihenfolge so, dass die Anforderungen ansteigen):

- Standhaftigkeit gegen 1st-Pre-Image-Attacks:
  Es sollte praktisch unmöglich sein, zu einer gegebenen Zahl eine Nachricht zu finden, die genau diese Zahl als Hashwert hat.
  Gegeben (fix): Hashwert H',
  Gesucht: Nachricht m, so dass gilt: H(m) = H'.
- Standhaftigkeit gegen 2nd-Pre-Image-Attacks:
  Es sollte praktisch unmöglich sein, zu einer gegebenen Nachricht eine zweite Nachricht zu finden, die genau denselben Hashwert hat.

[3] Hashfunktionen sind in CT1 an mehreren Stellen implementiert.
In den Menüs **Einzelverfahren \ Hashverfahren** bzw. **Analyse \ Hashverfahren** haben Sie die Möglichkeit

- 6 Hashfunktionen auf den Inhalt des aktiven Fensters anzuwenden,
- für eine Datei den Hashwert zu berechnen,
- in der Hash-Demo die Auswirkung von Textänderungen auf den Hashwert zu testen,
- aus einem Passwort gemäß dem PKCS#5-Standard einen Schlüssel zu berechnen,
- aus einem Text und einem geheimen Schlüssel HMACs zu berechnen, und
- aufgrund von gezielt gesuchten Hashwertkollisionen einen Angriff auf digitale Signaturen zu simulieren.

CT2 und JCT enthalten ebenfalls verschiedene Hashverfahren: Vergleiche die Funktionenliste im Anhang A.2 und A.3.

[4] Hashverfahren berechnen eine komprimierte Repräsentation elektronischer Daten (Message). Die Verarbeitung dieser Message durch das Hashverfahren ergibt als Output einen sogenannten Message Digest. Message Digests sind typischerweise zwischen 128 und 512 Bit lang – abhängig vom Algorithmus. Sichere Hashverfahren werden typischerweise mit anderen kryptographischen Algorithmen kombiniert, wie z.B. Digitale-Signatur-Algorithmen, Keyed-Hash Message Authentication Codes, oder bei der Erzeugung von Zufallszahlen (Bits) benutzt.

Gegeben (fix): Nachricht m1 [und damit der Hashwert H1 = H(m1)],
Gesucht: Nachricht m2, so dass gilt: H(m2) = H1.

- Standhaftigkeit gegen Kollisionsangriffe:
  Es sollte es praktisch unmöglich sein, zwei (beliebige) Nachrichten mit demselben Hashwert (welcher ist egal) zu finden.
  Gesucht: 2 Nachrichten m1 und m2, so dass gilt: H(m1) = H(m2).

6.1.2 Aktuelle Angriffe gegen Hashfunktionen // SHA-3

Bisher konnte die Existenz von perfekt sicheren kryptographischen Hashfunktionen nicht formal bewiesen werden.

Über mehrere Jahre gab es keine neuen Attacken gegen Hashverfahren, und allgemein wurde den Kandidaten, die in der Praxis bislang keine Schwächen in ihrer Struktur gezeigt hatten (zum Beispiel SHA-1[5] oder RIPEMD-160[6]) vertraut.

Auf der Crypto 2004 (August 2004)[7] wurde dieses Sicherheitsgefühl jedoch stark in Zweifel gezogen: Chinesische Wissenschaftler veröffentlichten Kollisionsangriffe gegen MD4, SHA-0 und Teile von SHA-1, die weltweit zu einer starken Beschäftigung mit neuen Hash-Angriffen führte.

Die zunächst veröffentlichten Resultate reduzierten den erwarteten Aufwand für die Suche nach einer SHA-1 Kollision von 2^{80} (brute-force) auf 2^{69} [WYY05b]. In der Folge wurden Verfahren angekündigt, die den Aufwand weiter auf 2^{63} [WYY05a] und 2^{52} [MHP12] reduzieren sollen. Damit wäre der Kollisionsangriff in den Bereich des praktisch möglichen gerückt, denn ähnliche Aufwände wurden in der Vergangenheit schon realisiert (s. 1.3.3).

Die Sicherheit bereits erstellter Signaturen wird durch den geschilderten Kollisionsangriff aber nicht gefährdet.

Nach dem aktuellen Kenntnisstand ist keine Panik angesagt, aber für digitale Signaturen sollten zumindest in Zukunft längere Hashwerte und/oder andere Verfahren benutzt werden.

Das U.S. National Institute of Standards and Technology (NIST) hat schon vor Bekanntwerden der neuen Ergebnisse angekündigt, SHA-1 in den nächsten Jahren auslaufen zu lassen. Es ist daher zu empfehlen, für neue Produkte zur Erstellung von Signaturen SHA-1 nicht mehr zu verwenden. Die SHA-2 Familie [NIS15] bietet stärkere Verfahren.

---

[5] SHA-1 ist eine in den Standards FIPS 180-1 (durch die US-Behörde NIST), ANSI X9.30 Part 2 und [NIS13] spezifizierte 160-Bit Hashfunktion.
SHA bedeutet „Secure Hash Algorithm“ und wird häufig benutzt, z.B. mit DSA, RSA oder ECDSA.
Der aktuelle Standard [NIS15] definiert vier sichere Hashverfahren – SHA-1, SHA-256, SHA-384 und SHA-512. Für diese Hashalgorithmen sind in der Testsuite FIPS 140-2 auch Validierungstests definiert.
Die Ausgabelänge der SHA-Algorithmen wurde vergrößert aufgrund der Möglichkeit von Geburtstagsangriffen: diese machen – grob gesprochen – den n-Bit AES und ein 2n-bit Hashverfahren äquivalent:
- 128-bit AES – SHA-256
- 192-bit AES – SHA-384
- 256-bit AES – SHA-512.

Mit CT1 können Sie den Geburtstagsangriff auf digitale Signaturen nachvollziehen:
über das Menü **Analyse \ Hashverfahren \ Angriff auf den Hashwert der digitalen Signatur**.
CT2 enthält einen MD5-Kollisionsgenerator.

[6] RIPEMD-160, RIPEMD-128 und die optionale Erweiterung RIPEMD-256 haben Object Identifier, definiert von der ISO-identifizierten Organisation TeleTrusT, sowohl für Hashverfahren als auch in Kombination mit RSA. RIPEMD-160 ist Teil des internationalen ISO/IEC-Standards ISO/IEC 10118-3:1998 für dedizierte Hashfunktionen, zusammen mit RIPEMD-128 and SHA-1. Weitere Details:
- `http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html`
- `http://www.ietf.org/rfc/rfc2857.txt` (“The Use of HMAC-RIPEMD-160-96 within ESP and AH”).

[7] `http://www.iacr.org/conferences/crypto2004/`

Um den neuen Erkenntnissen in der Kryptoanalyse von Hashfunktionen Rechnung zu tragen, hat das NIST 2008 einen Wettbewerb gestartet, um eine neue Hash-Funktion jenseits der SHA-2-Familie zu entwickeln: Als neue Hashfunktion „SHA-3“ wurde im Oktober 2012 Keccak verkündet.[8]

Weitere Informationen zu diesem Thema finden sich z.B. in dem Artikel „Hash mich – Konsequenzen der erfolgreichen Angriffe auf SHA-1“ von Reinhard Wobst und Jürgen Schmidt[9] von Heise Security.

6.1.3 Signieren mit Hilfe von Hashfunktionen

> „Manipulation war Sobols Spezialität ... das Hauptziel der Ermittlungen sollte sein, hinter Sobols Masterplan zu kommen.“

Zitat 17: Daniel Suarez[10]

Signatur-Verfahren mit Hashfunktion[11] funktionieren folgendermaßen: Anstatt das eigentliche Dokument zu signieren, berechnet der Absender nun zuerst den Hashwert der Nachricht und signiert diesen. Der Empfänger bildet ebenfalls den Hashwert der Nachricht (der benutzte Algorithmus muss bekannt sein). Er überprüft dann, ob die mitgeschickte Signatur eine korrekte Signatur des Hashwertes ist. Ist dies der Fall, so wurde die Signatur korrekt verifiziert. Die Authentizität der Nachricht ist damit gegeben, da wir angenommen hatten, dass aus der Kenntnis des öffentlichen Schlüssels nicht der geheime Schlüssel abgeleitet werden kann. Dieser geheime Schlüssel wäre jedoch notwendig, um Nachrichten in einem fremden Namen zu signieren.

Einige digitale Signaturverfahren basieren auf asymmetrischer Verschlüsselung, das bekannteste Beispiel dieser Gattung ist RSA. Für die RSA-Signatur verwendet man die gleiche mathematische Operation wie zum Entschlüsseln, nur wird sie auf den Hash-Wert des zu unterschreibenden Dokuments angewendet.

Andere Systeme der digitalen Signatur wurden, wie DSA (Digital Signature Algorithm), ausschließlich zu diesem Zweck entwickelt, und stehen in keiner direkten Verbindung zu einem entsprechenden Verschlüsselungsverfahren.

Beide Signaturverfahren, RSA und DSA, werden in den folgenden beiden Abschnitten näher beleuchtet. Anschließend gehen wir einen Schritt weiter und zeigen, wie basierend auf der elektronischen Unterschrift das digitale Pendant zum Personalausweis entwickelt wurde. Dieses Verfahren nennt man Public-Key-Zertifizierung.

---

[8] `http://csrc.nist.gov/groups/ST/hash/sha-3/`
Mit **CT2** können Sie im Startcenter über **Vorlagen \ Hash-Funktionen \ Keccak-Hash (SHA-3)** die Keccak-Hashfunktion ausführen und visualisieren.
Keccak kann auch als Zufallszahlengenerator und als Stromchiffre benutzt werden: Das finden Sie in den Startcenter-Vorlagen über **Werkzeuge \ Keccak PRNG**, bzw. **Kryptographie \ Modern \ Symmetrisch \ Keccak-Stromchiffre**.

[9] `http://www.heise.de/security/artikel/56555`.
Weitere Quellen sind z.B.:
`http://csrc.nist.gov/groups/ST/toolkit/index.html`.

[10] Daniel Suarez, „Daemon“, rororo, (c) 2010, Kapitel 14, „mem-payload“, S. 148, Ross.

[11] Vergleiche auch:
`http://de.wikipedia.org/wiki/Digitale_Signatur`,
`http://en.wikipedia.org/wiki/Digital_signature`.

6.2 RSA-Signatur

Wie im Kommentar am Ende von Abschnitt 4.10.3 bemerkt, ist es möglich, die RSA-Operationen mit dem privaten und öffentlichen Schlüssel in umgekehrter Reihenfolge auszuführen, d. h. M hoch d hoch e (mod N) ergibt wieder M. Wegen dieser simplen Tatsache ist es möglich, RSA als Signaturverfahren zu verwenden.

Eine RSA-Signatur S zur die Nachricht M wird durch folgende Operation mit dem privaten Schlüssel erzeugt:

$$S \equiv M^d \pmod{N}$$

Zur Verifikation wird die korrespondierende Public-Key-Operation auf der Signatur S ausgeführt und das Ergebnis mit der Nachricht M verglichen:

$$S^e \equiv (M^d)^e \equiv (M^e)^d \equiv M \pmod{N}$$

Wenn das Ergebnis S^e mit der Nachricht M übereinstimmt, dann akzeptiert der Prüfer die Signatur, andernfalls ist die Nachricht entweder verändert worden, oder sie wurde nicht vom Inhaber von d unterschrieben.

Wie weiter oben erklärt, werden Signaturen in der Praxis nie direkt auf der Nachricht ausführt, sondern auf einem kryptographischen Hashwert davon. Um verschiedene Attacken auf das Signaturverfahren (und seine Kombination mit Verschlüsselung) auszuschließen, ist es nötig, den Hashwert vor der Exponentiation auf spezielle Weise zu formatieren, wie in PKCS#1 (Public Key Cryptography Standard #1 [Lab02]) beschrieben. Der Tatsache, dass dieser Standard nach mehreren Jahren Einsatz revidiert werden musste, kann als Beispiel dafür dienen, wie schwer es ist, kryptographische Details richtig zu definieren.

6.3 DSA-Signatur

Im August 1991 hat das U.S. National Institute of Standards and Technology (NIST) einen digitalen Signaturalgorithmus (DSA, Digital Signature Algorithm) vorgestellt, der später zum U.S. Federal Information Processing Standard (FIPS 186 [NIS13]) wurde.

Der Algorithmus ist eine Variante des ElGamal-Verfahrens. Seine Sicherheit beruht auf dem Diskreten Logarithmus Problem. Die Bestandteile des privaten und öffentlichen DSA-Schlüssels, sowie die Verfahren zur Signatur und Verifikation sind in Krypto-Verfahren 6.1 zusammengefasst.

Obwohl DSA unabhängig von einem Verschlüsselungsverfahren so spezifiziert wurde, dass es aus Länder exportiert werden kann, die den Export von kryptographischer Hard- und Software einschränken (wie die USA zum Zeitpunkt der Spezifikation), wurde festgestellt [Sch96, S. 490], dass die Operationen des DSA dazu geeignet sind, nach RSA bzw. ElGamal zu verschlüsseln.

6.4 Public-Key-Zertifizierung

Ziel der Public-Key-Zertifizierung ist es, die Bindung eines öffentlichen Schlüssels an einen Benutzers zu garantieren und nach außen nachvollziehbar zu machen. In Fällen, in denen nicht sichergestellt werden kann, dass ein öffentlicher Schlüssel auch wirklich zu einer bestimmten Person gehört, sind viele Protokolle nicht mehr sicher, selbst wenn die einzelnen kryptographischen Bausteine nicht geknackt werden können.

Krypto-Verfahren 6.1 DSA-Signatur

Öffentlicher Schlüssel

p prim
q 160-Bit Primfaktor von $p-1$
$g = h^{(p-1)/q} \bmod p$, wobei $h < p-1$ und $h^{(p-1)/q} > 1 \pmod p$
$y \equiv g^x \bmod p$

Bemerkung: Die Parameter p, q und g können von einer Gruppe von Benutzern gemeinsam genutzt werden.

Privater Schlüssel

$x < q$ (160-Bit Zahl)

Signatur

m zu signierende Nachricht
k zufällig gewählte Primzahl, kleiner als q
$r = (g^k \bmod p) \bmod q$
$s = (k^{-1}(\text{SHA-1}(m) + xr)) \bmod q$

Bemerkung:

- (s, r) ist die Signatur.
- Die Sicherheit der Signatur hängt nicht nur von der Mathematik ab, sondern auch von der Verfügbarkeit einer guten Zufallsquelle für k.
- SHA-1 ist eine 160-Bit Hashfunktion.

Verifikation

$w = s^{-1} \bmod q$
$u_1 = (\text{SHA-1}(m)w) \bmod q$
$u_2 = (rw) \bmod q$
$v = (g^{u_1} y^{u_2}) \bmod p) \bmod q$

Bemerkung: Wenn $v = r$, dann ist die Signatur gültig.

6.4.1 Die Impersonalisierungsattacke

Angenommen Charlie hat zwei Schlüsselpaare (PK1, SK1) und (PK2, SK2). Hierbei bezeichnet SK den geheimen Schlüssel (secret key) und PK den öffentlichen Schlüssel (public key). Weiter angenommen, es gelingt ihm, Alice PK1 als öffentlichen Schlüssel von Bob und Bob PK2 als öffentlichen Schlüssel von Alice „unterzujubeln“ (etwa indem er ein öffentliches Schlüsselverzeichnis fälscht).

Dann ist folgender Angriff möglich:

- Alice möchte eine Nachricht an Bob senden. Sie verschlüsselt diese mit PK1, da sie denkt, dies sei Bobs öffentlicher Schlüssel. Anschließend signiert sie die Nachricht mit ihrem geheimen Schlüssel und schickt sie ab.
- Charlie fängt die Nachricht ab, entfernt die Signatur und entschlüsselt die Nachricht mit SK1. Wenn er möchte, kann er die Nachricht anschließend nach Belieben verändern. Dann verschlüsselt er sie wieder, aber diesmal mit dem echten öffentlichen Schlüssel von Bob, den er sich aus einem öffentlichen Schlüsselverzeichnis geholt hat, signiert sie mit SK2 und schickt die Nachricht weiter an Bob.
- Bob überprüft die Signatur mit PK2 und wird zu dem Ergebnis kommen, dass die Signatur

in Ordnung ist. Dann entschlüsselt er die Nachricht mit seinem geheimen Schlüssel.

Charlie ist so in der Lage, die Kommunikation zwischen Alice und Bob abzuhören und die ausgetauschten Nachrichten zu verändern, ohne dass dies von den beteiligten Personen bemerkt wird. Der Angriff funktioniert auch, wenn Charlie nur ein Schlüsselpaar hat.

Ein anderer Name für diese Art von Angriffen ist „Man-in-the-Middle-Attack“. Hilfe gegen diese Art von Angriffen verspricht die Public-Key-Zertifizierung, die die Authentizität öffentlicher Schlüssel garantieren kann. Die am weitesten verbreitete Zertifizierungsmethode ist der X.509-Standard.

6.4.2 X.509-Zertifikat

Jeder Teilnehmer, der sich per X.509-Zertifikat [IT97] die Zugehörigkeit seines öffentlichen Schlüssels zu seiner realen Person bestätigen lassen möchte, wendet sich an eine sogenannte Certification Authority (CA)[12]. Dieser beweist er seine Identität (etwa durch Vorlage seines Personalausweises). Anschließend stellt die CA ihm ein elektronisches Dokument (Zertifikat) aus, in dem im wesentlichen der Name des Zertifikatnehmers und der Name der CA, der öffentliche Schlüssel des Zertifikatnehmers und der Gültigkeitszeitraum des Zertifikats vermerkt sind. Die CA unterzeichnet das Zertifikat anschließend mit ihrem geheimen Schlüssel.

Jeder kann nun anhand des öffentlichen Schlüssels der CA überprüfen, ob das Zertifikat unverfälscht ist. Die CA garantiert also die Zugehörigkeit von Benutzer und öffentlichem Schlüssel.

Dieses Verfahren ist nur so lange sicher, wie die Richtigkeit des öffentlichen Schlüssels der CA sichergestellt ist. Aus diesem Grund lässt jede CA ihren öffentlichen Schlüssel bei einer anderen CA zertifizieren, die in der Hierarchie über ihr steht. In der obersten Hierarchieebene (Wurzelinstanz) gibt es in der Regel nur eine CA, die dann natürlich keine Möglichkeit mehr hat, sich ihren Schlüssel bei einer anderen CA zertifizieren zu lassen. Sie ist also darauf angewiesen, ihren Schlüssel auf andere Art und Weise gesichert zu übermitteln. Bei vielen Software-Produkten, die mit Zertifikaten arbeiten (zum Beispiel den Webbrowsern von Microsoft und Netscape) sind die Zertifikate dieser Wurzel-CAs schon von Anfang an fest in das Programm eingebettet und können auch vom Benutzer nachträglich nicht mehr geändert werden. Aber auch durch öffentliche Bekanntgabe in Zeitungen können (öffentliche) CA-Schlüssel gesichert übermittelt werden.

[12] Oft auch Trustcenter oder im deutschen Signaturgesetz „Zertifizierungsdiensteanbieter“ genannt, wenn die Zertifikate nicht nur einer geschlossenen Benutzergruppe angeboten werden.

Literaturverzeichnis (Kap. DigSig)

[IT97] ITU-T: *ITU-T Recommendation X.509 (1997 E): Information Technology – Open Systems Interconnection – The Directory: Authentication Framework.* Technischer Bericht, International Telecommunication Union ITU-T, Juni 1997.

[Lab02] Labs, RSA: *PKCS #1 v2.1 Draft 3: RSA Cryptography Standard.* Technischer Bericht, RSA Laboratories, April 2002.

[MHP12] McDonald, Cameron, Philip Hawkes und Josef Pieprzyk: *Differential Path for SHA-1 with complexity $O(2^{52})$.* Cryptology ePrint Archive, 2012. `http://eprint.iacr.org/2009/259`.

[NIS13] NIST: *Digital Signature Standard (DSS).* Technischer Bericht, NIST (U.S. Department of Commerce), 2013. Change note 4.
`http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`,
`http://csrc.nist.gov/publications/PubsFIPS.html`.

[NIS15] NIST: *Secure Hash Standard (SHS).* Technischer Bericht, NIST (U.S. Department of Commerce), August 2015. FIPS 180-4 supersedes FIPS 180-2.
`http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`,
`http://csrc.nist.gov/publications/PubsFIPS.html`.

[Sch96] Schneier, Bruce: *Applied Cryptography, Protocols, Algorithms, and Source Code in C.* Wiley, 2. Auflage, 1996.

[WYY05a] Wang, Xiaoyun, Andrew Yao und Frances Yao: *New Collision Search for SHA-1.* Technischer Bericht, Crypto 2005, Rump Session, 2005.
`http://www.iacr.org/conferences/crypto2005/rumpSchedule.html`.

[WYY05b] Wang, Xiaoyun, Yiqun Yin und Hongbo Yu: *Finding Collisions in the Full SHA-1.* Advances in Cryptology-Crypto, LNCS 3621, Seiten 17–36, 2005.

Alle Links wurden am 14.07.2016 überprüft.

Kapitel 7

Elliptische Kurven

(Bartol Filipovic / Matthias Büger / Bernhard Esslinger / Roger Oyono, April 2000, Updates: Dez. 2001, Juni 2002, März 2003, Nov. 2009, Aug. 2013, Aug. 2016)

7.1 Elliptische Kurven – Ein effizienter Ersatz für RSA?

Bei Datenübertragungen kommt es auf Sicherheit und Effizienz an. Viele Anwendungen verwenden den RSA-Algorithmus als asymmetrisches Signatur- und Verschlüsselungsverfahren.

Solange die verwendeten Schlüssel hinreichend lang sind, bestehen keinerlei Bedenken gegen die *Sicherheit* des RSA-Verfahrens. Allerdings hat die Entwicklung der Rechnerleistungen in der vergangenen Jahren dazu geführt, dass die benötigten Schlüssellängen mehrfach angehoben werden mussten (vergleiche Kapitel 4.11). Da die meisten Chips auf Smartcards nicht in der Lage sind, längere Schlüssel als z.B. 2000 Bit zu verarbeiten, besteht Bedarf für Alternativen zum RSA. Eine solche Alternative sind Elliptische Kurven. Diese werden inzwischen auch auf Smartcards eingesetzt.

Die *Effizienz* eines kryptographischen Algorithmus hängt wesentlich von der benötigten Schlüssellänge und vom Rechenaufwand ab, um ein vorgeschriebenes Sicherheitsniveau zu erreichen. Der entscheidende Vorteil Elliptischer Kurven im Vergleich zum RSA-Algorithmus liegt in der Tatsache, dass die sicheren Schlüssellängen erheblich kürzer sind.

Setzt man den Trend, dass sich die Leistung der verfügbaren Rechner im Schnitt alle 18 Monate verdoppelt (vgl. dazu das Gesetz von Moore[1]), in die Zukunft fort, so kann man von einer Entwicklung der sicheren Schlüssellängen wie in Abbildung 7.1 ausgehen, die anhand der Tabelle 1 (auf Seite 32 in [LV01]) erstellt wurde.[2]

[1] Das Mooresche Gesetz formuliert die empirische Beobachtung und die darauf aufbauende Prognose, dass sich die Anzahl der Komponenten bzw. Transistoren in einer integrierten Schaltung alle zwei Jahre verdoppelt. Es bezog sich ursprünglich nur auf die Transistorendichte in einer integrierten Schaltung, aber bspw. nicht auf den Zuwachs bei der Speicherdichte. Diese Erkenntnis zur Transistorendichte wurde 1965 (mit einer Korrektur 1975) von Gordon Moore, einem Mitbegründer von Intel, geäußert. In den letzten Jahren war der Zuwachs an Computerleistung sogar höher als die prognostizierte Verdopplung alle 2 Jahre. Grenzen setzt in Zukunft die Transistorengröße von wenigen Atomen, die bis zum Jahr 2020 erreicht werden könnte.

[2] Weitere Informationen zum Schlüssellängen-Vergleich von Arjen Lenstra und Eric Verheul, aber auch zu neueren Untersuchungen bis 2015 finden sich in der interaktiven Darstellung auf `http://www.keylength.com`.

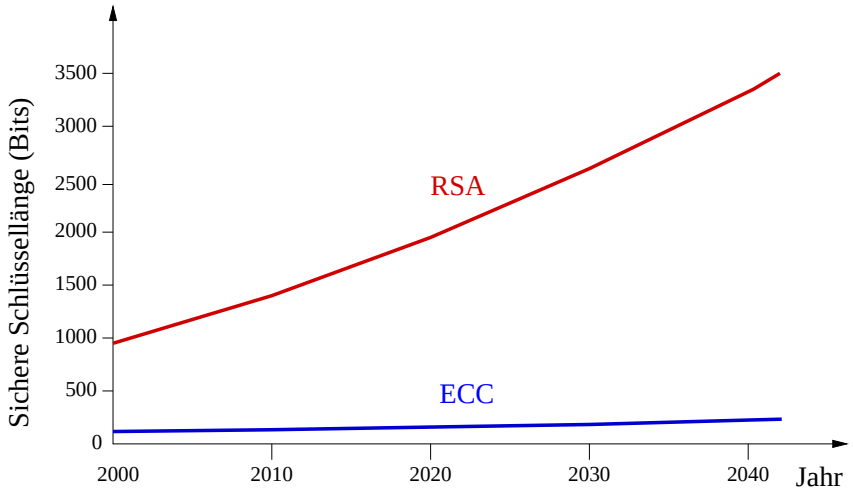


Abbildung 7.1: Prognose für die Entwicklung der als sicher betrachteten Schlüssellängen bei RSA und bei Elliptische Kurven

Bei der digitalen Signatur muss man differenzieren: für die *Erstellung* einer digitalen Signatur benötigen auf Elliptischen Kurven basierende Verfahren im Vergleich zu RSA nur gut ein Zehntel des Rechenaufwandes (66 zu 515 Ganzzahlmultiplikationen). Siehe hierzu Abbildung 7.2 (Quelle: J. Merkle, Elliptic Curve Cryptography Workshop, 2001). Betrachtet man die für eine *Verifikation* durchzuführenden Rechenschritte, dreht sich dieses Bild jedoch zu Gunsten von RSA um (112 zu 17 Ganzzahlmultiplikationen). Der Grund liegt darin, dass es bei Verwendung des RSA möglich ist, einen sehr kurzen Exponent für den öffentlichen Schlüssel zu wählen, solange der private Exponent nur hinreichend lang ist.

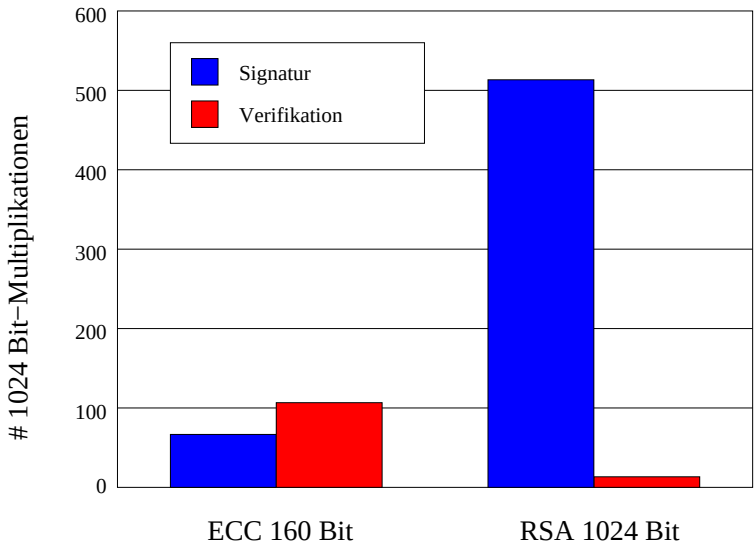


Abbildung 7.2: Gegenüberstellung des Aufwands der Operationen Signieren und Verifizieren bei RSA und bei Elliptischen Kurven

Da bei Smartcards, die auf RSA basieren, stets der lange (private) Schlüssel auf der Karte gespeichert werden muss und die Erstellung der digitalen Signatur, nicht aber die Verifikation, auf der Karte stattfindet, treten hier deutlich die Vorteile Elliptischer Kurven zutage.

Das größte Problem bei der Implementierung von Verfahren, die auf Elliptischen Kurven beruhen, ist bislang die mangelnde *Standardisierung*. Es gibt nur eine RSA-Implementierung, aber viele Arten, Elliptische Kurven einzusetzen. So können verschiedene Zahlkörper zugrunde gelegt, eine Vielzahl von (Elliptischen) Kurven — durch Parameter beschrieben[3] — eingesetzt und unterschiedliche Darstellungen der Kurvenpunkte verwendet werden. Jede Wahl hat ihre Vorzüge, so dass für jede Anwendung eine andere Implementierung optimal sein kann. Dies hat jedoch zur Konsequenz, dass Systeme, die auf Elliptischen Kurven beruhen, oftmals nicht interoperabel sind. Um mit einer beliebigen auf Elliptischen Kurven basierenden Anwendung kommunizieren zu können, müsste man eine Vielzahl von Implementierungen vorhalten, was den Effizienzvorteil gegenüber der Verwendung von RSA zunichte macht.

Deshalb bemühen sich internationale Organisationen um Standardisierung: IEEE (P1363), ASC (ANSI X9.62, X9.63), ISO/IEC sowie RSA Laboratories und Certicom. Im Gegensatz zur IEEE, die bisher nur eine Beschreibung der verschiedenen Implementierungen vorgenommen hat, hat die ASC konkret 10 Kurven ausgewählt und empfiehlt deren Verwendung. Der Vorteil des ASC-Ansatzes ist, dass ein einziges Byte ausreicht, um die verwendete Kurve zu spezifizieren. Zur Zeit ist jedoch nicht absehbar, ob es der ASC gelingen wird, einen de-facto-Standard durchzusetzen.

Obwohl aktuell kein Handlungsbedarf besteht[4], laufende RSA-Anwendungen umzustellen, sollte man ernsthaft den Einsatz von Elliptischen Kurven erwägen[5]. Neuere Diskussionen zur Sicherheit von ECC finden Sie in Kapitel 10.

7.2 Elliptische Kurven – Historisches

Auf dem Gebiet der Elliptischen Kurven wird seit über 100 Jahren geforscht. Im Laufe der Zeit hat man viele weitläufige und mathematisch tiefgründige Resultate im Zusammenhang mit Elliptischen Kurven gefunden und veröffentlicht. Ein Mathematiker würde sagen, dass die Elliptischen Kurven (bzw. die dahinterstehende Mathematik) gut verstanden sind. Ursprünglich war diese Forschung reine Mathematik, das heißt Elliptische Kurven wurden zum Beispiel in den mathematischen Teilgebieten Zahlentheorie und algebraische Geometrie untersucht, die allgemein sehr abstrakt sind. Auch in der nahen Vergangenheit spielten Elliptische Kurven eine bedeutende Rolle in der reinen Mathematik. In den Jahren 1993 und 1994[6] veröffentlichte Andrew Wiles mathematische Arbeiten, die weit über das Fachpublikum hinaus auf große Begeisterung gestoßen sind. In diesen Arbeiten bewies er die Richtigkeit einer — in den sechziger Jahren des 20. Jahrhunderts von zwei Japanern aufgestellten — Vermutung. Dabei geht es kurz und grob gesagt um den Zusammenhang zwischen Elliptischen Kurven und sogenannten Modulformen. Das für die meisten eigentlich Interessante daran ist, dass Wiles mit seinen Arbeiten auch den berühmten zweiten Satz von Fermat bewiesen hat. Dieser Satz hatte sich seit Jahrhunderten (Fermat lebte von 1601 bis 1665) einem umfassenden Beweis durch die Mathematik entzogen. Dementsprechend groß war die Resonanz auf den Beweis durch Wiles. In der Formulierung von Fermat lautet der nach ihm benannte Satz so (Fermat hat folgende Worte an den Rand des 1621 von Bachet de Meziriac herausgegebenen Werks von Diophant geschrieben):

[3] Siehe Kapitel 7.4
[4] Aktuelle Informationen zur Sicherheit des RSA-Verfahrens finden Sie in den Kapiteln 4.11 und 10.
[5] Siehe bspw. die Technische Richtlinie des BSI „Kryptographische Verfahren: Empfehlungen und Schlüssellängen“ vom 15. Februar 2016.
[6] 1994 wurden die Beweislücken durch Wiles und Richard Taylor geschlossen.

Cubum autem in duos cubos, aut quadratoquadratum in duos quadratoquadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos ejusdem nominis fas est dividere: cujus rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Frei übersetzt und mit der Schreibweise der heutigen Mathematik bedeutet dies:
Es gibt keine positiven ganzen Zahlen x, y und z größer als Null, so dass $x^n + y^n = z^n$ für $n > 2$ gilt. Ich habe einen bemerkenswerten Beweis für diese Tatsache gefunden, aber es ist nicht genug Platz am Rand [des Buches], um ihn niederzuschreiben.

Dies ist schon bemerkenswert: Eine relativ einfach zu verstehende Aussage (gemeint ist Fermats zweiter Satz) konnte erst nach so langer Zeit bewiesen werden, obwohl Fermat selber angab, schon einen Beweis gefunden zu haben. Im übrigen ist der Beweis von Wiles sehr umfangreich (alle im Zusammenhang mit dem Beweis stehenden Veröffentlichungen von Wiles ergeben schon ein eigenes Buch). Man sollte sich daher im klaren sein, dass die Elliptischen Kurven im allgemeinen sehr tiefgreifende Mathematik berühren.

Soweit zur Rolle der Elliptischen Kurven in der reinen Mathematik. Im Jahr 1985 haben Neal Koblitz und Victor Miller unabhängig voneinander vorgeschlagen, Elliptische Kurven in der Kryptographie einzusetzen. Damit haben die Elliptischen Kurven auch eine ganz konkrete praktische Anwendung gefunden. Ein weiteres interessantes Einsatzgebiet für Elliptische Kurven ist die Faktorisierung von ganzen Zahlen (auf der Schwierigkeit/Komplexität, die Primfaktoren einer sehr großen Zahl zu finden, beruht das RSA-Kryptosystem: vergleiche Kapitel 4.11). In diesem Bereich werden seit 1987 Verfahren untersucht und eingesetzt, die auf Elliptischen Kurven basieren (vergleiche Kapitel 7.8).
Es gibt auch Primzahltests, die auf Elliptischen Kurven basieren.

Elliptische Kurven werden in den verschiedenen Gebieten unterschiedlich eingesetzt: Verschlüsselungsverfahren auf Basis von Elliptischen Kurven beruhen auf der Schwierigkeit des als Elliptische Kurven Diskreter Logarithmus bekannten Problems. Zur Faktorisierung ganzer Zahlen wird die Tatsache benutzt, dass man eine große Zahl elliptischer Kurven für eine natürliche zusammengesetzte Zahl n erzeugen kann.

7.3 Elliptische Kurven – Mathematische Grundlagen

In diesem Abschnitt erhalten Sie Informationen über *Gruppen* und *Körper*.[7]

7.3.1 Gruppen

Da der Begriff der *Gruppe* umgangssprachlich anders als in der Mathematik eingesetzt wird, soll der Vollständigkeit halber an dieser Stelle die wesentliche Aussage der formalen Definition einer Gruppe kurz eingeführt werden:

- Eine Gruppe ist eine nichtleere Menge G mit einer Verknüpfung „$\cdot$“. Die Menge G ist unter der Verknüpfung $\cdot$ abgeschlossen, d.h, sind a, b Elemente aus G, so ist auch ihre Verknüpfung $ab = a \cdot b$ ein Element aus G.
- Für alle Elemente a, b und c aus G gilt: $(ab)c = a(bc)$ (Assoziativgesetz).

[7] Eine didaktisch sehr schöne Einführung in Elliptische Kurven finden Sie in [SWE15].

- Es gibt ein Element e in G, das sich bezüglich der Verknüpfung $\cdot$ neutral verhält, d.h., für alle a aus der Menge G : gilt $ae = ea = a$.
- Zu jedem Element a aus G gibt es ein *inverses Element*[8] a^{-1}(in G), so dass gilt: $aa^{-1} = a^{-1}a = e$.

Gilt zusätzlich noch $ab = ba$ (Kommutativgesetz) für alle a, b aus G, so nennt man die Gruppe G eine *abelsche* Gruppe.

Da man auf der selben Menge mehrere Verknüpfung erklären kann, unterscheidet man diese durch verschiedene Namensgebungen und Zeichen (z.B. + Addition oder $\cdot$ Multiplikation).

Als einfachstes Beispiel einer (abelschen) Gruppe sei die Gruppe der ganzen Zahlen mit der üblichen Addition genannt. Die Menge der ganzen Zahlen wird mit $\mathbb{Z}$ bezeichnet. $\mathbb{Z}$ hat unendlich viele Elemente: $\mathbb{Z} = \{\cdots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \cdots\}$. Die Verknüpfung von zum Beispiel $1 + 2$ liegt in $\mathbb{Z}$, denn $1 + 2 = 3$ und 3 liegt in $\mathbb{Z}$. Das neutrale Element der Gruppe $\mathbb{Z}$ ist 0. Das inverse Element von 3 ist -3, denn $3 + (-3) = 0$.

Für unsere Zwecke besonders interessant sind sogenannte *endliche* Gruppen. D.h. es gibt die zugrundeliegende Menge $\mathcal{M}$ mit einer endlichen Anzahl von Elementen und die Operation $+$, so dass die obigen Bedingungen erfüllt sind. Beispiele sind die Gruppen $\mathbb{Z}_n = \{0, 1, 2, 3, \cdots, n-1\}$ der Teilerreste bei der Division durch $n \in \mathbb{N}$, mit der Addition mod n als Verknüpfung.

Zyklische Gruppen Als *zyklische Gruppen*[9] bezeichnet man solche Gruppen G', die ein Element g besitzen, aus dem man mittels der Gruppen-Verknüpfung alle anderen Elemente der Gruppe erzeugen kann. Es gibt also für jedes Element a aus G' eine positive, ganze Zahl i, so dass die i-fache Verknüpfung von g mit sich selbst $g^i = g \cdot g \cdots g = a$. Das Element g ist ein *Generator* der zyklischen Gruppe — jedes Element in G' lässt sich mittels g und der Verknüpfung erzeugen.

Ordnung von Elementen einer Gruppe Nun zur Ordnung eines Elements der Gruppe: Sei a aus G. Die kleinste positive ganze Zahl r für die gilt, dass a^r, also r mal a mit sich selbst verknüpft, das neutrale Element der Gruppe G' ist (d.h. $a^r = e$), nennt man *Ordnung* von a.

Die *Ordnung der Gruppe* ist die Anzahl der Elemente in der Menge G. Ist die Gruppe G zyklisch und g ein Generator, so stimmt die Ordnung von g mit der Gruppenordnung überein. Man kann leicht zeigen, dass die Ordnung eines Gruppenelements stets die Gruppenordnung teilt. Hieraus folgt insbesondere, dass Gruppen mit Primzahlordnung (d.h. die Ordnung der Gruppe ist eine Primzahl) zyklisch sind.

7.3.2 Körper

In praktischen Anwendungen betrachtet man häufig Mengen, auf denen nicht nur eine (Gruppen-) Verknüpfung, sondern zwei Verknüpfungen definiert sind. Diese nennt man oft Addition und Multiplikation. Die mathematisch interessantesten Mengen dieser Art sind sogenannte Körper, wie z.B. die Menge der reellen Zahlen.

[8] Das inverse Element ist eindeutig bestimmt, denn sind $x, y \in G$ zwei Inverse zu a, d.h. gilt $ax = xa = e$ und $ay = ya = e$, so folgt $x = xe = x(ay) = (xa)y = ey = y$.

[9] Zyklische Gruppen können grundsätzlich auch unendlich sein wie z.B. die additive Gruppe der ganzen Zahlen. Wir betrachten hier jedoch nur endliche zyklische Gruppen.

Unter einem Körper versteht man in der Mathematik eine Menge K mit den zwei Verknüpfungen Addition und Multiplikation (mit $+$ und $\cdot$ bezeichnet), so dass die folgenden Bedingungen erfüllt sind:

- Die Menge K ist zusammen mit der Verknüpfung $+$ (Addition) eine abelsche Gruppe. Dabei sei 0 das neutrale Element der Verknüpfung $+$.
- Die Menge $K \setminus \{0\}$ (d.h. K ohne das Element 0) ist zusammen mit der Verknüpfung $\cdot$ (Multiplikation) ebenfalls eine abelsche Gruppe. Dabei sei 1 das neutrale Element der Verknüpfung $\cdot$.
- Für alle Elemente a, b und c aus K gilt $c \cdot (a+b) = c \cdot a + c \cdot b$ und $(a+b) \cdot c = a \cdot c + b \cdot c$ (Distributivgesetz).

Körper können endlich viele oder unendliche viele Elemente enthalten — je nachdem nennt man den Körper *endlich* oder *unendlich*. So sind die uns vertrauten Körper der rationalen bzw. der reellen Zahlen unendlich. Beispiele für endliche Körper sind die Primkörper $\mathbb{Z}_p = \{0, 1, 2, 3, \cdots, p-1\}$, p eine Primzahl, versehen mit der Addition modulo p und der Multiplikation modulo p (auch Restklassenkörper genannt).

Charakteristik eines Körpers Die Charakteristik eines Körper K ist die Ordnung des neutralen Elements der Multiplikation (1-Element) bezüglich der Addition, d.h. die kleinste natürliche Zahl n, so dass gilt

$$\underbrace{1 + 1 + \cdots + 1}_{n \text{ mal}} = 0\,,$$

wobei 0 das neutrale Element der Addition ist. Gibt es keine solche natürliche Zahl, d.h. ergibt $1 + 1 + \cdots + 1$ unabhängig von der Zahl der Summanden nie das neutrale Element der Addition 0, so sagt man, der Körper habe Charakteristik 0.

Körper mit Charakteristik 0 haben daher stets die (paarweise verschiedenen) Elemente $1, 1+1, 1+1+1, \ldots$ und sind folglich stets unendlich; andererseits können Körper mit endlicher Charakteristik durchaus endlich oder auch unendlich sein. Ist die Charakteristik n endlich, so muss sie eine Primzahl sein, denn wäre sie zusammengesetzt, d.h. $n = pq$, so sind $p, q < n$ und aufgrund der Minimalität der Charakteristik ist keines der Körperelemente $\bar{p} = \underbrace{1 + 1 + \cdots + 1}_{p \text{ mal}}$, $\bar{q} = \underbrace{1 + 1 + \cdots + 1}_{q \text{ mal}}$ gleich 0. Folglich existieren Inverse $\bar{p}^{-1}, \bar{q}^{-1}$ bezüglich der Multiplikation. Dann ist aber $(\bar{p}\bar{q})(\bar{p}^{-1}\bar{q}^{-1}) = 1$, andererseits ist nach Definition der Charakteristik $\bar{p}\bar{q} = \bar{n} = \underbrace{1 + 1 + \cdots + 1}_{n \text{ times}} = 0$ und somit $\underbrace{(\bar{p}\bar{q})}_{=0}(\bar{p}^{-1}\bar{q}^{-1}) = 0$, was zu einem Widerspruch führt.

Beispiel: Der Körper $\mathbb{Z}_p$, p prim, hat die Charakteristik p. Ist p nicht prim, so ist $\mathbb{Z}_p$ gar kein Körper.

Der einfachste, denkbare Körper ist $\mathbb{Z}_2 = \{0, 1\}$, der nur Null- und Einselement enthält. Dabei ist $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, $1 + 1 = 0$, $1 \cdot 1 = 1$, $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$.

Endliche Körper Wie bereits erwähnt, hat jeder endliche Körper eine Charakteristik $p \neq 0$, wobei p eine Primzahl ist. Zu jeder Primzahl p gibt es einen Körper mit p Elementen, nämlich $\mathbb{Z}_p$.

Die Anzahl der Elemente eines Körpers muss jedoch im allgemeinen keine Primzahl sein. So ist es nicht schwer, einen Körper mit 4 Elementen zu konstruieren[10].

Man kann zeigen, dass die Ordnung jedes Körpers eine Primzahlpotenz (d.h. die Potenz einer Primzahl) ist. Andererseits kann man zu jeder Primzahlpotenz p^n einen Körper konstruieren, der die Ordnung p^n hat. Da zwei endliche Körper mit gleicher Zahl von Elementen nicht unterscheidbar[11] sind, spricht man von **dem Körper mit p^n Elementen** und bezeichnet diesen mit $GF(p^n)$ oder im Amerikanischen mit $\mathbb{F}_p^n$. Dabei steht GF für *Galois Feld* in Erinnerung an den französischen Mathematiker Galois.

Eine besondere Rolle spielen die Körper $GF(p)$, deren Ordnung eine Primzahl ist. Man nennt solche Körper Primkörper zur Primzahl p und bezeichnet ihn meist ebenfalls mit $\mathbb{Z}_p$.[12]

7.4 Elliptische Kurven in der Kryptographie

In der Kryptographie sind elliptische Kurven ein nützliches Werkzeug. Solche Kurven ergeben sich als Lösungen einer Gleichung der Form[13]

$$F(x_1, x_2, x_3) = -x_1^3 + x_2^2 x_3 + a_1 x_1 x_2 x_3 - a_2 x_1^2 x_3 + a_3 x_2 x_3^2 - a_4 x_1 x_3^2 - a_6 x_3^3 = 0. \qquad (7.1)$$

Dabei sind die Variablen x_1, x_2, x_3 sowie die Parameter $a_1, \ldots, a_4, a_6$ Elemente eines gegebenen Kör"pers K. Körper und Parameter müssen so gewählt werden, dass die Kurve bestimmte, für die Kryptographie relevante Eigenschaften besitzt. Der zugrunde liegende Körper K kann einfach die bekannte Menge der reellen Zahlen oder auch ein endlicher Körper sein (vgl. letzter Abschnitt). Damit sich eine sinnvolle Kurve ergibt, müssen die Parameter so gewählt sein, dass die folgenden Nebenbedingungen gelten

$$\frac{\partial F}{\partial x_1} \neq 0, \quad \frac{\partial F}{\partial x_2} \neq 0, \quad \frac{\partial F}{\partial x_3} \neq 0.$$

Ferner betrachten wir Punkte, die sich nur durch eine Vervielfachung jeder Komponente ergeben, als identisch, denn mit (x_1, x_2, x_3) erfüllt stets auch $\alpha(x_1, x_2, x_3)$ $(\alpha \neq 0)$ die Ausgangsgleichung.

---

[10] Die Menge $K = \{0, 1, a, b\}$ ist mit den Verknüpfungen der folgenden Tabellen ein Körper:

$+$	0	1	a	b
0	0	1	a	b
1	1	0	b	a
a	a	b	0	1
b	b	a	1	0

und

$\cdot$	0	1	a	b
0	0	0	0	0
1	0	1	a	b
a	0	a	b	1
b	0	b	1	a

[11] Sind K, K' zwei Körper mit $k = p^n$ Elementen, so gibt es eine eineindeutige Abbildung $\varphi : K \to K'$, die sich mit der Körperarithmetik verträgt. Eine solche Abbildung nennt man Isomorphie. Isomorphe Körper verhalten sich mathematisch gleich, so dass es keinen Sinn macht, zwischen ihnen zu unterscheiden. Z.B. sind $\mathbb{Z}_2$ und $K' = \{NULL, EINS\}$ mit Nullelement $NULL$ und Einselement $EINS$ isomorph. Hierbei sei darauf hingewiesen, dass mathematische Objekte ausschließlich über ihre Eigenschaften definiert sind.

[12] Für Primkörper sind die additive Gruppe sowie die multiplikative Gruppe zyklisch. Ferner enthält jeder Körper $GF(p^n)$ einen zu $\mathbb{Z}_p$ isomorphen Primkörper.

[13] Die hier verwendete Kurve erhält man als Nullstellen des *Polynoms* F vom Grad drei in drei Variablen. Dabei bezeichnet man allgemein Ausdrücke der Form $P = \sum_{i_1,\ldots,i_n \in \mathbb{N}} a_{i_1 \ldots i_n} x_1^{i_1} \ldots x_n^{i_n}$ mit Koeffizienten $a_{i_1 \ldots i_n} \in K$ als Polynome in n Variablen $x_1, \ldots, x_n$ über dem Körper K, wenn $\operatorname{grad} P := \max\{i_1 + \cdots + i_n : a_{i_1 \ldots i_n} \neq 0\}$ einen endlichen Wert hat, die Summe also nur aus endlich vielen Summanden (Monomen) besteht. Die Summe der Exponenten der Variablen jedes einzelnen Summanden ist maximal 3, bei mindestens einem Summanden kommt 3 als Exponentwert einer Variablen auch wirklich vor.

Formal betrachten wir daher Äquivalenzklassen von Punkten (x_1, x_2, x_3), wobei wir zwei Punkte als gleich ansehen, wenn sie durch Multiplikation mit einer skalaren Konstante $\alpha \neq 0$ auseinander hervorgehen.
Setzt man in der Ausgangsgleichung $x_3 = 0$, so wird diese zu $-x_1^3 = 0$, also $x_1 = 0$. Folglich ist die Äquivalenzklasse, die das Element $(0, 1, 0)$ enthält, die einzige Punkt mit $x_3 = 0$. Für alle anderen Lösungspunkte können wir die Transformation

$$K \times K \times (K \setminus \{0\}) \ni (x_1, x_2, x_3) \mapsto (x, y) := \left(\frac{x_1}{x_3}, \frac{x_2}{x_3}\right) \in K \times K$$

vornehmen, die die Anzahl der Variablen von drei auf zwei reduziert. Die Ausgangsgleichung $F(x_1, x_2, x_3) = 0$ war so gewählt, dass sich auf diese Weise die sogenannte Weierstrass-Gleichung[14]

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{7.2}$$

ergibt. Da alle bis auf einen Lösungspunkt durch die Gleichung (7.2) beschrieben werden können, bezeichnet man (7.2) auch oft als die Elliptische Gleichung, ihre Lösungsmenge folglich mit

$$\mathbf{E} = \left\{(x, y) \in K \times K \,|\, y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\right\} \cup \{\mathcal{O}\}.$$

Dabei soll $\mathcal{O}$ den auf diese Weise nicht beschriebenen Punkt $(0, 1, 0)$ darstellen, der durch die Projektion (Division durch x_3) quasi in den unendlich fernen Punkt abgebildet wird.

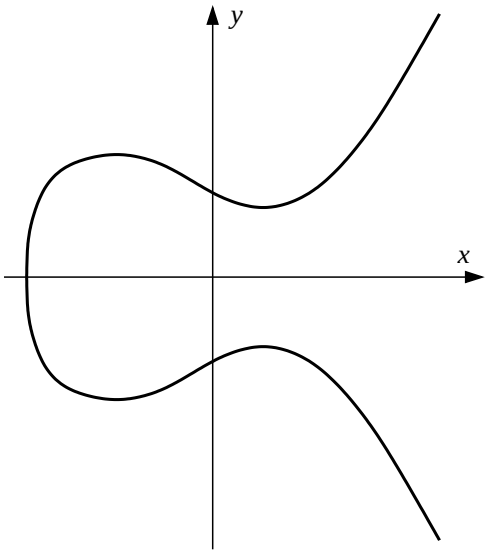


Abbildung 7.3: Beispiel einer Elliptischen Kurve über dem Körper der reellen Zahlen

Als zugrunde liegenden Körper für eine Elliptische Kurve verwendet man in der Kryptographie stets endliche Körper $K = GF(p^n)$, also nicht wie in Abbildung 7.3 die zu einer stetigen Kurve führenden reellen Zahlen. Der Grund liegt, einfach gesagt, darin, dass wir bei der Verarbeitung und Übertragung von Nachrichten stets nur endlich viele Zustände zur Verfügung haben (aufgrund der Arbeitsweise moderner Computer), Körper mit unendlich vielen Elementen wie z.B. die reellen Zahlen daher stets nur unvollständig darstellen können.

In der Praxis hat es sich als sinnvoll erwiesen, entweder $GF(p)$ mit einer großen Primzahl p oder $GF(2^n)$ mit einer (großen) natürlichen Zahl n zu betrachten. Der Grund für die Verwendung

[14] Karl Weierstrass, 31.10.1815−19.12.1897, deutscher Mathematiker, Verfechter der streng formalen Ausrichtung der Mathematik.

des Primkörpers $GF(p)$ liegt in seiner einfachen Arithmetik; andererseits kommt $GF(2^n)$ der binären Darstellung in Computersystemen entgegen. Andere Körper wie z.B. $GF(7^n)$ bieten keiner dieser beiden Vorteile und werden daher in der Praxis nicht verwendet, ohne dass dies theoretische Gründe hätte.

Durch Koordinatentransformation kann man die Weierstrass-Gleichung in einer einfacheren Form schreiben[15]. Je nachdem, ob $p > 3$ ist, verwendet man unterschiedliche Transformationen und erhält so

- im Fall $GF(p)$, $p > 3$, die Elliptische Kurven-Gleichung der Form
$$y^2 = x^3 + ax + b \tag{7.3}$$
mit $4a^3 + 27b^2 \neq 0$
- im Fall $GF(2^n)$ die Elliptische Kurven-Gleichung der Form
$$y^2 + xy = x^3 + ax^2 + b \tag{7.4}$$
mit $b \neq 0$[16].

Durch diese Bedingungen an die Parameter a, b ist gewährleistet, dass die Elliptische Gleichung für kryptographische Anwendungen geeignet ist[17].

Für die Anzahl $|E|$ der Elemente einer Elliptischen Kurve E über einem Körper $GF(k)$ (praktisch $k = p$ prim oder $k = 2^n$) gilt nach dem Satz von Hasse [Sil09] die einfache Beziehung $||E| - k - 1| \leq 2 \cdot \sqrt{k}$. Diese Ungleichung ist äquivalent zu $k+1-2\sqrt{k} < |E| < k+1+2\sqrt{k}$. Dies bedeutet, dass die Anzahl der Elemente der Elliptischen Kurve mit der Größe k gut abgeschätzt werden kann.

7.5 Verknüpfung auf Elliptischen Kurven

Um mit Elliptischen Kurven arbeiten zu können, definiert man eine Verknüpfung (meist additiv als + geschrieben) auf den Punkten der Elliptischen Kurve. Dabei definiert man bei Elliptischen Kurven über $GF(p)$ die kommutative Verknüpfung durch

1. $P + \mathcal{O} = \mathcal{O} + P = P$ für alle $P \in E$,
2. für $P = (x, y)$ und $Q = (x, -y)$ ist $P + Q = \mathcal{O}$,
3. für $P_1 = (x_1, x_2), P_2 = (x_2, y_2) \in E$ mit $P_1, P_2 \neq \mathcal{O}$ und $(x_2, y_2) \neq (x_1, -y_1)$ ist $P_3 := P_1 + P_2$, $P_3 = (x_3, y_3)$ definiert durch
$$x_3 := -x_1 - x_2 + \lambda^2\,, \qquad y_3 := -y_1 + \lambda(x_1 - x_3)$$
mit dem Hilfsquotienten
$$\lambda := \begin{cases} \frac{y_1 - y_2}{x_1 - x_2} & \text{falls } P_1 \neq P_2, \\ \frac{3x_1^2 + a}{2y_1} & \text{falls } P_1 = P_2. \end{cases}$$

[15] Anschaulich bedeutet eine solche Koordinatentransformation eine Drehung bzw. Streckung der Koordinatenachsen, ohne dass die zugrunde liegende Kurve selbst verändert wird.

[16] Die Form (7.3) ist die Standardform der Weierstrass-Gleichung. Ist die Charakteristik des Körpers jedoch 2 oder 3, so ist $4 = 0$ bzw. $27 = 0$, was dazu führt, dass man in der Bedingung an die Parameter a, b wesentliche Informationen verliert. Dies ist ein Hinweis darauf, dass die Transformation auf die Standardform in diesen Fällen nicht zu befriedigenden Ergebnissen führt.

[17] Formal sagt man, die Kurve ist nicht singulär.

Hieraus folgt insbesondere für $P = (x, y) \in E$, dass gilt $-P = (x, -y)$.

Über $GF(2^n)$ definiert man analog die Verknüpfung durch

1. $P + \mathcal{O} = \mathcal{O} + P = P$ für alle $P \in E$,
2. für $P = (x, y)$ und $Q = (x, x + y)$ ist $P + Q = \mathcal{O}$,
3. für $P_1 = (x_1, x_2), P_2 = (x_2, y_2) \in E$ mit $P_1, P_2 \neq \mathcal{O}$ und $(x_2, y_2) \neq (x_1, x_1 + y_1)$ ist $P_3 := P_1 + P_2$, $P_3 = (x_3, y_3)$ definiert durch

$$x_3 := -x_1 + x_2 + \lambda + \lambda^2 + a\,, \qquad y_3 := y_1 + x_3 + \lambda(x_1 + x_3)$$

mit

$$\lambda := \begin{cases} \frac{y_1+y_2}{x_1+x_2} & \text{falls } P_1 \neq P_2, \\ x_1 + \frac{y_1}{x_1} & \text{falls } P_1 = P_2. \end{cases}$$

Hieraus folgt insbesondere für $P = (x, y) \in E$, dass gilt $-P = (x, x + y)$.

(Beachte: $-(-P) = (x, x + (x + y)) = (x, 2x + y) = (x, y)$, da der zugrunde liegende Körper Charakteristik 2 hat.)[18]

Man kann nachrechnen, dass die Menge $E \cap \{\mathcal{O}\}$ mit der so definierten Addition eine Gruppe bildet. Dies bedeutet insbesondere, dass die Summe zweier Kurvenpunkte stets wieder ein Punkt auf der Elliptische Kurve ist. Diese Addition läßt sich auch geometrisch veranschaulichen, wie der folgende Abschnitt zeigt.

[18] Eine Animation der Punktaddition auf Elliptischen Kurven findet man auf der Certicom-Seite unter `https://www.certicom.com/ecc-tutorial` (Datum letzte Änderung unklar).
Vergleiche auch den Web-Link zum Java-Tutorial am Ende dieser Kapitels.

Addieren von Punkten auf einer Elliptischen Kurve

Die zwei folgenden Abbildungen zeigen, wie bei einer Elliptischen Kurve über den reellen Zahlen in affinen Koordinaten zwei Punkte addiert werden. Der unendlich ferne Punkt $\mathcal{O}$ kann nicht in der affinen Ebene dargestellt werden.

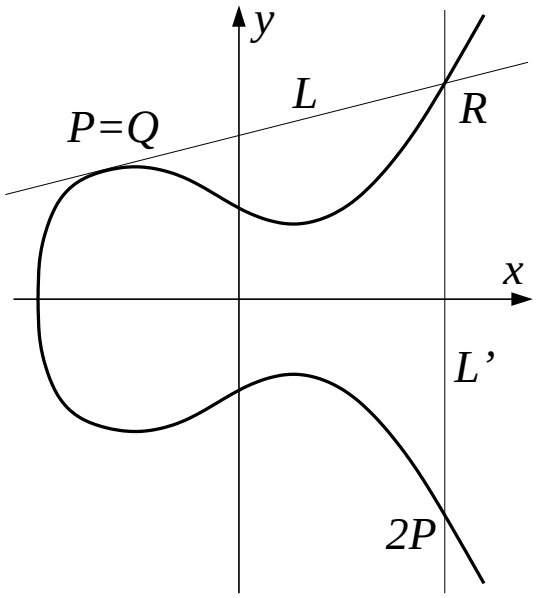


Abbildung 7.4: Verdoppelung eines Punktes

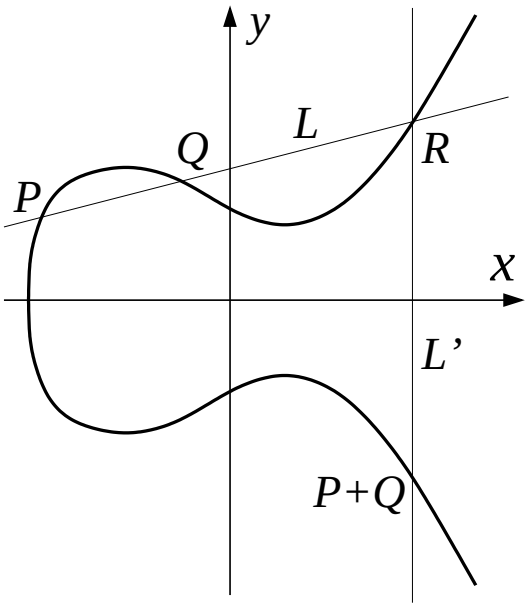


Abbildung 7.5: Addition zweier verschiedener Punkte im Körper der reellen Zahlen

7.6 Sicherheit der Elliptischen-Kurven-Kryptographie: Das ECDLP

Wie bereits in Abschnitt 7.4 erwähnt, betrachten wir in der Kryptographie Elliptische Kurven über diskreten[19] Körpern $GF(2^n)$ oder $GF(p)$ (für große Primzahlen p). Dies bedeutet, dass alle Parameter, die zur Beschreibung der Elliptischen Kurve notwendig sind, aus diesem zugrunde liegenden Körper stammen. Ist nun E eine Elliptische Kurve über einem solchen Körper und P ein Punkt auf der Kurve E, so kann man für jede natürliche Zahl m

$$mP := \underbrace{P + P + \cdots + P}_{m \text{ mal}}$$

bilden. Diese Operation ist aus kryptographischer Sicht deshalb besonders interessant, weil man einerseits um mP zu berechnen im allgemeinen nur $\log m$ Additionen durchführen muss — man bildet einfach P, $2P$, 2^2P, 2^3P, ..., schreibt m binär und addiert schließlich entsprechend der Binärdarstellung von m auf — es andererseits sehr aufwa"ndig zu sein scheint, zu gegebenen Punkten P und $Q = mP$ auf E die Zahl m zu bestimmen. Natürlich kann man die Folge $P, 2P, 3P, 4P, 5P, \ldots$ bilden und jeweils mit Q vergleichen. Hierzu benötigt man jedoch m Additionen.

Bisher ist noch kein Algorithmus bekannt, der effizient m aus P und Q berechnet. Die bisher besten Verfahren liegen z.B. im Fall $GF(p)$ in der Größenordnung $\sqrt{q}$, wobei q ein (großer) Primfaktor von $p-1$ ist; m selbst sollte in diesem Fall zwischen 1 und q liegen, so dass man für die Muliplikation mP maximal $\log q$ Schritte benötigt. Der Quotient $\frac{\sqrt{q}}{\log q}$ strebt jedoch (schnell) gegen $+\infty$.

Sind die Parameter hinreichend groß (ist zum Beispiel p prim und mehr als 160 Bit lang) ist der Computer ohne weiteres in der Lage, sehr schnell (in wenigen Bruchteilen einer Sekunden) den Punkt mP zu bestimmen. Das *inverse Problem*, m aus mP und P zu erhalten, ist jedoch nicht in akzeptabler Zeit möglich.

Dies wird als das „Diskrete Logarithmus Problem über Elliptischen Kurven“ bezeichnet (auch ECDLP – Elliptic Curve Discrete Logarithm Problem – abgekürzt).

Formal betrachten wir in der Elliptischen-Kurven-Kryptographie die Punkte der Kurve als Elemente einer Gruppe mit der Addition als Verknüpfung. Allerdings sind nur solche Elliptischen Kurven für kryptographische Anwendungen geeignet, bei der die Anzahl der Kurvenpunkte hinreichend groß ist. Ferner können in Spezialfällen Elliptische Kurven auch aus anderen Gründen ungeeignet sein. Das bedeutet, dass man bei der Definition einer Kurve auf die Wahl der Parameter achten muss. Denn für bestimmte Klassen von Elliptischen Kurven ist es möglich, das ECDLP leichter zu lösen als im allgemeinen Fall. Kryptographisch ungeeignete Elliptische Kurven sind die sogenannten *anormalen* Kurven (das sind Kurven über $\mathbb{Z}_p$, für die die Menge $\mathbf{E}$ genau p Elemente hat) und die *supersingulären* Kurven (das sind Kurven, für die man das Berechnen des ECDLP auf das Berechnen des „normalen“ Diskreten Logarithmus in anderen endlichen Körper reduzieren, d.h. vereinfachen, kann). Daher gibt es kryptographisch gute und schlechte Kurven. Allerdings kann man für gegebene Parameter a und b mit etwas Aufwand feststellen, ob die resultierende Elliptische Kurve kryptographisch brauchbar ist oder nicht. Die in der Kryptographie eingesetzten Kurven werden meist von Fachleuten zur Verfügung gestellt. Sie gewährleisten, dass die von ihnen als sicher eingestuften Elliptischen Kurven den aktuellen Sicherheitsanforderungen genügen.

[19] Diskret im Gegensatz zu kontinuierlich.

Bei sicheren Kurven wird hauptsächlich durch den Parameter p im Fall des zugrunde liegenden Körpers $GF(p)$ bzw. n im Fall des zugrunde liegenden Körpers $GF(2^n)$ bestimmt, wie lange es dauert, das ECDLP auf dieser Kurve zu lösen. Je größer diese Parameter sind, desto länger nimmt das Lösen des Problems in Anspruch. Von Fachleuten wird z.B. eine Bitlänge von über 200 Bit für den Parameter p empfohlen. Hier wird deutlich, warum die Elliptischen Kurven so interessant für die Kryptographie sind. Denn die Parameter bestimmen auch den Signatur-/Verschlüsselungsaufwand, wenn mit Elliptischen Kurven Kryptographie betrieben wird. Die Dauer einer Schlüsselpaar-Erzeugung ist ebenfalls von den Parametern abhängig. Daher sind kleine Werte (wenige Bits) wünschenswert (möglichst schnelle Laufzeiten der Verfahren); allerdings muss die geforderte Sicherheit dabei eingehalten werden. Mit einer Länge von zum Beispiel 200 Bit für p ist eine *gute* Elliptische Kurve genau so sicher wie ein RSA-Modulus von über 1024 Bit Länge (zumindest nach dem heutigen Forschungstand). Der Grund dafür ist, dass die schnellsten Algorithmen zum Lösen des *Elliptische Kurven Diskreter Logarithmus*-Problems eine exponentielle Laufzeit haben — im Gegensatz zu den subexponentiellen Laufzeiten, die die zur Zeit besten Faktorisierungsalgorithmen haben (Zahlkörpersieb, Quadratisches Sieb oder Faktorisieren mit Elliptischen Kurven). Dies erklärt, warum die Parameter von Kryptoverfahren, die auf dem Problem *Faktorisieren von ganzen Zahlen* beruhen, größer sind als die Parameter von Kryptoverfahren, die auf dem ECDL-Problem basieren.

7.7 Verschlüsseln und Signieren mit Hilfe Elliptischer Kurven

Das *Elliptische Kurven Diskreter Logarithmus Problem* (ECDLP) ist die Grundlage für die Elliptische-Kurven-Kryptographie. Darauf basierend gibt es verschiedene Signaturverfahren. Um ein solches Signaturverfahren anzuwenden, benötigt man:

- Eine Elliptische Kurve $\mathbf{E}$, beschrieben durch den zugrunde liegenden Körper $GF(p^n)$.
- Eine Primzahl $q \neq p$ sowie einen Punkt G auf der Elliptischen Kurve $\mathbf{E}$ mit Ordnung q. D.h., es gilt $qG = \mathcal{O}$ und $rG \neq \mathcal{O}$ für alle $r \in \{1, 2, \ldots, q-1\}$. Die Zahl q muss dann ein Teiler der Gruppenordnung (entspricht der Anzahl der Elemente) $\#\mathbf{E}$ sein. Aufgrund der Primordnung, erzeugt G eine zyklischen Untergruppe von $\mathbf{E}$ mit Ordnung q.

Die genannten Parameter bezeichnet man als *Domain*-Parameter. Durch sie wird festgelegt, auf welcher Elliptischen Kurve $\mathbf{E}$ und in welcher zyklischen Untergruppe von $\mathbf{E}$ ein Signaturverfahren eingesetzt wurde.

7.7.1 Verschlüsselung

Mit Hilfe Elliptischer Kurven kann ein sicherer Schlüsselaustausch nach dem Diffie-Hellman-Protokoll erfolgen (siehe Kapitel 5.4.2). Dieser Schlüssel kann dann für eine anschließende symmetrische Verschlüsselung verwendet werden. Ein Schlüsselpaar mit privatem und öffentlichem Schlüssel wird im Gegensatz zum RSA-Algorithmus nicht erzeugt!

In der Schreibweise der Elliptischen Kurven liest sich das Diffie-Hellman Verfahren wie folgt: Zunächst einigen sich beide Partner (A und B) öffentlich auf eine Gruppe G und eine ganze Zahl q. Danach wählen sie zufällig $r_A, r_B \in \{1, 2, \ldots, q-1\}$, bilden die Punkte $R_A = r_A G$, $R_B = r_B G$ auf der Elliptischen Kurve und tauschen diese aus. Danach berechnet A leicht $R = r_A R_B$. Denselben Punkt (nämlich $r_A r_B G$) erhält auch B, indem er $r_B R_A = r_B r_A G = r_A r_B G = R$ bildet. Dabei ist die Berechnung von R_A, R_B als r_A bzw. r_B-faches des Kurvenpunktes G leicht

durchzuführen; die umgekehrte Operation, aus R_A bzw. R_B den Wert r_A bzw. r_B zu erhalten, ist jedoch sehr aufwändig.
Für einen Dritten ist es nach heutigen Kenntnisstand nicht möglich, R zu berechnen, wenn er nicht mindestens einen der Werte r_A oder r_B ermitteln kann, d.h. das ECDLP löst.

Um einen „Man-in-the-Middle"-Angriff zu verhindern, kann man auch hier wie schon in Kapitel 6.4.1 beschrieben, die übertragenen Werte G, q, R_A, R_B digital signieren.

7.7.2 Signatur-Erstellung

Überträgt man den DSA auf Elliptische Kurve, so kann man wie folgt eine digitale Signatur erzeugen: Man wählt vorab eine (nicht-triviale) Zahl $s \in \mathbb{Z}_q$. Diese bildet den privaten Schlüssel. Hingegen werden q, G und $R = sG$ veröffentlicht. Aus G und R lässt sich jedoch s nicht ermitteln, worauf die Sicherheit des Signaturverfahrens beruht.

Für eine Nachricht m wird zunächst mit Hilfe eines Hash-Verfahrens h ein digitaler Fingerabdruck erstellt, wobei $h(m)$ im Wertebereich $\{0, 1, 2, \ldots, q-1\}$ liegt und $h(m)$ somit als Element von $\mathbb{Z}_q$ interpretiert werden kann. Dann wird ein zufälliges $r \in \mathbb{Z}_q$ gewählt und $R = (r_1, r_2) = rG$ berechnet. Die erste Komponente r_1 von R ist ein Element von $GF(p^n)$. Diese wird auf $\mathbb{Z}_q$ abgebildet, z.B. im Fall $n = 1$ als Element von $\{0, 1, \ldots, p-1\}$ interpretiert und dann der Teilerrest modulo q gebildet. Das so erhaltene Element von $\mathbb{Z}_q$ bezeichnen wir mit $\bar{r}_1$. Nun bestimmt man $x \in \mathbb{Z}_q$ mit

$$rx - s\bar{r}_1 - h(m) = 0.$$

Das Tripel (m, r_1, x) bildet nun die digitale Signatur.

7.7.3 Signatur-Verifikation

Zur Verifikation muss zunächst $u_1 = h(m)/x$, $u_2 = \bar{r}_1/x$ (in $\mathbb{Z}_q$ gebildet werden). Dann bestimmt man

$$V = u_1 G + u_2 Q.$$

Wegen $Q = sG$ ist $V = (v_1, v_2)$ mit $v_1 = u_1 + u_2 s$. Diese Addition findet formal im Raum $GF(p^n)$ statt. Die Projektion von $GF(p^n)$ auf $\mathbb{Z}_q$ sollte jedoch so gewählt sein, dass $\bar{v}_1 = u_1 + u_2 s$ in $\mathbb{Z}_q$ ist. Dann gilt nämlich

$$\bar{v}_1 = u_1 + u_2 s = h(m)/x + \bar{r}_1 s/x = (h(m) + \bar{r}_1 s)/x = rx/x = r.$$

Nun ist $R = rG$. Also folgt hier $\bar{v}_1 = \bar{r}_1$, d.h. R und V stimmen modulo der Projektion auf $\mathbb{Z}_q$ überein.

7.8 Faktorisieren mit Elliptischen Kurven

Es gibt Faktorisierungsalgorithmen[20], die auf Elliptischen Kurven basieren[21]. Genauer gesagt, machen sich diese Verfahren zunutze, dass man auch über $\mathbb{Z}_n$ (n zusammengesetzte Zahl) Elliptische Kurven definieren kann. Elliptische Kurven über $\mathbb{Z}_n$ bilden keine Gruppe, da es nicht zu jedem Punkt auf solchen Elliptischen Kurven einen inversen Punkt geben muss. Dies hängt damit zusammen, dass es – falls n eine zusammengesetzte Zahl ist – in $\mathbb{Z}_n$ Elemente gibt, die kein Inverses bezüglich der Multiplikation modulo n haben. Um zwei Punkte auf einer Elliptischen Kurve über $\mathbb{Z}_n$ zu addieren, kann prinzipiell genauso gerechnet werden wie auf Elliptischen Kurven über $\mathbb{Z}_p$. Eine Addition von zwei Punkten (auf einer Elliptischen Kurve über $\mathbb{Z}_n$) scheitert aber genau dann, wenn man einen Teiler von n gefunden hat. Der Grund dafür ist, dass das Verfahren zum Addieren von Punkten auf Elliptischen Kurven Elemente in $\mathbb{Z}_n$ ermittelt und zu diesen Elementen die inversen Elemente (bezüglich der Multiplikation modulo n) in $\mathbb{Z}_n$ berechnet. Dazu wird der erweiterte Euklidsche Algorithmus benutzt. Ergibt sich nun bei der Addition zweier Punkte (die auf einer Elliptischen Kurve über $\mathbb{Z}_n$ liegen) ein Element aus $\mathbb{Z}_n$, das kein inverses Element in $\mathbb{Z}_n$ hat, so gibt der erweiterte Euklidsche Algorithmus einen echten Teiler von n aus.

Das Faktorisieren mit Elliptischen Kurven funktioniert somit prinzipiell so: Man wählt zufällige Kurven über $\mathbb{Z}_n$, sowie zufällig irgendwelche Punkte (die auf diesen Kurve liegen) und addiert diese; dabei bekommt man wieder Punkte, die auf der Kurve liegen oder findet einen Teiler von n. Die Faktorisierungsalgorithmen auf Basis von Elliptischen Kurven arbeiten also probabilistisch. Durch die Möglichkeit, sehr viele Elliptische Kurven über $\mathbb{Z}_n$ zu definieren, kann man die Wahrscheinlichkeit erhöhen, zwei Punkte zu finden, bei deren Addition ein Teiler von n gefunden wird. Daher eignen sich diese Verfahren auch sehr gut für eine Parallelisierung.

[20] John M. Pollard war an der Entwicklung vieler verschiedener Faktorisierungsalgorithmen beteiligt; auch beim Faktorisieren mit ECC war er einer der führenden Köpfe. Als Mitarbeiter von British Telekom hat er leider nie viel selbst publiziert. Auf der RSA Konferenz 1999 wurde er für seine „outstanding contributions in mathematics“ ausgezeichnet.
Im Jahr 1987 stellte H.W. Lenstra einen häufig genutzten Faktorisierungsalgorithmus vor, der auf Elliptischen Kurven basiert (siehe [Len87]).

[21] Die größten mit Elliptischen Kurven faktorisierten Zahlen haben ca. 80 Dezimalstellen:
`https://members.loria.fr/PZimmermann/records/top50.html`.
Vergleiche auch den Web-Link über das ECMNET-Projekt am Ende dieser Kapitels.

7.9 Implementierung Elliptischer Kurven zu Lehrzwecken

Es gibt relativ wenig freie Programme mit graphischer Oberfläche, die ECC implementieren. Im Folgenden wird aufgezeigt, welche Funktionalität dazu in CrypTool und in SageMath vorhanden ist.

7.9.1 CrypTool

CT1 enthält Elliptische Kurven, um digitale Signaturen zu erzeugen[22] und um die ECC-AES-Hybridverschlüsselung durchzuführen[23].

Implementiert sind die Basisalgorithmen für Gruppenoperationen, für das Erzeugen von Elliptischen Kurven und für das Ein- und Auslesen von Parametern für Elliptische Kurven über endlichen Körpern $GF(p)$ mit p Elementen (p prim). Die Implementierung erfolgte in ANSI C und richtete sich nach dem Entwurf Nr. 8 der Arbeitsgruppe IEEE P1363 *Standard Specifications for Public Key Cryptography*

`http://grouper.ieee.org/groups/1363`.

Implementiert sind die kryptographischen Primitive zur Signaturerzeugung und Signaturverifikation für die auf Elliptischen Kurven basierenden Varianten von Nyberg-Rueppel-Signaturen und DSA-Signaturen.

Schritt-für-Schritt ist die Punkt-Addition auf elliptischen Kurven in CT1 and JCT visualisiert.[24]

7.9.2 SageMath

In SageMath finden sich sehr gute Beschreibungen über Elliptische Kurven unter:[25]

- `http://doc.sagemath.org/html/en/constructions/elliptic_curves.html`
- `http://doc.sagemath.org/html/en/reference/plane_curves/index.html#elliptic-curves`

Zusätzlich gibt es ein ausführliches, interaktives ECC-Tutorial von Maike Massierer. Diese Einführung in die Elliptische-Kurven-Kryptographie (ECC) ist als SageMath-Notebook aufgebaut.

SageMath-Notebooks werden im Browser nach einem Logon-Vorgang aufgerufen[26,27].

---

[22] Die Dialogbox, die in CT1 nach dem Menü **Digitale Signaturen/PKI \ Dokument signieren** erscheint, bietet die EC-Verfahren ECSP-DSA und ECSP-NR an.

[23] In CT1 finden Sie dieses Verfahren über das Menü **Ver-/Entschlüsseln \ Hybrid**.

[24] CT1: Menü **Digitale Signaturen/PKI \ Signaturdemo (Signaturerzeugung)**,
JCT (Standard-Perspektive): Menü **Visualisierungen \ Elliptische Kurven-Berechnungen**.

[25] SageMath-Beispiele dazu finden sich z.B. auch in dem „Elliptic Curve Cryptography (ECC) Tutorial“ `http://www.williamstein.org/simuw06/notes/notes/node12.html`

[26] Hat man SageMath auf einem eigenen (Unix-)Server installiert, muss man auf der SageMath-Kommandozeile erst den Befehl `notebook()` aufrufen.

[27] Das ECC-Notebook von Massierer benötigt die KASH3-Bibliothek: Deshalb muss (z.B. für SageMath 4.2.1)das Package „kash3-2008-07-31.spkg“ installiert worden sein (Befehl `sage -i`).

Das ECC-Notebook wurde 2008 von Massierer[28] erstellt und besteht aus 8 Teilen („Titelseite“ mit Inhaltsverzeichnis plus 7 Kapitel) und zielt darauf ab, dass selbst ein Einsteiger versteht, was Elliptische Kurven sind (es ist nur auf Englisch vorhanden):

0. ECC Notebook (title page and contents)
1. Introduction and Overview
2. Motivation for the use of Elliptic Curves in Cryptography
3. Elliptic Curves in Cryptography
4. Cryptographic Protocols in ECC
5. Domain Parameter Generation for ECC Systems
6. Conclusion and Further Topics
7. References

[28] Anleitung zur Benutzung eines interaktiven SageMath-Notebooks: Aktualisieren für die neue SageMathCloud xxxxxxxxxxxxx

- Manche SageMath-Server sind öffentlich und bieten Worksheets als „Published Worksheets“ an, die man ohne Log-in ausführen kann. Diese Worksheets werden aufgelistet, wenn man auf „Published“ in der oberen rechten Ecke klickt.
- Worksheets, die den `Interact`-Befehl nutzen, erfordern z.Zt. einige weitere Schritte vom Benutzer: Einloggen, Kopie erstellen, alle Kommandos nochmal ausführen.

- Teile des ECC-Tutorial benutzen einen speziellen Mathematik-Font, der standardmäßig bei den meisten Browser nicht mitinstalliert wird. Wenn Sie bemerken, dass Formeln nicht korrekt dargestellt sind oder Ihr Browser meldet, dass Fonts fehlen, installieren Sie bitte die Fonts jsMath für eine bessere Darstellung.
  Siehe `http://www.math.union.edu/~dpvc/jsMath/`.
  Nach der Installation dieser Fonts hat man das jsMath-Symbol am unteren Rand des Browsers. Klickt man dieses Symbol an, erhält man die Download-Seite dieser TIFF-Fonts. Diese Font-Installation muss an jedem PC einzeln erfolgen.

7.10 Patentaspekte

Wenn man statt des Primkörpers $GF(p)$ einen Körper der Form $GF(2^n)$ zugrunde legt, ergeben sich wesentliche Unterschiede in der Implementierung. Der Vorteil einer Implementierung unter Verwendung von $GF(2^n)$ liegt darin, dass Rechnungen aufgrund der binären Darstellung effizienter durchgeführt werden können. Dies betrifft insbesondere die Division, die in $GF(p)$ vergleichsweise aufwändig ist (was z.B. bei dem oben beschriebenen Signaturverfahren sowohl die Erstellung der Signatur als auch ihre spätere Verifikation betrifft, da beide eine Divisionsoperation enthalten).

Um das Potenzial der Effizienzsteigerung möglichst optimal zu nutzen, kann man z.B. Körper wählen, die besondere Basen besitzen, wie Polynomialbasen (besonders geeignet für Software-Implementierungen) oder Normalbasen (bevorzugt bei Hardware-Implementierungen). Für bestimmte Werte von n (wie z.B. $n = 163, 179, 181$) lassen sich sogar beide Vorteile kombinieren. Allerdings sind spezielle Darstellungen oft nicht Standard.

Um Speicherplatz zu sparen, wird zuweilen nur die erste Komponente sowie ein weiteres Bit für jeden Punkt auf der Elliptischen Kurve gespeichert. Hieraus kann jeweils der gesamte Punkt errechnet werden. Dies ist besonders bei Verwendung von Normalbasen effizient. Selbst bei der Durchführung der kryptographischen Protokolle kann eine signifikante Beschleunigung erreicht werden. Diese sogenannte *Punkt-Kompression*, die bei der Hälfte aller Kurven einsetzbar ist, ist jedoch patentiert (US Patent 6141420, Certicon) und daher nicht ohne weiteres frei einsetzbar.

Im allgemeinen Fall $GF(p^n)$ (aber auch für $n = 1$) werden oft sogenannte affine oder projektive Koordinaten eingesetzt, was je nach Anwendung zu Effizienzgewinnen führt.

Eine vollständige Beschreibung aller Implementierungen unter Abwägung ihrer Vor- und Nachteile würde an dieser Stelle zu weit führen. Abschließend kann festgehalten werden, dass die Vielfalt möglicher Implementierungen bei Elliptischen Kurven z.B. im Vergleich zu RSA-Implementierungen sehr groß ist. Aus diesem Grund gibt es Bestrebungen, sich auf wenige Standardimplementierungen, ja sogar auf eine kleine Schar fest vorgegebener Kurven zu beschränken (ASC-Ansatz).

Der Erfolg dieser Standardisierungsbemühungen ist noch nicht absehbar. Dies wäre aber eine Voraussetzung dafür, dass sich ECC dauerhaft als Alternative zu RSA etabliert. Die Arbeit der Standardisierungskomitees wird sich erheblich beschleunigen müssen, wenn sich abzeichnet, dass die aktuellen Forschungsprojekte im Bereich der Faktorisierung einen Durchbruch erzielen.

Aktuelle Informationen zur Patentlage finden sich hier[29].

7.11 Elliptische Kurven im praktischen Einsatz

Bereits heute werden Elliptische Kurven in der Praxis breit eingesetzt. Als prominentes Beispiel ist hier der Informationsverbund Bonn-Berlin (IVBB)[30] zu nennen, bei dem streng vertrauliche Dokumente der deutschen Bundesregierung zwischen Regierungsstellen mit Sitz in Berlin und Bonn ausgetauscht werden. Durch den Einsatz von ECC konnte eine Hochsicherheitslösung realisiert werden. Fragen der Interoperabilität spielten hingegen eine untergeordnete Rolle.

[29] `https://en.wikipedia.org/wiki/Elliptic_curve_cryptography`
`https://en.wikipedia.org/wiki/ECC_patents`
`https://cr.yp.to/ecdh/patents.html` Bernstein, Daniel J. (2006-05-23): „Irrelevant patents on elliptic-curve cryptography“. Abgerufen 2016-07-14.

[30] Der IVBB verbindet Regierungsstellen in der alten und neuen deutschen Hauptstadt.

In Österreich gibt es eine Massenanwendung auf Basis von ECC: Die Bankkarte mit Signaturfunktion.

Beide Beispiele zeigen typische Einsatzgebiete Elliptischer Kurven: Als Hochsicherheitslösungen und bei Implementierungen auf Smartcards, bei denen der Schlüssellänge (begrenzter Speicherplatz) eine entscheidende Bedeutung zukommt.

Literaturverzeichnis (EllCurves)

[Len87] Lenstra, H. W.: *Factoring Integers with Elliptic Curves.* Annals of Mathematics, 126:649–673, 1987.

[LV01] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes (1999 + 2001).* Journal of Cryptology, 14:255–293, 2001.
`http://www.cs.ru.nl/E.Verheul/papers/Joc2001/joc2001.pdf`,
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.69&rep=rep1&type=pdf`.

[Sil09] Silverman, Joe: *The Arithmetic of Elliptic Curves.* Graduate Texts in Mathematics 106. Springer, 2. Auflage, 2009, ISBN 978-0-387-09493-9.

[SWE15] Schulz, Ralph Hardo, Helmut Witten und Bernhard Esslinger: *Rechnen mit Punkten einer elliptischen Kurve.* LOG IN, 2015(181/182):103–115, 2015. Geschrieben für Lehrer; didaktisch aufbereitet, leicht verständlich, mit vielen SageMath-Beispielen.
`http://bscw.schule.de/pub/nj_bscw.cgi/d1024028/Schulz_Witten_Esslinger-Rechnen_mit_Punkten_einer_elliptischen_Kurve.pdf`.

Alle Links wurden am 14.07.2016 überprüft.

Web-Links

1. Umfassende interaktive Einführung in elliptische Kurven und Elliptische-Kurven-Kryptographie (ECC) mit SageMath von Maike Massierer und dem CrypTool-Team (alles in Englisch).
   ECC-Tutorial als SageMath-Notebook, Version 1.3, Januar 2011 `http://web.maths.unsw.edu.au/~maikemassierer/ecc-notebook/`

2. Online-Tutorial über Elliptische Kurven der Firma Certicom (alles in Englisch),
   `https://www.certicom.com/index.php/ecc-tutorial`

3. Overview of Elliptic Curve Cryptosystems,
   Revised June 27, 1997. M.J.B. Robshaw und Yiqun Lisa Yin.
   RSA Laboratories (alles in Englisch),

   `http://www.emc.com/emc-plus/rsa-labs/historical/overview-elliptic-curve-cryptosystems.htm`

4. Tutorial mit Java Applets – Krypto-Verfahren basierend auf elliptischen Kurven,
   Diplomarbeit Thomas Laubrock, 1999,
   `http://www.warendorf-freckenhorst.de/elliptische-kurven/frame.html`

5. Arbeitsgruppe IEEE P1363,
   `http://grouper.ieee.org/groups/1363`

6. Eine informative Seite zum Faktorisieren mit Elliptischen Kurven,
   Dort findet man Literatur zum Thema Faktorisieren mit Elliptischen Kurven sowie Links zu anderen ECC-Seiten.
   `https://members.loria.fr/PZimmermann/records/ecmnet.html`

7. BSI TR-02102-1,
   Technische Richtlinie „Kryptographische Verfahren: Empfehlungen und Schlüssellängen“ 15. Februar 2016.

   `https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm.html`
   `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=2`

Alle Links wurden am 14.07.2016 überprüft.

Kapitel 8

Einführung in die Bitblock- und Bitstrom-Verschlüsselung

(Klaus Pommerening, Januar–Juni 2015; Updates: Jan. 2016, Apr. 2016)

Während asymmetrische Verschlüsselung meistens zahlentheoretische Methoden verwendet, beruhen die heutigen symmetrischen Verschlüsselungsverfahren in der Regel auf Boolescher Algebra, d. h., auf Manipulationen von Bits. Das ist eine ganz andere Art von Mathematik, für Einsteiger vielleicht ungewohnt, so dass dieses Kapitel eine sanfte Einführung in dieses mathematische Gebiet sein soll. Vorkenntnisse aus einem Grundkurs Mathematik am Gymnasium sollen ausreichen. Ohne weitere Erklärung als bekannt angenommen werden die Begriffe „Variable“ und „Funktion“, auch für Argumente und Werte in anderen Mengen als den reellen Zahlen.

Beginnen wir also mit der Beschreibung, wie Bits interpretiert, verknüpft und durch Funktionen, sogenannte Boolesche Funktionen, verarbeitet werden. Namenspatron dieses Fachgebiets ist George Boole[1], der durch die Einführung der elementaren logischen Operationen die Logik mathematisch formalisierte („Logikkalkül“). Moderne symmetrische Verschlüsselungsverfahren, wie auch Hash-Funktionen, werden durch Systeme von Booleschen Funktionen beschrieben.

Der Schwerpunkt dieses Kapitels liegt in der Einführung in die mathematischen Grundlagen der Verschlüsselungstechniken, die auf Bits operieren. Konkrete Verfahren werden nicht detailliert beschrieben; hierfür sei auf die Bücher von Menezes/Orschot/Vanstone [MvOV01], Oppliger [Opp11], Paar und Pelzl [PP09], Schmeh [Sch03, Sch16] und Stamp [SL07] verwiesen.

Noch ein Wort zur Nomenklatur: Die Verfahren werden in der Literatur meist als „Blockchiffren“ oder „Stromchiffren“ bezeichnet, ohne das Präfix „Bit-“. Das ist manchmal missverständlich, da – besonders bei Stromchiffren – auch andere Zeichensätze (Alphabete, Buchstaben) als kleinste Einheiten verwendet werden. Der Deutlichkeit halber sollte man im Zweifelsfall also die „Bits“ mit in die Bezeichnung aufnehmen.

Das Thema dieses Kapitels ist also mit anderen Worten

Symmetrische Verschlüsselung von mit Bits dargestellten Informationen.

Die mathematischen Grundlagen und Methoden gehören zu den Gebieten

Boolesche Algebra und endliche Körper.

[1] George Boole, englischer Mathematiker, Logiker und Philosoph, 2.11.1815–8.12.1864.

8.1 Boolesche Funktionen

8.1.1 Bits und ihre Verknüpfung

Die Objekte, mit denen Computer auf der untersten Software-Ebene operieren, sind Bits oder Gruppen von Bits (z. B. Bytes, die meist aus 8 Bits bestehen, oder „Wörter“, je nach Computer-Architektur meist 32 oder 64 Bits). Der Umgang mit den Bits 0 und 1 und mit den elementaren logischen Operationen wie „und“ (AND), „oder“ (OR), „nicht“ (NOT) und „exklusives oder“ (XOR) ist zwar den meisten vertraut, soll aber hier kurz beschrieben werden, auch um die verwendete Terminologie einzuführen.

Bits können logisch interpretiert werden als die Wahrheitswerte „wahr“ (True, T) und „falsch“ (False, F). Sie können auch algebraisch interpretiert werden als die Werte 0 (entspricht F) und 1 (entspricht T). Mathematisch gesprochen sind sie dann Elemente der zweielementigen Menge $\{0, 1\}$, die wir hinfort in diesem Kapitel mit $\mathbb{F}_2$ bezeichnen werden; warum, wird gleich erklärt:

Betrachtet man nämlich den Restklassenring von $\mathbb{Z}$ modulo 2, so hat dieser zwei Elemente und ist ein Körper, da 2 eine Primzahl ist. Die Addition in diesem Körper entspricht genau der logischen Verknüpfung XOR, die Multiplikation der logischen Verknüpfung AND, wie man in Tabelle 8.1 sieht. Tabelle 8.2 listet die Umrechnungsformeln zwischen den elementaren logischen und algebraischen Operationen auf.

logisch					**algebraisch**			
Bits		Verknüpfung			Bits		Verknüpfung	
x	y	OR	AND	XOR	x	y	$+$	$\cdot$
F	F	F	F	F	0	0	0	0
F	T	T	F	T	0	1	1	0
T	F	T	F	T	1	0	1	0
T	T	T	T	F	1	1	0	1

Tabelle 8.1: Die wichtigsten Verknüpfungen von Bits. Dabei ist das logische XOR identisch mit dem algebraischen $+$, das logische AND mit dem algebraischen $\cdot$ (Multiplikation).

algebraisch nach logisch		
$x + y$	$=$	$(x \vee y) \wedge (\neg x \vee \neg y)$
$x \cdot y$	$=$	$x \wedge y$

logisch nach algebraisch		
$x \vee y$	$=$	$x + y + x \cdot y$
$x \wedge y$	$=$	$x \cdot y$
$\neg x$	$=$	$1 + x$

Tabelle 8.2: Umrechnung der algebraischen Operationen in logische und umgekehrt

Da die algebraische Struktur als Körper für die Kryptographie eine herausragende Rolle spielt, wird hier die in der Algebra übliche Bezeichnung für endliche Körper $\mathbb{F}_q$ (oft auch $\mathrm{GF}(q)$ für „Galois[2] Field“, dabei ist q die Anzahl der Elemente) übernommen[3]. In diesem Kontext ist es sinnvoll, für die Verknüpfungen die algebraischen Symbole $+$ (für XOR) und $\cdot$ (für AND) zu

[2] Évariste Galois, französischer Mathematiker, 25.10.1811–31.5.1832.
[3] Auch SageMath verwendet die Bezeichnung $\mathrm{GF}(q)$.

verwenden, wobei der Multiplikationspunkt wie auch sonst in der Mathematik oft weggelassen wird. Kryptographen benutzen gerne auch die Symbole $\oplus$ und $\otimes$, die allerdings in der Mathematik mit ganz anderen Bedeutungen[4] belegt sind und daher in diesem Text – abgesehen von Diagrammen – meist vermieden werden.

Zur Verdeutlichung sei noch explizit auf einige Besonderheiten des algebraischen Rechnens im binären Fall (d. h., in Charakteristik 2) hingewiesen:

- In einer Summe heben sich zwei gleiche Summanden gegenseitig weg, d. h., sie ergeben zusammen 0. Allgemeine Regel: $x + x = 0$ oder $2x = 0$.
- Allgemeiner ergibt eine gerade Anzahl gleicher Summanden immer 0, während eine ungerade Anzahl gleicher Summanden genau diesen Summanden ergibt. Allgemeine Regel:

$$m\,x := \underbrace{x + \cdots + x}_{m} \quad = \quad \begin{cases} 0 & \text{für gerades } m \\ x & \text{für ungerades } m. \end{cases}$$

- Bei algebraischen Umformungen ist eine Subtraktion dasselbe wie eine Addition; man kann Plus- und Minuszeichen beliebig gegeneinander austauschen. Allgemeine Regel: $x + y = x - y$.
- Alle drei binomischen Formeln, also für $(x + y)^2$, $(x - y)^2$, $(x + y)(x - y)$, fallen zu einer einzigen zusammen:

$$(x + y)^2 = x^2 + y^2.$$

  Denn das doppelte Produkt ist 0.

8.1.2 Beschreibung Boolescher Funktionen

Definieren wir zunächst ganz naiv: Eine **Boolesche Funktion** ist eine Vorschrift (oder eine Rechenregel oder ein Algorithmus), die aus einer bestimmten Anzahl von Bits ein neues Bit erzeugt. Bevor wir diese Definition mathematisch präzise fassen (siehe Definition 8.1.1), soll sie zunächst etwas anschaulicher gemacht werden.

Für eine vertiefte Darstellung sei auf [CS09] oder [Pom08, Pom14] sowie die beiden Artikel von Claude Carlet[5] in [CH10][6] verwiesen.

Als ganz einfaches Musterbeispiel dient die Funktion AND: Sie nimmt zwei Bits entgegen und erzeugt daraus ein neues Bit nach der bekannten Verknüpfungsregel des logischen „und“, siehe Tabelle 8.1.

Als etwas komplizierteres Beispiel möge die Funktion f_0 dienen, die aus drei Bits x_1, x_2 und x_3 den Wert

$$f_0(x_1, x_2, x_3) = x_1 \text{ AND } (x_2 \text{ OR } x_3) \tag{8.1}$$

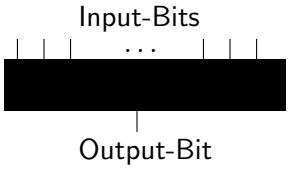
berechnet.

Veranschaulichen kann man sich eine Boolesche Funktion durch eine „Black Box“:

[4] direkte Summe und Tensorprodukt von Vektorräumen

[5] siehe auch dessen Publikationsverzeichnis unter `http://www.math.univ-paris13.fr/~carlet/pubs.html`

[6] online zu finden unter `http://www.math.univ-paris13.fr/~carlet/chap-fcts-Bool-corr.pdf` und `http://www.math.univ-paris13.fr/~carlet/chap-vectorial-fcts-corr.pdf`



Was innerhalb dieser „Black Box“ passiert, kann man auf verschiedene Arten beschreiben:

- **mathematisch** durch eine Formel,
- **informatisch** durch einen Algorithmus,
- **technisch** durch ein Schaltnetz (oder Schaltdiagramm),
- **pragmatisch** durch eine Wahrheitstafel (das ist die Wertetabelle).

Die Beispielfunktion f_0 ist mathematisch definiert in der Gleichung (8.1). Der entsprechende Algorithmus wird hier am besten ebenfalls durch diese Formel beschrieben, da keinerlei Verzweigungen oder bedingte Anweisungen nötig sind. Als Schaltnetz kann man f_0 etwa wie in Abbildung 8.1 visualisieren. Die Wahrheitstafel gibt zu jedem Input-Tripel einfach den Wert von f_0 an, siehe Tabelle 8.3.

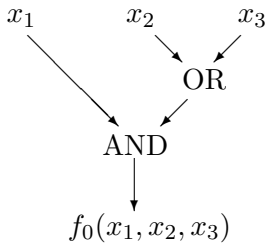


Abbildung 8.1: Beispiel eines Schaltnetzes

x_1	x_2	x_3	$f_0(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabelle 8.3: Beispiel einer Wahrheitstafel

Die Bezeichnung „Wahrheitstafel“ kommt von der Interpretation der Bits im Logikkalkül: 0 (= F) bedeutet „falsch“, 1 (= T) bedeutet „wahr“. Der Wert $f(x_1, \ldots, x_n)$ einer Booleschen Funktion f sagt dann, ob der gesamte Ausdruck wahr oder falsch ist, wenn die einzelnen Input-Bits $x_1, \ldots, x_n$ die angegebenen Wahrheitswerte haben.

Die Verbindung zur Technik, also die Beziehung zwischen Logikkalkül und elektrischen Schaltungen, wurde im Wesentlichen von Shannon[7] entwickelt.

[7] Claude Elwood Shannon, amerikanischer Mathematiker und Elektrotechniker, 30.4.1916–24.2.2001.

8.1.3 Die Anzahl Boolescher Funktionen

Die obige Wahrheitstafel für f_0 suggeriert eine einfache Abzählung aller Booleschen Funktionen: Bei drei Variablen gibt es $8 = 2^3$ verschiedene Input-Tripel, denn jedes einzelne Input-Bit kann unabhängig von den beiden anderen Bits die Werte 0 oder 1 annehmen. Eine Boolesche Funktion f wiederum kann für jedes Input-Tripel unabhängig von den sieben anderen Tripeln 0 oder 1 werden, das sind 8 unabhängige Möglichkeiten für 0 oder 1, also insgesamt 2^8. Also gibt es $256 = 2^8$ Boolesche Funktionen von drei Variablen.

Im allgemeinen Fall haben wir $N = 2^n$ verschiedene Besetzungen für die n Input-Variablen, und für jeden dieser N Inputs kann die Funktion 0 oder 1 werden, das macht 2^N verschiedene Möglichkeiten. Die allgemeine Formel ist also:

Satz 8.1.1. *Es gibt genau 2^{2^n} verschiedene Boolesche Funktionen von n Variablen.*

Bei vier Variablen sind das schon $2^{16} = 65536$ Stück, und die Formel sagt, dass die Anzahl superexponenziell anwächst: Der Exponent wächst ja selbst schon exponenziell.

Alle 16 Booleschen Funktionen von zwei Variablen sind im Abschnitt 8.1.7, Tabelle 8.4, aufgelistet.

8.1.4 Bitblöcke und Boolesche Funktionen

Für Gruppierungen von Bits gibt es je nach Kontext verschiedene Bezeichnungen[8]: Vektoren, Listen, (n-) Tupel, ..., bei bestimmten Größen auch spezielle Bezeichnungen wie Bytes (für 8 Bits), Wörter (für 32 oder 64 Bytes, je nach Prozessorarchitektur) ... In diesem Kapitel wird überwiegend die in der Kryptographie gängige Bezeichnung „Bitblöcke“ verwendet. Ein **Bitblock** der Länge n ist also eine Liste $(x_1, \ldots, x_n)$ von Bits. Hierbei kommt es auf die Reihenfolge an. Es gibt acht verschiedene Bitblöcke der Länge 3. Dies sind sie:

$$(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1).$$

Gelegentlich werden sie, wenn dadurch kein Missverständnis zu befürchten ist, auch ohne Klammern und Kommas als Bitketten geschrieben[9]:

$$000, 001, 010, 011, 100, 101, 110, 111.$$

Oft wird die abgekürzte Schreibweise x für $(x_1, \ldots, x_n)$ verwendet, die ausdrückt, dass Bitblöcke Objekte „eigenen Rechts“ sind. Die 2^n verschiedenen Bitblöcke der Länge n sind genau die Elemente des kartesischen Produkts $\mathbb{F}_2^n = \mathbb{F}_2 \times \cdots \times \mathbb{F}_2$. Dieses kartesische Produkt hat eine „natürliche“ Vektorraum-Struktur – man kann Bitblöcke x und $y \in \mathbb{F}_2^n$ addieren und mit Skalaren $a \in \mathbb{F}_2$ multiplizieren:

$$(x_1, \ldots, x_n) + (y_1, \ldots, y_n) = (x_1 + y_1, \ldots, x_n + y_n),$$

$$a \cdot (x_1, \ldots, x_n) = (a \cdot x_1, \ldots, a \cdot x_n).$$

Damit können wir nun die mathematisch exakte Definition formulieren:

[8] Begrifflich beschreiben sie das gleiche. In Python bzw. SageMath entsprechen den verschiedenen Bezeichnungen aber z. T. unterschiedliche Typen.

[9] Manchmal werden sie auch in Spaltenform, als $n \times 1$-Matrizen, geschrieben, wenn die Interpretation als Vektor im Vordergrund steht.

Definition 8.1.1. *Eine* **Boolesche Funktion von n Variablen** *ist eine Abbildung*

$$f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2.$$

Eine solche nimmt als Argument also einen Bitblock der Länge n und produziert daraus ein Bit.

Die Menge aller Booleschen Funktionen auf $\mathbb{F}_2^n$ wird im Folgenden gelegentlich mit $\mathcal{F}_n$ bezeichnet. Nach Satz 8.1.1 hat sie 2^{2^n} Elemente.

Konvention: Beschreibt man eine Boolesche Funktion durch ihre Wahrheitstafel, so ordnet man diese, wie auch oben im Beispiel schon gesehen, in der Regel lexikographisch[10] nach $x \in \mathbb{F}_2^n$; diese Ordnung ist, anders ausgedrückt, die natürliche Ordnung der Zahlen $a = 0, \ldots, 2^n - 1$, wenn diese binär als

$$a = x_1 \cdot 2^{n-1} + \cdots + x_{n-1} \cdot 2 + x_n$$

dargestellt und auf diese Weise den Bitblöcken $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$ zugeordnet werden.

8.1.5 Logische Ausdrücke und disjunktive Normalform

Für die mathematische Beschreibung Boolescher Funktionen, also wie oben gesagt die Beschreibung durch eine Formel, sind im wesentlichen außer der Wahrheitstafel zwei Ansätze gebräuchlich:

- In der Logik werden Boolesche Funktionen durch Disjunktionen (die Operation OR, auch $\vee$ geschrieben), Konjunktionen (die Operation AND, auch $\wedge$ geschrieben) und Negationen (die Operation NOT, auch $\neg$ geschrieben) ausgedrückt. Zusammensetzungen dieser Operationen heißen **logische Ausdrücke**.
- In der Algebra werden Boolesche Funktionen durch die Addition $+$ und die Multiplikation $\cdot$ des Körpers $\mathbb{F}_2$ ausgedrückt. Zusammensetzungen dieser Operationen heißen **(binäre) polynomiale Ausdrücke**[11].

Wir werden bald sehen, dass man auf beide Weisen alle Booleschen Funktionen beschreiben kann und dass dabei sogar zusätzliche Anforderungen an die Gestalt der Formeln, sogenannte Normalformen, gestellt werden können. Selbstverständlich kann man auch für jede Boolesche Funktion zwischen den drei Darstellungsarten Wahrheitstafel, logischer Ausdruck und binärer polynomialer Ausdruck hin- und herwechseln. Dass die Algorithmen dafür bei großer Zahl n von Variablen effizient sind, ist aber nicht zu erwarten, denn schon allein das Aufschreiben einer Wahrheitstafel erfordert 2^n Bits. Für die algorithmische Behandlung Boolescher Funktionen in SageMath siehe auch Anhang 8.4.

[10] Bei einer lexikographischen Ordnung werden die zu ordnenden Zeichenketten (wie in einem Lexikon) nach der Größe des ersten Zeichens – hier 0 oder 1 mit $0 < 1$ – geordnet. Falls dieses gleich ist, nach der Ordnung des zweiten Zeichens usw. Lexikographisch geordnet ist die Folge $011, 100, 101$. Nicht lexikographisch geordnet die Folge $100, 101, 011$, weil hier die dritte Zeichenkette mit einer 0 beginnt, die kleiner ist als das Anfangszeichen der davor stehenden Zeichenkette. Die zu Beginn von 8.1.4 aufgeschriebene Reihenfolge der acht Bitblöcke der Länge 3 folgt der lexikographischen Ordnung.

[11] Nicht polynomial wären Ausdrücke, in denen andere Verknüpfungen vorkommen. Bei Zahlen könnte man hier auch daran denken, Inputvariablen als Exponenten zu verwenden, das ergibt bei den Booleschen Variablen 0 und 1 allerdings keinen rechten Sinn.

Die algebraische Form scheint für kryptologische Zwecke aufgrund ihrer (noch zu erkundenden) Strukturiertheit etwas besser zu handhaben sein. Die logische Form führt dagegen einfacher zu einer Hardware-Realisierung durch ein Schaltnetz, weil die elementaren Booleschen Operationen direkte Entsprechungen in Schaltelementen („Gatter“) haben.

Da die logische Form im Folgenden eine geringere Rolle spielt, wird das Ergebnis hier ohne weitere Begründung angegeben; die bloße Möglichkeit der Darstellung durch die logischen Operationen (ohne Normalisierung) folgt im Abschnitt 8.1.7 noch einmal als Nebenergebnis, siehe Satz 8.1.5.

Satz 8.1.2. *Jede Boolesche Funktion von n Variablen $x_1, \ldots, x_n$ lässt sich mit einem geeigneten r in der Form (Konjunktion)*

$$f(x) = s_1(x) \wedge \ldots \wedge s_r(x)$$

schreiben, wobei die $s_j(x)$ für $j = 1, \ldots, r$ jeweils die Gestalt (Disjunktionen)

$$s_j(x) = t_{j1}(x) \vee \ldots \vee t_{jn_j}(x)$$

mit einer Anzahl n_j von Termen $t_{jk}(x)$ ($j = 1, \ldots, r$ und $k = 1, \ldots, n_j$) haben, die selbst jeweils von der Gestalt x_i (Input-Bit) oder $\neg x_i$ (negiertes Input-Bit) für jeweils einen Index i sind[12].

Mit anderen Worten: Man kann jede Boolesche Funktion aufbauen, indem man einige Ausdrücke (die $s_j(x)$) durch OR-Verknüpfung von einigen der Input-Bits oder deren Negation bildet, und diese Ausdrücke dann mit AND verbindet („Konjunktion von Disjunktionen“). Die AND- und OR-Verknüpfungen sind in dieser „Normalform“ also sauber in zwei Schichten getrennt, eine weitere Vermischung kommt nicht vor. Die Beispielfunktion f_0 aus Abschnitt 8.1.2 hat die Definitionsgleichung

$$f_0(x_1, x_2, x_3) = \underbrace{x_1}_{s_1(x)} \wedge \underbrace{(x_2 \vee x_3)}_{s_2(x)}.$$

Diese hat schon die gewünschte „konjunktive“ Form aus Satz 8.1.2 mit

$$n_1 = 1,\ \ s_1(x) = t_{11}(x) = x_1, \quad n_2 = 2,\ \ t_{21}(x) = x_2,\ \ t_{22}(x) = x_3.$$

Das gilt nicht mehr, wenn man sie in expandiert:

$$f_0(x) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3).$$

Negierte Input-Bits kommen in diesem Beispiel nicht vor. Solche sieht man aber in Tabelle 8.4 recht häufig.

Die Gestalt einer Booleschen Funktion nach Satz 8.1.2 heißt **konjunktive Normalform (CNF)**. Sie ist nicht eindeutig[13,14]. Ohne weitere Erklärung sei vermerkt, dass man sie weiter zu einer „kanonischen CNF“ vereinfachen kann und damit eine gewisse Eindeutigkeit erhält. Auch eine analoge disjunktive Normalform (**DNF**) ist möglich („Disjunktion von Konjunktionen“).

[12] Insbesondere ist $n_j \leq n$ für $j = 1, \ldots, r$. Ein einzelnes Input-Bit x_i kommt in jedem der $t_{jk}(x)$ entweder direkt oder negiert oder gar nicht vor.

[13] Z. B. könnte man der Normalform von f_0 noch die Terme $\wedge\, (x_1 \vee x_2) \wedge (x_1 \vee x_3)$ hinzufügen.

[14] Die Umwandlung eines logischen Ausdrucks in die CNF wird von der Funktion `convert_cnf()` in der mitgelieferten SageMath-Klasse `sage.logic.boolformula.BooleanFormula` geleistet, die Bestimmung der zugehörigen Wahrheitstafel durch die Funktion `truthtable()` dieser Klasse.

8.1.6 Polynomiale Ausdrücke und algebraische Normalform

Betrachten wir (binäre) polynomiale Ausdrücke in den Variablen $x_1, \ldots, x_n$, wie etwa $x_1^2 x_2 + x_2 x_3 + x_3^2$, so verwenden wir als Koeffizienten, da wir im Körper $\mathbb{F}_2$ rechnen, nur die Konstanten 0 und 1, und diese brauchen in einem solchen Ausdruck gar nicht explizit hingeschrieben zu werden. Eine weitere Vereinfachung beruht auf der Beobachtung, dass $a^2 = a$ für alle Elemente $a \in \mathbb{F}_2$ (denn $0^2 = 0$ und $1^2 = 1$). Daher gilt sogar stets $a^e = a$ für alle Exponenten $e \geq 1$. Für binäre polynomiale Ausdrücke bedeutet das, dass wir die Variablen $x_1, \ldots, x_n$ nur höchstens in der ersten Potenz zu berücksichtigen brauchen. Den Beispielausdruck können wir also auch als $x_1 x_2 + x_2 x_3 + x_3$ schreiben. Ein anderes Beispiel: $x_1^3 x_2 + x_1 x_2^2 = x_1 x_2 + x_1 x_2 = 0$.

Allgemein hat ein **monomialer Ausdruck** (oder einfach nur „Monom“) die Gestalt

$$x^I := \prod_{i \in I} x_i \quad \text{mit einer Teilmenge } I \subseteq \{1, \ldots, n\},$$

d. h., er ist ein Produkt aus einigen der Variablen, wobei die Teilmenge I die Auswahl der „einigen“ angibt. Ein Beispiel mit $n = 3$ soll das illustrieren:

$$I = \{2, 3\} \Longrightarrow x^I = x_2 x_3.$$

Solche monomialen Ausdrücke gibt es also genau 2^n Stück, nämlich so viele, wie man Teilmengen aus einer n-elementigen Mengen bzw. Teilprodukte aus n potenziellen Faktoren bilden kann – die leere Menge entspricht dem Produkt aus 0 Faktoren, und das wird hier, wie auch sonst üblich, gleich 1 gesetzt[15]. Also:

$$I = \emptyset \Longrightarrow x^I = 1.$$

Einen monomialen Ausdruck kann man direkt als Boolesche Funktion interpretieren. Ob diese Funktionen alle verschieden sind, wissen wir noch nicht, werden es aber gleich sehen.

Ein polynomialer Ausdruck ist eine Summe von monomialen Ausdrücken (die Koeffizienten können hier im binären Fall ja nur 0 oder 1 sein). Der allgemeinst mögliche (binäre) polynomiale Ausdruck hat also die Gestalt

$$\sum_{I \subseteq \{1,\ldots,n\}} a_I x^I,$$

wobei die Koeffizienten a_I alle 0 oder 1 sind. D. h., es wird eine Teilmenge aller 2^n möglichen monomialen Ausdrücke aufaddiert, und dafür gibt es 2^{2^n} Möglichkeiten. Die dadurch beschriebenen Booleschen Funktionen sind alle verschieden, aber das müssen wir erst noch zeigen. Zunächst wird bewiesen, dass sich jede Boolesche Funktion so ausdrücken lässt.

Satz 8.1.3 (ANF). *Für jede Boolesche Funktion $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ gibt es Koeffizienten $a_I \in \mathbb{F}_2$ (also $= 0$ oder 1), wobei I alle Teilmengen von $\{1, \ldots, n\}$ durchläuft, so dass f sich als polynomialer Ausdruck in n Variablen so schreiben lässt:*

$$f(x_1, \ldots, x_n) = \sum_{I \subseteq \{1,\ldots,n\}} a_I x^I. \tag{8.2}$$

Beweis
(Induktion über n) Nehmen wir $n = 1$ als Induktionsanfang[16]. Die vier möglichen Booleschen

[15] während man „leere“ Summen üblicherweise gleich 0 setzt.

[16] Eine „typische mathematische Spitzfindigkeit“, aber dennoch korrekt, wäre es auch, den Trivialfall $n = 0$ als Induktionsanfang zu nehmen – die beiden konstanten polynomialen Ausdrücke 0 und 1 entsprechen den beiden konstanten Funktionen von 0 Variablen.

Funktionen von einer Variablen x sind die Konstanten 0 und 1 sowie x und $1+x$ (= die Negation von x). Sie haben alle die behauptete Form.

Sei also jetzt $n \geq 1$. Ist $x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$, so wird im Folgenden abgekürzt: $x' = (x_2, \ldots, x_n) \in \mathbb{F}_2^{n-1}$. Man kann dann auch $x = (x_1, x')$ statt $x = (x_1, \ldots, x_n)$ schreiben.

Sei nun eine Funktion $f \in \mathcal{F}_n$ gegeben. Für jeden festen Wert b an Stelle der ersten Variablen x_1, also $b = 0$ oder 1, betrachten wir die Funktion $x' \mapsto f(b, x')$ von den $n-1$ Variablen, die in x' stecken. Für diese ist aufgrund der Induktionsannahme (sowohl für $b = 0$ als auch für $b = 1$) jeweils

$$f(b, x') = p_b(x') \qquad \text{für alle } x' \in \mathbb{F}_2^{n-1}$$

mit polynomialen Ausdrücken p_0, p_1 in x' von der behaupteten Form, also

$$p_0(x') = \sum_{J \subseteq \{2,\ldots,n\}} b_J x^J, \qquad p_1(x') = \sum_{J \subseteq \{2,\ldots,n\}} c_J x^J.$$

Damit ist

$$f(x_1, x') = \begin{cases} p_0(x'), & \text{wenn } x_1 = 0, \\ p_1(x'), & \text{wenn } x_1 = 1, \end{cases} \qquad \text{für alle } x = (x_1, x') \in \mathbb{F}_2^n,$$

da x_1 ja nur 0 oder 1 sein kann. Das kann man auch als

$$f(x_1, x') = (1 + x_1)p_0(x') + x_1 p_1(x') \qquad \text{für alle } x \in \mathbb{F}_2^n, \tag{8.3}$$

schreiben, wie man sofort sieht, wenn man in (8.3) $x_1 = 0$ bzw. $x_1 = 1$ einsetzt. Durch Ausmultiplizieren und Entfernen doppelt vorkommender Monome erhält man somit wieder einen polynomialen Ausdruck in x von der behaupteten Form:

$$\begin{aligned} f(x_1, x') &= p_0(x') + x_1(p_0(x') + p_1(x')) \\ &= \underbrace{\sum_{J \subseteq \{2,\ldots,n\}} b_J x^J}_{\text{alle Monome ohne } x_1} + \underbrace{\sum_{J \subseteq \{2,\ldots,n\}} (b_J + c_J) x_1 x^J}_{\text{alle Monome mit } x_1}. \end{aligned}$$

□

Die mathematisch kompakte Formulierung dieses Satzes wird durch die zweite Spalte der Tabelle 8.4 veranschaulicht, wobei die Variablen dort x und y statt x_1 und x_2 heißen und die Koeffizienten a, b, c und d statt $a_\emptyset$, $a_{\{1\}}$, $a_{\{2\}}$ und $a_{\{1,2\}}$. Jede Zeile der Tabelle beschreibt eine Boolesche Funktion in zwei Variablen, und diese ist die Summe derjenigen der Terme 1, x, y, xy, die in der Darstellung nach Gleichung (8.2) den Koeffizienten 1 haben, während man Terme mit Koeffizienten 0 natürlich nicht hinzuschreiben braucht.

Die durch Satz 8.1.3 garantierte Darstellung einer Booleschen Funktion als polynomialer Ausdruck heißt **algebraische Normalform (ANF)**[17]. Bemerkenswert ist, dass diese sogar eindeutig ist: Da es 2^{2^n} polynomiale Ausdrücke gibt und diese alle 2^{2^n} verschiedenen Booleschen Funktionen darstellen, müssen erstens diese polynomialen Ausdrücke als Funktionen alle verschieden sein, und zweitens muss diese Darstellung einer Booleschen Funktion als polynomialer Ausdruck eindeutig sein. Damit ist gezeigt:

Satz 8.1.4. *Die Darstellung einer Booleschen Funktion in algebraischer Normalform ist eindeutig.*

[17] Die Umwandlung zwischen ANF und Wahrheitstafel wird von der (internen) Funktion `__convert()` der Klasse `BoolF()` geleistet, siehe das SageMath-Beispiel 8.42. Das in SageMath enthaltene Modul `sage.crypto.boolean_function` bietet ebenfalls die Initialisierung durch eine Wahrheitstafel oder durch ein Boolesches Polynom sowie Funktionen `algebraic_normal_form()` und `truth_table()` zur Umwandlung.

Definition 8.1.2. *Der Grad einer Booleschen Funktion $f \in \mathcal{F}_n$ als polynomialer Ausdruck in algebraischer Normalform,*

$$\deg f = \max\{\#I \mid a_I \neq 0\},$$

wird als **(algebraischer) Grad** *von f bezeichnet. Er ist stets $\leq n$.*

Der Grad gibt also an, wieviele verschiedene Variablen in einem Monom der ANF maximal miteinander multipliziert werden.

Beispiel: Es gibt (unabhängig von der Variablenzahl) genau zwei Boolesche Funktionen vom Grad 0: die beiden Booleschen Konstanten 0 und 1.

Funktionen vom Grad ≤ 1 werden auch als affine Funktionen bezeichnet; sie sind die Summe einer Konstanten und einer Booleschen Linearform, siehe dazu Abschnitt 8.1.9. Ist der Grad > 1, spricht man auch von nicht-linearen Funktionen, obwohl die Bezeichnung „nicht-affin“ korrekt wäre.

Beispiel: Die durch $x \mapsto x_1x_2 + x_2x_3 + x_3$ gegebene Boolesche Funktion hat den Grad 2.

Bemerkung: Ein hoher Grad von Booleschen Funktionen wird also nicht durch höhere Potenzen von Variablen, sondern „nur“ durch Produkte verschiedener Variablen erreicht. Jede einzelne Variable kommt in jedem Monom der ANF stets nur maximal in der ersten Potenz vor. Man sagt auch, sämtliche partiellen Grade – das sind die Grade in den einzelnen Variablen x_i ohne Berücksichtigung der anderen Variablen – seien ≤ 1.

8.1.7 Boolesche Funktionen von zwei Variablen

Die $2^4 = 16$ Booleschen Funktionen in zwei Variablen x und y sind alle in Tabelle 8.4 aufgezählt. Sie werden als polynomiale Ausdrücke in algebraischer Normalform $a + bx + cy + dxy$ sowie als logische Ausdrücke beschrieben. Die in Satz 8.1.3 verwendeten Parameter a_I sind hier $a = a_\emptyset$, $b = a_{\{1\}}$, $c = a_{\{2\}}$, $d = a_{\{1,2\}}$, die Inputvariablen $x = x_1$, $y = x_2$.

Wir haben bereits gesehen, dass sich jede Boolesche Funktion in beliebig vielen Variablen als polynomialer Ausdruck schreiben lässt. Um die Darstellbarkeit aller Booleschen Funktionen als logische Ausdrücke zu beweisen, muss man sich nur noch vergewissern, dass die algebraischen Operationen $+$ und $\cdot$ durch die logischen Operationen $\vee$, $\wedge$ und $\neg$ ausgedrückt werden können. Das liest man aus den entsprechenden Zeilen von Tabelle 8.4 ab. Damit ist auch (als schwache Form des hier unbewiesenen Satzes 8.1.2) gezeigt:

Satz 8.1.5. *Jede Boolesche Funktion lässt sich durch einen logischen Ausdruck, d. h. als Formel in den logischen Operationen $\vee$, $\wedge$ und $\neg$ darstellen.*

Hinweis. Die logische Verneinung $\neg$ entspricht in der algebraischen Interpretation der Addition von 1.

Bemerkung. Analog kann man die ANF einer Booleschen Funktion von drei Variablen x, y, z in der Gestalt

$$(x, y, z) \mapsto a + bx + cy + dz + exy + fxz + gyz + hxyz$$

a	b	c	d	ANF	logische Operation		CNF
0	0	0	0	0	False	Konstante	$x \wedge \neg x$
1	0	0	0	1	True	Konstante	$x \vee \neg x$
0	1	0	0	x	x	Projektion	x
1	1	0	0	$1 + x$	$\neg x$	Negation	$\neg x$
0	0	1	0	y	y	Projektion	y
1	0	1	0	$1 + y$	$\neg y$	Negation	$\neg y$
0	1	1	0	$x + y$	x XOR y	XOR	$(x \vee y) \wedge (\neg x \vee \neg y)$
1	1	1	0	$1 + x + y$	$x \iff y$	Äquivalenz	$(\neg x \vee y) \wedge (x \vee \neg y)$
0	0	0	1	xy	$x \wedge y$	AND	$x \wedge y$
1	0	0	1	$1 + xy$	$\neg(x \wedge y)$	NAND	$(\neg x) \vee (\neg y)$
0	1	0	1	$x + xy$	$x \wedge (\neg y)$		$x \wedge (\neg y)$
1	1	0	1	$1 + x + xy$	$x \Longrightarrow y$	Implikation	$(\neg x) \vee y$
0	0	1	1	$y + xy$	$(\neg x) \wedge y$		$(\neg x) \wedge y$
1	0	1	1	$1 + y + xy$	$x \Longleftarrow y$	Implikation	$x \vee (\neg y)$
0	1	1	1	$x + y + xy$	$x \vee y$	OR	$x \vee y$
1	1	1	1	$1 + x + y + xy$	$\neg(x \vee y)$	NOR	$(\neg x) \wedge (\neg y)$

Tabelle 8.4: Die 16 zweistelligen Bitoperationen (= Boolesche Funktionen von 2 Variablen) unter Benutzung von Tabelle 8.2 (Die Anordnung in der ersten Spalte ist lexikographisch, wenn man die Reihenfolge a, b, c, d umkehrt.)

schreiben[18]. Hier kommen also 8 Koeffizienten $a, \ldots, h$ vor. Das passt zu der Erkenntnis, dass

- eine Boolesche Funktion von drei Variablen bis zu $8 = 2^3$ Monome enthalten kann
- und es $2^{2^3} = 2^8 = 256$ solche Funktionen gibt.

Beispiel. Wie sieht die ANF der Funktion f_0 aus Abschnitt 8.1.2, hier mit den Variablen x, y, z als $f_0(x, y, z) = x \wedge (y \vee z)$ geschrieben, aus? Nach Tabelle 8.4 ist $(y \vee z) = y + z + yz$, während die AND-Verknüpfung $\wedge$ einfach nur das Produkt im Körper $\mathbb{F}_2$ ist. Daher ist

$$f_0(x, y, z) = x \cdot (y + z + yz) = xy + xz + xyz,$$

und daran sieht man, dass f_0 den Grad 3 hat.

Bemerkung. Aus der Tabelle 8.4 kann man direkt einen naiven Algorithmus zur Umwandlung von logischen Ausdrücken in (binäre) polynomiale und umgekehrt ablesen.

8.1.8 Boolesche Abbildungen

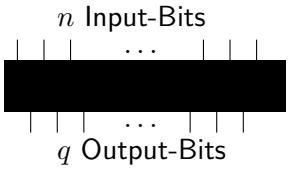
In der Kryptographie braucht man meistens Prozesse, die nicht nur ein Bit, sondern gleich mehrere produzieren. Dies wird durch das Konzept einer **Booleschen Abbildung** abstrakt

[18] In dieser Formel hat der Buchstabe f – abweichend vom sonst üblichen Gebrauch in diesem Text – die Bedeutung eines Koeffizienten, nicht einer Funktion. In der Mathematik werden Buchstaben als Symbole fast immer relativ zum Kontext und nur ganz selten in absoluter Bedeutung gebraucht. Solche Ausnahmen sind etwa die Zahlen e, i und π. Aber auch i wird oft, wenn im Kontext keine komplexen Zahlen vorkommen, anders, z. B. als Summationsindex, verwendet. Oder e als Exponent oder Koeffizient.

beschrieben, also als Abbildung[19]

$$f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$$

mit natürlichen Zahlen n und q, illustriert durch die folgende Grafik



Die Bilder von f sind also Bitblöcke der Länge q. Zerlegt man diese in ihre Komponenten,

$$f(x) = (f_1(x), \ldots, f_q(x)) \in \mathbb{F}_2^q,$$

so sieht man, dass eine Boolesche Abbildung nach $\mathbb{F}_2^q$ auch einfach als q-Tupel (oder System) von Booleschen Funktionen

$$f_1, \ldots, f_q\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$$

beschrieben werden kann.

Definition 8.1.3. *Der* **(algebraische) Grad** *einer Booleschen Abbildung $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ ist das Maximum der algebraischen Grade ihrer Komponenten,*

$$\deg f = \max\{\deg f_i \mid i = 1, \ldots, q\}.$$

Satz 8.1.6. *Jede Boolesche Abbildung $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ hat eine eindeutige Darstellung als*

$$f(x_1, \ldots, x_n) = \sum_{I \subseteq \{1,\ldots,n\}} x^I a_I$$

mit $a_I \in \mathbb{F}_2^q$ und Monomen x^I wie in Satz 8.1.3.

Diese Darstellung einer Booleschen Abbildung wird natürlich ebenfalls **algebraische Normalform** genannt. Sie entsteht aus der Zusammenfassung der algebraischen Normalformen der Komponentenfunktionen $f_1, \ldots, f_q$. Im Vergleich zu Satz 8.1.3 sind die x^I und a_I in umgekehrter Reihenfolge geschrieben, weil man per Konvention meistens die „Skalare“ (hier $x^I \in \mathbb{F}_2$) vor die „Vektoren“ (hier $a_I \in \mathbb{F}_2^q$) schreibt. Die a_I sind einfach die Zusammenfassungen der jeweiligen Koeffizienten der Komponentenfunktionen.

Beispiel

Eine Boolesche Abbildung $g\colon \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2^2$ sei durch ein Paar von logischen Ausdrücken in drei Variablen x, y, z definiert:

$$g(x, y, z) := \begin{pmatrix} x \wedge (y \vee z) \\ x \wedge z \end{pmatrix},$$

[19] Die begriffliche Unterscheidung zwischen „Funktion“ und „Abbildung“ ist etwas willkürlich, wird aber in der Mathematik oft so wie hier getroffen, um auszudrücken, ob ein Wertebereich ein- oder evtl. mehrdimensional ist. Boolesche Abbildungen werden in Form von Systemen Boolescher Funktionen oft auch als „vektorwertige Boolesche Funktionen“, englisch „Vectorial Boolean Functions“ (VBF) bezeichnet.

wobei die Komponenten übersichtlich untereinander, also in Spaltenform geschrieben sind. In der ersten Komponente erkennen wir die Funktion f_0 wieder, in der zweiten das Produkt $x \cdot z$. Die ANF von g ist also

$$g(x,y,z) = \begin{pmatrix} xy + xz + xyz \\ xz \end{pmatrix} = xy \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + xz \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} + xyz \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Der algebraische Grad ist 3, und die Wertetabelle steht in Tabelle 8.5. Die Werte $g(x,y,z) \in \mathbb{F}_2^2$ von g sind dabei als Bitketten der Länge 2 geschrieben[20].

x	y	z	$g(x,y,z)$
0	0	0	00
0	0	1	00
0	1	0	00
0	1	1	00
1	0	0	00
1	0	1	11
1	1	0	10
1	1	1	11

Tabelle 8.5: Beispiel der Wertetabelle einer Booleschen Abbildung

8.1.9 Linearformen und lineare Abbildungen

Eine Boolesche Funktion $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ heißt **Linearform**, wenn sie vom Grad 1 und ihr absolutes Glied 0 ist. Insbesondere kommen in ihrer algebraischen Normalform nur lineare Terme vor, d. h., sie hat die Gestalt

$$f(x) = \sum_{i=1}^{n} s_i x_i \qquad \text{für alle } x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$$

mit $s_i \in \mathbb{F}_2$ für $i = 1, \ldots, n$. Da alle s_i nur 0 oder 1 sein können, hat jede Linearform also die Gestalt einer Teilsumme

$$\alpha_I(x) = \sum_{i \in I} x_i \qquad \text{für alle } x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$$

über eine Teilmenge $I \subseteq \{1, \ldots, n\}$ der Menge aller Indizes, nämlich

$$I = \{i \mid s_i = 1\}.$$

Es gibt also genau 2^n Boolesche Linearformen in n Variablen, und diese entsprechen auf natürliche Weise genau der Potenzmenge $\mathfrak{P}(\{1, \ldots, n\})$.

Andere übliche Schreibweisen sind für $I = \{i_1, \ldots, i_r\}$:

$$\alpha_I(x) = x[I] = x[i_1, \ldots, i_r] = x_{i_1} + \cdots + x_{i_r}.$$

Der folgende Satz sagt, dass die Linearformen genau dem in der Linearen Algebra üblichen Begriff entsprechen:

[20] Die verschiedenen Schreibweisen von Bitblöcken, hier einmal untereinander zwischen Klammern – als Spaltenvektoren – und einmal nebeneinander als Zeichenketten der Länge 2, sollen nicht die Verwirrung befördern, sondern darauf hinweisen, dass verschiedene Schreibweisen möglich und üblich sind, die in unterschiedlichen Situationen je nach Zweckmäßigkeit gewählt werden.

Satz 8.1.7. *Eine Boolesche Funktion $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ ist genau dann Linearform, wenn folgende beiden Bedingungen erfüllt sind:*

(i) $f(x+y) = f(x) + f(y)$ *für alle* $x, y \in \mathbb{F}_2^n$.

(ii) $f(ax) = af(x)$ *für alle* $a \in \mathbb{F}_2$ *und alle* $x \in \mathbb{F}_2^n$.

Beweis
Dass jede Linearform diese beiden Bedingungen erfüllt, folgt direkt aus der Darstellung als Teilsumme.

Sei nun umgekehrt f eine Boolesche Funktion, die (i) und (ii) erfüllt. Seien $e_1 = (1, 0, \ldots, 0)$, $\ldots$, $e_n = (0, \ldots, 1)$ die „kanonischen Einheitsvektoren“. Jedes $x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ lässt sich dann als Summe

$$x = x_1 e_1 + \cdots + x_n e_n$$

schreiben. Damit ist

$$f(x) = f(x_1 e_1) + \cdots + f(x_n e_n) = x_1 f(e_1) + \cdots + x_n f(e_n)$$

die Teilsumme der x_i über die Indexmenge derjenigen i, für die der konstante Wert $f(e_i)$ nicht 0, also 1 ist. Also ist f eine Linearform im Sinne der obigen Definition. □

Eine Boolesche Abbildung $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ heißt linear, wenn alle ihre Komponentenfunktionen $f_1, \ldots, f_q$ Linearformen sind. Genau wie im Fall $q = 1$ zeigt man:

Satz 8.1.8. *Eine Boolesche Abbildung $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ ist genau dann linear, wenn folgende beiden Bedingungen erfüllt sind:*

(i) $f(x+y) = f(x) + f(y)$ *für alle* $x, y \in \mathbb{F}_2^n$.

(ii) $f(ax) = af(x)$ *für alle* $a \in \mathbb{F}_2$ *und alle* $x \in \mathbb{F}_2^n$.

Satz 8.1.9. *Eine Boolesche Abbildung $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ ist genau dann linear, wenn sie die Gestalt*

$$f(x) = \sum_{i=1}^{n} x_i s_i$$

mit $s_i \in \mathbb{F}_2^q$ hat.

(Hier sind x_i und s_i wieder in umgekehrter Reihenfolge geschrieben.)

Affine (Boolesche) Abbildungen sind solche, deren algebraischer Grad ≤ 1 ist. Das sind genau die, die sich durch Addition einer linearen Abbildung und einer Konstanten bilden lassen.

Im Fall $q = 1$, also bei Funktionen, gibt es als mögliche Konstanten nur 0 und 1. Die Addition der Konstanten 1 entspricht genau der logischen Negation, d. h. dem „Umkippen“ aller Bits. Daher kann man auch sagen: *Die affinen Booleschen Funktionen sind genau die Linearformen und deren Negationen.*

8.1.10 Boolesche lineare Gleichungssysteme

Die Algebra über dem Körper $\mathbb{F}_2$ ist so einfach, dass manche Komplikation, die man aus anderen mathematischen Fachgebieten kennt, hier in sich zusammenfällt. Das gilt auch für das Lösen linearer Gleichungssysteme. Ein solches hat die Gestalt

$$\begin{array}{ccccccc} a_{11}x_1 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array}$$

mit gegebenen a_{ij} und $b_i \in \mathbb{F}_2$ und unbekannten x_j, für die eine Lösung gefunden werden soll. In Matrix-Schreibweise drückt man das eleganter durch die Gleichung

$$Ax = b$$

aus, wobei hier x und b wieder als Spaltenvektoren, also als $(n \times 1)$- bzw. $(m \times 1)$-Matrizen gedacht werden.

Lineare Gleichungssysteme in SageMath

Um die Verbindung mit der „gewöhnlichen“ Linearen Algebra herzustellen, betrachten wir ein Beispiel über den rationalen Zahlen, das Gleichungssystem

$$\begin{array}{rcrcrcr} x_1 & + & 2x_2 & + & 3x_3 & = & 0 \\ 3x_1 & + & 2x_2 & + & x_3 & = & -4 \\ x_1 & + & x_2 & + & x_3 & = & -1 \end{array}$$

und sehen, wie man das in SageMath behandeln kann; die fertige Lösung steht im SageMath-Beispiel 8.1. Die einzelnen Schritte sind:

1. Wir definieren die „Koeffizienten-Matrix“ $A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$.
2. Wir definieren den „Bildvektor“ $b = (0, -4, 1)$.
3. Wir lassen SageMath einen „Lösungsvektor“ x ausrechnen. Da wir die linke Seite des Gleichungssystems als Matrix-Produkt Ax geschrieben haben, müssen wir die Methode `solve_right()` verwenden.
4. Das lineare Gleichungssystem könnte noch weitere Lösungen haben. Diese findet man, indem man das zugehörige „homogene“[21] Gleichungssystem $Az = 0$ löst; ist z eine Lösung davon, so ist $A \cdot (x + z) = Ax + Az = b + 0 = b$, also $x + z$ eine weitere Lösung des ursprünglichen („inhomogenen“) Gleichungssystems. Auf diese Weise erhält man alle Lösungen, denn ist $Ax = b$ und $Ay = b$, so $A \cdot (y - x) = 0$, also die Differenz $y - x$ Lösung des homogenen Systems. Die Lösung des homogenen Systems bestimmt man mit der SageMath-Methode `right_kernel()`.
5. Die etwas kryptische Ausgabe besagt, dass alle Lösungen des homogenen Systems Vielfache des Vektors $z = (1, -2, 1)$ sind[22].
6. Zur Probe prüfen wir, ob $y = x - 4z$ tatsächlich Lösung ist, d. h., ob $Ay = b$.

Lineare Gleichungssysteme im Booleschen Fall

Im allgemeinen Fall (über einem beliebigen Körper) findet man die Lösung eines linearen Gleichungssystems durch Gaußsche[23] Elimination; dieser Algorithmus steckt natürlich auch in der SageMath-Methode `solve_right()`.

[21] d. h., die rechte Seite b wird gleich 0 gesetzt
[22] Da alle Koeffizienten ganzzahlig waren, hat SageMath sogar in $\mathbb{Z}$ (= `Integer Ring`) gerechnet.
[23] Johann Carl Friedrich Gauß, deutscher Mathematiker, Astronom, Geodät und Physiker, 30.4.1777–23.2.1855

SageMath-Beispiel 8.1 Auflösung eines linearen Gleichungssystems über $\mathbb{Q}$

```
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: b = vector([0,-4,-1])
sage: x = A.solve_right(b); x
(-2, 1, 0)
sage: K = A.right_kernel(); K
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -2  1]
sage: y = x - 4*vector([1,-2,1]); y
(-6, 9, -4)
sage: A*y
(0, -4, -1)
```

Im Booleschen Fall (über dem Körper $\mathbb{F}_2$) ist die Lösung linearer Gleichungssysteme mit Gaußscher Elimination extrem einfach, da nur Koeffizienten 0 und 1 vorkommen und Multiplikationen oder Divisionen völlig entfallen; auch komplizierte Koeffizienten (wie Brüche über $\mathbb{Q}$) oder ungenaue Koeffizienten (wie Gleitkommazahlen über $\mathbb{R}$) gibt es hier nicht. Die Methode ist so einfach, dass selbst für sechs Unbekannte das Rechnen per „Papier und Bleistift“ fast noch schneller geht, als dem zugehörigen SageMath-Prograämmchen die richtigen Werte zu übergeben. Dies wird durch ein Beispiel illustriert.

Die Startidee der Elimination ist die Reduktion auf ein Gleichungssystem mit nur $n-1$ Unbekannten – eine Unbekannte wird „eliminiert“.

1. Fall: x_n kommt nur mit Koeffizienten $a_{in} = 0$ für $i = 1, \ldots, m$, d. h., de facto gar nicht vor. Dann ist das System schon reduziert.

2. Fall: x_n kommt in einer der Gleichungen mit Koeffizient 1 vor. Dann wird diese[24] nach x_n aufgelöst und der resultierende Wert für x_n in die übrigen $m-1$ Gleichungen eingesetzt, die dann höchstens noch die Unbekannten $x_1, \ldots, x_{n-1}$ enthalten:

$$x_n = a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} + b_i.$$

Das wird rekursiv fortgesetzt, bis nur noch eine Unbekannte oder eine Gleichung übrig ist. Jetzt zu dem Beispiel, das zeigt, wie einfach es geht.

Beispiel:

$$\begin{array}{rrrrrrcl} x_1 & & +x_3 & & & +x_6 & = & 1 \\ x_1 & +x_2 & & +x_4 & & +x_6 & = & 0 \\ & x_2 & +x_3 & & +x_5 & +x_6 & = & 0 \\ x_1 & & & +x_4 & +x_5 & & = & 1 \\ & x_2 & & +x_4 & +x_5 & & = & 1 \end{array}$$

Aus der ersten Gleichung folgt $x_6 = x_1 + x_3 + 1$ (unter Verwendung der Regel, dass Plus und Minus dasselbe bedeuten). Elimination ergibt als Restsystem aus den Gleichungen 2 bis 5 (da

[24] Wenn es mehrere gibt, ist es egal, welche man nimmt – im Gegensatz zur Situation über anderen Grundkörpern, wo die Suche nach einem geeigneten „Pivotelement“ ein wesentlicher Bestandteil des Verfahrens ist.

z. B. $x_1 + x_1 = 0$ usw.):

$$\begin{array}{rrrrrcl} & x_2 & +x_3 & +x_4 & & = & 1 \\ x_1 & +x_2 & & & +x_5 & = & 1 \\ x_1 & & & +x_4 & +x_5 & = & 1 \\ & x_2 & & +x_4 & +x_5 & = & 1 \end{array}$$

Wird die zweite Gleichung des Restsystems nach $x_5 = x_1 + x_2 + 1$ aufgelöst und das in die übrigen eingesetzt, so bleibt

$$\begin{array}{rrrrcl} & x_2 & +x_3 & +x_4 & = & 1 \\ & x_2 & & +x_4 & = & 0 \\ x_1 & & & +x_4 & = & 0 \end{array}$$

Die beiden letzten Gleichungen ergeben $x_4 = x_2 = x_1$ und die erste dann noch $x_3 = 1$. Damit ist die komplette Lösung

$$x_1 = x_2 = x_4 = x_6 = a \quad \text{mit } a \in \mathbb{F}_2 \text{ beliebig}, \quad x_3 = 1, \quad x_5 = 1.$$

Da a die Werte 0 und 1 annehmen kann, sind das also insgesamt genau zwei Lösungen: $(0, 0, 1, 0, 1, 0)$ und $(1, 1, 1, 1, 1, 1)$.

Das Beispiel in SageMath

Das Gleiche ist im SageMath-Beispiel 8.2 zu finden. Die SageMath-Methode `solve_right()` liefert nur eine Lösung $(0, 0, 1, 0, 1, 0)$. Um alle Lösungen zu erhalten, muss man noch die Lösung der homogenen Gleichung, bestimmen: Das sind alle Vielfachen des Vektors $v = (1, 1, 0, 1, 0, 1)$, also die beiden Vektoren $(0, 0, 0, 0, 0, 0) = 0 \cdot v$ und $(1, 1, 0, 1, 0, 1) = 1 \cdot v$. Die zweite Lösung ist dann $(0, 0, 1, 0, 1, 0) + (1, 1, 0, 1, 0, 1) = (1, 1, 1, 1, 1, 1)$.

SageMath-Beispiel 8.2 Auflösung eines Booleschen linearen Gleichungssystems

```
sage: M = MatrixSpace(GF(2), 5, 6) # GF(2) = field with two elements
sage: A = M([[1,0,1,0,0,1],[1,1,0,1,0,1],[0,1,1,0,1,1],[1,0,0,1,1,0],\
      [0,1,0,1,1,0]]); A
[1 0 1 0 0 1]
[1 1 0 1 0 1]
[0 1 1 0 1 1]
[1 0 0 1 1 0]
[0 1 0 1 1 0]
sage: b = vector(GF(2),[1,0,0,1,1])
sage: x = A.solve_right(b); x
(0, 0, 1, 0, 1, 0)
sage: K = A.right_kernel(); K
Vector space of degree 6 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1 0 1 0 1]
```

Aufwandsabschätzung

Was kann man allgemein über den Aufwand zur Lösung eines Booleschen linearen Gleichungssystems sagen? Betrachten wir m Gleichungen für n Unbekannte, also eine Koeffizientenmatrix A der Größe $m \times n$ bzw. die erweiterte Matrix (A, b) der Größe $m \times (n + 1)$.

Da es hier nur auf eine grobe Abschätzung ankommt, machen wir uns über Optimierungen des Ablaufs keine weiteren Gedanken und nehmen auch o. B. d. A. an, dass $m = n$; im Fall $m > n$ würden wir überzählige Gleichungen ignorieren[25], im Fall $m < n$ „Null-Gleichungen“ (der Art $0 \cdot x_1 + \cdots + 0 \cdot x_n = 0$) anfügen.

Der Eliminationsschritt, also die Reduktion der Problemgröße von n auf $n-1$, bedeutet genau einen Durchlauf durch alle n Zeilen der *erweiterten* Matrix:

- Zunächst wird in der Spalte n, also bei den Koeffizienten von x_n der erste Eintrag 1 gesucht. Dazu ist jeweils ein Bit-Vergleich nötig.
- Danach wird die gefundene Zeile (die mit dem ersten Eintrag 1 in Spalte n) auf alle diejenigen unter ihr folgenden Zeilen aufaddiert, die ebenfalls eine 1 in Spalte n haben. Das bedeutet jeweils wieder einen Bit-Vergleich und gegebenenfalls n Bit-Additionen – den n-ten Eintrag können wir dabei übergehen, da schon klar ist, dass dort eine 0 hinkommt.

Insgesamt haben wir dazu n Bit-Vergleiche und höchstens $n \cdot (n-1)$ Bit-Additionen auszuführen, zusammen also höchstens n^2 solcher Bit-Operationen. Bezeichnen wir die Anzahl der nötigen derartigen Operationen, die wir bis zur völligen Auflösung des Gleichungssystems brauchen, mit $N(n)$, so gilt also folgende Ungleichung:

$$N(n) \leq n^2 + N(n-1) \quad \text{für alle } n \geq 2.$$

Nun ist $N(1) = 1$: Wir prüfen den einen Koeffizienten der einen Unbekannten, ob er 0 oder 1 ist; das führt zur Entscheidung, ob die Gleichung eine eindeutige Lösung hat (Koeffizient 1), oder ob sie für keinen oder für beliebige Werte der Unbekannten erfüllt ist (Koeffizient 0, rechte Seite $b = 1$ oder 0).

Damit folgt dann $N(2) \leq 2^2 + 1$, $N(3) \leq 3^2 + 2^2 + 1$ usw. Mit vollständiger Induktion ergibt sich daraus sofort

$$N(n) \leq \sum_{i=1}^{n} i^2.$$

Der explizite Wert dieser Summe ist bekannt, und wir haben bewiesen:

Satz 8.1.10. *Die Anzahl $N(n)$ der nötigen Bit-Vergleiche und Bit-Additionen zur Lösung eines Booleschen linearen Gleichungssystems aus n Gleichungen mit n Unbekannten wird abgeschätzt durch*

$$N(n) \leq \frac{1}{6} \cdot n \cdot (n+1) \cdot (2n+1).$$

Etwas vergröbernd sagt man dazu, der Aufwand sei $O(n^3)$. Auf jeden Fall ist er „polynomial von kleinem Grad“ in Abhängigkeit von der Größe n des Problems.

Bemerkung: Die O-Notation verschleiert den Unterschied zum Aufwand über anderen Körpern, der ebenfalls mit $O(n^3)$ abgeschätzt wird. Die „gefühlt“ wesentlich höhere Effizienz im Booleschen Fall begründet sich zum ersten durch die genaue Abschätzung im Satz 8.1.10, die selbst im schlechtesten Fall nur unwesentlich größer als $\frac{1}{3} \cdot n^3$ ist. Zum zweiten handelt es sich dabei um Bit-Operationen, nicht etwa um deutlich kompliziertere arithmetische oder gar Gleitkomma-Operationen.

[25] Man muss dann allerdings nachprüfen, ob die gefundenen Lösungen auch die überzähligen Gleichungen erfüllen.

8.1.11 Die Repräsentation Boolescher Funktionen und Abbildungen

Verschiedene Interpretationen von Bitblöcken

Ein Bitblock $b = (b_1, \ldots, b_n) \in \mathbb{F}_2^n$ hat uns bis jetzt zur Beschreibung ganz unterschiedlicher Objekte gedient. Er beschreibt:

- einen Bitblock (oder Vektor) $b \in \mathbb{F}_2^n$ (also sich selbst, in Zeilen- oder Spaltenform geschrieben),
- ein Argument einer Booleschen Funktion oder Abbildung, z. B. als Zeilenschlüssel in einer Wertetabelle (Wahrheitstafel),
- eine Bitkette (Bitstring) der Länge n,
- eine Teilmenge $I \subseteq \{1, \ldots, n\}$, die definiert ist durch b als Indikator: $i \in I \Leftrightarrow b_i = 1$,
- eine Linearform auf $\mathbb{F}_2^n$, ausgedrückt als Summe der Variablen x_i, für die $b_i = 1$ ist,
- ein Monom in n Variablen $x_1, \ldots, x_n$ mit allen partiellen Graden ≤ 1; hier gibt b_i den Exponenten 0 oder 1 der Variablen x_i an,
- eine ganze Zahl zwischen 0 und $2^n - 1$ in Binärdarstellung (also im Dual- oder Zweiersystem); die Folge der binären „Ziffern“ (= Bits) stimmt genau mit der entsprechenden Bitkette überein[26]. Umgekehrt entspricht die Zahl dem Index (beginnend ab 0) der Bitkette, wenn diese aufsteigend alphabetisch angeordnet werden.

Natürlich sind auch noch andere Interpretationen möglich – schließlich lässt sich ja jede Information binär codieren. Die Bitblöcke im Beispiel $n = 3$ sind in Tabelle 8.6 aufgelistet. Einige Umwandlungsroutinen stehen im SageMath-Beispiel 8.4.3.

Zahl	Bitkette	Teilmenge	Linearform	Monom
0	000	$\emptyset$	0	1
1	001	$\{3\}$	x_3	x_3
2	010	$\{2\}$	x_2	x_2
3	011	$\{2,3\}$	$x_2 + x_3$	x_2x_3
4	100	$\{1\}$	x_1	x_1
5	101	$\{1,3\}$	$x_1 + x_3$	x_1x_3
6	110	$\{1,2\}$	$x_1 + x_2$	x_1x_2
7	111	$\{1,2,3\}$	$x_1 + x_2 + x_3$	$x_1x_2x_3$

Tabelle 8.6: Interpretationen der Bitblöcke der Länge 3

Repräsentation der Wahrheitstafel einer Booleschen Funktion

Wie im vorigen Unterabschnitt beschrieben und in Tabelle 8.6 für das Beispiel der Blocklänge $n = 3$ exemplarisch dargestellt, können wir alle Bitblöcke $x = (x_1, \ldots, x_n)$ der Länge n als binär dargestellte ganze Zahlen $i(x) = 0, 1, \ldots, 2^n - 1$ interpretieren. Das Beispiel in Tabelle 8.7 legt

[26] Die SageMath-Methode `binary()` wandelt eine Zahl in eine Bitkette um, wobei führende Nullen unterdrückt werden. Beispiel: `10.binary()` ergibt `'1010'`.

x_1	x_2	x_3	$i(x)$	$f_0(x_1, x_2, x_3)$
0	0	0	0	0
0	0	1	1	0
0	1	0	2	0
0	1	1	3	0
1	0	0	4	0
1	0	1	5	1
1	1	0	6	1
1	1	1	7	1

Tabelle 8.7: Erweiterte Wahrheitstafel [für $f_0(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$] mit $n = 3$ und $2^n = 8$

dann nahe, wie man die Wahrheitstafel einer Booleschen Funktion $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ sehr sparsam durch einen Bitblock $b = (b_0, \ldots, b_{2^n-1})$ der Länge 2^n beschreiben kann: Man liest dazu einfach die letzte Spalte in der Reihenfolge der Indizes $i(x)$ ab. Die allgemeine Prozedur bei beliebigem n läuft dann so ab:

$$b_{i(x)} = f(x), \quad \text{wobei } i(x) = x_1 \cdot 2^{n-1} + \cdots + x_{n-1} \cdot 2 + x_n$$
$$\text{für} \quad x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n.$$

Das sieht vielleicht kompliziert aus, bedeutet aber einfach: „Deute x als Binärdarstellung einer ganzen Zahl $i(x)$ und suche aus dem Bitblock b das Bit heraus, das an der Stelle $i(x)$ steht“. Eine zusätzliche Spalte für $i(x)$ in der Wahrheitstafel der Funktion f_0 (f_0 wurde definiert in Gleichung 8.1) macht das beispielhaft deutlich – siehe Tabelle 8.7. Die letzte Spalte dieser Tabelle, zeilenweise geschrieben, ist dann der Bitblock b.

Die Wahrheitstafel von f_0 kann also einfach durch den Bitblock $(0, 0, 0, 0, 0, 1, 1, 1)$ oder noch sparsamer durch die Bitkette

$$00000111$$

der Länge $2^3 = 8$ vollständig beschrieben werden.[27]

Repräsentation der algebraischen Normalform

Die algebraische Normalform (ANF) wird ebenfalls durch 2^n Bits beschrieben, nämlich durch die Koeffizienten der 2^n verschiedenen Monome[28] (siehe Satz 8.1.3, Seite 274). Auch diese Monome kamen als Interpretation von Bitblöcken in der obigen Liste vor. Daher können wir einen Bitblock $a = (a_0, \ldots, a_{2^n-1})$ als Repräsentation der ANF einer Booleschen Funktion $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ ansehen:

$$f(x) = \sum_{i=0}^{2^n-1} a_i x_1^{e_1(i)} \cdots x_n^{e_n(i)}, \quad \text{wobei } i = e_1(i) \cdot 2^{n-1} + \cdots + e_n(i)$$
$$\text{mit} \quad e_1(i), \ldots, e_n(i) = 0 \text{ oder } 1.$$

Verbal beschrieben heißt das: „Interpretiere das n-Tupel e der Exponenten eines Monoms als Binärdarstellung einer ganzen Zahl i. An der Stelle i steht im Bitblock a, ob das Monom in der ANF von f vorkommt oder nicht.“

[27] Bitblöcke werden in Python/SageMath als Liste implementiert.

[28] Die ANF ist eine Summe von Monomen. Jedes Monom ist Produkt einer Teilmenge von $\{x_1, \ldots, x_n\}$ und kann also, wie eben gesehen, durch eine ganze Zahl zwischen 0 und $2^n - 1$ repräsentiert werden.

Für das Beispiel f_0 haben wir schon gesehen (oder prüfen durch Einsetzen leicht nach[29]), dass die ANF durch

$$f_0(x) = x_1x_3 + x_1x_2 + x_1x_2x_3$$

gegeben ist. Es kommen also genau die Monome mit den Exponenten-Tripeln $101, 110, 111$ vor, entsprechend den ganzen Zahlen $5, 6, 7$. Setzt man also die Bits an den Stellen $5, 6, 7$ auf 1 und die übrigen auf 0, erhält man die sparsame Darstellung der ANF durch eine Bitkette:

$$00000111.$$

Achtung: Dass das die gleiche Bitkette wie für die Wahrheitstafel ist, ist *Zufall* – eine spezielle Eigenschaft der Funktion f_0! Die Funktion $f(x_1, x_2) = x_1$ hat die Wahrheitstafel 0011 (sie hat genau dann den Wert 1, wenn $x_1 = 1$ bzw. $x = (x_1, \text{beliebig})$) und die ANF 0010 (weil genau das Monom x_1 mit Koeffizient 1 auftritt).

Für die Bestimmung der ANF im Allgemeinen kann man die SageMath-Klasse `BoolF()` verwenden, die im folgenden Unterabschnitt und im Anhang 8.4 beschrieben wird[30], zusammen mit einer Anleitung zur Nutzung. Die Anwendung auf f_0 wird im SageMath-Beispiel 8.3 demonstriert.

SageMath-Beispiel 8.3 Boolesche Funktion mit Wahrheitstafel und ANF

```
sage: bits = "00000111"
sage: x = str2bbl(bits); x
[0, 0, 0, 0, 0, 1, 1, 1]
sage: f = BoolF(x)
sage: y = f.getTT(); y
[0, 0, 0, 0, 0, 1, 1, 1]
sage: z = f.getANF(); z
[0, 0, 0, 0, 0, 1, 1, 1]
```

Bemerkung: Die naive Auswertung einer Booleschen Funktion f an allen Stellen $x \in \mathbb{F}_2^n$ bedeutet 2^n Auswertungen $f(x)$ mit je maximal 2^n Summanden à maximal $n-1$ Multiplikationen. Der Aufwand liegt also in der Größenordnung $n \cdot 2^n \cdot 2^n$. Fairerweise muss man den Aufwand aber auf die Größe des Inputs beziehen, die hier $N = 2^n$ ist. So gesehen ist der Aufwand im wesentlichen quadratisch: $N^2 \cdot \log_2(N)$. Wie so oft führt auch hier eine binäre Rekursion, also eine Aufteilung in zwei Teilprobleme von halber Inputgröße zu einem wesentlich effizienteren Algorithmus. Hierzu startet man mit der Gleichung (8.3) und benötigt im Endeffekt nur noch den fast linearen Aufwand $3N \cdot \log_2 N$. Dieser Algorithmus[31] ist in der Klasse `BoolF()` implementiert, siehe Abschnitt 8.4.6.

Objektorientierte Implementation

Für eine Implementation von Booleschen Funktionen in SageMath (bzw. Python) siehe den Anhang 8.4.6 (Klasse `BoolF()`). SageMath selbst bringt schon eine Klasse `sa-`

[29] Nicht vergessen: $f(1,1,1) = 1+1+1 = 1$, da $\bmod 2$ gerechnet wird.

[30] Diese Transformation, die eine Bitkette der Länge 2^n – die Wahrheitstafel – in eine andere Bitkette der Länge 2^n – die Liste der Koeffizienten der ANF – umwandelt, wird gelegentlich als Reed-Muller-Transformation oder als binäre Möbius-Transformation bezeichnet.

[31] auch als schnelle binäre Möbius-Transformation bezeichnet

`ge.crypto.boolean_function` mit, die viele der benötigten Methoden, auch die Umwandlung von Wahrheitstafel in ANF, enthält. Die folgende Implementation ist davon unabhängig.

Allgemein kann man in einer objektorientierten Programmiersprache eine Klasse definieren, die die Struktur eines Objekts „Boolesche Funktion“ abstrahiert:

Klasse `BoolF`:

Attribute:

- `blist`: Wahrheitstafel als Liste der Bits (= Bitblock in der „natürlichen“ Reihenfolge wie in Abschnitt 8.1.4 beschrieben); diese wird als interne Repräsentation der Booleschen Funktion verwendet.
- `dim`: die Dimension des Urbildraums

Methoden:

- `setTT`: Besetzung der Wahrheitstafel mit einem Bitblock („TT“ für Truth Table = Wahrheitstafel)
- `setANF`: Eingabe der ANF und interne Umwandlung in eine Wahrheitstafel
- `setDim`: Eingabe der Dimension des Urbildraums
- `getTT`: Ausgabe der Wahrheitstafel als Bitblock
- `valueAt`: Wert der Booleschen Funktion für ein gegebenes Argument
- `getDim`: Ausgabe der Dimension des Urbildraums
- `getANF`: Ausgabe der algebraischen Normalform (ANF) als Bitblock (in der „natürlichen“ Reihenfolge wie oben beschrieben)
- `deg`: Ausgabe des algebraischen Grades

Die ersten drei davon, die „`set`-Methoden“, werden nur implizit bei der Initialisierung benötigt. Für eine leicht „menschenlesbare“ Ausgabe fügen wir noch die Methoden `printTT` und `printANF` hinzu.

Die nötigen Funktionen zur Umwandlung von Bitlisten in Zahlen oder Bitketten und umgekehrt stehen im Anhang 8.4.3.

Die Implementation Boolescher Abbildungen leitet sich daraus ab: Man definiert eine Klasse `BoolMap` als Liste von Objekten der Klasse `BoolF` mit (mindestens) den analogen Methoden. Einige davon sind auch im SageMath-eigenen Modul `sage.crypto.mq.sbox`[32] zu finden.

[32] In der Kryptographie werden Boolesche Abbildungen bei kleiner Dimension oft „S-Boxen“ genannt.

8.2 Bitblock-Chiffren

In der klassischen Kryptographie wurde die Schwäche der einfachen monoalphabetischen Substitution auf zwei Arten behoben: einmal durch polygraphische Substitutionen, bei denen Gruppen von Buchstaben gemeinsam verschlüsselt werden, zum zweiten durch polyalphabetische Substitutionen, bei denen sich das Substitutionsalphabet mit der Position im Text ändert.

Geht man von Buchstaben zu Bits über, so sind monoalphabetische Substitutionen kryptographisch unbrauchbar, da es nur zwei Möglichkeiten gibt: entweder alle Bits unverändert lassen oder alle Bits invertieren. Das ändert den Klartext gar nicht oder nur unwesentlich. Aber die beiden Prinzipien zur Verstärkung der monoalphabetischen Substitution führen zu zwei Klassen von brauchbaren Verschlüsselungsverfahren für Informationen, die als Ketten von Bits repräsentiert werden:

- Bei Bitblock-Verschlüsselung werden die Bitketten in Blöcke fester Länge eingeteilt, die jeweils als Ganzes substituiert werden.
- Bei Bitstrom-Verschlüsselung wird der Reihe nach jedes Bit nach einer anderen Vorschrift verschlüsselt (d. h., entweder ungeändert gelassen oder negiert).

Ein mathematisch vollständiger Sicherheitsbeweis für Bitblock- oder Bitstrom-Chiffren existiert genausowenig wie für asymmetrische Verfahren. Im Gegenteil kommt dort die Reduktion auf die Schwierigkeit gut untersuchter mathematischer Probleme einem glaubhaften Sicherheitsbeweis sogar wesentlich näher. Ein symmetrisches Bitblock-Verschlüsselungsverfahren wird als *nach menschlichem Ermessen* sicher angesehen, wenn keine der bekannten Angriffsmethoden wesentlich effizienter ist als die vollständige Exhaustion des Schlüsselraums („Brute-Force-Attacke“)[33].

8.2.1 Allgemeine Beschreibung

Bitblock-Chiffren transformieren Blöcke fester Länge n längentreu in Bitblöcke gleicher Länge in Abhängigkeit von einem Schlüssel, der seinerseits als Bitblock einer bestimmten Länge l gegeben ist[34].

Eine solche Chiffre wird also beschrieben durch eine Boolesche Abbildung

$$F\colon \mathbb{F}_2^n \times \mathbb{F}_2^l \longrightarrow \mathbb{F}_2^n$$

bzw. als Familie $(F_k)_{k\in K}$ von Booleschen Abbildungen

$$F_k\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n \quad \text{für alle } k \in K = \mathbb{F}_2^l$$

mit $F_k(a) = F(a, k)$.

Wahl der Schlüssellänge

Für die Schlüssellänge l gibt es ein sehr einleuchtendes Kriterium: Sie soll so groß sein, dass eine Exhaustion des Schlüsselraums, also eine „Brute-Force-Attacke“, aussichtslos ist. Da der Schlüsselraum die Menge $\mathbb{F}_2^l$ bildet, gibt es 2^l verschiedene Schlüssel. Die Wahrscheinlichkeit,

[33] Dabei würde man einen Faktor von unter 10 noch nicht als „wesentlich effizienter“ ansehen.

[34] Die Fortsetzung auf Bitketten beliebiger Länge ist Thema des Abschnitts 8.2.4 und kümmert uns vorläufig nicht, ebensowenig wie die Frage, wie man zu kurze Blöcke auffüllt („Padding“).

einen bestimmten Schlüssel zu wählen, sollte für alle Schlüssel gleich, also $= 1/2^l$ sein. D. h. wir nehmen eine Schlüsselauswahl nach reinem Zufall an.

Unter diesen Voraussetzungen gilt eine Schlüssellänge von etwa 80 Bits heute als gerade noch sicher [LV00]. Mit der meistens gewählten Schlüssellänge von 128 Bits ist man auf der sicheren Seite. Das überholte, lange Jahre als Standard geltende DES-Verfahren hat nur einen 56-Bit-Schlüssel und ist daher heute durch Exhaustion recht schnell zu brechen.

Wahl der Blocklänge

Die Blocklänge n soll groß genug sein, um Muster- und Häufigkeitsanalysen unmöglich zu machen; noch besser ist es, jede Art von Informationspreisgabe über den Klartext, z. B. jede Wiederholung, im Geheimtext zu vermeiden.

Falls die Gegnerin ca. $2^{n/2}$ Geheimtexte von zufälligen Klartextblöcken zum gleichen Schlüssel beobachten kann, ist die Wahrscheinlichkeit einer „Kollision“[35] schon etwa $\frac{1}{2}$. Daher sollte diese Zahl $2^{n/2}$ die Zahl der verfügbaren Speicherplätze überschreiten; und auch Schlüssel sollten oft genug gewechselt werden – deutlich vor dieser Anzahl verschlüsselter Blöcke.

Die bisher meist verwendete Blocklänge 64 ist so gesehen also schon bedenklich; sie ist allenfalls noch bei häufigem Schlüsselwechsel zu rechtfertigen, und auch nur, wenn der Klartext nicht überdurchschnittlich viele Wiederholungen enthält[36]. Besser ist eine Blocklänge von 128 Bit, wie sie auch im neuen Standard AES vorgesehen ist.

Die Überlegungen zur Schlüssel- und Blocklänge sind typische Beispiele für die Sicherheitsabwägungen in der modernen Kryptographie: Es wird mit breiten Sicherheitsabständen gearbeitet; erkennbare Schwächen werden vermieden, selbst wenn sie noch weit von einer praktischen Auswertbarkeit für die Gegnerin entfernt sind. Da es aber tatsächlich gute und schnelle Verschlüsselungsverfahren gibt, die diese Sicherheitsabstände einhalten, besteht überhaupt keine Notwendigkeit, weniger starke („übertrieben strenge“) Verfahren einzusetzen.

8.2.2 Algebraische Kryptoanalyse

Der Angriff mit bekanntem Klartext[37]

Sei eine Bitblock-Chiffre durch eine Boolesche Abbildung

$$F\colon \mathbb{F}_2^n \times \mathbb{F}_2^l \longrightarrow \mathbb{F}_2^n$$

beschrieben. Dann ist F nach Satz 8.1.6 ein n-Tupel $F = (F_1, \dots F_n)$ von polynomialen Ausdrücken in $n + l$ Variablen, deren sämtliche partiellen Grade ≤ 1 sind.

Ein Angriff mit bekanntem Klartext $a \in \mathbb{F}_2^n$ und zugehörigem Geheimtext $c \in \mathbb{F}_2^n$ ergibt ein Gleichungssystem

$$F(a, x) = c$$

von n Polynomgleichungen für den unbekannten Schlüssel $x \in \mathbb{F}_2^l$.

Solche Gleichungssysteme (über beliebigen Körpern) sind Gegenstand der Algebraischen Geometrie. Die allgemeine Theorie hierzu ist hochkompliziert, insbesondere, wenn man konkrete

[35] nach dem „Geburtstagsphänomen“

[36] Durch die „Betriebsarten“ werden diese vermieden, siehe Abschnitt 8.2.4.

[37] Bei einem Angriff mit bekanntem Klartext nimmt man an, dass die Angreiferin ein kleines Stück Klartext kennt oder vermutet, und dann daraus den Schlüssel oder weiteren, ihr bisher unbekannten Klartext ermitteln will. In diesem Abschnitt nehmen wir an, dass der bekannte Klartext ein ganzer Bitblock ist.

Lösungsverfahren haben will. Aber vielleicht hilft die Beobachtung, dass man nur partielle Grade ≤ 1 benötigt?

Beispiel 1, Linearität: Ist F eine *lineare* Abbildung, so ist das Gleichungssystem mit den Methoden der Linearen Algebra effizient lösbar, siehe 8.1.10 (n lineare Gleichungen in l Unbekannten – ist $l < n$, braucht man natürlich mehrere bekannte Klartextblöcke, oder man führt eine Exhaustion über die verbliebenen $n-l$ Schlüsselbits durch). Es reicht dazu schon, wenn F linear in x ist.

Beispiel 2, nichtlinear in x: Sei $n = l = 2$,

$$F(a_1, a_2, x_1, x_2) = (a_1 + a_2x_1, a_2 + a_1x_2 + x_1x_2),$$

$a = (0, 1)$, $c = (1, 1) \in \mathbb{F}_2^2$. Dann sieht das Gleichungssystem für den Schlüssel $(x_1, x_2) \in \mathbb{F}_2^2$ so aus:

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 + x_1 \\ 1 + 0 + x_1x_2 \end{pmatrix},$$

die Lösung ist offensichtlich $x_1 = 1$, $x_2 = 0$.

Beispiel 3, Substitution: Dass man Polynomgleichungen nicht immer auf den ersten Blick ihre Komplexität ansieht, zeigt das Beispiel (über $\mathbb{F}_2$)

$$x_1x_2x_3 + x_1x_2 + x_1x_3 + x_2x_3 + x_2 + x_3 = 0.$$

Es geht durch die Substitutionen $x_i = z_i + 1$ über in

$$z_1z_2z_3 + z_1 = 0$$

(umgekehrt sieht man das leichter) mit der Lösungsmenge

$$z_1 = 0,\ z_2, z_3 \text{ beliebig } \textit{oder } z_1 = z_2 = z_3 = 1.$$

Die vollständige Lösung der ursprünglichen Gleichung ist also

$$x_1 = 1,\ x_2, x_3 \text{ beliebig } \textit{oder } x_1 = x_2 = x_3 = 0.$$

Als allgemeine Lösungsmethoden für Gleichungssysteme über $\mathbb{F}_2$ stehen zur Verfügung

- SAT-Solver [GJ79][38],
- Elimination, am effizientesten mit Hilfe von Gröbner-Basen [Bri10][39].

[38] Mit SAT wird das Erfüllbarkeitsproblem der Aussagenlogik bezeichnet. Man betrachtet einen logischen Ausdruck in Booleschen Variablen $x_1, \ldots, x_n$ und fragt, ob es Werte für die Variablen gibt, die den Ausdruck „wahr“ machen. Anders ausgedrückt betrachtet man eine Boolesche Funktion f und fragt, ob sie den Wert 1 annimmt. Ein **SAT-Solver** ist ein Algorithmus, der einen solchen logischen Ausdruck in CNF nimmt und die Erfüllbarkeit entscheidet, indem er eine Lösung für x findet oder eben nicht. Das naive Verfahren ist die Aufstellung der Wahrheitstafel für alle möglichen 2^n Argumente. Es gibt allerdings deutlich effizientere Verfahren, die gängigsten sind der DPLL-Algorithmus (nach Davis, Putnam, Logemann und Loveland) und BDD-basierte Verfahren (Binary Decision Diagram). Einiges davon findet man in den SageMath-Modulen `sage.sat.solvers` und `sage.sat.boolean_polynomials`.

[39] Zum Einstieg in dieses Gebiet eignen sich die Lehrbücher [Bar09, CLO07, vzGG99] sowie das Skript [Seg04] und der Artikel [Laz83].

Beide Methoden funktionieren gut, wenn die Zahl der Unbekannten klein ist. Sie stoßen mit wachsender Zahl von Unbekannten aber bald an Komplexitätsgrenzen[40]. Natürlich lassen sich die Lösungen immer durch Aufstellen der kompletten Wertetabelle finden, aber das ist sehr ineffizient (exponenziell in der Zahl der Unbekannten, und damit spätestens für 80 Unbekannte hoffnungslos). Aber auch der Aufwand von SAT-Solvern und Gröbner-Basis-Methoden ist immer noch im wesentlichen exponenziell in der Zahl der Unbekannten. Selbst die Tatsache, dass man nur partielle Grade ≤ 1 berücksichtigen muss, hilft da nicht weiter.

Die Komplexität des algebraischen Angriffs

Die theoretische Analyse des Aufwands für das Finden einer Lösung führt auf einen der zentralen Begriffe der Komplexitätstheorie, die **NP**-Vollständigkeit.

Satz 8.2.1 (Garey/Johnson)**.** *Das Problem, eine Lösung eines Systems von Polynomgleichungen über $\mathbb{F}_2$ zu finden, ist* **NP***-vollständig.*

Für den Beweis siehe das Buch von Garey/Johnson [GJ79].[41]

Der Begriff „**NP**-vollständig“ wird hier nicht erklärt. Er bedeutet nach der bisher unbewiesenen „$\mathbf{P} \neq \mathbf{NP}$“-Vermutung, dass ein Problem algorithmisch nicht effizient lösbar ist, also dass kein Lösungsalgorithmus bekannt ist, dessen Zeitbedarf höchstens polynomial mit der Anzahl n der Input-Variablen wächst.

Diesen Satz deutet man gerne so: Bei günstig gewählter Blockverschlüsselungsabbildung $F\colon \mathbb{F}_2^n \times \mathbb{F}_2^l \longrightarrow \mathbb{F}_2^n$ ist ein Angriff mit bekanntem Klartext (auf den Schlüssel $k \in \mathbb{F}_2^l$) nicht effizient durchführbar. Mathematisch streng genommen besagt der Satz für die praktische Anwendung aber *gar nichts*:

1. Er bezieht sich nur auf den Fall eines Algorithmus für *beliebige* Polynomgleichungen (über $\mathbb{F}_2$). Er macht keine Aussage für spezielle Klassen von Polynomen oder gar für ein bestimmtes Polynomsystem.

2. Er ist ein reiner (Nicht-) Existenzbeweis und liefert kein konkretes Beispiel eines „schwierigen“ Polynomsystems. Einzelne konkrete Systeme sind ja durchaus leicht zu lösen.

3. Und selbst wenn man konkrete Beispiele für „schwierige“ Systeme kennte, würde der Satz doch nichts darüber sagen, ob nur einzelne, wenige Instanzen schwierig sind oder – was der Kryptologe eigentlich braucht – fast alle. Es könnte ja immer noch einen Algorithmus geben, der ein Polynomsystem für fast alle Tupel von Unbekannten effizient löst und nur an wenigen Tupeln scheitert.

Dennoch beruht auf diesem Satz die Hoffnung, dass es „sichere“ Bitblock-Verfahren gibt, und die Konstruktion von Bitblock-Chiffren folgt der

Faustregel: *Lineare Gleichungssysteme für Bits sind sehr effizient lösbar, nichtlineare Gleichungssysteme für Bits sind dagegen in so gut wie allen Fällen nicht effizient lösbar.*

[40] In der Tat ist SAT das historisch erste Problem, für das die **NP**-Vollständigkeit gezeigt wurde.
[41] Ein neuerer Artikel über die Schwierigkeit, Systeme von Polynom-Gleichungen zu lösen ist [CGH$^+$03].

8.2.3 Aufbau von Bitblock-Chiffren

Mangels einer konkreten, allgemein anwendbaren Sicherheitsaussage hat sich für die Konstruktion von Bitblock-Chiffren ein Aufbau eingebürgert, der zwar nicht zwingend ist, aber in der Praxis z. Z. der beste bekannte Weg ist, um plausible Sicherheit zu erreichen. Auch die anerkannten Standard-Verschlüsselungsverfahren DES und AES wurden so konstruiert.

Ideal wäre, wenn man Sicherheitseigenschaften von Booleschen Abbildungen

$$F\colon \mathbb{F}_2^n \times \mathbb{F}_2^l \longrightarrow \mathbb{F}_2^n$$

für realistische Werte der Blocklänge n und der Schlüssellänge l leicht messen könnte, etwa für n und l in der Größenordnung von 128 oder mehr.

Nun gibt es tatsächlich solche Maße für die Sicherheit – wie das lineare Potenzial und das differenzielle Potenzial, die die Abweichung von der Linearität beschreiben; ferner die algebraische Immunität und andere. Diese sind allerdings nur für kleine Blocklängen n, etwa in der Größenordnung bis 8, effizient bestimmbar und liefern nur notwendige, nicht aber hinreichende Bedingungen für die Sicherheit.

Daher beginnt man die Konstruktion mit Booleschen Abbildungen kleiner Blocklänge und baut diese schrittweise bis zu den gewünschten Blocklängen aus. Die Schritte dahin sind:

1. Definition von einer oder mehreren Booleschen Abbildungen in kleiner Dimension q (= Blocklänge des Definitionsbereichs), etwa 4, 6 oder 8, die bezüglich aller bekannten und messbaren Sicherheitseigenschaften hinreichend gut sind. Diese werden **S-Boxen** genannt und bilden die elementaren Bausteine des Verschlüsselungsverfahrens. („S“ steht für Substitution.)

2. Nach „Einmischen“ einiger Schlüsselbits wird durch parallele Anwendung der S-Boxen (oft wird auch dieselbe S-Box parallel angewendet) eine Abbildung auf der gewünschten Input-Breite erzeugt.

3. Dann wird der gesamte Bitblock auf voller Breite permutiert.

4. Diese Schritte zusammen bilden eine **„Runde“** des gesamten Schemas. Die Schwächen dieser Runde, die hauptsächlich aus der zu niedrigen Dimension der S-Boxen resultieren, werden analysiert. Durch Iteration über mehrere gleichartige Runden – mit wechselnder Auswahl der jeweiligen Schlüsselbits – werden diese Schwächen in (halbwegs) kontrollierter Weise sukzessive abgebaut.

5. Hat man die gewünschte Sicherheit erreicht, legt man als Sicherheitsreserve noch ein paar Runden drauf.

In Abbildung 8.2 ist das Schema für eine Runde skizziert.

Das gesamte Schema ist eine Version eines etwas allgemeineren Ansatzes, der auf Shannon zurückgeht. Nach Shannon sollen Blockchiffren folgendes leisten:

Diffusion (Durchmischung): Die Bits des Klartextblocks werden über den gesamten Block „verschmiert“. Grundbausteine zur Erreichung von Diffusion sind Permutationen (Transpositionen).

Konfusion (Komplexität des Zusammenhangs): Die Beziehung zwischen Klartextblock und Schüssel einerseits sowie Geheimtextblock andererseits soll möglichst kompliziert sein (insbesondere hochgradig nichtlinear). Grundbausteine hierfür sind vor allem Substitutionen.

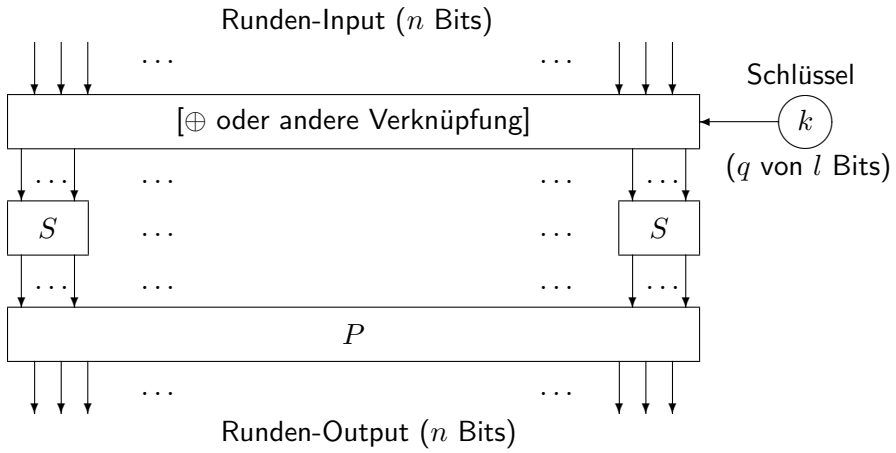


Abbildung 8.2: Eine Runde eines Bitblock-Verfahrens (S ist je eine, evtl. unterschiedliche, S-Box, P eine Permutation, k der Schlüssel)

Beides zusammen soll insbesondere bewirken, dass sich bei Änderung eines Schlüsselbits möglichst viele Geheimtextbits ändern, und zwar möglichst unvorhersagbar.

> *Es soll für die Angreiferin unmöglich sein zu erkennen, dass sie einen Schlüssel „fast" richtig geraten hat.*

Shannon schlug daher als Konstruktionsprinzip für starke Blockchiffren vor, diese aus einer wechselnden Folge von **S**ubstitutionen und **P**ermutationen zu bilden – sogenannte **SP-Netze**. Das sieht so aus:

$$\begin{aligned}\mathbb{F}_2^n \overset{S_1(\bullet,k)}{\longrightarrow} \mathbb{F}_2^n \overset{P_1(\bullet,k)}{\longrightarrow} \mathbb{F}_2^n \longrightarrow \ldots \\ \ldots \longrightarrow \mathbb{F}_2^n \overset{S_r(\bullet,k)}{\longrightarrow} \mathbb{F}_2^n \overset{P_r(\bullet,k)}{\longrightarrow} \mathbb{F}_2^n\end{aligned}$$

abhängig von einem Schlüssel $k \in \mathbb{F}_2^l$. Dabei ist

$$\begin{aligned}S_i &= i\text{-te Substitution},\\ P_i &= i\text{-te Permutation},\\ P_i \circ S_i &= i\text{-te } \textbf{Runde},\end{aligned}$$

wobei insgesamt r Runden nacheinander ausgeführt werden.

Die Permutationen sind spezielle lineare Abbildungen $P\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$. In neueren Bitblock-Chiffren wie AES werden sie oft durch allgemeinere lineare Abbildungen ersetzt, die eine noch bessere Diffusion bewirken. Der pasende Begriff **„LP-Netz"** hat sich bisher aber nicht eingebürgert.

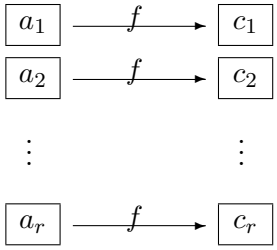
8.2.4 Betriebsarten

Eine Blockverschlüsselungsfunktion $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$ soll auch auf längere (oder kürzere) Bitfolgen angewendet werden können. (Die Abhängigkeit vom Schlüssel spielt in diesem Abschnitt keine Rolle und wird daher in der Notation weggelassen.) Das erfordert zwei Maßnahmen:

1. Die Bitfolge a wird in n-Bit-Blöcke $a_1, \ldots, a_r$ aufgespalten.

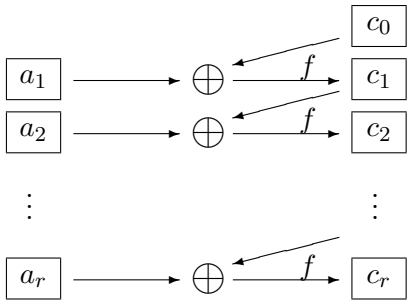
2. Der letzte Block a_r wird bei Bedarf auf die Länge n aufgefüllt („padding“) mit
   - Nullen oder
   - Zufallswerten oder
   - Strukturinformationen.

Das nächstliegende Verfahren ist dann, jeden Block der Reihe nach einzeln zu verschlüsseln. Das nennt man ECB-Modus (für „Electronic Code Book“) oder -Betriebsart. Das Verfahren sieht also so aus:



Man kann das als monoalphabetische Substitution interpretieren, wenn man die Bitblöcke aus $\mathbb{F}_2^n$ als „Buchstaben“ ansieht. Falls n sehr groß ist, wirkt das zunächst hinreichend sicher. Nachteilig ist aber, dass Information über identische Blöcke preisgegeben wird. Bei manchen Dateien kommt das durchaus vor, z. B. enthalten MS-Word-Dateien lange Ketten aus den Bytes `00000000` und `00000001`. Es geht aber auch krasser: Bilddateien aus grafischen Darstellungen mit großen einfarbigen Flächen enthalten so viele identische Blöcke, dass die Struktur des Bildes im Geheimtext durchscheinen kann. Ein deutliches Beispiel findet man im Wikipedia-Artikel „Electronic Code Book Mode“.

Besser ist es daher, eine Diffusion über die Klartextblöcke hinweg zu erzeugen. Ein einfacher, aber wirksamer Ansatz dazu ist die Betriebsart CBC (= Cipher Block Chaining). Mit einem zufällig gewähltem Startwert c_0 (auch IV = „Initialisierungsvektor“ genannt) sieht das Verfahren so aus:



Beim CBC-Modus wird also nach folgender Formel verschlüsselt:

$$\begin{aligned} c_i &:= f(a_i + c_{i-1}) \quad \text{für } i = 1, \ldots, r \\ &= f(a_i + f(a_{i-1} + \cdots f(a_1 + c_0) \ldots)). \end{aligned}$$

Hier hängt nun jeder Geheimtextblock von *allen vorhergehenden* Klartextblöcken ab (Diffusion), und gleiche Klartextblöcke werden im allgemeinen verschieden chiffriert.

Die Formel für die Entschlüsselung ist

$$a_i = f^{-1}(c_i) + c_{i-1} \quad \text{für } i = 1, \ldots, r.$$

Frage: *Kann der Startwert c_0 bei Geheimhaltung als zusätzlicher Schlüssel dienen?* (Das wären im Beispiel DES 56 Bits des eigentlichen Schlüssels plus 64 Bits des Startwerts, also insgesamt 120 Bits.)

Antwort: Nein!

Begründung: Nur a_1 hängt beim Entschlüsseln von c_0 ab, d. h., es wird lediglich bekannter Klartext am Anfang etwas besser verschleiert, wenn c_0 geheim bleibt. Ist der zweite oder ein späterer Klartextblock bekannt, kann die Angreiferin wie bei ECB den Schlüssel bestimmen (durch vollständige Suche oder einen weiteren Angriff mit bekanntem Klartext).

Für weitere Betriebsarten kann man den Wikipedia-Eintrag „Betriebsmodus (Kryptographie)“ nachlesen. Erwähnt werden soll noch, dass die Betriebsarten OFB (= Output Feedback) und CTR (= Counter) aus einer Bitblock-Verschlüsselung eine Bitstrom-Verschlüsselung machen.

8.2.5 Statistische Analysen

Für die Kryptoanalyse von Bitblock-Chiffren sind folgende allgemeine Ansätze bekannt:

1. Exhaustion = vollständige Schlüsselsuche
2. algebraischer Angriff, siehe Abschnitt 8.2.2
3. statistische Angriffe auf versteckte Linearität:
   (a) Lineare Kryptoanalyse (Matsui/Yamagishi 1992). Sie ist das Thema der Abschnitte 8.2.6 ff.
   (b) Differenzielle Kryptoanalyse (Murphy, Shamir, Biham 1990 – bei IBM und NSA schon 1974 bekannt).
   (c) Verallgemeinerungen und Mischformen der Ansätze (a) und (b).

Die statistischen Angriffe sind allerdings kaum konkret zum Brechen einer Chiffre im Sinne der klassischen Kryptoanalyse geeignet. Sie setzen meist so viele bekannte Klartexte voraus, wie man in realistischen Situationen kaum je erhalten kann. Daher sollte man tatsächlich eher von Analysen als von Angriffen sprechen. Ihr Sinn liegt vor allem darin, sinnvolle Maße für Teilaspekte der Sicherheit von Bitblock-Chiffren zu gewinnen. Ein solches Sicherheitsmaß ist z. B. die Anzahl bekannter Klartextblöcke, die man für einen Angriff benötigt. Chiffren, die selbst unter unrealistischen Annahmen über die Kenntnisse der Angreiferin sicher sind, können als in der Praxis besonders sicher gelten.

Bei SP-Netzen startet man die Analyse bei den nichtlinearen Bestandteilen der einzelnen Runden, insbesondere bei den S-Boxen, und versucht, einen potenziellen Angriff über mehrere Runden auszudehnen. Dabei sieht man oft, wie die Schwierigkeit des Angriffs mit der Rundenzahl zunimmt. So erhält man Kriterien, ab wievielen Runden eine Chiffre „sicher“ ist – zumindest vor diesem speziellen Angriff.

Kriterien für Bitblock-Chiffren

Zur Vermeidung der Angriffe sollten Bitblock-Chiffren bzw. deren Runden-Abbildungen oder deren nichtlineare Bausteine, die S-Boxen, einige Kriterien erfüllen.

- **Balanciertheit:** Alle Urbildmengen sind gleich groß, d. h., die Werte der Abbildung sind gleichmäßig verteilt. Unregelmäßigkeiten in der Verteilung würden einen Ansatz zu statistischen Auswertungen bieten.
- **Diffusion/Lawineneffekt:** Bei Änderung eines Klartextbits ändern sich sehr viele, am besten ca. 50%, der Geheimtextbits. Hierdurch soll die Ähnlichkeit von Klartexten (und Schlüsseln) verschleiert werden, d. h., die Angreiferin soll nicht erkennen, wenn sie den Klartext (oder Schlüssel) fast richtig geraten hat.
- **Algebraische Komplexität:** Die Bestimmung von Urbildern oder Teilen davon soll auf möglichst schwer lösbare Gleichungen führen. Diese Forderung hängt mit der Nichtlinearität der Abbildung zusammen. Eine genauere Untersuchung führt auf den Begriff der algebraischen Immunität.
- **Nichtlinearität:** Hier gibt es Kriterien, die auch „versteckte“ Nichtlinearität messen und vergleichsweise leicht zu beschreiben und handzuhaben sind; sie zeigen u. a., ob die Abbildungen anfällig für lineare oder differenzielle Kryptoanalyse sind [Pom08].
  - Das lineare Potenzial soll möglichst gering, das lineare Profil möglichst ausgeglichen sein.
  - Das differenzielle Potenzial soll möglichst gering, das Differenzenprofil möglichst ausgeglichen sein.

Einige dieser Kriterien lassen sich gleichzeitig erfüllen, andere widersprechen sich teilweise, so dass das Design einer Bitblock-Chiffre insbesondere eine Abwägung der verschiedenen Kriterien erfordert; statt der Optimierung nach einem Kriterium ist ein möglichst gleichmäßig hohes Niveau bezüglich aller Kriterien anzustreben.

8.2.6 Die Idee der linearen Kryptoanalyse

Die genauere Beschreibung der statistischen Angriffe würde jeweils umfangreiche Extra-Kapitel erfordern. Am leichtesten zugänglich ist die lineare Kryptoanalyse, deren Anfangsgründe wir wenigstens exemplarisch etwas näher ansehen wollen.

Wir betrachten eine Bitblock-Chiffre F mit Blocklänge n und Schlüssellänge l,

$$F\colon \mathbb{F}_2^n \times \mathbb{F}_2^l \longrightarrow \mathbb{F}_2^n,$$

und stellen uns die Argumente von F als Klartexte $a \in \mathbb{F}_2^n$ und Schlüssel $k \in \mathbb{F}_2^l$, die Werte von F als Geheimtexte $c \in \mathbb{F}_2^n$ vor. Eine **lineare Relation** zwischen Klartext $a \in \mathbb{F}_2^n$, Schlüssel $k \in \mathbb{F}_2^l$ und Geheimtext $c = F(a,k) \in \mathbb{F}_2^n$ kann man durch drei Linearformen

$$\alpha\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2, \quad \beta\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2 \quad \text{und} \quad \kappa\colon \mathbb{F}_2^l \longrightarrow \mathbb{F}_2$$

als

$$\kappa(k) = \alpha(a) + \beta(c) \tag{8.4}$$

beschreiben. Im einfachsten Fall würden α, β und κ jeweils ein bestimmtes Bit aus den jeweiligen Bitblöcken herauspicken, und Gleichung (8.4) würde ein Bit des Schlüssels als Summe bestimmter Bits von Klartext und Geheimtext ausdrücken. Im allgemeinen Fall werden nicht einzelne Bits herausgepickt, sondern es wird jeweils eine Summe über mehrere Bits gebildet. Sei $I = (i_1, \ldots, i_r)$ die Indexmenge, die der Linearform κ entspricht, d. h. $\kappa(k) = k_{i_1} + \cdots + k_{i_r}$.

Dann erhalten wir, ausführlich geschrieben, aus (8.4) eine Gleichung für die Summe dieser Schlüsselbits $k_{i_1}, \ldots, k_{i_r}$:

$$k_{i_1} + \cdots + k_{i_r} = \alpha(a) + \beta(c),$$

und müssten für einen algebraischen Angriff bei bekanntem Klartext a nach Elimination eines dieser Bits nur noch $l - 1$ unbekannte Schlüsselbits bestimmen.

Nun ist im Allgemeinen die Chance, dass die Relation (8.4) für konkrete zufällige Werte von k, a und c gilt, ungefähr $\frac{1}{2}$ – auf beiden Seiten steht ja nach Auswertung jeweils 0 oder 1. Das Beste, was man mit Blick auf die Sicherheit erwarten kann, ist, dass eine solche Relation bei festem zu bestimmenden Schlüssel k für ziemlich genau die Hälfte aller Klartexte a gilt (mit zugehörigen Geheimtexten $c = F(a, k)$), weil das dem Zufall entspräche. Ist die **Wahrscheinlichkeit** der Relation,

$$p_{F,\alpha,\beta,\kappa}(k) := \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \kappa(k) = \alpha(a) + \beta(F(a,k))\},$$

deutlich größer als $\frac{1}{2}$, gilt die Relation überzufällig oft. Das ergäbe eine auffällige Wahrscheinlichkeit für die Werte der Bits von k und würde für die Kryptoanalyse einen kleinen Vorteil mit sich bringen. Ist die Wahrscheinlichkeit andererseits deutlich kleiner als $\frac{1}{2}$, gilt die komplementäre Relation $\kappa(k) = \alpha(a) + \beta(c) + 1$ überzufällig oft – das wäre ein ebenso ausnützbares Ungleichgewicht. Weil die Situation hinsichtlich der Abweichung der Wahrscheinlichkeit vom Idealwert $\frac{1}{2}$ also symmetrisch ist[42], ist es oft zweckmäßig, symmetrische Größen zu verwenden[43], die **Input-Output-Korrelation**[44]:

$$\tau_{F,\alpha,\beta,\kappa}(k) := 2p_{F,\alpha,\beta,\kappa}(k) - 1$$

(kurz: I/O-Korrelation) und das **Potenzial** der linearen Relation[45]:

$$\lambda_{F,\alpha,\beta,\kappa}(k) := \tau_{F,\alpha,\beta,\kappa}(k)^2.$$

Die I/O-Korrelation liegt zwischen -1 und 1. Das Potenzial liegt zwischen 0 und 1 und misst die Abweichung der Wahrscheinlichkeit vom Wert $\frac{1}{2}$. Es ist im guten Fall 0, im schlechten Fall 1. Dieser „schlechte“ Extremfall würde eine exakte und direkt nutzbare Relation für die Schlüsselbits implizieren. Abbildung 8.3 zeigt den Zusammenhang. Sie wurde mit dem SageMath-Beispiel 8.4 erzeugt.

Der Schlüssel k ist allerdings das Ziel des Angriffs und die Wahrscheinlichkeit $p_{F,\alpha,\beta,\kappa}(k)$ in der Angriffssituation unbekannt. Für die Kryptoanalyse ist es daher angebracht, die Wahrscheinlichkeit einer linearen Relation noch über alle Schlüssel zu mitteln:

$$p_{F,\alpha,\beta,\kappa} := \frac{1}{2^{n+l}} \#\{(a,k) \in \mathbb{F}_2^n \times \mathbb{F}_2^l \mid \kappa(k) = \alpha(a) + \beta(F(a,k))\}. \tag{8.5}$$

[42] und weil I/O-Korrelation und Potenzial eine multiplikative Eigenschaft haben, siehe Satz 8.2.6.

[43] Diese kommen in der Literatur meist ohne explizite Namensgebung vor, so z. B. in den Originalarbeiten von Matsui.

[44] Das ist die Korrelation zwischen zwei Booleschen Funktionen auf $\mathbb{F}_2^n$, nämlich $\alpha + \kappa(k)$ und $\beta \circ F_k$. (Bei festem k ist $\kappa(k)$ eine Konstante, also 0 oder 1). Die erstere Funktion pickt Input-Bits heraus, die letztere Output-Bits. Allgemein bezeichnet man als Korrelation zweier Boolescher Funktionen $f, g \colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ die Differenz

$$c(f,g) := \frac{1}{2^n} \cdot [\#\{x \in \mathbb{F}_2^n \mid f(x) = g(x)\} - \#\{x \in \mathbb{F}_2^n \mid f(x) \neq g(x)\}]$$

[45] Für diese Bezeichnung („potential“) ist der älteste bekannte Nachweis der Beitrag von Kaisa Nyberg auf der EuroCrypt 1994. Mathematisch weniger elegant, aber oft verwendet, ist der „Bias“ $\left|p - \frac{1}{2}\right| = \sqrt{\lambda}/2$.

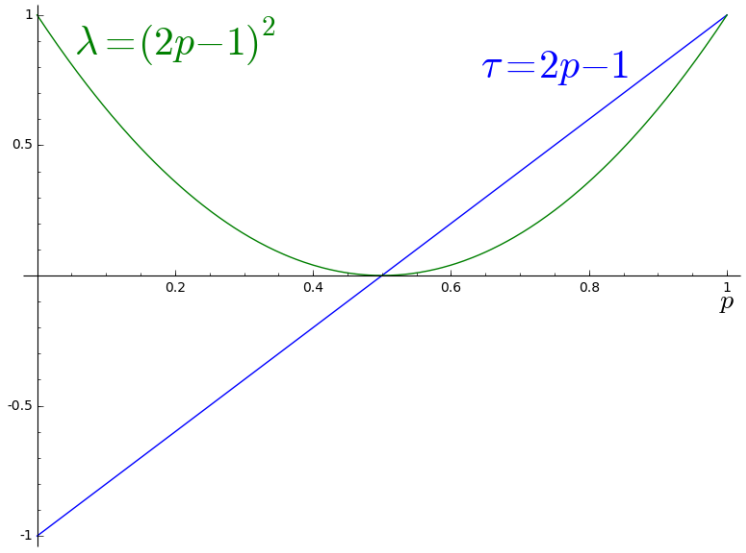


Abbildung 8.3: Zusammenhang zwischen Wahrscheinlichkeit p, I/O-Korrelation τ und Potenzial λ

SageMath-Beispiel 8.4 Plot von I/O-Korrelation und Potenzial

```
sage: plot1 = plot(2*x-1, (x,0,1))
sage: plot2 = plot((2*x - 1)**2, (x,0,1), color = 'green')
sage: xlabel = text('$p$', (1.0, -0.1), fontsize = 20, color = 'black')
sage: legend1 = text('$\tau = 2p - 1$', (0.75,0.8), fontsize = 30)
sage: legend2 = text('$\lambda = (2p - 1)^2$', (0.2,0.9), fontsize = 30,\\
        color = 'green')
sage: show(plot1 + plot2 + xlabel + legend1 + legend2)
```

Diese Größe ist, zumindest theoretisch, wenn man die Effizienzfrage außer Acht lässt, allein aus der Definition der Chiffre F bestimmbar. Ihre Bestimmung läuft allerdings auf eine Exhaustion aller Klartexte und Schlüssel hinaus und ist daher bei einer realen Chiffre mit genügend großen Blocklängen oft wirklich nur theoretisch. Auch hierfür werden I/O-Korrelation und Potenzial definiert:

$$\tau_{F,\alpha,\beta,\kappa} := 2p_{F,\alpha,\beta,\kappa} - 1,$$
$$\lambda_{F,\alpha,\beta,\kappa} := \tau^2_{F,\alpha,\beta,\kappa}.$$

Shamir[46] bemerkte schon 1985, dass es überzufällige lineare Relationen für die S-Boxen des DES-Verfahrens gibt. Es dauerte allerdings weitere sieben Jahre, bis es Matsui[47] (nach ersten Versuchen von Gilbert und Chassé 1990 mit der Chiffre FEAL) gelang, diese Beobachtung systematisch auszunutzen. Er ging zur Schätzung[48] von $\kappa(k)$ wie folgt vor (im Fall $p_{F,\alpha,\beta,\kappa} > \frac{1}{2}$, sonst muss man bei der Entscheidung die Werte 0 und 1 vertauschen[49]):

[46] Adi Shamir, israelischer Kryptologe, Miterfinder des RSA-Verfahrens, *6.7.1952.
[47] Mitsuru Matsui, japanischer Kryptologe, *16.9.1961.
[48] Das ist eine sogenannte Maximum-Likelihood-Schätzung, d. h., man entscheidet sich für diejenige von mehreren Hypothesen (hier sind es nur zwei), unter deren Annahme das Beobachtungsergebnis die höchste Wahrscheinlichkeit hat.
[49] Im Fall $p_{F,\alpha,\beta,\kappa} = \frac{1}{2}$ ist die Methode in dieser Form unbrauchbar.

1. [**Sammelphase**] Man sammelt N Klartext-Geheimtextpaare $(a_1, c_1), \ldots, (a_N, c_N)$.
2. [**Auszählung**] Man bestimmt die Anzahl
$$t := \#\{i = 1, \ldots, N \mid \alpha(a_i) + \beta(c_i) = 0\}.$$
3. [**Mehrheitsentscheidung**] aufgrund von t:
   - Ist $t > \frac{N}{2}$, schätzt man $\kappa(k) = 0$.
   - Ist $t < \frac{N}{2}$, schätzt man $\kappa(k) = 1$.

Der Fall $t = \frac{N}{2}$ ist unergiebig, kommt aber selten vor – man entscheidet zufällig zwischen 0 und 1 oder gibt eine entsprechende Rückmeldung[50]. Im SageMath-Beispiel 8.5 steht der Programmcode, eine konkrete Anwendung folgt gleich als Beispiel im nächsten Unterabschnitt.

SageMath-Beispiel 8.5 Matsui-Test. Die Linearformen sind `a` für α und `b` für β. Die Liste `pc` enthält `N` Paare von Klartext und Geheimtext. Der Boolesche Wert `compl` gibt an, ob das als Ergebnis geschätzte Bit invertiert werden soll. Die Ausgabe ist ein Tripel aus der Anzahl `t` der gezählten Nullen, dem geschätzten Bit und einem Booleschen Wert, der angibt, ob das Bit deterministisch bestimmt (`True`) oder im Grenzfall zufällig bestimmt (`False`) wurde. Verwendet wird die Funktion `binScPr` aus dem SageMath-Beispiel 8.39 im Anhang 8.4.3.

```
def Matsui_Test(a, b, pc, compl = False):
  """Matsui's test for linear cryptanalysis"""
  N = len(pc)
  results = []
  for pair in pc:
    ax = binScPr(a,pair[0])
    by = binScPr(b,pair[1])
    result = (ax + by) % 2
    results.append(result)
  t = 0
  for bb in results:
    if bb == 0:
      t = t + 1
  if 2*t > N:
    if compl:
      return [t,1,True]
    else:
      return [t,0,True]
  elif 2*t_0 < N:
    if compl:
      return [t,0,True]
    else:
      return [t,1,True]
  else:
    return [t,randint(0,1),False]
```

[50] am besten beides wie im SageMath-Beispiel 8.5

Wenn man eine lineare Relation mit hinreichend großem Potenzial, d. h. hinreichend weit von $\frac{1}{2}$ abweichender Wahrscheinlichkeit, erwischt hat, wird die Erfolgswahrscheinlichkeit dieses Verfahrens bei hinreichend großem N gut sein. Das erlaubt dann, die Anzahl der unbekannten Schlüsselbits durch Elimination um 1 zu verringern.

Als theoretisches Ergebnis aus diesen Überlegungen werden wir einen Zusammenhang zwischen der Anzahl N von benötigten Klartextblöcken und der Erfolgswahrscheinlichkeit erhalten, siehe Tabelle 8.11.

Je mehr solcher linearen Relationen die Angreiferin mit genügend hoher Gewissheit findet, desto stärker kann sie die Größe des Schlüsselraums einschränken, bis schließlich eine Exhaustion über die noch in Frage kommenden Schlüssel in den Bereich des Machbaren rückt. Ein konkretes Beispiel in Abschnitt 8.2.12 wird dies illustrieren.

Beispiel

Für ein konkretes Beispiel mit $n = l = 4$ betrachten wir die Boolesche Abbildung[51] f, die durch die Wertetabelle 8.8 gegeben ist, und bilden damit die Bitblock-Chiffre (s. a. Abbildung 8.4)

$$F\colon \mathbb{F}_2^4 \times \mathbb{F}_2^4 \longrightarrow \mathbb{F}_2^4, \quad F(a,k) := f(a+k).$$

Das SageMath-Beispiel 8.6 definiert diese Boolesche Abbildung f unter dem Namen `S0` und verwendet die Klassen `BoolF` und `BoolMap` aus dem Anhang 8.4.6. Dabei geben die Spalten der (impliziten) definierenden Matrix genau die Werte der Abbildung an, wie man sie auch in Tabelle 8.8 in der Spalte $y = f(x)$ wiederfindet. (D. h., das SageMath-Beispiel 8.6 und die Tabelle 8.8 enthalten äquivalente Definitionen der Abbildung f.) Eine exemplarische Evaluation illustriert das (für die dritte Spalte, entsprechend dem Argument `0010`).

SageMath-Beispiel 8.6 Eine Boolesche Abbildung (S-Box S_0 von LUCIFER)

```
f1 = BoolF([1,1,0,1,1,1,1,0,0,0,0,0,1,0,0,1])
f2 = BoolF([1,1,1,0,1,1,0,0,0,1,0,0,0,1,1,0])
f3 = BoolF([0,1,1,1,1,0,1,0,1,1,1,0,0,0,0,0])
f4 = BoolF([0,1,1,0,0,1,1,0,0,0,1,1,1,0,1,0])
S0 = BoolMap([f1,f2,f3,f4])
# Sample evaluation
sage: S0.valueAt([0,0,1,0])
[0, 1, 1, 1]
```

Wir verschlüsseln mit dem Schlüssel $k =$ `1000`, den wir später zur Probe angreifen wollen. Für eine lineare Relation betrachten wir die Linearformen

$$\alpha(a) = a_4, \quad \beta(c) = c_1 + c_2 + c_4, \quad \kappa(k) = k_4;$$

wir werden in Abschnitt 8.2.7 sehen, dass mit diesen Linearformen die Relation $\kappa(k) = \alpha(a) + \beta(c)$ für F eine Wahrscheinlichkeit deutlich $> \frac{1}{2}$ hat. Tabelle 8.9 zeigt die Verschlüsselung dreier Klartexte a, die wir später als bekannte Klartexte annehmen wollen. Die Werte für c wurden aus der Wertetabelle 8.8 abgelesen. Die Anzahl t der beobachteten Werte 0 von $\alpha(a) + \beta(c)$ ist $t = 2$. Die Mehrheitsentscheidung führt also zu der Schätzung $k_4 = 0$ (von der wir wissen, dass sie korrekt ist).

[51] f ist nebenbei bemerkt die S-Box S_0 von LUCIFER, das um 1970 als Vorgänger-Verfahren von DES entwickelt wurde.

x	$y = f(x)$	$\alpha(x) = x_4$	$\beta(y) = y_1 + y_2 + y_4$
0 0 0 0	1 1 0 0	0	0
0 0 0 1	1 1 1 1	1	1
0 0 1 0	0 1 1 1	0	0
0 0 1 1	1 0 1 0	1	1
0 1 0 0	1 1 1 0	0	0
0 1 0 1	1 1 0 1	1	1
0 1 1 0	1 0 1 1	0	0
0 1 1 1	0 0 0 0	1	0
1 0 0 0	0 0 1 0	0	0
1 0 0 1	0 1 1 0	1	1
1 0 1 0	0 0 1 1	0	1
1 0 1 1	0 0 0 1	1	1
1 1 0 0	1 0 0 1	0	0
1 1 0 1	0 1 0 0	1	1
1 1 1 0	0 1 0 1	0	0
1 1 1 1	1 0 0 0	1	1

Tabelle 8.8: Wertetabelle einer Booleschen Abbildung $f : \mathbb{F}_2^4 \longrightarrow \mathbb{F}_2^4$ und zwei Linearformen

a	$a + k$	c	$\alpha(a)$	$\beta(c)$	$\alpha(a) + \beta(c)$
0010	1010	0011	0	1	1
0101	1101	0100	1	1	0
1010	0010	0111	0	0	0

Tabelle 8.9: Schätzung eines Schlüsselbits nach Matsui unter Verwendung von drei bekannten Klartexten

Das war mit dem Bleistift auf Papier ganz leicht auszuzählen. Dennoch ist der Nachvollzug in SageMath im Hinblick auf kompliziertere Beispiele instruktiv. Das geschieht im SageMath-Beispiel 8.7. Dabei wird die Funktion `xor` aus dem SageMath-Beispiel 8.39 im Anhang 8.4.3 sowie die im SageMath-Beispiel 8.6 definierte Abbildung `S0` verwendet. Das Ergebnis `[2, 0, True]` besagt, dass 2 Nullen unter den ausgezählten Werten waren, was zur Mehrheitsentscheidung 0 führt, die (wegen `True`) nicht durch zufällige Auslosung bei Gleichstand ermittelt wurde.

SageMath-Beispiel 8.7 Ein Beispiel für den Matsui-Test

```
sage: k = [1,0,0,0]
sage: alpha = [0,0,0,1]
sage: beta = [1,1,0,1]
sage: plist = [[0,0,1,0],[0,1,0,1],[1,0,1,0]]
sage: xlist = []
sage: xclist = []
sage: pclist = []
sage: for i in range(0,len(plist)):
....:     x = xor(plist[i],k)
....:     xlist.append(x)
....:
sage: xlist
[[1, 0, 1, 0], [1, 1, 0, 1], [0, 0, 1, 0]]
sage: for i in range(0,len(plist)):
....:     val = S0.valueAt(xlist[i])
....:     xclist.append([xlist[i],val])
....:     pclist.append([plist[i],val])
....:
sage: Matsui_Test(alpha,beta,pclist,False)
[2, 0, True]
```

Damit das Verfahren im allgemeinen Fall Erfolg verspricht, sind folgende Fragen zu klären:

1. Wie findet man lineare Relationen von möglichst auffälliger Wahrscheinlichkeit? Insbesondere im Hinblick darauf, dass die Auswertung der Formel (8.5) am Effizienzproblem scheitert.

2. Da Bitblock-Chiffren meistens aus vielen Runden zusammengesetzt sind, fragt man weiter:

   (a) Wie findet man bei einer iterierten Bitblock-Chiffre brauchbare lineare Relationen für die Rundenfunktion?

   (b) Wie setzt man diese über die Runden hinweg zu linearen Relationen für die ganze Chiffre zusammen?

   (c) Wie bestimmt man die Wahrscheinlichkeit einer zusammengesetzten linearen Relation für die ganze Chiffre aus der für die einzelnen Runden?

Die Antwort auf die erste Frage und Teil (a) der zweiten heißt: Aus dem linearen Profil, siehe Abschnitt 8.2.8. Die anschließenden Teilfragen führen zur Untersuchung von linearen Pfaden, siehe Abschnitt 8.2.10, und zur Kumulation von Wahrscheinlichkeiten, siehe Satz 8.2.7. Für (c) kommt dabei letztendlich eine brauchbare Faustregel heraus.

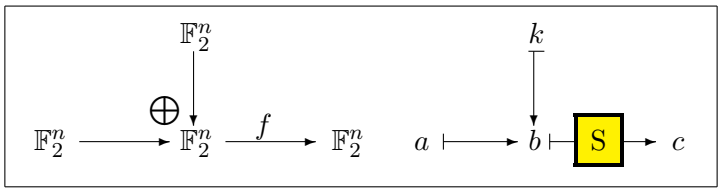


Abbildung 8.4: Ein (viel) zu einfaches Beispiel

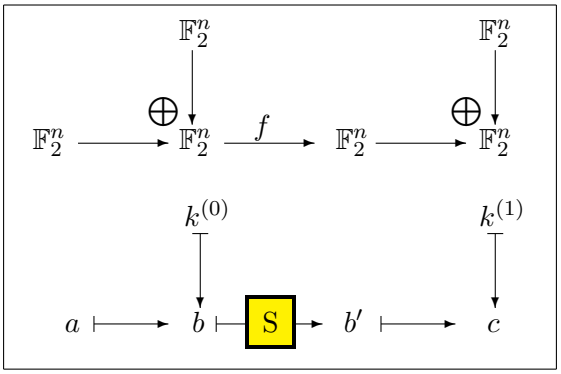


Abbildung 8.5: Beispiel A: Eine Einrunden-Chiffre

8.2.7 Beispiel A: Eine Einrunden-Chiffre

Es werden Beispiele betrachtet, die als ernsthafte Blockchiffren viel zu einfach sind, aber das Prinzip der linearen Kryptoanalyse anschaulich und nachvollziehbar demonstrieren. Dabei werden stets Rundenfunktionen der Gestalt $f(a+k)$ betrachtet, d. h., der Schlüssel bzw. ein n-bittiger Teil davon wird vor der Anwendung einer bijektiven S-Box $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$ binär auf den Klartext aufaddiert[52]. Das einfachste denkbare Modell, die Verschlüsselung nach der Vorschrift

$$c = f(a+k)$$

wie im Beispiel des Abschnitts 8.2.6, siehe Abbildung 8.4[53], ist dabei witzlos, da bei bekanntem Klartext die Gleichung nach dem Schlüssel k auflösbar ist[54]:

$$k = f^{-1}(c) + a.$$

Dieser einfache Angriff wird bei dem etwas komplizierteren Beispiel A mit

$$c = f(a + k^{(0)}) + k^{(1)}$$

verhindert (siehe Abbildung 8.5); hier ist der Ansatz der linearen Kryptoanalyse bereits sinnvoll: Sei (α, β) ein Paar von Linearformen mit

$$\beta \circ f(x) \overset{p}{\approx} \alpha(x), \tag{8.6}$$

[52] Das ist zwar eine sehr spezielle Art, den Schlüssel in das Verfahren einzubringen, aber dennoch realistisch. Die paradigmatischen Beispiel-Chiffren LUCIFER, DES und AES verfahren so. Bei AES [DR02] heißt das „key-alternating cipher structure“.

[53] In den grafischen Darstellungen hier und später wird die Abbildung f auf der elementweisen Ebene durch die S-Box S repräsentiert.

[54] Die Umkehrabbildung f^{-1} wird hier als der Angreiferin bekannt angenommen. Sie ist ja Teil des Algorithmus zur Entschlüsselung. Einweg-Verschlüsselungen, bei denen f^{-1} aus f nicht effizient herleitbar ist, bilden ein anderes Kapitel der Kryptographie.

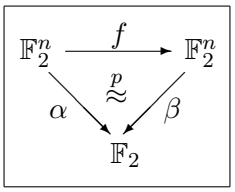


Abbildung 8.6: Diagramm für eine „approximative“ lineare Relation

wobei das Symbol $\overset{p}{\approx}$ zu lesen ist als „gleich mit Wahrscheinlichkeit p“, also

$$p = p_{f,\alpha,\beta} := \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \beta \circ f(x) = \alpha(x)\}.$$

Repräsentiert wird die Formel (8.6) durch das Diagramm in Abbildung 8.6. Die Linearform κ der allgemeinen Theorie tritt hier nicht explizit auf: Da die Schlüsselbits auf Klartext und („intermediären“) Geheimtext einfach aufaddiert werden, ist $\kappa = \alpha$ für $k^{(0)}$ und $\kappa = \beta$ für $k^{(1)}$, also $\kappa(k^{(0)}, k^{(1)}) = \alpha(k^{(0)}) + \beta(k^{(1)})$.

Wie hängt das mit der allgemeinen Situation aus Abschnitt 8.2.6 zusammen? Für das Beispiel A ist

- die Schlüssellänge $l = 2n$, der Schlüsselraum ist $\mathbb{F}_2^{2n}$, und Schlüssel haben die Gestalt $k = (k^{(0)}, k^{(1)})$ mit $k^{(0)}, k^{(1)} \in \mathbb{F}_2^n$.
- Die Chiffre ist durch die Abbildung

  $$F\colon \mathbb{F}_2^n \times \mathbb{F}_2^n \times \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n, \quad (a, k^{(0)}, k^{(1)}) \mapsto f(a + k^{(0)}) + k^{(1)},$$

  definiert.
- Die Linearform $\kappa\colon \mathbb{F}_2^n \times \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ ist durch $\kappa(k^{(0)}, k^{(1)}) = \alpha(k^{(0)}) + \beta(k^{(1)})$ gegeben.

Die Wahrscheinlichkeit einer linearen Relation für einen festen Schlüssel $k = (k^{(0)}, k^{(1)})$ ist damit

$$\begin{aligned} p_{F,\alpha,\beta,\kappa}(k) &= \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \kappa(k) = \alpha(a) + \beta(F(a,k))\} \\ &= \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \alpha(k^{(0)}) + \beta(k^{(1)}) = \alpha(a) + \beta(f(a + k^{(0)}) + k^{(1)})\} \\ &= \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \alpha(k^{(0)}) = \alpha(a) + \beta(f(a + k^{(0)}))\}, \end{aligned}$$

da $\beta(k^{(1)})$ auf beiden Seiten der Gleichung innerhalb der Mengenklammern vorkommt und daher einfach weggelassen werden kann.

Dieser Ausdruck ist unabhängig von $k^{(1)}$, und an der leicht umgeformten Gleichung

$$p_{F,\alpha,\beta,\kappa}(k) = \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \alpha(a + k^{(0)}) = \beta(f(a + k^{(0)}))\}$$

sieht man, dass er für alle $k^{(0)}$ den gleichen Wert hat, da bei festem $k^{(0)}$ mit a auch $a + k^{(0)}$ ganz $\mathbb{F}_2^n$ durchläuft. Dieser Wert muss daher mit dem Mittelwert über alle k übereinstimmen:

$$p_{F,\alpha,\beta,\kappa}(k) = p_{F,\alpha,\beta,\kappa} = \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \alpha(x) = \beta(f(x))\} = p.$$

Mit dieser Überlegung ist gezeigt:

Satz 8.2.2. *In der Situation des Beispiels A nimmt die Wahrscheinlichkeit $p_{F,\alpha,\beta,\kappa}(k)$ für jeden Schlüssel $k \in \mathbb{F}_2^{2n}$ den gleichen Wert*

$$p = \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \alpha(x) = \beta(f(x))\}$$

an, insbesondere ist p auch der Mittelwert nach Gleichung (8.5).

Mit den Bezeichnungen aus Abbildung 8.5 gilt nun

$$\begin{aligned}\beta(c) &= \beta(b' + k^{(1)}) = \beta(b') + \beta(k^{(1)}) \\ &\overset{p}{\approx} \alpha(b) + \beta(k^{(1)}) = \alpha(a + k^{(0)}) + \beta(k^{(1)}) = \alpha(a) + \alpha(k^{(0)}) + \beta(k^{(1)}).\end{aligned}$$

Als lineare Relation für die Bits des Schlüssels $k = (k^{(0)}, k^{(1)})$ erhalten wir

$$\alpha(k^{(0)}) + \beta(k^{(1)}) \overset{p}{\approx} \alpha(a) + \beta(c).$$

Ein analoger Schluss lässt sich für die komplementäre Relation

$$\beta \circ f(x) \overset{1-p}{\approx} \alpha(x) + 1$$

durchführen. Insgesamt ist damit gezeigt:

Satz 8.2.3. *Im Beispiel A sei (α, β) ein Paar von Linearformen für f mit der Wahrscheinlichkeit p wie in Formel (8.6). Dann ist $\hat{p} = \max\{p, 1-p\}$ die Erfolgswahrscheinlichkeit für die Bestimmung eines Schlüsselbits aus* einem *bekannten Klartext durch diese lineare Relation.*

Beispiel

Nehmen wir als konkretes Beispiel $n = 4$ und für f wieder die S-Box S_0 von LUCIFER. Die beiden rechten Spalten der Tabelle 8.8 zeigen, dass die durch (α, β) mit $\alpha(x) = x_4$ und $\beta(y) = y_1 + y_2 + y_4$ definierte lineare Relation die Wahrscheinlichkeit $p_{f,\alpha,\beta} = \frac{14}{16} = \frac{7}{8}$ hat[55].

Die konkreten Rundenschlüssel seien $k^{(0)} = \texttt{1000}$ und $k^{(1)} = \texttt{0001}$. Die Tabelle 8.10 über alle 16 möglichen Klartexte zeigt, dass $\alpha(a) + \beta(c)$ den Wert $1 = \alpha(k^{(0)}) + \beta(k^{(1)})$ für die Summe der Schlüsselbits genau 14-mal annimmt, wie es sein soll.

Wie groß ist nun die Erfolgswahrscheinlichkeit p_N dafür, diesen Wert richtig zu schätzen, wenn man $N = 1, 2, \ldots$ zufällige bekannte Klartexte aus der Menge der 2^n möglichen zur Verfügung hat (zu gegebenen festen Linearformen α und β mit $p = p_{f,\alpha,\beta}$)? Das ist genau eine konkrete Einkleidung der Fragestellung der hypergeometrischen Verteilung, und daher gilt (ohne Beweis und ohne Erklärung der hypergeometrischen Verteilung):

Satz 8.2.4. *Im Beispiel A sei (α, β) ein Paar von Linearformen, das eine lineare Relation für f mit der Wahrscheinlichkeit p definiert. Dann ist die Erfolgswahrscheinlichkeit für die Bestimmung eines Schlüsselbits aus N bekannten Klartexten durch diese lineare Relation gerade die kumulierte Wahrscheinlichkeit $p_N = p_N^{(s)}$ der hypergeometrischen Verteilung zu den Parametern 2^n, $s = \hat{p} \cdot 2^n$ und N mit $\hat{p} = \max\{p, 1-p\}$.*

[55] Deren Größe ist ein starkes Indiz dafür, dass die Designer von LUCIFER die lineare Kryptoanalyse noch nicht kannten.

a	b	b'	c	$\alpha(a)+\beta(c)=a_4+c_1+c_2+c_4$
0000	1000	0010	0011	1
0001	1001	0110	0111	1
0010	1010	0011	0010	0
0011	1011	0001	0000	1
0100	1100	1001	1000	1
0101	1101	0100	0101	1
0110	1110	0101	0100	1
0111	1111	1000	1001	1
1000	0000	1100	1101	1
1001	0001	1111	1110	1
1010	0010	0111	0110	1
1011	0011	1010	1011	1
1100	0100	1110	1111	1
1101	0101	1101	1100	1
1110	0110	1011	1010	1
1111	0111	0000	0001	0

Tabelle 8.10: Eine lineare Relation für die Schlüsselbits (b entsteht aus a durch Addition von $k^{(0)}$, also "Umkippen" des ersten Bits, b' aus b durch Anwendung von f, c aus b' durch Addition von $k^{(1)}$.

Verlässt man die exakte Mathematik und geht, wie oft in der angewandten Statistik, zu asymptotischen Näherungsformeln über, so kann man unter den (sehr vage formulierten) Voraussetzungen „p nicht zu weit von $\frac{1}{2}$ entfernt, $N \ll 2^n$, aber N nicht zu klein," die hypergeometrische Verteilung durch die Normalverteilung approximieren und erhält

$$p_N \approx \frac{1}{\sqrt{2\pi}} \cdot \int_{-\infty}^{\sqrt{N\lambda}} e^{-t^2/2}\, dt, \tag{8.7}$$

wobei $\lambda = (2p-1)^2$ das Potenzial der linearen Relation ist. Zusammen mit den bekannten Werten für die Normalverteilung[56] ergibt das die Tabelle 8.11. D. h., um eine Erfolgswahrscheinlichkeit von etwa 95% zu erreichen, braucht man $N \approx \frac{3}{\lambda}$ bekannte Klartexte. Im obigen konkreten Beispiel war $p = \frac{7}{8}$, also $\lambda = \frac{9}{16}$, und die Zahl der nötigen bekannten Klartexte für die 95-prozentige Erfolgswahrscheinlichkeit ist nach der Formel $N \approx 5$. Wir waren, siehe Tabelle 8.9, schon mit $N = 3$ erfolgreich gewesen; das ist nicht sehr überraschend, denn wie wir jetzt sehen, war die Erfolgswahrscheinlichkeit dafür immerhin knapp 90% (hier ist nämlich $N\lambda = \frac{27}{16} \approx 1,68\ldots$)[57].

[56] Man kann statt mit der Approximation durch die Normalverteilung auch direkt mit der hypergeometrischen Verteilung rechnen. Dann erhält man, besonders bei kleinem N, einen genaueren Wert, aber keine so einfache geschlossene Formel wie (8.7).

[57] Hier wird man die Voraussetzung „N nicht zu klein" der Approximation durch die Normalverteilung zu Recht anzweifeln müssen. In der Tat kann man leicht die exakten Werte für die hypergeometrische Verteilung bestimmen: Zieht man aus einer Urne mit 16 Kugeln, von denen 14 schwarz und 2 weiß sind, zufällig 3 Kugeln, so werden diese mit Wahrscheinlichkeit $\frac{26}{40}$ alle drei schwarz sein, und mit Wahrscheinlichkeit $\frac{13}{40}$ zwei davon schwarz und eine weiß, also mit Wahrscheinlichkeit $\frac{39}{40} = 97,5\%$ mindestens zwei schwarz. Das ist also deutlich mehr als die aus der Approximationsformel (8.7) bestimmten 90%. Die übrigen Wahrscheinlichkeiten sind $\frac{1}{40}$ für genau eine schwarze Kugel und 0 für drei weiße.

$N\lambda$	1	2	3	4	...	8	9
p_N	84,1%	92,1%	95,8%	97,7%	...	99,8%	99,9%

Tabelle 8.11: Zusammenhang zwischen der Anzahl der bekannten Klartexte und der Erfolgswahrscheinlichkeit

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1	8	6	6	8	8	6	6	8	8	6	6	8	8	14	6	8
2	8	10	8	6	4	6	8	6	6	12	6	8	10	8	6	8
3	8	12	10	6	12	8	10	6	6	6	8	8	10	10	8	8
4	8	8	4	8	8	8	8	4	10	6	6	6	10	6	10	10
5	8	10	10	12	8	10	6	8	10	8	4	10	10	8	8	6
6	8	10	8	10	8	10	8	10	8	10	8	2	8	10	8	10
7	8	8	10	6	8	8	2	6	8	8	10	6	8	8	10	6
8	8	8	6	10	6	10	8	8	4	8	10	10	10	10	12	8
9	8	10	8	10	6	4	10	8	8	6	8	6	6	8	10	4
10	8	6	10	8	6	8	8	10	6	4	8	6	12	6	6	8
11	8	12	8	8	6	6	6	10	10	6	10	10	8	8	8	12
12	8	8	10	10	6	10	8	4	6	6	8	8	4	8	6	10
13	8	6	12	6	6	8	10	8	10	8	6	8	8	10	12	8
14	8	6	10	12	10	4	8	6	8	10	10	8	10	8	8	10
15	8	8	8	8	10	6	6	10	4	8	4	8	6	6	10	10

Tabelle 8.12: Approximationstabelle der S-Box S_0 von LUCIFER – Zeilen- und Spaltenindizes sind durch Zahlen repräsentierte Linearformen, siehe Abschnitt 8.1.11. Die Wahrscheinlichkeiten erhält man nach Division durch 16.

8.2.8 Approximationstabelle, Korrelationsmatrix und lineares Profil

Die Häufigkeiten, mit denen lineare Relationen für eine Boolesche Abbildung (oder S-Box) $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ gelten, werden in einer $2^n \times 2^q$-Matrix zusammengefasst, die zu jedem Paar von Linearformen (α, β) die Anzahl der Argumente x mit $\beta \circ f(x) = \alpha(x)$ angibt und **Approximationstabelle** genannt wird[58]. Für die oben verwendete S-Box S_0 von LUCIFER ist sie in Tabelle 8.12 wiedergegeben. Der Eintrag 16 in der linken oberen Ecke besagt, dass die Relation $0 = 0$ immer, also in allen 16 Fällen gilt, und ist gleichzeitig der Hauptnenner, durch den man alle Einträge teilen muss, um die Wahrscheinlichkeiten zu erhalten; im allgemeinen Fall würde dort 2^n stehen. Die übrigen Einträge in der ersten Spalte (entsprechend $\beta = 0$) sind 8, weil jede von Null verschiedene Linearform α den Wert 0 in genau der Hälfte aller Fälle, hier also 8-mal, annimmt[59]. Für die erste Zeile gilt das analoge Argument – vorausgesetzt, f ist bijektiv[60].

Die **Korrelationsmatrix** und das **lineare Profil**[61] sind die entsprechenden Matrizen, deren

[58] Vorsicht mit den Literatur-Referenzen: Oft wird von allen Einträgen noch der Wert 2^{n-1} abgezogen, so z. B. bei der SageMath-Funktion `linear_approximation_matrix()`.

[59] In der Sprache der Linearen Algebra: Der Kern einer Linearform $\neq 0$ ist ein $(n-1)$-dimensionaler Unterraum.

[60] Im allgemeinen Fall, wo $q \neq n$ sein kann, müsste man hier „balanciert“ sagen, d. h., alle Urbildmengen sind gleich groß. Das geht natürlich nur im Fall $q \leq n$ wirklich.

[61] oder auch Linearitätsprofil; nicht zu verwechseln mit dem linearen Komplexitätsprofil einer Bitfolge, das mit Hilfe von linearen Schieberegistern definiert wird und auch oft Linearitätsprofil genannt wird.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	$\frac{3}{4}$	$-\frac{1}{4}$	0
2	0	$\frac{1}{4}$	0	$-\frac{1}{4}$	$-\frac{1}{2}$	$-\frac{1}{4}$	0	$-\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{2}$	$-\frac{1}{4}$	0	$\frac{1}{4}$	0	$-\frac{1}{4}$	0
3	0	$\frac{1}{2}$	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{2}$	0	$\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	$\frac{1}{4}$	$\frac{1}{4}$	0	0
4	0	0	$-\frac{1}{2}$	0	0	0	0	$-\frac{1}{2}$	$\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
5	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	0	$\frac{1}{4}$	$-\frac{1}{4}$	0	$\frac{1}{4}$	0	$-\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	0	0	$-\frac{1}{4}$
6	0	$\frac{1}{4}$	0	$\frac{1}{4}$	0	$\frac{1}{4}$	0	$\frac{1}{4}$	0	$\frac{1}{4}$	0	$-\frac{3}{4}$	0	$\frac{1}{4}$	0	$\frac{1}{4}$
7	0	0	$\frac{1}{4}$	$-\frac{1}{4}$	0	0	$-\frac{3}{4}$	$-\frac{1}{4}$	0	0	$\frac{1}{4}$	$-\frac{1}{4}$	0	0	$\frac{1}{4}$	$-\frac{1}{4}$
8	0	0	$-\frac{1}{4}$	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	0	0	$-\frac{1}{2}$	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	0
9	0	$\frac{1}{4}$	0	$\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{2}$	$\frac{1}{4}$	0	0	$-\frac{1}{4}$	0	$-\frac{1}{4}$	$-\frac{1}{4}$	0	$\frac{1}{4}$	$-\frac{1}{2}$
10	0	$-\frac{1}{4}$	$\frac{1}{4}$	0	$-\frac{1}{4}$	0	0	$\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{2}$	0	$-\frac{1}{4}$	$\frac{1}{2}$	$-\frac{1}{4}$	$-\frac{1}{4}$	0
11	0	$\frac{1}{2}$	0	0	$-\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	$\frac{1}{2}$
12	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	0	$-\frac{1}{2}$	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	$-\frac{1}{2}$	0	$-\frac{1}{4}$	$\frac{1}{4}$
13	0	$-\frac{1}{4}$	$\frac{1}{2}$	$-\frac{1}{4}$	$-\frac{1}{4}$	0	$\frac{1}{4}$	0	$\frac{1}{4}$	0	$-\frac{1}{4}$	0	0	$\frac{1}{4}$	$\frac{1}{2}$	0
14	0	$-\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$	$-\frac{1}{2}$	0	$-\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{1}{4}$	0	$\frac{1}{4}$	0	0	$\frac{1}{4}$
15	0	0	0	0	$\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Tabelle 8.13: Korrelationsmatrix der S-Box S_0 von LUCIFER – Zeilen- und Spaltenindizes sind durch Zahlen repräsentierte Linearformen.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{9}{16}$	$\frac{1}{16}$	0
2	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0
3	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0
4	0	0	$\frac{1}{4}$	0	0	0	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
5	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$
6	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{9}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$
7	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{9}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$
8	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0
9	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$
10	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0
11	0	$\frac{1}{4}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	0	$\frac{1}{4}$
12	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{16}$
13	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{4}$	0
14	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$
15	0	0	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$

Tabelle 8.14: Lineares Profil der S-Box S_0 von LUCIFER – Zeilen- und Spaltenindizes sind durch Zahlen repräsentierte Linearformen.

Einträge jeweils die I/O-Korrelation bzw. das Potenzial der linearen Relation enthalten. Man erhält die Korrelationsmatrix aus der Approximationstabelle, indem man erst die Einträge durch 2^n dividiert, um die jeweiligen Wahrscheinlichkeiten p zu erhalten; dann muss man noch die Wahrscheinlichkeiten nach der Formel $\tau = 2p - 1$ in I/O-Korrelationen umrechnen. Das lineare Profil entsteht dann, indem man die Einträge der Korrelationsmatrix einzeln quadriert.

Für S_0 ist die Korrelationsmatrix in Tabelle 8.13 und das lineare Profil in Tabelle 8.14 wiedergegeben. Auch hier sind die erste Zeile und die erste Spalte auffällig; die Nullen besagen, dass eine lineare Relation, an der die Linearform 0 beteiligt ist, das Potenzial 0 hat, also nutzlos ist. Die 1 links oben in der Ecke drückt aus, dass die Relation $0 = 0$ immer gilt, ist aber ebenso nutzlos. Das oben herausgepickte Paar (α, β) mit $\alpha(x) = x_4$ (repräsentiert durch `0001` $\hat{=} 1$) und $\beta(y) = y_1 + y_2 + y_4$ (repräsentiert durch `1101` $\hat{=} 13$) in Zeile 1, Spalte 13, hat den Maximalwert[62] $\frac{9}{16}$ für das Potenzial, der aber auch noch an den Stellen $(6, 11)$ und $(7, 6)$ des linearen Profils vorkommt.

Effiziente Berechnung per Fourier-Transformation

Man kann die Approximationstabelle „naiv“ durch Auszählen gewinnen und daraus die Korrelationsmatrix und das lineare Profil durch einfache (elementweise) Umrechnung herleiten. Ein effizienterer Algorithmus verwendet die Fourier[63]-Transformation, die im uns betreffenden binären Fall besonders einfach ist und wegen historisch unabhängiger Erfindungen hier auch Hadamard[64]-Transformation oder Walsh[65]-Transformation genannt wird. Diese Transformation wandelt eine *reellwertige* (!) Funktion $\varphi : \mathbb{F}_2^m \longrightarrow \mathbb{R}$ wieder in eine reellwertige Funktion $\hat{\varphi} : \mathbb{F}_2^m \longrightarrow \mathbb{R}$ um, die durch

$$\hat{\varphi}(u) := \sum_{x \in \mathbb{F}_2^m} \varphi(x) \cdot (-1)^{u \cdot x}.$$

definiert ist[66]. Dabei ist $u \cdot x$ das kanonische Skalarprodukt in $\mathbb{F}_2^m$. Der Exponent ist also ein Bit, aber das passt schon, da über der Basis -1 für jeden ganzzahligen Exponenten nur die Restklasse modulo 2 relevant ist.

Wir betrachten nun eine Boolesche Abbildung $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ und ihre **Indikatorfunktion** $\vartheta_f : \mathbb{F}_2^n \times \mathbb{F}_2^q \longrightarrow \mathbb{R}$,

$$\vartheta_f(x, y) := \begin{cases} 1, & \text{wenn } y = f(x), \\ 0 & \text{sonst.} \end{cases}$$

Bestimmen wir die Fourier-Transformation davon (mit $m = n + q$, wobei die Variablen auf Blöcke der Längen n und q verteilt werden):

$$\begin{aligned} \hat{\vartheta}_f(u, v) &= \sum_{x \in \mathbb{F}_2^n} \sum_{y \in \mathbb{F}_2^q} \vartheta_f(x, y)(-1)^{u \cdot x + v \cdot y} \\ &= \sum_{x \in \mathbb{F}_2^n} (-1)^{u \cdot x + v \cdot f(x)}. \end{aligned}$$

[62] Der eigentliche Maximalwert 1 in der linken oberen Ecke wird ignoriert, da er nutzlos ist.
[63] Joseph Fourier, französischer Mathematiker und Physiker, 21.3.1768–16.5.1830
[64] Jacques Hadamard, französischer Mathematiker, 8.12.1865–17.10.1963
[65] Joseph L. Walsh, US-amerikanischer Mathematiker, 21.9.1895–6.12.1973
[66] Das ist ein Spezialfall der diskreten Fourier-Transformation. Im allgemeinen Fall würde man statt -1 die komplexe N-te Einheitswurzel $\zeta = e^{2\pi i/N}$ verwenden und komplexwertige Funktionen über dem Ring $\mathbb{Z}/N\mathbb{Z}$ statt über $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ transformieren – oder Funktionen auf $\mathbb{Z}^m$, die in jeder Variablen die Periode N haben. Im binären Fall ist $N = 2$, und weil die zweite Einheitswurzel -1 reell ist, reicht es hier, die Diskussion auf reellwertige Funktionen zu beschränken.

Im Exponenten stehen die Linearformen $x \mapsto u \cdot x$ auf $\mathbb{F}_2^n$, die wir mit α bezeichnen wollen, und $y \mapsto v \cdot y$ auf $\mathbb{F}_2^q$, der wir den Namen β geben. Dann ist u die Bitblock-Interpretation von α und v die von β, und wir sehen im Exponenten etwas Bekanntes, das uns an die lineare Kryptoanalyse erinnert:

$$\alpha(x) + \beta \circ f(x).$$

Ist $\alpha(x) = \beta \circ f(x)$, so ist der Exponent 0, der Summand also 1. Andernfalls ist der Exponent 1 und der Summand -1. Es werden also $2^n \cdot p_{f,\alpha,\beta}$ Einsen und $2^n - 2^n \cdot p_{f,\alpha,\beta}$ „Minus-Einsen“ aufsummiert, d. h., die Summe ist

$$2^n \cdot [p_{f,\alpha,\beta} - (1 - p_{f,\alpha,\beta})] = 2^n \cdot \tau_{f,\alpha,\beta}.$$

Somit ist $\hat{\vartheta}_f$ bis auf den Normierungsfaktor 2^n die I/O-Korrelation von (α, β).

Die Fourier-Transformierte der Indikatorfunktion einer Booleschen Abbildung $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$, also $\hat{\vartheta_f}\colon \mathbb{F}_2^n \times \mathbb{F}_2^q \longrightarrow \mathbb{R}$, wird oft das (Walsh-) **Spektrum** von f genannt. Wir haben also gezeigt:

Satz 8.2.5. *Für eine Boolesche Abbildung $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ ist das Spektrum genau das 2^n-fache der Korrelationsmatrix.*

Dieser Satz hat große theoretische und praktische Bedeutung:

- Auf der theoretischen Seite führt er zu sehr eleganten und knappen Beweisen von Aussagen über die Korrelationsmatrix und die mit ihr verwandten Objekte [Pom08].
- Auf der praktischen Seite ermöglicht er die Berechnung der Korrelationsmatrix (und damit auch der Approximationstabelle und des lineren Profils) durch die *schnelle Fourier-Transformation*[67], die mithilfe binärer Rekursion den Aufwand (fast) um einen Faktor 2^n drückt.

Wie effizient ist das? Der Einfachheit halber beschränken wir uns auf den wichtigsten Fall $n = q$. Der naive Aufwand erfordert zur Bestimmung von $p_{f,\alpha,\beta}$ (und damit auch von $\tau_{f,\alpha,\beta}$) bei festen α und β das Durchzählen von 2^n Argumenten, wenn die Abbildung durch die Wertetabelle gegeben ist. Der gesamte Aufwand dafür ist also $2^n \cdot 2^n \cdot 2^n$.

Die Erklärung der schnellen Fourier-Transformation würde hier zu weit führen (siehe dazu [Pom08]). Sie steckt in der Funktion `wtr()` aus dem Anhang 8.4.5. Ohne Beweis sei vermerkt, dass die schnelle Fourier-Transformation einer reellwertigen Funktion $\mathbb{F}_2^m \longrightarrow \mathbb{R}$ insgesamt $3m \cdot 2^m$ einfache reelle Operationen erfordert, die man für Funktionen mit Werten in $\{-1, 1\}$ naiv zählen kann, denn es kommen nur ganze Zahlen vor, die nicht allzu groß werden. Das macht hier also $3 \cdot 2n \cdot 2^{2n}$ Operationen.

Eigentlich sollte man den Aufwand eines Algorithmus aber durch die Größe N des Inputs beschreiben. Dieser besteht hier aus der Wertetabelle einer Booleschen Abbildung $\mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$, also ist $N = n \cdot 2^n$ – es müssen n Komponentenfunktionen für je 2^n Argumente definiert werden. So gesehen ist der naive Aufwand (fast) kubisch, der Aufwand des schnellen Algorithmus nur noch (im wesentlichen) quadratisch.

Für den Kryptographen ist allerdings die Blocklänge der relevante Parameter. Unter diesem Gesichtspunkt wächst der Aufwand so oder so mehr oder weniger stark exponenziell. Immerhin ist die Berechnung für „kleine“ S-Boxen, mindestens bis zur Blocklänge 8, algorithmisch sehr effizient.

[67] englisch: Fast Fourier Transformation, abgekürzt FFT

Die Berechnung von Korrelationsmatrix, Approximationstabelle[68] und linearem Profil von S_0 wird im SageMath-Beispiel 8.8 wiedergegeben. (Die Einträge der Ergebnismatrix `Spec` sind durch 16, die von `linProf` durch 256 zu dividieren.)

SageMath-Beispiel 8.8 Korrelationsmatrix, Approximationstabelle und lineares Profil der S-Box S_0

```
sage: Spec = S0.wspec()
sage: ApprT = S0.linApprTable()
sage: linProf = S0.linProf()
```

Wird die Methode `linProf()` mit dem zusätzlichen Parameter `extended=True` aufgerufen, siehe SageMath-Beispiel 8.9, gibt sie auch den maximalen Eintrag aus, zusammen mit allen Index-Paaren, bei denen dieser auftritt. In der Approximationstabelle oder der Korrelationsmatrix kann man dann nachsehen, ob die entsprechende Relation eine Wahrscheinlichkeit größer oder kleiner als $\frac{1}{2}$ hat, ob also bei der Schätzung eines Bits nach Matsui das Komplement zu wählen ist.

SageMath-Beispiel 8.9 Lineares Profil der S-Box S_0 mit Auswertung

```
sage: lProf = S0.linProf(extended=True)
sage: lProf[0]
[...]
sage: print("Maximum entry:", lProf[1], "| with denominator:", lProf[2])
('Maximum entry:', 144, '| with denominator:', 256)
sage: print("at indices:", lProf[3])
('at indices:', [[1, 13], [6, 11], [7, 6]])
sage: Spec = S0.wspec()
sage: for coord in lProf[3]:
....:     if (Spec[coord[0]][coord[1]] < 0):
....:         print ("For relation at", coord, "take complement.")
....:
('For relation at', [6, 11], 'take complement.')
('For relation at', [7, 6], 'take complement.')
```

8.2.9 Beispiel B: Eine Zweirunden-Chiffre

Die Rundenabbildung

$$f\colon \mathbb{F}_2^n \times \mathbb{F}_2^q \longrightarrow \mathbb{F}_2^n$$

einer Bitblock-Chiffre wird jetzt über zwei Runden iteriert mit Rundenschlüsseln $k^{(i)} \in \mathbb{F}_2^q$ wie in Abbildung 8.7 grafisch dargestellt[69].

[68] Zur Berechnung der Approximationstabelle kann man auch die Funktion `S0.linear_approximation_matrix()` der SageMath-Klasse `sage.crypto.mq.sbox.SBox` verwenden, wenn man zuvor `S0 = mq.SBox(12,15,7,10,14,13,11,0,2,6,3,1,9,4,5,8)` definiert. Achtung: siehe Fußnote auf Seite 308.

[69] Im Grunde genommen war das Beispiel A schon eine Zweirunden-Chiffre: Man könnte in Abbildung 8.5 am Ende noch eine weitere S-Box anfügen; diese wäre kryptologisch irrelevant, da sie als nicht-geheimer Teil des Algorithmus der Kryptoanalytikerin bekannt wäre und einfach „abgestreift“ (d. h., ihre Inverse auf den Geheimtext angewendet) werden könnte. Dem entspricht, dass in Beispiel A ja zwei Teilschlüssel verwendet werden. Analog

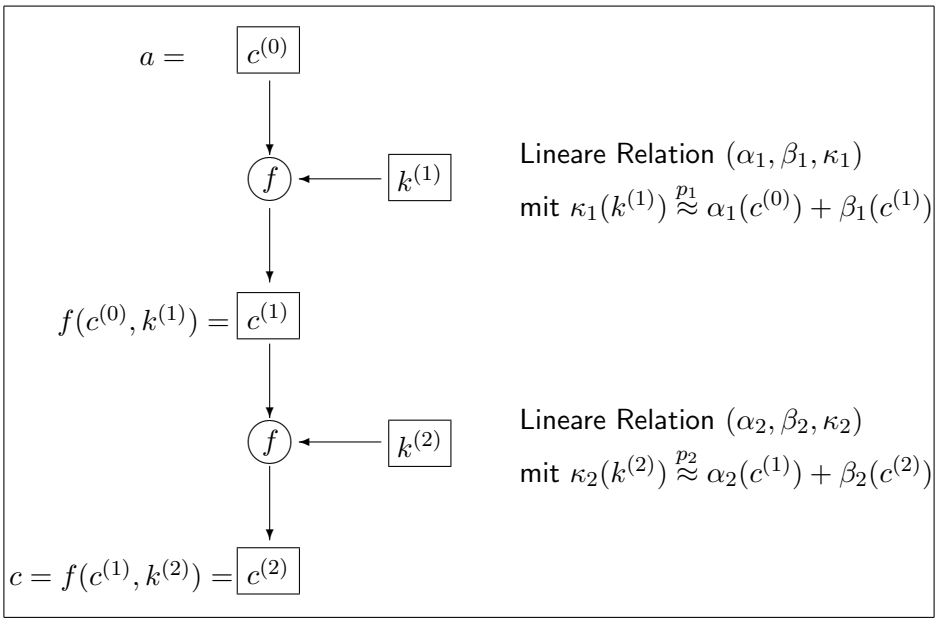


Abbildung 8.7: Allgemeine Zweirunden-Chiffre

Es gelten also die linearen Relationen

$$\kappa_1(k^{(1)}) \overset{p_1}{\approx} \alpha_1(c^{(0)}) + \beta_1(c^{(1)})$$

mit Wahrscheinlichkeit p_1, I/O-Korrelation $\tau_1 = 2p_1 - 1$ und Potenzial $\lambda_1 = \tau_1^2$ und

$$\kappa_2(k^{(2)}) \overset{p_2}{\approx} \alpha_2(c^{(1)}) + \beta_2(c^{(2)})$$

mit Wahrscheinlichkeit p_2, I/O-Korrelation $\tau_2 = 2p_2 - 1$ und Potenzial $\lambda_2 = \tau_2^2$. Die beiden linearen Relationen sind **kombinierbar**, wenn $\alpha_2 = \beta_1$. Dann gilt eine lineare Relation für Schlüsselbits, ausgedrückt durch den (bekannten) Klartext $c^{(0)} = a$ und den Geheimtext $c^{(2)} = c$:

$$\kappa_1(k^{(1)}) + \kappa_2(k^{(2)}) \overset{p}{\approx} \alpha_1(c^{(0)}) + \beta_2(c^{(2)})$$

mit einer gewissen Wahrscheinlichkeit p, einer I/O-Korrelation τ und einem Potenzial λ, die im Allgemeinen von $k = (k^{(1)}, k^{(2)})$ abhängen und nicht leicht zu bestimmen sind. Daher betrachten wir wieder ein vereinfachtes Beispiel, das Beispiel B aus Abbildung 8.8. Die Verschlüsselung geschieht also sukzessive nach den Formeln

$$b^{(0)} = a + k^{(0)},\ a^{(1)} = f_1(b^{(0)}),\ b^{(1)} = a^{(1)} + k^{(1)},\ a^{(2)} = f_2(b^{(1)}),\ c = a^{(2)} + k^{(2)}.$$

(Dabei wird f_1 durch die S-Box S_0 und f_2 durch die S-Box S_1 beschrieben, die auch mit S_0 identisch sein kann[70].) Auch hier verhindert das zusätzliche Aufaddieren von Schlüsselbits nach der letzten Runde, dass diese, also f_2, einfach „abgestreift“ werden kann, wie schon bei Beispiel A.

Im Vergleich zur allgemeinen Situation aus Abschnitt 8.2.6 gilt im Beispiel B:

ist das Beispiel B in diesem Abschnitt auch schon als Dreirunden-Chiffre interpretierbar. Wir schließen uns hier aber der üblichen Rundenzählung an.

[70] D. h., wir lassen hier zu, dass die Rundenfunktionen der verschiedenen Runden unterschiedlich sind. Das hat den Grund, dass in den praktisch wichtigen Chiffren die Rundenfunktion aus mehreren parallelen S-Boxen besteht und wegen der zwischengeschalteten Permutationen ein Input-Bit auf seinem Weg durch die Runden durch verschiedene S-Boxen geleitet werden kann, s. Abschnitt 8.2.12.

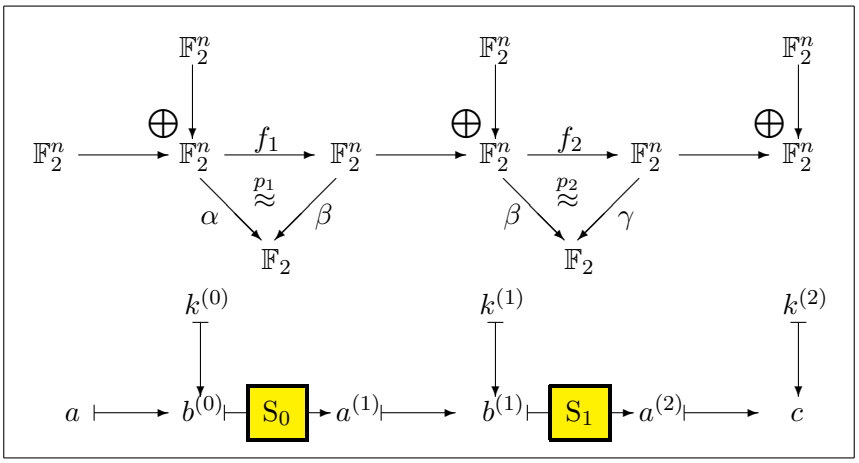


Abbildung 8.8: Beispiel B: Eine Zweirunden-Chiffre

- Die Schlüssellänge ist $l = 3n$, der Schlüsselraum ist $\mathbb{F}_2^{3n}$, und Schlüssel haben die Gestalt $k = (k^{(0)}, k^{(1)}, k^{(2)})$ mit $k^{(0)}, k^{(1)}, k^{(2)} \in \mathbb{F}_2^n$.
- Die Chiffre ist durch die Abbildung

$$F\colon \mathbb{F}_2^n \times \mathbb{F}_2^n \times \mathbb{F}_2^n \times \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n, \quad (a, k^{(0)}, k^{(1)}, k^{(2)}) \mapsto f_2(f_1(a + k^{(0)}) + k^{(1)}) + k^{(2)},$$

  definiert.
- Die Linearform $\kappa\colon \mathbb{F}_2^n \times \mathbb{F}_2^n \times \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ ist durch $\kappa(k^{(0)}, k^{(1)}, k^{(2)}) = \alpha(k^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)})$ gegeben.

Dabei ist (α, β) eine lineare Relation für f_1 mit Wahrscheinlichkeit p_1, I/O-Korrelation τ_1 und Potenzial λ_1 und (β, γ) eine für f_2 mit Wahrscheinlichkeit p_2, I/O-Korrelation τ_2 und Potenzial λ_2 (das gleiche β, d. h., die linearen Relationen sind kombinierbar), und

$$\begin{aligned} p_1 &= \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \beta \circ f_1(x) = \alpha(x)\} \\ p_2 &= \frac{1}{2^n} \cdot \#\{y \in \mathbb{F}_2^n \mid \gamma \circ f_2(y) = \beta(y)\} \end{aligned}$$

Dann gilt mit den Bezeichnungen aus Abbildung 8.8

$$\begin{aligned} \gamma(c) = \gamma(a^{(2)}) + \gamma(k^{(2)}) &\overset{p_2}{\approx} \beta(b^{(1)}) + \gamma(k^{(2)}) = \beta(a^{(1)}) + \beta(k^{(1)}) + \gamma(k^{(2)}) \\ &\overset{p_1}{\approx} \alpha(b^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)}) = \alpha(a) + \alpha(k^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)}) \end{aligned}$$

Wir erhalten also eine lineare Relation für die Schlüsselbits als Spezialfall von Gleichung (8.4) in der Form

$$\alpha(k^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)}) \overset{p}{\approx} \alpha(a) + \gamma(c)$$

mit einer gewissen Wahrscheinlichkeit p, die durch die folgende Formel gegeben ist:

$$\begin{aligned} p &= p_{F,\alpha,\beta,\gamma}(k) \\ &= \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \alpha(k^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)}) = \alpha(a) + \gamma(F(a,k))\}. \end{aligned}$$

Wir versuchen, in diesem vereinfachten Fall p explizit zu bestimmen. Wie im Einrunden-Fall fragen wir zunächst, wie weit p von k abhängt. Wird in der definierenden Gleichung in der Mengenklammer die Definition von $F(a,k)$ eingesetzt, so hebt sich $\gamma(k^{(2)})$ weg, und es bleibt

$$p_{F,\alpha,\beta,\gamma}(k) = \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \alpha(k^{(0)} + a) + \beta(k^{(1)}) = \gamma(f_2(k^{(1)} + f_1(k^{(0)} + a)))\}.$$

Dies ist von $k^{(2)}$ unabhängig und hat für alle $k^{(0)}$ den gleichen Wert

$$p_{F,\alpha,\beta,\gamma}(k) = \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \alpha(x) = \beta(k^{(1)}) + \gamma(f_2(k^{(1)} + f_1(x)))\},$$

denn mit a durchläuft auch $x = k^{(0)} + a$ ganz $\mathbb{F}_2^n$. Dieser Wert hängt also tatsächlich noch von k, aber immerhin nur von der mittleren Komponente $k^{(1)}$ ab. Was geschieht, wenn wir den Mittelwert $\bar{p} := p_{F,\alpha,\beta,\gamma}$ über die möglichen Schlüssel bilden, also

$$\bar{p} = \frac{1}{2^{2n}} \cdot \#\{(x, k^{(1)}) \in \mathbb{F}_2^{2n} \mid \alpha(x) = \beta(k^{(1)}) + \gamma(f_2(k^{(1)} + f_1(x)))\}?$$

In der Mengenklammer tritt der Ausdruck $\gamma(f_2(k^{(1)} + f_1(x)))$ auf. Über diesen wissen wir:

$$\gamma(f_2(k^{(1)} + f_1(x))) = \begin{cases} \beta(k^{(1)} + f_1(x)) & \text{mit Wahrscheinlichkeit } p_2, \\ 1 + \beta(k^{(1)} + f_1(x)) & \text{mit Wahrscheinlichkeit } 1 - p_2. \end{cases}$$

Dabei bedeutet etwa „Wahrscheinlichkeit p_2“, dass die Aussage in $p_2 \cdot 2^{2n}$ von allen möglichen Fällen $(x, k^{(1)}) \in \mathbb{F}_2^{2n}$ gilt. Setzen wir das ein, so erhalten wir

$$\begin{aligned} \bar{p} = \frac{1}{2^{2n}} \cdot \Big[& p_2 \cdot \#\{(x, k^{(1)}) \in \mathbb{F}_2^{2n} \mid \alpha(x) = \beta(f_1(x))\} \\ & + (1 - p_2) \cdot \#\{(x, k^{(1)}) \in \mathbb{F}_2^{2n} \mid \alpha(x) \neq \beta(f_1(x))\} \Big] \end{aligned}$$

wobei jetzt die definierenden Gleichungen beider Mengen auch von $k^{(1)}$ unabhängig sind. Und wir erkennen die Definition von p_1 wieder und setzen sie ein:

$$\bar{p} = p_1 p_2 + (1 - p_1)(1 - p_2) = 2p_1 p_2 - p_1 - p_2 + 1.$$

Eingängiger wird diese Formel, wenn man sie durch die I/O-Korrelationen $\bar{\tau} = 2\bar{p} - 1$ und $\tau_i = 2p_i - 1$ für $i = 1$ und 2 ausdrückt:

$$\bar{\tau} = 2\bar{p} - 1 = 4p_1 p_2 - 2p_1 - 2p_2 + 1 = (2p_1 - 1)(2p_2 - 1) = \tau_1 \tau_2.$$

Zusammengefasst:

Satz 8.2.6. *In der Situation des Beispiels B gilt:*

(i) *Die Wahrscheinlichkeit $p_{F,\alpha,\beta,\gamma}(k)$ hängt nur von der mittleren Komponente $k^{(1)}$ des Schlüssels $k = (k^{(0)}, k^{(1)}, k^{(2)}) \in \mathbb{F}_2^n \times \mathbb{F}_2^n \times \mathbb{F}_2^n$ ab.*

(ii) *Der Mittelwert dieser Wahrscheinlichkeiten über alle Schlüssel k ist $p_{F,\alpha,\beta,\gamma} = \bar{p} = 2p_1 p_2 - p_1 - p_2 + 1$.*

(iii) *Für die I/O-Korrelation und das Potenzial gelten die multiplikativen Formeln*

$$\tau_{F,\alpha,\beta,\gamma} = \tau_1 \tau_2 \quad \textit{und} \quad \lambda_{F,\alpha,\beta,\gamma} = \lambda_1 \lambda_2.$$

Beweis
Das alles ist schon bewiesen. □

Bei der Entscheidung im Matsui-Test, ob die lineare Relation selbst oder ihre Negation zur Schätzung des Bits verwendet werden sollte, greift man dann, da der Schlüssel ja noch unbekannt ist, auf diesen Mittelwert $p_{F,\alpha,\beta,\gamma}$ zurück. Dabei kann man natürlich einen Fehler machen, weil für den konkreten gesuchten Schlüssel k die tatsächliche Wahrscheinlichkeit $p_{F,\alpha,\beta,\gamma}(k)$ auf der anderen Seite von $\frac{1}{2}$ liegen kann als der verwendete Mittelwert.

a	$b^{(0)}$	$a^{(1)}$	$b^{(1)}$	$a^{(2)}$	c	$\beta(b^{(1)})$	$\gamma(a^{(2)})$	$\alpha(a)+\gamma(c)$
0000	1000	0010	0011	1001	1111	1	1	0
0001	1001	0110	0111	0100	0010	0	1	1
0010	1010	0011	0010	1110	1000	0	0	1
0011	1011	0001	0000	0111	0001	0	1	1
0100	1100	1001	1000	1100	1010	1	0	1
0101	1101	0100	0101	1011	1101	0	1	1
0110	1110	0101	0100	0011	0101	1	0	1
0111	1111	1000	1001	1101	1011	0	0	0
1000	0000	1100	1101	1111	1001	1	0	1
1001	0001	1111	1110	1000	1110	0	1	1
1010	0010	0111	0110	0000	0110	1	0	1
1011	0011	1010	1011	1010	1100	0	1	1
1100	0100	1110	1111	0101	0011	1	1	0
1101	0101	1101	1100	0110	0000	0	1	1
1110	0110	1011	1010	0001	0111	1	0	1
1111	0111	0000	0001	0010	0100	1	0	0

Tabelle 8.15: Der Datenfluss für das konkrete Beispiel zu B und einige Linearformen

SageMath-Beispiel 8.10 Eine Boolesche Abbildung (S-Box S_1 von LUCIFER)

```
g1 = BoolF([0,0,1,1,0,1,0,0,1,1,0,1,0,1,1,0])
g2 = BoolF([1,0,1,0,0,0,0,1,1,1,0,0,1,1,0,1])
g3 = BoolF([1,1,1,0,1,1,0,0,0,0,0,1,1,1,0,0])
g4 = BoolF([1,0,0,1,1,1,0,0,0,1,1,0,0,1,0,1])
S1 = BoolMap([g1,g2,g3,g4])
```

Beispiel

Betrachten wir das folgende konkrete Beispiel: Es sei $n = 4$, S_0 sei wie in 8.2.7 gewählt und S_1 so, wie es im SageMath-Beispiel 8.10 definiert[71] ist und wie man es in Tabelle 8.15 in umgeordneter Form als Übergang von $b^{(1)}$ nach $a^{(2)}$ sieht. (Diese Tabelle kann man leicht von Hand berechnen oder mit dem SageMath-Beispiel 8.11.) Die Linearformen $\alpha \mathrel{\hat{=}} \texttt{0001}$ und $\beta \mathrel{\hat{=}} \texttt{1101}$ seien wie in Abschnitt 8.2.7 definiert, also $p_1 = \frac{7}{8}$, $\tau_1 = \frac{3}{4}$, $\lambda_1 = \frac{9}{16}$. Ferner sei $\gamma \mathrel{\hat{=}} \texttt{1100}$, so dass die zum Paar (β, γ) gehörige lineare Relation für f_2 (nach Tabelle 8.16, Zeilenindex 13 und Spaltenindex 12) die Wahrscheinlichkeit $p_2 = \frac{1}{4}$, die I/O-Korrelation $\tau_2 = -\frac{1}{2}$ und das Potenzial $\lambda_2 = \frac{1}{4}$ hat, was nach Tabelle 8.17 der maximal mögliche Wert ist[72].

Die konkreten Rundenschlüssel seien $k^{(0)} = \texttt{1000}$, $k^{(1)} = \texttt{0001}$ – wie schon in Abschnitt 8.2.7 – und $k^{(2)} = \texttt{0110}$. Wir wollen das Bit $\alpha(k^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)})$ finden (dessen Wert 0 wir im „Cheat-Modus“ ja schon kennen). Da $\tau_1\tau_2 < 0$, sollte der dazu verwendete Wert $\alpha(a) + \gamma(c)$ in der Mehrzahl der Fälle das komplementäre Bit 1 ergeben. Die Tabelle 8.15 sagt, dass dies in 12 von 16 Fällen korrekt geschieht. Also ist $1 - p = \frac{3}{4}$, $p = \frac{1}{4}$, $\tau = -\frac{1}{2}$, $\lambda = \frac{1}{4}$. Wir erinnern uns aber, dass dieser Wert von der Schlüsselkomponente $k^{(1)}$ abhängt. Tatsächlich stimmt er nicht

[71] Das ist übrigens die zweite S-Box von Lucifer.
[72] Das lineare Profil von S_1 ist deutlich ausgeglichener als das von S_0.

SageMath-Beispiel 8.11 Beispiel-Berechnungen für das Beispiel B (Zweirunden-Chiffre)

```
sage: n = 4
sage: alpha = [0,0,0,1]; beta = [1,1,0,1]; gamma = [1,1,0,0]
sage: k0 = [1,0,0,0]; k1 = [0,0,0,1]; k2 = [0,1,1,0]
sage: for i in range(0,2**n):
....:     a = int2bbl(i,n); b0 = xor(a,k0); a1 = S0.valueAt(b0)
....:     b1 = xor(k1,a1); a2 = S1.valueAt(b1); c = xor(a2,k2)
....:     bit1 = binScPr(beta,b1); bit2 = binScPr(gamma,a2)
....:     bit3 = (binScPr(alpha,a) + binScPr(gamma,c)) % 2
....:     print(a, b0, a1, b1, a2, c, bit1, bit2, bit3)
```

ganz mit dem Mittelwert

$$\bar{p} = 2 \cdot \frac{7}{8} \cdot \frac{1}{4} - \frac{7}{8} - \frac{1}{4} + 1 = \frac{7}{16} - \frac{14}{16} - \frac{4}{16} + \frac{16}{16} = \frac{5}{16}$$

überein; das zu diesem Mittelwert gehörige Potenzial wäre $\frac{9}{16} \cdot \frac{1}{4} = \frac{9}{64}$.

Die Variation der Wahrscheinlichkeit in Abhängigkeit vom Teilschlüssel $k^{(1)}$ wird mit dem SageMath-Beispiel 8.12 ermittelt. Im Ergebnis erscheinen je 8-mal die Wahrscheinlichkeiten $\frac{1}{4}$ und $\frac{3}{8}$, alle auf der gleichen Seite von $\frac{1}{2}$ und mit dem korrekten Mittelwert $\frac{5}{16}$.

Es gibt auch andere „Pfade“ von α nach γ, man kann ja jedes β dazwischen schieben. Die durchschnittlichen Wahrscheinlichkeiten (die man in SageMath mit einer zusätzlichen Programmschleife ermitteln kann) sind außer der schon bekannten $\frac{5}{16}$ dreimal die $\frac{15}{32}$, elfmal genau $\frac{1}{2}$ und einmal sogar $\frac{17}{32}$, also auf der „falschen“ Seite von $\frac{1}{2}$. Wirklich gut ist also nur der eine optimale Fall, den wir ausführlich behandelt haben.

Als konkretes anderes Beispiel nehmen wir $\beta \mathrel{\hat{=}} \texttt{0001}$. Dafür ist $\lambda_1 = \frac{1}{16}$, $p_1 = \frac{3}{8}$, $\tau_1 = -\frac{1}{4}$ und $\lambda_2 = \frac{1}{16}$, $p_2 = \frac{5}{8}$, $\tau_2 = \frac{1}{4}$. Also ist $\tau = -\frac{1}{16}$ und $p = \frac{15}{32}$. Hier wird versucht, das Bit $\alpha(k^{(0)}) + \beta(k^{(1)}) + \gamma(k^{(2)}) + 1 = 1$ zu finden, und die Erfolgswahrscheinlichkeit dafür ist $1 - p = \frac{17}{32}$. Der Mittelwert von p für dieses β über alle Schlüssel ist $\frac{15}{32}$, also mit dem schlüsselspezifischen Wert in Übereinstimmung.

8.2.10 Lineare Pfade

Betrachten wir nun den allgemeinen Fall, in dem die Rundenabbildung $f\colon \mathbb{F}_2^n \times \mathbb{F}_2^q \longrightarrow \mathbb{F}_2^n$ über mehrere Runden iteriert wird mit Rundenschlüsseln $k^{(i)} \in \mathbb{F}_2^q$ analog zu Abbildung 8.7. Um den begrifflichen Rahmen zu präzisieren, definiert man: Gegeben sei eine über r Runden iterierte Bitblock-Chiffre. Sei $(\alpha_i, \beta_i, \kappa_i)$ eine lineare Relation für die i-te Runde. Es sei $\alpha_i = \beta_{i-1}$ für $i = 2, \ldots, r$. Sei $\beta_0 := \alpha_1$. Dann heißt die Kette $\beta = (\beta_0, \ldots, \beta_r)$ ein **linearer Pfad** für die Chiffre.

Auch hier lässt sich in einem vereinfachten Szenario, nennen wir es Beispiel C – als Verallgemeinerung von Beispiel B –, eine nützliche Aussage über die Wahrscheinlichkeiten herleiten. Wir betrachten also wieder den speziellen, aber praktisch relevanten Fall, in dem die Rundenschlüssel nur additiv ins Verfahren eingebracht werden, siehe Abbildung 8.9.

Für einen Schlüssel $k = (k^{(0)}, \ldots, k^{(r)}) \in \mathbb{F}_2^{n\cdot(r+1)}$ bilden wir die Verschlüsselungsfunktion F sukzessive über die Zwischenergebnisse

$$a^{(0)} = a \mid b^{(0)} = a^{(0)} + k^{(0)} \mid a^{(1)} = f_1(b^{(0)}) \mid b^{(1)} = a^{(1)} + k^{(1)} \mid \ldots$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1	8	10	8	10	8	6	12	10	10	4	6	8	10	8	10	8
2	8	6	4	10	6	8	6	8	8	10	4	6	10	8	10	8
3	8	8	8	8	6	6	6	6	10	6	6	10	4	8	8	12
4	8	8	8	4	8	8	8	4	6	6	6	10	10	10	10	6
5	8	6	8	10	4	6	8	6	8	6	12	6	8	10	8	6
6	8	10	12	10	6	12	6	8	10	8	6	8	8	10	8	6
7	8	8	8	12	10	10	10	6	4	8	8	8	6	10	10	10
8	8	8	6	10	10	6	8	8	10	10	8	12	8	12	6	6
9	8	6	6	8	6	12	8	10	8	6	10	12	10	8	8	10
10	8	6	6	8	12	10	6	8	10	4	8	6	6	8	8	6
11	8	4	10	10	8	8	10	6	8	8	6	10	8	4	6	6
12	8	8	6	6	6	10	12	8	8	8	6	6	6	10	4	8
13	8	10	6	8	6	8	8	10	6	8	8	10	4	6	10	4
14	8	10	6	8	8	10	10	4	12	10	10	8	8	6	10	8
15	8	4	10	6	8	8	10	10	10	10	8	8	6	10	12	8

Tabelle 8.16: Approximationstabelle der S-Box S_1 von LUCIFER – Zeilen- und Spaltenindizes sind durch Zahlen repräsentierte Linearformen, siehe Abschnitt 8.1.11. Die Wahrscheinlichkeiten erhält man nach Division durch 16.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0
2	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0
3	0	0	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	0	$\frac{1}{4}$
4	0	0	0	$\frac{1}{4}$	0	0	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
5	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$
6	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	$\frac{1}{16}$
7	0	0	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
8	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$
9	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$
10	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	$\frac{1}{4}$	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$
11	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$
12	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0
13	0	$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$
14	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0
15	0	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	0

Tabelle 8.17: Lineares Profil der S-Box S_1 von LUCIFER – Zeilen- und Spaltenindizes sind durch Zahlen repräsentierte Linearformen.

SageMath-Beispiel 8.12 Abhängigkeit der Wahrscheinlichkeit vom Schlüssel

```
sage: n = 4; NN = 2**n
sage: alpha = [0,0,0,1]; beta = [1,1,0,1]; gamma = [1,1,0,0]
sage: reslist = []
sage: sum = 0
sage: for j in range(0,NN):
....:     k1 = int2bbl(j,n)
....:     ctr = 0
....:     for i in range(0,NN):
....:         x = int2bbl(i,n)
....:         u = S0.valueAt(x); y = xor(k1,u); z = S1.valueAt(y)
....:         bit1 = binScPr(alpha,x)
....:         bit2 = binScPr(beta,k1); bit3 = binScPr(gamma,z)
....:         if (bit1 == (bit2 + bit3) % 2):
....:             ctr += 1
....:     prob = ctr/NN
....:     reslist.append([k1, ctr, prob])
....:     sum += ctr
....:
sage: reslist
[[[0, 0, 0, 0], 4, 1/4],
 [[0, 0, 0, 1], 4, 1/4],
 [[0, 0, 1, 0], 4, 1/4],
 [[0, 0, 1, 1], 4, 1/4],
 [[0, 1, 0, 0], 6, 3/8],
 [[0, 1, 0, 1], 6, 3/8],
 [[0, 1, 1, 0], 6, 3/8],
 [[0, 1, 1, 1], 6, 3/8],
 [[1, 0, 0, 0], 6, 3/8],
 [[1, 0, 0, 1], 4, 1/4],
 [[1, 0, 1, 0], 4, 1/4],
 [[1, 0, 1, 1], 6, 3/8],
 [[1, 1, 0, 0], 4, 1/4],
 [[1, 1, 0, 1], 6, 3/8],
 [[1, 1, 1, 0], 6, 3/8],
 [[1, 1, 1, 1], 4, 1/4]]
sage: meanprob = sum/(NN*NN)
sage: print("Sum of counters:", sum, "| Mean probability:", meanprob)
('Sum of counters:', 80, '| Mean probability:', 5/16)
```

$$b^{(r-1)} = a^{(r-1)} + k^{(r-1)} \mid a^{(r)} = f_r(b^{(r-1)}) \mid b^{(r)} = a^{(r)} + k^{(r)} = c =: F(a,k)$$

Die allgemeine Formel ist also

$$b^{(i)} = a^{(i)} + k^{(i)} \text{ für } i = 0, \ldots, r,$$

$$a^{(0)} = a \text{ und } a^{(i)} = f_i(b^{(i-1)}) \text{ für } i = 1, \ldots, r.$$

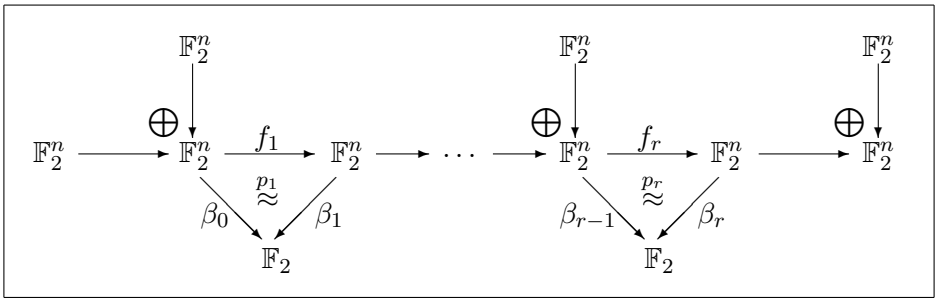


Abbildung 8.9: Beispiel C: Mehrere Runden, Schlüssel kommen additiv in den Algorithmus

Zielobjekt ist die lineare Relation

$$\kappa(k) \overset{p}{\approx} \beta_0(a) + \beta_r(c),$$

wobei

$$\kappa(k) = \beta_0(k^{(0)}) + \cdots + \beta_r(k^{(r)})$$

und p die vom Schlüssel k abhängige Wahrscheinlichkeit

$$p_{F,\beta}(k) = \frac{1}{2^n} \cdot \#\{a \in \mathbb{F}_2^n \mid \sum_{i=0}^{r} \beta_i(k^{(i)}) = \beta_0(a) + \beta_r(F(a,k))\}$$

ist. Den Mittelwert dieser Wahrscheinlichkeiten über alle k bezeichnen wir hier mit q_r; er hängt von $(f_1, \ldots, f_r)$ und vom linearen Pfad $\beta = (\beta_0, \ldots, \beta_r)$ ab:

$$q_r := \frac{1}{2^{n\cdot(r+2)}} \cdot \#\{a, k^{(0)}, \ldots, k^{(r)} \in \mathbb{F}_2^n \mid \sum_{i=0}^{r} \beta_i(k^{(i)}) = \beta_0(a) + \beta_r(F(a,k))\}.$$

Setzen wir in der definierenden Gleichung dieser Menge $F(a,k) = a^{(r)} + k^{(r)} = f_r(b^{(r-1)}) + k^{(r)}$ ein:

$$\beta_0(k^{(0)}) + \cdots + \beta_r(k^{(r)}) = \beta_0(a) + \beta_r(f_r(b^{(r-1)})) + \beta_r(k^{(r)}),$$

so kürzt sich $\beta_r(k^{(r)})$ heraus, und wir sehen, dass die Zählung von $k^{(r)}$ unabhängig ist; es bleibt also

$$q_r = \frac{1}{2^{n\cdot(r+1)}} \cdot \#\{a, k^{(0)}, \ldots, k^{(r-1)} \in \mathbb{F}_2^n \mid \sum_{i=0}^{r-1} \beta_i(k^{(i)}) = \beta_0(a) + \beta_r(f_r(b^{(r-1)}))\}.$$

Hierin steckt die Wahrscheinlichkeit p_r: Es ist

$$\beta_r(f_r(b^{(r-1)})) = \begin{cases} \beta_{r-1}(b^{(r-1)}) & \text{mit Wahrscheinlichkeit } p_r, \\ 1 + \beta_{r-1}(b^{(r-1)}) & \text{mit Wahrscheinlichkeit } 1 - p_r, \end{cases}$$

wobei „mit Wahrscheinlichkeit p_r“ hier bedeutet: in $p_r \cdot 2^{n\cdot(r+1)}$ von allen $2^{n\cdot(r+1)}$ möglichen Fällen. Also folgt

$$\begin{aligned} q_r &= \frac{1}{2^{n\cdot(r+1)}} \cdot \Bigg[p_r \cdot \#\{a, k^{(0)}, \ldots, k^{(r-1)} \mid \sum_{i=0}^{r-1} \beta_i(k^{(i)}) = \beta_0(a) + \beta_{r-1}(b^{(r-1)})\} \\ &\qquad + (1-p_r) \cdot \#\{a, k^{(0)}, \ldots, k^{(r-1)} \mid \sum_{i=0}^{r-1} \beta_i(k^{(i)}) = 1 + \beta_0(a) + \beta_{r-1}(b^{(r-1)})\} \Bigg] \\ &= p_r \cdot q_{r-1} + (1-p_r) \cdot (1 - q_{r-1}), \end{aligned}$$

denn die übrig gebliebenen Mengenzählungen entsprechen genau den analog gebildeten Wahrscheinlichkeiten über die reduzierte Rundenzahl $r-1$.

Damit haben wir den perfekten Ansatz für eine vollständige Induktion, um zu beweisen:

Satz 8.2.7 (Matsuis Piling-Up-Theorem)**.** *In der Situation des Beispiels C gilt für den Mittelwert $p_{F,\beta}$ der Wahrscheinlichkeiten $p_{F,\beta}(k)$ über alle Schlüssel $k \in \mathbb{F}_2^{n(r+1)}$*

$$2p_{F,\beta} - 1 = \prod_{i=1}^{r} (2p_i - 1).$$

Insbesondere multiplizieren sich die I/O-Korrelationen und die Potenziale.

Beweis
Der Induktionsanfang $r = 1$ ist trivial[73].

Aus der Vorbetrachtung folgt

$$2q_r - 1 = 4p_r q_{r-1} - 2p_r - 2q_{r-1} + 1 = (2p_r - 1)(2q_{r-1} - 1),$$

und daraus die Behauptung durch Induktion. □

Bei der Anwendung auf reale Verschlüsselungsverfahren sind im Allgemeinen die Rundenschlüssel nicht unabhängig, sondern werden aus einem „Generalschlüssel“ nach einem Schlüsselauswahlverfahren abgeleitet. In der Praxis macht das aber kaum etwas aus. Die Methode der linearen Kryptoanalyse beruht also auf der Faustregel:

Entlang eines linearen Pfades multiplizieren sich die Potenziale.

Satz 8.2.7, obwohl nur eine spezielle Situation betreffend und in der konkreten Anwendung nicht unbedingt genau, vermittelt doch eine gute Vorstellung davon, wie der kryptoanalytische Nutzen (also das Potenzial) von linearen Approximationen mit jeder Runde weiter abnimmt, d. h., wie die Sicherheit der Chiffre vor linearer Kryptoanalyse mit zunehmender Rundenzahl steigt, denn das Produkt von Zahlen kleiner 1 (und größer 0) wird ja mit zunehmender Länge immer kleiner. Die ungefähre Anzahl N benötigter bekannter Klartexte ergibt sich aus der Formel (8.7) für p_N.

8.2.11 Parallelschaltung von S-Boxen

Für die Rundenabbildung eines SP-Netzes werden in der Praxis mehrere „kleine“ S-Boxen parallel betrieben. Zur Analyse dieser Situation betrachten wir wieder ein einfaches Beispiel, Beispiel D, siehe Abbildung 8.10.

Satz 8.2.8. *Seien $S_1, \ldots, S_m \colon \mathbb{F}_2^q \longrightarrow \mathbb{F}_2^q$ Boolesche Abbildungen, $n = m \cdot q$ und f die Boolesche Abbildung*

$$f \colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n, \quad f(x_1, \ldots, x_m) = (S_1(x_1), \ldots, S_m(x_m)) \text{ für } x_1, \ldots, x_m \in \mathbb{F}_2^q.$$

Sei (α_i, β_i) für $i = 1, \ldots, m$ je eine lineare Relation für S_i mit Wahrscheinlichkeit p_i. Dann ist (α, β) mit

$$\begin{aligned} \alpha(x_1, \ldots, x_m) &= \alpha_1(x_1) + \cdots + \alpha_m(x_m) \\ \beta(y_1, \ldots, y_m) &= \beta_1(y_1) + \cdots + \beta_m(y_m) \end{aligned}$$

[73] Im Satz 8.2.6 steht der Fall $r = 2$, der hier noch einmal mit erledigt wird. Trotzdem war die separate Behandlung nützlich zur Vorbereitung und zur Motivation.

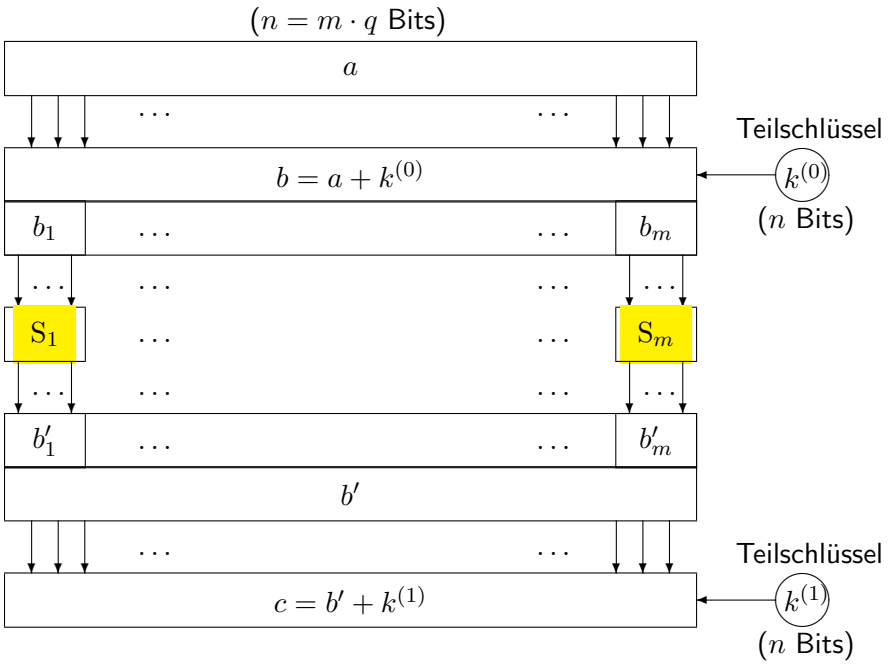


Abbildung 8.10: Beispiel D: Parallelbetrieb von m S-Boxen $S_1, \ldots, S_m$ der Breite q

eine lineare Relation für f mit einer Wahrscheinlichkeit p, die gegeben ist durch

$$2p - 1 = (2p_1 - 1) \cdots (2p_m - 1).$$

Beweis
Wir beschränken uns auf den Fall $m = 2$; der allgemeine Fall folgt durch eine einfache Induktion wie beim Satz 8.2.7.

Im Fall $m = 2$ ist $\beta \circ f(x_1, x_2) = \alpha(x_1, x_2)$ genau dann, wenn

- *entweder* $\beta_1 \circ S_1(x_1) = \alpha_1(x_1)$ und $\beta_2 \circ S_2(x_2) = \alpha_2(x_2)$
- *oder* $\beta_1 \circ S_1(x_1) = 1 + \alpha_1(x_1)$ und $\beta_2 \circ S_2(x_2) = 1 + \alpha_2(x_2)$.

Also ist $p = p_1p_2 + (1 - p_1)(1 - p_2)$, und daraus folgt die Behauptung wie bei Satz 8.2.6. □

Also verhalten sich die I/O-Korrelationen und die Potenziale bei Parallelschaltung ebenfalls multiplikativ. Das sieht auf den ersten Blick aus, als ob dadurch eine Verstärkung der Sicherheit gegeben wäre, dieser Schein trügt aber etwas! Niemand kann die Angreiferin hindern, alle Linearformen außer der „besten“ als Null zu wählen, also mit Wahrscheinlichkeit $p_i = 1$ und Potenzial 1. Sie sucht also das Paar (α_j, β_j) mit dem maximalen Potenzial und wählt $\alpha(x_1, \ldots, x_m) = \alpha_j(x_j)$ und $\beta(y_1, \ldots, y_m) = \beta_j(y_j)$. In gewisser Weise werden die anderen S-Boxen außer S_j also „inaktiv“ gesetzt. Dann hat die gesamte lineare Relation genau die Wahrscheinlichkeit und das Potenzial der „aktiven“ S-Box S_j.

Beispiel

Auch hierzu wieder ein konkretes Rechenbeispiel mit $m = 2$ und $q = 4$, also $n = 8$. Als S-Boxen nehmen wir die beiden von LUCIFER, S_0 links und S_1 rechts (vergleiche Abbildung 8.10). Für

a	a_3	c	$c_0 + c_1 + c_3$	Schätzung
00011110	1	00000010	0	1
00101100	0	00111111	1	1
10110010	1	01011101	0	1
10110100	1	01010000	0	1
10110101	1	01010111	0	1

Tabelle 8.18: Rechenbeispiel zu Beispiel D (Parallelbetrieb von m S-Boxen)

die linke S-Box S_0 nehmen wir wieder die lineare Relation mit $\alpha \mathrel{\hat{=}} \texttt{0001}$ und $\beta \mathrel{\hat{=}} \texttt{1101}$, von der wir wissen, dass sie die Wahrscheinlichkeit $p_1 = \frac{7}{8}$ hat. Für die rechte S-Box S_1 nehmen wir die Relation $(0, 0)$, wo also beide beteiligten Linearformen 0 sind; da $0 = 0$ immer gilt, ist ihre Wahrscheinlichkeit 1. Die zusammengesetzte lineare Relation für $f = (S_0, S_1)$ hat dann nach Satz 8.2.8 ebenfalls die Wahrscheinlichkeit $p = \frac{7}{8}$ und das Potenzial $\lambda = \frac{9}{16}$, und wir wissen, dass die lineare Kryptoanalyse mit $N = 5$ Klartext-Geheimtext-Paaren schon eine (mehr als) 95-prozentige Erfolgswahrscheinlichkeit erreicht. Wir zerlegen alle relevanten Bitblöcke in Bits:

Klartext: $a = (a_0, \ldots, a_7) \in \mathbb{F}_2^8$,

Geheimtext: $c = (c_0, \ldots, c_7) \in \mathbb{F}_2^8$,

Schlüssel: $k = (k_0, \ldots, k_{15}) \in \mathbb{F}_2^{16}$, davon $(k_0, \ldots, k_7)$ als „Eingangsschlüssel“ (entspricht dem $k^{(0)}$ in Abbildung 8.10) und $(k_8, \ldots, k_{15})$ als „Ausgangsschlüssel“ (entspricht $k^{(1)}$).

Dann ist $\alpha(a) = a_3$, $\beta(c) = c_0 + c_1 + c_3$ und $\kappa(k) = \alpha(k_0, \ldots, k_7) + \beta(k_8, \ldots, k_{15}) = k_3 + k_8 + k_9 + k_{11}$. Die anzugreifende Relation ist also

$$k_3 + k_8 + k_9 + k_{11} = a_3 + c_0 + c_1 + c_3.$$

Wir wählen jetzt konkret den Schlüssel $k = \texttt{1001011000101110}$, dessen zu schätzendes Bit also $k_3 + k_8 + k_9 + k_{11} = 1$ ist. Damit erzeugen wir fünf zufällige Klartext-Geheimtext-Paare[74], siehe Tabelle 8.18, und stellen fest, dass in diesem konkreten Fall das fragliche Schlüsselbit durch den Matsui-Algorithmus einstimmig ohne Gegenstimme korrekt geschätzt wird.

8.2.12 Mini-Lucifer

Im nächsten Schritt soll die Überlegung aus dem vorigen Abschnitt über mehrere Runden ausgedehnt werden. Dazu definieren wir eine Spiel-Chiffre unter dem Namen „Mini-Lucifer“, die die S-Boxen und eine Permutation von Lucifer verwendet. Wir bauen sie wie folgt auf:

- Vor und nach jeder Runden-Abbildung wird ein Teilschüssel aufaddiert. Wir verwenden abwechselnd die Schlüssel $k^{(0)}$ und $k^{(1)}$ (d. h., die ersten oder letzten 8 Bits des 16-bittigen Gesamtschlüssels). Insbesondere sind die Rundenschlüssel dann nicht mehr unabhängig.
- Die Rundenfunktion besteht aus der Parallelschaltung der beiden S-Boxen wie im Beispiel des Abschnitts 8.2.11, gefolgt von der Permutation P.

[74] D. h., wir simulieren eine Angreiferin, die fünf Klartext-Geheimtextpaare aufgefangen hat, indem wir (zufällig, willkürlich) fünf solche erzeugen und verwenden. Damit haben wir eine gute Chance, das gesuchte Schlüsselbit richtig zu bestimmen.

- Die Permutation P permutiert ein Byte (Oktett) in sich und ist im SageMath-Beispiel 8.13 definiert. Sie wird in der letzten Runde, wie bei SP-Netzen üblich, weggelassen.

Das gesamte Schema ist in Abbildung 8.11 veranschaulicht, das SageMath-Beipiel 8.14 enthält den SageMath/Python-Code dafür.

SageMath-Beispiel 8.13 Die Bit-Permutation P von LUCIFER

```
def P(b):
  """Lucifer's permutation"""
  pb = [b[2],b[5],b[4],b[0],b[3],b[1],b[7],b[6]]
  return pb
```

SageMath-Beispiel 8.14 Mini-Lucifer über r Runden

```
def miniLuc(a,k,r):
  """Mini-Lucifer, encrypts 8-bit a with 16-bit key k over r rounds."""
  k0 = k[0:8]           # split into subkeys
  k1 = k[8:16]
  aa = a                # round input
  # --- begin round
  for i in range(0,r): # round number is i+1
    if (i % 2 == 0):   # select round key
      rndkey = k0
    else:
      rndkey = k1
    b = xor(aa,rndkey)      # add round key
    bleft = b[0:4]          # begin substitution
    bright = b[4:8]
    bbleft = S0.valueAt(bleft)
    bbright = S1.valueAt(bright)
    bb = bbleft + bbright  # end substitution
    if (i+1 == r):          # omit permutation in last round
      aa = bb
    else:
      aa = P(bb)
  # --- end round
  if (r % 2 == 0):          # add subkey after last round
    finkey = k0
  else:
    finkey = k1
  c = xor(aa,finkey)
  return c
```

Bisher hatten wir bei der linearen Kryptoanalyse nicht mit Permutationen zu tun. Können diese das Vorgehen beeinträchtigen?

Nun, sei f eine Boolesche Abbildung, (α, β) eine lineare Relation für f mit Wahrscheinlichkeit p und P eine Permutation im Bildbereich von f. Dann setzen wir $\beta' = \beta \circ \mathrm{P}^{-1}$ und sehen

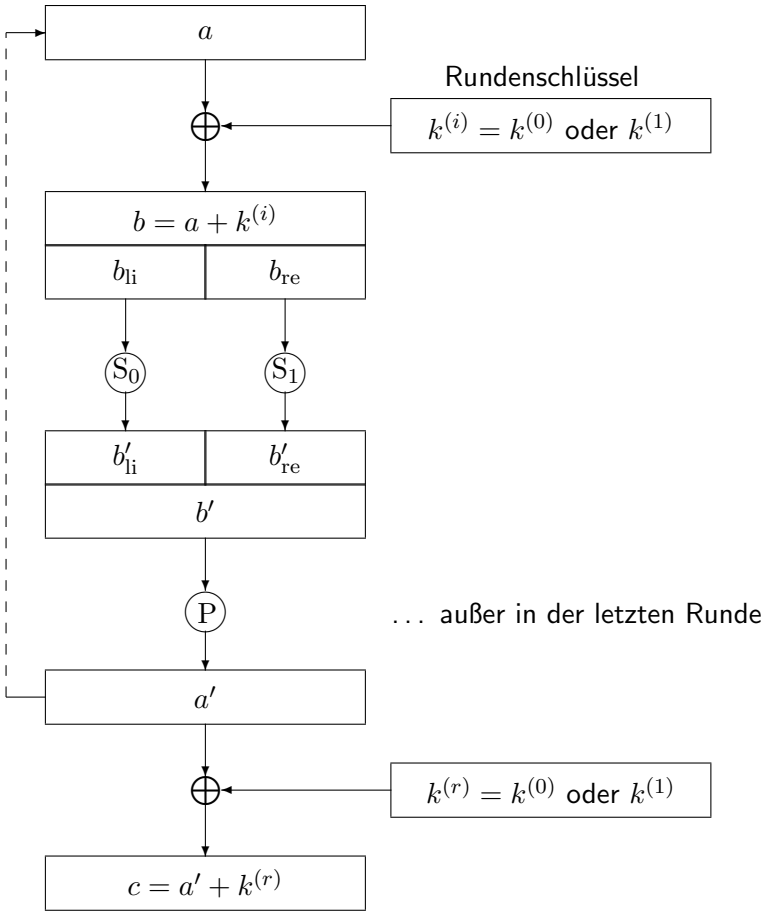


Abbildung 8.11: Mini-Lucifer über r Runden

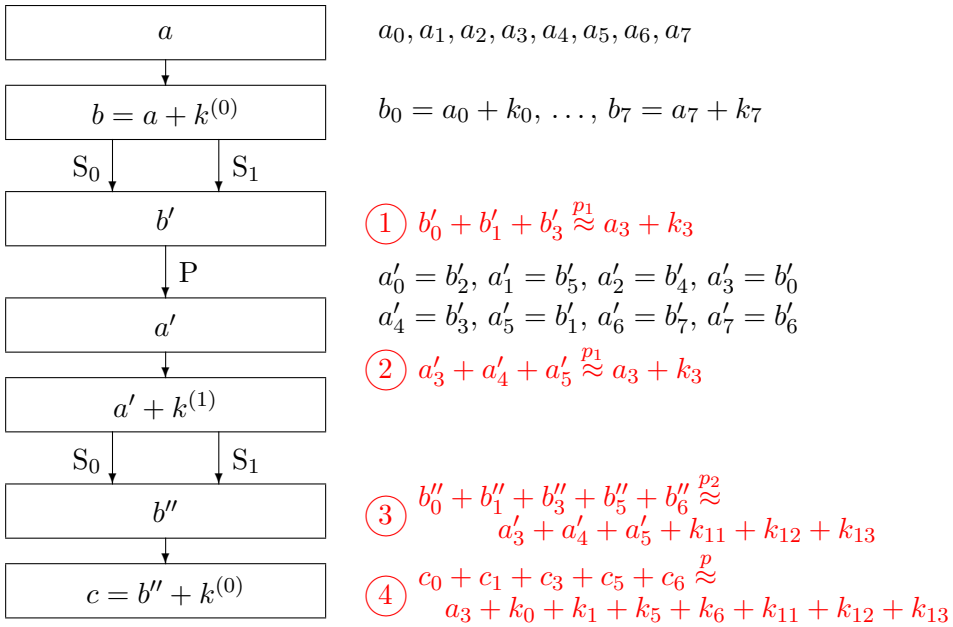


Abbildung 8.12: Mini-Lucifer mit 2 Runden

sofort, dass (β', α) eine lineare Relation für $\mathrm{P} \circ f$ mit der gleichen Wahrscheinlichkeit p ist:

$$\begin{aligned} p &= \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid \beta(f(x)) = \alpha(x)\} \\ &= \frac{1}{2^n} \cdot \#\{x \in \mathbb{F}_2^n \mid (\beta \circ \mathrm{P}^{-1})(\mathrm{P} \circ f(x)) = \alpha(x)\}. \end{aligned}$$

Die Zuordnung $\beta \mapsto \beta'$ ist einfach eine Permutation der Linearformen β. D. h., durch das Anfügen einer Permutation werden in der Approximationstabelle und im linearen Profil von f einfach nur die Spalten permutiert[75].

> *Das Einschieben der Permutationen in die Rundenfunktion eines SP-Netzes beeinträchtigt die lineare Kryptoanalyse nicht wesentlich.*

Wir werden das gleich am konkreten Beispiel nachvollziehen und dabei auch sehen, was „nicht wesentlich“ bedeutet.

Beispiel

Das konkrete Beispiel (als Fortsetzung des Beispiels in 8.2.11) ist in Abbildung 8.12 beschrieben. Die Relation 1, nämlich

$$\beta(b') \overset{p_1}{\approx} \alpha(a + k^{(0)}) \quad \text{oder explizit} \quad b'_0 + b'_1 + b'_3 \overset{p_1}{\approx} a_3 + k_3$$

besteht zwischen $\alpha \mathrel{\hat{=}} \texttt{0001}$ und $\beta \mathrel{\hat{=}} \texttt{1101}$ mit Wahrscheinlichkeit $p_1 = \frac{7}{8}$. Die Permutation P macht daraus die Relation 2, nämlich

$$\beta \circ \mathrm{P}^{-1}(a') \overset{p_1}{\approx} \alpha(a + k^{(0)}) = \alpha(a) + \alpha(k^{(0)}), \quad \text{explizit} \quad a'_3 + a'_4 + a'_5 \overset{p_1}{\approx} a_3 + k_3.$$

[75] Das gilt übrigens genauso, wenn man statt einer Permutation allgemeiner eine bijektive lineare Abbildung anfügt.

Sie verteilt aber auch die Bits auf der linken Seite der Relation auf die beiden S-Boxen der nächsten Runde. Der kryptoanalytische Trick, pro Runde nur eine S-Box aktiv werden zu lassen, beschränkt sich also im Wesentlichen auf die erste Runde.

> *Das Einschieben der Permutationen in die Rundenfunktion eines SP-Netzes sorgt dafür, dass bei der linearen Kryptoanalyse in späteren Runden mehrere parallele S-Boxen aktiv sind.*

Wie wir gleich im Beispiel sehen werden, wird das Potenzial dadurch verringert. Die relevanten Bits a'_3, a'_4, a'_5, bzw. nach Schlüsseladdition a'_3+k_{11}, a'_4+k_{12}, a'_5+k_{13}, werden als Input auf die linke S-Box S_0 der zweiten Runde (nämlich a'_3+k_{11}) und auf die rechte, S_1, (nämlich a'_4+k_{12} und a'_5+k_{13}) aufgeteilt. Auf der linken Seite, für S_0, ist die Linearform für den Input $\beta'_1 \mathrel{\hat{=}} \texttt{0001} \mathrel{\hat{=}} 1$, auf der rechten Seite, für S_1, müssen wir $\beta'_2 \mathrel{\hat{=}} \texttt{1100} \mathrel{\hat{=}} 12$ setzen. Aus dem linearen Profil von S_0 sehen wir, dass wir bei β'_1 das Potenzial $\lambda'_2 = \frac{9}{16}$ mit $p'_2 = \frac{7}{8}$ für $\gamma_1 \mathrel{\hat{=}} 13 \mathrel{\hat{=}} \texttt{1101}$ erreichen können. Bei β'_2 erreichen wir maximal das Potenzial $\lambda''_2 = \frac{1}{4}$. Dafür gibt es zwei Möglichkeiten; wir wählen etwa $\gamma_2 \mathrel{\hat{=}} 6 \mathrel{\hat{=}} \texttt{0110}$ mit Wahrscheinlichkeit $p''_2 = \frac{3}{4}$. Die kombinierte lineare Relation mit $\beta'(x) = \beta'_1(x_0,\ldots,x_3) + \beta'_2(x_4,\ldots,x_7)$ und auf der Output-Seite $\gamma(y) = \gamma_1(y_0,\ldots,y_3) + \gamma_2(y_4,\ldots,y_7)$ hat dann nach Satz 8.2.8 die I/O-Korrelation $2p_2 - 1 = (2p'_2 - 1)(2p''_2 - 1) = \frac{3}{8}$, also $p_2 = \frac{11}{16}$, $\lambda_2 = \frac{9}{64}$.

Die Relation zwischen $\beta'(a' + k^{(1)})$ und $\gamma(b'')$ ist die in Abbildung 8.12 mit 3 markierte und explizit ausgeschriebene, nämlich

$$\gamma(b'') \overset{p_2}{\approx} \beta'(a' + k^{(1)}) = \beta'(a') + \beta'(k^{(1)}).$$

Die Kombination von 2 und 3 ergibt (nach Kürzung von k_3) die Relation

$$\gamma(c) + \gamma(k^{(0)}) = \gamma(c + k^{(0)}) = \gamma(b'') \overset{p}{\approx} \alpha(a) + \alpha(k^{(0)}) + \beta'(k^{(1)}),$$

in der Abbildung mit 4 markiert und explizit ausgeschrieben, deren Wahrscheinlichkeit p nach Satz 8.2.7 bestimmt werden kann, da die beiden verwendeten Rundenschlüssel unabhängig sind. Es ergibt sich $2p - 1 = (2p_1 - 1)(2p_2 - 1) = \frac{3}{4} \cdot \frac{3}{8} = \frac{9}{32}$, also $p = \frac{41}{64}$. Das zugehörige Potenzial ist $\lambda = \frac{81}{1024}$.

Die Anzahl N der für 95-prozentige Erfolgswahrscheinlichkeit benötigten Klartexte erhalten wir nach der Näherungsformel aus Tabelle 8.11 als

$$N = \frac{3}{\lambda} = \frac{1024}{27} \approx 38$$

(von 256 überhaupt möglichen).

Im Beispiel ergab sich die Erfolgswahrscheinlichkeit durch Multiplikation der I/O-Korrelationen (oder der Potenziale) aller aktiven S-Boxen. Wir hatten das Glück, dass hier die jeweils verwendeten Teilschlüssel unabhängig waren. Im Allgemeinen wird das nicht der Fall sein. Dass man mit dieser Unabhängigkeitsannahme trotzdem arbeiten kann, ist nur empirisch belegt, und daraus folgt die Faustregel:

> *Die Erfolgswahrscheinlichkeit der linearen Kryptoanalyse bestimmt sich (ungefähr) durch die Multiplikativität der I/O-Korrelationen (oder der Potenziale) aller entlang des betrachteten Pfades (samt seiner Verzweigungen) aktiven S-Boxen.*

Die Einschränkung in dieser Faustregel betrifft aber nicht das *Vorgehen* bei der linearen Kryptoanalyse, sondern nur die *Erfolgswahrscheinlichkeit*. Die Kryptoanalytikerin hat genau dann

Recht, wenn sie Erfolg hat, egal ob ihre Methode in allen Details mathematisch exakt begründet war oder nicht.

Wir haben jetzt ein Bit bestimmt. Und was nun? Wir finden natürlich weitere Relationen und können damit weitere Schlüsselbits aufdecken, aber wir müssen dabei immer geringere Potenziale zulassen und laufen auch zunehmend in die Gefahr, dass die Wahrscheinlichkeit für den konkreten (gesuchten) Schlüssel auf der „falschen“ Seite von $\frac{1}{2}$ liegt. Außerdem werden natürlich die Erfolgswahrscheinlichkeiten durch Multiplikation immer kleiner. Und drittens laufen wir zunehmend in das Problem des multiplen Testens, wenn wir immer wieder die gleichen bekannten Klartexte verwenden, was eine weitere Korrektur der Erfolgswahrscheinlichkeit nach unten bewirkt.

Die systematische Suche nach linearen Relationen

Das Finden nutzbarer linearer Relationen über mehrere Runden ist algorithmisch im Allgemeinen nicht elegant lösbar; in den publizierten Beispielen werden oft mehr oder weniger zufällig gefundene lineare Pfade verwendet, ohne dass klar ist, ob es nicht wesentlich besser geeignete gibt.

Ist n die Blocklänge der Chiffre und r die Zahl der Runden, so hat man in jeder Runde 2^n Linearformen zur Auswahl, insgesamt also $2^{n(r+1)}$. Diesen Aufwand muss man treiben, wenn man gute Relationen durch vollständige Suche bestimmen will. Es gibt Vereinfachungen, die aber die Größenordnung des Gesamtaufwands nicht wesentlich verringern:

- Man kann sich in der ersten Runde auf Linearformen beschränken, die nur eine S-Box aktivieren.
- Und dann kann man die nächste Linearform so wählen, dass sie möglichst wenige weitere S-Boxen aktiviert (bei hohem, wenn auch vielleicht nicht maximalem Potenzial).
- Hat eine der Relationen in einem linearen Pfad die Wahrscheinlichkeit $\frac{1}{2}$, also die I/O-Korrelation 0, so ist die Gesamtkorrelation wegen der Multiplikativität ebenfalls 0, der Pfad kann also ignoriert werden. Das gilt auch komponentenweise, wenn die betrachteten Linearformen auf die einzelnen S-Boxen der Runde aufgesplittert werden. Allerdings können in dieser Vernachlässigung auch Schwächen liegen, da wir ja immer mit den durchschnittlichen Wahrscheinlichkeiten rechnen, aber eigentlich die schlüsselabhängigen bräuchten.

Für unser 2-Runden-Beispiel mit Mini-Lucifer ist die systematische Suche noch leicht durchführbar; selbstverständlich kann man sich dabei auch mit SageMath (oder einem Python-Programm) behelfen, das soll hier aber nicht ausgewalzt werden. Das obige Beispiel hatte folgende Kenngrößen:

- $\alpha = (\alpha_1, \alpha_2)$[76] mit $\alpha_1 \mathrel{\hat{=}} 1 \mathrel{\hat{=}} \texttt{0001}$ und $\alpha_2 \mathrel{\hat{=}} 0 \mathrel{\hat{=}} \texttt{0000}$
- $\beta = (\beta_1, \beta_2)$ mit $\beta_1 \mathrel{\hat{=}} 13 \mathrel{\hat{=}} \texttt{1101}$ und $\beta_2 \mathrel{\hat{=}} 0 \mathrel{\hat{=}} \texttt{0000}$
- $\beta' = (\beta'_1, \beta'_2)$ mit $\beta'_1 \mathrel{\hat{=}} 1 \mathrel{\hat{=}} \texttt{0001}$, $\beta'_2 \mathrel{\hat{=}} 12 \mathrel{\hat{=}} \texttt{1100}$
- $\gamma = (\gamma_1, \gamma_2)$ mit $\gamma_1 \mathrel{\hat{=}} 13 \mathrel{\hat{=}} \texttt{1101}$, $\gamma_2 \mathrel{\hat{=}} 6 \mathrel{\hat{=}} \texttt{0110}$

[76] Oben war α_1 als α bezeichnet worden. Hier werden jetzt der Einheitlichkeit halber bei allen Linearformen beide Komponenten aufgeführt und mit 1 und 2 indiziert.

- $\tau_1 = \frac{3}{4}$, $\tau_2' = \frac{3}{4}$, $\tau_2'' = \frac{1}{2}$, $\tau_2 = \frac{3}{8}$, $\tau = \frac{9}{32}$, $p = \frac{41}{64} = 0,640625$
- $c_0 + c_1 + c_3 + c_5 + c_6 \overset{p}{\approx} a_3 + k_0 + k_1 + k_5 + k_6 + k_{11} + k_{12} + k_{13}$

Für γ_2 hätten wir noch die Möglichkeit $\gamma_2 \mathrel{\hat{=}} 14 \mathrel{\hat{=}}$ `1110` gehabt; das ergibt einen linearen Pfad mit den Kenngrößen

- $\alpha \mathrel{\hat{=}} (1,0)$, $\beta \mathrel{\hat{=}} (13,0)$, $\beta' \mathrel{\hat{=}} (1,12)$, $\gamma \mathrel{\hat{=}} (13,14)$
  - wobei $\tau = -\frac{9}{32}$, $p = \frac{23}{64} = 0,359375$
  - $c_0 + c_1 + c_3 + c_4 + c_5 + c_6 \overset{p}{\approx} a_3 + k_0 + k_1 + k_4 + k_5 + k_6 + k_{11} + k_{12} + k_{13}$

Die systematische Suche ergibt zwei noch „bessere“ lineare Pfade, charakterisiert durch

- $\alpha \mathrel{\hat{=}} (8,0)$, $\beta \mathrel{\hat{=}} (8,0)$, $\beta' \mathrel{\hat{=}} (1,0)$, $\gamma \mathrel{\hat{=}} (13,0)$
  - wobei $\tau = -\frac{3}{8}$, $p = \frac{5}{16} = 0,3125$
  - $c_0 + c_1 + c_3 \overset{p}{\approx} a_0 + k_1 + k_3 + k_{11}$
- $\alpha \mathrel{\hat{=}} (15,0)$, $\beta \mathrel{\hat{=}} (8,0)$, $\beta' \mathrel{\hat{=}} (1,0)$, $\gamma \mathrel{\hat{=}} (13,0)$
  - wobei $\tau = -\frac{3}{8}$, $p = \frac{5}{16} = 0,3125$
  - $c_0 + c_1 + c_3 \overset{p}{\approx} a_0 + a_1 + a_2 + a_3 + k_2 + k_{11}$

die zwar das Potenzial der einzelnen S-Boxen nicht voll ausschöpfen, aber dafür jeweils nur eine S-Box der zweiten Runde aktivieren und damit insgesamt das höhere Potenzial $\lambda = \frac{9}{64}$ erreichen. Dieses führt zu einer 95-prozentigen Erfolgswahrscheinlichkeit schon mit

$$N = \frac{3}{\lambda} = \frac{64}{3} \approx 21$$

bekannten Klartexten zur Bestimmung eines Bits.

Umgekehrt wird der Designer einer Chiffre darauf achten, dass die Permutation in jeder Runde die aktiven Bits auf möglichst viele S-Boxen verteilt; die Erfinder von AES, Daemen[77] und Rijmen[78], nennen diesen Konstruktionsansatz die „Wide-Trail“-Strategie[79]. Abbildung 8.13 zeigt ein Beispiel eines solchen verzweigten linearen Pfades.

Beispiel (Fortsetzung)

Zur Illustration des konkreten Vorgehens erzeugen wir 25 Paare von bekanntem Klartext und zugehörigem Geheimtext mit dem Schlüssel $k \mathrel{\hat{=}}$ `1001011000101110`. Das geschieht im ersten Teil des SageMath-Beispiels 8.16. Die dabei verwendete Funktion `randsel()` steht im SageMath-Beispiel 8.15. Sie liefert `NN` verschiedene ganze Zahlen im Intervall $[0, 255]$. Die Klartext-Geheimtext-Paare eines Beispiel-Laufes sind im SageMath-Beispiel 8.17 wiedergegeben.

[77] Joan Daemen, belgischer Kryptologe, Miterfinder des AES-Verfahrens, *1965
[78] Vincent Rijmen, belgischer Kryptologe, Miterfinder des AES-Verfahrens, *1970
[79] Dieser Effekt wird bei AES noch dadurch verstärkt, dass das „P“ des SP-Netzes zu einem „L“ verallgemeinert wird, nämlich zu einer linearen Abbildung.

SageMath-Beispiel 8.15 Erzeugung *verschiedener* Zufallszahlen

```
def randsel(max,NN):
  """Generates NN different random integers between 0 and max."""
  rndlist = []
  while (len(rndlist) < NN):
    new = randint(0,max)
    if (not(new in rndlist)):
      rndlist.append(new)
  rndlist.sort()
  return rndlist
```

SageMath-Beispiel 8.16 Lineare Kryptoanalyse von Mini-Lucifer über 2 Runden

```
sage: key = str2bbl("1001011000101110")
sage: bit = [0,0,0,0]
sage: bit[0] = (key[0]+key[1]+key[5]+key[6]+key[11]+key[12]+key[13]) % 2
sage: bit[1] = (key[0]+key[1]+key[4]+key[5]+key[6]+key[11]+key[12]+key[13])%2
sage: bit[2] = (key[1]+key[3]+key[11]) % 2
sage: bit[3] = (key[2]+key[11]) % 2
sage: NN = 25
sage: plist = randsel(255,NN)
sage: klist = [[], [], [], []]
sage: for i in range (0,NN):
....:     plain = int2bbl(plist[i],8)
....:     ciph = miniLuc(plain,key,2)
....:     print("pc pair nr", i+1, "is", plain, ciph)
....:     kbit = (plain[3]+ciph[0]+ciph[1]+ciph[3]+ciph[5]+ciph[6]) % 2
....:     klist[0].append(kbit)
....:     kbit = (1+plain[3]+ciph[0]+ciph[1]+ciph[3]+ciph[4]+ciph[5]+ciph[6])%2
....:     klist[1].append(kbit)
....:     kbit = (1+plain[0]+ciph[0]+ciph[1]+ciph[3]) % 2
....:     klist[2].append(kbit)
....:     kbit=(1+plain[0]+plain[1]+plain[2]+plain[3]+ciph[0]+ciph[1]+ciph[3])%2
....:     klist[3].append(kbit)
....:
[...]
sage: for j in range(0,4):
....:     sum = 0
....:     for jj in range(0,NN):
....:         sum += klist[j][jj]
....:     if (bit[j] == 0):
....:         sum = NN - sum
....:     print("True bit:", bit[j], klist[j])
....:     print("    Relation", j+1, ":", sum, "of", NN ,"guesses are correct.")
....:
[...]
```

Die zu bestimmenden Schlüssel-Bits, die wir im Cheat-Modus kennen, sind `bit[j]` für $j = 0, 1, 2, 3$ – wir verwenden alle vier oben identifizierten Relationen gleichzeitig ohne Furcht vor der eventuell verringerten Erfolgswahrscheinlichkeit. Diese Relationen behaupten die wahrscheinliche Gleichheit der Bits `bit[j]` mit den in den Zeilen `kbit` angegebenen Summen von

SageMath-Beispiel 8.17 25 Klartext-Geheimtext-Paare von Mini-Lucifer über 2 Runden

```
pc pair nr  1 is [0,0,0,0,1,1,1,1] [0,0,0,0,1,0,1,0]
pc pair nr  2 is [0,0,0,1,0,0,0,1] [1,1,0,0,1,1,1,0]
pc pair nr  3 is [0,0,0,1,0,1,1,0] [1,1,0,0,1,0,0,1]
pc pair nr  4 is [0,0,1,1,1,1,0,1] [1,0,1,1,0,0,1,0]
pc pair nr  5 is [0,1,0,0,0,0,0,0] [1,1,1,0,0,1,1,1]
pc pair nr  6 is [0,1,0,0,1,0,0,0] [0,1,0,1,0,1,1,1]
pc pair nr  7 is [0,1,0,0,1,1,0,0] [1,1,1,0,1,0,1,0]
pc pair nr  8 is [0,1,0,0,1,1,0,1] [0,1,0,1,1,1,0,0]
pc pair nr  9 is [0,1,0,0,1,1,1,1] [0,1,1,1,1,0,1,0]
pc pair nr 10 is [0,1,1,0,0,1,1,1] [0,0,1,1,0,0,1,1]
pc pair nr 11 is [1,0,0,0,0,0,1,1] [1,1,1,1,0,1,0,0]
pc pair nr 12 is [1,0,0,1,0,0,1,1] [0,1,1,0,1,0,1,1]
pc pair nr 13 is [1,0,0,1,1,0,0,0] [0,1,1,0,0,1,1,1]
pc pair nr 14 is [1,0,1,0,1,0,1,1] [1,1,0,1,1,0,0,1]
pc pair nr 15 is [1,0,1,1,0,0,0,1] [1,1,0,0,1,0,0,0]
pc pair nr 16 is [1,0,1,1,0,0,1,0] [1,0,1,0,0,1,0,0]
pc pair nr 17 is [1,0,1,1,0,1,1,0] [1,1,0,0,0,1,0,0]
pc pair nr 18 is [1,0,1,1,1,0,0,1] [1,1,0,0,0,0,0,1]
pc pair nr 19 is [1,0,1,1,1,1,0,1] [1,0,1,1,1,1,1,1]
pc pair nr 20 is [1,1,0,0,0,1,0,0] [0,1,0,0,1,1,1,1]
pc pair nr 21 is [1,1,0,0,0,1,1,1] [0,0,1,1,1,1,1,1]
pc pair nr 22 is [1,1,0,1,1,1,1,1] [1,1,0,1,1,0,1,0]
pc pair nr 23 is [1,1,1,0,0,0,0,0] [1,1,1,0,1,1,1,0]
pc pair nr 24 is [1,1,1,0,0,1,0,0] [0,1,1,1,0,0,1,1]
pc pair nr 25 is [1,1,1,1,0,1,0,1] [1,1,1,1,0,1,0,1]
```

Klartext- und Geheimtextbits. Die letzten drei dieser Summen sind zu komplementieren, da die entsprechende I/O-Korrelation negativ (bzw. die Wahrscheinlichkeit $< \frac{1}{2}$) ist; das geschieht durch die zusätzliche Addition des Bits `1`.

Das Ergebnis der Analyse steht im SageMath-Beispiel 8.18. Wir sehen, dass wir alle vier Bits richtig geschätzt haben[80].

SageMath-Beispiel 8.18 Lineare Kryptoanalyse von Mini-Lucifer über 2 Runden

```
True bit: 1 [1,1,1,0,0,0,1,1,1,0,0,1,0,1,1,1,0,1,1,1,1,1,0,1,1]
    Relation 1 : 17 of 25 guesses are correct.
True bit: 1 [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,1,0,0,0]
    Relation 2 : 20 of 25 guesses are correct.
True bit: 1 [1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,1,0,0,0,1,1,1,0,0,1]
    Relation 3 : 18 of 25 guesses are correct.
True bit: 0 [1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,1,0,0,0]
    Relation 4 : 20 of 25 guesses are correct.
```

[80] Wir haben die Funktion `Matsui_Test()` nicht verwendet, weil wir etwas Einblick in die Zwischenergebnisse haben wollten.

Als Folge dieser Analyse haben wir ein lineares Gleichungssystem aus vier Gleichungen für die 16 unbekannten Schlüsselbits:

$$\begin{aligned} 1 &= k_0 + k_1 + k_5 + k_6 + k_{11} + k_{12} + k_{13} \\ 1 &= k_0 + k_1 + k_4 + k_5 + k_6 + k_{11} + k_{12} + k_{13} \\ 1 &= k_1 + k_3 + k_{11} \\ 0 &= k_2 + k_{11} \end{aligned}$$

wodurch die Anzahl der bei einer Exhaustion zu probierenden Schlüssel von $2^{16} = 65536$ auf $2^{12} = 4096$ reduziert wird. Zwei direkte Vereinfachungen sind $k_{11} = k_2$ aus der letzten Gleichung und $k_4 = 0$ aus den beiden ersten. Vier weitere Simulationsläufe ergeben

- $15, 16, 19, 16$
- $15, 16, 13, 17$
- $15, 20, 19, 17$
- $19, 19, 20, 18$

korrekte Schätzungen, also stets korrekte Ergebnisse. In unserer Simulation ergab erst der zehnte Lauf ein falsches Bit (das zweite):

- $17, 12, 14, 17$

danach erst wieder der 25. Lauf. Der empirische Eindruck ist also, dass die Erfolgswahrscheinlichkeit in dieser Situation über 90% liegt.

Analyse über vier Runden

Wir wollen uns nun vergewissern, wie eine Erhöhung der Rundenzahl die lineare Kryptoanalyse entscheidend behindert.

Dazu betrachten wir unsere Spiel-Chiffre Mini-Lucifer über vier Runden. Die Suche nach einem optimalen linearen Pfad über vier Runden ist schon etwas aufwendig, wir begnügen uns daher exemplarisch mit der Ausdehnung des besten Beispiels für zwei Runden, nämlich des obigen dritten, über weitere zwei Runden. Mit einer auf die jetzige Situation angepassten Bezeichnung haben wir:

- für die erste Runde $\beta_0 = \alpha \mathrel{\hat{=}} (8, 0)$ und $\beta_1 \mathrel{\hat{=}} (8, 0)$ (das „alte“ β) mit $\tau_1 = -\frac{1}{2}$,
- für die zweite Runde (die Permutation P auf β_1 angewendet) $\beta_1' \mathrel{\hat{=}} (1, 0)$ und $\beta_2 \mathrel{\hat{=}} (13, 0)$ (das „alte“ γ) mit $\tau_2 = \frac{3}{4}$,
- für die dritte Runde $\beta_2' \mathrel{\hat{=}} (1, 12)$ und $\beta_3 \mathrel{\hat{=}} (13, 6)$ mit $\tau_3 = \frac{3}{8}$,
- für die vierte Runde $\beta_3' \mathrel{\hat{=}} (5, 13)$ und $\beta = \beta_4 \mathrel{\hat{=}} (3, 12)$ (das „neue“ β) mit $\tau_4 = -\frac{1}{4}$.

Dieser lineare Pfad mit seinen Verzweigungen ist in Abbildung 8.13 skizziert.

Da wir Rundenschlüssel wiederholt haben, sind diese nicht unabhängig. Die Multiplikativität der I/O-Korrelationen ist also nur durch die Faustregel begründet. Sie ergibt daher für die I/O-Korrelation der linearen Relation (α, β) über die gesamten vier Runden nur einen ungefähren Wert

$$\tau \approx \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{3}{8} \cdot \frac{1}{4} = \frac{9}{256} \approx 0,035.$$

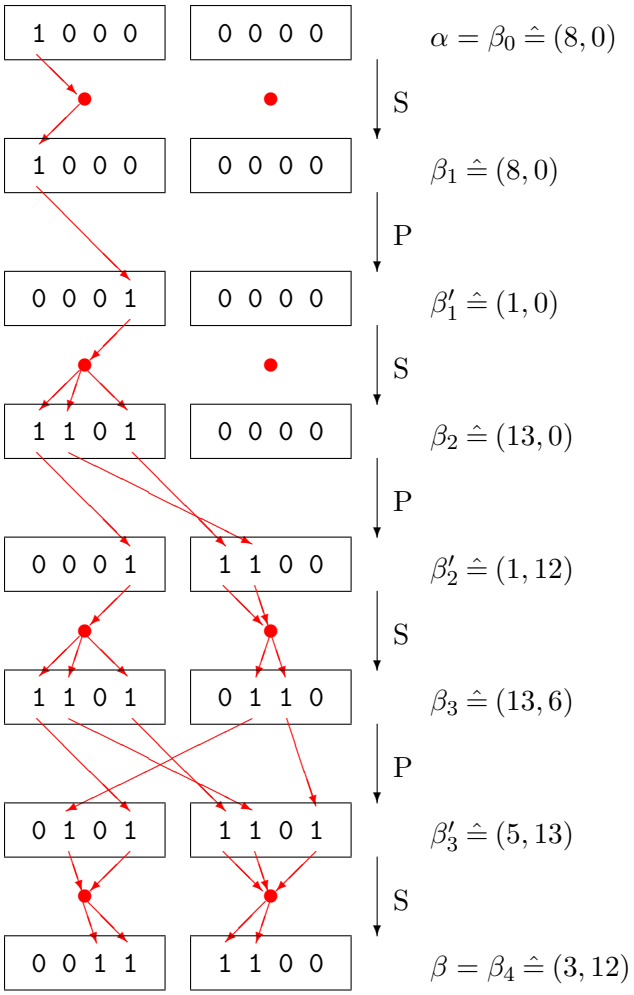


Abbildung 8.13: Ein linearer Pfad mit Verzweigungen („Trail“). Bei S wird die Linearform im Bild jeweils *gewählt* (anhand des Potenzials), angedeutet durch den roten Punkt; bei P entsteht die Linearform im Bild jeweils durch Transformation.

Die übrigen Kennzahlen sind

$$p \approx \frac{265}{512} \approx 0,518, \quad \lambda \approx \frac{81}{65536} \approx 0,0012, \quad N \approx \frac{65536}{27} \approx 2427,$$

letzteres als die Zahl der für den 95-prozentigen Erfolg benötigten bekannten Klartexte.

Das wäre im Aufwand immer noch geringer als die Exhaustion über alle 65536 möglichen Schlüssel – aber es gibt ja insgesamt nur 256 verschiedene mögliche Klartexte, so dass wir gar keinen Ansatzpunkt für die Analyse finden – die lineare Kryptoanalyse hat durch die Erhöhung der Rundenzahl ihren Sinn verloren.

8.2.13 Ausblick

Wir haben gesehen, dass die lineare Kryptoanalyse Anhaltspunkte für die Sicherheit einer Chiffre gibt, insbesondere für die Zunahme der Sicherheit durch Erhöhung der Rundenzahl. Eine mathematisch befriedigende Grundlage gibt es aber nur für einen Teil der Theorie. Die vorhandenen

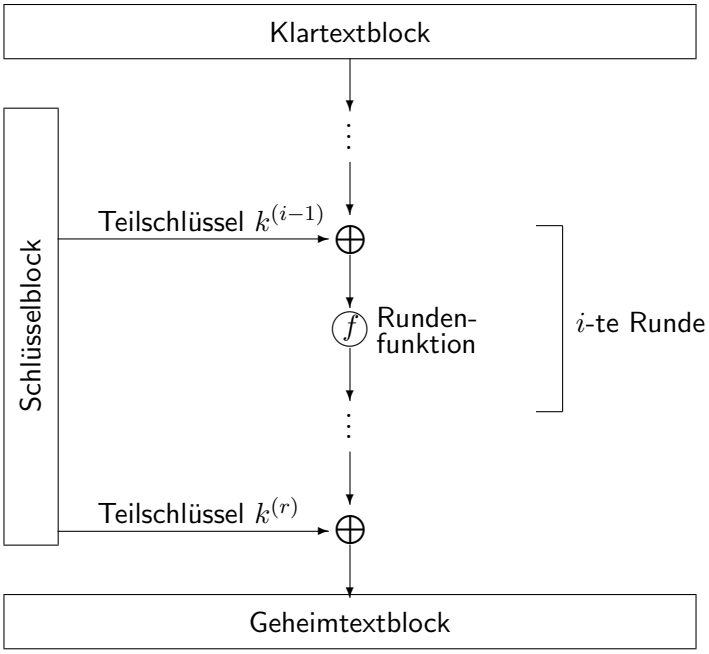


Abbildung 8.14: Grobe Struktur des AES-Verfahrens

Publikationen beschäftigen sich meist mit ad-hoc-Analysen konkreter Verschlüsselungsverfahren. So hat etwa Matsui gezeigt, wie man beim DES-Verfahren mit 2^{43} bekannten Klartexten 14 Schlüsselbits mit hoher Gewissheit bestimmen kann, was die Exhaustion auf die übrigen $42 = 56 - 14$ Schlüsselbits reduzierte und somit auch praktisch durchführbar war (vorausgesetzt, man kennt so viele Klartexte).

Die Betrachtung der linearen Kryptoanalyse ist als Beispiel zu sehen. Für die differenzielle Kryptoanalyse sowie die verallgemeinerten und gemischten Varianten laufen die Überlegungen einen analogen Gang. Zum Weiterlesen sei das Buch [Sti06] empfohlen. Dort werden auch die wichtigen Verfahren DES und AES beschrieben.

AES

Die Abbildungen 8.14 und 8.15 beschreiben grob den Aufbau des AES-Verfahrens[81] und zeigen, wie die hier hergeleiteten Konstruktionsprinzipien dabei umgesetzt wurden.

- Die Blocklänge ist $n = 128$, die Schlüssellänge $l = 128$, 192 oder 256, die Zahl der Runden $r = 10$, 12 oder 14.
- Zu Beginn jeder Runde und am Ende des Verfahrens wird ein Teilschlüssel auf den aktuellen Bitblock aufaddiert wie in den Beispielen A, B und C, Abbildungen 8.5, 8.8 und 8.9, insgesamt also $r + 1$ Teilschlüssel.
- Die 128-Bit-„Teilschlüssel“ $k^{(i)}$ sind nicht wirklich Teilschlüssel, sondern werden aus dem Gesamtschlüssel k nach einem etwas komplizierten Verfahren („Schlüsselauswahl“) extrahiert. Insbesondere sind sie nicht unabhängig.

[81] In CrypTool 2 unter „Moderne Verfahren“/ „AES“ zu finden.

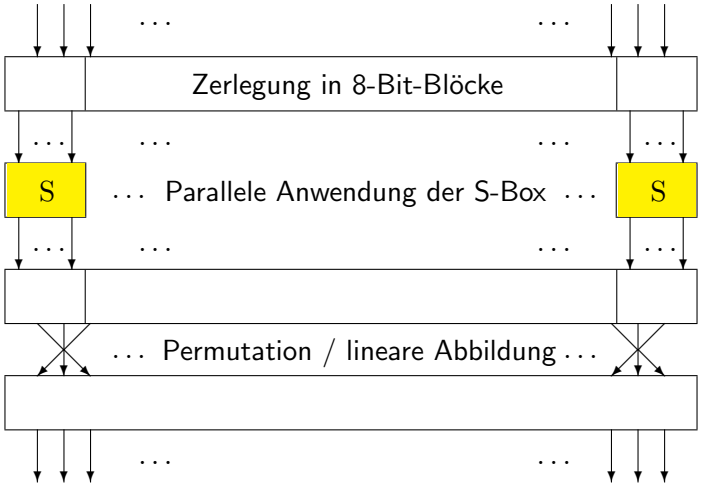


Abbildung 8.15: Die Rundenfunktion f des AES-Verfahrens

- Zu Beginn jeder Runde wird der aktuelle 128-Bitblock in 16 Teilblöcke zu 8-Bit zerlegt. Auf jeden dieser Teilblöcke wird die gleiche S-Box $S : \mathbb{F}_2^8 \longrightarrow \mathbb{F}_2^8$ angewendet. Diese hat eine mathematisch sehr elegante Beschreibung, für die aber einige zusätzliche Kenntnisse in abstrakter Algebra nötig sind, weshalb sie hier nicht gegeben wird. Das lineare Potenzial der S-Box von AES ist $\frac{1}{64}$. Dies kann man mit der Methode `linProf()` der Klasse `boolMap`, SageMath-Beispiel 8.47, explizit bestimmen, es gibt aber auch eine „tiefliegende“ mathematische Herleitung davon[82].

- Der Permutationsschritt besteht aus einer Permutation gefolgt von einer linearen Abbildung. Dieser „Diffusionsschritt“ ist also etwas komplexer als bei einem reinen SP-Netz nach Abbildung 8.2.

Eine Bemerkung noch zur Schlüsselauswahl: Werden die „Rundenschlüssel“, die wir mit $k^{(i)}$ bezeichnet haben, nicht einfach durch eine Teilauswahl von Bits bestimmt, sondern nach einem komplizierteren Verfahren, so wird der „wahre“ Schlüssel verschleiert. Die kryptoanalytischen Angriffe richten sich gegen den „effektiven“ Schlüssel, also gegen die Rundenschlüssel $k^{(i)}$. Diese reichen der Kryptoanalytikerin, um die Chiffre zu brechen. Ein kompliziertes Schlüsselauswahlverfahren kann aber verhindern, dass die Angreiferin eine Abhängigkeit der verschiedenen Rundenschlüssel ausnützt, etwa wenn diese als überlappende Teilblöcke aus dem „wahren“ Schlüssel gebildet werden.

[82] bei der als mathematisches Wunder die Zählung von Punkten elliptischer Kurven über endlichen Körpern der Charakteristik 2 vorkommt

8.3 Bitstrom-Chiffren

Bei einer Bitstrom-Chiffre wird der Reihe nach jedes einzelne Bit einer Bitkette nach einer anderen Vorschrift verschlüsselt, entweder unverändert gelassen oder negiert. Da man das Unverändert-Lassen als (algebraische) Addition (also XOR) von 0, das Negieren als Addition von 1 beschreiben kann, ist jede Bitstrom-Chiffre als XOR-Verschlüsselung im Sinne des folgenden Abschnitts interpretierbar[83]. Allerdings macht es einen Unterschied, ob der zu addierende Schlüsselstrom unabhängig vom Klartext vorher festgelegt wird – man spricht dann auch von einer **synchronen Bitstrom-Chiffre** –, oder ob er sich abhängig vom Klartext oder anderen Kontext-Parametern ändern kann – dann handelt es sich um eine **asynchrone Bitstrom-Chiffre**.

In diesem Abschnitt werden nur synchrone Bitstrom-Chiffren behandelt. Auch Stromchiffren für andere Zeichenvorräte als die Bits 0, 1 bleiben hier außen vor.

8.3.1 XOR-Verschlüsselung

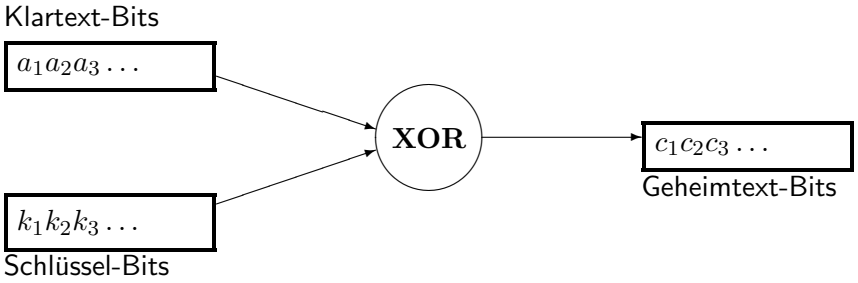


Abbildung 8.16: Das Prinzip der XOR-Verschlüsselung

Die einfachste und gängigste Art von Bitstrom-Chiffren ist die XOR-Verschlüsselung. Hierbei wird der Klartext als Folge von Bits aufgefasst[84]. Auch der Schlüssel ist eine Folge von Bits, die **Schlüsselstrom** genannt wird. Verschlüsselt wird, indem das jeweils nächste Bit des Klartexts mit dem nächsten Bit des Schlüsselstroms binär addiert wird. Abbildung 8.16 illustriert dieses Vorgehen[85], Abbildung 8.17 zeigt ein Beispiel.

```
a: 01000100011101 ...
k: 10010110100101 ...
   ------------------
c: 11010010111000 ...
```

Abbildung 8.17: Ein Beispiel zur XOR-Verschlüsselung

[83] Der Schlüssel wäre die „Differenz“ zwischen Geheimtext und Klartext wie in Abbildung 8.19.

[84] „Gewöhnliche“ Texte kann man etwa mit der SageMath-Methode `ascii_to_bin()` aus dem Modul `sage.crypto.util` in Bitketten umwandeln. Für den Rückweg dient die Methode `bin_to_ascii()`. Diese Bitketten gehören aber zur Klasse `StringMonoidElement` und sind umständlich weiter zu verarbeiten. Wir definieren daher im SageMath-Beispiel 8.38 eine Funktion `txt2bbl`, die ASCII-Texte als Bitblöcke wiedergibt. Für die weitere Verwandlung in Bitketten kann man `bbl2str` anschließen.

[85] In CrypTool 2 unter „Klassische Verfahren“/ „XOR“ zu finden, im Anhang 8.4.3 als `xor`, siehe SageMath-Beispiel 8.39.

Historisch wurde die XOR-Verschlüsselung in den Zwanzigerjahren des 20. Jahrhunderts zur Verschlüsselung von Fernschreiber-Nachrichten eingesetzt. Solche Nachrichten wurden auf Lochstreifen gestanzt, jeweils 5 Bits nebeneinander. Als Schlüssel diente ein weiterer Lochstreifen. Die entsprechenden Verschlüsselungsgeräte hießen Chiffrier-Fernschreiber und Schlüsselzusätze. Zum US-Patent angemeldet wurde das Verfahren 1918 von Vernam[86], zunächst mit periodischem Schlüssel, der aus einem an den Enden zusammengeklebten Lochstreifen bestand. Dass man unbedingt einen nichtperiodischen Schlüssel verwenden sollte, wurde von Mauborgne[87], einem Offizier und späteren Chief Signal Officer der US Army, bemerkt.

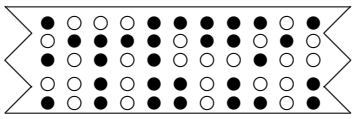


Abbildung 8.18: Lochstreifen (Punched tape) – jede Spalte repräsentiert ein 5-Bit-Zeichen

Als One-Time-Pad (OTP) lieferte die XOR-Chiffre später ein Beispiel für perfekte Sicherheit im Sinne von Shannon. Als Algorithmus A5 bzw. E_0 wirkt sie mit, die Mobil-Telefonie bzw. das Bluetooth-Protokoll für Datenübertragung per Funk scheinbar sicher zu machen. Sie kommt als RC4 im SSL-Protokoll vor, das die Client-Server-Kommunikation im World Wide Web (gelegentlich) verschlüsselt, und in der PKZIP-Verschlüsselung. Viele weitere aktuelle Anwendungen kann man leicht finden, und viele davon erfüllen nicht die erwarteten Sicherheitsansprüche.

> *Die Spannweite der XOR-Verschlüsselung reicht von trivial zu brechenden Verfahren bis hin zu unbrechbaren Chiffren.*

Vorteile der XOR-Verschlüsselung:

- Der Verschlüsselungsalgorithmus und der Entschlüsselungsalgorithmus sind identisch: Da $c_i = a_i + k_i$, ist $a_i = c_i + k_i$, d. h. zur Entschlüsselung wird der Schlüsselstrom auf den Geheimtext addiert (elementweise binär).
- Das Verfahren ist extrem einfach zu verstehen und zu implementieren ...
- ... und sehr schnell – vorausgesetzt, der Schlüsselstrom ist schon vorhanden. Für hohe Datenübertragungsraten kann man den Schlüsselstrom auf beiden Seiten vorherberechnen.
- Bei gut gewähltem Schlüsselstrom ist eine sehr hohe Sicherheit möglich.

Nachteile der XOR-Verschlüsselung:

- Das Verfahren ist anfällig gegen bekannten Klartext; jedes erratene Klartextbit ergibt ein Schlüsselbit.
- Die Angreiferin kann bei bekanntem Klartextstück das entsprechende Schlüsselstück ermitteln und dann diesen Klartext beliebig austauschen – z. B. „ich liebe dich“ durch „ich hasse dich“ ersetzen oder einen Geldbetrag von 1000 € auf 9999 € ändern. D. h., die Integrität der Nachricht ist unzureichend geschützt[88].

[86] Gilbert Vernam, US-amerikanischer Ingenieur, 4.4.1890–7.2.1960.
[87] Joseph Mauborgne, US-amerikanischer Offizier, 26.2.1881–7.7.1971.
[88] Die Nachrichten-Integrität ist durch zusätzliche Maßnahmen zu sichern.

- Es gibt keine Diffusion im Sinne der Shannonschen Kriterien, da jedes Klartext-Bit nur das entsprechende Geheimtext-Bit beeinflusst[89].
- Jegliche Wiederverwendung eines Teils der Schlüsselfolge (auch in Gestalt einer Periode) macht die betroffenen Geheimtexte angreifbar. Die historischen Erfolge beim Brechen von Strom-Chiffren beruhen meist auf diesem Effekt, so zum Beispiel bei Chiffrier-Fernschreibern und Schlüsselzusätzen im zweiten Weltkrieg oder beim Projekt Venona im Kalten Krieg.

Im Zusammenhang mit dem ersten Nachteil, der Anfälligkeit für Angriffe mit bekanntem Klartext, hat der gewöhnliche ISO-Zeichensatz für Texte eine systematische Schwachstelle: Die Kleinbuchstaben `a..z` beginnen im 8-Bit-Code alle mit `011`, die Großbuchstaben `A..Z` alle mit `010`[90]. Eine vermutete Folge von sechs Kleinbuchstaben (egal welche das sind) enthüllt schon $6 \cdot 3 = 18$ Schlüsselbits.

Mit anderen Worten: Dass die Angreiferin eine gute Portion Klartext kennt oder erraten kann, ist bei einer XOR-Chiffre gar nicht zu vermeiden. Die Sicherheit vor einem Angriff mit bekanntem Klartext ist hier noch wichtiger als bei anderen kryptographischen Verfahren.

8.3.2 Erzeugung des Schlüsselstroms

Für die Erzeugung des Schlüsselstroms sind drei naive Methoden gängig:

- periodische Bitfolge,
- Lauftext,
- „echte“ Zufallsfolge.

Eine bessere Methode verwendet eine

- Pseudozufallsfolge

und führt zu wirklich praktikablen Verfahren. Hierbei ist allerdings die Qualität des Schlüsselstroms sehr kritisch.

Periodische Bitfolgen

Beispiel: Schlüsselfolge der Periode 8 mit $k =$ `10010110`. Die Buchstaben werden nach dem ISO-Zeichensatz durch Bytes repräsentiert.

```
      D    |   u    |        |   b    |   i    |   s    |
a: 01000100|01110101|00100000|01100010|01101001|01110011|
k: 10010110|10010110|10010110|10010110|10010110|10010110|
   -------- -------- -------- -------- -------- --------
c: 11010010|11100011|10110110|11110100|11111111|11100101|

      t    |        |   d    |   o    |   o    |   f
```

[89] Bei Blockchiffren war die Diffusion eines der grundlegenden Kriterien.

[90] Das Auftreten vieler Nullen in den Leitbits der Bytes ist übrigens ein sehr wichtiges Erkennungsmerkmal für natürlichsprachigen Text in europäischen Sprachen.

```
01110100|00100000|01100100|01101111|01101111|01100110
10010110|10010110|10010110|10010110|10010110|10010110
-------- -------- -------- -------- -------- --------
11100010|10110110|11110010|11111001|11111001|11110000
```

Das kann man leicht per Hand ausführen, aber auch mit dem SageMath-Beispiel 8.19 nachvollziehen.

Werden in diesem Beispiel die Geheimtext-Bytes in Zeichen des ISO-9960-1-Zeichensatzes zurückgewandelt, sieht der Geheimtext so aus

Ò ã ¶ ô œ å â ¶ ò ù ù ■

und kann Laien vielleicht beeindrucken. Einem Fachmann fällt sofort auf, dass alle Zeichen in der oberen Hälfte der möglichen 256 Bytes liegen. Das legt die Vermutung nahe, dass ein gewöhnlicher Text mit einem Schlüssel behandelt wurde, dessen Leitbit eine 1 ist. Versucht er, das auffällig wiederholte Zeichen ¶ = `10110110` als Leerzeichen `00100000` zu deuten, kann er sofort den Schlüssel als Differenz `10010110` bestimmen und hat die Verschlüsselung gebrochen.

> *Bekannter oder vermuteter Klartext führt leicht zu einem erfolgreichen Angriff auf die periodische XOR-Verschlüsselung.*

SageMath-Beispiel 8.19 XOR-Verschlüsselung in Python/SageMath

```
sage: testtext = "Du bist doof"
sage: bintext = txt2bbl(testtext)
sage: binstr = bbl2str(bintext)
sage: binstr
'010001000111010100100000011000100110100101110011
011101000010000001100100011011110110111101100110'
sage: testkey = [1,0,0,1,0,1,1,0]
sage: keystr = bbl2str(testkey)
sage: keystr
'10010110'
sage: cipher = xor(bintext,testkey)
sage: ciphstr = bbl2str(cipher)
sage: ciphstr
'110100101110001110110110111101001111111111100101
111000101011011011110010111110011111100111110000'
```

MS-Word und periodisches XOR

Die folgende Tabelle (die man leicht selbst erzeugen kann) gibt typische Häufigkeiten an für die häufigsten Bytes in MS-Word-Dokumenten.

Byte (hexadezimal)	**Bits**	**Häufigkeit**
`00`	00000000	7–70%
`01`	00000001	0.8–17%
`20` (Leerzeichen)	00100000	0.8–12%
`65` (e)	01100101	1–10%
`FF`	11111111	1–10%

Die Häufigkeiten hängen allerdings sehr stark von der Art des Dokuments ab und ändern sich mit jeder Version. Die Schwankungen sind sehr groß, es gibt immer wieder unerwartete Spitzen, und alle Bytes `00`–`FF` können vorkommen. Aber darauf kommt es hier gar nicht an. Jedenfalls beobachtet man:

> *Es gibt lange Ketten von* `00`*-Bytes.*

Ist ein Word-Dokument XOR-verschlüsselt mit periodisch wiederholtem Schlüssel, so ergibt sich aus der Häufung von Nullen eine effiziente Analyse-Methode: den Strom der Geheimtext-Bits in Blöcke entsprechend der Periodenlänge[91] einteilen und die Blöcke paarweise addieren. Besteht der eine Block im Klartext im wesentlichen aus Nullen, entsteht als Summe lesbarer Klartext. Wir betrachten also die Situation:

	…	Block 1			…	Block 2			…
Klartext:	…	a_1	…	a_s	…	0	…	0	…
Schlüssel:	…	k_1	…	k_s	…	k_1	…	k_s	…
Geheimtext:	…	c_1	…	c_s	…	c'_1	…	c'_s	…

mit $c_i = a_i + k_i$ und $c'_i = 0 + k_i = k_i$ für $i = 1, \ldots, s$. D. h., der Schlüssel scheint bei Block 2 durch, aber das muss die Angreiferin erst mal merken. Bei der versuchsweisen paarweisen Addition aller Blöcke erhält sie unter anderem

$$c_i + c'_i = a_i + k_i + k_i = a_i \quad \text{für } i = 1, \ldots, s,$$

also einen Klartextblock. Falls sie das erkennt (an typischen Strukturen), hat sie auch den Schlüssel $k_1, \ldots, k_s$.

Sollte aber bei der Addition zweier Geheimtextblöcke sogar ein Nullblock herauskommen, d. h., sind zwei Geheimtextblöcke gleich, so waren auch beide Klartextblöcke gleich. Dann ist die Wahrscheinlichkeit groß, dass beide Klartextblöcke nur aus Nullen bestanden. Auch in diesem Fall ist der Schlüssel enthüllt. Wir halten fest:

> *Die XOR-Verschlüsselung mit periodischem Schlüssel ist für Dateien mit bekannter Struktur ziemlich einfach zu brechen.*

Das gilt auch bei einer großen Periode, etwa von 512 Bytes = 4096 Bits, trotz des überastronomisch riesigen Schlüsselraums aus 2^{4096} verschiedenen potenziellen Schlüsseln.

Lauftext-Verschlüsselung

Eine Möglichkeit, die Periodizität zu vermeiden, besteht darin, als Schlüssel eine Datei zu verwenden, die mindestens die gleiche Länge wie der Klartext hat. Die Bezeichnung als Lauftext-Verschlüsselung entstand in der klassischen Kryptographie, weil als Schlüssel oft der Text eines Buches ab einer bestimmten Stelle gewählt wurde. Gebrochen wurden solche Verschlüsselungen meistens dadurch, dass das Buch erraten wurde. Das ist auch der offensichtlichste Schwachpunkt, wenn man diese Idee auf die XOR-Verschlüsselung von Dateien überträgt. Sobald die Quelle der Bits, etwa eine Datei oder eine CD oder DVD, der Gegnerin bekannt ist, ist der Schlüsselraum viel zu klein – das (lineare!) Durchprobieren von mehreren Gigabyte Daten, um die richtige Anfangsstelle zu finden, ist mit Computer-Hilfe wenig aufwendig.

[91] Falls die Periodenlänge nicht schon bekannt ist, kann man sie oft mit den nach Kasiski, Friedman oder Sinkov benannten Methoden ermitteln, die in der klassischen Kryptoanalyse auf periodische polyalphabetische Chiffren angewendet werden. Ansonsten hilft Durchprobieren.

Aber auch wenn die Schlüsselquelle nicht erraten wird, ist die Kryptoanalyse möglich, da sowohl Klartext als auch Schlüssel Strukturen haben, die durch die Verschlüsselung nicht völlig verschleiert werden. Darauf soll hier aber nicht weiter eingegangen werden[92]. Festzuhalten ist jedenfalls:

> *Die XOR-Verschlüsselung mit Lauftext ist für Dateien mit bekannter Struktur nicht allzu schwer zu brechen.*

Echte Zufallsfolgen

Das andere Extrem ist, als Schlüsselstrom eine rein zufällige Folge von Bits zu verwenden. Dann heißt das Verfahren **(binäres) One-Time-Pad (OTP)**. Insbesondere darf kein Teil des Schlüsselstroms irgendwann wiederholt verwendet werden. Die Bezeichnung „Pad“ rührt daher, dass man sich die Bits wie auf einem Abreißkalender vorstellt – jedes Blatt wird nach Benutzung sofort vernichtet. Eine solche Verschlüsselung kann nicht gebrochen werden, d. h., das Verfahren ist perfekt sicher. Shannon hat das formal bewiesen, siehe etwa [Sti06].

Man kann die Sicherheit aber auch ohne mathematischen Formalismus wie folgt begründen: Der Geheimtext gibt – außer der Länge – keinerlei Informationen über den Klartext preis. Er kann aus *jedem beliebigen* Klartext gleicher Länge entstanden sein. Man muss nur die Differenz zwischen dem Geheimtext und dem beliebigen Text als Schlüssel verwenden: Der Geheimtext sei $c = a + k$ mit dem Klartext a und dem Schlüssel k, alles als Bitströme gedacht und Bit für Bit addiert wie in Abbildung 8.16. Für einen beliebigen anderen Klartext b ist dann $c = b + k'$ ebenfalls eine mögliche gültige Verschlüsselung, wenn man als Schlüssel einfach $k' = b + c$ verwendet.

Diese Eigenschaft des OTP kann man ausnutzen, um bei erzwungener Entschlüsselung einen unverfänglichen Klartext zu produzieren, wie in Abbildung 8.19 demonstriert.

Wenn das One-Time-Pad so perfekt ist – warum wird es denn nicht generell verwendet?

- Unhandliches Schlüssel-Management: Die Vereinbarung eines Schlüssels wird zum Problem – er ist ja ebenso lang wie der Klartext und schwer zu merken. Die Kommunikationspartner müssen also im Voraus den Schlüsselstrom vereinbaren und aufzeichnen. Wollen sie die Schlüssel erst bei Bedarf vereinbaren, benötigen sie dazu einen sicheren Kommunikationsweg, aber dann können sie den (auch nicht längeren) Klartext gleich direkt verschicken.
- Keine Eignung zur Massenanwendung: Das Verfahren ist bestenfalls zur Kommunikation zwischen *zwei* Partnern geeignet, wegen des Aufwands bei der Schlüsselverwaltung nicht für eine Mehr-Parteien-Kommunikation.
- Das Problem der Nachrichtenintegrität besteht beim OTP wie bei jeder XOR-Verschlüsselung.

Ein weiteres praktisches Problem stellt sich bei der Verschlüsselung am Computer: Woher erhält man „echten Zufall“? Als echter Zufall gelten physikalische Ereignisse wie der radioaktive Zerfall oder das Rauschen auf einem optischen Sensor. Aber auch die vermeintlich deterministische Maschine „Computer“ produziert solchen Zufall. Es gibt spezielle Chips, die auslesbares Rauschen produzieren; es gibt aber auch unvorhersehbare Ereignisse, z. B. die genauen Mausbewegungen eines Nutzers oder eingehende Netzpakete, die zwar nicht völlig zufällig sind, aus

[92] In JCrypTool ist unter „Analysen“/„Viterbi-Analyse“ eine automatische Erkennung der beiden Klartexte zu finden, die nur das per Lauftext-Verschlüsselung erzeugte Chiffrat benötigt.

```
Plain bits and text:
01000100 01101001 01100101 01110011 01100101 01110010  Dieser
00100000 01010100 01100101 01111000 01110100 00100000   Text
01101001 01110011 01110100 00100000 01100010 01110010  ist br
01101001 01110011 01100001 01101110 01110100 00101110  isant.

Key bits:
11001000 11010110 00110011 11000000 00111011 10001110
00001000 11101111 01001001 11100101 10111100 10111001
00010010 11000110 01110011 11010111 11000100 01100000
11100110 00010111 01101010 10111011 00010101 11011000

Cipher bits:
10001100 10111111 01010110 10110011 01011110 11111100
00101000 10111011 00101100 10011101 11001000 10011001
01111011 10110101 00000111 11110111 10100110 00010010
10001111 01100100 00001011 11010101 01100001 11110110

Pseudokey bits:
11001000 11010110 00110011 11000000 00111011 10001110
00001000 11101111 01001001 11100101 10111100 10111001
00010010 11000110 01110011 11010111 11001110 01110011
11111101 00001001 01100111 10111010 00010010 11011000

Pseudodecrypted bits and text:
01000100 01101001 01100101 01110011 01100101 01110010  Dieser
00100000 01010100 01100101 01111000 01110100 00100000   Text
01101001 01110011 01110100 00100000 01101000 01100001  ist ha
01110010 01101101 01101100 01101111 01110011 00101110  rmlos.
```

Abbildung 8.19: Eine XOR-Verschlüsselung und ein vorgetäuschter passender Klartext

denen man aber zufällige Anteile extrahieren und etwa auf Unix-Systemen von `/dev/random` abrufen kann.

Solche wie „echt“ auch immer erzeugten Zufallsbits sind aber zur direkten Verschlüsselung per OTP kaum brauchbar. Das Problem ist, dass der Empfänger sie nicht reproduzieren kann, also auf eine explizite Schlüsselvereinbarung angewiesen ist. Dennoch hat auch der „echte“ Zufall wichtige kryptographische Anwendungen: Schlüssel für beliebige Verschlüsselungsverfahren sollen möglichst zufällig und damit für die Gegnerin nicht erratbar erzeugt werden. Viele kryptographische Protokolle verwenden sogenannte Nonces, die außer ihrer Zufälligkeit keine Bedeutung haben wie z. B. die Initialisierungsvektoren der Betriebsarten bei Blockverschlüsselung oder die „Challenge“ bei einem starken Authentisierungsverfahren („Challenge-Response-Verfahren“).

Für die XOR-Verschlüsselung als Approximation an das OTP sind dagegen algorithmisch erzeugte Bitfolgen viel praktikabler. Diese sollen aber für die Gegnerin nach Möglichkeit nicht von echt zufälligen Folgen unterscheidbar sein. Man spricht dann von „Pseudozufall“, und die Erzeugung solcher Folgen ist von hoher kryptologischer Relevanz.

> *Wenn die XOR-Verschlüsselung statt mit einer Zufallsfolge mit einer Pseudozufallsfolge durchgeführt wird, geht die perfekte Sicherheit des One-Time-Pad verloren. Wenn die Pseudozufallsfolge allerdings kryptographisch sicher ist (Abschnitt 8.3.9), hat die Gegnerin keine Chance, das auszunutzen.*

8.3.3 Pseudozufallsgeneratoren

Man versucht, die idealen Eigenschaften des One-Time-Pad zu approximieren, indem man statt einer „echten“ Zufallsfolge eine von einem Algorithmus („Zufallsgenerator“) aus einem „effektiven Schlüssel“ (= kurzen Startwert) erzeugte „pseudozufällige“ Bitfolge verwendet. Ein wesentlicher Unterschied zu einer echten Zufallsfolge besteht also in der leichten Reproduzierbarkeit. Dennoch ist sogar bei mäßiger Qualität des Zufallsgenerators der Geheimtext resistent gegen statistische Analysen. Es bleibt das Problem, die Sicherheit gegen einen Angriff mit bekanntem Klartext in den Griff zu bekommen.

Die entscheidende Frage an eine Pseudozufallsfolge bzw. an den sie erzeugenden Zufallsgenerator[93] ist:

> Kann man aus einem bekannten (auch fragmentierten) Stück der Folge weitere Bits – vorwärts oder rückwärts – bestimmen?

Die Antwort für die „klassischen“, in statistischen Anwendungen und Simulationen verwendeten Zufallsgeneratoren ist JA, siehe etwa Abschnitt 8.3.4. Wir werden aber auch Zufallsgeneratoren kennen lernen, die in diesem Sinne – vermutlich – kryptographisch sicher sind. Ein wesentliches Problem ist dabei, einen akzeptablen Kompromiss zwischen der Geschwindigkeit der Erzeugung und der Sicherheit zu finden.

Methodisch gibt es zwei ernsthaft praktizierte Hauptrichtungen zur Erzeugung eines Schlüsselstroms:

- (rückgekoppelte) Schieberegister und Kombinationen davon als Anwendung der Booleschen Algebra,
- perfekte Zufallsgeneratoren als Anwendung zahlentheoretischer Methoden.

[93] Der Vorsatz „Pseudo-“ wird beim Zufallsgenerator meist weggelassen, wenn aus dem Kontext keine Missverständnisse zu befürchten sind.

Die schematische Funktionsweise eines Zufallsgenerators ist in Abbildung 8.20 skizziert. Er verbirgt einen Zustand, der sich von Schritt zu Schritt nach einem vorgegebenen Algorithmus ändert. Dieser Algorithmus wird von Parametern gesteuert, die z. T. „öffentlich“ bekannt sein können, z. T. als Komponente des Schlüssels geheim gehalten werden. Der Anfangszustand (= Startwert) wird echt zufällig gewählt und ebenfalls geheim gehalten. In jedem Schritt gibt der Zufallsgenerator, abhängig von seinem momentanen Zustand, einen Wert aus, solange bis er durch externen Eingriff gestoppt wird.

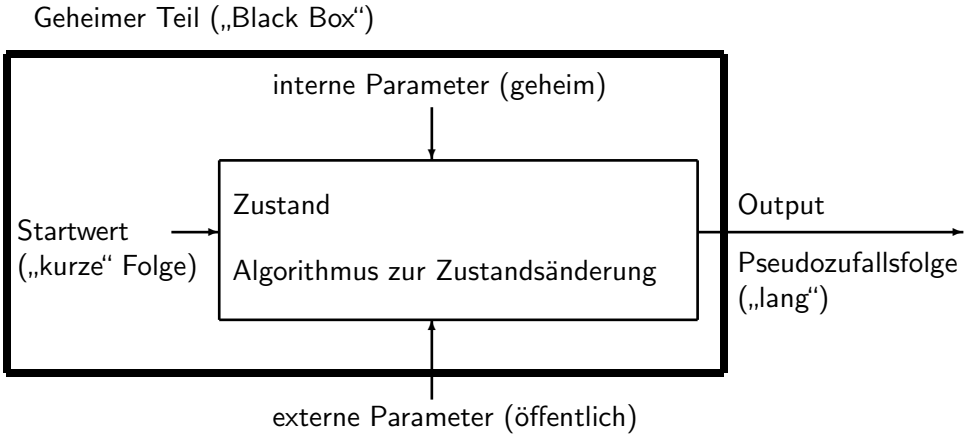


Abbildung 8.20: Das Prinzip des (Pseudo-)Zufallsgenerators

Der Zufallsgenerator wandelt also eine kurze, echt zufällige, Bitfolge, den Startwert, in eine lange pseudozufällige Folge um. Man nennt diesen Vorgang in der Kryptologie auch „Schlüsselexpansion“.

Rückgekoppelte Schieberegister

Eine klassische und weit verbreitete Methode zur Erzeugung von Pseudozufallsfolgen ist die Schieberegister-Methode. Im Englischen heißen Rückgekoppelte Schieberegister Feedback shift registers (FSR). Diese Methode wurde von Golomb[94] 1955 erstmals vorgeschlagen, wird aber oft nach Tausworthe benannt, der die Idee 1965 in einer Arbeit aufgriff. Sie ist besonders leicht in Hardware zu realisieren.

Hat das Register die Länge l, wird es durch eine Boolesche Funktion $f\colon \mathbb{F}_2^l \longrightarrow \mathbb{F}_2$, die „Rückkopplungsfunktion“, repräsentiert. Die Funktionsweise ist in Abbildung 8.21 skizziert. Das rechte Bit u_0 wird ausgegeben, alle anderen Bits rücken eine Zelle nach rechts, und von links wird das Bit $u_l = f(u_{l-1}, \ldots, u_0)$ nachgeschoben. Die gesamte Folge wird also nach der Rekursionsformel

$$u_n = f(u_{n-1}, \ldots, u_{n-l}) \quad \text{für } n \geq l \tag{8.8}$$

berechnet. Das SageMath-Beispiel 8.20 definiert ein allgemeines rückgekoppeltes Schieberegister mit Rückkopplungsfunktion f. Im SageMath-Beispiel 8.21 wird es mit einer konkreten exemplarischen Rückkopplungsfunktion eingesetzt, um eine Bitfolge zu erzeugen. Die Fortschaltung des Registers in diesem konkreten Beispiel wird in Tabelle 8.19 veranschaulicht. Dass das Ergebnis nicht besonders zufällig aussieht, ist ein Warnhinweis darauf, dass man bei der Wahl der Parameter größere Sorgfalt walten lassen sollte.

[94] Solomon W. Golomb, amerikanischer Mathematiker und Ingenieur, *30.5.1932.

Die Bits $u_0, \ldots, u_{l-1}$ dienen als Startwerte. Die Schlüsselexpansion besteht also darin, dass aus dem kurzen Startwert $u = (u_0, \ldots, u_{l-1})$ als effektivem Schlüssel der Länge l ein beliebig langer Schlüsselstrom $u_0, u_1, \ldots$ erzeugt wird. Allerdings kann man in diesem Kontext auch die internen Parameter, das ist hier die Rückkopplungsfunktion f oder zumindest einige ihrer Parameter, als Komponente des Schlüssels ansehen, so dass die tatsächliche effektive Schlüssellänge dann größer als l ist.

Hierin unterscheidet sich eine Implementierung in Hardware von einer in Software: In Hardware wird man nur mit zusätzlichen Schaltkreisen die Rückkopplungsfunktion als veränderlich implementieren können, so dass man sie hier in der Regel als konstant und damit auch (zumindest langfristig) als der Gegnerin bekannt annehmen muss. In Software dagegen ist die Rückkopplungsfunktion jederzeit leicht änderbar und damit als Bestandteil des Schlüssels geeignet.

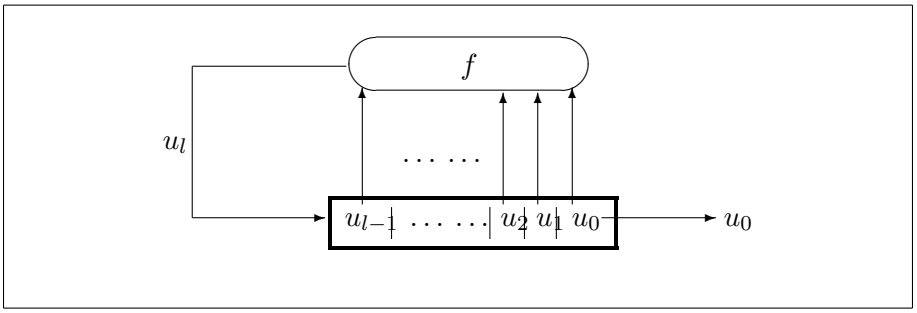


Abbildung 8.21: Ein rückgekoppeltes Schieberegister (beim ersten Iterationsschritt). Die Boolesche Funktion f berechnet aus dem aktuellen Zustand des Registers ein neues Bit, das von links nachgeschoben wird.

SageMath-Beispiel 8.20 Ein rückgekoppeltes Schieberegister in Python/SageMath

```
def fsr(f,x,n):
  """Generate a feedback shift register sequence.
  Parameters: Boolean function f, start vector x,
  number n of output bits."""
  u = x
  outlist = []
  for i in range (0,n):
    b = f.valueAt(u)
    c = u.pop()
    u.insert(0,b)
    outlist.append(c)
  return outlist
```

Perioden endlicher Automaten

Ein rückgekoppeltes Schieberegister ist für den Informatiker ein Spezialfall eines endlichen Zustandsautomaten. Die Abfolge seiner Zustände ist periodisch oder zyklisch, d.h. er hat stets eine Periode. Das sieht man wie folgt.

SageMath-Beispiel 8.21 Eine Pseudozufallsfolge in Python/SageMath

```
sage: bits = "1000010111001001"
sage: x = str2bbl(bits)
sage: f = BoolF(x)
sage: start = [0,1,1,1]
sage: bitlist = fsr(f, start, 32)
sage: print(bbl2str(bitlist))
11101010101010101010101010101010
```

Rückkopplung		Zustand		Output
Start		0111	$\longrightarrow$	1
$f(0111) = 1$	$\longrightarrow$	1011	$\longrightarrow$	1
$f(1011) = 0$	$\longrightarrow$	0101	$\longrightarrow$	1
$f(0101) = 1$	$\longrightarrow$	1010	$\longrightarrow$	0
$f(1010) = 0$	$\longrightarrow$	0101	ab hier	periodisch

Tabelle 8.19: Fortschaltung eines (rückgekoppelten) Schieberegisters

Sei M eine endliche Menge mit $m = \#M$ Elementen. Wir stellen uns M als die Menge der möglichen „Zustände“ eines Automaten vor. Dazu sei eine Abbildung („Zustandsänderung“) gegeben:

$$g\colon M \longrightarrow M.$$

Für jedes Element („Anfangszustand“) $x_0 \in M$ definieren wir eine Folge $(x_i)_{i\geq 0}$ in M durch die Rekursionsformel $x_i = g(x_{i-1})$ für $i \geq 1$. Nach einer Vorperiode der Länge μ wird diese Folge periodisch mit einer Periode ν, siehe Abbildung 8.22, die Erklärung folgt gleich.

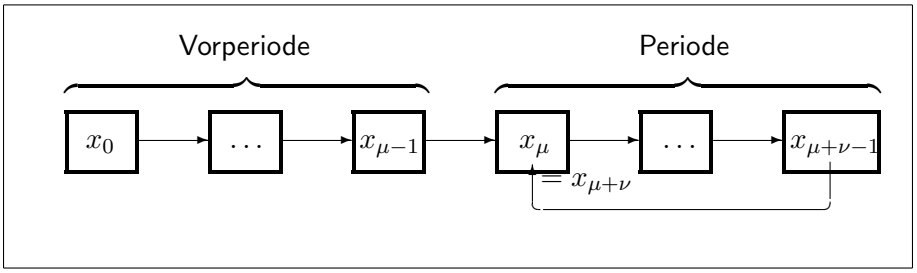


Abbildung 8.22: Periode und Vorperiode

Da die Menge M der möglichen Zustände endlich ist, müssen diese sich irgendwann wiederholen. D. h., es gibt kleinste ganze Zahlen $\mu \geq 0$ und $\nu \geq 1$, so dass $x_{\mu+\nu} = x_\mu$: Um das zu sehen, nimmt man einfach als μ den kleinsten Index, für den das Element x_μ irgendwo in der Folge noch einmal auftritt, und als $\mu+\nu$ den Index, bei dem diese erste Wiederholung auftritt. Dann ist auch

$$x_{i+\nu} = x_i \quad \text{für } i \geq \mu.$$

Dabei ist $0 \leq \mu \leq m-1$, $1 \leq \nu \leq m$, $\mu+\nu \leq m$. Die Werte $x_0, \ldots, x_{\mu+\nu-1}$ sind alle verschieden, und die Werte $x_0, \ldots, x_{\mu-1}$ erscheinen niemals wieder in der Folge.

Definition: μ heißt die (Länge der) **Vorperiode**, ν die (Länge der) **Periode**.

Pseudozufallsgeneratoren im Sinne von Abbildung 8.20 erzeugen immer periodische Folgen. Die Periode sollte aber so riesig sein, dass ihre Größenordnung in der praktischen Anwendung nie erreicht wird.

Lineare Schieberegister

Der einfachste und am besten verstandene Spezialfall rückgekoppelter Schieberegister sind die linearen[95]. Hier ist die Rückkopplungsfunktion f linear, also nach 8.1.9 nichts anderes als eine Vorschrift, aus einem l-Bit-Block eine Teilsumme zu bilden:

$$f(u_{n-1}, \ldots, u_{n-l}) = \sum_{j=1}^{l} s_j u_{n-j}, \tag{8.9}$$

wobei als Koeffizienten s_j ja nur 0 oder 1 vorkommen. Ist I die Teilmenge der Indizes j mit $s_j = 1$, so lässt sich die Iterationsformel (8.8) in der folgender Form schreiben:

$$u_n = \sum_{j \in I} u_{n-j} \tag{8.10}$$

Eine einfache graphische Repräsentation eines LFSR zeigt Abbildung 8.23. Hier definiert die Teilmenge I die Kontakte („Taps“), die die entsprechenden Zellen der Rückgabe-Summe definieren.

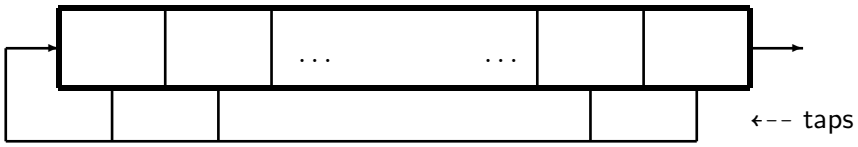


Abbildung 8.23: Einfache graphische Repräsentation eines LFSR

Bei geschickter Wahl der Parameter, die hier nicht weiter behandelt wird, hat die Folge eine Periode nahe 2^l – der Anzahl der möglichen verschiedenen Zustände des Registers – und ist durch statistische Tests praktisch nicht von einer gleichverteilten Zufallsfolge zu unterscheiden [Gol82] – erstaunlich, dass man mit einem so simplen Ansatz schon so guten Pseudozufall erzeugen kann! Den Startblock $u = (0, \ldots, 0)$ sollte man natürlich vermeiden. Für einen Startblock $\neq 0$ ist die maximal mögliche Periode $2^l - 1$. Ohne das hier näher auszuführen sei erwähnt, dass diese Periode leicht zu erreichen ist[96].

Bei der Anwendung für die Bitstrom-Verschlüsselung werden die geheimen internen Parameter, also die Koeffizienten $s_1, \ldots, s_l$, sowie die Startwerte $u_0, \ldots, u_{l-1}$ als Schlüssel betrachtet. Die Länge l des Registers wird dagegen meist als der Gegnerin bekannt angenommen.

Das SageMath-Beispiel 8.22, die Funktion `lfsr()`, implementiert ein lineares Schieberegister in SageMath[97,98]; Ausgabe ist ein pseudozufälliger Bitstrom. (Dabei ist `binScPr()` aus dem

[95] abgekürzt LFSR für Linear Feedback Shift Register; im Deutschen wird der Zusatz „rückgekoppelt“ meistens implizit angenommen.

[96] Genau dann, wenn das „Rückkopplungspolynom“ $1 + s_1 x + s_2 x^2 + \cdots + s_l x^l$ als Polynom über dem Körper $\mathbb{F}_2$ primitiv ist, ist die Periode $2^l - 1$. Achtung: Nicht das Rückkopplungspolynom mit der Rückkopplungsfunktion verwechseln: Ersteres ist ein (formales) Polynom in einer Variablen, letztere eine Boolesche Linearform in l Variablen.

[97] oder die Funktion `sage.crypto.lfsr.lfsr_sequence` von SageMath

[98] Im SageMath-Beispiel 8.51 wird ein systematischerer Zugang durch Definition einer geeigneten Klasse `LFSR` angeboten.

SageMath-Beispiel 8.39 das „Skalarprodukt“ zweier binärer Vektoren, also die Auswertung der durch `s` definierten Linearform für den Bitblock `u`.) Ein exemplarischer Aufruf dieser Funktion mit einem Schieberegister der Länge 16, aus dem 1024 Bits erzeugt werden sollen, ist im SageMath-Beispiel 8.23 zu finden und ergibt den Output in Tabelle 8.20 (ohne Klammern und Kommata wiedergegeben).

Man könnte auf diesen Bitstrom jetzt eine Reihe statistischer Tests, z. B. auf Gleichverteilung, anwenden und würde stets gute Ergebnisse sehen. Statt dessen wird die Folge in Abbildung 8.24 zur optischen Inspektion visualisiert – was natürlich noch weniger beweist. Man sieht aber, dass zumindest der äußere Eindruck der einer ziemlich zufälligen Bitfolge ist. Die Grafik wurde mit dem zweiten Teil des SageMath-Beispiels 8.23 erzeugt.

An den 9 aufeinanderfolgenden (schwarzen) Einsen in der drittletzten Zeile sollte man sich nicht stören: Die Wahrscheinlichkeit für 9 Einsen bei 9 Bits ist $(1/2)^9 = 1/512$. Bei einem zufälligen Bitstrom der Länge 1024 ist ein solcher „Run“ also mit hoher Wahrscheinlichkeit zu erwarten.

> *Über die kryptographische Eignung einer Pseudozufallsfolge sagen weder die üblichen statistischen Tests noch der visuelle Eindruck etwas aus.*

SageMath-Beispiel 8.22 Ein lineares Schieberegister (LSFR) in Python/SageMath

```
def lfsr(s,x,n):
  """Generate a linear feedback shift register sequence.
  Parameters: Coefficient vector s, start vector x, number n of
  output bits."""
  l = len(s)
  assert l == len(x), "lfsr_Error: Bad length of start vector."
  u = x                           # in Python use u = x.copy()
  outlist = []
  for i in range (0,n):
    b = binScPr(s, u)
    c = u.pop()
    u.insert(0,b)
    outlist.append(c)
  return outlist
```

```
11001000110101100011001111000000
00111011100011100000100011101111
01001001111001011011110010111001
00010010110001100111001111010111
11000100011000001110011000010111
01101010101110110001010111011000
11110000010000100010111100011110
10100111000001111000100001011000
01010101000101111110110011011101
11001001110111110001011000100010
11100100101111110011011001010011
00001100100001100110100011100100
11101000100101110110011011001010
11011100100110111001011100000011
00100010111101111000110000010001
01110100001110011111101000100101
00111010001111000100000000110110
10000101110101110001100000010001
11011011011110111001000110101001
10001111110110101010011111100001
11101110111101011001010110001010
00000100001001100110001110100110
00010100101110100000010101100100
10010110101011111110111111011101
11001010010100010010110111111110
10100101001111110110100100010001
10111100011001111001011111010110
01110111010100100010100101101111
01100111011000000111011111010000
11011101111111110000010001000100
10010111111110101011101110111111
01110010110000010001111001100111
```

Tabelle 8.20: Eine pseudozufälligen Bitfolge aus einem linearen Schieberegister (LSFR)

SageMath-Beispiel 8.23 Eine Pseudozufallsfolge in Python/SageMath

```
sage: coeff = [0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1]
sage: start = [0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1]
sage: bitlist = lfsr(coeff, start, 1024)
sage: print(bitlist)
### Visualization
sage: m = matrix(GF(2),32,bitlist)
sage: row = [1]*32
sage: n = matrix(GF(2),32,row*32)  # All entries 1
sage: p = m+n                    # Toggle bits of m ---> 1 = black
sage: p.subdivide(range(0,33),range(0,33))
sage: matrix_plot(p, subdivisions=True)
```

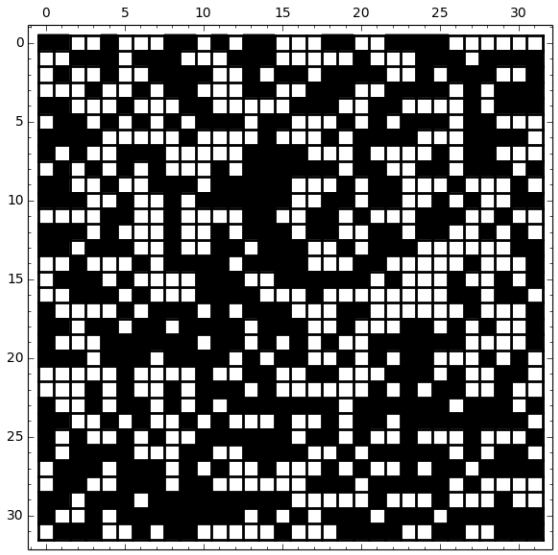


Abbildung 8.24: Visualisierung der pseudozufälligen Bitfolge aus Abbildung 8.20, erzeugt mit dem SageMath-Beispiel 8.20 (1 = schwarz, 0 = weiß)

8.3.4 Algebraischer Angriff auf lineare Schieberegister

Auch einfache Zufallsgeneratoren wie die linearen Schieberegister liefern, wie gesehen, Bitfolgen, die mit statistischen Mitteln nicht ohne weiteres von „echtem“ Zufall unterschieden werden können und für statistische Methoden der Kryptoanalyse keinen unmittelbar erfolgversprechenden Ansatz bieten. Anders sieht es aus, wenn man bekannten Klartext annimmt – dann erhält man Gleichungen für die Schlüsselbits. Werden die Schlüsselbits nach einem einfachen Algorithmus erzeugt, kann man mit algebraischer Kryptoanalyse, d. h. durch Auflösung dieser Gleichungen, auf Erfolg hoffen. Dies ist besonders für lineare Schieberegister[99] der Fall.

Nehmen wir an, ein lineares Schieberegister hat den Schlüsselbitstrom $u_0, u_1, \ldots$ nach den Formeln (8.9) und (8.10) erzeugt. Mit diesem Schlüsselstrom wurde ein Klartext a per XOR zum Geheimtext c verschlüsselt, $c_i = a_i + u_i$ für $i = 0, 1, \ldots$ Was kann die Gegnerin erreichen, wenn sie ein Stück vom Anfang des Klartexts kennt?

Nun, kennt sie die ersten $l + 1$ Bits des Klartexts, so hat sie sofort die entsprechenden Bits $u_0, \ldots, u_l$ des Schlüsselstroms, insbesondere den Startwert. Für die noch unbekannten Koeffizienten s_i kennt sie eine lineare Relation:

$$s_1 u_{l-1} + \cdots + s_l u_0 = u_l.$$

Jedes weitere bekannte Bit des Klartexts liefert eine weitere Relation, und mit l solchen Relationen, also insgesamt $2l$ Bits an bekanntem Klartext, liefert die sehr einfache Lineare Algebra über dem Körper $\mathbb{F}_2$ im allgemeinen eine eindeutige Lösung. Die $l \times l$-Koeffizientenmatrix dieses linearen Gleichungssystems ist im Wesentlichen die Matrix U des nächsten Abschnitts. Ein kleiner Umweg macht die Lösung noch ein wenig eleganter – in den nächsten, etwas mehr Mathematik voraussetzenden Abschnitten wird bewiesen:

Satz 8.3.1. *Ein lineares Schieberegister der Länge l ist aus den ersten $2l$ Bits vorhersagbar. Der Aufwand dazu beträgt ungefähr $\frac{1}{3} \cdot l^3$ Bitoperationen.*

Vorhersage linearer Schieberegister

Nehmen wir an, wir kennen die ersten $2l$ Bits $u_0, \ldots, u_{2l-1}$ aus einem linearen Schieberegister der Länge l. Die Lineare Algebra lässt sich eleganter formulieren, wenn man die **Zustandsvektoren**

$$u_{(i)} = (u_i, \ldots, u_{i+l-1}) \quad \text{für } i = 0, 1, \ldots$$

verwendet. Dabei ist $u_{(i)}$ gerade der Inhalt des Registers beim Schritt Nummer i (in umgekehrter Reihenfolge) – d. h., die Analyse konzentriert sich nicht auf den Output, sondern setzt bei den Zuständen an. Die Rekursionsformel (8.9) kann man dann für $n \geq l$ in Matrixform als

$$\begin{pmatrix} u_{n-l+1} \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ s_l & s_{l-1} & \ldots & s_1 \end{pmatrix} \begin{pmatrix} u_{n-l} \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix}$$

schreiben oder sehr knapp und übersichtlich in der Form (mit substituierten Indizes $m = n-l+1$)

$$u_{(m)} = S \cdot u_{(m-1)} \quad \text{für } m \geq 1\,,$$

[99] Die „Rückkopplung“ wird in der Bezeichnungsweise meistens weggelassen, wenn keine Missverständnisse zu befürchten sind, ist hier aber implizit immer mit gemeint.

wobei S die Koeffizientenmatrix ist. Noch einen Schritt weiter gehend fasst man l aufeinanderfolgende Zustandsvektoren $u_{(i)}, \ldots, u_{(i+l-1)}$ zu einer Zustandsmatrix

$$U_{(i)} = \begin{pmatrix} u_i & u_{i+1} & \ldots & u_{i+l-1} \\ u_{i+1} & u_{i+2} & \ldots & u_{i+l} \\ \vdots & \vdots & \ddots & \vdots \\ u_{i+l-1} & u_{i+l} & \ldots & u_{2l-2} \end{pmatrix}$$

zusammen, setzt $U = U_{(0)}$, $V = U_{(1)}$, und erhält damit die Formel

$$V = S \cdot U,$$

die die unbekannten Koeffizienten $s_1, \ldots, s_l$ durch die bekannten Klartextbits $u_0, \ldots, u_{2l-1}$ ausdrückt. Vor allem aber gestattet sie, als Lohn für die Bemühungen sofort die Lösung hinzuschreiben – vorausgesetzt, die Matrix U ist invertierbar:

$$S = V \cdot U^{-1},$$

denn aus der Matrix S kann man ja die Koeffizienten s_j auslesen. Mehr zur Invertierbarkeit später.

Beispiel

Wir haben einen Geheimtext vorliegen:

```
10011100 10100100 01010110 10100110 01011101 10101110
01100101 10000000 00111011 10000010 11011001 11010111
00110010 11111110 01010011 10000010 10101100 00010010
11000110 01010101 00001011 11010011 01111011 10110000
10011111 00100100 00001111 01010011 11111101
```

Wir vermuten, dass er XOR-chiffriert ist mit einem Schlüsselstrom, der durch ein lineares Schieberegister der Länge $l = 16$ erzeugt wurde. Der Kontext lässt vermuten, dass der Text mit dem Wort „Treffpunkt“ beginnt. Zum Brechen benötigen wir nach der Theorie 32 Bits, also nur die ersten vier Buchstaben. Daraus ermitteln wir 32 Bits des Schlüsselstroms:

```
01010100 01110010 01100101 01100110 = T r e f
10011100 10100100 01010110 10100110   cipher bits
-------- -------- -------- --------
11001000 11010110 00110011 11000000   key bits
```

Im SageMath-Beispiel 8.24 wird die Koeffizientenmatrix bestimmt. Die letzte Zeile sagt uns, dass die $s_i = 0$ sind außer $s_{16} = s_5 = s_3 = s_2 = 1$.

Damit kennen wir das Schieberegister und den Startwert, können den gesamten Schlüsselstrom berechnen – ja, es ist der aus Abbildung 8.20 – und damit den Klartext herstellen (der übrigens nicht ganz so beginnt wie vermutet).

Beweis des Satzes

Für den Fall, dass die Zustandsmatrix $U = U_{(0)}$ invertierbar ist, haben wir oben schon gezeigt, dass die Koeffizienten eindeutig bestimmbar sind. Insbesondere ist dann das Schieberegister

SageMath-Beispiel 8.24 Bestimmung einer Koeffizientenmatrix

```
sage: l = 16
sage: kbits =
      [1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,1,1,1,0,0,0,0,0,0]
sage: ulist = []
sage: for i in range(0,l):
        state = kbits[i:(l+i)]
        ulist.append(state)
sage: U = matrix(GF(2),ulist)
sage: det(U)
1
sage: W = U.inverse()
sage: vlist = []
sage: for i in range(1,l+1):
        state = kbits[i:(l+i)]
        vlist.append(state)
sage: V = matrix(GF(2),vlist)
sage: S = V*W
sage: S
[0 1 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 0 1 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 0 1 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 0 1 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 0 1 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 0 1 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 0 1 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 0 1]
[1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0]
```

bekannt und aller weitere Output vorhersagbar. Wir müssen uns also noch mit der Möglichkeit auseinandersetzen, dass die Matrix U nicht invertierbar sein könnte.

Falls einer der ersten l Zustandsvektoren (= Zeilen der Matrix U) Null ist, sind auch alle weiteren Null, die Vorhersage ist also trivial.

Wir können also annehmen, dass diese Vektoren alle nicht Null, aber linear abhängig sind. Dann gibt es einen kleinsten Index $k \geq 1$, so dass $u_{(k)}$ in dem von $u_{(0)}, \ldots, u_{(k-1)}$ aufgespannten Unterraum liegt. D. h., es gibt Koeffizienten $t_1, \ldots, t_k \in \mathbb{F}_2$ mit

$$u_{(k)} = t_1 u_{(k-1)} + \cdots + t_k u_{(0)}.$$

Dann gilt aber auch $u_{(k+1)} = S \cdot u_{(k)} = t_1 S \cdot u_{(k-1)} + \cdots + t_k S \cdot u_{(0)} = t_1 u_{(k)} + \cdots + t_k u_{(1)}$ und

weiter durch Induktion

$$u_{(n)} = t_1 u_{(n-1)} + \cdots + t_k u_{(n-k)} \quad \text{für alle } n \geq k.$$

Durch diese Formel werden also auch alle weiteren Bits vorhergesagt.

Die Aussage über den Aufwand folgt aus Satz 8.1.10.

Diskussion:

- Diese Überlegung ergibt für den Fall einer nicht invertierbaren Zustandsmatrix ein kürzeres lineares Schieberegisters (der Länge $k < l$), das die gleiche Folge erzeugt. In diesem Fall werden die Koeffizienten des originalen Registers also nicht bestimmt, aber die weitere Folge trotzdem korrekt vorhergesagt.
- Wenn nicht die ersten Bits, sondern $2l$ zusammenhängende Bits an späterer Stelle bekannt sind, ergibt der Satz zunächst nur eine Berechnung der späteren Bits. Im Regelfall, wo die Zustandsmatrix U invertierbar ist, ist das Register aber dann völlig bekannt und kann natürlich auch rückwärts zur Bestimmung der Vorgängerbits eingesetzt werden. Ist die Zustandsmatrix nicht invertierbar, kann man das gleiche mit dem eben konstruierten kürzeren linearen Schieberegister erreichen.
- Etwas komplizierter wird die Situation, wenn man zwar $2l$ Bits des Schlüsselstroms kennt, diese aber nicht zusammenhängen. Auch dann erhält man lineare Relationen, in denen aber zusätzlich unbekannte Zwischenbits vorkommen. Ist m die Zahl der Lücken, so hat man dann insgesamt $l + m$ lineare Gleichungen für $l + m$ unbekannte Bits.
- Was aber, wenn auch die Länge l des Registers unbekannt ist? Das Durchprobieren aller Werte $l = 1, 2, 3, \ldots$ ist sicher lästig, aber machbar. Es gibt aber auch den Algorithmus von Berlekamp-Massey[100], der ohne vorherige Kenntnis von l sehr effizient ist. Dessen Behandlung würde aber hier zu weit führen.

Fazit

Kryptoanalyse bedeutet für Zufallsgeneratoren wie in Abbildung 8.20, aus einem Teil ihres Outputs eine der folgenden Informationen zu bestimmen:

- die geheimen Parameter,
- den Startwert,
- weitere Teile des Outputs („Vorhersageproblem“).

Wie wir bei den linearen Schieberegistern gesehen haben, ist es realistisch, das Vorhersageproblem anzugehen, das auch lösbar sein kann, wenn die Bestimmung der internen Parameter nicht gelingt. Wir halten fest:

> *Die Kryptoanalyse eines Zufallsgenerators bedeutet in erster Linie die Lösung des Vorhersageproblems. Ein Zufallsgenerator ist kryptographisch sicher, wenn sein Vorhersageproblem nicht effizient lösbar ist.*
>
> *Lineare Schieberegister sind kryptographisch nicht sicher.*

[100] in SageMath als `sage.crypto.lfsr.berlekamp_massey` enthalten, in CrypTool 2 unter „Kryptoanalyse“/„Generisch“/„Berlekamp-Massey-Algorithmus“ zu finden

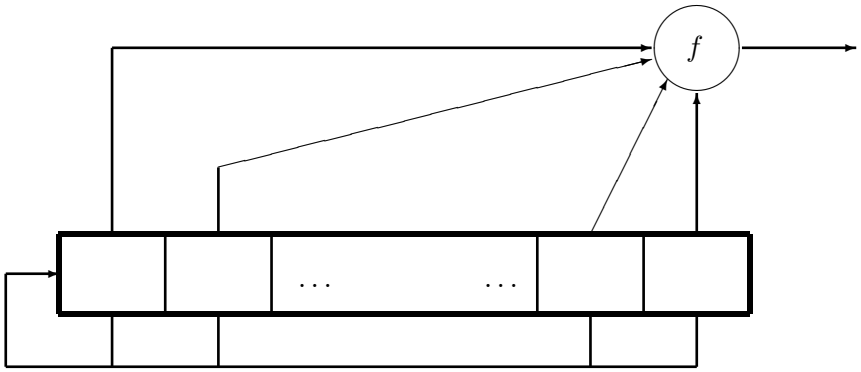


Abbildung 8.25: Nichtlinearer Ausgabefilter für ein lineares Schieberegister

8.3.5 Nichtlinearität für Schieberegister – Ansätze

Lineare Schieberegister sind beliebt – vor allem bei Elektro-Ingenieuren und beim Militär – denn sie sind

- sehr einfach zu realisieren,
- extrem effizient in Hardware,
- als Zufallsgeneratoren für statistische Zwecke sehr gut geeignet,
- problemlos in großer Anzahl parallel zu betreiben,
- aber leider kryptologisch völlig unsicher.

Um die positiven Eigenschaften zu nutzen und die kryptologische Schwäche zu vermeiden, gibt es verschiedene Ansätze.

Ansatz 1: Nichtlineare Rückkopplung

Die nichtlineare Rückkopplung folgt dem Schema aus Abbildung 8.21 mit einer nichtlinearen Booleschen Funktion f. Sie wird hier nicht weiter behandelt; ein ganz einfaches, kryptographisch ungeeignetes Beispiel war das SageMath-Beispiel 8.21. Man kann aber auch allgemein zeigen, dass solche nichtlinearen Schieberegister (nonlinear feedback, NLFSR[101]), für sich allein genommen, unter realistischen Annahmen im praktischen Einsatz nicht kryptographisch sicher sind [Pom16].

Ansatz 2: Nichtlinearer Ausgabefilter

Der nichtlineare Ausgabefilter (Non-Linear Feedforward) folgt dem Schema aus Abbildung 8.25. Das Schieberegister selbst ist linear. Der nichtlineare Ausgabefilter ist ein Spezialfall des nächsten Ansatzes.

[101] für Non-Linear Feedback Shift Register

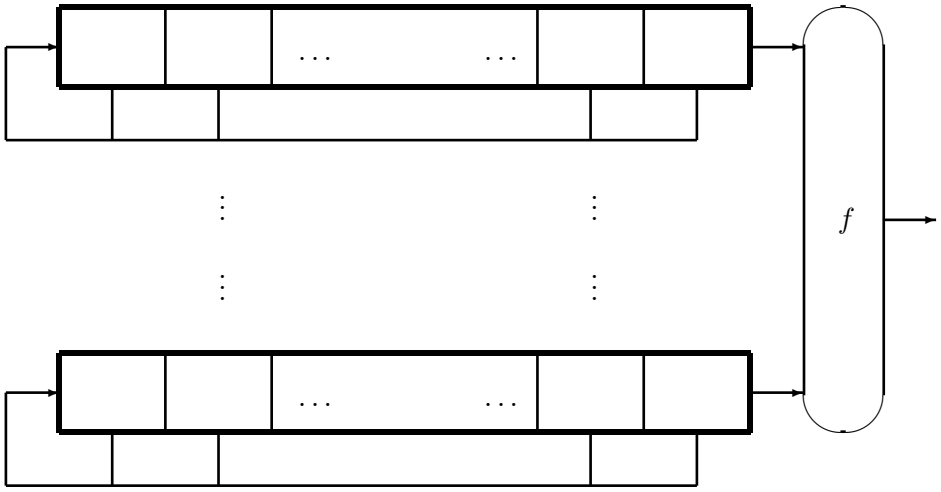


Abbildung 8.26: Nichtlinearer Kombinierer

Ansatz 3: Nichtlinearer Kombinierer

Hier wird eine „Batterie“ aus n linearen Schieberegistern – die durchaus unterschiedliche Länge haben können und sollen – parallel betrieben. Ihre Outputfolgen werden in eine Boolesche Funktion $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ gefüttert[102], siehe Abbildung 8.26. Ein Ansatz zur Analyse dieses Verfahrens folgt in Abschnitt 8.3.7.

Ansatz 4: Auswahlsteuerung/Dezimierung/Taktung

Weitere Möglichkeiten bestehen in verschiedenen Methoden zur Steuerung einer Batterie von n parallel betriebenen linearen Schieberegistern durch ein weiteres lineares Schieberegister:

- Bei der **Auswahlsteuerung** wird je nach Zustand des „Hilfsregisters“ das aktuelle Output-Bit von genau einem der „Batterie-Register“ als Output des Zufallsgenerators ausgewählt. Allgemeiner kann man auch eine Auswahl „r aus n“ treffen.
- Bei der **Dezimierung** nimmt man im allgemeinen $n = 1$ an und gibt das Output-Bit des einen Batterie-Registers nur dann aus, wenn das Hilfsregister einen bestimmten Zustand hat. Diese Art der Dezimierung kann man natürlich analog auf jede Bitfolge anwenden.
- Bei der **Taktung** gibt der Zustand des Hilfsregisters an, welche der Batterie-Register im aktuellen Taktzyklus weitergeschoben werden (und um wieviele Positionen) und welche in ihrem momentanen Zustand bleiben. Das ist vergleichbar mit der Steuerlogik von Rotor-Maschinen.

Diese Ansätze lassen sich oft bequem auch als nichtlineare Kombinierer schreiben, so dass Ansatz 3 als der gängigste Ansatz zur Rettung der linearen Schieberegister angesehen werden kann.

Der Mobilfunk-Verschlüsselungsstandard A5/1 verwendet drei leicht unterschiedlich getaktete lineare Schieberegister der Längen 19, 22 und 23 mit jeweils maximaler Periode, deren Outputströme linear (nämlich einfach durch binäre Addition) kombiniert werden. Bei A5/2 –

[102] daher auch hierfür gelegentlich die Bezeichnung Non-Linear Feedforward

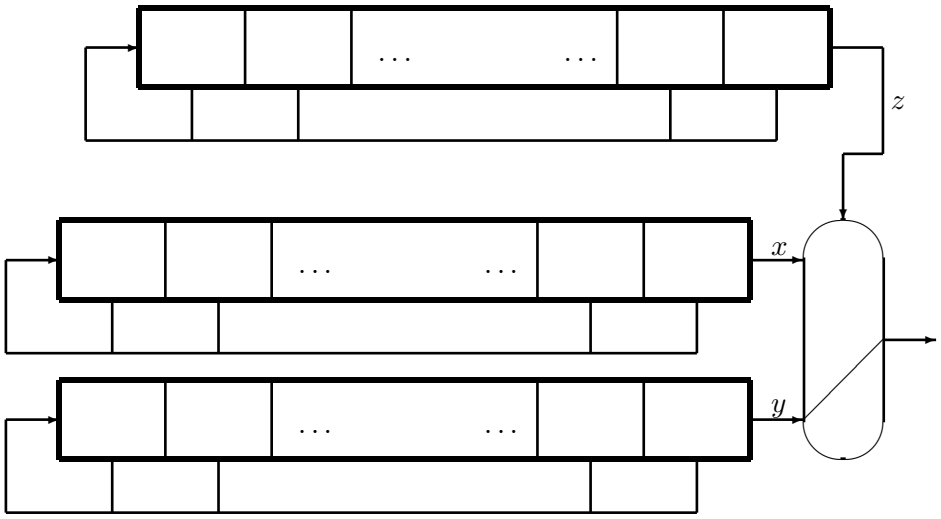


Abbildung 8.27: Geffe-Generator

das noch schwächer ist – wird die Taktung durch ein Hilfsregister geregelt. Beide Varianten lassen sich auf handelsüblichen PCs in Echtzeit brechen.

Der Bluetooth-Verschlüsselungsstandard E_0 verwendet vier lineare Schieberegister, die nichtlinear kombiniert werden. Dieses Verfahren ist etwas stärker als A5, aber auch zu schwach für echte Sicherheit [Sch16].

Beispiel: Der Geffe-Generator

Das einfachste Beispiel für die Auswahlsteuerung ist der Geffe-Generator, der durch das Schema in Abbildung 8.27 beschrieben wird. Die Ausgabe ist x, wenn $z = 0$, und y, wenn $z = 1$. Das kann man so als Formel ausdrücken:

$$\begin{aligned} u &= \begin{cases} x, & \text{wenn } z = 0, \\ y, & \text{wenn } z = 1 \end{cases} \\ &= (1-z)x + zy = x + zx + zy. \end{aligned}$$

Also lässt sich der Geffe-Generator auch durch einen nichtlinearen Kombinierer mit einer Booleschen Funktion $f\colon \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2$ vom Grad 2 beschreiben. Diese wird zur späteren Verwendung im SageMath-Beispiel 8.25 erzeugt.

8.3.6 Implementation eines nichtlinearen Kombinierers

Ein nichtlinearer Kombinierer benötigt mehrere parallel betriebene lineare Schieberegister. Dies legt nahe, diese als Objekte zu implementieren, d. h., eine Klasse `LFSR` zu definieren[103].

Klasse `LSFR`:

Attribute:

- `length`: die Länge des Registers

[103] siehe auch in CrypTool 2 unter „Protokolle“/„LFSR“ bzw. „NLFSR“

SageMath-Beispiel 8.25 Die Geffe-Funktion

```
sage: geff = BoolF(str2bbl("00011100"),method="ANF")
sage: geff.printTT()
Value at 000 is 0
Value at 001 is 0
Value at 010 is 0
Value at 011 is 1
Value at 100 is 1
Value at 101 is 0
Value at 110 is 1
Value at 111 is 1
```

- `taplist`: die Liste der Koeffizienten („Taps“[104]), die die rückzukoppelnden Bits definieren (konstant)
- `state`: der Zustand des Registers (veränderbar)

Methoden:

- `setLength`: Definition der Länge (nur implizit bei der Initialisierung verwendet)
- `setTaps`: Besetzung der Tap- (= Koeffizienten-) Liste (nur implizit bei der Initialisierung verwendet)
- `setState`: Belegung des Registers mit einem Zustand
- `getLength`: Ausgabe der Länge
- `nextBits`: Erzeugung einer vorgegebenen Anzahl von Ausgabebits und kontinuierliche Weiterschaltung des Zustands

Dazu ist es zur Überwachung des Registers praktisch, noch eine Methode (die in Python generisch `__str__` heißt) zu haben, die die Attribute in lesbarer Form ausgibt.

Für die Implementation siehe das SageMath-Beispiel 8.51 im Abschnitt 8.4.9.

Beispiel: Geffe-Generator

Zunächst wählen wir drei lineare Schieberegister der Längen 15, 16 und 17, deren Perioden[105] $2^{15}-1 = 32767$, $2^{16}-1 = 65535$ und $2^{17}-1 = 131071$ sind, und diese sind paarweise teilerfremd, siehe SageMath-Beispiel 8.26. Fasst man ihren Output in jedem Takt zu Bitblöcken der Länge 3 zusammen, so hat diese Folge eine Periode der eindrucksvollen Länge 281459944554495, also knapp 300×10^{12} (300 Billionen[106]). Die drei Register werden im SageMath-Beispiel 8.27 definiert; die Rekursionsformel für das dritte davon, das Steuerungsregister `reg17`, ist z. B. $u_n = u_{n-3} + u_{n-17}$, da genau die Taps 3 und 17 gesetzt sind. Mit jedem von ihnen wird eine Folge der Länge 100 erzeugt, siehe SageMath-Beispiel 8.28. Diese werden im SageMath-Beispiel 8.29 mithilfe der Geffe-Funktion kombiniert.

[104] auf deutsch etwa „Abzweig“ oder „Abgriff“, weil bei einer Hardware-Implementierung genau an diesen Stellen die für die Rückkopplung verwendeten Bits abgegriffen werden

[105] nach den Listen primitiver Polynome in [MvOV01]

[106] europäische Billionen. Amerikanisch wären das 300 Trillionen.

SageMath-Beispiel 8.26 Eine Periodenberechnung

```
sage: n15 = 2**15 - 1; n15
32767
sage: n15.factor()
7 * 31 * 151
sage: n16 = 2**16 - 1; n16
65535
sage: n16.factor()
3 * 5 * 17 * 257
sage: n17 = 2**17 - 1; n17
131071
sage: n17.factor()
131071
sage: period = n15 * n16 * n17; period
281459944554495
```

SageMath-Beispiel 8.27 Drei lineare Schieberegister

```
sage: reg15 = LFSR([1,0,0,0,0,0,0,0,0,0,0,0,0,0,1])
sage: reg15.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1])
sage: print(reg15)
Length: 15 | Taps: 100000000000001 | State: 011010110001001
sage: reg16 = LFSR([0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1])
sage: reg16.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1])
sage: print(reg16)
Length: 16 | Taps: 0110100000000001 | State: 0110101100010011
sage: reg17 = LFSR([0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1])
sage: reg17.setState([0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1,1])
sage: print(reg17)
Length: 17 | Taps: 00100000000000001 | State: 01101011000100111
```

SageMath-Beispiel 8.28 Drei LFSR-Folgen

```
sage: nofBits = 100
sage: outlist15 = reg15.nextBits(nofBits)
sage: print(outlist15)
[1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0,
 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,
 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1]
sage: outlist16 = reg16.nextBits(nofBits)
sage: print(outlist16)
[1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1,
 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0,
 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1,
 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1]
sage: outlist17 = reg17.nextBits(nofBits)
sage: print(outlist17)
[1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1,
 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0,
 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0]
```

SageMath-Beispiel 8.29 Die kombinierte Folge

```
sage: outlist = []
sage: for i in range(0,nofBits):
....:     x = [outlist15[i],outlist16[i],outlist17[i]]
....:     outlist.append(geff.valueAt(x))
....:
sage: print(outlist)
[1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1,
 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1,
 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
```

8.3.7 Korrelationsattacken – die Achillesferse der Kombinierer

Sei $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ die Kombinierfunktion eines nichtlinearen Kombinierers. Die Anzahl

$$K_f := \#\{x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n \mid f(x) = x_1\}$$

gibt an, wie oft der Funktionswert mit dem ersten Argument übereinstimmt. Ist sie $> 2^{n-1}$, so ist die Wahrscheinlichkeit für diese Übereinstimmung,

$$p = \frac{1}{2^n} \cdot K_f > \frac{1}{2},$$

also überdurchschnittlich. Die kombinierte Outputfolge „korreliert“ also stärker mit dem Output des ersten linearen Schieberegisters, als zufällig zu erwarten wäre. Ist $p < \frac{1}{2}$, so weicht die Korrelation nach unten vom zufälligen Wert ab.

Diesen Effekt kann sich die Kryptoanalytikerin bei einem Angriff mit bekanntem Klartext zunutze machen. Angenommen wird, dass ihr die „Hardware“, also die Rekursionsformeln für die Register (die Taps) und auch die Kombinierfunktion f, bekannt ist. Gesucht sind die als Schlüssel betrachteten Startvektoren aller Register. Die Bits $k_0, \ldots, k_{r-1}$ des Schlüsselstroms[107] seien bekannt. Mit einer Exhaustion über die 2^{l_1} Startvektoren des ersten Registers erzeugt man jedesmal die Folge $u_0, \ldots, u_{r-1}$ und zählt die Übereinstimmungen. Zu erwarten ist

$$\frac{1}{r} \cdot \#\{i \mid u_i = k_i\} \approx \begin{cases} p & \text{beim richtigen Startvektor,} \\ \frac{1}{2} & \text{sonst.} \end{cases}$$

Falls r groß genug ist, kann man also den echten Startvektor des ersten Registers mit einem Aufwand $\sim 2^{l_1}$ (mit hoher Wahrscheinlichkeit) bestimmen. Macht man dann mit den anderen Registern genauso weiter, gelingt die Identifikation des gesamten Schlüssels mit einem Aufwand $\sim 2^{l_1} + \cdots + 2^{l_n}$. Das ist zwar exponenziell, aber wesentlich geringer als der Aufwand $\sim 2^{l_1} \cdots 2^{l_n}$ für die naive vollständige Schlüsselsuche.

In der Sprache der linearen Kryptoanalyse aus 8.2.6 haben wir hier die lineare Relation

$$f(x_1, \ldots, x_n) \overset{p}{\approx} x_1$$

[107] der Einfachheit der Darstellung halber die ersten – für irgendwelche r bekannten Schlüsselbits funktioniert die Argumentation genauso.

x	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1
$f(x,y,z)$	0	0	0	1	1	0	1	1

Tabelle 8.21: Wahrheitstafel der Geffe-Funktion (waagerecht angeordnet)

Linearform	0	z	y	$y+z$	x	$x+z$	$x+y$	$x+y+z$
Repräsentation	000	001	010	011	100	101	110	111
Potenzial	0	0	1/4	1/4	1/4	1/4	0	0
Wahrsch. p	1/2	1/2	3/4	1/4	3/4	3/4	1/2	1/2

Tabelle 8.22: Korrelationswahrscheinlichkeiten der Geffe-Funktion

für f ausgenutzt. Klar ist, dass man analog jede lineare Relation ausnutzen kann, um die Komplexität der vollständigen Schlüsselsuche zu reduzieren[108].

Korrelationen des Geffe-Generators

Für den Geffe-Generator kann man die Korrelationen aus der Wahrheitstafel, Tabelle 8.21, ablesen: Als Wahrscheinlichkeit für die Übereinstimmung erhält man also

$$p = \begin{cases} \frac{3}{4} & \text{für das Register 1 } (x), \\ \frac{3}{4} & \text{für das Register 2 } (y), \\ \frac{1}{2} & \text{für das Register 3 } (z = \text{Steuerung}). \end{cases}$$

Daher lassen sich bei einer Korrelationsattacke die Startwerte für die Register 1 und 2 – die Batterieregister – leicht schon aus kurzen Outputfolgen bestimmen; den Startwert für Register 3, das Steuerungsregister, findet man dann auch leicht durch Exhaustion.

Diese Schwachstelle des Geffe-Generators wird im SageMath-Beispiel 8.30 nachvollzogen, das als Fortsetzung des SageMath-Beispiels 8.25 einzugeben ist. Da wir das lineare Profil nur in der Klasse `BoolMap` definiert haben, müssen wir zuerst die Funktion `geff` als Boolesche Abbildung interpretieren – also als Liste der Länge 1 von Booleschen Funktionen. Das lineare Profil wird als Matrix mit 2 Spalten und 8 Zeilen gedacht. Die erste Spalte `[64, 0, 0, 0, 0, 0, 0, 0]` misst die Übereinstimmung mit der Linearform 0 des Bildbereichs. Sie enthält also keine nennenswerte Information, außer dass alles durch 64 zu dividieren ist. Die zweite Spalte `[0, 0, 16, 16, 16, 16, 0, 0]` wird (nach dieser Division) in der Tabelle 8.22 als Liste der Korrelationswahrscheinlichkeiten p interpretiert. Dabei wird die Formel

$$p = \frac{1}{2} \cdot (\pm\sqrt{\lambda} + 1)$$

verwendet. Ist $\lambda = 0$, so $p = 1/2$. Ist $\lambda = 1/4$, so $p = 1/4$ oder $3/4$. Die Entscheidung zwischen diesen beiden Werten für p kann man anhand der Tabelle 8.21 treffen.

Im SageMath-Beispiel 8.31 wird diese Erkenntnis auf die mit dem Geffe-Generator erzeugte Folge der Länge 100 angewendet. Zur Zählung der Koinzidenzen (= Übereinstimmungen) wird

[108] Eine genauere Analyse der Situation führt auf den Begriff der Korrelationsimmunität, die mit dem linearen Potenzial verwandt ist.

SageMath-Beispiel 8.30 Lineares Profil der Geffe-Funktion

```
sage: g = BoolMap([geff])
sage: linProf = g.linProf(); linProf
[[64, 0], [0, 0], [0, 16], [0, 16], [0, 16], [0, 16], [0, 0], [0, 0]]
```

die Funktion `coinc` aus dem SageMath-Beispiel 8.39 (im Anhang) verwendet. Mit dem ersten Register gibt es 73, mit dem zweiten 76 Koinzidenzen, mit dem dritten dagegen nur 41. Das passt sehr gut zu den im Rahmen statistischer Schwankungen theoretisch erwarteten Werten 75, 75, 50.

SageMath-Beispiel 8.31 Koinzidenzen des Geffe-Generators

```
sage: coinc(outlist15,outlist)
73
sage: coinc(outlist16,outlist)
76
sage: coinc(outlist17,outlist)
41
```

Analyse des Geffe-Generators

Diese deutlichen Ergebnisse legen nahe, dass die Analyse der beispielhaft erzeugten Folge leicht sein sollte. Für eine grobe Erfolgsabschätzung kann man auf mathematische Strenge verzichten.

Betrachtet wird ein fest vorgegebener Bitblock $b \in \mathbb{F}_2^r$. Wir fragen zunächst, wie groß die Wahrscheinlichkeit für einen zufälligen Bitblock $u \in \mathbb{F}_2^r$ ist, an genau t Stellen mit b übereinzustimmen, also t Koinzidenzen zu haben. Das ist genau die Fragestellung der symmetrischen Binomialverteilung (also mit $p = \frac{1}{2}$ als Wahrscheinlichkeit einer einzelnen Übereinstimmung): Die Wahrscheinlichkeit für genau t Koinzidenzen ist

$$B_{r,\frac{1}{2}}(t) = \frac{\binom{r}{t}}{2^r}.$$

Die Wahrscheinlichkeit für bis zu T Koinzidenzen ist also

$$\sum_{t=0}^{T} B_{r,\frac{1}{2}}(t) = \frac{1}{2^r} \cdot \sum_{t=0}^{T} \binom{r}{t}.$$

Wenn r nicht zu groß ist, kann man diesen Wert für eine konkrete Schranke T explizit ausrechnen. Wenn r nicht zu klein ist, approximiert man ihn mithilfe der Normalverteilung. Dazu benötigt man den Erwartungswert für die Anzahl der Koinzidenzen, der $r/2$ ist, die Varianz $r/4$ und die Standardabweichung $\sqrt{r}/2$.

Wie auch immer man das macht, im Fall $r = 100$ ist (exemplarisch) die Wahrscheinlichkeit, maximal 65 Koinzidenzen zu finden, ziemlich genau $0,999$, die Überschreitungswahrscheinlichkeit also 1 ‰. Die Exhaustion der Startwerte des Registers 1 umfasst $2^{15} = 32786$ Möglichkeiten (den eigentlich ausgeschlossenen Startwert $0 \in \mathbb{F}_2^{15}$ zählen wir großzügig mit). Dabei können wir also etwa 33 „Grenzüberschreitungen“ mit mindestens 66 Koinzidenzen erwarten. Darunter sollte der wahre Startwert von Register 1 sein, der etwa 75 Koinzidenzen produzieren sollte und sich vielleicht sogar durch das Maximum der Koinzidenzen verrät.

Das SageMath-Beispiel 8.32 zeigt, dass das tatsächlich so ist. Das Maximum der Koinzidenzen, 73, ist im Histogramm allerdings zweimal vertreten. Zum ersten Mal tritt es beim Index 13705, also beim Startwert 011010110001001 auf, den wir damit korrekt identifiziert haben. Das zweite Auftreten, im SageMath-Beispiel 8.33 ermittelt, liefert das falsche Ergebnis 111100110001011, das letztlich durch Ausprobieren ausgeschieden werden muss.

SageMath-Beispiel 8.32 Analyse des Geffe-Generators – Register 1

```
sage: clist = []
sage: histogr = [0] * (nofBits + 1)
sage: for i in range(0,2**15):
....:     start = int2bbl(i,15)
....:     reg15.setState(start)
....:     testlist = reg15.nextBits(nofBits)
....:     c = coinc(outlist,testlist)
....:     histogr[c] += 1
....:     clist.append(c)
....:
sage: print(histogr)
[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, 4, 12, 12, 37, 78, 116, 216,
 329, 472, 722, 1003, 1369, 1746, 1976, 2266, 2472, 2531, 2600,
 2483, 2355, 2149, 1836, 1574, 1218, 928, 726, 521, 343, 228, 164,
 102, 60, 47, 36, 13, 8, 7, 4, 2, 1, 2, 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]
sage: mm = max(clist)
sage: ix = clist.index(mm)
sage: block = int2bbl(ix,15)
sage: print "Maximum =", mm, "at index", ix, ", start value", block
Maximum = 73 at index 13705 , start value\
 [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
```

SageMath-Beispiel 8.33 Analyse des Geffe-Generators – Fortsetzung

```
sage: ix = clist.index(mm,13706); ix
31115
sage: print int2bbl(ix,15)
[1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1]
```

Die analoge Analyse von Register 2 wird im SageMath-Beispiel 8.34 durchgeführt. Hier ist das Maximum der Koinzidenzen, 76, tatsächlich deutlich herausgehoben. Es tritt beim Index 27411, also beim Startwert 0110101100010011 auf, den wir damit ebenfalls korrekt identifiziert haben.

SageMath-Beispiel 8.34 Analyse des Geffe-Generators – Register 2

```
sage: clist = []
sage: histogr = [0] * (nofBits + 1)
sage: for i in range(0,2**16):
....:     start = int2bbl(i,16)
....:     reg16.setState(start)
....:     testlist = reg16.nextBits(nofBits)
....:     c = coinc(outlist,testlist)
....:     histogr[c] += 1
....:     clist.append(c)
....:
sage: print(histogr)
[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, 2, 3, 4, 8, 17, 25, 51, 92, 171,
 309, 477, 750, 1014, 1423, 1977, 2578, 3174, 3721, 4452, 4821,
 5061, 5215, 5074, 4882, 4344, 3797, 3228, 2602, 1974, 1419,
 1054, 669, 434, 306, 174, 99, 62, 38, 19, 10, 3, 0, 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]
sage: mm = max(clist)
sage: ix = clist.index(mm)
sage: block = int2bbl(ix,16)
sage: print "Maximum =", mm, "at index", ix, ", start value", block
Maximum = 76 at index 27411 , start value\
 [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1]
```

$u_{17} = u_{14} + u_0$	$0 = 1 + u_0$	$u_0 = 1$	
$u_{19} = u_{16} + u_2$	$1 = 0 + u_2$	$u_2 = 1$	
$u_{20} = u_{17} + u_3$	$u_{20} = 0 + 0$	$u_{20} = 0$	
$u_{22} = u_{19} + u_5$	$u_{22} = u_5 + 1$	$u_5 = u_{22} + 1$	
$u_{23} = u_{20} + u_6$	$0 = u_{20} + u_6$	$u_6 = u_{20}$	$u_6 = 0$
$u_{25} = u_{22} + u_8$	$u_{25} = u_{22} + u_8$	$u_8 = u_{22} + u_{25}$	$u_8 = u_{22}$
$u_{27} = u_{24} + u_{10}$	$u_{27} = 0 + 1$	$u_{27} = 1$	
$u_{28} = u_{25} + u_{11}$	$0 = u_{25} + 0$	$u_{25} = 0$	
$u_{30} = u_{27} + u_{13}$	$u_{30} = u_{27} + u_{13}$	$u_{13} = u_{27} + u_{30}$	$u_{13} = u_{30} + 1$
$u_{33} = u_{30} + u_{16}$	$u_{33} = u_{30} + 0$	$u_{30} = u_{33}$	$u_{30} = 1$
$u_{36} = u_{33} + u_{19}$	$0 = u_{33} + 1$	$u_{33} = 1$	
$u_{39} = u_{36} + u_{22}$	$u_{39} = 0 + u_{22}$	$u_{22} = u_{39}$	
$u_{42} = u_{39} + u_{25}$	$0 = u_{39} + u_{25}$	$u_{39} = u_{25}$	$u_{39} = 0$

Tabelle 8.23: Bestimmung des Steuerungsregisters

Zur vollständigen Analyse ist jetzt noch der Startwert von Register 3, dem Steuerungsregister, zu bestimmen. Das könnte durch Exhaustion über die 2^{17} verschiedenen Möglichkeiten geschehen. Man kann das deutlich verkürzen, denn von den ersten 100 Bits des Steuerungsregisters sind 51 bereits bekannt: Nur wenn die Werte von Register 1 und 2 übereinstimmen, ist das entsprechende Bit des (Steuerungs-) Registers 3 unbestimmt. Sind sie aber verschieden, so ist das Bit 0, wenn der Gesamt-Output mit Register 1 übereinstimmt, und sonst 1.

```
Register 1: 10010001101011011100001001101101000001110110110000
Register 2: 11001000110101100011001111000000001110111000111000
Register 3: -1-00--0-1101-110001---00-1-00-1--1101--110---0---
Bitfolge:   11010001110001101101001001001100001100111010110000

        ... 00101101101111111001001001010101110001110011001011
        ... 00100011101111010010011110010110111100101110010001
        ... ----110-------1-1-11-0-100----01--01-1-001-1-00-1-
        ... 00100001101111010010001101010100110100110110001001
```

Insbesondere sind 11 der 17 Bits des Startwerts schon bekannt und daher nur noch $2^6 = 64$ Möglichkeiten durchzuprobieren.

Aber auch das geht noch einfacher, da zwischen den bekannten und den unbekannten Bits lineare Relationen der Art $u_n = u_{n-3} + u_{n-17}$ bestehen. Unbekannt von der Startbelegung sind die Bits u_0, u_2, u_5, u_6, u_8, u_{13}. Ihre Berechnung folgt spaltenweise der Tabelle 8.23, in der schon $u_0 = 1$, $u_2 = 1$ und $u_6 = 0$ abzulesen sind. Die übrigen Ergebnisse liefern $u_8 = u_{22} = u_{39} = 0$, $u_5 = u_{22} + 1 = u_8 + 1 = 1$ und $u_{13} = u_{30} + 1 = 0$. Der Startwert des Steuerungsregisters ist also als `01101011000100111` (und somit korrekt) bestimmt. Eigentlich müssten wir jetzt noch die zweite mögliche Lösung für das Register 1 durchprobieren, aber da in der jetzt bestimmten Konstellation die Folge korrekt reproduziert wird, ist das überflüssig.

8.3.8 Design-Kriterien für nichtlineare Kombinierer

Aus der bisherigen Diskussion lassen sich als Design-Kriterien für nichtlineare Kombinierer herleiten:

- Die einzelnen Batterieregister müssen möglichst lang sein.
- Die Kombinierfunktion f soll ein möglichst geringes lineares Potenzial haben.

Wie lang sollen die Batterieregister sein? Es gibt verschiedene Ansätze zu „schnellen“ Korrelationsattacken, z. B. mit Hilfe der Walsh-Transformation, besonders gegen dünn besetzte lineare Rückkopplungsfunktionen [MS89]. Diese reduzieren zwar nicht die Komplexitätsklasse des Angriffs („mindestens exponenziell in der Länge des kürzesten Registers“), aber der Aufwand wird um einen beträchtlichen Proportionalitätsfaktor verringert. Auf diese Weise werden Register angreifbar, deren Rückkopplungsfunktion in der ANF weniger als 100 Monome mit Koeffizienten 1 enthält. Folgerung:

- Die einzelnen linearen Schieberegister sollten mindestens 200 Bits lang sein und eine „dicht besetzte“ Rückkopplung besitzen.

Für die Anzahl n der zu kombinierenden linearen Schieberegister muss man beachten, dass die Kombinationsfunktion möglichst „korrelationsimmun“ sein, insbesondere ein möglichst geringes lineares Potenzial haben soll. Hier sollte man mit einer Booleschen Funktion von 16 Variablen schon gut hinkommen[109].

Ein eleganter Ausweg, der die Korrelationsattacke zusammenbrechen lässt, wurde von Rueppel vorgeschlagen: eine „zeitabhängige“ Kombinierfunktion, also eine Familie $(f_t)_{t\in\mathbb{N}}$ zu verwenden. D. h., zur Berechnung des Bits u_t des Schlüsselstroms wird die Kombinierfunktion f_t verwendet. Die Sicherheit dieses Ansatzes wird hier nicht weiter analysiert.

Man kann aber auch daran denken, dass die Korrelationsattacke darauf angewiesen ist, dass die Rückkopplungskoeffizienten, die Taps, bekannt sind. Sind sie das nicht, so muss auch für sie eine Exhaustion durchgeführt werden, was die Komplexität z. B. für das erste Schieberegister um einen weiteren Faktor 2^{l_1} vergrößert. In dieser Situation kann man die Anforderungen an die Länge der einzelnen Register etwas abmildern. Es sei aber daran erinnert, dass bei einer Hardware-Implementation die Rückkopplungskoeffizienten eher als Teil des Algorithmus und eher nicht als Teil des Schlüssels anzusehen sind, also im Sinne von Abbildung 8.20 zu den öffentlich bekannten externen Parametern zu rechnen sind.

Effizienz

Lineare Schieberegister und nichtlineare Kombinierer lassen sich mit speziell dafür gefertigter Hardware effizient realisieren, so dass pro Prozessortakt ein Bit herauspurzelt, durch Parallelisierung auch mehrere. Eine Aufwandsabschätzung für einen gängigen PC-Prozessor ist insofern etwas unfair. Hier müsste man jedes der 16 Register à ≥ 200 Bit auf 4 Stücke mit bis zu 64 Bit aufteilen, so dass allein das Weiterschieben eines der Register schon mindestens 4 Takte beansprucht, für 16 Register also 64 Takte. Selbst wenn die Kombinierfunktion ihre Aufgabe in einem Takt erledigt, hätten wir 65 Takte pro Bit benötigt, würden auf einem 2-GHz-Prozessor also bei maschinennaher Implementation und optimistischer Schätzung maximal $2\cdot 10^9/65 \approx 30$ Millionen Bits pro Sekunde produzieren.

Als Fazit kann man festhalten:

> *Mit linearen Schieberegistern und nichtlinearen Kombinierern lassen sich brauchbare, ziemlich schnelle Pseudozufallsgeneratoren aufbauen, besonders in Hardware.*

[109] Empfehlungen hierfür aus der Literatur sind nicht bekannt.

Für die kryptologische Sicherheit dieser Pseudozufallsgeneratoren gibt es zwar keine umfassende befriedigende Theorie und schon gar keinen mathematischen Beweis, aber durchaus eine plausible Absicherung, die – ähnlich wie bei Bitblock-Chiffren – mit der Nichtlinearität Boolescher Funktionen zu tun hat.

8.3.9 Perfekte Pseudozufallsgeneratoren

Anfang der 1980er Jahre entwickelte sich im Umkreis der asymmetrischen Kryptographie eine Vorstellung davon, wie man die Unvorhersagbarkeit eines Zufallsgenerators mathematisch modellieren könnte, nämlich im Rahmen der Komplexitätstheorie: Die Vorhersage soll nicht effizient möglich sein, d. h., auf ein bekanntes „hartes“ Problem zurückgeführt werden können. Dadurch wurde ein neuer Qualitätsstandard für Zufallsgeneratoren gesetzt, der allerdings letztlich auf der mathematisch völlig unbewiesenen Grundlage aufbaut, dass es für gewisse zahlentheoretische Probleme wie die Primzerlegung oder den diskreten Logarithmus keine effizienten Algorithmen gibt. – Die Situation ist also die gleiche wie bei der Sicherheit der asymmetrischen Verschlüsselung.

Interessanterweise stellte sich bald heraus, dass die scheinbar viel stärkere Forderung, die erzeugte Zufallsfolge solle sich durch *überhaupt keinen* effizienten Algorithmus von einer echten Zufallsfolge unterscheiden lassen, zur Unvorhersagbarkeit äquivalent ist, siehe Satz 8.3.2 (Satz von Yao). Dadurch ist die Bezeichnung „perfekt“ für die entsprechenden Zufallsgeneratoren gerechtfertigt. Insbesondere gibt es keinen effizienten statistischen Test, der in der Lage ist, eine Folge aus einem perfekten Zufallsgenerator von einer echt zufälligen Folge zu unterscheiden. Auf der theoretischen Seite ist damit ein sehr gutes Modell für Zufallsgeneratoren vorhanden, die statistisch absolut einwandfrei und kryptologisch unangreifbar sind. – Also:

> *Perfekte Zufallsgeneratoren sind kryptographisch sicher und statistisch nicht von echten Zufallsquellen zu unterscheiden.*
>
> *Es gibt vermutlich perfekte Zufallsgeneratoren, aber ein vollständiger mathematischer Beweis dafür steht noch aus.*

Die ersten konkreten Ansätze, von denen der BBS- (= Blum[110]-Blum[111]-Shub[112]-) Generator der bekannteste ist, lieferten Zufallsgeneratoren, die für den praktischen Einsatz (mit damaligen Prozessoren) meist zu langsam waren. Modifizierte Ansätze führten aber bald zu einigermaßen schnellen und trotzdem (vermutlich) kryptographisch sicheren Zufallsgeneratoren.

8.3.10 Der BBS-Generator

Wie beim RSA-Verfahren betrachtet man einen ganzzahligen Modul m, der Produkt zweier großer Primzahlen ist. Für den BBS-Generator wählt man – aus technischen Gründen, die hier nicht weiter erläutert werden – **Blum-Primzahlen** p; das sind solche, die $\equiv 3 \bmod 4$ sind. Ein Produkt zweier Blum-Primzahlen heißt **Blum-Zahl**.

Der BBS-Generator funktioniert dann so: Als ersten Schritt bildet man eine große Blum-Zahl m als Produkt zweier zufälliger Blum-Primzahlen p und q. Als zweites wählt man dann einen

[110] Lenore Blum, US-amerikanische Mathematikerin und Informatikerin, *18.12.1942
[111] Manuel Blum, US-amerikanischer Mathematiker und Informatiker, *26.4.1938
[112] Michael Shub, US-amerikanischer Mathematiker, *17.8.1943

(zufälligen) ganzzahligen Ausgangswert s mit $1 \leq s \leq m-1$, der zu m teilerfremd ist[113][114].

Nun kann man an die Erzeugung einer Zufallsfolge gehen: Man wählt $x_0 = s^2 \bmod m$ als Startwert und bildet die Folge $x_i = x_{i-1}^2 \bmod m$ für $i = 1, 2, 3, \ldots$ als Folge der inneren Zustände des Zufallsgenerators. Ausgegeben wird nur das jeweils letzte Bit der Binärdarstellung, nämlich $u_i = x_i \bmod 2$ für $i = 0, 1, 2, \ldots$, also die Parität von x_i.

Beispiel:

Ein Beispiel mit ganz kleinen Zahlen ist natürlich nicht praxistauglich, verdeutlicht aber das Vorgehen: $p = 7$, $q = 11$, $m = 77$, $s = 53$. Dann ist $s^2 = 2809$, also $x_0 = 37$ und $u_0 = 1$, da x_0 ungerade. Die Fortsetzung entnimmt man dem ganz naiven SageMath-Beispiel 8.35:

i	0	1	2	3	...
x_i	37	60	58	53	...
u_i	1	0	0	1	...

SageMath-Beispiel 8.35 (Viel zu) einfaches Beispiel für BBS

```
sage: p = 7
sage: q = 11
sage: m = p*q; m
77
sage: s = 53
sage: x0 = (s^2) % m; x0
37
sage: x1 = (x0^2) % m; x1
60
sage: x2 = (x1^2) % m; x2
58
sage: x3 = (x2^2) % m; x3
53
```

Die Zahlen p und q werden nur zur Bildung von m gebraucht und können dann sogar vernichtet werden, da sie im Gegensatz zum RSA-Verfahren nicht weiter benötigt werden; insbesondere sind sie als Geheimnis des Zufallsgenerators zu behandeln. Ebenso bleiben alle nicht ausgegebenen Bits der Folgenglieder x_i, also des inneren Zustands, geheim.

Der BBS-Generator wird von SageMath schon in der Standard-Distribution mitgebracht. Man benötigt die Prozeduren:

- `random_blum_prime()` aus dem Modul `sage.crypto.util`. Um eine zufällige Blum-Primzahl p mit einer vorgegebenen Zahl k von Bits (= Stellen in der Binärdarstellung) zu erzeugen, ruft man sie in der Form `p = random_blum_prime(2**(k-1), 2**k)` auf.

[113] Falls man ein s erwischt, das nicht zu m teilerfremd ist, hat man m per Zufall faktorisiert. Dass das vorkommt, ist äußerst unwahrscheinlich, kann aber natürlich bei der Initialisierung abgefangen werden.

[114] Falls man s im Bereich $< \sqrt{m}$ wählt, kann es passieren, dass man beim ganzzahligen Quadrieren eine Zeitlang die Grenze m nicht überschreitet. Dann sind die Ausgabebits so lange konstant, weil das Quadrat einer natürlichen Zahl dieselbe Parität hat wie die Zahl selbst. Ähnlich sieht es für s im Bereich ab $m - \sqrt{m}$ aus. Wird s aber tatsächlich zufällig gewählt, so ist es extrem unwahrscheinlich, dass es in diesen Randbereichen liegt. Wenn man es ganz sicher vermeiden möchte, kann man die Randbereiche natürlich schon bei der Wahl ausschließen.

Die Korrektheit des Algorithmus ist nur empirisch gesichert: Zwischen 2^{k-1} und 2^k gibt es zwar für $k \geq 2$ immer eine Primzahl[115], aber das muss keine Blum-Primzahl sein. Die Empirie sagt aber, dass es sogar sehr viele solche gibt, nämlich um die $2^k/(k \log(2))$, so dass eine Angreiferin mit vollständiger Suche keinen Erfolg erwarten kann.

- `blum_blum_shub()` aus `sage.crypto.stream`. Um eine Folge von r Pseudozufallsbits zu erzeugen, ruft man diese Prozedur in der Form `blum_blum_shub(r,x_0,p,q)` auf, nachdem man zuvor zwei zufällige Blum-Primzahlen p und q sowie einen Startwert $x_0 = s^2 \bmod pq$ erzeugt hat.

Das SageMath-Beispiel 8.36 demonstriert das Vorgehen. Die Zwischenergebnisse p, q und x_0 sind in den Tabellen 8.24, 8.25 und 8.26 wiedergegeben, das Ergebnis in der Tabelle 8.27. Gemäß der Konvention sind s und die Faktoren p und q geheim zu halten, es gibt aber auch keinen Grund, das Produkt $m = pq$ herauszugeben. Im Hinblick auf den Fortschritt der Faktorisierungsalgorithmen sollte man allerdings lieber Blum-Zahlen in der Größenordnung ab 2048 Bit verwenden[116]. Und in jedem Fall sollte s zufällig gewählt werden! Gegen diese Pflicht haben wir im Beispiel verstoßen, denn unser s ist eine reine Potenz.

SageMath-Beispiel 8.36 Erzeugung einer Folge von BBS-Pseudozufallsbits

```
sage: from sage.crypto.util import random_blum_prime
sage: from sage.crypto.stream import blum_blum_shub
sage: p = random_blum_prime(2^511, 2^512)
sage: q = random_blum_prime(2^511, 2^512)
sage: x0 = 11^248 % (p*q)              # s = 11^124 % (p*q)
sage: blum_blum_shub(1000,x0,p,q)
```

```
  8 445 834 617 855 090 512 176 000 413 196 767 417 799 332
626 936 992 170 472 089 385 128 414 279 550 732 184 808 226
736 683 775 727 426 619 339 706 269 080 823 255 441 520 165
438 397 334 657 231 839 251
```

Tabelle 8.24: Eine Blum-Primzahl p mit 512 Bits (154 Dezimalstellen)

```
 12 580 605 326 957 495 732 854 671 722 855 802 182 952 894
232 088 903 111 155 705 856 898 413 602 721 771 810 991 595
365 229 641 230 483 180 760 744 910 366 324 916 344 823 400
588 340 927 883 444 616 787
```

Tabelle 8.25: Eine Blum-Primzahl q mit 512 Bits (155 Dezimalstellen)

[115] Das ist ein Spezialfall des Bertrandschen Postulats, das 1850 von Tschebyschow bewiesen wurde: Zwischen n und $2n$ gibt es stets eine Primzahl (wenn $n \geq 2$).
[116] mehr dazu im Abschnitt 8.3.11

```
  1 842 408 460 334 540 507 430 929 434 383 083 145 786 026
412 146 359 363 362 017 837 922 966 741 162 861 257 645 571
680 482 798 249 771 263 305 761 292 545 408 040 659 753 561
970 871 645 393 254 757 072 936 076 922 069 587 163 804 708
256 246 366 137 431 776 175 309 050 064 068 198 002 904 756
218 898 942 856 431 647 438 473 529 312 261 281
```

Tabelle 8.26: Ein Startwert x_0

```
1010 0110 0011 0100 0000 0111 1111 0100 1111 0111 0010 1001
0000 0100 1111 0000 0010 1010 1011 1111 1000 0101 1110 0011
1110 1000 1001 1100 1000 1000 0110 0111 0011 0011 1010 0011
1100 1111 0011 1000 1011 0110 1011 1110 0110 1110 0111 1000
1101 0011 1101 0010 1000 1101 0000 1100 0100 1011 1110 0011
0110 0010 1011 0000 1010 1001 0110 0000 0011 1010 0011 1111
1010 0110 0101 1000 1011 0100 0100 1111 1010 1011 0001 1100
0000 0011 1101 1001 0001 0000 1111 1010 1001 0111 0111 0111
0000 1010 0101 0111 0111 0001 0110 1001 0011 1011 0000 0011
1000 0000 0111 0110 0110 1010 0110 0011 0111 1100 0010 0110
0011 1001 1010 1111 0001 0010 1111 0010 1100 1111 0110 0100
0001 1000 0101 0011 0000 0101 1111 1100 0101 0000 0100 0100
0100 0101 0010 1110 1010 1011 1011 0110 0101 1011 1111 1110
1100 1001 1011 0110 1001 0111 0111 1110 0101 0111 0011 0100
1101 1110 0011 1111 1101 0100 1111 1011 1010 0010 0111 1111
1010 1000 1100 1001 1010 1001 1010 0111 0100 0100 1010 0110
0011 0010 1110 0111 0101 0111 1101 0000 0110 0000 1110 1100
0101 1010 0111 1000 0101 1111 0010 1101 0110 0100 0010 1101
0000 1101 0111 1011 0010 1010 1000 0110 0100 0111 1100 0000
1101 0000 1011 1111 0101 1011 0011 1110 0010 1110 1101 0001
1110 1111 1000 0111 1010 0000 1100 0101 0110 0001
```

Tabelle 8.27: 1000 BBS-Pseudozufallsbits

8.3.11 Perfektheit und Faktorisierungsvermutung

Informell definiert man einen **Pseudozufallsgenerator** (kurz: einen Zufallsgenerator) als einen effizienten Algorithmus, der eine „kurze“ Bitkette $s \in \mathbb{F}_2^n$ in eine „lange“ Bitkette $s \in \mathbb{F}_2^r$ umwandelt.

Mathematisch exakt kann man das in der Terminologie der Komplexitätstheorie formulieren, indem man parameterabhängige Familien von Booleschen Abbildungen $G_n \colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^{r(n)}$ betrachtet und den Parameter n gegen unendlich gehen lässt. Damit ein solcher Algorithmus – repräsentiert durch die Familie (G_n) Boolescher Abbildungen – überhaupt effizient sein kann, darf die „Streckungsfunktion“ $r : \mathbb{N} \longrightarrow \mathbb{N}$ höchstens polynomial mit dem Parameter n wachsen, sonst wäre ja schon das Hinschreiben der Output-Folge nicht mehr effizient möglich. Dann misst man den Aufwand in einer irgendwie sinnvollen Weise – z. B. die Anzahl der notwendigen Bit-Operationen – und betrachtet dessen Asymptotik, die eben auch ein höchstens polynomiales Wachstum zeigen darf. Auch der Aufwand von Algorithmen, die weitere Bits vorhersagen oder sonstwie Schwächen des Zufallsgenerators aufdecken sollen, wird in Abhängigkeit von n betrachtet. Wächst dieser Aufwand stärker als jedes Polynom, z. B. exponenziell, so gilt der Angriff über einen solchen Algorithmus als nicht effizient. Dieser Zugang liefert allerdings nur qualitative Aussagen und ist daher nicht sehr befriedigend, ist aber wie auch sonst oft in der Komplexitätstheorie das Beste, was man beweisen kann.

Diesen Ansatz weiter zu verfolgen, würde hier bei weitem zu viel zusätzlichen Formalismus erfordern, zumal für kryptoanalytische Angriffe auch probabilistische Algorithmen zuzulassen sind. Es ist aber gut zu wissen, dass man die intuitive Vorstellung von Effizienz mathematisch korrekt formulieren kann. Es ist also durchaus sinnvoll mit dem naiven Ansatz zu argumentieren. Das gilt auch für die folgende Definition, die in dieser Form mathematisch nicht korrekt ist, aber eben korrekt gemacht werden kann.

Definition 8.3.1. *Gegeben sei ein Pseudozufallsgenerator. Ein* **Vorhersageverfahren**[117] *ist ein Algorithmus, der aus einem Anfangsstück $u_0, \ldots, u_{r-1}$ der erzeugten Folge das nächste Bit u_r berechnet, ohne dabei auf die internen Parameter des Pseudozufallsgenerators zuzugreifen.*

Der Pseudozufallsgenerator **besteht den Vorhersagetest**, *wenn es kein effizientes Vorhersageverfahren gibt.*

Zum Beispiel bestehen lineare Schieberegister wegen des effizienten Vorhersageverfahrens aus Satz 8.3.1 den Vorhersagetest nicht.

Definition 8.3.2. *Gegeben sei ein Pseudozufallsgenerator. Ein* **Unterscheidungsverfahren**[118] *ist ein Algorithmus, der, ohne dabei auf die internen Parameter des Pseudozufallsgenerators zuzugreifen, eine davon erzeugte Folge von einer echten Zufallsfolge unterscheiden kann.*

Der Pseudozufallsgenerator ist **perfekt**, *wenn es kein effizientes Unterscheidungsverfahren gibt.*

Ein perfekter Pseudozufallsgenerator ist insbesondere durch keinen effizienten statistischen Test von einer echten Zufallsquelle zu unterscheiden. Überraschenderweise reicht das Bestehen des Vorhersagetests für die Perfektheit schon aus, d. h., der Vorhersagetest ist „universell“.

Satz 8.3.2. (Yaos Kriterium) *Für einen Pseudozufallsgenerator sind folgende Aussagen äquivalent:*

[117] englisch: next bit predictor
[118] englisch: distinguisher

(i) *Er besteht den Vorhersagetest.*

(ii) *Er ist perfekt.*

Hier ohne Beweis.

Die (vermutete) Perfektheit des BBS-Generators

Die **Faktorisierungsvermutung** besagt, dass sich große natürliche Zahlen nicht effizient in Primfaktoren zerlegen lassen. Diese Vermutung ist die Begründung für die Sicherheit des RSA-Verfahrens, und auch, wie Satz 8.3.3 sagt, für die Perfektheit des BBS-Generators.

Satz 8.3.3. (Blum/Blum/Shub/Vazirani/Vazirani) *Wenn die Faktorisierungsvermutung richtig ist, ist der BBS-Generator perfekt.*

Der (ziemlich komplizierte) Beweis wird hier nicht ausgeführt. Salopp kann man den Satz so formulieren:

> *Wer in der Lage ist, aus einer Teilfolge des BBS-Generators auch nur ein einziges weiteres Bit vorherzusagen, kann auch den Modul faktorisieren.*

Das gilt freilich unter der Annahme, dass die Gegnerin den Modul m des BBS-Generators überhaupt kennt. Dieser kann aber auch geheim gehalten, d. h., als Teil des Schlüssels behandelt werden. Unter dieser Annahme sollte die kryptographische Sicherheit noch größer sein – aber hierfür scheint es keine Beweise, auch keine heuristischen, zu geben.

8.3.12 Beispiele und praktische Überlegungen

Der BBS-Generator ist also perfekt unter einer plausiblen, aber unbewiesenen Annahme, nämlich der Faktorisierungsvermutung. Wir wissen aber nichts Konkretes, z. B., welche Parameter möglicherweise schlecht sind. So gibt es Startwerte, die eine Output-Folge von kurzer Periode erzeugen. Dafür kennt man zwar einige Kriterien, die aber weit von einer vollständigen Antwort entfernt sind. Der Sicherheitsbeweis (relativ zur Faktorisierungsvermutung) erfordert allerdings keine zusätzlichen Annahmen. Man darf daher den BBS-Generator getrost mit der pragmatischen Einstellung verwenden: Es ist bei zufälliger Wahl der Parameter (Primfaktoren und Startwert) extrem unwahrscheinlich, dass man schlechte Werte erwischt. Jedenfalls wesentlich unwahrscheinlicher als das bekannte „Glücksspiel kann süchtig machen – Chance auf einen Hauptgewinn 1 zu 140 Millionen“.

Relevante Fragen für die Sicherheit des BBS-Generators sind aber jedenfalls:

- Wie groß muss man den Parameter m wählen?
- Wieviele Bits am Stück darf man verwenden bei gegebenem Modul und Startwert, ohne die Sicherheit zu gefährden?

Die beweisbaren Aussagen – relativ zur Faktorisierungsvermutung – sind qualitativ, nicht quantitativ. Die Empfehlung, den Modul so groß zu wählen, dass er nicht mit den bekannten Methoden faktorisiert werden kann, basiert auch nur auf heuristischen Überlegungen und ist nicht ganz zwingend, wenn der Modul auch noch geheim gehalten wird. Die tatsächliche Qualität der erzeugten Zufallsbits, sei es für statistische oder kryptographische Anwendungen, kann bis auf weiteres nur empirisch beurteilt werden. Man kann davon ausgehen, dass für Moduln, die

sich den gegenwärtigen Faktorisierungsalgorithmen noch sicher entziehen, also etwa ab 2048 Bit Länge, bei zufälliger Wahl des Moduls und des Startwerts die Gefahr extrem gering, auf jeden Fall vernachlässigbar, ist, eine „schlechte“ Bitfolge zu erzeugen[119].

Für die Länge der nutzbaren Folge gibt es nur die qualitative Aussage „höchstens polynomial“, mit der man in der konkreten Anwendung nichts anfangen kann. Aber selbst wenn man nur „quadratisch viele“ Bits zulässt, kann man bei einem $\geq$ 2000-Bit-Modul ohne weiteres 4 Millionen erzeugte Bits verwenden; bei wesentlich höherem Bedarf sollte man dann irgendwann mit neuen Parametern weitermachen.

Als weitere Frage könnte man stellen: Darf man, um die praktische Verwertbarkeit des Generators zu verbessern, in jedem Iterationsschritt mehr als nur ein Bit des inneren Zustands ausgeben? Wenigstens 2? Diese Frage wurde durch Vazirani[120]/Vazirani[121] und unabhängig von ihnen durch Alexi/Chor/Goldreich[122]/Schnorr[123] teilweise, aber auch wieder nur qualitativ, beantwortet: Wenigstens $O(\log_2 \log_2 m)$ der niedrigsten Bits sind „sicher“. Je nach Wahl der Konstanten, die in dem „O“ steckt, muss man die Bitzahl des Moduls genügend groß machen und auf empirische Erfahrungen vertrauen. Üblicherweise entscheidet man sich für genau $\lfloor \log_2 \log_2 m \rfloor$ Bits. Hat dann m 2048 Bits, also etwa 600 Dezimalstellen, so kann man also in jedem Schritt 11 Bits verwenden. Um $x^2 \bmod m$ zu berechnen, wenn m eine n-Bit-Zahl ist, braucht man $(\frac{n}{64})^2$ Multiplikationen von 64-Bit-Zahlen und anschließend ebensoviele Divisionen „128 Bit durch 64 Bit“. Bei $n = 2048$ sind das $2 \cdot (2^5)^2 = 2048$ solcher elementaren Operationen, um 11 Bits zu erzeugen, also etwa 200 Operationen pro Bit. Als gängige Faustregel kann man heute annehmen, dass ein Prozessor eine Multiplikation pro Takt erledigt[124]. Auf einem 2-GHz-Prozessor mit 64-Bit-Architektur sind das dann $2 \cdot 10^9/200 \approx 10$ Millionen Bits pro Sekunde, allerdings nur bei maschinennaher, optimierter Implementation. Der BBS-Generator ist also mit der Software-Implementation eines hinreichend sicheren nichtlinearen Kombinierers fast schon konkurrenzfähig und auf heutigen Prozessoren für viele Zwecke durchaus schnell genug.

In der Literatur werden einige weitere Pseudozufallsgeneratoren betrachtet, die nach ähnlichen Prinzipien funktionieren wie der BBS-Generator.

Der RSA-Generator (Shamir). Man wählt einen zufälligen Modul m der Stellenzahl n, der ein Produkt zweier großer Primzahlen p, q ist, und einen Exponenten d, der teilerfremd zu $(p-1)(q-1)$ ist, ferner einen zufälligen Startwert $x = x_0$. Die interne Transformation ist $x \mapsto x^d \bmod m$. Man bildet also $x_i = x_{i-1}^d \bmod m$ und gibt das letzte Bit oder auch die letzten $\lfloor \log_2 \log_2 m \rfloor$ Bits aus. Wenn dieser Pseudozufallsgenerator nicht perfekt ist, dann gibt es einen effizienten Algorithmus zum Brechen der RSA-Verschlüsselung. Der Rechenaufwand ist größer als beim BBS-Generator in dem Maße, wie das Potenzieren mit d aufwendiger als das Quadrieren ist; ist d zufällig gewählt, so ist der Zeitbedarf pro Bit $O(n^3)$, da das Potenzieren mit einer n-Bit-Zahl im Vergleich zum schlichten Quadrieren in jedem Schritt den Aufwand mit dem Faktor n vergrößert.

Der Index-Generator (Blum/Micali). Man wählt als Modul zufällig eine große Primzahl p

[119] Auf Émile Borel geht die folgende informelle Abstufung der Vernachlässigbarkeit extrem kleiner Wahrscheinlichkeiten zurück: aus menschlicher Perspektive $\leq 10^{-6}$, aus irdischer Perspektive $\leq 10^{-15}$, aus kosmischer Perspektive $\leq 10^{-45}$. Diese Schranken werden bei genügend großer Wahl des Moduls m für das BBS- (oder RSA-) Verfahren mühelos unterboten.

[120] Umesh Vazirani, indisch-US-amerikanischer Informatiker

[121] Vijay Vazirani, indisch-US-amerikanischer Informatiker, *20.4.1957

[122] Oded Goldreich, israelischer Mathematiker und Informatiker, *4.2.1957

[123] Claus-Peter Schnorr, deutscher Mathematiker und Informatiker, *4.8.1943

[124] Es geht auch wesentlich schneller auf speziellen Prozessoren, etwa durch „Pipelining“ und Parallelisierung.

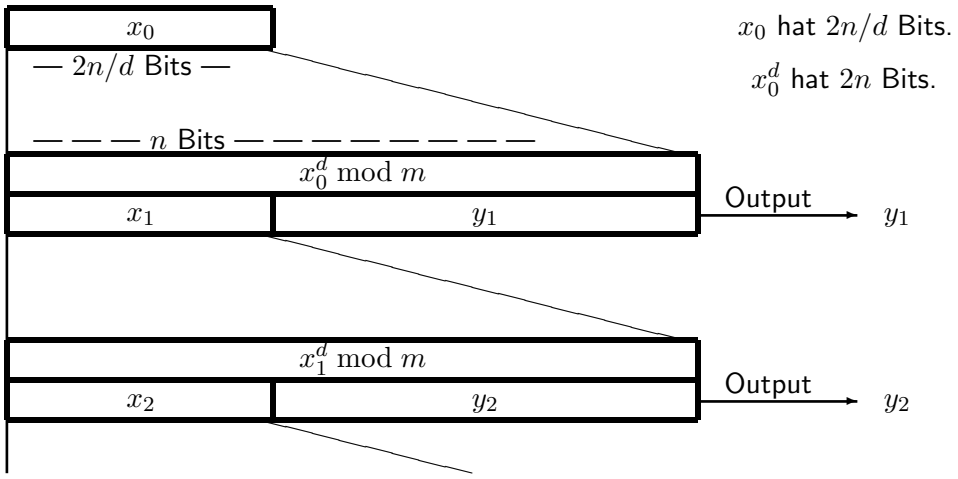


Abbildung 8.28: Micali-Schnorr-Generator

und bestimmt dazu eine Primitivwurzel[125] a. Ferner wählt man einen zufälligen Startwert $x = x_0$, teilerfremd zu $p-1$. Dann bildet man $x_i = a^{x_{i-1}} \bmod p$ und gibt das erste oder die ersten $\lfloor \log_2 \log_2 p \rfloor$ Bits aus. Die Perfektheit dieses Pseudozufallsgenerators beruht auf der Vermutung, dass die Berechnung diskreter Logarithmen mod$p$ hart ist. Auch hier ist der Zeitbedarf pro Bit $O(n^3)$.

Der elliptische Index-Generator (Kaliski). Er funktioniert wie der Index-Generator, nur dass man die Gruppe der invertierbaren Elemente des Körpers $\mathbb{F}_p$ durch eine elliptische Kurve über $\mathbb{F}_p$ ersetzt (eine solche Kurve ist auf kanonische Weise eine endliche Gruppe).

8.3.13 Der Micali-Schnorr-Generator

Ein von Micali[126] und Schnorr vorgeschlagener Pseudozufallsgenerator ist ein Abkömmling des RSA-Generators. Sei dazu $d \geq 3$ ungerade. Als Parametermenge dient die Menge aller Produkte m von zwei Primzahlen p und q, die sich in ihrer Bitanzahl höchstens um 1 unterscheiden und für die d zu $(p-1)(q-1)$ teilerfremd ist. Wenn m eine n-Bit-Zahl ist, sei $h(n) \approx \frac{2n}{d}$ ganzzahlig; die d-te Potenz einer $h(n)$-Bit-Zahl ist dann (ungefähr) eine $2n$-Bit-Zahl.

Im i-ten Schritt wird $z_i = x_{i-1}^d \bmod m$ gebildet; davon werden die ersten $h(n)$ Bits, also $\lfloor z_i / 2^{n-h(n)} \rfloor$, als x_i genommen, die übrigen Bits, also $y_i = z_i \bmod 2^{n-h(n)}$ werden ausgegeben. Bemerkenswert ist, dass die Bits auf zwei *disjunkte* Teile verteilt werden: den Wert x_i für den nächsten Schritt und die Ausgabe y_i. Die Abbildung 8.28 macht das deutlich.

Mit welcher Rechtfertigung wird dieser Zufallsgenerator als perfekt vermutet? Dazu wird folgendes angenommen: Kein effizienter Test kann die Gleichverteilung auf $\{1, \ldots, m-1\}$ von der Verteilung von $x^d \bmod m$ für gleichverteiltes $x \in \{1, \ldots, 2^{h(n)}\}$ unterscheiden. Ist diese Annahme richtig, so ist der Micali-Schnorr-Generator perfekt. Es gibt heuristische Überlegungen, die die Annahme in enge Beziehung zur Sicherheit des RSA-Verschlüsselungsverfahrens und zur Primzerlegung bringen, man muss aber konstatieren, dass der „Sicherheitsbeweis“ doch deutlich dünner als der für das BBS-Verfahren ist.

[125] Das ist eine Zahl, deren Potenzen alle Restklassen $\neq 0 \bmod p$ durchlaufen, oder, algebraisch ausgedrückt, ein erzeugendes Element der multiplikativen Gruppe mod p.

[126] Silvio Micali, US-amerikanischer Informatiker, *13.10.1954

Wie schnell purzeln nun die Pseudozufallsbits aus der Maschine? Als elementare Operationen gezählt werden sollen wieder die Multiplikation zweier 64-Bit-Zahlen und die Division einer 128-Bit-Zahl durch eine 64-Bit-Zahl mit 64-Bit-Quotient. Multipliziert und dividiert wird nach der klassischen Methode[127]; das Produkt von s (64-Bit-)Wörtern mit t Wörtern kostet also st elementare Operationen, bei der Division ist der Aufwand das Produkt der Wortzahlen von Divisor und Quotient.

Die Erfinder machten nun einen konkreten Vorschlag[128]: $d = 7$, $n = 512$. Ausgegeben werden jeweils 384 Bits, zurückbehalten werden 128 Bits. Das binäre Potenzieren einer 128-Bit-Zahl x mit 7 kostet eine Reihe elementarer Operationen:

- x hat 128 Bits, also 2 Wörter.
- x^2 hat 256 Bits, also 4 Wörter, und kostet $2 \cdot 2 = 4$ elementare Operationen.
- x^3 hat 384 Bits, also 6 Wörter, und kostet $2 \cdot 4 = 8$ elementare Operationen.
- x^4 hat 512 Bits, also 8 Wörter, und kostet $4 \cdot 4 = 16$ elementare Operationen.
- x^7 hat 896 Bits, also 14 Wörter, und kostet $6 \cdot 8 = 48$ elementare Operationen.
- $x^7 \bmod m$ hat ≤ 512 Bits und kostet ebenfalls $6 \cdot 8 = 48$ elementare Operationen.

Insgesamt braucht man also 124 elementare Operationen; es war nur eine Reduktion $\bmod m$ nötig. Die Belohnung besteht aus 384 Bits. Man erhält also etwa 3 Bits pro elementarer Operation, also nach der Annahme in Abschnitt 8.3.12 etwa 6 Milliarden Bits pro Sekunde. Gegenüber dem BBS-Generator ist das ein Faktor von fast 1000.

Eine fast beliebige Geschwindigkeitssteigerung ergibt sich durch Parallelisierung: Der Micali-Schnorr-Generator ist vollständig parallelisierbar; das bedeutet, dass eine Verteilung der Arbeit auf k Prozessoren einen Gewinn um den echten Faktor k bedeutet: Die Prozessoren können unabhängig voneinander arbeiten ohne Notwendigkeit zur Kommunikation.

8.3.14 Zusammenfassung und Ausblick

Bitstrom-Chiffren brauchen kryptographisch sichere Zufallsgeneratoren. Diese scheint es zu geben, auch wenn ihre Sicherheit – wie die auch aller anderen Verschlüsselungsverfahren – mathematisch nicht vollständig bewiesen ist.

Ein guter Zufallsgenerator reicht aber nicht aus, um eine praxistaugliche Bitstrom-Chiffre zu implementieren:

- Für die Nachrichten-Integrität sind zusätzliche Überlegungen erforderlich, z. B. die Kombination mit einer kryptographischen Hash-Funktion.
- Bei der Benutzung muss zuverlässig verhindert werden, dass der gleiche Schlüsselstrom irgendwann noch einmal vorkommt, d. h., das Schlüsselmanagement erfordert zusätzliche Sorgfalt. Ein möglicher Ansatz[129] dafür ist es, einen längerfristig gültigen Schlüssel aus internen Parametern des Zufallsgenerators zusammenzusetzen und die übrigen Parameter einschließlich Startwert als einmaligen Nachrichtenschlüssel zu verwenden.

[127] Die Multiplikation mithilfe der schnellen Fourier-Transformation bringt erst bei viel größeren Stellenzahlen Vorteile.
[128] Heute würde man n wesentlich größer wählen.
[129] Dies war der übliche Ansatz bei den Verschlüsselungsmaschinen im zweiten Weltkrieg.

Im Gegensatz zu Bitblock-Chiffren mit ihrem akzeptierten Standard AES (und davor dem Standard DES) gibt es für Bitstrom-Chiffren kein gleichermaßen etabliertes Verfahren. Einer Standardisierung am nächsten kommt das eSTREAM-Portfolio, das von 2004 bis 2008 in einem europäischen Projekt erarbeitet wurde und mehrere Verfahren empfiehlt [Sch16].

Leider wurden gerade im Bereich der Bitstrom-Chiffren in der Vergangenheit viele im Hinterzimmer entwickelte („proprietäre“) Verfahren in sicherheitsrelevante Anwendungen eingebracht, deren Funktionsweise geheim gehalten wurde, aber durch „Reverse Engineering“ ermittelt werden konnte, und deren Sicherheit dann von Kryptologen leicht zerpflückt werden konnte. Daher schließt dieses Kapitel mit dem guten Rat, der hier wie überall in der Kryptographie gilt:

> *Traue keinem Zufallsgenerator, dessen Algorithmus dir vorenthalten wird und für den es keine öffentlich zugänglichen Analyse-Informationen gibt. Statistische Analysen reichen als Sicherheitsnachweis nicht aus, ebensowenig wie eine gigantische Periode oder eine riesige Auswahl an Startwerten.*

8.4 Anhang: Boolesche Abbildungen in SageMath

8.4.1 Was liefert SageMath mit?

SageMath bietet einige Funktionen zur Behandlung von Bitblöcken und Booleschen Funktionen. Diese sind allerdings etwas unsystematisch über unterschiedliche Module verteilt und operieren mit unterschiedlichen Datentypen und Einschränkungen. Man könnte sich damit behelfen. In diesem Text stützen wir uns allerdings überwiegend auf eine eigene unabhängige (freie) Implementation[130].

Bitblöcke

Die SageMath-Funktion `binary()` wandelt Zahlen in Bitketten um. Anwendung `123456789.binary()`, Ergebnis `'111010110111100110100010101'`.

Im Modul `sage.crypto.util` gibt es die Funktionen `ascii_to_bin()`, `bin_to_ascii()` und `ascii_integer()`. Außerdem gibt es ganz woanders, in `sage.crypto.block_cipher.sdes`, noch die Funktion `sdes.string_to_list()`, die eine Bitkette in eine Liste umwandelt.

Logische Ausdrücke

Funktionen zur Behandlung logischer Ausdrücke sind in den Modulen `sage.logic.boolformula`, `sage.sat.converters`, `sage.sat.solvers` und `sage.sat.boolean_polynomials` enthalten, die allerdings eine ganz andere Denkweise als die in diesem Text vorherrschende, für kryptographische Überlegungen zweckmäßige, erfordern.

Boolesche Funktionen und Abbildungen

Zum Umgang mit Booleschen Funktionen kann man aus dem Modul `sage.crypto.boolean_function` die Funktionen `truth_table()`, `algebraic_normal_form()` und `walsh_hadamard_transform()` heranziehen, die als Methoden der Klasse `BooleanFunction` implementiert sind.

Zur Behandlung von Booleschen Abbildungen dienen die Funktionen `cnf()`, `linear_approximation_matrix()` und `difference_distribution_matrix()` als Methoden der Klasse `SBox` aus dem Modul `sage.crypto.mq.sbox`.

8.4.2 Neu implementierte SageMath-Funktionen

Die im Folgenden beschriebenen SageMath-Klassen und -Funktionen sind im Modul `bitciphers.sage` zusammengefasst Sie wurden mit Python 3.4.1[131] unter OS X entwickelt und mit SageMath 6.2 getestet.

Zur Nutzung benötigt man eine entsprechende SageMath-Installation und bindet das Modul in einem Kommando-Fenster mit den SageMath-Kommandos

[130] Man nennt das „das Rad neu erfinden“. Die angestrebte Einheitlichkeit und Vollständigkeit mögen es entschuldigen. Auch sollte das Durcharbeiten einen Lerneffekt mit sich bringen. Weiterer Aspekt: Die folgenden Module sind reines Python und deshalb auch ohne eine SageMath-Implementation verwendbar.

[131] Der einzige Unterschied zu Python 2, der verwendet wurde, ist der Operator `//` für die Ganzzahl-Division.

```
load_attach_path(path='/mein/Pfad', replace=False)

attach('bitciphers.sage')
```

ein. Nutzt man das Worksheet-Frontend (z. B. mit einem SageMath-Server), so kopiert man besser die einzelnen Funktionen in jeweils eine Eingabezelle[132].

8.4.3 Konversionen von Bitblöcken

SageMath-Beispiel 8.37 Konversion von Bitblöcken

```
def int2bbl(number,dim):
  """Converts number to bitblock of length dim via base-2
  representation."""
  n = number                          # catch input
  b = []                              # initialize output
  for i in range(0,dim):
    bit = n % 2                       # next base-2 bit
    b = [bit] + b                     # prepend
    n = (n - bit)//2
  return b

def bbl2int(bbl):
  """Converts bitblock to number via base-2 representation."""
  ll = len(bbl)
  nn = 0                              # initialize output
  for i in range(0,ll):
    nn = nn + bbl[i]*(2**(ll-1-i))    # build base-2 representation
  return nn
```

[132] Ein vollständiges SageMath-Worksheet mit allen Beispielen dieses Texts ist als `bitciphers.sws` bzw. in lesbarem PDF-Format als `bitciphers_sws.pdf` erhältlich.

SageMath-Beispiel 8.38 Konversion von Bitblöcken (Fortsetzung)

```
def str2bbl(bitstr):
  """Converts bitstring to bitblock."""
  ll = len(bitstr)
  xbl = []
  for k in range(0,ll):
    xbl.append(int(bitstr[k]))
  return xbl

def bbl2str(bbl):
  """Converts bitblock to bitstring."""
  bitstr = ""
  for i in range(0,len(bbl)):
    bitstr += str(bbl[i])
  return bitstr

def txt2bbl(text):
  """Converts ASCII-text to bitblock."""
  ll = len(text)
  xbl = []
  for k in range(0,ll):
    n = ord(text[k])
    by = int2bbl(n,8)
    xbl.extend(by)
  return xbl

def bbl2sub(bbl):
  """Converts bitblock to subset."""
  ll = len(bbl)
  set = []
  for i in range(0,ll):
    if (bbl[i] == 1):
      set.append(i+1)
  return set
```

SageMath-Beispiel 8.39 Diverse Verknüpfungen von Bitblöcken

```
def coinc(x,y):
  """Counts coincidences between 2 lists."""
  ll = len(x)
  assert ll <= len(y), "coinc_Error: Second bitblock too short."
  nn = 0
  for i in range(0,ll):
    if (x[i] == y[i]):
      nn += 1
  return nn

def binScPr(x,y):
  """Scalar product of two binary vectors (lists) mod 2."""
  l = len(x)
  assert l == len(y), "binScPr_Error: Blocks have different lengths."
  res = 0
  for i in range (0,l):
    res += x[i] * y[i]
  return res %2

def xor(plain,key):
  """Binary addition of bitblocks.
  Crops key if longer than plain.
  Repeats key if shorter than plain.
  """
  lk = len(key)
  lp = len(plain)
  ciph = []
  i = 0
  for k in range(0,lp):
    cbit = (plain[k] + key[i]) % 2
    ciph.append(cbit)
    i += 1
    if i >= lk:
      i = i-lk
  return ciph
```

8.4.4 Matsui-Test

SageMath-Beispiel 8.40 Matsui-Test

```
def Mats_tst(a, b, pc, compl = False):
  """Matsui's test for linear cryptanalysis"""
  NN = len(pc)
  results = []
  for pair in pc:
    ax = binScPr(a,pair[0])
    by = binScPr(b,pair[1])
    result = (ax + by) % 2
    results.append(result)
  t_0 = 0
  for bb in results:
    if bb == 0:
      t_0 = t_0 + 1
  if 2*t_0 > NN:
    if compl:
      return [t_0,1,True]
    else:
      return [t_0,0,True]
  elif 2*t_0 < NN:
    if compl:
      return [t_0,0,True]
    else:
      return [t_0,1,True]
  else:
    return [t_0,randint(0,1),False]
```

8.4.5 Walsh-Transformation

SageMath-Beispiel 8.41 Walsh-Transformation von Bitblöcken

```
def wtr(xx):
  """Fast Walsh transform of a list of numbers"""
  max = 4096                  # max dim = 12
  ll = len(xx)
  assert ll <= max, "wtr_Error: Bitblock too long."
  dim = 0                     # dimension
  m = 1                       # 2**dimension
  while m < ll:
    dim = dim+1
    m = 2*m
  assert ll == m, "wtr_Error: Block length not a power of 2."
  x = copy(xx)                # initialize auxiliary bitblock
  y = copy(xx)                # initialize auxiliary bitblock
  mi = 1                      # actual power of 2
  for i in range(0,dim):      # binary recursion
    for k in range(0,ll):
      if ((k//mi) % 2 == 1):  # picks bit nr i
        y[k] = x[k-mi] - x[k]
      else:
        y[k] = x[k+mi] + x[k]
    for k in range(0,ll):
      x[k] = y[k]
    mi = 2*mi                 # equals 2**i in the next step
  return x
```

8.4.6 Klasse für Boolesche Funktionen

SageMath-Beispiel 8.42 Klasse für Boolesche Funktionen

```
class BoolF(object):
  """Boolean function
  Attribute: a list of bits describing the truth table of the function
  Attribute: the dimension of the domain"""

  __max = 4096                                # max dim = 12

  def __init__(self,blist,method="TT"):
    """Initializes a Boolean function with a truth table
    or by its algebraic normal form if method is ANF."""
    ll = len(blist)
    assert ll <= self.__max, "BoolF_Error: Bitblock too long."
    dim = 0                                   # dimension
    m = 1                                     # 2**dim
    while m < ll:
      dim = dim+1
      m = 2*m
    assert ll == m, "booltestError: Block length not a power of 2."
    self.__dim = dim
    if method=="TT":
      self.__tlist = blist
    else:
      self.__tlist=self.__convert(blist)

  def __convert(self,xx):
    """Converts a truth table to an ANF or vice versa."""
    x = copy(xx)                  # initialize auxiliary bitblock
    y = copy(xx)                  # initialize auxiliary bitblock
    mi = 1                        # actual power of 2
    for i in range(0,self.__dim): # binary recursion
      for k in range(0,2**(self.__dim)):
        if ((k//mi) % 2 == 1): # picks bit nr i
          y[k] = (x[k-mi] + x[k]) % 2 # XOR
        else:
          y[k] = x[k]
      for k in range(0,2**(self.__dim)):
        x[k] = y[k]
      mi = 2*mi                   # equals 2**i in the next step
    return x
```

SageMath-Beispiel 8.43 Boolesche Funktionen – Fortsetzung

```
  def getTT(self):
    """Returns truth table as bitlist."""
    return self.__tlist

  def valueAt(self,xx):
    """Evaluates Boolean function."""
    ll = len(xx)
    assert ll == self.__dim, "booltestError: Block has false length."
    index = bbl2int(xx)
    return self.__tlist[index]

  def getDim(self):
    """Returns dimension of definition domain."""
    return self.__dim

  def getANF(self):
    """Returns algebraic normal form as bitlist."""
    y = self.__convert(self.__tlist)
    return y

  def deg(self):
    """Algebraic degree of Boolean function"""
    y = self.__convert(self.__tlist)
    max = 0
    for i in range (0,len(y)):
      if y[i] != 0:
        b = int2bbl(i,self.__dim)
        wt = sum(b)
        if wt > max:
          max = wt
    return max
```

SageMath-Beispiel 8.44 Boolesche Funktionen – Walsh-Spektrum und menschenlesbare Ausgabe

```
def wspec(self):
  """Calculate Walsh spectrum."""
  ff = copy(self.__tlist)
  ll = len(ff)
  gg = []
  for i in range(0,ll):
    bit = ff[i]
    if (bit):
      gg.append(-1)
    else:
      gg.append(1)
  ff = wtr(gg)
  return ff

def printTT(self):
  """Prints truth table to stdout."""
  for i in range(0,2**(self.__dim)):
    bb = int2bbl(i,self.__dim)
    print "Value at " + bbl2str(bb) + " is " + repr(self.__tlist[i])

def printANF(self):
  """Prints algebraic normal form to stdout."""
  y = self.__convert(self.__tlist)
  for i in range(0,2**(self.__dim)):
    monom = int2bbl(i,self.__dim)
    print "Coefficient at " + bbl2str(monom) + " is " + repr(y[i])
```

8.4.7 Klasse für Boolesche Abbildungen

SageMath-Beispiel 8.45 Klasse für Boolesche Abbildungen

```
class BoolMap(object):
  """Boolean map
  Attribute: a list of Boolean functions
  Attribute: the dimensions of domain and range"""

  __max = 8                                  # max dim = 8

  def __init__(self,flist):
    """Initializes a Boolean map with a list of Boolean functions."""
    qq = len(flist)
    assert qq <= self.__max, "BoolMap_Error: Too many components."
    ll = len(flist[0].getTT())
    dim = 0                                   # dimension
    m = 1                                     # 2**dim
    while m < ll:
      dim = dim+1
      m = 2*m
    assert ll == m, "BoolMap_Error: Block length not a power of 2."
    assert dim <= self.__max, "BoolMap_Error: Block length exceeds maximum."
    self.__dimd = dim
    self.__dimr = qq
    for i in range(1,qq):
      li = len(flist[i].getTT())
      assert li == ll,  "BoolMap_Error: Blocks of different lengths."
    self.__flist = flist

  def getFList(self):
    """Returns component list."""
    return self.__flist

  def getDim(self):
    """Returns dimension of preimage and image domain."""
    return [self.__dimd, self.__dimr]
```

SageMath-Beispiel 8.46 Boolesche Abbildungen – Fortsetzung

```
  def getTT(self):
    """Returns truth table as list of bitlists."""
    nn = 2**(self.__dimd)
    qq = self.__dimr
    clist = []
    for j in range(0,qq):
      clist.append(self.__flist[j].getTT())
    transp = []
    for j in range(0,nn):
      trrow = []
      for i in range(0,qq):
        trrow.append(clist[i][j])
      transp.append(trrow)
    return transp

  def printTT(self):
    """Prints truth table to stdout."""
    nn = 2**(self.__dimd)
    qq = self.__dimr
    print("Dimensions of truth table:", nn, "by", qq)
    clist = []
    for j in range(0,qq):
      clist.append(self.__flist[j].getTT())
    transp = []
    for j in range(0,nn):
      trrow = []
      for i in range(0,qq):
        trrow.append(clist[i][j])
      transp.append(trrow)
    for j in range(0,nn):
      bb = int2bbl(j,self.__dimd)
      print("Value at", bb, "is", transp[j])

  def valueAt(self,xx):
    """Evaluates Boolean map."""
    ll = len(xx)
    assert ll == self.__dimd, "boolF_Error: Block has false length."
    index = bbl2int(xx)
    vlist = []
    for j in range(0,self.__dimr):
      vlist.append(self.__flist[j].getTT()[index])
    return vlist
```

SageMath-Beispiel 8.47 Boolesche Abbildungen – Fortsetzung

```
  def wspec(self):
    """Calculate Walsh spectrum."""
    dd = self.getDim()
    tt = self.getTT()
    m = 2**(dd[0])
    t = 2**(dd[1])
    nullv = [0] * t
    charF = []
    for k in range(0,m):
      charF.append(copy(nullv))
    for k in range(0,m):
      index = bbl2int(tt[k])
      charF[k][index] = 1
    blist = []
    for k in range(0,m):
      blist.extend(charF[k])
    speclist = wtr(blist)
    specmat = []
    for k in range(0,m):
      specmat.append(speclist[k*t:k*t+t])
    return specmat

  def linApprTable(self):
    """Calculate the linear approximation table."""
    lpr = self.wspec()
    dd = self.getDim()
    m = 2**(dd[0])
    t = 2**(dd[1])
    for k in range(0,m):
      for i in range(0,t):
        lpr[k][i] = (lpr[k][i] + m)//2
    return lpr
```

SageMath-Beispiel 8.48 Boolesche Abbildungen – lineares Profil

```
def linProf(self, extended=False):
  """Calculate linear profile. If extended is True, also
     calculate maximum potential and corresponding linear forms."""
  lpr = self.wspec()
  dd = self.getDim()
  m = 2**(dd[0])
  t = 2**(dd[1])
  for k in range(0,m):
    for i in range(0,t):
      lpr[k][i] = lpr[k][i] * lpr[k][i]
  if extended:
    flatlist = []
    for row in lpr:
      flatlist.extend(row)
    denominator = flatlist.pop(0)
    mm = max(flatlist)
    ixlist = []
    for k in range(0,m):
      for i in range(0,t):
        if lpr[k][i] == mm:
          ixlist.append([k,i])
    return [lpr, mm, denominator, ixlist]
  else:
    return lpr
```

8.4.8 Lucifer und Mini-Lucifer

SageMath-Beispiel 8.49 S-Boxen und Bitpermutation von Lucifer

```
#-------------- Define S0 ----------------------------
f1 = BoolF([1,1,0,1,1,1,1,0,0,0,0,0,1,0,0,1])
f2 = BoolF([1,1,1,0,1,1,0,0,0,1,0,0,0,1,1,0])
f3 = BoolF([0,1,1,1,1,0,1,0,1,1,1,0,0,0,0,0])
f4 = BoolF([0,1,1,0,0,1,1,0,0,0,1,1,1,0,1,0])
S0 = BoolMap([f1,f2,f3,f4])

#-------------- Define S0 inverse --------------------
fi1 = BoolF([0,1,1,1,1,1,1,0,1,1,0,0,0,0,0,0])
fi2 = BoolF([1,0,0,0,1,1,0,0,1,1,0,1,0,1,1,0])
fi3 = BoolF([1,1,0,1,0,1,0,1,1,0,1,1,0,0,0,0])
fi4 = BoolF([1,1,0,0,1,0,1,0,1,0,1,0,0,1,0,1])
S0inv = BoolMap([fi1,fi2,fi3,fi4])

#-------------- Define S1 ----------------------------
g1 = BoolF([0,0,1,1,0,1,0,0,1,1,0,1,0,1,1,0])
g2 = BoolF([1,0,1,0,0,0,0,1,1,1,0,0,1,1,0,1])
g3 = BoolF([1,1,1,0,1,1,0,0,0,0,0,1,1,1,0,0])
g4 = BoolF([1,0,0,1,1,1,0,0,0,1,1,0,0,1,0,1])
S1 = BoolMap([g1,g2,g3,g4])

#-------------- Define S1 inverse --------------------
gi1 = BoolF([0,1,0,0,0,1,1,0,1,0,1,0,1,1,0,1])
gi2 = BoolF([1,0,0,1,1,1,1,0,1,0,0,1,0,0,0,1])
gi3 = BoolF([1,1,0,0,1,1,0,0,1,1,1,0,0,0,1,0])
gi4 = BoolF([0,0,1,0,1,1,0,0,0,1,1,1,0,1,0,1])
S1inv = BoolMap([gi1,gi2,gi3,gi4])

def P(b):
  """Lucifer's bit permutation"""
  pb = [b[2],b[5],b[4],b[0],b[3],b[1],b[7],b[6]]
  return pb
```

SageMath-Beispiel 8.50 Mini-Lucifer über r Runden

```
def miniLuc(a,k,r):
  """Mini-Lucifer, encrypts 8-bit a with 16-bit key k over r rounds."""
  ll = len(a)
  assert ll == 8, "miniLuc_Error: Only blocks of length 8 allowed."
  lk = len(k)
  assert lk == 16, "miniLuc_Error: Only keys of length 16 allowed."
  k0 = k[0:8]           # split into subkeys
  k1 = k[8:16]
  aa = a                # round input
  # --- begin round
  for i in range(0,r): # round number is i+1
    if (i % 2 == 0):   # select round key
      rndkey = k0
    else:
      rndkey = k1
    b = xor(aa,rndkey)     # add round key
    bleft = b[0:4]         # begin substitution
    bright = b[4:8]
    bbleft = S0.valueAt(bleft)
    bbright = S1.valueAt(bright)
    bb = bbleft + bbright  # end substitution
    if (i+1 == r):         # omit permutation in last round
      aa = bb
    else:
      aa = P(bb)
  # --- end round
  if (r % 2 == 0):         # add subkey after last round
    finkey = k0
  else:
    finkey = k1
  c = xor(aa,finkey)
  return c
```

8.4.9 Klasse für lineare Schieberegister

SageMath-Beispiel 8.51 Klasse für linear rückgekoppelte Schieberegister

```
class LFSR(object):
  """Linear Feedback Shift Register
  Attributes: the length of the register
              a list of bits describing the taps of the register
              the state
  """

  __max = 1024                                # max length

  def __init__(self,blist):
    """Initializes a LFSR with a list of taps
    and the all 0 state."""
    ll = len(blist)
    assert ll <= self.__max, "LFSR_Error: Bitblock too long."
    self.__length = ll
    self.__taplist = blist
    self.__state = [0] * ll

  def __str__(self):
    """Defines a printable string telling the internals of
    the register."""
    outstr = "Length: " + str(self.__length)
    outstr += " | Taps: " + bbl2str(self.__taplist)
    outstr += " | State: " + bbl2str(self.__state)
    return outstr

  def getLength(self):
    """Returns the length of the LFSR."""
    return self.__length

  def setState(self,slist):
    """Sets the state."""
    sl = len(slist)
    assert sl == self.__length, "LFSR_Error: Bitblock has wrong length."
    self.__state = slist
```

SageMath-Beispiel 8.52 Klasse für linear rückgekoppelte Schieberegister – Fortsetzung

```
  def nextBits(self,n):
    """Returns the next n bits as a list and updates the state."""
    outlist = []
    a = self.__taplist
    u = self.__state
    for i in range (0,n):
      b = binScPr(a,u)
      c = u.pop()
      u.insert(0,b)
      outlist.append(c)
    self.__state = u
    return outlist
```

Literaturverzeichnis (BitCiphers)

[Bar09] Bard, Gregory V.: *Algebraic Cryptanalysis*. Springer, Dordrecht, 2009.

[Bri10] Brickenstein, Michael: *Boolean Gröbner Bases – Theory, Algorithms and Applications*. Dissertation, TU Kaiserslautern, department of Mathematics, 2010.
See also "PolyBoRi – Polynomials over Boolean Rings", online:
`http://polybori.sourceforge.net/`".

[CGH+03] Castro, D., M. Giusti, J. Heintz, G. Matera und L. M. Pardo: *The hardness of polynomial equation solving*. Found. Comput. Math., 3:347–420, 2003.

[CH10] Crama, Yves und Peter L. Hammer (Herausgeber): *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Cambridge University Press, 2010.

[CLO07] Cox, David, John Little und Donal O'Shea: *Ideals, Varieties, and Algorithms*. Springer, 3. Auflage, 2007.

[CS09] Cusick, Thomas W. und Pantelimon Stănică: *Cryptographic Boolean Functions and Applications*. Elsevier Academic Press, 2009.

[DR02] Daemen, Joan und Vincent Rijmen: *The Design of Rijndael. AES – The Advanced Encryption Standard*. Springer, 2002.

[GJ79] Garey, Michael R. und David S. Johnson: *Computers and Intractability*. Freeman, 1979.

[Gol82] Golomb, Solomon W.: *Shift Register Sequences*. Aegean Park Press, 1982. Revised Edition.

[Laz83] Lazard, Daniel: *Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations*. In: *Lecture Notes in Computer Science 162*, Seiten 146–156. Springer, 1983. EUROCAL '83.

[LV00] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes*. In: *Lecture Notes in Computer Science 558*, Seiten 446–465, 2000. PKC2000.
See also "BlueKrypt Cryptographic Key Length Recommendation", last update 2015, online: `http://www.keylength.com/en/2/`.

[MS89] Meier, W. und O. Staffelbach: *Fast correlation attacks on certain stream ciphers*. Journal of Cryptology, 1:159–176, 1989.

[MvOV01] Menezes, Alfred J., Paul C. van Oorschot und Scott A. Vanstone: *Handbook of Applied Cryptography*. Series on Discrete Mathematics and Its Application. CRC

Press, 5. Auflage, 2001, ISBN 0-8493-8523-7. (Errata last update Jan 22, 2014).
http://cacr.uwaterloo.ca/hac/,
http://www.cacr.math.uwaterloo.ca/hac/.

[Opp11] Oppliger, Rolf: *Contemporary Cryptography, Second Edition*. Artech House, 2. Auflage, 2011. http://books.esecurity.ch/cryptography2e.html.

[Pom08] Pommerening, Klaus: *Linearitätsmaße für Boolesche Abbildungen*, 2008. Manuskript, 30. Mai 2000. Letzte Revision 4. Juli 2008.
English equivalent: *Fourier Analysis of Boolean Maps – A Tutorial*.
http://www.staff.uni-mainz.de/pommeren/Kryptologie/Bitblock/A_Nonlin/nonlin.pdf.

[Pom14] Pommerening, Klaus: *Fourier Analysis of Boolean Maps – A Tutorial*, 2014. Manuscript: May 30, 2000. Last revision August 11, 2014.
German aequivalent: *Linearitätsmaße für Boolesche Abbildungen*.
http://www.staff.uni-mainz.de/pommeren/Cryptology/Bitblock/Fourier/Fourier.pdf.

[Pom16] Pommerening, Klaus: *Cryptanalysis of nonlinear shift registers*. Cryptologia, 30, 2016.
http://www.tandfonline.com/doi/abs/10.1080/01611194.2015.1055385.

[PP09] Paar, Christof und Jan Pelzl: *Understanding Cryptography – A Textbook for Students and Practioners*. Springer, 2009.

[Sch03] Schmeh, Klaus: *Cryptography and Public Key Infrastructure on the Internet*. John Wiley, 2003. In German, the 6th edition was published in 2016.

[Sch16] Schmeh, Klaus: *Kryptographie – Verfahren, Protokolle, Infrastrukturen*. dpunkt.verlag, 6. Auflage, 2016. Sehr gut lesbares, aktuelles und umfangreiches Buch über Kryptographie. Geht auch auf praktische Probleme (wie Standardisierung oder real existierende Software) ein.

[Seg04] Segers, A. J. M.: *Algebraic Attacks from a Gröbner Basis Perspective*. Diplomarbeit, Technische Universiteit Eindhoven, 2004.
http://www.win.tue.nl/~henkvt/images/ReportSegersGB2-11-04.pdf.

[SL07] Stamp, Mark und Richard M. Low: *Applied Cryptanalysis: Breaking Ciphers in the Real World*. Wiley-IEEE Press, 2007.
http://cs.sjsu.edu/faculty/stamp/crypto/.

[Sti06] Stinson, Douglas R.: *Cryptography – Theory and Practice*. Chapman & Hall/CRC, 3. Auflage, 2006.

[vzGG99] Gathen, Joachim von zur und Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 1999.

Alle Links wurden am 15.07.2016 überprüft.

Kapitel 9

Homomorphe Chiffren

(Martin Franz, Januar 2013)

9.1 Einführung

Homomorphe Chiffren sind Public-Key-Verfahren mit besonderen Eigenschaften. Sie erlauben es, bestimmte Berechnungen auf verschlüsselten Daten durchzuführen, ohne die Daten selbst zu kennen oder diese entschlüsseln zu müssen. Dies findet in der Praxis relevante Anwendungen, z.B. im Bereich Cloud-Computing. Ein sehr bekanntes homomorphes Kryptosystem ist das von Paillier. Aber auch ältere Kryptosysteme wie das von ElGamal oder RSA besitzen homomorphe Eigenschaften.

9.2 Ursprung und Begriff „homomorph“

Zunächst klären wir den Ursprung des Begriffs„homomorph“. Dieser stammt aus der Mathematik: Hier bezeichnet ein Homomorphismus eine Struktur-erhaltende Abbildung zwischen zwei algebraischen Strukturen. Umgangssprachlich gesagt bildet ein Homomorphismus $f : X \to Y$ dabei die Struktur von X auf die von Y ab. An einem Beispiel lässt sich dies sehr gut verdeutlichen. Seien $(X, +)$ und $(Y, *)$ zwei Gruppen mit den Operationen $+$ bzw. $*$. Ein Homomorphismus $f : X \to Y$ bildet nun jedes $x \in X$ so auf ein $y \in Y$ ab, dass gilt:

$$f(x_1 + x_2) = f(x_1) * f(x_2)$$

für beliebige x_1, x_2 aus X. Es spielt also für die beiden Werte x_1, x_2 keine Rolle, ob man sie zunächst addiert (Gruppenoperation von X) und dann f anwendet (linke Seite der Gleichung); oder ob man zuerst f auf die beiden Werte x_1, x_2 anwendet, und dann die Gruppenoperation von Y, die Multiplikation, anwendet. Die Operationen $+$ bzw. $*$ wurden hier nur beispielhaft verwendet, sie hängen immer von der jeweiligen Gruppe ab. Beispielsweise gibt es auch Homomorphismen zwischen Gruppen mit derselben Gruppenoperation.

Beispiel: Nehmen wir für X die Menge der ganzen Zahlen $\mathbb{Z}$, diese bilden zusammen mit der Addition eine Gruppe $G_1 = (\mathbb{Z}, +)$. Genauso bilden die reellen Zahlen ohne Null zusammen mit der Multiplikation eine Gruppe $G_2 = (\mathbb{R}\backslash\{0\}, *)$. Die Funktion $f : \mathbb{Z} \to \mathbb{R}\backslash\{0\}, z \to e^z$ ist ein Homomorphismus, denn für alle $z_1, z_2 \in \mathbb{Z}$ gilt: $f(z_1 + z_2) = e^{(z_1+z_2)} = f(z_1) * f(z_2)$. Die Funktion $f : \mathbb{Z} \to \mathbb{R}\backslash\{0\}, z \to z^2$ dagegen ist kein Gruppenhomomorphismus.

9.3 Entschlüsselungsfunktion ist Homomorphismus

Wir betrachten im Folgenden Public-Key-Kryptosysteme mit einer besonderen Eigenschaft: Eine Public-Key-Chiffre wird homomorph genannt, wenn ihre Entschlüsselungsfunktion ein Homomorphismus ist.

Sei nun angenommen, der obige Homomorphismus f sei die Entschlüsselungsfunktion eines Kryptosystems. Das bedeutet, dass wir in der Algebra der Geheimtexte Operationen durchführen können und dabei wissen, welche Auswirkungen dies auf die Klartexte hat. Angewendet auf das obige Beispiel:

Y ist die Menge der Geheimtexte, X die Menge der Klartexte. Für zwei Klartexte x_1, x_2 mit zugehörigen Geheimtexten y_1, y_2 gilt:

$$f(y_1 * y_2) = f(y_1) + f(y_2) = x_1 + x_2$$

Übersetzt bedeutet diese Gleichung: Multipliziere ich zwei Geheimtexte y_1, y_2 miteinander und entschlüssele dann deren Produkt, so erhalte ich die Summe der ursprünglich verschlüsselten Werte x_1 und x_2. Jedermann kann – ohne Kenntnis der Klartexte und ohne Kenntnis der Entschlüsselungsfunktion – ein Produkt zweier Geheimtexte berechnen und weiß, dass der autorisierte Entschlüsseler aus dem berechneten Produkt die Summe der beiden ursprünglichen Klartexte erhalten wird.

9.4 Beispiele für homomorphe Chiffren

9.4.1 Paillier-Kryptosystem

Das wohl bekannteste Kryptosystem mit solchen homomorphen Eigenschaften ist das von Paillier[Pai99]. Wir sehen zunächst, wie die Schlüsselerzeugung, die Verschlüsselung und die Entschlüsselung funktionieren, und zeigen dann, dass das Paillier-Kryptosystem homomorphe Eigenschaften besitzt.

9.4.1.1 Schlüsselerzeugung

Zuerst werden zwei zufällige Primzahlen p, q erzeugt, so dass das Produkt $n = pq$ einen gültigen RSA-Modulus formt. Hierbei sollte n eine Bitlänge von mindestens 1024 Bit haben. Damit kann der private Schlüssel $\lambda = kgV(p-1, q-1)$ berechnet werden. kgV bezeichnet hierbei das kleinste gemeinsame Vielfache. Der öffentliche Schlüssel besteht nur aus dem RSA-Modulus n.

9.4.1.2 Verschlüsselung

Sei m die zu verschlüsselnde Nachricht aus dem Klartextraum $\mathbb{Z}_n$. Für jeden Verschlüsselungsvorgang wählen wir zunächst ein zufälliges Element r aus $\mathbb{Z}_n$ und berechnen dann mit Hilfe des öffentlichen Schlüssels n den Geheimtext:

$$c = E(m, r) = (n+1)^m * r^n \bmod n^2$$

9.4.1.3 Entschlüsselung

Sind der private Schlüssel λ und ein Geheimtext $c \in \mathbb{Z}_{n^2}^*$ gegeben, berechnen wir zunächst

$$S = c^\lambda \bmod n^2 \text{ und } T = \phi(n)^{(-1)} \bmod n^2,$$

wobei ϕ die Eulersche Funktion ist. Und dann $m = D(c) = (S-1)/n * T \bmod n$.

9.4.1.4 Homomorphe Eigenschaft

Um die homomorphe Eigenschaft nachzuweisen, betrachten wir die Verschlüsselungsfunktion E und die Entschlüsselungsfunktion D des Paillier-Kryptosystems. Zur Vereinfachung setzen wir im Folgenden $g := n + 1$. Aus zwei Klartexten m_1, m_2 ergeben sich die dazugehörigen Geheimtexte c_1, c_2 als

$$c_1 = g^{m_1} * {r_1}^n \bmod n^2 \text{ bzw. } c_2 = g^{m_2} * {r_2}^n \bmod n^2$$

Wir sehen, dass für das Produkt $c_3 = c_1 * c_2$ gilt

$$c_3 = (g^{m_1} * {r_1}^n \bmod n^2) * (g^{m_2} * {r_2}^n \bmod n^2) = g^{m_1+m_2} * (r_1 * r_2)^n \bmod n^2 = E(m_1 + m_2, r_1 * r_2)$$

Das Produkt zweier Geheimtexte ist also wieder ein Geheimtext, und zwar eine Verschlüsselung der Summe der ursprünglichen Nachrichten. Nun ist es leicht zu sehen, dass die Entschlüsselungsfunktion ein Homomorphismus ist: Gegeben zwei Klartexte m_1, m_2 dann gilt

$$D(E(m_1, r_1) * E(m_2, r_2)) = D(E(m_1 + m_2, r_1 r_2)) = m_1 + m_2 = D(E(m_1, r_1)) + D(E(m_2, r_2))$$

9.4.2 Weitere Kryptosysteme

Auch ältere Public-Key-Kryptosysteme können homomorphe Eigenschaften haben. Das ElGamal-Kryptosystem und das Standard RSA-Kryptosystem sind bekannte Beispiele dafür. Wir zeigen diese homomorphen Eigenschaften anhand einfacher Beispiele auf.

9.4.2.1 RSA

Sei (e, n) der öffentliche RSA-Schlüssel (e der Verschlüsselungskoeffizient, n der RSA-Modulus). Für zwei Nachrichten m_1, m_2 erhält man die Verschlüsselungen $c_1 = {m_1}^e \bmod n$ und $c_2 = {m_2}^e \bmod n$. Nun gilt für das Produkt dieser beiden Verschlüsselungen: $c_1 * c_2 = {m_1}^e * {m_2}^e \bmod n = (m_1 * m_2)^e \bmod n$. Man erhält also eine Verschlüsselung des Produkts der ursprünglichen Nachrichten. Wie man leicht nachprüfen kann gilt diese Eigenschaft für beliebige Nachrichten m_1, m_2, somit ist die Entschlüsselungsfunktion ein Homomorphismus. RSA ist dabei ein Beispiel für einen Homomorphismus, bei dem in beiden Gruppen die gleiche Gruppenoperation verwendet wird.

9.4.2.2 ElGamal

Ähnlich wie bei RSA verhält es sich auch im ElGamal-Kryptosystem. Sei (p, g, K) der öffentliche Schlüssel, der private Schlüssel sei k (es gilt also $g^k \bmod p = K$). Für Nachrichten m_1, m_2 erhält man nun Verschlüsselungen $(R, c_1) = (K^r \bmod p, m_1 * g^r \bmod p)$ und $(S, c_2) = (K^s \bmod p, m_2 * g^s \bmod p)$. Auch hier ist das Produkt $(R * S, c_1 * c_2)$ eine Verschlüsselung von $m_1 * m_2$ und man kann leicht überprüfen, dass die Entschlüsselungsfunktion ein Homomorphismus ist.

9.5 Anwendungen

Die homomorphe Eigenschaft lässt sich dazu verwenden, um verschlüsselte Werte zu addieren oder verschlüsselte Werte mit unverschlüsselten Werten zu multiplizieren (dies entspricht einer wiederholten Anwendung der Addition). Damit werden homomorphe Chiffren zu einer wichtigen Funktion in vielen kryptographischen Anwendungen.

1. Eine dieser Anwendungen ist das sogenannte „Electronic Voting“. Hierbei wird es mehreren Wahlberechtigten ermöglicht, ihre Stimme verschlüsselt abzugeben. Dies ist wichtig in Situationen, in denen die Wahlberechtigten nicht direkt zusammen kommen können. Zum Beispiel könnte es sein, dass die Wahlberechtigten nur per Email über das Internet kommunizieren können. Wenn die Abstimmung geheim bleiben soll, und es niemanden gibt, dem alle Wahlberechtigten uneingeschränkt vertrauen, bieten homomorphe Chiffren eine gute Lösung für dieses Problem. Im Wesentlichen funktioniert Electronic Voting mittels homomorpher Chiffren so:

   - Alle Wahlberechtigen (links in der Abbildung 9.1) verschlüsseln ihre Stimme. Sie verschlüsseln den Wert 1, wenn sie für die Entscheidung sind, und den Wert 0, wenn sie dagegen sind.
   - Über die homomorphe Eigenschaft wird die Summe der abgegebenen Stimmen berechnet. Da dies auf den verschlüsselten Werten passiert, bleiben die Stimmen der einzelnen Wahlberechtigten geheim.
   - Am Ende werden die Stimmen ausgezählt, indem nur die Summe der Stimmen entschlüsselt wird.

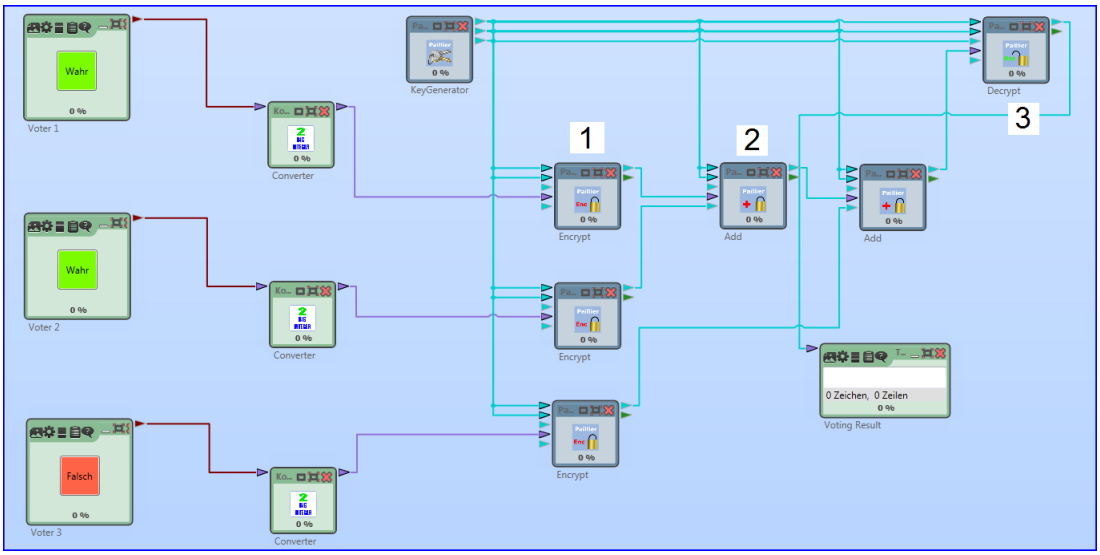


Abbildung 9.1: Voting-Beispiel für Paillier

2. Ein weiteres Anwendungsgebiet für homomorphe Chiffren ist „Secure Multiparty Computation“. Hierbei berechnen mehrere Parteien gemeinsam eine vorgegebene Funktion. Jede der Parteien steuert einen Input für die zu berechnende Funktion bei. Das Ziel der Berechnung ist es, alle Inputs und auch die Zwischenergebnisse geheim zu halten, während nur

das Ergebnis der Funktion bekannt wird. Die Verwendung homomorpher Chiffren hilft dabei, diese Berechnungen auf verschlüsselten Daten durchzuführen. Da sich allerdings unter der homomorphen Chiffre von Paillier nur Additionen (und z.B. keine Multiplikationen durchführen lassen), müssen noch weitere geeignete Methoden verwendet werden. Einen guten Einstieg in dieses Thema bietet Wikipedia [Wikb].

3. Weiterhin wird erwartet, dass homomorphe Chiffren im Bereich Cloud Computing enorme Vorteile bringen können. Mittels sogenannter voll-homomorpher Kryptosysteme [Wika] wird es möglich sein, komplette Anwendungen auf verschlüsselten Daten durchzuführen. Hierzu ist es notwendig, dass unter der homomorphen Verschlüsselung die beiden Operationen Addition und Multiplikation durchgeführt werden können (im Gegensatz zum Paillier-Kryptosystem, welches nur die Addition unterstützt). Ein solches Kryptosystem wurde erstmals 2009 von Craig Gentry vorgestellt [Gen09].

9.6 Homomorphe Chiffren in CrypTool

9.6.1 CrypTool 2

In CrypTool 2 findet man bereits eine Implementierung des Paillier-Kryptosystems (siehe Bild 9.2). Unter den fertigen Vorlagen finden sich Methoden zur Erzeugung der kryptographischen Schlüssel (Paillier Key Generator), ein Beispiel für eine Ver- und Entschlüsselung mittels Paillier (Paillier Text), sowie Beispiele, die die homomorphen Eigenschaften von Paillier aufzeigen (Paillier Addition, Paillier Blinding und Paillier Voting).

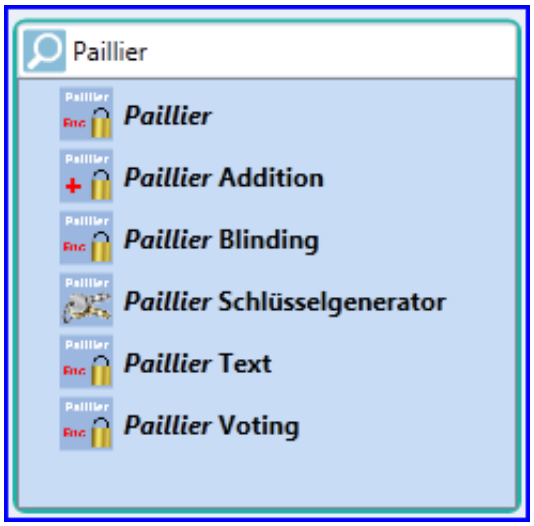


Abbildung 9.2: Paillier-Kryptosystem in CrypTool 2 (CT2)

9.6.2 JCrypTool

Im JCrypTool gibt es eine Implementierung (siehe Bild 9.3), die die homomorphen Eigenschaften verschiedener Kryptosysteme visualisiert. Für RSA und Paillier wird gezeigt, dass jeweils Multiplikationen (für RSA) und Additionen (für Paillier) auf verschlüsselten Werten möglich sind. Für das voll-homomorphe Kryptosystem von Gentry können sowohl Multiplikationen als auch Additionen auf verschlüsselten Werten durchgeführt werden.

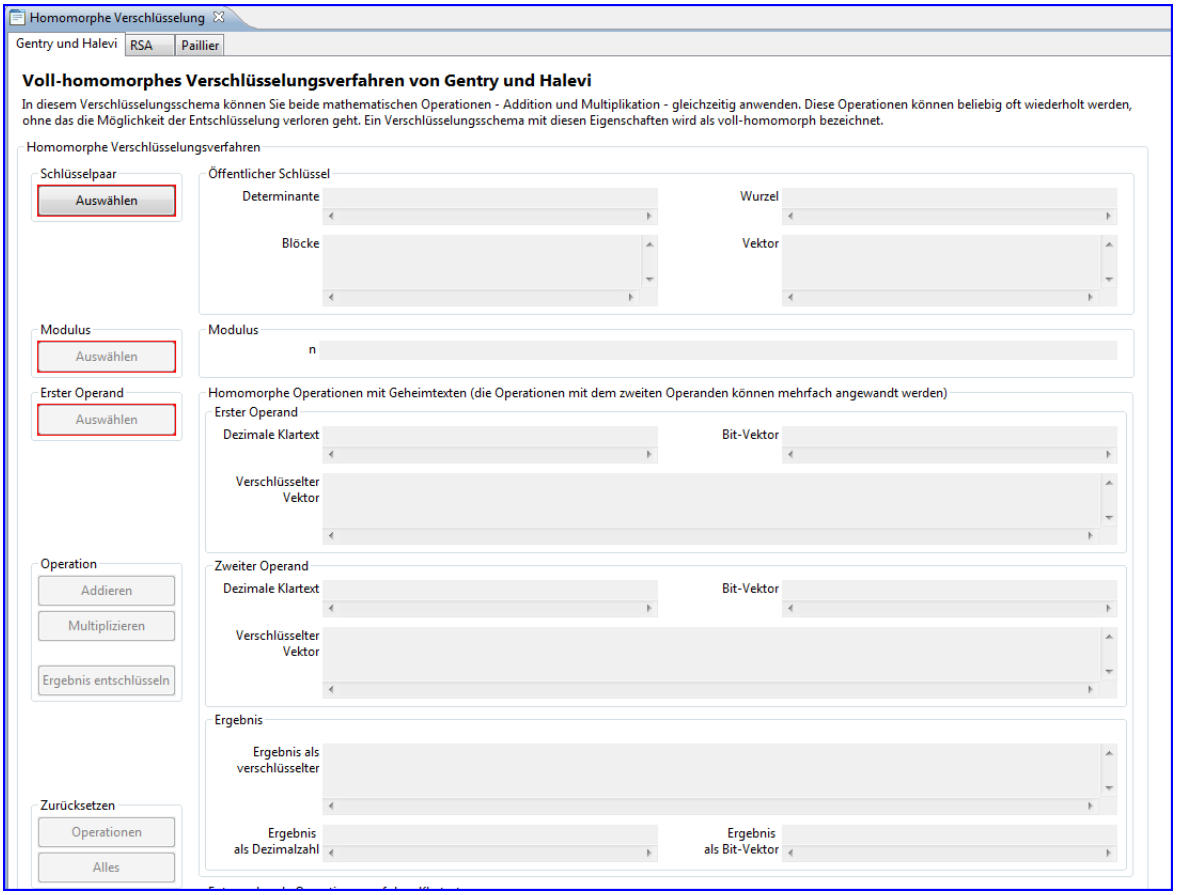


Abbildung 9.3: Kryptosysteme mit homomorphen Eigenschaften in JCrypTool (JCT)

Literaturverzeichnis (Kap. HE)

[Gen09] Gentry, Craig: *Fully Homomorphic Encryption Using Ideal Lattices.* In: *41st ACM Symposium on Theory of Computing (STOC)*, 2009.

[Pai99] Paillier, Pascal: *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes.* In: *Advances in Cryptology – EUROCRYPT'99*, 1999.

[Wika] Wikipedia: *Homomorphic Encryption & Homomorphismus.* `https://en.wikipedia.org/wiki/Homomorphic_encryption`, `https://de.wikipedia.org/wiki/Homomorphismus`.

[Wikb] Wikipedia: *Secure Multiparty Computation.* `http://en.wikipedia.org/wiki/Secure_multi-party_computation`.

Alle Links im Artikel wurden am 15.07.2016 überprüft.

Kapitel 10

Studie über aktuelle akademische Resultate für das Lösen diskreter Logarithmen und zur Faktorisierung – und wie man in der Praxis reagiert

(Antoine Joux, Arjen Lenstra & Alexander May; April 2014)

(Übersetzung ins Deutsche: Patricia Wienen, 2016)

Abstract: Neuste algorithmische Entwicklungen für das Lösen diskreter Logarithmen in endlichen Körpern mit kleiner Charakteristik führten zu einer gewissen (publizistisch beförderten) Unsicherheit bei kryptographischen Nutzern bezüglich des Einflusses auf die Sicherheit kürzlich entwickelter Kryptoverfahren (siehe dazu beispielsweise die Diskussion in [PRSS13] unter dem Stichwort „Cryptocalypse“). Dieses Kapitel gibt einen Überblick über die zur Zeit besten Algorithmen für das Berechnen diskreter Logarithmen in verschiedenen Gruppen und über den Status des Faktorisierungsproblems. Unser Ziel ist zu klären, was im Moment algorithmisch machbar ist und wofür erst weitere bedeutende Durchbrüche nötig sind. Zusammenfassend sehen wir im Moment keinen Weg, wie man die bestehenden algorithmischen Prozesse für endliche Körper mit kleiner Charakteristik entweder auf Körper mit großer Charakteristik oder auf das ganzzahlige Faktorisierungsproblem erweitern könnte.

10.1 Generische Algorithmen für das Dlog-Problem in beliebigen Gruppen

Management Summary: Die Widerstandsfähigkeit des diskreten Logarithmus-Problems hängt von der Gruppe ab, über die es definiert ist. In diesem Kapitel betrachten wir kryptoanalytische Algorithmen, die für beliebige Gruppen funktionieren. Aus kryptographischem Blickwinkel ist es erstrebenswert, solche Gruppen zu identifizieren, für die sich keine besseren Algorithmen finden lassen. Ein Kandidat dafür sind Gruppen über Elliptischen Kurven.

In diesem Kapitel beschreiben wir *allgemeine* kryptoanalytische Algorithmen, die sich auf *jede* endliche abelsche Gruppe anwenden lassen. Das heißt, jede in der Kryptographie verwendete Gruppe – z. B. multiplikative Gruppen über endlichen Körpern oder über Elliptischen Kurven – ist für diese Algorithmen anfällig. Wir werden sehen, dass wir mit der Pollard-Rho-Methode in einer Gruppe der Ordnung n immer einen diskreten Logarithmus in $\mathcal{O}(\sqrt{n})$ Schritten berechnen können. Umgekehrt bedeutet das, dass man eine Gruppe mit einer Ordnung von mindestens 2^{2k} wählen muss, um ein Sicherheitslevel von 2^k zu erreichen. Z.B. muss man für ein Sicherheitslevel von 80 Bits eine Gruppe der Ordnung 160 Bits oder mehr wählen. Das erklärt, warum wir in der Praxis üblicherweise Gruppen über Elliptischen Kurven mit einer Ordnung von mindestens 160 Bits wählen.

Des Weiteren sei G eine Gruppe der Ordnung n und sei $n = p_1^{e_1} \cdot \ldots \cdot p_\ell^{e_\ell}$ die Primfaktorzerlegung von n. Wir werden sehen, dass diskrete Logarithmen in G in Zeit $\mathcal{O}(e_1\sqrt{p_1} + \ldots + e_\ell\sqrt{p_\ell})$ berechnet werden können. Man bemerke, dass diese Beschränkung äquivalent ist zu Pollard's Beschränkung $O(\sqrt{n})$ g.d.w. n prim ist. Andernfalls wird die Komplexität der Berechnung des diskreten Logarithmus hauptsächlich beschränkt durch die Größe des größten Primteilers der Gruppenordnung. Dies erklärt, warum z.B. Schnorr-/DSA-Signaturen in Gruppen implementiert werden, die per Konstruktion einen Primfaktor von mindestens 160 Bit Länge enthalten. Das erklärt außerdem, warum Gruppen über Elliptischen Kurven üblicherweise primer Ordnung sind oder deren Ordnung nur einen sehr kleinen **glatten Cofaktor** enthält.

10.1.1 Die Pollard-Rho-Methode

Sei G eine endliche abelsche Gruppe. Sei g ein Generator einer großen Untergruppe $G' = \{g, g^2, \ldots, g^n\} \subseteq G$ (z.B. könnte g die Gruppe G selbst generieren). Sei $y = g^x$. Dann beschreibt das diskrete Logarithmus-Problem den Versuch, bei Eingabe von g und y den Wert $x \mod n$ auszugeben. Wir schreiben $x = \mathrm{dlog}_g(y)$.

Die Pollard-Rho-Methode versucht, Elemente $g^{a_i}y^{b_i} \in G'$ mit $a_i, b_i \in \mathbb{N}$ in einer pseudozufälligen, aber deterministischen Art und Weise zu erzeugen. Der Einfachheit halber nehmen wir an, dass wir zufällige Elemente von den n Elementen in G' erzeugen. Dann erwarten wir wegen des Geburtstagsparadoxons nach höchstens $\mathcal{O}(\sqrt{n})$ Schritten zwei identische Elemente zu erhalten. In unserem Fall bedeutet das

$$g^{a_i}y^{b_i} = g^{a_j}y^{b_j}.$$

Dies kann umgeschrieben werden als $g^{\frac{a_i-a_j}{b_j-b_i}} = y$. Daraus wiederum folgt, dass wir unseren diskreten Logarithmus aus $x \equiv \frac{a_i-a_j}{b_j-b_i} \mod n$ erhalten können.

Somit kann man mit der Pollard-Rho-Methode den diskreten Logarithmus in jeder endlichen abelschen Gruppe der Ordnung n in $\mathcal{O}(\sqrt{n})$ Schritten berechnen. Durch die Nutzung

von Techniken zum Auffinden von Schleifen (sogenannter cycle-finding techniques) kann man außerdem zeigen, dass die Pollard-Rho-Methode mit konstantem Speicherbedarf implementiert werden kann.

Außerdem ist es auch möglich, die Effizienz von Quadratwurzel-Algorithmen zu verbessern, wenn mehrere diskrete Logarithmen in derselben Gruppe erwünscht sind: Bei der Berechnung von L verschiedenen Logarithmen kann man die globalen Kosten von $\mathcal{O}(L\sqrt{n})$ auf $\mathcal{O}(\sqrt{Ln})$ reduzieren [FJM14].

10.1.2 Der Silver-Pohlig-Hellman-Algorithmus

Wie zuvor sei $y = g^x$ für einen Generator g der Ordnung n. Wir wollen den diskreten Logarithmus $x \bmod n$ berechnen. Außerdem sei $n = p_1^{e_1} \cdot \ldots \cdot p_\ell^{e_\ell}$ die Primfaktorzerlegung von n. Dann gilt nach dem Chinesischen Restsatz, dass $x \bmod n$ eindeutig definiert wird durch folgendes System von Kongruenzen:

$$\begin{aligned} x &\equiv x_1 \bmod p_1^{e_1} \\ &\vdots \\ x &\equiv x_\ell \bmod p_\ell^{e_\ell}. \end{aligned} \tag{10.1}$$

Der Algorithmus von Silver-Pohlig-Hellman berechnet alle diskreten Logarithmen $x_i \bmod p_i$ in den Untergruppen mit den Ordnungen p_i in $\mathcal{O}(\sqrt{p_i})$ Schritten durch Verwendung der Pollard-Rho-Methode. Danach ist es relativ einfach einen Logarithmus modulo der Primzahlpotenz $x_i \bmod p_i^{e_i}$ mittels eines **Hensel-Lift-Prozesses** zu finden, der e_i Aufrufe an die diskrete Logarithmus-Prozedur modulo p_i ausführt. In einem **Hensel-Lift-Prozess** starten wir mit einer Lösung $x_i \bmod p_i$ und berechnen dann nacheinander $x_i \bmod p_i^2$, $x_i \bmod p_i^3$ usw. bis zu $x_i \bmod p_i^{e_i}$ (siehe [May13] für Hensels Schema).

Schlussendlich berechnet man den gewünschten diskreten Logarithmus $x \bmod n$ aus dem obigen System von Gleichungen (10.1) über den Chinesischen Restsatz. Insgesamt wird die Laufzeit hauptsächlich festgelegt durch die Berechnung von $x_i \bmod p_i$ für den größten Primfaktor p_i. Damit ist die Laufzeit ungefähr $\mathcal{O}(\max_i\{\sqrt{p_i}\})$.

10.1.3 Wie man Laufzeiten misst

Im Verlauf dieser Studie wollen wir die Laufzeit von Analyse-Algorithmen für diskrete Logarithmen abschätzen als Funktion der Bitgröße n. Es gilt, dass jede Ganzzahl n mit (ungefähr) $\log n$ Bits geschrieben werden kann, wobei der Logarithmus zur Basis 2 gewählt wird. Die *Bitgröße* von n ist damit $\log n$.

Um unsere Laufzeiten auszudrücken nutzen wir die Notation $L_n[b, c] = \exp^{c\cdot(\ln n)^b(\ln\ln n)^{1-b}}$ für Konstanten $b \in [0, 1]$ und $c > 0$. Bemerke, dass $L_n[1, c] = e^{c\cdot\ln n} = n^c$ eine Funktion ist, die für konstantes c polynomiell in n ist. Deshalb sagen wir, dass $L_n[1, c]$ *polynomiell* in n ist. Bemerke außerdem, dass $L_n[1, c] = n^c = (2^c)^{\log_2 n}$ eine in $\log n$ exponentielle Funktion ist. Deshalb sagen wir, dass $L_n[1, c]$ *exponentiell* in der Bitgröße $\log n$ von n ist. Damit erreicht unser Pollard-Rho-Algorithmus eine erwartete Laufzeit von $L[1, \frac{1}{2}]$.

Auf der anderen Seite ist $L_n[0, c] = e^{c\cdot\ln\ln n} = (\ln n)^c$ *polynomiell* in der Bitgröße von n. Merke, dass der erste Parameter b für die Laufzeit wichtiger ist als der zweite Parameter c, da b zwischen polynomieller und exponentieller Laufzeit interpoliert. Als Kurzschreibweise definieren wir $L_n[b]$, wenn wir die Konstante c nicht spezifizieren wollen.

Einige der wichtigsten Algorithmen, die wir in den nachfolgenden Kapitels besprechen, errei-

chen eine Laufzeit von $L_n[\frac{1}{2} + o(1)]$ oder $L_n[\frac{1}{3} + o(1)]$ (wobei das $o(1)$ für $n \to \infty$ verschwindet), welche eine Funktion ist die schneller wächst als jedes Polynom, aber langsamer als exponentiell. Für kryptographische Verfahren sind solche Angriffe völlig akzeptabel, da das gewünschte Sicherheitslevel einfach erreicht werden kann durch moderates Anpassen der Schlüsselgrößen.

Der aktuelle Algorithmus von Joux et al. für die Berechnung diskreter Logarithmen in endlichen Körpern mit kleiner Charakteristik erreicht jedoch eine Laufzeit von $L_n[o(1)]$, wobei $o(1)$ gegen 0 geht für $n \to \infty$. Das bedeutet, dass diese Algorithmen in quasi-polynomieller Zeit laufen, und dass die zugrunde liegenden Körper nicht länger akzeptabel sind für kryptographische Anwendungen. Ein endlicher Körper $\mathbb{F}_{p^n}$ hat eine kleine Charakteristik, wenn p klein ist, d.h. der Basiskörper (base field) $\mathbb{F}_p$ ist klein und der Grad n der Körpererweiterung ist üblicherweise groß. In den aktuellen Algorithmen brauchen wir ein kleines p, da diese Algorithmen über alle p Elemente im Basiskörper $\mathbb{F}_p$ laufen.

10.1.4 Unsicherheit durch Quantencomputern

In 1995 veröffentlichte Shor einen Algorithmus für die Berechnung diskreter Logarithmen und Faktorisierungen auf einem Quantencomputer. Er zeigte, dass die Berechnung diskreter Logarithmen in *jeder* Gruppe der Ordnung n in polynomieller Zeit durchgeführt werden kann, die fast $\mathcal{O}(\log n^2)$ entspricht. Dieselbe Laufzeit gilt für die Berechnung der Faktorisierung einer Ganzzahl n. Diese Laufzeit ist nicht nur polynomiell, die Angriffe sind sogar noch effizienter als die kryptographischen Verfahren selbst! Das wiederum bedeutet, dass das Problem nicht durch bloßes Anpassen der Schlüsselgrößen behoben werden kann.

Sollten wir also in den nächsten Jahrzehnten die Entwicklung groß angelegter Quantencomputer miterleben, muss folglich die ganze klassische, auf Dlog oder Faktorisierung basierende Kryptographie ersetzt werden. Man sollte allerdings betonen, dass die Konstruktion großer Quantencomputer mit vielen Qubits sehr viel schwieriger zu sein scheint als die seines klassischen Gegenstücks, da die meisten kleinen Quantensysteme nicht gut skalieren und ein Problem mit Dekohärenz haben.

Empfehlung: Es scheint schwierig zu sein, die Entwicklungen in der Konstruktion von Quantencomputern vorherzusagen. Experten der Quantenphysik sehen aber derzeit keine unüberwindbaren Hindernisse, die die Entwicklung großer Quantencomputer auf lange Sicht behindern würden. Es dürfte sehr wichtig sein, den aktuellen Fortschritt in diesem Gebiet im Blick zu behalten und in den nächsten 15 Jahren alternative quanten-resistente Kryptoverfahren zur Hand zu haben.

Referenzen und weiterführende Literatur: Wir empfehlen die Bücher von Menezes, van Oorschot und Vanstone [MvOV01], Joux [Jou09] und Galbraith [Gal12] zur Studie kryptoanalytischer Techniken. Einen einführenden Kurs in Kryptoanalyse stellt Mays Vorlesungsskript zur Kryptoanalyse zur Verfügung [May08, May12]. Eine Einleitung zu Quantenalgorithmen findet sich in den Büchern von Homeister [Hom07] und Mermin [Mer08].

Die Algorithmen in diesem Kapitel wurden ursprünglich in den hervorragenden Arbeiten von Pollard [Pol75, Pol00] und Shor [Sho94] vorgestellt. Generische Algorithmen für viele Dlogs wurden kürzlich untersucht in [FJM14].

10.2 Beste Algorithmen für Primkörper $\mathbb{F}_p$

Management Summary: Primkörper $\mathbb{F}_p$ sind – neben Elliptischen Kurven – die Standardgruppe für das diskrete Logarithmus-Problem. In den letzten 20 Jahren gab es für diese Gruppen keinen signifikanten algorithmischen Fortschritt. Sie sind immer noch eine gute Wahl für Kryptographie.

In Kapitel 10.1 haben wir gelernt, dass wir in jeder endlichen abelschen Gruppe der Ordnung n den diskreten Logarithmus in $\mathcal{O}(\sqrt{n})$ Schritten bestimmen können. Merke, dass sowohl die Pollard-Rho-Methode als auch der Silver-Pohlig-Hellman-Algorithmus aus Kapitel 10.1 keine andere Eigenschaften der *Repräsentation* von Gruppenelementen nutzen als ihre Eindeutigkeit. In diesen Methoden werden Gruppenelemente einfach durch Gruppenoperationen und Test auf Gleichheit von Elementen berechnet. Algorithmen dieser Art werden in der Literatur als *generisch* bezeichnet.

Es ist bekannt, dass generische Algorithmen diskrete Logarithmen nicht in besserer Zeit als der Silver-Pohlig-Hellman-Algorithmus [Sho97] berechnen können. Damit können die Algorithmen aus Kapitel 10.1 als optimal betrachtet werden, wenn keine weitere Information über die Gruppenelemente bekannt ist.

Wenn wir unsere Gruppe G spezifizieren als die multiplikative Gruppe über den endlichen Körper $\mathbb{F}_p$ mit p prim, können wir sogar die Repräsentation der Gruppenelemente ausnutzen. Natürliche Repräsentanten von $\mathbb{F}_p$ sind die Ganzzahlen $0, \ldots, p-1$. Damit können wir z.B. die Primfaktorzerlegung dieser Ganzzahlen verwenden. Dies wird gemacht in den Algorithmen des sogenannten Typs *Index-Calculus* für diskrete Logarithmen. Dieser Typ von Algorithmen bildet derzeit die Klasse mit den besten Laufzeiten für diskrete Logarithmen über Primkörper, prime Körpererweiterungen (Kapitel 10.3) und für das Faktorisierungsproblem (Kapitel 10.4).

Wir werden jetzt einen Index-Calculus-Algorithmus an Hand eines sehr einfachen Beispiels veranschaulichen.

10.2.1 Eine Einleitung zu Index-Calculus-Algorithmen

Ein Index-Calculus-Algorithmus besteht aus drei grundlegenden Schritten.

Faktorbasis: Definition der Faktorbasis $F = \{f_1, \ldots, f_k\}$. Wir wollen die Gruppenelemente ausdrücken als Potenzen von Elementen der Faktorbasis.

Relationen finden: Finde Elemente $z_i := g^{x_i} \in G$ für eine ganze Zahl x_i, die mit der Faktorbasis bestimmt werden können. Das bedeutet

$$g^{x_i} = \prod_{j=1}^{k} f_j^{e_{ij}}.$$

Schreiben wir diese Gleichung zur Basis g erhalten wir eine *Relation*

$$x_i \equiv \sum_{j=1}^{k} e_{ij} \mathrm{dlog}_g(f_j) \bmod n,$$

wobei n die Ordnung von g ist. Dies ist eine lineare Gleichung in den k Unbekannten $\mathrm{dlog}_g(f_1), \ldots, \mathrm{dlog}_g(f_k)$. Sobald wir k linear unabhängige Relationen dieses Typs haben,

können wir die Unbekannten mit Linearer Algebra berechnen. Das bedeutet, dass wir erst alle diskreten Logarithmen der Faktorbasis berechnen müssen, bevor wir den gewünschten individuellen Logarithmus von y bestimmen.

Dlog-Berechnung: Drücke $yg^r = g^{x+r} = \prod_{j=1}^{k} f_j^{e_j}$ in der Faktorbasis für eine ganze Zahl r aus. Das gibt uns eine neue Relation

$$x + r \equiv \sum_{j=1}^{k} e_j \text{dlog}_g(f_j) \bmod n,$$

die in der einzigen Unbekannten $x = \text{dlog}_g y$ einfach gelöst werden kann.

Lassen Sie uns ein einfaches Beispiel für einen Index-Calculus-Algorithmus geben, das $x = \text{dlog}_2(5)$ in $\mathbb{F}_{11}^*$ berechnet. Da 2 die multiplikative Gruppe $\mathbb{F}_{11}^*$ generiert, ist 2 von der Ordnung 10.

Faktorbasis: Definiere $F = \{-1, 2\}$.

Relationen finden: $2^1 = (-1)^0 2^1$ gibt uns eine erste triviale Relation

$$1 \equiv 0 \cdot \text{dlog}_2(-1) + 1 \cdot \text{dlog}_2(2) \bmod 10.$$

Wenn wir $2^6 = 64 \equiv -2 \bmod 11$ berechnen, erhalten wir eine zweite Relation

$$6 \equiv 1 \cdot \text{dlog}_2(-1) + 1 \cdot \text{dlog}_2(2) \bmod 10.$$

Damit können wir das System von linearen Gleichungen lösen:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \text{dlog}_2(-1) \\ \text{dlog}_2(2) \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 6 \end{pmatrix} \bmod 10.$$

Als eindeutige Lösung erhalten wir $\text{dlog}_2(-1) \equiv 5$ und $\text{dlog}_2(2) \equiv 1$.

Dlog-Berechnung: Wegen $5 \cdot 2^1 = 10 \equiv -1 \bmod 11$ erhalten wir

$$x + 1 \equiv 1 \cdot \text{dlog}(-1) + 0 \cdot \text{dlog}(2) \bmod 10.$$

Dies führt zu der Lösung $x \equiv 4 \bmod 10$.

Laufzeit: Eine große Faktorbasis zu wählen macht es einfacher, Relationen zu finden, da es die Wahrscheinlichkeit erhöht, dass sich eine bestimmte Zahl in der Faktorbasis aufteilt. Auf der anderen Seite müssen wir für eine große Faktorbasis mehr Relationen finden, um die Dlogs aller Faktorbasiselemente zu berechnen. Eine Optimierung dieses Kompromisses führt zu einer Laufzeit von $L_p[\frac{1}{2}]$ für den „relation finding"-Schritt und ebenfalls $L_p[\frac{1}{2}]$ für die Berechnung des individuellen diskreten Logarithmus in Schritt 3.

Lasst uns kurz die Vor- und Nachteile des obigen, simplen Index-Calculus-Algorithmus aus der Sicht eines Kryptoanalysten diskutieren.

Vorteile:

- Für $g^{x_i} = \prod_{j=1}^{k} f_j^{e_{ij}}$ ist es trivial, den diskreten Logarithmus auf der linken Seite zu berechnen.

Nachteile:

- Wir müssen relativ große Zahlen g^{x_i} über die ganzen Zahlen mit einbeziehen. Man kann zeigen, dass dies zwangsläufig zu einer Laufzeit von $L_p[\frac{1}{2}]$ führt, und dass es keine Hoffnung gibt, unter die Konstante $\frac{1}{2}$ zu kommen.
- Wir müssen alle diskreten Logarithmen für die Faktorbasiselemente berechnen. Dies ist allen Index-Calculus-Algorithmen zu eigen.

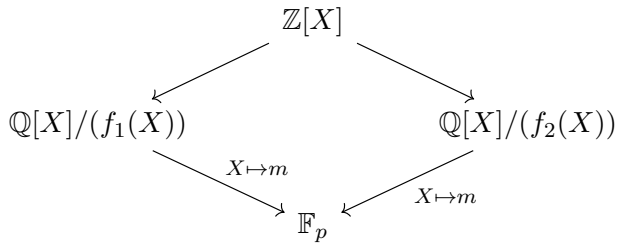
Wir werden den ersten Nachteil eliminieren, indem wir Faktorisierung über Zahlkörpern erlauben. Der zweite Nachteil wird eliminiert, indem man eine Faktorbasis wählt, bei der man die Dlogs ihrer Elemente sehr effizient berechnen kann.

10.2.2 Das Zahlkörpersieb zur Berechnung des Dlog[1]

Ein Zahlkörper $\mathbb{Q}[\alpha]$ ist ein k-dimensionaler Vektorraum über $\mathbb{Q}$ und kann erzeugt werden durch Anfügen einer Nullstelle α eines irreduziblen Polynoms f vom Grad k an $\mathbb{Q}$. Das bedeutet, wir können jedes Element von $\mathbb{Q}[\alpha]$ schreiben als $a_0 + a_1\alpha + \ldots a_{k-1}\alpha^{k-1}$ mit $a_i \in \mathbb{Q}$. Wenn wir die a_i auf die ganzen Zahlen beschränken, befinden wir uns im Ring $\mathbb{Z}[\alpha]$.

Das Zahlkörpersieb ist ebenfalls ein Index-Calculus-Algorithmus. Verglichen mit dem vorigen Ansatz hat er den Vorteil, kleinere Zahlen zu verwenden. Das wird erreicht durch die Wahl einer spezifischen Repräsentation des Primkörpers $\mathbb{F}_p$, der implizit definiert ist als endlicher Körper, bei dem zwei Polynome kleinen Grades mit kleinen Koeffizienten eine gemeinsame Nullstelle besitzen. Es gibt mehrere Methoden, mit denen man solche Polynome mit einer gemeinsamen Nullstelle modulo p konstruieren kann. Insbesondere mit Primzahlen einer speziellen Form, *d.h.* mit einer dünn besetzten Präsentation, ist es möglich, Polynome zu konstruieren, die viel besser sind als im allgemeinen Fall. Eine typische Konstruktion, die gut funktioniert, ist eine Zahl m zu wählen und p mit Basis m als $\sum_{i=0}^{t} a_i m^i$ zu schreiben. Dann haben $f_1(X) = X - m$ und $f_2(X) = \sum_{i=0}^{t} a_i m^i$ den Wert m als gemeinsame Nullstelle modulo p.

Ausgerüstet mit zwei Polynomen f_1 und f_2 von dieser Form, mit m als gemeinsamer Nullstelle modulo p, erhalten wir folgendes kommutatives Diagramm:



Seien r_1, r_2 die Nullstellen von f_1, f_2. Wir betrachten die Zahlkörper $\mathbb{Q}[r_1] \simeq \mathbb{Q}[X]/(f_1(X))$ und $\mathbb{Q}[r_2] \simeq \mathbb{Q}[X]/(f_2(X))$.

Faktorbasis: Besteht aus primen Elementen mit kleiner Norm aus beiden Zahlkörpern.

Relationen finden: Das grundlegende Prinzip des Zahlkörpersiebs besteht darin, Elemente der Form $a+bX$ an beide Seiten des Diagramms zu senden und eine Relation zu schreiben,

[1] Beim Zahlkörpersieb zur Berechnung des Dlog gibt es – im Gegensatz zum Zahlkörpersieb zur Faktorisierung in Abschnitt 10.4.1 – nur **das** Zahlkörpersieb. Die Unterscheidung Special versus General fällt hier weg.

wenn sich beide Seiten in die Faktorbasis faktorisieren lassen. Technisch ist das ziemlich herausfordernd, weil wir mehrere Werkzeuge einführen müssen, um zu erklären, dass die linken und rechten Seiten nicht notwendigerweise *Faktorielle Ringe* (unique factorization domains) sind. Als Konsequenz müssen wir die Elemente in Ideale faktorisieren und uns um die Hindernisse kümmern, die aus den Idealklassengruppen und Einheitsklassen resultieren. Diese Prozedur gibt uns den diskreten Logarithmus der Faktorbasiselemente.

Dlog-Berechnung: Drücke den gesuchten Logarithmus als Linearkombination der Faktorbasiselemente aus.

Laufzeit: Das Zahlkörpersieb ist der derzeit effizienteste bekannte Algorithmus für das diskrete Logarithmus-Problem mit großer Charakteristik. Im allgemeinen Fall – d.h. p hat keine spezielle Form, z.B. nah an einer Primzahlpotenz – ist seine Komplexität $L_p[\frac{1}{3}, \left(\frac{64}{9}\right)^{1/3}]$.

Referenzen und weiterführende Literatur: Für eine Einleitung zu Index-Calculus und die damit verbundenen mathematischen Werkzeuge siehe Mays Vorlesungsskript zur Zahlentheorie [May13] und das Buch zur Zahlentheorie von Müller-Stach, Piontkowski [MSP11]. Um ein tieferes Verständnis des Zahlkörpersiebs zu erlangen, sollte man das Buch von Lenstra und Lenstra [LL93] studieren, das alle Originalarbeiten enthält, die zur Entwicklung des Zahlkörpersieb-Algorithmus in den späten 80ern und frühen 90ern geführt haben.

Als guten Start, um das Zahlkörpersieb zu verstehen, empfehlen wir, zunächst seine Vorgänger zu studieren, die in den Originalarbeiten von Adleman [Adl79], Coppersmith [COS86] und Pomerance [Pom84, Pom96] beschrieben sind.

10.3 Beste bekannte Algorithmen für Erweiterungskörper $\mathbb{F}_{p^n}$ und aktuelle Fortschritte

Management Summary: Die Gruppen über Erweiterungskörpern werden von neuen Algorithmen von Joux et al. angegriffen. Vor der Erfindung dieser Angriffe schien die Sicherheit von Körpererweiterungsgruppen ähnlich der Sicherheit der Gruppen primer Ordnung aus dem letzten Kapitel zu sein. Die neuen Angriffe ließen diese Gruppen völlig unsicher werden, beeinflussten allerdings nicht die Sicherheit der Gruppen primer Ordnung.

Als erstes werden wir den ehemals besten Algorithmus von 2006 von Joux und Lercier besprechen, der eine Laufzeit von $L_n[\frac{1}{3}]$ erreicht. Anschließend beschreiben wir aktuelle Entwicklungen, die zu einer dramatischen Verbesserung der Laufzeit zu $L_n[o(1)]$ geführt haben, was quasi polynomiell ist.

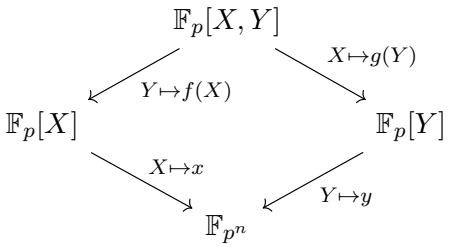
10.3.1 Der Joux-Lercier Function-Field-Sieve (FFS)

Jeder endliche Körper $\mathbb{F}_{p^n}$ kann repräsentiert werden durch einen Polynomring $\mathbb{F}_p[x]/f(x)$, wobei $f(x)$ ein irreduzibles Polynom über $\mathbb{F}_p$ vom Grad n ist. Damit kann jedes Element in $\mathbb{F}_{p^n}$ durch ein univariates Polynom mit Koeffizienten in $\mathbb{F}_p$ und einem Grad kleiner als n repräsentiert werden. Addition zweier Elemente ist die übliche Addition von Polynomen, wobei die Koeffizienten modulo p genommen werden. Die Multiplikation zweier Elemente ist die übliche Multiplikation von Polynomen, wobei das Ergebnis modulo $f(x)$ reduziert wird, um erneut ein Polynom mit einem Grad kleiner als n zu erhalten.

Es ist wichtig zu bemerken, dass die **Beschreibungslänge** eines Elementes $n\mathcal{O}(\log p)$ ist. Damit erreicht ein polynomieller Algorithmus eine Laufzeit, die polynomiell in n und $\log p$ ist. Wir werden außerdem Körper mit kleiner Charakteristik p in Betracht ziehen, wobei p konstant ist. Dann bedeutet polynomielle Laufzeit polynomiell in n.

Es ist bekannt, dass es für jedes p immer Polynome $f(x)$ mit Grad n gibt, die irreduzibel über $\mathbb{F}_p$ sind. Üblicherweise gibt es viele solcher Polynome, was umgekehrt bedeutet, dass wir für verschiedene Polynome $f(x)$ verschiedene Repräsentationen eines endlichen Körpers erhalten. Es ist jedoch ebenfalls bekannt, dass all diese Repräsentationen isomorph sind, und dass Isomorphismen effizient berechenbar sind.

Diese Tatsache wird im Algorithmus von Joux und Lercier verwendet, die verschiedene Repräsentationen $\mathbb{F}_p[x]/f(x)$ und $\mathbb{F}_p[y]/g(y)$ desselben Körpers ausnutzen. Dies wird veranschaulicht durch das folgende kommutative Diagramm.



Faktorbasis: Wir wählen alle Grad-1 Polynome $x-a$ und $y-b$ aus $\mathbb{F}_p[x]\cup\mathbb{F}_p[y]$. Damit besitzt die Faktorbasis $2p$ Elemente.

Relationen finden: Auf beiden Seiten, also für Polynome h aus $\mathbb{F}_p[x]/f(x)$ und aus $\mathbb{F}_p[y]/g(y)$, versuchen wir in Linearfaktoren aus der Faktorbasis zu faktorisieren. Das kann für jedes Polynom durch eine einfache ggT-Berechnung $\gcd(h, x^p - x)$ in Zeit $\mathcal{O}(p)$ gemacht werden. Man kann zeigen, dass die Anzahl der Polynome, die getestet werden müssen, begrenzt ist durch $L_{p^n}[\frac{1}{3}]$.

Dlog-Berechnung: Dieser Schritt wird durchgeführt, indem ein Polynom als Linearkombination von Polynomen kleineren Grades geschrieben und dies rekursiv wiederholt wird, bis Grad-1 gefunden ist. Diese Rekursion wird (Grad-)Abstieg (degree decent) genannt und erfordert ebenso wie der „relation finding"-Schritt eine Laufzeit von $L_{p^n}[\frac{1}{3}]$.

10.3.2 Kürzliche Verbesserungen für den Function Field Sieve

Die erste kürzliche Verbesserung für den Joux-Lercier-FFS wurde präsentiert bei der Eurocrypt 2013 von Joux, der zeigte, dass es möglich ist, die Komplexität für das Finden der Relationen drastisch zu reduzieren, indem man den klassischen siebenden Ansatz durch eine neue Technik ersetzt, die auf der linearen Änderung von Variablen basiert und *pinpointing* genannt wird.

Auf der Crypto Conference 2013 präsentierten Gölöglu, Granger, McGuire und Zumbrägel einen weiteren Ansatz, der mit dem pinpointing verwandt ist und sehr effizient mit Unterkörpern mit Charakteristik 2 arbeitet. Ihr Paper wurde von der kryptographischen Community als so wichtig eingestuft, dass sie den Preis für das beste Paper erhielten.

Die neuen Ergebnisse gelten für endliche Körper $\mathbb{F}_{q^n}$ mit Charakteristik zwei, d.h. $q = 2^\ell$. Bemerke, dass wir die Standardkonvention verwenden, die Primzahlen mit p und Primzahlpotenzen mit $q = p^\ell$ bezeichnet. Für diese Körper $\mathbb{F}_{q^n}$ wird der „relation finding"-Schritt im Joux-Lercier-Algorithmus einfacher, da man Polynome konstruieren kann, die sich mit einer höheren Wahrscheinlichkeit teilen lassen als allgemeine Polynome desselben Grades.

Lassen Sie uns eine high-level Beschreibung der Ideen zu ihrer Verbesserungen geben.

Faktorbasis: Alle Grad-1 Polynome wie im Joux-Lercier-Algorithmus.

Relationen finden: Gölöglu, Granger, McGuire und Zumbrägel zeigen, dass man einen speziellen Typ von Polynomen über $\mathbb{F}_q[x]$ konstruieren kann – die sogenannten Bluher-Polynome – die sich per Konstruktion über $\mathbb{F}_q[x]$ teilen lassen. Somit erhalten wir ähnlich zu unserer simplen Version des Index-Calculus für ganze Zahlen in Abschnitt 10.2.1 kostenlos eine Seite der Gleichung. Die Kosten für das Teilen der Polynome in $\mathbb{F}_q[y]$ sind ungefähr $\mathcal{O}(q)$, und die Kosten für das Finden des diskreten Logarithmus der Faktorbasiselemente sind ungefähr $\mathcal{O}(n \cdot q^2)$. Wir werden weiter unten erklären, warum uns das – für geeignet gewählte Parameter – die diskreten Logarithmen der Faktorbasis in *polynomieller Zeit* verschafft.

Dlog-Berechnung: Die Berechnung des individuellen diskreten Logarithmus ist ähnlich wie im Joux-Lercier-Algorithmus.

Laufzeit: Wir rechnen in einem Körper $\mathbb{F}_{q^n}$, mit $q = 2^\ell$. Somit würde ein Polynomialzeit-Algorithmus eine Laufzeit erfordern, die polynomiell in den Parametern n und $\log q$ ist. Das „relation finding" oben benötigt jedoch eine Zeit von $\mathcal{O}(n \cdot q^2)$, was polynomiell in n ist, aber exponentiell in $\log q$. Damit arbeitet der Algorithmus aber nur unzureichend, wenn man den Basiskörper $\mathbb{F}_q = \mathbb{F}_{2^\ell}$ in Betracht zieht.

Der Trick, um das zu umgehen, ist die Größe der Basis q auf q' zu reduzieren, während man den Erweiterungsgrad n etwas auf n' erhöht. Unser Ziel dabei ist, dass die neue Basiskörpergröße q' ungefähr dem neuen Erweiterungsgrad n' entspricht, also $q' \approx n'$. In diesem Fall erhalten wir erneut eine Laufzeit, die polynomiell in n' und q' ist, aber jetzt ist q' ebenfalls polynomiell beschränkt durch n'. Insgesamt ist unsere Laufzeit für Schritt 2 damit polynomiell beschränkt durch n'.

Lassen Sie uns ein einfaches Beispiel angeben, wie das für konkrete Parameter gehandhabt wird. Angenommen wir wollen einen diskreten Logarithmus in $\mathbb{F}_{(2^{100})^{100}}$ berechnen. Dann würden wir den Basiskörper zu $q' = 2^{10}$ verringern und gleichzeitig den Erweiterungsgrad zu $n' = 1000$ erhöhen, d.h. wir rechnen in $\mathbb{F}_{(2^{10})^{1000}}$. Bemerke, dass dies immer gemacht werden kann durch Nutzung effizient berechenbarer Isomorphismen zwischen endlichen Körpern gleicher Kardinalität.

Warnung: Man könnte versucht sein, das Obige mit der Wahl von Exponenten zu umgehen, die sich nicht geeignet teilen lassen, d.h. durch Wahl von $\mathbb{F}_{2^p}$ mit p prim. Man kann jedoch immer den endlichen Körper in einen größeren Körper einbetten – ebenso wie die entsprechenden diskreten Logarithmen. Deshalb werden endliche Körper mit kleiner Charakteristik als unsicher angesehen, unabhängig von der speziellen Form des Erweiterungsgrades n.

Während das „relation finding“ in Schritt 2 von Göloglu, Granger, McGuire und Zumbrägel in *polynomieller Zeit* erledigt werden kann, ist die Berechnung des individuellen Logarithmus immer noch zeitraubend. Macht man dies auf naive Art und Weise, ist Schritt 3 wegen des erhöhten Erweiterungsgrades n' sogar noch zeitintensiver als in Joux-Lercier. Balanciert man die Laufzeiten von Schritt 2 und Schritt 3 aus, erhält man eine verbesserte Gesamtlaufzeit von $L_{q^n}[\frac{1}{3}, (\frac{4}{9})^{\frac{1}{3}}]$.

10.3.3 Quasi-polynomielle Dlog-Berechnung von Joux et al

Im vorigen Abschnitt wurde gezeigt, dass der diskrete Logarithmus von allen Elementen einer Faktorbasis in polynomieller Zeit berechnet werden kann. Es blieb jedoch ein hartes Problem, diese Tatsache für die Berechnung individueller Logarithmen zu verwenden.

Dieses Problem wurde kürzlich gelöst von Joux [Jou13a] und Barbulesu, Gaudry, Joux und Thomé [BGJT13]. Im Paper von Joux wurde gezeigt, dass der individuelle Logarithmus-Schritt in $L[\frac{1}{4}]$ durchgeführt werden kann. Kurz danach wurde dies verbessert zu $L[o(1)]$ durch Barbulescu, Gaudry, Joux und Thomé, was eine Funktion ist, die langsamer wächst als $L[\epsilon]$ für jedes $\epsilon > 0$. Damit erreichen sie quasi-polynomielle Zeit.

Lasst uns kurz die Modifikationen dieser beiden Papers für den Function-Field-Sieve-Algorithmus beschreiben.

Faktorbasis: Besteht wie zuvor aus den Grad-1 Polynomen.

Relationen finden: Man startet mit dem trivialen initialen Polynom

$$h(x) = x^q - x = \prod_{\alpha \in \mathbb{F}_q} (x - \alpha)$$

das sich offensichtlich in die Faktorbasis faktorisieren lässt. Jetzt wendet man lineare und rationale Transformationen (Homographien genannt) auf $h(x)$ an, die seine Eigenschaft,

sich über der Faktorbasis faktorisieren zu lassen, erhalten. Man kann zeigen, dass es genügend viele Homographien gibt, um ausreichend viele Relationen zu konstruieren. Somit können wir aus einem trivialen Polynom $h(x)$ kostenfrei alle $\mathcal{O}(q)$ Relationen erhalten. Das befähigt uns dazu, die diskreten Logarithmen der Faktorbasiselemente in Zeit $\mathcal{O}(q)$ zu berechnen.

Dlog-Berechnung: Barbulescu et al präsentieren einen effizienten *Gradabstiegs*-Algorithmus, der bei Eingabe eines Polynoms $p(x)$ von Grad n eine lineare Relation zwischen dem diskreten Logarithmus von $p(x)$ und $\mathcal{O}(nq^2)$ Polynomen von Grad $\frac{n}{2}$ ausgibt, in einer Zeit, die polynomiell in q und D ist. Das bedeutet, dass wir einen Baum von Polynomen bekommen, bei dem der Grad mit jedem Level um den Faktor zwei fällt, was umgekehrt eine Baumtiefe von $\log n$ impliziert. Das resultiert in einer Laufzeit von $\mathcal{O}(q^{\mathcal{O}(\log n)})$.

Laufzeit: Wie im vorigen Abschnitt 10.3.2 nehmen wir an, dass die Größe q des Basiskörpers die gleiche Größe hat wie der Erweiterungsgrad n, d.h. $q = \mathcal{O}(n)$. Dann läuft Schritt 2 in Zeit $\mathcal{O}(q) = \mathcal{O}(n)$, was polynomiell in n ist. Schritt 3 läuft in Zeit $\mathcal{O}(q^{\mathcal{O}(\log n)}) = \mathcal{O}(n^{\mathcal{O}(\log n)}) = L_{q^n}[o(1)]$. Bemerke, dass $n^{\log n} = 2^{\log^2 n}$ schneller wächst als jede polynomielle Funktion in n, aber langsamer als jede sub-exponentielle Funktion 2^{n^c} für ein $c > 0$.

10.3.4 Schlussfolgerungen für endliche Körper mit kleiner Charakteristik

Um einige Beispiele zu geben, was die theoretische quasi-polynomielle Laufzeit der vorigen Ergebnisse in der Praxis bedeutet, veranschaulichen wir in Tabelle 10.1, was derzeit bei der Berechnung diskreter Logarithmen erreicht werden kann.

Datum	Körper	Bitgröße	Kosten (CPU-Stunden)	Algorithmus
2012/06/17	$3^{6\cdot 97}$	923	895 000	[JL06]
2012/12/24	p^{47}	1175	32 000	[Jou13b]
2013/01/06	p^{57}	1425	32 000	[Jou13b]
2013/02/11	2^{1778}	1778	220	[Jou13a]
2013/02/19	2^{1778}	1991	2200	[GGMZ13]
2013/03/22	2^{4080}	4080	14 100	[Jou13a]
2013/04/11	2^{6120}	6120	750	[Jou13a]
2013/05/21	2^{6168}	6168	550	[Jou13a]

Tabelle 10.1: Rekorde für kleine Charakteristik

Empfehlung: Der Gebrauch von Körpern mit kleiner Charakteristik für Dlog-basierte Verfahren auf Basis diskreter Logarithmen ist **gänzlich unsicher**, unabhängig davon welche Schlüsselgröße verwendet wird. Glücklicherweise wird davon – nach unserem Wissen – in weit verbreiteten/standardisierten kryptographischen Verfahren kein Gebrauch gemacht.

10.3.5 Lassen sich diese Ergebnisse auf andere Index-Calculus-Algorithmen übertragen?

Aus der Sicht eines Kryptoanwenders würde man sich sorgen, dass sich die aktuellen bahnbrechenden Ergebnisse, die die Komplexität für die Berechnung diskreter Logarithmen in Körpern mit kleiner Charakteristik von $L[\frac{1}{3}]$ auf $L[o(1)]$ senken, auch auf diskrete Logarithmen in anderen Gruppen anwenden lassen. Man könnte zum Beispiel besorgt sein über das tatsächliche

Sicherheitslevel von auf diskreten Logarithmen basierender Kryptographie in endlichen Körpern $\mathbb{F}_p$ mit *großer* Charakteristik.

Vermutung: Wir glauben, dass sich die neuen Techniken nicht auf endliche Körper mit großer Charakteristik oder auf Elliptische Kurven übertragen lassen, die gegenwärtig den Standard für kryptographische Konstruktionen darstellen.

Lasst uns kurz einige Gründe sammeln, warum sich die aktuellen Techniken nicht auf diese Gruppen übertragen lassen, und welche Probleme gelöst werden müssen, bevor wir einen signifikanten Fortschritt in der Laufzeit für diese Gruppen sehen.

- **Laufzeit:** Man bemerke, dass alle in diesem Abschnitt beschriebenen Index-Calculus-Algorithmen polynomiell in der Größe q des Basiskörpers sind und somit exponentiell in der Bitlänge $\mathcal{O}(\log q)$. Damit scheint sich die Härte des diskreten Logarithmus-Problems von der Härte im Basiskörper abzuleiten, wobei der Erweiterungsgrad n nicht dazu beiträgt das Problem erheblich schwieriger zu machen.

  Insbesondere merken wir an, dass jede – wie in den Algorithmen für kleine Charakteristik aus dem Polynom $x^q - x$ konstruierte – Gleichung mindestens q Terme enthält. Damit würde, sobald q größer als $L[1/3]$ wird, sogar das Schreiben einer einzigen Gleichung dieses Typs mehr kosten als die volle Komplexität des Zahlkörpersiebs aus Abschnitt 10.2.2.

  Bemerke, dass es eine ähnliche Situation für diskrete Logarithmen in Gruppen Elliptischer Kurven gibt. Nutzen wir eine Elliptische Kurve über $\mathbb{F}_q$, ist im Allgemeinen der beste bekannte Algorithmus der generische Pollard-Rho-Algorithmus aus Kapitel 10.1 mit der Laufzeit $\mathcal{O}(\sqrt{q})$. Allerdings benötigt Gaudry's Algorithmus – den wir in Abschnitt 10.5.2 besprachen – für Elliptische Kurven über $\mathbb{F}_{q^n}$ nur eine Laufzeit von $q^{2-\frac{2}{n}}$, was wesentlich besser ist als die generische Grenze $\mathcal{O}(q^{\frac{n}{2}})$. Wie die Algorithmen in diesem Kapitel ist auch Gaudry's Algorithmus ein Index-Calculus-Algorithmus. Und ähnlich wie bei den Algorithmen in diesem Kapitel scheint die Komplexität des diskreten Logarithmus-Problems im Parameter q statt im Parameter n konzentriert zu sein.

- **Polynome versus Zahlen:** Bemerke, dass die aktuellen Ergebnisse starken Gebrauch von polynomieller Arithmetik und Unterkörpern von $\mathbb{F}_{q^n}$ machen. Allerdings ist weder die polynomielle Arithmetik verfügbar für $\mathbb{F}_p$ noch existieren Unterkörper für Gruppen primer Ordnung. Wir möchten argumentieren, dass viele Probleme für Polynome effizient lösbar sind, während sie bekanntermaßen hart für ganze Zahlen zu sein scheinen. Es ist zum Beispiel bekannt, dass Polynome über endliche Körper und über rationale Zahlen von den Algorithmen von Berlekamp und Lenstra-Lenstra-Lovasz effizient faktorisiert werden können, während es keinen äquivalenten Algorithmus für ganze Zahlen gibt. Nach von zur Gathen gibt es auch einen effizienten Algorithmus um kürzeste Vektoren in Polynomringen zu finden, während das Gegenstück in Ganzahlgittern (integer lattice) NP-hart ist.

  Was ganze Zahlen eigentlich härter macht als Polynome ist der Effekt der Übertragsbits. Multiplizieren wir zwei Polynome, dann wissen wir durch das Konvolutionsprodukt genau, welche Koeffizienten bei welchen Koeffizienten des Produktes mitwirken, was aber bei der Multiplikation ganzer Zahlen wegen der Übertragsbits nicht der Fall ist.

- **Komplexität der Schritte 2 & 3:** Jeglicher algorithmische Durchbruch für diskrete Logarithmen vom Typ Index-Calculus müsste die diskreten Logarithmen einer wohldefinierten Faktorbasis effizient lösen *und* den gewünschten Logarithmus in Termen aus dieser

Faktorbasis ausdrücken. Zur Zeit haben wir aber im Fall großer Primkörper $\mathbb{F}_p$ für keinen dieser Schritte eine effiziente Methode.

Referenzen und weiterführende Literatur: Coppersmiths Algorithmus [Cop84] aus der Mitte der 80er war lange Zeit die Referenzmethode für die Berechnung diskreter Logarithmen in Körpern mit kleiner Charakteristik. Der Joux-Lercier-Function-Field-Sieve wurde 2006 in [JL06] vorgestellt.

Die aktuellen Fortschritte begannen auf der Eurocrypt 2013 mit Jouxs Pinpointing-Technik [Jou13b]. Auf der Crypto 2013 verbesserten Gölogluu, Granger, McGuire und Jens Zumbrägel [GGMZ13] bereits die Konstante c in der $L[\frac{1}{3}, c]$-Laufzeit. Die Verbesserung zur Laufzeit $L[\frac{1}{4}]$ wurde dann vorgestellt in der Arbeit von Joux [Jou13a]. Letztendlich schlugen Barbulescu, Gaudry, Joux und Thomé [BGJT13] einen Algorithmus für den Abstieg vor, der zur Laufzeit $L[o(1)]$ führte.

10.4 Beste bekannte Algorithmen für die Faktorisierung natürlicher Zahlen

Management Summary: Der beste Algorithmus zur Faktorisierung zeigt starke Ähnlichkeit zum besten Algorithmus für die Berechnung diskreter Logarithmen in Gruppen primer Ordnung. Es scheint, dass die neuen Angriffe nicht dabei helfen, einen der beiden Algorithmen zu verbessern.

Der beste Algorithmus für die Berechnung der Primfaktorzerlegung einer ganzen Zahl, das sogenannte Zahlkörpersieb, ist dem besten Algorithmus für die Berechnung diskreter Logarithmen in $\mathbb{F}_p$ aus Abschnitt 10.2.2 sehr ähnlich. Sehr viel weniger ähnelt er dem Algorithmus für $\mathbb{F}_{q^n}$ aus Kapitel 10.3.

Kurz gesagt beruhen alle bekannten, komplexen Algorithmen, die RSA-Module $n = pq$ für p, q prim faktorisieren, auf derselben simplen, grundlegenden Idee. Unser Ziel ist es, $x, y \in \mathbb{Z}/n\mathbb{Z}$ zu konstruieren so dass

$$x^2 \equiv y^2 \bmod n \text{ und } x \not\equiv \pm y \bmod n.$$

Dies liefert sofort die Faktorisierung von n, da n wegen der ersten Eigenschaft das Produkt $x^2 - y^2 = (x+y)(x-y)$ teilt, aber wegen der zweiten Eigenschaft teilt n weder $x+y$ noch $x-y$. Damit teilt ein Primfaktor von n den Term $x + y$, während der andere $x - y$ teilen muss. Das bedeutet umgekehrt, dass $\gcd(x \pm y, n) = \{p, q\}$.

Die Faktorisierungsalgorithmen unterscheiden sich nur in der Art, in der die x, y berechnet werden. Die Absicht ist, x, y mit $x^2 \equiv y^2 \bmod n$ in einer „unabhängigen“ Art und Weise zu berechnen. Falls diese Unabhängigkeit gegeben ist, ist es einfach zu zeigen, dass $x \not\equiv \pm y \bmod n$ mit Wahrscheinlichkeit $\frac{1}{2}$ gilt, da nach dem Chinesischen Restsatz jedes Quadrat in $\mathbb{Z}/n\mathbb{Z}$ 4 Quadratwurzel besitzt – zwei verschiedene Wurzeln modulo p und zwei verschiedene Wurzeln modulo q.

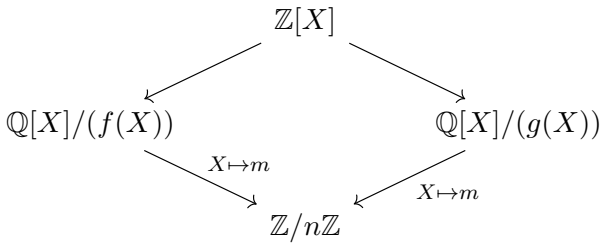
10.4.1 Das Zahlkörpersieb zur Faktorisierung (GNFS)[2]

Sei $n \in \mathbb{N}$ die ganze Zahl, die wir faktorisieren wollen. Der Zahlkörpersieb-Algorithmus startet damit, zwei Polynome f, g zu konstruieren, die eine gemeinsame Nullstelle m modulo N teilen. Üblicherweise wird das gemacht, indem man $g(X) = X - m \bmod n$ definiert und ein Polynom $f(X)$ kleinen Grades konstruiert mit $f(m) \equiv 0 \bmod n$ (z.B. durch Erweitern von n zur Basis m wie in Abschnitt 10.2.2).

Da f und g verschieden sind, definieren sie verschiedene Ringe $\mathbb{Z}[X]/f(X)$ und $\mathbb{Z}[X]/g(X)$. Da aber f und g die gleiche Nullstelle m modulo n teilen, sind beide Ringe isomorph zu $\mathbb{Z}/n\mathbb{Z}$; und dieser Isomorphismus kann explizit berechnet werden durch Mappen von $X \mapsto m$. Dies

[2]Mit Zahlkörpersieb (Number Field Sieve) ist hier immer das **Allgemeine** Zahlkörpersieb (GNFS) gemeint. Im Gegensatz zu Abschnitt 10.2.2 unterscheidet man bei der Faktorisierung zwischen einem Special und General Number Field Sieve.
CT2 enthält eine Implementierung des GNFS mittels msieve und YAFU.

wird im folgenden kommutativen Diagramm illustriert.



Faktor base: Besteht aus primen Elementen mit kleiner Norm aus beiden Zahlkörpern.

Relationen finden: Wir suchen nach Argumenten $\tilde{x}$, so dass sich gleichzeitig $\pi_f := f(\tilde{x})$ in $\mathbb{Q}[X]/(f(X))$ und $\pi_g := g(\tilde{x})$ in $\mathbb{Q}[X]/(g(X))$ in die Faktorbasiselemente teilen lassen. Solche Elemente werden Relationen genannt.

Lineare Algebra: Mit Hilfe Linearer Algebra suchen wir ein Produkt der Elemente π_f, das ein Quadrat ist und dessen korrespondierendes Produkt der π_g ebenfalls ein Quadrat ist. Bilden wir diese Elemente mit unserem Homomorphismus $X \mapsto m$ auf $\mathbb{Z}/n\mathbb{Z}$ ab, so erhalten wir Elemente $x^2, y^2 \in \mathbb{Z}/n\mathbb{Z}$, so dass $x^2 \equiv y^2 \bmod n$. Berechnen wir erst die Quadratwurzeln von π_f und π_g in deren entsprechenden Zahlkörpern, ohne vorher den Homomorphismus anzuwenden, so erhalten wir wie gewünscht $x, y \in \mathbb{Z}/n\mathbb{Z}$ mit $x^2 \equiv y^2 \bmod N$. Die Unabhängigkeit von x, y leitet sich ab aus den verschiedenen Repräsentationen in beiden Zahlkörpern.

Laufzeit: Der obige Algorithmus ist bis auf manche Details – z.B. die Quadratwurzelberechnung im Zahlkörper – identisch zum Algorithmus aus Abschnitt 10.2.2 und besitzt die gleiche Laufzeit $L[\frac{1}{3}, \left(\frac{64}{9}\right)^{1/3}]$.

10.4.2 Die Verbindung zum Index-Calculus-Algorithmus in $\mathbb{F}_p$

Erstens wissen wir, dass die Berechnung diskreter Logarithmen in Gruppen $\mathbb{Z}/n\mathbb{Z}$ mit zusammengesetzter Ordnung mindestens so hart ist wie das Faktorisieren von $n = pq$. Das bedeutet umgekehrt, dass jeder Algorithmus, der diskrete Logarithmen in $\mathbb{Z}/n\mathbb{Z}$ berechnet, im Prinzip die Faktorisierung von n berechnet:

$$\text{Dlogs in } \mathbb{Z}/n\mathbb{Z} \Rightarrow \text{Faktorisierung von } n.$$

Lasst uns kurz die Idee dieser Faktorisierung beschreiben. Wir berechnen die Ordnung $k = \text{ord}(a)$ für ein beliebiges $a \in \mathbb{Z}/n\mathbb{Z}$ durch unseren Dlog-Algorithmus, d.h. wir berechnen die kleinste positive Ganzzahl k, s.d. $a^k \equiv 1 \bmod n$. Ist k gerade, dann ist $a^{\frac{k}{2}} \not\equiv 1$ eine Quadratwurzel von 1. Wir haben $a^{\frac{k}{2}} \not\equiv -1$ mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$, da die 1 genau 4 Quadratwurzeln modulo n besitzt. Setze $x \equiv a^{\frac{k}{2}} \bmod n$ und $y = 1$. Dann erhalten wir $x^2 \equiv 1 \equiv y^2 \bmod n$ und $x \not\equiv \pm y \bmod n$. Laut der Diskussion am Beginn des Kapitels erlaubt uns das, n zu faktorisieren.

Zweitens wissen wir außerdem, dass beide Probleme, das Faktorisieren und das Berechnen diskreter Logarithmen in $\mathbb{F}_p$, zusammen mindestens so hart sind wie das Berechnen diskreter Logarithmen in $\mathbb{Z}/n\mathbb{Z}$. Kurz gesagt:

$$\text{Faktorisierung} + \text{Dlogs in } \mathbb{F}_p \Rightarrow \text{Dlogs in } \mathbb{Z}/n\mathbb{Z}.$$

Diese Tatsache kann einfach gesehen werden, indem man bemerkt, dass Faktorisierung und Dlogs in $\mathbb{F}_p$ zusammen direkt eine effiziente Version des Silver-Pohlig-Hellman-Algorithmus aus Abschnitt 10.1 geben. Erst faktorisieren wir die Gruppenordnung n in die Primzahlpotenzen $p_i^{e_i}$ und berechnen dann den diskreten Logarithmus in $\mathbb{F}_{p_i}$ für jedes i. Genau wie im Silver-Pohlig-Hellman-Algorithmus heben wir die Lösung modulo $p_i^{e_i}$ und kombinieren diese gehobenen Lösungen mittels Chinesischem Restsatz.

Wir möchten betonen, dass diese zwei bekannten Relationen nicht viel darüber aussagen, ob es eine Reduktion

$$\text{Faktorisierung} \Rightarrow \text{Dlog in } \mathbb{F}_p \qquad \text{oder} \qquad \text{Dlog in } \mathbb{F}_p \Rightarrow \text{Faktorisierung}.$$

gibt. Beide Richtungen sind ein lang bekanntes offenes Problem in der Kryptographie. Merke jedoch, dass die besten Algorithmen für Faktorisierung und Dlog in $\mathbb{F}_p$ aus den Abschnitten 10.2.2 und 10.4.1 bemerkenswert ähnlich sind. Außerdem bedeutete historisch ein Fortschritt bei einem Problem immer auch Fortschritt beim anderen. Obwohl wir keinen formellen Beweis haben, dürfte es fair sein zu sagen, dass beide Probleme aus algorithmischer Sicht eng verknüpft zu sein scheinen.

10.4.3 Integer-Faktorisierung in der Praxis

Gegeben den aktuellen Stand der Technik der akademischen Forschung über Integer-Faktorisierung stellen selbst RSA-Module moderater – aber sorgfältig gewählter – Größe einen angemessenen Grad an Sicherheit gegen offene kryptoanalytische Anstrengungen der Community dar. Die größte RSA-Challenge, die durch öffentliche Anstrengungen faktorisiert wurde, hatte lediglich 768 Bits [KAF+10] und erforderte ein Äquivalent von etwa 2000 Jahren Berechnung auf einem einzelnen 2 GHz-Kern. Ein Angriff auf 1024-Bit RSA-Module ist etwa tausendmal härter. Ein solcher Aufwand sollte für akademische Anstrengungen für mehrere weitere Jahre außer Reichweite sein. Eine Verdopplung der Größe zu 2048-Bit Modulen erhöht den rechnerischen Aufwand um einen weiteren Faktor von 10^9. Ohne substantielle neue, mathematische oder algorithmische Erkenntnisse muss 2048-Bit RSA für mindestens zwei weitere Jahrzehnte als außer Reichweite betrachtet werden.

10.4.4 Die Relation von Schlüsselgröße versus Sicherheit für Dlog in $\mathbb{F}_p$ und Faktorisierung

Die Laufzeit des besten Algorithmus für ein Problem definiert den Sicherheitslevel eines Kryptosystems. Z.B. brauchen wir für 80-Bit Sicherheit, dass der beste Algorithmus mindestens 2^{80} Schritte benötigt.

Wie wir bereits anmerkten, ist die beste Laufzeit für diskrete Logarithmen in $\mathbb{F}_p$ und für Faktorisierung $L[\frac{1}{3}, \left(\frac{64}{9}\right)^{1/3}]$. Der akkurateste Weg, diese Formel zu nutzen, ist in der Tat, die Laufzeit für eine große reale Faktorisierung/Dlog-Berechnung zu messen und dann große Werte zu extrapolieren. Angenommen wir wissen, dass es Zeit T brauchte, eine Zahl n_1 zu faktorisieren. Dann extrapolieren wir die Laufzeit für ein $n_2 > n_1$ mit der Formel

$$T \cdot \frac{L_{n_1}[\frac{1}{3}, \left(\frac{64}{9}\right)^{1/3}]}{L_{n_2}[\frac{1}{3}, \left(\frac{64}{9}\right)^{1/3}]}.$$

Bitgröße	Sicherheit
768	67.0
896	72.4
1024	77.3
1152	81.8
1280	86.1
1408	90.1
1536	93.9
1664	97.5
1792	100.9
1920	104.2
2048	107.4

Tabelle 10.2: Bitgröße von n, p versus Sicherheitslevel

Somit nutzen wir die L-Formel, um den relativen Faktor abzuschätzen, den wir zusätzlich aufwenden müssen. Merke, dass dies die Sicherheit (geringfügig) überschätzt, da die L-Formel asymptotisch ist und somit im Zähler akkurater ist als im Nenner – der Nenner sollte einen größeren Fehler-Term beinhalten. Somit erhält man in der Praxis eine (nur geringfügig) kleinere Sicherheit als von der Formel vorhergesagt.

Wir berechneten die Formel für mehrere Bitgrößen einer RSA-Zahl n, beziehungsweise eine Dlog Primzahl p, in Tabelle 10.2. Man erinnere sich von Abschnitt 10.4.1, dass die Laufzeit des Number-Field-Sieve-Algorithmus für Faktorisierung tatsächlich eine Funktion in n und nicht in den Primfaktoren von n ist.

Wir beginnen mit RSA-768, das 2009 erfolgreich faktorisiert wurde [KAF+10]. Um die Anzahl der Instruktionen für die Faktorisierung von RSA-768 zu zählen, muss man definieren, was eine *Instruktionseinheit* (instruction unit) ist. In der Kryptographie ist es ein bewährtes Verfahren, die Zeit für die Berechnung von DES als Maßeinheit zu definieren, um eine Vergleichbarkeit von Sicherheitsleveln zwischen Secret- und Public-Key-Primitiven zu erhalten. Dann bietet DES nach Definition dieser Maßeinheit 56-Bit Sicherheit gegen Brute-force-Schlüsselangriffe.

In Bezug auf diese Maßeinheit benötigt die Faktorisierung von RSA-768 $T = 2^{67}$ Instruktionen. Von diesem Startpunkt aus extrapolierten wir das Sicherheitslevel für größere Bitgrößen in Tabelle 10.2.

Wir erhöhten nacheinander die Bitgröße um 128 bis zu 2048 Bits. Wir sehen, dass dies zu Beginn zu einem Anstieg der Sicherheit um etwa 5 Bits pro 128-Bit Schritt führt, während wir gegen Ende nur einen Anstieg von etwa 3 Bits pro 128-Bit Schritt haben.

Nach Moores Gesetz verdoppelt sich die Geschwindigkeit von Computern alle 1,5 Jahre. Damit haben wir nach $5 \cdot 1,5 = 7,5$ Jahren einen Anstieg von 2^5, was bedeutet, dass wir derzeit alle $7,5$ Jahre unsere Bitgröße um etwa 128 Bits erhöhen sollten; und wenn wir uns den 2000 Bits nähern, sollten die Intervalle unserer Erhöhung in 128-Bit Schritten nicht länger sein als 4,5 Jahre. Für vorsichtigere Wahlen, die außerdem einen gewissen algorithmischen Fortschritt voraussetzt statt nur einen Anstieg in der Geschwindigkeit von Computern, siehe die Empfehlungen in Kapitel 10.7.

Referenzen und weiterführende Literatur: Eine Einleitung zu mehreren Faktorisierungsalgorithmen inklusive des Quadratic Sieve – dem Vorgänger des Zahlkörpersiebs – findet sich in Mays Skript zur Zahlentheorie [May13]. Wir empfehlen Blömers Skript zur Algorithmischen

Zahlentheorie [Blö99] als Einleitung zum Zahlkörpersieb.

Die Entwicklung des Zahlkörpersiebs wird beschrieben im Lehrbuch von Lenstra und Lenstra [LL93], das alle Originalarbeiten beinhaltet. Die Relation von diskreten Logarithmen und Faktorisierung wurde diskutiert von Bach [Bac84]. Details zum aktuellen Faktorisierungsrekord für RSA-768 kann man in [KAF+10] finden.

10.5 Beste bekannte Algorithmen für Elliptische Kurven E

Management Summary: Elliptische Kurven sind die zweite Standardgruppe für das diskrete Logarithmus-Problem. Die neuen Angriffe betreffen diese Gruppen nicht, und ihre Sicherheit bleibt unverändert.

Wir möchten Elliptische Kurven $E[p^n]$ über endlichen Erweiterungskörpern $\mathbb{F}_{p^n}$ und elliptische Kurven $E[p]$ über Primkörpern $\mathbb{F}_p$ diskutieren. Die letzteren werden üblicherweise für kryptographische Zwecke verwendet. Der Grund, aus dem wir auch die ersteren diskutieren, ist – ähnlich wie in den vorigen Kapiteln – dass wir auch die Schwächen von Erweiterungskörpern $\mathbb{F}_{p^n}$ gegenüber Primkörpern $\mathbb{F}_p$ illustrieren wollen. Wir möchten jedoch darauf hinweisen, dass wir im Folgenden – im Gegensatz zu den vorigen Kapiteln – annehmen, dass n fest ist. Das liegt daran, dass anders als im Algorithmus von Joux et al die Algorithmen für $E[p^n]$ Komplexitäten besitzen, die exponentiell von n abhängen.

Wir präsentieren zwei verschiedene Ansätze für Elliptische Kurven über Erweiterungskörpern: zum einen die von Gaundry, Hess und Smart (GHS) vorgestellten Cover- (oder Weil-Descent-)Angriffe, und zum zweiten die von Semaev and Gaudry vorgeschlagenen Dekompositionsangriffe. In manchen Fällen ist es möglich, die beiden Ansätze zu einem noch effizienteren Algorithmus zu kombinieren, wie von Joux und Vitse gezeigt wurde [JV11].

10.5.1 Der GHS-Ansatz für Elliptische Kurven $E[p^n]$

Dieser von Gaudry, Hess und Smart vorgestellte Ansatz zielt darauf ab, das diskrete Logarithmus-Problem von einer über einem Erweiterungskörper $\mathbb{F}_{p^n}$ definierten Elliptischen Kurve E zu einer über einem kleineren Körper, z.B. $\mathbb{F}_p$, definierten Kurve mit höherem Geschlecht zu transportieren. Dies kann erreicht werden durch das Finden einer Kurve H über $\mathbb{F}_p$ zusammen mit einem surjektiven Morphismus von H nach E. In diesem Kontext sagen wir, H ist eine Überdeckung von E. Sobald wir eine solche Kurve H gefunden haben, ist es möglich, die sogenannte coNorm-Technik anzuwenden, um das diskrete Logarithmus-Problem auf E auf ein diskretes Logarithmus-Problem auf der Jacobischen von H zurückzuführen. Falls das Geschlecht g der Zielkurve nicht zu groß ist, kann dies zu einem effizienten Algorithmus für diskrete Logarithmen führen. Dies verwendet die Tatsache, dass es einen Index-Calculus-Algorithmus für Kurven mit hohem Geschlecht g über $\mathbb{F}_p$ gibt mit Komplexität $\max(g!\,p, p^2)$. Das wurde vorgestellt von Enge, Gaudry und Thomé [EGT11].

Idealerweise hätte man gern, dass das Geschlecht g gleich ist zu n. Das ist im Allgemeinen jedoch nicht möglich. Die möglichen Überdeckungen für Elliptische Kurven zu klassifizieren scheint eine schwierige Aufgabe zu sein.

10.5.2 Gaudry-Semaev-Algorithmus für Elliptische Kurven $E[p^n]$

Sei $Q = \alpha P$ ein diskreter Logarithmus auf einer Elliptischen Kurve $E[p^n]$. Das Ziel ist es, eine ganze Zahl $\alpha \in \mathbb{N}$ zu finden, so dass k-maliges Addieren des Punktes $P \in E[p^n]$ zu sich selbst gleich ist zu dem Punkt $Q \in E[p^n]$.

Gaudry's Diskreter-Logarithmus-Algorithmus ist vom Typ Index-Calculus. Wir umreißen kurz die grundlegenden Schritte.

Faktorbasis: Besteht aus allen Punkten (x, y) auf der Elliptischen Kurve $E[p^n]$ mit $x \in \mathbb{F}_p$. Somit liegt x im Basiskörper $\mathbb{F}_p$ statt in der Erweiterung.

Relationen finden: Gegeben einen zufälligen Punkt $R = aP$ mit $a \in \mathbb{N}$, versuchen wir R als Summe von exakt n Punkten der Faktorbasis zu schreiben, wobei n der Erweiterungsgrad ist. Dies wird erzielt durch Nutzen des n-ten Semaevpolynoms f_{n+1}. Dieses Polynom ist ein symmetrisches Polynom von Grad 2^{n-2} in $n+1$ Unbekannten $x_1, \ldots, x_{n+1}$, welche die Tatsache kodieren, dass es Punkte mit entsprechenden Abszissen $x_1, \ldots, x_{n+1}$ gibt, die zu Null summieren. Die Koeffizienten von f hängen natürlich von der Kurve E ab. Das Ersetzen von x_{n+1} durch die Abszisse von R ermöglicht das Finden einer Dekomposition von R als Summe von Punkten aus der Faktorbasis, indem man eine Lösung $(x_1, \cdots, x_n)$ im Basiskörper $\mathbb{F}_p$ sucht. Um das zu tun, schreibt man f in ein multivariates System aus n Gleichungen um, indem man die Konstanten, die im Polynom auftauchen, über einer Basis von $\mathbb{F}_{p^n}$ über $\mathbb{F}_p$ zerlegt. Dieses System von n Gleichungen in n Unbekannten kann mit Hilfe der Berechnung einer Gröbnerbasis gelöst werden.

Individuelle Dlog-Berechnung: Um den diskreten Logarithmus von Q zu berechnen, genügt es, eine zusätzliche Relation zu finden, die ein zufälliges Multiplikatives von Q darstellt, sprich $R = aQ$ in Bezug auf die Punkte der Faktorbasis. Dies wird erreicht in genau derselben Weise wie die Erzeugung von Relationen im vorigen Schritt.

Laufzeit: Die Faktorbasis kann in Zeit $\mathcal{O}(p)$ berechnet werden. Jedes R kann geschrieben werden als Summe von n Faktorbasiselementen, d.h. es liefert eine Relation mit einer Wahrscheinlichkeit, die exponentiell klein ist in n (aber unabhängig von p). Falls es eine Lösung liefert, ist die Laufzeit für die Berechnung einer Gröbnerbasis ebenfalls exponentiell in n (aber polynomiell in $\log p$). Insgesamt benötigen wir ungefähr p Relationen, die berechnet werden können in einer Zeit, die linear in p und exponentiell in n ist. Da wir annehmen, dass n fest ist, müssen wir uns nicht um das schlechte Verhalten in n kümmern. Der Schritt mit Linearer Algebra auf einer $(p \times p)$-Matrix kann in $\mathcal{O}(p^2)$ durchgeführt werden, da die Matrix dünn besetzt ist – jede Zeile enthält genau n Einträge, die nicht Null sind. Mit Hilfe zusätzlicher Tricks erzielt man eine Laufzeit von $\mathcal{O}(p^{2-\frac{2}{n}})$ für Gaudry's Algorithmus.

Dies sollte verglichen werden mit der generischen Schranke von $\mathcal{O}(p^{\frac{n}{2}})$, die wir erreichen, wenn wir den Pollard-Rho-Algorithmus aus Kapitel 10.1 verwenden. Ähnlich wie in Kapitel 10.3 scheint sich fast die ganze Komplexität des Problems in der Größe des Basiskörpers p zu konzentrieren, und nicht im Erweiterungsgrad n. Bemerke, dass in Kapitel 10.3 Gaudry's Algorithmus exponentiell ist in $\log p$.

10.5.3 Beste bekannte Algorithmen für Elliptische Kurven $E[p]$ über Primkörpern

Generisches Lösen diskreter Logarithmen: Allgemein ist der beste uns bekannte Algorithmus für beliebige Elliptische Kurven $E[p]$ die Pollard-Rho-Methode mit einer Laufzeit von $\mathcal{O}(\sqrt{p})$. Für den Moment scheint niemand zu wissen, wie man die Struktur einer Gruppe Elliptischer Kurven oder seiner Elemente ausnutzt, um die generische Schranke zu verbessern.

Wir möchten außerdem betonen, dass *zufällige* Elliptische Kurven, d.h. deren Parameter a, b aus der definierenden Weierstrassgleichung $y^2 \equiv x^3 + ax + b \bmod p$ zufällig gleichverteilt gewählt werden, zu den harten Instanzen gehören. Um Elliptische Kurven noch härter zu machen, wählt man für die Standardisierung nur solche Kurven, die (nahezu) Primordnung haben. Das bedeutet, dass der Co-Faktor der größten Primzahl in der Gruppenordnung üblicherweise 1 ist, so dass die Nutzung des Silver-Pohlig-Hellman-Algorithmus nichts nützt.

Einbetten von $E[p]$ in $\mathbb{F}_{p^k}$: Es ist bekannt, dass im Allgemeinen Elliptische Kurven $E[p]$ in einen endlichen Körper $\mathbb{F}_{p^k}$ eingebettet werden können, wobei k der sogenannte *Grad der*

Einbettung ist. In $\mathbb{F}_{p^k}$ könnten wir das Zahlkörpersieb für die Berechnung diskreter Logarithmen verwenden. Damit wäre solch eine Einbettung attraktiv, wenn $L_{p^k}[\frac{1}{3}]$ kleiner ist als $\sqrt{p}$, was nur der Fall ist, wenn der Grad der Einbettung k sehr klein ist. Allerdings ist für fast alle Elliptischen Kurven der Grad der Einbettung bekannterweise groß, nämlich vergleichbar zu p selbst.

Manche Konstruktionen in der Kryptographie, z.B. solche, die bilineare Paarungen (bilinear pairings) verwenden, nutzen die Vorteile eines kleinen Einbettungsgrades aus. Damit werden in diesen Verfahren Elliptische Kurven explizit mit kleinem Einbettungsgrad gewählt, z.B. $k = 6$, was die Härte des diskreten Logarithmus-Problems auf $E[p]$ und in $\mathbb{F}_p^k$ ausbalanciert.

Der Xedni-Calculus-Algorithmus: In 2000 veröffentlichte Silverman seinen *Xedni-Calculus-Algorithmus* (man lese Xedni rückwärts), der die Gruppenstruktur von $E[p]$ für die Berechnung diskreter Logarithmen verwendet, und der somit der einzige bekannte nicht-generische Algorithmus ist, der direkt auf $E[p]$ arbeitet. Allerdings wurde kurz nach seiner Veröffentlichung entdeckt, dass der sogenannte Hebungsprozess in Silvermans Algorithmus nur mit vernachlässigbarer Wahrscheinlichkeit erfolgreich einen diskreten Logarithmus berechnet.

10.5.4 Die Relation von Schlüsselgröße versus Sicherheit für Elliptische Kurven $E[p]$

Ähnlich wie in der Diskussion in Abschnitt 10.4.4 über Schlüsselgrößen für Dlog in $\mathbb{F}_p$ und für Faktorisierung, möchten wir evaluieren, wie die Schlüsselgrößen für Elliptische Kurven $E[p]$ angepasst werden müssen, um vor einem Anstieg der Computergeschwindigkeit zu schützen. Für Elliptische Kurven ist eine solche Analyse vergleichsweise simpel. Der beste Algorithmus für Dlog in $E[p]$, den wir kennen, ist die Pollard-Rho-Methode mit der Laufzeit

$$L_p[1, \frac{1}{2}] = \sqrt{p} = 2^{\frac{\log p}{2}}.$$

Das bedeutet, dass wir für ein Sicherheitslevel von k Bits eine Primzahl p mit $2k$ Bits wählen müssen. Mit anderen Worten bewirkt das Erhöhen der Bitgröße unserer Gruppe um 2 Bits eine Erhöhung der Sicherheit um 1 Bit. Nach Moores Gesetz verlieren wir alle 1,5 Jahre 1 Bit an Sicherheit nur durch den Anstieg der Computergeschwindigkeit. Um diesem Verlust über 10 Jahre vorzubeugen, sollte es somit ausreichen, die Gruppengröße um $10/1,5 \cdot 2 = 7 \cdot 2 = 14$ Bits zu erhöhen. Bemerke, dass dieser Anstieg im Gegensatz zum Fall von Dlog in $\mathbb{F}_p$ und der Faktorisierung in Abschnitt 10.4.4 linear ist und unabhängig vom Startpunkt. Das bedeutet, dass ein Anstieg von 28 Bit ausreicht, um der technologischen Beschleunigung über 20 Jahre vorzubeugen.

Natürlich gilt diese Analyse nur, wenn wir keinen großen Durchbruch in der Computertechnologie oder den Algorithmen erleben. Für eine konservativere Wahl siehe den Hinweis in Kapitel 10.7.

10.5.5 Wie man sichere Parameter für Elliptische Kurven wählt

Eine umfangreiche Beschreibung, wie man die Domain-Parameter für Elliptische Kurven über endlichen Körpern wählt, kann man in dem RFC 5639 "ECC Brainpool Standard Curves and Curve Generation" von Manfred Lochter und Johannes Merkle [LM10, LM05] finden. Dieser RFC definiert einen öffentlich verifizierbaren Weg, pseudozufällige Parameter für die Parameter Elliptischer Kurven zu wählen, und schließt damit eine Hauptquelle für das Einfügen einer Trapdoor in die Gruppendefinition aus. Die Autoren besprechen alle *bekannten* Eigenschaften einer Kurve $E[p]$, die ihre Sicherheit potenziell schwächen könnten:

- **Ein kleiner Einbettungsgrad** für die Einbettung in einen endlichen Körper. Dies würde die Nutzung effizienterer Algorithmen für endliche Körper erlauben. Insbesondere schließt diese Voraussetzung supersinguläre Kurven der Ordnung $p + 1$ aus.
- **Kurven mit Spur 1** mit der Ordnung $|E[p]| = p$. Diese Kurven sind bekannterweise schwach durch die Algorithmen für diskrete Logarithmen von Satoh-Araki [SA98], Semaev [Sem98] und Smart [Sma99].
- **Eine große Klassenzahl**. Dies schließt aus, dass $E[p]$ effizient zu einer über einen algebraischen Zahlkörper definierten Kurve gehoben werden kann. Diese Voraussetzung ist recht konservativ, da derzeit sogar für kleine Klassenzahlen kein effizienter Angriff bekannt ist.

Außerdem bestehen die Autoren auf folgenden nützlichen Eigenschaften.

- **Primzahlordnung**. Dies schließt einfach Untergruppenangriffe aus.
- **Verifizierbare Pseudozufallszahlengeneration**. Die Seeds für einen Pseudozufallsgenerator werden in einer systematischen Art nach Lochter und Merkle gewählt, die in ihrer Konstruktion die ersten 7 Substrings mit Länge 160 Bit der fundamentalen Konstante $\pi = 3,141\ldots$ verwenden.

Zusätzlich spezifizieren Lochter und Merkle mehrere Kurven für p's mit Bitlängen zwischen 160 und 512. Für TLS/SSL gibt es außerdem ein neues Set von vorgeschlagenen Brainpool-Kurven [LM13].

Die Arbeit von Bos, Costello, Longa und Naehrig [BCLN14] gibt eine wertvolle Einleitung für Anwender dazu, wie man Parameter für Elliptische Kurven wählt, die sicher sind und eine effiziente Implementierung in mehreren Koordinatensystemen (Weierstrass, Edwards, Montgomery) erlauben. Zusätzlich konzentrieren sich Bos et al auf Seitenkanalabwehr gegen Timing-Angriffe durch das Vorschlagen skalarer Multiplikationen in konstanter Zeit.

Wir empfehlen sehr das SafeCurve-Projekt von Daniel Bernstein und Tanja Lange [BL14], das einen exzellenten Überblick für verschiedene Auswahlmethoden und deren Vor- und Nachteile zur Verfügung stellt. Das Ziel von Bernstein und Lange ist es, Sicherheit für die Kryptographie mit Elliptischen Kurven zu liefern – und nicht nur Stärke von Elliptischen Kurven gegenüber Angriffen auf diskrete Logarithmen. Deshalb berücksichtigen sie verschiedene Typen von Seitenkanälen, die Geheimnisse in einer Implementierung durchsickern lassen könnten.

Referenzen und weiterführende Literatur: Für eine Einleitung zur Mathematik Elliptischer Kurven und deren kryptographische Anwendungen beziehen wir uns auf die Lehrbücher von Washington [Was08], Galbraith [Gal12] und Silverman [Sil99].

In diesem Abschnitt wurden die Ergebnisse der Originalarbeiten von Gaudry, Hess und Smart [GHS02], Gaudry [Gau09], Semaev [Sem04] und der Xedni-Algorithmus von Silverman [Sil99] beschrieben.

10.6 Die Möglichkeit des Einbettens von Falltüren in kryptographische Schlüssel

Management Summary: Alle Kryptographie scheint die Möglichkeit zu bieten, Falltüren einzubetten. Dlog-Verfahren haben einen gewissen Vorteil gegenüber faktorisierungs-basierten Verfahren in dem Sinne, dass sorgfältig gewählte systemweite Parameter *alle* Nutzer beschützen.

Die Möglichkeit des Einbettens von Falltüren in kryptographische Verfahren und damit des Entschlüsselns/Signierens/Authentifizierens ohne die Nutzung eines geheimen Schlüssels ist ein lange bekanntes Problem, das intensiv in der kryptographischen Gemeinschaft diskutiert wurde – z.B. bei der Podiumsdiskussion der Eurocrypt 1990. Allerdings hat die weitgestreute Nutzung von Falltüren der NSA, die von Edward Snowden aufgedeckt wurde, das Interesse an diesem Thema erneuert.

Es scheint, dass manche Verfahren per Konstruktion wesentlich anfälliger sind als andere. Für auf diskreten Logarithmen basierten Verfahren ist z.B. die Definition der Gruppenparameter ein systemweiter Parameter, der von jedem Nutzer im System verwendet wird. Damit kann ein Beteiligter, der in der Lage ist, die Definition einer Gruppe so zu manipulieren, dass er diskrete Logarithmen in dieser Gruppe effizient berechnen kann, *jegliche* Kommunikation entschlüsseln. Auf der anderen Seite bietet eine sorgfältig definierte sichere Gruppe Sicherheit für *alle* Anwender.

Derzeit gibt es einige Spekulationen darüber, ob die NSA das amerikanische Institut für Standards und Technologie NIST dahingehend beeinflusst hat, bestimmte Elliptische Kurven zu standardisieren. Aber die Definition einer Gruppe ist nicht der einzige Weg, Falltüren einzubetten. Alle bekannten kryptographischen Verfahren hängen grundsätzlich von einer guten Quelle (pseudo-)zufälliger Bits ab. Es ist bekannt, dass die sogenannte semantische Sicherheit von Verschlüsselungsverfahren nicht ohne Zufälligkeit erreicht werden kann, und angenommen wird, dass jeder kryptographische, geheime Schlüssel zufällig gewählt wird. Damit öffnet ein schwacher Pseudozufallsgenerator die Tür, um Kryptographie zu umgehen. Solch ein schwacher Pseudozufallsgenerator wurde vom NIST standardisiert als Special Publication 800-90, obwohl es Warnungen von der kryptographischen Gemeinschaft gab.

Für auf Faktorisierung basierende Verfahren ist die Situation etwas anders als bei solchen, die auf diskreten Logarithmen basieren. Im Gegensatz zu auf diskreten Logarithmen basierenden Verfahren gibt es keine systemweiten Parameter, die eine Gruppe definieren. Dennoch gibt es bekannte Wege, um z.B. Informationen über die Faktorisierung des RSA-Moduls N in den öffentlichen RSA-Exponenten e einzubetten. Außerdem zeigen neueste Angriffe auf die Infrastrukturen der öffentlichen RSA-Schlüssel [LHA+12, HDWH12], dass es ein schwieriges Problem zu sein scheint, öffentliche RSA-Schlüssel mit verschiedenen Primzahlen in der Öffentlichkeit zu generieren, hauptsächlich wegen schlechter Initialisierungen von Pseudozufallsgeneratoren. Dies betrifft natürlich nur schlecht gewählte Schlüssel von einzelnen anstatt von allen Anwendern eines kryptographischen Verfahrens.

Empfehlung: Dlog-basierte Verfahren scheinen aus der Sicht eines Krypto-Designers einfacher zu kontrollieren zu sein, da hier alle Anwender die gleichen systemweiten Parameter nehmen müssen.

Wir diskutieren an dieser Stelle nicht die Möglichkeit von Malware – welche jeglichen kryptographischen Versuch des Schutzes hinfällig machen könnte – oder wie man sich dagegen schützt. Aber wir möchten folgende (einigermaßen triviale) Warnung betonen, die sich auf einen bedeutenden Punkt in der Praxis bezieht.

Warnung: Kryptographie kann Daten nur beschützen, wenn sie korrekt implementiert ist und ihr immanentes Geheimnis nicht preisgibt. Somit müssen wir zusätzlich zur mathematischen Härte des zugrunde liegenden Problems auch dem Implementierer des kryptographischen Verfahrens vertrauen. Dieses Vertrauen beinhaltet nicht nur, dass das Verfahren so implementiert ist wie es ursprünglich designed wurde – ohne das Einbetten einer Falltür – sondern auch, dass der Implementierer einer dritten Partei nicht die generierten geheimen Schlüssel offenlegt.

Es scheint, dass in der NSA-Affäre manche Firmen gezwungen wurden, geheime Schlüssel zu offenbaren. Somit sollte man bedenken, dass man kryptographische Verfahren von einer völlig verlässlichen Firma kaufen muss, die nicht kompromittiert wurde.

Referenzen und weiterführende Literatur: Für eine nette Diskussion, wie man unaufspürbare Falltüren in verschiedene kryptographische Verfahren einbettet, siehe die Originalarbeiten von Young und Yung [YY96, YY97]. Siehe [LHA+12] für einen aktuellen Angriff auf eine signifikant große Menge von RSA-Schlüsseln in der Praxis infolge schlechter Pseudozufallsgeneration.

10.7 Vorschlag für die kryptographische Infrastruktur

Management Summary: Trotz der aktuellen Dlog-Angriffe bleiben auf diskreten Logarithmen basierende Verfahren über *Gruppen primer Ordnung* und über *Gruppen über Elliptischen Kurven* weiterhin sicher. Das Gleiche gilt für auf Faktorisierung basierende Verfahren. Alle Dlog-basierten Gruppen mit kleiner Charakteristik sind komplett unsicher. Unsere Empfehlung ist die Wahl von Gruppen über Elliptischen Kurven.

10.7.1 Empfehlung für die Wahl des Verfahrens

Wie wir in den vorangegangenen Kapiteln sahen, bleiben Dlog-basierte Verfahren über $\mathbb{F}_p$ und über $E[p]$ sicher, ebenso wie faktorisierungsbasierte Verfahren. Im Folgenden empfehlen wir Schlüsselgrößen für diese Verfahren, die ein ausreichendes Sicherheitslevel **für die nächsten zwei Jahrzehnte** liefern unter der Voraussetzung, dass kein bedeutender algorithmischer Durchbruch erzielt wird.

System	Schlüsselgröße in Bits
Dlog in $\mathbb{F}_p$	2000 bis 2019, danach 3000
Factorisierung	2000 bis 2019, danach 3000
Dlog in $E[p]$	224 bis 2015, danach 250

Tabelle 10.3: Sicherheitslevel 100 Bit, Quelle: BSI [BSI12], ANSSI [Age13]

Unsere Präferenz ist, **Gruppen über Elliptischen Kurven** $E[p]$ zu verwenden, da sie die folgenden Vorteile bieten:

- Algorithmen für diskrete Logarithmen in $\mathbb{F}_p$ und Faktorisierung sind eng miteinander verbunden. Somit könnte jeglicher Fortschritt bei einem der beiden auch Fortschritt für das andere bedeuten. Es ist aber unwahrscheinlich, dass solch ein Fortschritt auch die Sicherheit von Gruppen über Elliptische Kurven beeinflusst.
- Die besten Algorithmen für $E[p]$ sind solche generischen Typs aus Kapitel 10.1, die den besten Algorithmen für diskrete Logarithmen mit Primordnung und Faktorisierung mit einer Laufzeit von $L[\frac{1}{3}]$ unterlegen sind. Dies bedeutet umgekehrt, dass das Schlüsselwachstum, welches den technologischen Fortschritt schnellerer Computer kompensiert, für $E[p]$ viel kleiner ist – ungefähr 2 Bits alle 1,5 Jahre laut Moores Gesetz.
- Algorithmischen Fortschritt durch die Nutzung der Gruppenstruktur von $E[p]$ zu erhalten, scheint härter zu sein als für $\mathbb{F}_p$, da wir im Gegensatz zu $\mathbb{F}_p$ nicht einmal einen initial beginnenden Index-Calculus-Algorithmus für die Gruppenstruktur haben, den wir verbessern könnten.
- Falls eine Elliptische Kurve $E[p]$ sorgfältig gewählt ist, d.h. dass die Gruppe rechnerisch hart und ohne Hintertür ist, dann profitieren alle Anwender von der Härte des diskreten Logarithmus-Problems. Man merke, dass diese Wahl entscheidend ist: Falls die Gruppe nicht sicher ist, dann leiden auch alle Anwender durch ihre Unsicherheit.

Warnung: Man sollte bedenken, dass obige Empfehlungen nur in einer Welt ohne große Quantencomputer gelten. Es scheint sehr wichtig zu sein, den aktuellen Fortschritt in diesem Gebiet

zu verfolgen und innerhalb der nächsten 15 Jahre einige alternative, Quantencomputer-resistente Kryptoverfahren bereit zu haben.

Referenzen und weiterführende Literatur: Für eine gute und konservative Wahl von Schlüsselgrößen empfehlen wir sehr, den Vorschlägen des Bundesamtes für Sicherheit in der Informationstechnik (BSI) [BSI12] und der Agence nationale de la sécurité des systèmes d'information [Age13] zu folgen. Beide Quellen bieten außerdem mehrere wertvolle Empfehlungen, wie man verschiedene kryptographische Primitiven korrekt implementiert und kombiniert.

Anmerkung des Editors im Juni 2016: Seit April 2014 haben sich viele Dinge geändert (es gab es neue Rekorde bei dlog-endlichen Körpern und kleinere Verbesserungen des L(1/3)-Algorithmus in einigen Kontexten). Trotzdem bleibt die Gesamtaussage bestehen, dass (nur) endliche Körper mit kleiner Charakteristik nicht mehr sicher sind.

Literaturverzeichnis (DLogFact)

[Adl79] Adleman, Leonard M.: *A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography (Abstract).* In: *FOCS*, Seiten 55–60, 1979.

[Age13] Agence nationale de la sécurité des systèmes d'information: *Référentiel général de sécurité Version 2.02*, 2013.
`http://www.ssi.gouv.fr/administration/reglementation/`.

[Bac84] Bach, Eric: *Discrete Logarithms and Factoring.* Technischer Bericht UCB/CSD-84-186, EECS Department, University of California, Berkeley, Juni 1984.
`http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5973.html`,
`http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-186.pdf`.

[BCLN14] Bos, Joppe W., Craig Costello, Patrick Longa und Michael Naehrig: *Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis*, 2014. Microsoft Research.
`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/selecting.pdf`.

[BGJT13] Barbulescu, Razvan, Pierrick Gaudry, Antoine Joux und Emmanuel Thomé: *A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic.* CoRR, abs/1306.4244, 2013.

[BL14] Bernstein, Daniel und Tanja Lange: *SafeCurves: choosing safe curves for elliptic-curve cryptography.* `http://safecurves.cr.yp.to/`, 2014.

[Blö99] Blömer, J.: *Vorlesungsskript Algorithmische Zahlentheorie*, 1999. Ruhr-University Bochum.
`http://www.math.uni-frankfurt.de/~dmst/teaching/lecture_notes/bloemer.algorithmische_zahlentheorie.ps.gz`.

[BSI12] BSI (Bundesamt für Sicherheit in der Informationstechnik): *BSI TR-02102 Kryptographische Verfahren: Empfehlungen und Schlüssellängen*, 2012.
`https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm`.

[Cop84] Coppersmith, Don: *Evaluating Logarithms in $GF(2^n)$.* In: *STOC*, Seiten 201–207, 1984.

[COS86] Coppersmith, Don, Andrew M. Odlyzko und Richard Schroeppel: *Discrete Logarithms in GF(p).* Algorithmica, 1(1):1–15, 1986.

[EGT11] Enge, Andreas, Pierrick Gaudry und Emmanuel Thomé: *An L(1/3) Discrete Logarithm Algorithm for Low Degree Curves.* J. Cryptology, 24(1):24–41, 2011.

[FJM14] Fouque, Pierre Alain, Antoine Joux und Chrysanthi Mavromati: *Multi-user collisions: Applications to Discrete Logarithm, Even-Mansour and Prince.* Cryptology ePrint Archive, 2014. `https://eprint.iacr.org/2013/761`.

[Gal12] Galbraith, Steven D.: *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012, ISBN 9781107013926.

[Gau09] Gaudry, Pierrick: *Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem.* J. Symb. Comput., 44(12):1690–1702, 2009.

[GGMZ13] Gölоglu, Faruk, Robert Granger, Gary McGuire und Jens Zumbrägel: *On the Function Field Sieve and the Impact of Higher Splitting Probabilities – Application to Discrete Logarithms.* In: *CRYPTO (2)*, Seiten 109–128, 2013.

[GHS02] Gaudry, Pierrick, Florian Hess und Nigel P. Smart: *Constructive and Destructive Facets of Weil Descent on Elliptic Curves.* J. Cryptology, 15(1):19–46, 2002.

[HDWH12] Heninger, Nadia, Zakir Durumeric, Eric Wustrow und J. Alex Halderman: *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.* In: *Proceedings of the 21st USENIX Security Symposium*, August 2012. `https://factorable.net/paper.html`.

[Hom07] Homeister, Matthias: *Quantum Computer Science: An Introduction.* Vieweg+Teubner Verlag, 2007, ISBN 9780521876582.

[JL06] Joux, Antoine und Reynald Lercier: *The Function Field Sieve in the Medium Prime Case.* In: *EUROCRYPT*, Seiten 254–270, 2006.

[Jou09] Joux, Antoine: *Algorithmic Cryptanalysis.* CRC Cryptography and Network Security Series. Chapman & Hall, 2009, ISBN 1420070029.

[Jou13a] Joux, Antoine: *A new index calculus algorithm with complexity L(1/4+o(1)) in very small characteristic.* IACR Cryptology ePrint Archive, 2013:95, 2013.

[Jou13b] Joux, Antoine: *Faster Index Calculus for the Medium Prime Case Application to 1175-bit and 1425-bit Finite Fields.* In: *EUROCRYPT*, Seiten 177–193, 2013.

[JV11] Joux, Antoine und Vanessa Vitse: *Cover and Decomposition Index Calculus on Elliptic Curves made practical. Application to a seemingly secure curve over* F_{p^6}. IACR Cryptology ePrint Archive, 2011:20, 2011.

[KAF+10] Kleinjung, Thorsten, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev und Paul Zimmermann: *Factorization of a 768-Bit RSA Modulus.* In: *CRYPTO*, Seiten 333–350, 2010.

[LHA+12] Lenstra, Arjen K., James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung und Christophe Wachter: *Public Keys.* In: *CRYPTO*, Seiten 626–642, 2012.

[LL93] Lenstra, A. K. und H. W. Jr. Lenstra: *The Development of the Number Field Sieve.* Lecture Notes in Mathematics. Springer Verlag, 1993, ISBN 0387570136.

[LM05] Lochter, Manfred und Johannes Merkle: *ECC Brainpool Standard Curves and Curve Generation v. 1.0*, 2005.
`www.ecc-brainpool.org/download/Domain-parameters.pdf`.

[LM10] Lochter, Manfred und Johannes Merkle: *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, 2010. RFC 5639.
`http://www.rfc-base.org/txt/rfc-5639.txt`.

[LM13] Lochter, Manfred und Johannes Merkle: *Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)*, 2013. RFC 7027.
`http://tools.ietf.org/search/rfc7027`.

[May08] May, Alexander: *Vorlesungsskript Kryptanalyse 1*, 2008. Ruhr-University Bochum.
`http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/pkk08/skript.pdf`.

[May12] May, Alexander: *Vorlesungsskript Kryptanalyse 2*, 2012. Ruhr-University Bochum.
`http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/12/ws1213/kryptanal12/kryptanalyse_2013.pdf`.

[May13] May, Alexander: *Vorlesungsskript Zahlentheorie*, 2013. Ruhr-University Bochum.
`http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/13/ss13/zahlenss13/zahlentheorie.pdf`.

[Mer08] Mermin, David N.: *Quantum Computing verstehen.* Cambridge University Press, 2008, ISBN 3834804363.

[MSP11] Müller-Stach und Piontkowski: *Elementare und Algebraische Zahlentheorie.* Vieweg Studium, 2011, ISBN 3834882631.

[MvOV01] Menezes, Alfred J., Paul C. van Oorschot und Scott A. Vanstone: *Handbook of Applied Cryptography.* Series on Discrete Mathematics and Its Application. CRC Press, 5. Auflage, 2001, ISBN 0-8493-8523-7. (Errata last update Jan 22, 2014).
`http://cacr.uwaterloo.ca/hac/`,
`http://www.cacr.math.uwaterloo.ca/hac/`.

[Pol75] Pollard, John M.: *A Monte Carlo method for factorization.* BIT Numerical Mathematics 15, 3:331–334, 1975.

[Pol00] Pollard, John M.: *Kangaroos, Monopoly and Discrete Logarithms.* J. Cryptology, 13(4):437–447, 2000.

[Pom84] Pomerance, Carl: *The Quadratic Sieve Factoring Algorithm.* In: Blakley, G.R. und D. Chaum (Herausgeber): *Proceedings of Crypto '84, LNCS 196*, Seiten 169–182. Springer, 1984.

[Pom96] Pomerance, Carl: *A tale of two sieves.* Notices Amer. Math. Soc, 43:1473–1485, 1996.

[PRSS13] Ptacek, Thomas, Tom Ritter, Javed Samuel und Alex Stamos: *The Factoring Dead – Preparing for the Cryptopocalypse.* Black Hat Conference, 2013.

[SA98] Satoh, T. und K. Araki: *Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves.* Commentarii Mathematici Universitatis Sancti Pauli 47, 1998.

[Sem98] Semaev, I.: *Evaluation of Discrete Logarithms on Some Elliptic Curves.* Mathematics of Computation 67, 1998.

[Sem04] Semaev, Igor: *Summation polynomials and the discrete logarithm problem on elliptic curves.* IACR Cryptology ePrint Archive, 2004:31, 2004.

[Sho94] Shor, Peter W.: *Algorithms for Quantum Computation: Discrete Logarithms and Factoring.* In: *FOCS*, Seiten 124–134, 1994.

[Sho97] Shoup, Victor: *Lower Bounds for Discrete Logarithms and Related Problems.* In: *EUROCRYPT*, Seiten 256–266, 1997.

[Sil99] Silverman, Joseph H.: *The Xedni Calculus And The Elliptic Curve Discrete Logarithm Problem.* Designs, Codes and Cryptography, 20:5–40, 1999.

[Sma99] Smart, N.: *The Discrete Logarithm Problem on Elliptic Curves of Trace One.* Journal of Cryptology 12, 1999.

[Was08] Washington, Lawrence C.: *Elliptic Curves: Number Theory and Cryptography.* Discrete Mathematics and its Applications. Chapman and Hall/CRC, 2008, ISBN 9781420071467.

[YY96] Young, Adam L. und Moti Yung: *The Dark Side of Black-Box Cryptography, or: Should We Trust Capstone?* In: *CRYPTO*, Seiten 89–103, 1996.

[YY97] Young, Adam L. und Moti Yung: *Kleptography: Using Cryptography against Cryptography.* In: *EUROCRYPT*, Seiten 62–74, 1997.

Alle Links wurden am 15.07.2016 überprüft.

Kapitel 11

Krypto 2020 — Perspektiven für langfristige kryptographische Sicherheit

(Johannes Buchmann, Erik Dahmen, Alexander May und Ulrich Vollmer, TU Darmstadt, Mai 2007)

Kryptographie ist ein Grundbaustein aller IT-Sicherheitslösungen. Aber wie lange sind die heute benutzten kryptographischen Verfahren noch sicher? Reicht diese Zeit, um zum Beispiel medizinische Daten lang genug geheim zu halten? Doch auch kurzfristig ließe sich großer Schaden anrichten, wenn auch nur bestimmte Schlüssel gebrochen würden: Etwa bei den digitalen Signaturen, welche die Authentizität von automatischen Windows-Updates sichern.

11.1 Verbreitete Verfahren

In ihrer berühmten Arbeit aus dem Jahr 1978 stellten Rivest, Shamir und Adleman [RSA78] das RSA Public-Key-Verschlüsselungs- und Signaturverfahren vor. RSA ist auch heute noch das in der Praxis meistverwendete Public-Key-System. Die Sicherheit von RSA beruht auf der Schwierigkeit, so genannte RSA-Moduln $N = pq$ in ihre (großen) Primfaktoren p und q zu zerlegen. In ihrer Arbeit schlugen die Erfinder von RSA damals vor, für langfristige Sicherheit 200-stellige RSA-Moduln zu verwenden. Später veröffentlichte die Firma RSA Security eine Liste von RSA-Moduln wachsender Größe (RSA Challenge numbers) und setzte Preise von insgesamt 635.000 US-$ für das Faktorisieren dieser Zahlen aus (siehe `http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-challenge-numbers.htm`).

Im Jahr 2005, also 27 Jahre nach der Erfindung von RSA, gelang es Bahr, Boehm, Franke und Kleinjung von der Universität Bonn bereits innerhalb von fünf Monaten eine 200-stellige RSA-Challenge zu faktorisieren und damit die ursprüngliche Empfehlung langfristiger Sicherheit zu brechen (`www.mat.uniroma2.it/~eal/rsa640.txt`). Dies belegt anschaulich die Fortschritte den letzten 30 Jahre bei der Faktorisierung von RSA-Moduln. Diese beruhen sowohl auf bahnbrechenden mathematischen Ideen, zum Beispiel der Entwicklung des Zahlkörpersiebs durch John Pollard, als auch auf bedeutenden Fortschritten in der Computer- und Implementierungstechnik.[1]

[1] Vergleiche Kapitel 4.11 Zur Sicherheit des RSA-Verfahrens, und speziell die Kapitel 4.11.4 und 4.11.5. Details zu

Lenstra und Verheul entwickelten 2000 eine Interpolationsformel zur Voraussage der Sicherheit von RSA und anderen wichtigen kryptographischen Verfahren (vgl. `www.keylength.com`). Dieser Formel zufolge muss man derzeit schon 850-stellige RSA-Moduln verwenden, um Sicherheit für die nächsten dreißig Jahre zu gewährleisten.

Aber auch eine solche Interpolationsformel ist keine Sicherheitsgarantie! Brillante mathematische Ideen könnten jederzeit dazu führen, dass das Lösen des Faktorisierungsproblems leicht und RSA damit generell unbrauchbar wird. So bewies beispielsweise Peter Shor 1996, dass ein Quantencomputer – ein neuer Computertyp, der die Gesetze der Quantenmechanik ausnutzt – große Zahlen im Prinzip sehr schnell faktorisieren könnte [Sho97]. Trotz intensiver Forschungsbemühungen ist es aber auch heute noch unklar, ob sich jemals hinreichend leistungsfähige Quantencomputer bauen lassen.[2] Aktuelle Verlautbarungen der Start-up-Firma D-Wave (`www.dwavesys.com`) trafen auf verbreitete Skepsis, ja Spott.

Analog zu RSA verläuft die Entwicklung bei Angriffen auf die meistverwendeten Public-Key-Alternativen: den Digital Signature Algorithm (DSA) und Elliptic Curve Cryptography (ECC), die beide auf der Schwierigkeit der Berechnung diskreter Logarithmen beruhen. Es gibt schon heute deutliche algorithmische Fortschritte und Quantencomputer würden auch diese Verfahren unsicher machen.

Wie steht es um die langfristige Sicherheit von so genannten Secret-Key-Verschlüsselungsverfahren? DES wurde 1977 als Data Encryption Standard eingeführt [DES77] – 21 Jahre später stellte die Electronic Frontier Foundation (EFF) den Spezialcomputer Deep Crack vor, der DES in 56 Stunden bricht. Das Problem von DES war die zu kurz gewählte Schlüssellänge: Offenbar hatten seine Erfinder die rasante Entwicklung bei der Hardware nicht richtig einkalkuliert. Der DES-Nachfolger Advanced Encryption Standard (AES) [AES02] gilt heute als sicher, wenngleich interessante Angriffsversuche mit algebraischen Methoden existieren.

11.2 Vorsorge für morgen

Ist die heutige Kryptographie angesichts ihrer wachsenden Bedeutung noch sicher genug? Die Erfahrung zeigt: Sorgfältig konzipierte und implementierte kryptographische Verfahren haben eine Lebensdauer von 5 bis 20 Jahren. Wer heute RSA, ECC oder AES zur kurzfristigen Absicherung verwendet, darf sich sicher fühlen. Und langfristige Verbindlichkeit lässt sich beispielsweise mit den von Jan Sönke Maseberg vorgeschlagenen multiplen Signaturen lösen [Mas02].

Aber langfristige Vertraulichkeit können wir heute mit den genannten Verfahren nicht garantieren. Und was ist in 20 Jahren? Was kann man tun, wenn ein unerwarteter mathematischer Fortschritt ein wichtiges Kryptoverfahren plötzlich – quasi über Nacht – unsicher macht? Drei Dinge sind zur Vorbereitung nötig:

---

aktuelleren kryptoanalytischen Ergebnissen gegen RSA und Dlog finden Sie in Kapitel 10.

[2] Benötigte qbits für Angriffe auf RSA, DSA und ECDSA für Schlüssel der Bit-Länge n:

RSA	2n + 3
DSA	2n + 3
ECDSA 2^n	˜2n + 8 log n
ECDSA p	˜4n

Vergleiche Kap. 5.3 in „SicAri – Eine Sicherheitsplattform und deren Werkzeuge für die ubiquitäre Internetnutzung, KB2.1 – Abschlussbericht, Übersicht über Angriffe auf relevante kryptographische Verfahren“, Version 1.0, 17. Mai 2005, Prof. Dr. Johannes Buchmann et al., TUD-KryptC und cv cryptovision GmbH (`http://www.cdc.informatik.tu-darmstadt.de/~schepers/kb_21_angriffe.pdf`) und die Dissertation von Axel Schmidt am gleichen Lehrstuhl.

- ein Pool alternativer sicherer Kryptoverfahren,
- Infrastrukturen, die es ermöglichen, unsichere kryptographische Verfahren leicht gegen sichere auszutauschen und
- Verfahren, die langfristige Vertraulichkeit sicherstellen.

Diese Ziele verfolgen auch die Kryptoarbeitsgruppe der TU Darmstadt und der daraus entstandene Spin-off FlexSecure (`www.flexsecure.de`) seit vielen Jahren. Die Trustcenter-Software FlexiTrust, die in der deutschen nationalen Root-Zertifizierungsstelle und in der deutschen Country-Signing-Zertifizierungsstelle verwendet wird, ist eine Infrastruktur, die den einfachen Austausch von Kryptographie-Verfahren möglich macht. Die Open-Source-Bibliothek FlexiProvider (siehe `www.flexiprovider.de`) stellt zudem eine Vielzahl unterschiedlicher kryptographischer Verfahren zur Verfügung und in jüngerer Zeit laufen intensive Forschungen zur „Post Quantum Cryptography“ (PQC): Kryptographie, die auch dann noch sicher bleibt, wenn es (leistungsfähige) Quantencomputer tatsächlich gibt.

Die Sicherheit der Public-Key-Kryptographie beruht traditionell auf der Schwierigkeit, bestimmte mathematische Probleme zu lösen. Heute diskutierte Alternativen zum Faktorisierungs- und Diskrete-Logarithmen- Problem sind: das Dekodierungsproblem, das Problem, kürzeste und nächste Gittervektoren zu berechnen, und das Problem, quadratische Gleichungen mit vielen Variablen zu lösen. Es wird vermutet, dass diese Probleme auch von Quantencomputern nicht effizient lösbar wären.

11.3 Neue mathematische Probleme

Wie sehen diese Alternativen näher aus? Verschlüsselung auf der Basis des Dekodierungsproblems wurde von McEliece erfunden [McE78]. Der Hintergrund: Fehlerkorrigierende Codes dienen dazu, digitale Informationen so zu speichern, dass sie selbst dann noch vollständig lesbar bleiben, wenn einzelne Bits auf dem Speichermedium verändert werden. Diese Eigenschaft nutzen zum Beispiel CDs, sodass Informationen auf leicht verkratzten Datenträgern immer noch vollständig rekonstruierbar sind.

Bei der codebasierten Verschlüsselung werden Daten verschlüsselt, indem zu ihrer Kodierung mit einem öffentlich bekannten Code gezielt Fehler addiert werden, das heißt einzelne Bits werden verändert. Die Entschlüsselung erfordert die Kenntnis einer geeigneten Dekodierungsmethode, welche diese Fehler effizient zu entfernen vermag. Diese Methode ist der geheime Schlüssel – nur wer sie kennt, kann dechiffrieren. Codebasierte Public-Key-Verschlüsselung ist im Allgemeinen sehr effizient durchführbar. Derzeit wird daran geforscht, welche Codes zu sicheren Verschlüsselungsverfahren mit möglichst kleinen Schlüsseln führen.

Verschlüsselung auf der Grundlage von Gitterproblemen ist den codebasierten Verschlüsselungsverfahren sehr ähnlich. Gitter sind regelmäßige Strukturen von Punkten im Raum. Zum Beispiel bilden die Eckpunkte der Quadrate auf kariertem Papier ein zweidimensionales Gitter. In der Kryptographie verwendet man jedoch Gitter in viel höheren Dimensionen. Verschlüsselt wird nach dem folgenden Prinzip: Aus dem Klartext wird zunächst ein Gitterpunkt konstruiert und anschließend geringfügig verschoben, sodass er kein Gitterpunkt mehr ist, aber in der Nähe eines solchen liegt. Wer ein Geheimnis über das Gitter kennt, kann diesen Gitterpunkt in der Nähe finden und damit entschlüsseln. Ein besonders praktikables Gitterverschlüsselungsverfahren ist NTRU (`https://www.securityinnovation.com/products/ntru-crypto`). Da NTRU vor vergleichsweise kurzer Zeit eingeführt wurde (1998) und seine Spezifizierung aufgrund ver-

schiedener Angriffe mehrfach geändert wurde, sind allerdings noch weitere kryptanalytische Untersuchungen erforderlich, um Vertrauen in dieses Verfahren zu gewinnen.

11.4 Neue Signaturen

1979 schlug Ralph Merkle einen bemerkenswerten Ansatz für die Konstruktion von Signaturverfahren in seiner Dissertation [Mer79] vor. Im Gegensatz zu allen anderen Signaturverfahren beruht seine Sicherheit nicht darauf, dass ein zahlentheoretisches, algebraisches oder geometrisches Problem schwer lösbar ist. Benötigt wird ausschließlich, was andere Signaturverfahren ebenfalls voraussetzen: eine sichere kryptographische Hash-Funktion und ein sicherer Pseudozufallszahlengenerator. Jede neue Hash-Funktion führt somit zu einem neuen Signaturalgorithmus, wodurch das Merkle-Verfahren das Potential hat, das Problem der langfristigen Verfügbarkeit digitaler Signaturverfahren zu lösen.

Merkle verwendet in seiner Konstruktion so genannte Einmal-Signaturen: Dabei benötigt jede neue Signatur einen neuen Signierschlüssel und einen neuen Verifikationsschlüssel. Die Idee von Merkle ist es, die Gültigkeit vieler Verifikationsschlüssel mittels eines Hash-Baums auf die Gültigkeit eines einzelnen öffentlichen Schlüssels zurückzuführen. Im Merkle-Verfahren muss man bei der Schlüsselerzeugung eine Maximalzahl möglicher Signaturen festlegen, was lange wie ein gravierender Nachteil aussah. In [BCD+06] wurde jedoch eine Variante des Merkle-Verfahrens vorgestellt, die es ermöglicht, äußerst effizient mit einem einzigen Schlüsselpaar bis zu 2^{40} Signaturen zu erzeugen und zu verifizieren.[3]

11.5 Quantenkryptographie – Ein Ausweg?

Ungelöst bleibt aus Sicht der heutigen Kryptographie das Problem der langfristigen Vertraulichkeit: Ein praktikables Verfahren, das die Vertraulichkeit einer verschlüsselten Nachricht über einen sehr langen Zeitraum sicherstellt, ist derzeit nicht bekannt.

Einen Ausweg kann hier möglicherweise die Quantenkryptographie liefern: Sie ermöglicht Verfahren zum Schlüsselaustausch (sehr lange Schlüssel für One-time-Pads), deren Sicherheit auf der Gültigkeit der Gesetze der Quantenmechanik beruhen, vgl. z.B. [BB85]. Jedoch sind bekannte Verfahren der Quantenkryptographie derzeit noch sehr ineffizient und es ist unklar, welche kryptographischen Funktionen damit realisiert werden können.

11.6 Fazit

Wie lautet die Bilanz? Die heutige Kryptographie liefert gute Werkzeuge, um kurzfristige und mittelfristige Sicherheit zu gewährleisten. Anwendungen können diese Werkzeuge ruhigen Gewissens verwenden, solange sie in der Lage sind, unsichere Komponenten schnell gegen Alternativen auszutauschen.

Um IT-Sicherheit auch für die Zukunft zu garantieren, müssen wir ein Portfolio sicherer kryptographischer Funktionen vorbereiten. Es benötigt Verfahren, die sowohl für die Welt der allgegenwärtigen (weniger leistungsfähigen) Computer geeignet sind als auch dann sicher bleiben, wenn es leistungsfähige Quantencomputer gibt. Einige viel versprechende Kandidaten für

[3] In JCT findet man in der Standard-Perspektive unter dem Hauptmenü-Eintrag **Visualisierungen** verschiedene Komponenten und Ausprägungen davon: die Einmalsignatur WOTS+, die normale Merkle-Signatur (MSS) und die Extended-Merkle-Signatur (XMSS).

ein solches Portfolio wurden in diesem Artikel vorgestellt; diese müssen jedoch noch sorgfältig erforscht und für die Praxis aufbereitet werden. Die Frage nach einem Verfahren zur Sicherung langfristiger Vertraulichkeit bleibt ein wichtiges offenes Problem, auf das sich zukünftige Forschung fokussieren sollte.

Literaturverzeichnis (Crypto2020)

[AES02] National Institute of Standards and Technology (NIST): *Federal Information Processing Standards Publication 197: Advanced Encyption Standard*, 2002.

[BB85] Bennett, Charles H. und Gilles Brassard: *An Update on Quantum Cryptography.* In: Blakley, G. R. und David Chaum (Herausgeber): *Advances in Cryptology – CRYPTO '84*, Band 196 der Reihe *Lecture Notes in Computer Science*, Seiten 475–480. Springer-Verlag, 1985.

[BCD+06] Buchmann, Johannes, Luis Carlos Coronado García, Erik Dahmen, Martin Döring und Elena Klintsevich: *CMSS – an improved Merkle signature scheme.* In: Barua, Rana und Tanja Lange (Herausgeber): *7th International Conference on Cryptology in India - Indocrypt'06*, Nummer 4392 in *Lecture Notes in Computer Science*, Seiten 349–363. Springer-Verlag, 2006.

[DES77] U.S. Department of Commerce, National Bureau of Standards, National Technical Information Service, Springfield, Virginia: *Federal Information Processing Standards Publication 46: Data Encryption Standard*, 1977.

[Mas02] Maseberg, Jan Sönke: *Fail-Safe-Konzept für Public-Key-Infrastrukturen.* Dissertation, TU Darmstadt, 2002.

[McE78] McEliece, Robert J.: *A public key cryptosystem based on algebraic coding theory.* DSN progress report, 42–44:114–116, 1978.

[Mer79] Merkle, Ralph C.: *Secrecy, authentication, and public key systems.* Dissertation, Department of Electrical Engineering, Stanford University, 1979.

[RSA78] Rivest, Ron L., Adi Shamir und Leonard Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21(2):120–126, April 1978.

[Sho97] Shor, Peter W.: *Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer.* SIAM Journal on Computing, 26(5):1484–1509, 1997.

Alle Links im Artikel wurden am 15.07.2016 überprüft.

Anhang A

Anhang

A.1 CrypTool-1-Menübaum

Dieser Anhang enthält auf der folgenden Seite den kompletten Menübaum der CrypTool-Version 1.4.31[1].

Das Haupt-Menü von CT1 enthält die generellen Service-Funktionen in den sechs Haupt-Menü-Einträgen

- Datei
- Bearbeiten
- Ansicht
- Optionen
- Fenster
- Hilfe,

und die eigentlichen Krypto-Funktionen in den vier Haupt-Menü-Einträgen

- Ver-/Entschlüsseln
- Digitale Signaturen/PKI
- Einzelverfahren
- Analyse.

Unter `Einzelverfahren` finden sich auch die Visualisierungen von Einzelalgorithmen und von Protokollen. Manche Verfahren sind sowohl als schnelle Durchführung (meist unter dem Menü `Ver-/Entschlüsseln`) als auch als Schritt-für-Schritt-Visualisierung implementiert.

Welche Menüeinträge in CrypTool 1 gerade aktiv (also nicht ausgegraut) sind, wird durch den Typ des aktiven Dokumentfensters bestimmt: So ist z. B. die Brute-Force-Analyse für DES nur verfügbar, wenn das aktive Fenster in Hexadezimal-Darstellung geöffnet ist, während der Menüeintrag „Zufallsdaten erzeugen...“ immer verfügbar ist (auch wenn kein Dokument geöffnet ist).

[1] Während sich seit 2010 Änderungen an der langjährig stabilen Version CrypTool 1 (**CT1**) vor allem auf Bugfixes beschränkten, fließen viele Neuerungen in die beiden Nachfolgeversionen CrypTool 2 (**CT2**) und JCrypTool (**JCT**) ein:
- Webseite CT2: `http://www.cryptool.org/de/ct2-dokumentation`
- Webseite JCT: `http://www.cryptool.org/de/jct-machmit`
Von beiden Nachfolgern stehen stabile Versionen zur Verfügung. Stand Februar 2017 gibt es CT 2.0 Release und CT 2.1 Beta-1 plus tägliche Nightly Builds; von JCT gibt es den RC8 und jeden Samstag ein neues Weekly Build.

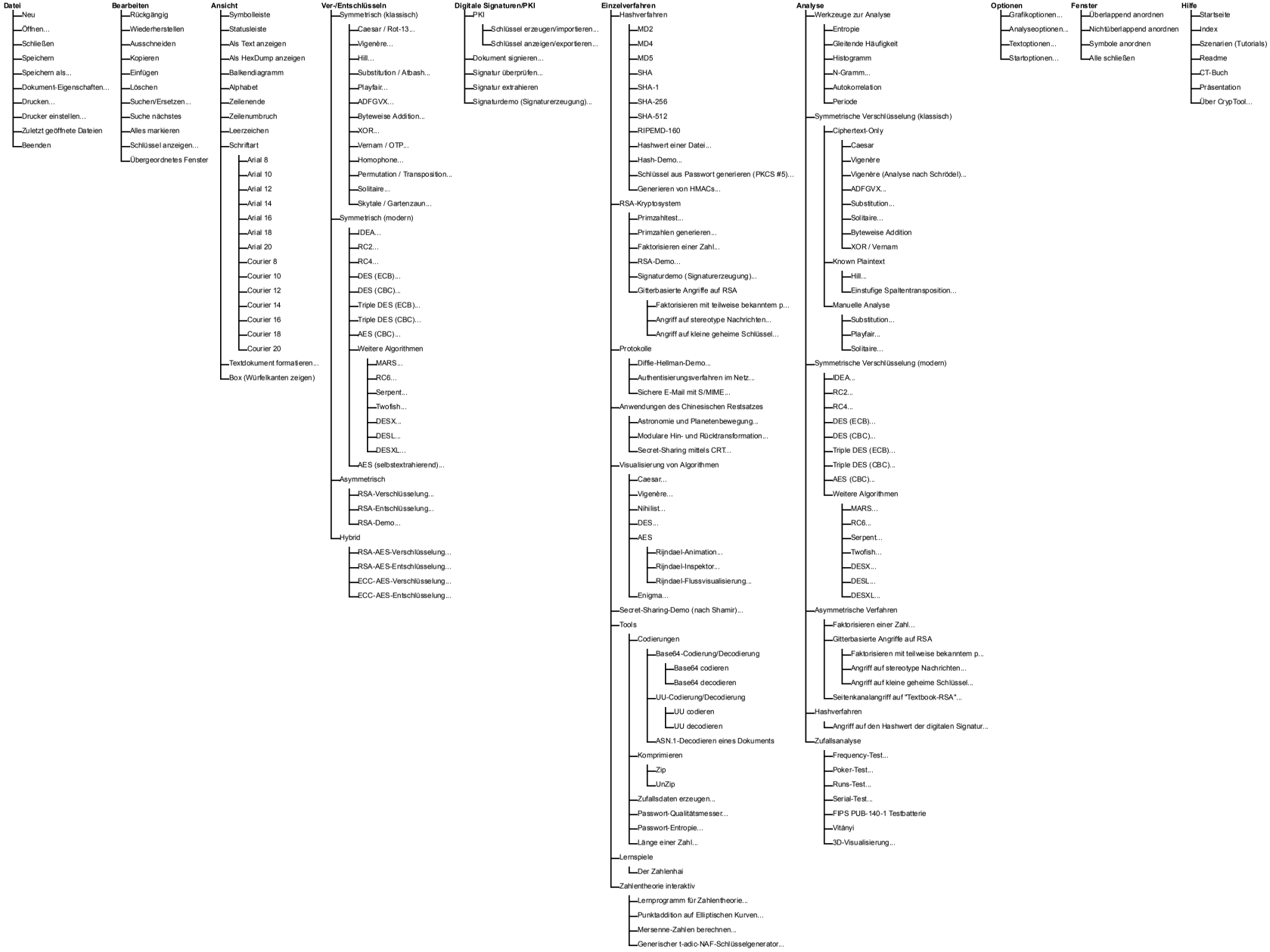


Abbildung A.1: Komplett-Übersicht über den Menü-Baum von CT1 (CrypTool 1.4.31)

A.2 CrypTool-2-Vorlagen

Dieser Anhang enthält auf den folgenden Seiten den Baum mit allen Vorlagen in CrypTool 2.[2] Beim Start von CT2 öffnet sich das Startcenter.

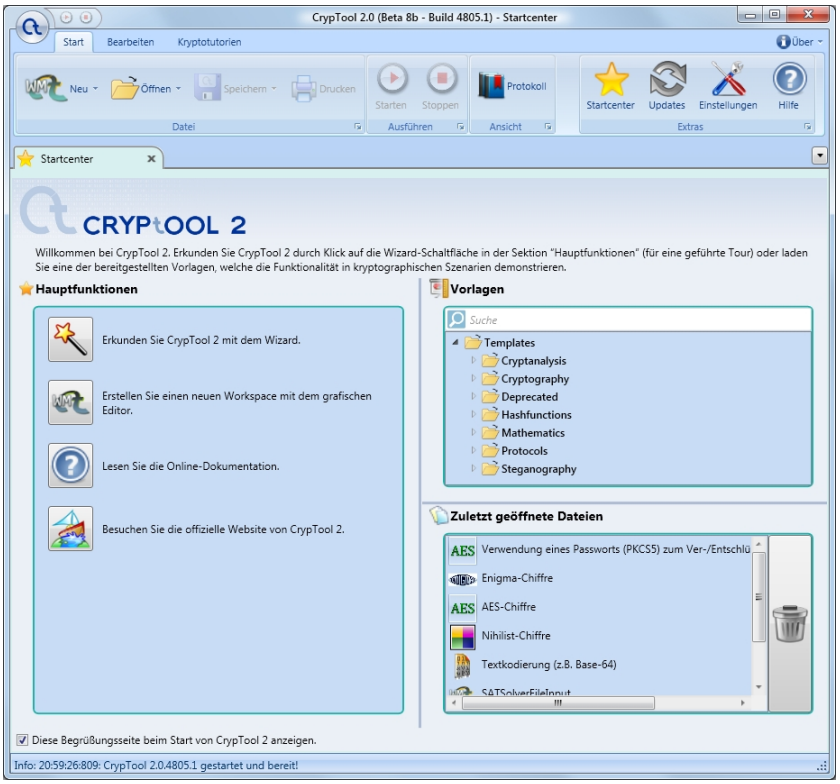


Abbildung A.2: Startcenter in CT2 (Beta 8b, Mai 2012)

Darin hat man die Auswahl, die Funktionalität auf drei verschiedenen Wegen aufzurufen:

- Über den Wizard: Er leitet einen geführt zu den Funktionen.
- Über die Arbeitsfläche, auf der man die Komponenten (z.B. eine Verschlüsselungsfunktion, eine Texteingabefunktion, ...) anhand der visuellen Programmierung selbst zusammenstellen kann.
- Über den Vorlagen-Baum, aus dem man fertige Workflows auswählen kann.

Der Wizard stellt Fragen zu dem gewünschten Szenario (z.B. Base64-Codierung) und führt den Benutzer dann zu den Funktionen. Das gewählte Szeanrio mit den eigenen Eingaben kann man anschließend auch als Vorlage abspeichern.

Auf die leere Arbeitsfläche kann man aus der linken Navigationsleiste alle Komponenten ziehen und diese dann wie gewünscht miteinander verbinden. Die implementierte Krypto-Funktionalität steckt vor allem in diesen Komponenten (z.B. Enigma, AES).

Im Vorlagen-Baum gibt es zu jeder Komponente mindestens eine Vorlage. Die angebotenen Vorlagen enthalten sofort lauffähige komplette Workflows. Wenn man z.B. in der Vorlage zu

[2] Weitere Informationen zu CT2 finden Sie auf: `https://www.cryptool.org/de/ct2-dokumentation`

AES seine Eingaben ändert, sieht man dynamisch und sofort, wie sich Ausgaben entsprechend ändern (wie z.B. durch Padding ein Block hinzukommt, wie sich das Chaining auswirkt, ...).

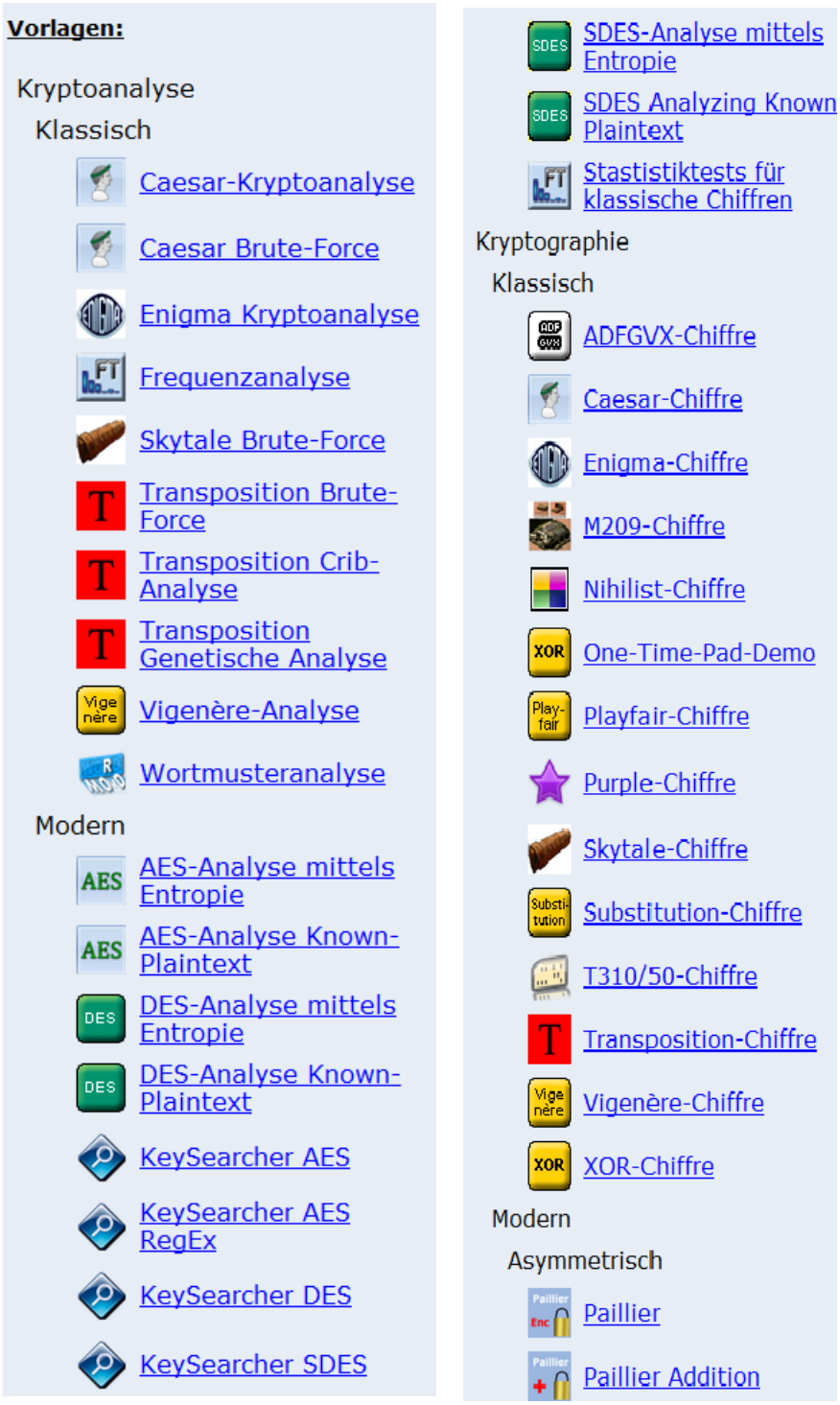


Abbildung A.3: Screenshot über den Template-Baum von CT2 (NB4882.1, Juli 2012), Teil 1

A.3 JCrypTool-Funktionen

Dieser Anhang enthält auf den folgenden Seiten eine Liste aller Funktionen in JCrypTool.[3] Beim ersten Start von JCT öffnet sich das Willkommen-Fenster.

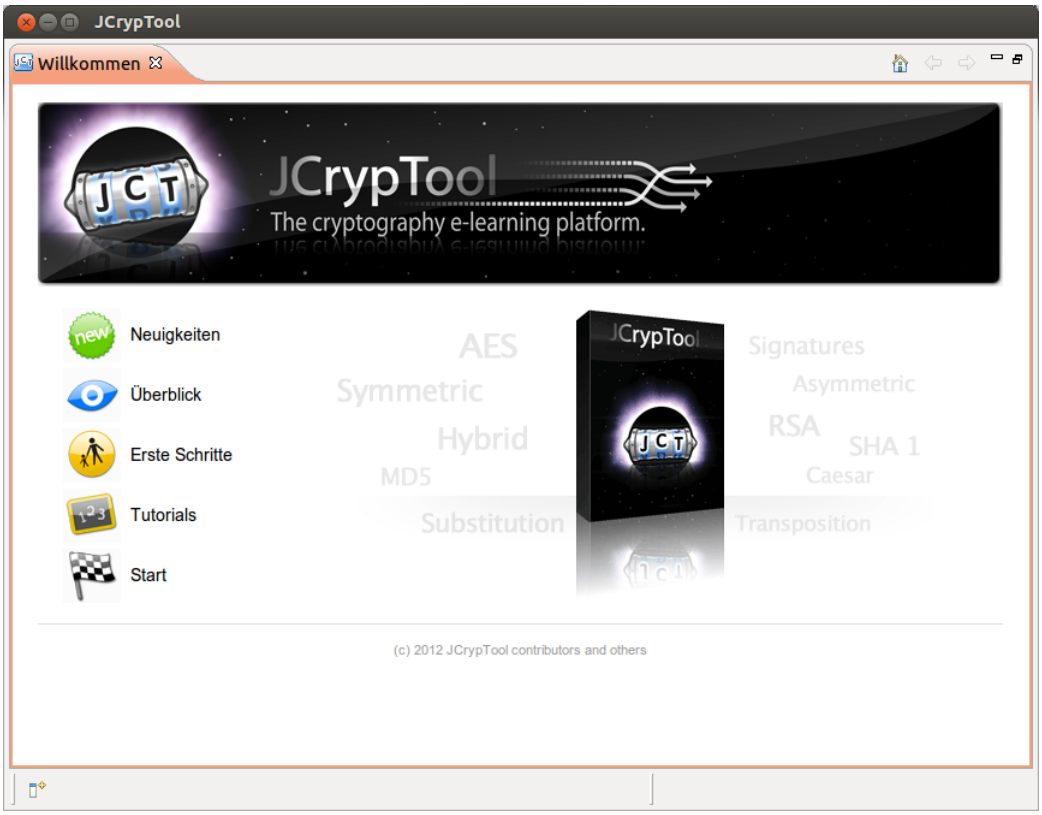


Abbildung A.4: Willkommen-Fenster in JCT (RC6, Juli 2012)

Mit Klick auf „Start“ kann man die verschiedenen Funktionen direkt nutzen. Die in JCT implementierten Funktionen werden über zwei unterschiedliche Perspektiven angeboten:

- Standard-Perspektive
- Algorithmen-Perspektive

Alle Funktionen in der **Standard-Perspektive** finden sich sowohl in den Menüs als auch in der „Krypto-Explorer“ genannten Navigationsleiste (rechts). Die Standard-Perspektive enthält alle wichtigen Verfahren (wie z.B. die klassische Transposition oder den modernen AES) und viele Visualisierungen (z.B. Diffie-Hellman-Schlüsselaustausch oder Berechnungen auf Elliptischen Kurven).

Alle Funktionen der **Algorithmen-Perspektive** finden sich in der „Algorithmen“ genannten Navigationsleiste (in dieser Perspektive ebenfalls rechts). Die Algorithmen-Perspektive enthält alle Detaileinstellungen der verschiedenen Algorithmen und bietet insbesondere auch Algorithmen aus dem Bereich des Post-Quantum-Computings (PQC) an.

[3] Weitere Informationen zu JCT finden Sie auf: `http://www.cryptool.org/de/jct-machmit`
Die Liste wurde mit Hilfe der CT-Portal-Webseite gewonnen.

Function	JCrypTool*	Path in JCT
ADFGVX	D	Algorithms\ Classic\ ADFGVX
Autokey Vigenère	D	Algorithms\ Classic\ Autokey-Vigenère
Bifid	D	Algorithms\ Classic\ Bifid
Caesar	D	Algorithms\ Classic\ Caesar
Double Box	D	Algorithms\ Classic\ Double Box
Playfair	D	Algorithms\ Classic\ Playfair
Substitution	D	Algorithms\ Classic\ Substitution
Transposition	D	Algorithms\ Classic\ Transposition
Vigenère	D	Algorithms\ Classic\ Vigenère
XOR	D	Algorithms\ Classic\ XOR
3DES	F	Algorithms\ Block Ciphers\ DESede
AES	D\F	Algorithms\ Symmetric\ AES Algorithms\ Block Ciphers\ Rijndael
Camellia	F	Algorithms\ Block Ciphers\ Camellia
Dragon	D	Algorithms\ Symmetric\ Dragon
ECIES	F	Algorithms\ Hybrid Ciphers\ ECIES
ElGamal	D\F	Algorithms\ Asymmetric\ ElGamal Algorithms\ Asymmetric Block Ciphers\ ElGamal
IDEA	D\F	Algorithms\ Symmetric\ IDEA Algorithms\ Block Ciphers\ IDEA
LFSR	D	Algorithms\ Symmetric\ LFSR
MARS	F	Algorithms\ Block Ciphers\ MARS
McEliece Fujisaki	F	Algorithms\ Hybrid Ciphers\ McElieceFujisakiCipher
McEliece Kobara Imai	F	Algorithms\ Hybrid Ciphers\ McElieceKobaraImaiCipher
McEliece PKCS	F	Algorithms\ Asymmetric Block Ciphers\ McEliecePKCS
McEliece Pointcheval	F	Algorithms\ Hybrid Ciphers\ McEliecePointchevalCipher
MeRSA	F	Algorithms\ Asymmetric Block Ciphers\ MeRSA
Misty-1 (Kasumi)	F	Algorithms\ Block Ciphers\ Misty1
MpRSA	F	Algorithms\ Asymmetric Block Ciphers\ MpRSA
Niederreiter	F	Algorithms\ Asymmetric Block Ciphers\ Niederreiter
RC2	F	Algorithms\ Block Ciphers\ RC2
RC5	F	Algorithms\ Block Ciphers\ RC5
RC6	D\F	Algorithms\ Symmetric\ RC6 Algorithms\ Block Ciphers\ RC6
RSA	D\F	Algorithms\ Asymmetric\ RSA Algorithms\ Asymmetric Block Ciphers\ RSA_PKCS1_v1_5 Algorithms\ Asymmetric Block Ciphers\ RSA_PKCS1_v2_1
SAFER+	F	Algorithms\ Block Ciphers\ SAFER+
SAFER++	F	Algorithms\ Block Ciphers\ SAFER++
Serpent	F	Algorithms\ Block Ciphers\ Serpent
Shacal	F	Algorithms\ Block Ciphers\ Shacal
Shacal2	F	Algorithms\ Block Ciphers\ Shacal2
Twofish	F	Algorithms\ Block Ciphers\ Twofish
XML-Security	D	Algorithms\ XML Security\
CBC MACs	F	Algorithms\ Message Authentication Codes\ CBCMac\
Cipher-based MACs (CMAC)	F	Algorithms\ Message Authentication Codes\ CMac\
DHA-256	F	Algorithms\ Message Digests\ DHA256
FORK-256	F	Algorithms\ Message Digests\ FORK256
HMACs	D\F	Algorithms\ MAC\ HMacMD5 Algorithms\ Message Authentication Codes\ Hmac\
MD4	F	Algorithms\ Message Digests\ MD4
MD5	D\F	Algorithms\ Hash\ MD5 Algorithms\ Message Digests\ MD5
PKCS#5	F	Algorithms\ Password-based ciphers\
RIPEMD	F	Algorithms\ Message Digests\ RIPEMD
SHA3 candidates	D	Algorithms\ Hash\ SHA3-Candidates

Abbildung A.5: Screenshot zu den Funktionen in JCT (RC6, Juli 2012), Teil 1

Function	JCrypTool*	Path in JCT
SHA family	D\F	Algorithms\ Hash\ SHA Algorithms\ Message Digests\ SHA
Tiger	F	Algorithms\ Message Digests\ Tiger
Two-Track-MAC	F	Algorithms\ Message Authentication Codes\ TwoTrackMac\
VSH	F	Algorithms\ Message Digests\ VSH
CMSS	F	Algorithms\ Signatures\ CMSSSignature\
DSA	D\F	Algorithms\ Signature\ DSA Algorithms\ Signatures\ DSASignature\
ECDSA	F	Algorithms\ Signatures\ ECDSASignature\
GMSS	F	Algorithms\ Signatures\ GMSSSignature\
IQDSA	F	Algorithms\ Signatures\ IQDSASignature\
IQGQ	F	Algorithms\ Signatures\ IQGQSignature\
IQRDSA	F	Algorithms\ Signatures\ IQRDSASignature\
Merkle OTS	F	Algorithms\ Signatures\ MerkleOTSSignature\
MeRSA	F	Algorithms\ Signatures\ MeRSA
MpRSA	F	Algorithms\ Signatures\ MpRSA
Niederreiter CFS	F	Algorithms\ Signatures\ NiederreiterCFS
PKCS#1 (RSA Signatures)	F	Algorithms\ Signatures\ RSASignaturePKCS1v15\
RSASSA-PSS	F	Algorithms\ Signatures\ RSASSA-PSS
SHA1 ECNR	F	Algorithms\ Signatures\ SHA1\ECNR
SSL (MD5 and SHA1 with RSA)	F	Algorithms\ Signatures\ SHA1\SSL_MD5andSHA1withRSA
Blum Blum Shub (B.B.S.)	F	Algorithms\ Pseudo Random Number Generators\ BBSRandom
Elliptic Curve PRNG	F	Algorithms\ Pseudo Random Number Generators\ ECPRNG
SHA1	D	Algorithms\ Random Number Generator\ SHA1
SHA1 PRNG	F	Algorithms\ Pseudo Random Number Generators\ SHA1PRNG
Entropy	D	Analysis\ Entropy Analysis
Frequency Test	D	Analysis\ Frequency Analysis
Friedman Test	D	Analysis\ Friedman Test
Transposition Analysis	D	Analysis\ Transposition Analysis
Vigenère Analysis	D	Analysis\ Vigenère Breaker
Ant Colony Optimization	D	Visuals\ Ant Colony Optimization
Chinese Remainder Theorem	D	Visuals\ Chinese Remainder Theorem
Differential Power Analysis	D	Visuals\ Differential Power Analysis \ Double and Add
Diffie-Hellman Key Exchange	D	Visuals\ Diffie-Hellman Key Exchange (EC)
ElGamal	D	Visuals\ ElGamal Cryptosystem
Elliptic Curve Cryptography	D	Visuals\ ECC Demonstration
Extended Euclidian	D	Visuals\ Extended Euclidian
Feige Fiat Shamir	D	Visuals\ Feige Fiat Shamir
Fiat Shamir	D	Visuals\ Fiat Shamir
Graph Isomorphism	D	Visuals\ Graph Isomorphism
Grille	D	Visuals\ Grille
Homomorphic Encryption	D	Visuals\ Homomorphic Encryption
Kleptography	D	Visuals\ Kleptography
Magic Door	D	Visuals\ Magic Door
Multipartite Key Exchange	D	Visuals\ Multipartite Key Exchange
RSA	D	Visuals\ RSA Cryptosystem
Shamir's Secret Sharing	D	Visuals\ Shamir's Secret Sharing
Verifiable Secret Sharing (VSS)	D	Visuals\ Verifiable Secret Sharing
Simple Power Analysis	D	Visuals\ Simple Power Analysis \ Square and Multiply
Viterbi	D	Visuals\ Viterbi
XML-Security	D	Algorithms\ XML Security\
Number Shark	D	Games\ Number Shark
Sudoku Solver	D	Games\Sudoku
	*	
	D = Default Perspective	
	F = Functional Perspective	

Abbildung A.6: Screenshot zu den Funktionen in JCT (RC6, Juli 2012), Teil 2

A.4 CrypTool-Online-Funktionen

Dieser Anhang enthält eine Liste aller Funktionen in CrypTool-Online (CTO).[4]
Der folgende Screenshot zeigt die auf CTO implementierten Krypto-Funktionen:

[4]Weitere Informationen zu CTO finden Sie auf: `www.cryptool-online.org`
Die Liste wurde mit Hilfe der Funktionsliste auf der CT-Portal-Webseite gewonnen:
`https://www.cryptool.org/de/ctp-dokumentation/ctp-funktionsumfang`

Function	Path at CTO Website	
ADFGVX	Ciphers \ ADFGVX	
Alberti	Ciphers \ Alberti	
AMSCO	Ciphers \ AMSCO	
Autokey Vigenère	Ciphers \ Autokey	
Beaufort	Ciphers \ Beaufort	
Bifid	Ciphers \ Bifid	
Caesar	Ciphers \ Caesar / Rot-13	
Enigma	Ciphers \ Enigma	
Four-Square	Ciphers \ Four-Square	
Freemason	Ciphers \ Freemason	
Gronsfeld	Ciphers \ Gronsfeld	
Hill	Ciphers \ Hill	
Homophone	Ciphers \ Homophonic	
Kamasutra	Ciphers \ Kamasutra	
Larrabee	Ciphers \ Larrabee	
Multiplicative	Ciphers \ Multiplicative	
Navajo	Ciphers \ Navajo	
Nihilist	Ciphers \ Nihilist	
Playfair	Ciphers \ Playfair	
Pollux	Ciphers \ Pollux	
Polybius	Ciphers \ Polybius	
Porta	Ciphers \ Porta	
Rotation	Ciphers \ Rotation	
Scytale	Ciphers \ Scytale	
Substitution	Ciphers \ Monoalphabetic Substitution	
Templar	Ciphers \ Templar	
Transposition	Ciphers \ Transposition	
Trithemius	Ciphers \ Trithemius	
Vigenère	Ciphers \ Vigenère	
Zigzag (Rail Fence)	Ciphers \ Zigzag (Rail Fence)	
AES	Highlights \ AES	
Autocorrelation	Cryptanalysis \ Autocorrelation	
Break Autokey	Cryptanalysis \ Break Autokey	
Frequency Test	Cryptanalysis \ Frequency Analysis	
N-Gram Analysis	Cryptanalysis \ N-Gram Analysis	
Vigenère Analysis	Cryptanalysis \ Break Vigenère	
ASCII	Codings \ ASCII	
Bacon	Codings \ Bacon	
Barcode Generator	Codings \ Barcode generator	
Base 64	Codings \ Base64	
Code 39	Codings \ Code39	
Huffman	Codings \ Huffman	
Morse code	Codings \ Morse code	
Matrix Screensaver	Highlights \ Matrix Screensaver	
Password Quality	Highlights \ Password Check	
Taxman	Highlights \ Taxman	

Abbildung A.7: Screenshot zu den Funktionen in CTO (November 2012)

A.5 Filme und belletristische Literatur mit Bezug zur Kryptographie

Kryptographische Verfahren – sowohl klassische wie moderne – fanden auch Eingang in die Literatur und in Filme. In manchen Medien werden diese nur erwähnt und sind reine Beigabe, in anderen sind sie tragend und werden genau erläutert, und manchmal ist die Rahmenhandlung nur dazu da, dieses Wissen motivierend zu transportieren. Anbei der Beginn eines Überblicks.

A.5.1 Für Erwachsene und Jugendliche

[Poe1843] Edgar Allan Poe,
Der Goldkäfer, 1843.[5]
Diese Kurzgeschichte erschien in Deutsch z.B. in der illustrierten und mit Kommentaren in den Marginalspalten versehenen Ausgabe „Der Goldkäfer und andere Erzählungen“, Gerstenbergs visuelle Weltliteratur, Gerstenberg Verlag, Hildesheim, 2002.
In dieser Kurzgeschichte beschreibt Poe als Ich-Erzähler seine Bekanntschaft mit dem sonderbaren Legrand. Mit Hilfe eines an der Küste Neuenglands gefundenen Goldkäfers, einem alten Pergament und den Dechiffrierkünsten von Legrand finden Sie den sagenhaften Schatz von Kapitän Kidd.
Die Geheimschrift besteht aus 203 kryptischen Zeichen und erweist sich als allgemeine monoalphabetische Substitutions-Chiffre (vgl. Kapitel 2.2.1). Ihre schrittweise Dechiffrierung durch semantische und syntaktische Analyse (Häufigkeit der einzelnen Buchstaben in englischen Texten) wird in der Geschichte ausführlich erläutert.[6]
Der Entschlüsseler Legrand sagt darin (S. 39) den berühmten Satz: „Und es ist wohl sehr zu bezweifeln, ob menschlicher Scharfsinn ein Rätsel ersinnen kann, das menschlicher Scharfsinn bei entsprechender Hingabe nicht wieder zu lösen vermag.“

[Verne1885] Jules Verne,
Mathias Sandorf, 1885.
Dies ist einer der bekanntesten Romane des französischen Schriftstellers Jules Verne (1828-1905), der auch als „Vater der Science Fiction“ bezeichnet wurde.
Erzählt wird die spannende Geschichte des Freiheitskämpfers Graf Sandorf, der an die Polizei verraten wird, aber schließlich fliehen kann.
Möglich wurde der Verrat nur, weil seine Feinde eine Geheimbotschaft an ihn abfangen und entschlüsseln konnten. Dazu benötigten sie eine besondere Schablone, die sie ihm stahlen. Diese Schablone bestand aus einem quadratischen Stück Karton mit 6x6 Kästchen, wovon 1/4, also neun, ausgeschnitten waren (vgl. die Fleißner-Schablone in Kapitel 2.1.1).

---

[5] Siehe `https://de.wikipedia.org/wiki/Der_Goldk%C3%A4fer`.
Eine didaktisch aufbereitete Beschreibung zum Einsatz im Schulunterricht findet sich in Teil 1 der Artikelserie *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle*. Siehe [WLS98], S. 52 ff („Das Gold des Gehenkten“).
Alle Materialien zu der Goldkäfer-Unterrichtsstunde (oder Doppelstunde) finden sich unter `http://www.informatik-im-kontext.de/` via „E-Mail (nur?) für Dich“ => „Vertraulichkeit mit Verschlüsselungsverfahren“.
Poe war nicht nur ein bekannter – in seiner Heimat Amerika zunächst verkannter – Schriftsteller und Erfinder des Kriminalromans, sondern auch ein begabter Kryptologe. Die Geschichte dazu wird auch in dem Krypto-Buch *Verschlüsselte Botschaften* [Kip97] erzählt.

[6] Teil der oben genannten Unterrichtsmaterialien ist auch ein Python-Programm. Damit lässt sich das Kryptogramm mit Python *und* mit SageMath entschlüsseln. Siehe das Code-Beispiel in A.5.3.

[Kipling1901] Rudyard Kipling,
Kim, 1901.
Dieser Roman wird in der Besprechung von Rob Slade[7] folgendermaßen beschrieben: „Kipling packte viele Informationen und Konzepte in seine Geschichten. In „Kim“ geht es um das große „Spiel“ Spionage und Bespitzelung. Schon auf den ersten 20 Seite finden sich Authentisierung über Besitz, Denial of Service, Sich-für-jemand-anderen-Ausgeben (Impersonation), Heimlichkeit, Maskerade, Rollen-basierte Autorisierung (mit Ad-hoc-Authentisierung durch Wissen), Abhören, und Vertrauen basierend auf Datenintegrität. Später kommen noch Contingency Planning gegen Diebstahl und Kryptographie mit Schlüsselwechsel hinzu.“
Das Copyright des Buches ist abgelaufen.[8]

[Doyle1905] Arthur Conan Doyle,
Die tanzenden Männchen, 1905.
In der Sherlock-Holmes-Erzählung *Die tanzenden Männchen* (erschienen erstmals 1903 im „Strand Magazine“, und dann 1905 im Sammelband „Die Rückkehr des Sherlock Holmes“ erstmals in Buchform) wird Sherlock Holmes mit einer Geheimschrift konfrontiert, die zunächst wie eine harmlose Kinderzeichnung aussieht.
Sie erweist sich als monoalphabetische Substitutions-Chiffre (vgl. Kapitel 2.2.1) des Verbrechers Abe Slaney. Holmes knackt die Geheimschrift mittels Häufigkeitsanalyse.

[Sayers1932] Dorothy L. Sayers,
Zur fraglichen Stunde und Der Fund in den Teufelsklippen (Orginaltitel: Have his carcase), Harper, 1932
(Erste dt. Übersetzung *Mein Hobby: Mord* bei A. Scherz, 1964;
dann *Der Fund in den Teufelsklippen* bei Rainer Wunderlich-Verlag, 1974;
Neuübersetzung 1980 im Rowohlt-Verlag).
In diesem Roman findet die Schriftstellerin Harriet Vane eine Leiche am Strand und die Polizei hält den Tod für einen Selbstmord. Doch Harriet Vane und der elegante Amateurdetektiv Lord Peter Wimsey klären in diesem zweiten von Sayers's berühmten Harriet Vane's Geschichten den widerlichen Mord auf.
Dazu ist ein Chiffrat zu lösen. Erstaunlicherweise beschreibt der Roman nicht nur detailliert die Playfair-Chiffre, sondern auch deren Kryptoanalyse (vgl. Playfair in Kapitel 2.2.3).

[Simmel1970] Johannes Mario Simmel,
Und Jimmy ging zum Regenbogen, Knaur Verlag, 1970.
Der Roman spielt zwischen 1938 und 1969 in Wien. Der Held Manuel Aranda deckt – von mehreren Geheimdiensten verfolgt – im Laufe der Handlung Stück für Stück die Vergangenheit seines ermordeten Vaters auf. Ein wichtiger Mosaikstein ist dabei ein verschlüsseltes Manuskript, das in Kapitel 33 entschlüssselt wird. Im Roman wird der Code als ein „fünfundzwanzigfacher Caesar Code“ beschrieben, tatsächlich ist es eine Vigenère-Chiffre mit einem 25 Buchstaben langen Schlüssel.

[7] Siehe `http://catless.ncl.ac.uk/Risks/24.49.html#subj12`.
[8] Sie können es lesen unter:
`http://whitewolf.newcastle.edu.au/words/authors/K/KiplingRudyard/prose/Kim/index.html`,
`http://kipling.thefreelibrary.com/Kim` oder
`http://www.readprint.com/work-935/Rudyard-Kipling`.

Das Buch wurde 1971 verfilmt.

[**Crichton1988**] Michael Crichton,
Die Gedanken des Bösen (Orginaltitel: Sphere), Rororo, 1988.
Ein Team verschiedener Wissenschaftler wird auf den Meeresgrund geschickt, um ein 900 m langes hoch entwickeltes Raumschiff zu untersuchen. Die Eigenheiten und psychischen Probleme der Forscher treten durch lebensbedrohliche Ereignisse und ihr Abgeschnittensein von oben immer mehr in den Vordergrund. Es gibt viele Rätsel: Das Raumschiff liegt schon 300 Jahre da, es hat englische Beschriftungen, es führt scheinbar ein Eigenleben, die menschliche Vorstellungskraft materialisiert sich. Unter anderem erscheint auf dem Bildschirm ein im Buch vollständig abgedruckter Code, der von dem genialen Mathematiker Harry entschlüsselt werden kann: ein einfacher spiralenförmiger Ersetzungscode.

[**Seed1990**] Regie Paul Seed (Paul Lessac),
Das Kartenhaus (Orginaltitel: House of Cards), 1990 (dt. 1992).
In diesem Film versucht Ruth, hinter das Geheimnis zu kommen, das ihre Tochter verstummen ließ. Hierin unterhalten sich Autisten mit Hilfe von 5- und 6-stelligen Primzahlen (siehe Kapitel 3). Nach über eine Stunde kommen im Film die folgenden beiden (nicht entschlüsselten) Primzahlfolgen vor:

21.383, 176.081, 18.199, 113.933, 150.377, 304.523, 113.933
193.877, 737.683, 117.881, 193.877

[**Robinson1992**] Regie Phil Alden Robinson,
Sneakers - Die Lautlosen (Orginaltitel: Sneakers), Universal Pictures Film, 1992.
In diesem Film versuchen die „Sneakers“ (Computerfreaks um ihren Boss Martin Bishop), den „Bösen“ das Dechiffrierungsprogramm SETEC abzujagen. SETEC wurde von einem genialen Mathematiker vor seinem gewaltsamen Tod erfunden und kann alle Geheimcodes dieser Welt entschlüsseln.
In dem Film wird das Verfahren nicht beschrieben[9].

[**Baldacci1997**] David Baldacci,
Das Labyrinth. Total Control, Lübbe, 1997.
Jason Archer, Direktor einer Technologie-Firma, verschwindet plötzlich. Seine Frau Sidney Archer versucht, den Grund seines plötzlichen Todes herauszufinden, und entdeckt, wie das Finanzsystem missbraucht wird und dass die reale Macht bei denen mit dem meisten Geld liegt. Hier helfen dann auch gute Passworte nicht...

[**Natali1997**] Regie Vincenzo Natali,
Cube (Orginaltitel: Sneakers), Mehra Meh Film, 1997.
In diesem kanadischen Low-Budget-Film finden sich 7 sehr unterschiedliche Personen in

[9] An dem Film hatte Leonard Adleman (das „A“ von RSA) als mathematischer Berater mitgearbeitet. Die recht lustige Geschichte über seine Mitwirkung bei Sneakers beschreibt er selbst auf seiner Homepage unter `http://www.usc.edu/dept/molecular-science/fm-sneakers.htm`. Man kann man davon ausgehen, dass es sich bei dem überall benutzten Verschlüsselungsverfahren um RSA handelt. In dem Chip ist demnach ein bis dahin unbekanntes, schnelles Faktorisierungsverfahren implementiert.

einem endlos scheinenden Labyrinth von würfelartigen Räumen.
Die Personen wollen nach draußen, müssen dazu aber die Räume durchqueren, von denen manche tödliche Fallen darstellen. Um herauszufinden, welche Räume gefährlich sind, spielt Mathematik eine entscheidende Rolle: Jeder Raum hat am Eingang eine Folge von 3 mal 3 Ziffern. Zuerst nehmen sie an, dass alle Räume Fallen sind, wo wenigstens eine der 3 Zahlen eine Primzahl ist. Später stellt sich heraus, dass auch alle diejenigen Räume Fallen sind, bei denen eine der 3 Zahlen eine Potenz von genau einer Primzahl ist (Fallen sind also p^n, z.B. $128 = 2^7$ oder $101 = 101^1 = prim$, aber nicht $517 = 11 * 47$).

[**Becker1998**] Regie Harold Becker,
Das Mercury Puzzle (Orginaltitel: Mercury Rising), Universal Pictures Film, 1998.
Die NSA hat einen neuen Code entwickelt, der angeblich weder von Menschen noch von Computern geknackt werden kann. Um die Zuverlässigkeit zu testen, verstecken die Programmierer eine damit verschlüsselte Botschaft in einem Rätselheft.
Simon, eine neunjähriger autistischer Junge, knackt den Code. Statt den Code zu fixen, schickt ihm ein Sicherheitsbeamter einen Killer. Der FBI-Agent Art Jeffries (Bruce Willis) beschützt den Jungen und stellt den Killern eine Falle.
Das Chiffrier-Verfahren wird nicht beschrieben.

[**Brown1998**] Dan Brown,
Diabolus (Orginaltitel: Digital Fortress), Lübbe, 2005.
Dan Browns erster Roman „The Digital Fortress“ erschien 1998 als E-Book, blieb jedoch damals weitgehend erfolglos.
Die National Security Agency (NSA) hat für mehrere Milliarden US-Dollar einen gewaltigen Computer gebaut, mit dem sie in der Lage ist, auch nach modernsten Verfahren verschlüsselte Meldungen (natürlich nur die von Terroristen und Verbrechern) innerhalb weniger Minuten zu entziffern.
Ein abtrünniger Angestellter erfindet einen unbrechbaren Code und sein Computerprogramm Diabolus zwingt damit den Supercomputer zu selbstzerstörerischen Rechenoperationen. Der Plot, in dem auch die schöne Computerexpertin Susan Fletcher eine Rolle spielt, ist ziemlich vorhersehbar.
Die Idee, dass die NSA oder andere Geheimdienste jeden Code knacken können, wurde schon von mehreren Autoren behandelt: Hier hat der Supercomputer 3 Millionen Prozessoren – trotzdem ist es aus heutiger Sicht damit auch nicht annäherungsweise möglich, diese modernen Codes zu knacken.

[**Elsner1999**] Dr. C. Elsner,
Der Dialog der Schwestern, c't, Heise-Verlag, 1999.
In dieser Geschichte, die dem CrypTool-Paket als PDF-Datei beigelegt ist, unterhalten sich die Heldinnen vertraulich mit einer Variante des RSA-Verfahrens (vgl. Kapitel 4.10 ff.). Sie befinden sich in einem Irrenhaus unter ständiger Bewachung.

[**Stephenson1999**] Neal Stephenson,
Cryptonomicon, Harper, 1999.
Der sehr dicke Roman beschäftigt sich mit Kryptographie sowohl im zweiten Weltkrieg als auch in der Gegenwart. Die zwei Helden aus den 40er-Jahren sind der glänzende Mathematiker und Kryptoanalytiker Lawrence Waterhouse, und der übereifrige, morphiumsüchtige

Bobby Shaftoe von den US-Marines. Sie gehören zum Sonderkommando 2702, einer Alliiertengruppe, die versucht, die gegnerischen Kommunikationscodes zu knacken und dabei ihre eigene Existenz geheim zu halten.
In der Gegenwartshandlung tun sich die Enkel der Weltkriegshelden – der Programmierfreak Randy Waterhouse und die schöne Amy Shaftoe – zusammen.
Cryptonomicon ist für nicht-technische Leser teilweise schwierig zu lesen. Mehrere Seiten erklären detailliert Konzepte der Kryptographie. Stephenson legt eine ausführliche Beschreibung der Solitaire-Chiffre (siehe Kapitel 2.4) bei, ein Papier- und Bleistiftverfahren, das von Bruce Schneier entwickelt wurde und im Roman „Pontifex“ genannt wird. Ein anderer, moderner Algorithmus namens „Arethusa“ wird dagegen nicht offengelegt.

[**Elsner2001**] Dr. C. Elsner,
Das Chinesische Labyrinth, c't, Heise-Verlag, 2001.
In dieser Geschichte, die dem CrypTool-Paket als PDF-Datei beigelegt ist, muss Marco Polo in einem Wettbewerb Probleme aus der Zahlentheorie lösen, um Berater des großen Khan zu werden. Alle Lösungen sind angefügt und erläutert.

[**Colfer2001**] Eoin Colfer,
Artemis Fowl, List-Verlag, 2001.
In diesem Jugendbuch gelangt der junge Artemis, ein Genie und Meisterdieb, an eine Kopie des streng geheimen „Buches der Elfen“. Nachdem er es mit Computerhilfe entschlüsselt hat, erfährt er Dinge, die kein Mensch erfahren dürfte.
Der Code wird in dem Buch nicht genauer beschrieben.

[**Howard2001**] Regie Ron Howard,
A Beautiful Mind, 2001.
Verfilmung der von Sylvia Nasar verfassten Biographie des Spieltheoretikers John Nash. Nachdem der brillante, aber unsoziale Mathematiker geheime kryptographische Arbeiten annimmt, verwandelt sich sein Leben in einen Alptraum. Sein unwiderstehlicher Drang, Probleme zu lösen, gefährden ihn und sein Privatleben. Nash ist in seiner Vorstellungswelt ein staatstragender Codeknacker.
Konkrete Angaben zur seinen Analyseverfahren werden nicht beschrieben.

[**Apted2001**] Regie Michael Apted,
Enigma – Das Geheimnis, 2001.
Verfilmung des von Robert Harris verfassten „historischen Romans“ *Enigma* (Hutchinson, London, 1995) über die berühmteste Verschlüsselungsmaschine in der Geschichte, die in Bletchley Park nach polnischen Vorarbeiten gebrochen wurde. Die Geschichte spielt 1943, als der eigentliche Erfinder Alan Turing schon in Amerika war. So kann der Mathematiker Tom Jericho als Hauptperson in einem spannenden Spionagethriller brillieren.
Konkrete Angaben zu dem Analyseverfahren werden nicht gemacht.

[**Isau2003**] Ralf Isau,
Das Museum der gestohlenen Erinnerungen, Thienemann-Verlag, 1997/2003.
Ein sehr spannender, hervorragend recherchierter und doch leicht zu lesender Roman mit einem tiefen Hintersinn.

Als die Zwillinge Oliver und Jessica von ihren Ferien zurückkommen, haben sie ihren Vater vergessen. Die Realität verschiebt sich und niemand scheint es zu bemerken. An einigen Stellen bleiben manchmal Spuren zurück, die man entziffern kann. Zentrum der Geschichte ist das Ischtar-Tor im Berliner Pergamon-Museum. Nur mit dem Scharfsinn einer irischen Professorin (die gleichzeitig Computerexpertin, Archäologin und Philologin ist), den besonderen Beziehungen zwischen Zwillingen und den vereinten Kräften der Computergemeinschaft kann der letzte Teil des Spruches gelöst werden.
Das Buch wurde als bestes Jugendbuch ausgezeichnet und liegt in 8 Sprachen vor.

[**Brown2003**] Dan Brown,
Sakrileg (Orginaltitel: The Da Vinci Code), Lübbe, 2004.
Der Direktor des Louvre wird in seinem Museum vor einem Gemälde Leonardos ermordet aufgefunden, und der Symbolforscher Robert Langdon gerät in eine Verschwörung.
Innerhalb der Handlung werden verschiedene klassische Chiffren (Substitution wie z.B. Caesar oder Vigenère, sowie Transposition und Zahlencodes) angesprochen. Außerdem klingen interessante Nebenbemerkungen über Schneier oder die Sonnenblume an. Der zweite Teil des Buches ist sehr von theologischen Betrachtungen geprägt.
Das Buch ist einer der erfolgreichsten Romane der Welt.

[**McBain2004**] Scott McBain,
Der Mastercode (Orginaltitel: Final Solution), Knaur, 2005.
In einer nahen Zukunft haben Politiker, Militärs und Geheimdienstchefs aus allen Staaten in korrupter Weise die Macht übernommen. Mit einem gigantischen Computernetzwerk names „Mother“ und vollständiger Überwachung wollen sie die Machtverteilung und Kommerzialisierung für immer festschreiben. Menschen werden ausschließlich nach ihrem Kredit-Rating bewertet und global agierende Unternehmen entziehen sich jeder demokratischen Kontrolle. Innerhalb des Thrillers wird die offensichtliche Ungerechtigkeit, aber auch die realistische Möglichkeit dieser Entwicklung immer wieder neu betont.
In den Supercomputer „Mother“ wurde m.H. eines Kryptologen ein Code zur Deaktivierung eingebaut: In einem Wettrennen mit der Zeit versuchen Lars Pedersen, Oswald Plevy, die amerikanische Präsidentin, der britische Regierungschef und eine unbekannte Finnin namens Pia, die den Tod ihres Bruders rächen will, den Code zur Deaktivierung zu starten. Auf der Gegenseite agiert eine Gruppe mörderischer Verschwörer unter Führung des britischen Außenministers und des CIA-Chefs.
Die englische Originalfassung „The Final Solution“ wurde als Manuskript an Harper Collins, London verkauft, ist dort aber nicht erschienen.

[**Burger2006**] Wolfgang Burger,
Heidelberger Lügen, Piper, 2006.
In diesem Kriminalroman mit vielen oft unabhängigen Handlungssträngen und lokalen Geschichten geht es vor allem um den Kriminalrat Gerlach aus Heidelberg. Auf S. 207 f. wird aber auch der kryptologische Bezug von einem der Handlungsstränge kurz erläutert: der Soldat Hörrle hatte Schaltpläne eines neuen digitalen NATO-Entschlüsselungsgerätes kopiert und der Ermordete hatte versucht, seine Erkenntnisse an die Chinesen zu verkaufen.

[**Vidal2006**] Agustin Sanchez Vidal,
Kryptum, Dtv, 2006.
Der erste Roman des spanischen Professors der Kunstgeschichte ähnelt Dan Browns „Sakrileg“ aus dem Jahre 2003, aber angeblich hat Vidal schon 1996 begonnen, daran zu schreiben. Vidals Roman ist zwischen historischem Abenteuerroman und Mystery-Thriller angesiedelt und war in Spanien ein Riesenerfolg.
Im Jahre 1582 wartet Raimundo Randa, der sein Leben lang einem Geheimnis auf der Spur war, im Alkazar auf seinen Inquisitionsprozess. Dieses Geheimnis rankt sich um ein mit kryptischen Zeichen beschriftetes Pergament, von dem eine mysteriöse Macht ausgeht. Rund 400 Jahre später kann sich die amerikanische Wissenschaftlerin Sara Toledano dieser Macht nicht entziehen, bis sie in Antigua verschwindet. Ihr Kollege, der Kryptologe David Calderon, und ihre Tochter Rachel machen sich auf die Suche nach ihr und versuchen gleichzeitig, den Code zu knacken. Aber auch Geheimorganisationen wie die NSA sind hinter dem Geheimnis des „letzten Schlüssels“ her. Sie sind bereit, dafür über Leichen zu gehen.

[**Larsson2007**] Stieg Larsson,
Verdammnis (Originaltitel: Flickan som lekte med elden), Heyne, 2007.
Der Autor wurde 2006 postum mit dem Skandinavischen Krimipreis als bester Krimiautor Skandinaviens geehrt. Die Superheldin Lisbeth Salander nutzt PGP und beschäftigt sich nebenbei auch mit mathematischen Rätseln wie dem Satz von Fermat.

[**Preston2007**] Douglas Preston,
Der Canyon (Orginaltitel: Tyrannosaur Canyon), Knauer, 2007.
Ein spannender Thriller, bei dem es auch darum geht, warum die Dinosaurier ausstarben.

Archäologe Stem Weathers wird im Labyrinth-Canyon erschossen. Noch bevor der Mörder ihn ausrauben kann, übergibt er sein Notizbuch an Tom Broadbent, einen dortigen Tierarzt, der zufällig vorbei kommt.

In dem Notizbuch stehen auf 60 Seiten nur Ziffern. Deshalb bringt Tom es zu dem Ex-CIA-Kryptoanalytiker Wyman Ford, der sich in ein nahegelegenes Wüstenkloster zurückzog, nachdem seine Frau bei einem Einsatz getötet wurde. Zuerst lehnt Wyman jede Unterstützung ab und bezeichnet selbst gebastelte Codes als „Idiotenchiffren“ – von einem Idioten ausgedacht, von jedem Idioten zu entziffern. Mit dem Notizbuch verhält es sich aber nicht ganz so einfach. Nach intensiver Kryptoanalyse findet er heraus, dass die Ziffern keinen Code darstellen, sondern dass es der Output eines Bodenradargeräts mit dem Bild eines gut erhaltenen Tyrannosaurus Rex ist.

Nach rund 250 Seiten gibt es eine überraschende Wende bei den endlosen Verfolgungsjagden: Masago, Chef einer sogenannten Black-Detachment-Einheit der CIA, kommt ins Spiel. Er erklärt: Waffen, die einmal erfunden wurden, werden immer auch eingesetzt. Die Menschheit wird sich ausrotten, aber seine Aufgabe sei es, das möglichst weit hinauszuzögern. Als Leiter der Abteilung LS480 will er mit allen Mitteln verhindern, dass Terroristen Zugang zu neuen gefährlichen biologischen Waffen erhalten.

Der Mörder von Weathers hatte beim Durchsuchen der Leiche nur ein paar Gesteinsproben gefunden und mitgenommen. Diese wurden dann von einer jungen Forscherin namens Melodie Crookshank untersucht, ohne dass sie weiß, woher diese kommen. Sie findet darin eine besondere Virenform, die anscheinend eine außerirdische Lebensform darstellt.

[**Twinig2008**] James Twinig,
Die schwarze Sonne (Orginaltitel: The Black Sun), Bastei Lübbe, 2008.
Ein historisch-basierter Thriller mit einigen konstruierten Elementen, bei dem es auch darum geht, an das versteckte Uran der Nazis zu kommen, natürlich um die Menschheit zu retten ...

Helden sind Tom Kirk, ein in London lebender Ex-CIA-Agent und früherer Kunstdieb, und Dominique de Lecourt – sie liebt Herausforderungen inklusive Rätsel und Codes.

Die einzigen kryptographischen Elemente sind ein „Sprungcode“ (die Verbrecher nutzen das Verfahren zur Kommunikation via Zeitungsanzeigen), Steganographie (um die Enigma-Einstellungen zu verstecken), und eine Enigma-Nachricht (in der die Koordinaten des „Schatzes“ verschlüsselt sind).

Zu Beginn wird eine Enigma mit hohem Aufwand gestohlen, was notwendig ist, um die angelegte Handlung so zustande kommen zu lassen. In der Realität wäre heutzutage ein solcher Diebstahl völlig überflüssig, da es inzwischen hervorragende Software-Emulationen für die Enigma gibt ...

[**Schröder2008**] Rainer M. Schröder,
Die Judas-Papiere, Arena, 2008.
„Historienthriller“: Lord Pembroke hat im Jahre 1899 drei Männer und eine Frau in der Hand und beauftragt sie, die verschlüsselten Botschaften in dem Notizbuch seines verstorbenen Bruders Mortimer zu entschlüsseln und das Judas-Evangelium zu finden, das das Ende der Christenheit einläuten könnte. Dazu müssen sie Rätsel an vielen Orten der Welt lösen. Im Buch finden sich klassische Verfahren wie Polybius (S. 195) oder die Freimaurer-Chiffre (S. 557).

[**Hill2009**] Tobias Hill,
Der Kryptograph (Orginaltitel: The Cryptographer), C. Bertelsmann, 2009.
London 2021: Die Firma SoftMark hat eine elektronische Währung entwickelt und etabliert, die durch einen nicht entschlüsselbaren Code allen Nutzern höchste Sicherheit garantiert. Der Erfinder und Firmengründer John Law, wegen seiner mathematischen Begabung auch der Kryptograph genannt, ist damit zum reichsten Mann der Welt geworden. Doch dann wird der Code geknackt, und in einer dadurch verursachten Weltwirtschaftskrise geht auch die Firma von John Law pleite. Außerdem wird die Steuerfahnderin Anna Moore auf ihn angesetzt.

[**Eschbach2009**] Andreas Eschbach,
Ein König für Deutschland, Lübbe, 2009.
Der Roman dreht sich um die Manipulierbarkeit von Wahlcomputern.
Vincent Merrit, ein junger US-amerikanischer Programmierer, wird erpresst, ein solches Programm zu schreiben. Neben kommerziell orientierten Erpressern kommen z.B. auch Online-Rollenspiele und Live-Rollenspiele (LARPs) in dem Roman vor. Weil Merrit den Missbrauch seines Programms ahnte, baute er eine Hintertür ein: Nimmt eine Partei namens VWM an der Wahl teil, erhält sie automatisch 95 % der Stimmen ...
Die fiktive Handlung des Romans beruht auf zahlreichen überprüfbaren und genau recherchierten Tatsachen, auf die in Fußnoten hingewiesen wird.
Während die kryptographischen Protokolle sicher gemacht werden können, bleiben ihre

Implementierung und ihre Organisation anfällig gegen Missbrauch.

[Juels2009] Ari Juels,
Tetraktys, Emerald Bay Books, 2009 (bisher nur in Englisch).
Die Geschichte deckt die Verwundbarkeit der computer-basierten Identitäten und Sicherheiten auf, indem sie moderne Kyptographie mit klassischer Wissenschaft und Literatur verbindet. Der Kryptograph und Altphilologe Ambrose Jerusalem ist Abgänger der UC Berkeley mit einer schönen Freundin und einer aussichtsreichen Zukunft, bis ihn die NSA rekrutiert, um eine Serie mysteriöser Computereinbrüche zu verfolgen. Viele kleine Puzzlestücke lassen vermuten, dass jemand die RSA-Verschlüsselung gebrochen hat. Hinter den Angriffen scheint ein geheimer Kult zu stecken, Anhänger von Pythagoras, dem großen griechischen Mathematiker und Philosophen, der glaubte, die Wirklichkeit könne nur mit Hilfe eines mystischen Zahlensystems verstanden werden.

[Suarez2010] Daniel Suarez,
Daemon: Die Welt ist nur ein Spiel (Orginaltitel: Daemon), rororo, 2010.
Dies gilt als eines der spannendsten Bücher der letzten Jahre – ein Near-Science Fiction-Thriller, der die Entwicklungen in der realen Welt und die Möglichkeiten von aktuellen Forschungen wie denen von Google-X-Labs (Google-Brille, selbst-steuernde Autos, 3-D-Drucker, ...) in einer plausiblen Geschichte vereint.

Nach dem Tod des Computergenies und Spieleentwicklers Matthew Sobol agiert ein Daemon im Internet, der scheinbar skrupellos immer mehr Menschen und Firmen geschickt manipuliert und ausbildet.

Durch die Beherrschung der Daten ist ihm jeder ausgeliefert. Die Kommunikation seiner Söldner ist geprägt von High-Tech und Verschlüsselung – ebenso die Kommunikation der verteilten Instanzen seiner Inkarnation. Kern ist ein MMORPG-Spiel (Massive Multiplayer Online Role-Playing Game), das stark an WoW erinnert. Auch hierin gibt es verschlüsselte Botschaften, z.B. um die besten Spieler anzuwerben:
m0wFG3PRCoJVTs7JcgBwsOXb3U7yPxBB

Die Handlung ist wiederholungsfrei, komplex, vielfältig, sehr spannend und enthält mit ihrer Kritik an den Plutokraten auch konkrete gesellschaftskritische Elemente. Das Ende ist offen. Und die Ideen scheinen realisierbar in allernächster Zukunft ...

[Olsberg2011] Karl Olsberg,
Rafael 2.0, Thienemann Verlag, 2011, 240 Seiten.
Michael und Rafael Ogilvy sind begabte Zwillinge, die sich sehr gut verstehen. Bevor der unheilbar kranke Rafael stirbt, entwickelt sein Vater ein virtuelles Computer-Ebenbild von ihm, eine künstliche Intelligenz (KI). Das ist ein gut gehütetes Geheimnis, bis Michael eines Tages dahinter kommt, was sein Vater da vor ihm versteckt. Sein erstes Entsetzen verwandelt sich jedoch bald in Freude. So hat er noch etwas, das ihn an seinen Bruder erinnert.
Doch dieses Computersystem ist auch für das Militär interessant. Eines Tages wird Michaels Vater entführt und die Firma und somit auch das Computerprogramm Rafael 2.0 geraten in die falschen Hände. Michael wird von seinem Onkel in ein Internat verbannt, aus dem er aber fliehen kann. Fortan versuchen Michael und seine Freunde alles, um seinen Vater zu finden, von dem sie annehmen, dass er von einer konkurrierenden Firma entführt

wurde. Ab hier wird die Geschichte richtig spannend ... Michael erfährt, dass es eine weitere künstliche Intelligenz, Metraton, gibt, die den Menschen nicht so wohlgesonnen ist. Nichts wird zu sehr vertieft, junge Jugendliche sind die Zielgruppe. Trotzdem entsteht auch Tiefgang, wenn es bspw. um Machenschaften bei Firmenübernahmen geht.
Aus kryptologischer Sicht: Spannend ist der Abschnitt zur Faktorisierung: Mit einer Variante kann Michael erkennen, ob der Computer betrügt ...

[**Burger2011**] Wolfgang Burger,
Der fünfte Mörder, Piper, 2011.
Ort & Zeit der Handlung: Deutschland / Heidelberg, 1990 - 2009. Folge 7 der Alexander-Gerlach-Serie. Beinahe wäre Kriminaloberrat Alexander Gerlach (Ich-Erzähler) Opfer eines Bombenanschlags geworden, als der Geländewagen eines bulgarischen Zuhälters explodiert. Als Gerlach ermittelt, weil er einen Bandenkrieg verhindern will, wird er von oberster Stelle zurückgepfiffen. Journalist Machatschek unterstützt Gerlach, tauscht mit ihm Informationen aber nur per Skype und einem Zusatzprogramm dazu aus, da er nur das für abhörsicher hält.

[**Suarez2011**] Daniel Suarez,
Darknet (Orginaltitel: Freedom (TM)), rororo, 2011.
Dies ist der erschreckend plausible Nachfolger zu „Daemon“ (siehe oben). Gleich zu Beginn werden einige offene Fäden aus dem ersten Buch aufgenommen und geklärt. Die Beschreibungen sind direkter, die Charaktere werden ausgearbeitet, insbesondere Loki. Nachdem in „Daemon“ die Grundlagen gelegt wurden, nutzt Suarez dies, um ein neues Konzept gesellschaftlicher Organisation zu erläutern, die durch Informationstechnologie neue Fähigkeiten erlangt. Dabei werden die Motive deutlich, die sowohl die alten Potentaten als auch die neue Daemon-Gesellschaft treibt, die sich noch während der Geschichte stark weiter entwickelt. Kryptographie wird in diesem Buch als ein natürlicher Teil der modernen Technologie und der modernen Kriegsführung beschrieben. Die neue Gesellschaft in „Darknet“ basiert auf dem Darknet, einer Alternative zum Internet, aufgebaut auf schnellen drahtlosen Meshnetzen, die eine sehr hohe Standfestigkeit und Verfügbarkeit haben. Auch wenn die Geschichte in einigen Teilen schockierend ist, scheint sie realistisch und nicht weit weg von der simultanen Nutzung moderner Technologie, die unser aller Leben durchdringt als virtuelle Welt, die sich über die reale Welt legt.

[**Eschbach2011**] Andreas Eschbach,
Herr aller Dinge, Lübbe, 2011.
Dieser Roman hätte es verdient, viel bekannter zu werden: Die Idee darin des „schrecklichsten aller Verbrechen“, die der Grund der ganzen Geschichte wird, ist neu und geradzu revolutionär, aber auch unendlich traurig. Anhand der scheiternden Paarbeziehung von Hiroshi (Erfindergenie) und Charlotte werden große Themen wie Gerechtigkeit, Wohlstand und Macht behandelt.
Aus kryptographischer Sicht: Hiroshi benutzt verteilte Berechnung und hat ein Verschlüsselungs- und Backup-System entwickelt, dass die Regierung, die ihn verwanzt, in die Irre leitet.

[**Elsberg2012**] Marc Elsberg,
Blackout – Morgen ist es zu spät, Blanvalet, 2012, 800 Seiten.

An einem kalten Wintertag brechen in Europa alle Stromnetze zusammen. Die Behörden, Stromversorger und Sicherheitsfirmen tappen im Dunkeln und können das Problem nicht beheben. Der italienische Informatiker Piero Manzano vermutet, dass hier Terroristen mit Hilfe von Hackern angreifen: In den bei allen Abnehmern eingesetzten Smart-Metern, Software-gesteuerten Stromzählern, wurde die Software manipuliert. Die Sicherheits- und Verschlüsselungskomponenten wurden geknackt, so dass Fremde sie mit falschen Steuerungsbefehlen außer Betrieb setzen konnten. Die erschreckenden Folgen an den unterschiedlichen Orten sind realistisch und spannend erzählt. Ebenso die Reaktionen der Menschen ...

[**Olsberg2013**] Karl Olsberg,
Die achte Offenbarung, Aufbau Taschenbuch, 2013, 460 Seiten.
Kann eine Botschaft aus der Vergangenheit unsere Zukunft verändern? Dem Historiker Paulus Brenner fällt ein uraltes, verschlüsseltes Manuskript aus dem Besitz seiner Familie in die Hände. Doch je mehr er von dem Text dekodiert, desto rätselhafter wird der Inhalt: Denn das Buch sagt mit erstaunlicher Präzision Ereignisse voraus, die zum Zeitpunkt seiner vermuteten Entstehung noch nicht geschehen sind. Während aus einem US-Labor hoch gefährliches Genmaterial verschwindet, will irgendjemand um jeden Preis verhindern, dass Paulus auch die letzte, die achte Offenbarung entziffert. Ein packender Thriller um eine erschreckend realistische Apokalypse mit vielen menschlichen Seiten ...
Als Leser kann man an der Entschlüsselung des Manuskripts teilhaben.
Die Versuche Pauls, seine Entdeckung an die richtigen Stellen zu bringen und sie später zu berichtigen, sind sehr spannend beschrieben – auch Chefredakteure haben ein Dilemma mit Verschwörungstheorien.
Das Chiffrat auf der letzten Buchseite wurde auch als Challenge im Krypto-Wettbewerb MTC3 veröffentlicht: `https://www.mysterytwisterc3.org/de/challenges/level-1-kryptographie-challenges/die-letzte-notiz`

[**Elsberg2014**] Marc Elsberg,
ZERO – Sie wissen, was du tust, Blanvalet Verlag, 2014, 480 Seiten.
London. Bei einer Verfolgungsjagd wird ein Junge erschossen. Sein Tod führt die Journalistin Cynthia Bonsant zu der gefeierten Internetplattform Freemee. Diese sammelt und analysiert Daten, und verspricht dadurch ihren Millionen Nutzern – zurecht – ein besseres Leben und mehr Erfolg. Nur einer warnt vor Freemee und vor der Macht, die der Online-Newcomer einigen wenigen verleihen könnte: ZERO, der meistgesuchte Online-Aktivist der Welt. Als Cynthia anfängt, genauer zu recherchieren, wird sie selbst zur Gejagten. Doch in einer Welt voller Kameras, Datenbrillen und Smartphones gibt es kein Entkommen ...
Hochaktuell und bedrohlich: Der gläserne Mensch unter Kontrolle. Der Roman spielt in der nahen Zukunft (near fiction) und enthält viele aktuelle Bezüge wie PRISM, Predictive Analytics, Gamification. Ganz nebenbei werden Verweise auf bekannte Science-Fiction-Medien wie „The Running Man“, „Monkey Wrench Gang“, „V wie Vendetta“ (V trägt eine Guy-Fawkes-Maske, mittlerweile das Markenzeichen von Anonymous), „Network“ und „Body Snatchers“ verarbeitet.
Technologisch-kryptologisch bewegen sich die Protagonisten auf dem höchsten Level, der aber nicht weiter erklärt wird: Alice Kinkaid kommuniziert mit einem Raspberry Pi. Cynthias Tochter Vi nutzt Mesh-Netze. Siehe
`https://de.wikipedia.org/wiki/Zero_%E2%80%93_Sie_wissen,_was_du_tust`,
`http://www.zero-das-buch.de/actiontrailer.php`

[**Takano2015**] Kazuaki Takano,

Extinction, C. Bertelsmann, 2015, 559 Seiten. (Orginal in Japanisch: „Jenosaido“, 2011; in Englisch unter dem Titel „Genocide of One“, 2014)

Jonathan Yeager wird im Auftrag der amerikanischen Regierung in den Kongo geschickt. Bei einem Pygmäenstamm sei ein tödliches Virus ausgebrochen. Die Verbreitung muss mit allen Mitteln verhindert werden. Doch im Dschungel erkennt Yeager, dass es um etwas ganz anderes geht: Ein kleiner Junge, der über unglaubliche Fähigkeiten und übermenschliche Intelligenz verfügt, ist das eigentliche Ziel der Operation. Ein sehr spannendes Buch: Wenn man die ersten 100-200 Seiten hinter sich hat, belohnt das überraschende Ende umso mehr. Aus den Rezensionen wird klar, dass es kein Buch für oberflächliche Leser ist: „Dieser Thriller liest sich sehr gut und ist zudem informationshaltig. Ein herausforderndes Buch. Sehr detailliert recherchiert.“ Das Buch schafft ein Szenario mit viel Realität: Auch die menschliche Evolution ist nicht zu Ende. Was wenn ein höherer Mensch auftaucht? Der Autor verbindet Abenteuer, fundierte und gut recherchierte Wissenschaft und Politik mit Globalität, politischen Verschwörungen, Kriegspsychologie, Psychologie der Mächtigen, Cyberterrorismus ... und leistet einen Appell an die Toleranz.

Aus kryptographischer Sicht: RSA und OTP kommen handlungsbestimmend und korrekt zum Einsatz. Das Brechen von RSA durch Faktorisierung ist so bedeutsam, dass der CIA nicht dulden kann, dass dieses Wissen nicht in seinen Händen liegt ...

Siehe `https://de.wikipedia.org/wiki/Extinction_(Roman)`.

[**Lagercrantz2015**] David Lagercrantz,

Verschwörung, Heyne, 2015.

Dies ist der vierte Band in der Millenium-Reihe, und der erste, der nicht von Stieg Larsson geschrieben wurde. Während Mikael Blomkvists Printmedium mit dem Überleben kämpft, erfährt man immer mehr von den inneren Strukturen und Verquickungen von Verlegern, Geheimdiensten, Behörden, organisierter Kriminalität und Wirtschaftsspionen. Auf einzelne Menschen wird dabei keine Rücksicht genommen und normale Menschen haben bei diesen Interessen auch keine Chance. Durch die besonderen Fähigkeiten von Lisbeth Salander wendet sich das Blatt und die NSA erfährt, dass Teile von ihr von der organisierten Kriminalität geführt und missbraucht werden. Die Millenium-Charaktere werden dabei glaubwürdig weiter entwickelt. Sehr spannend.

Aus kryptographischer Sicht: Lisbeth und der Autist August beschäftigen sich mit Elliptischen Kurven, um RSA zu knacken.

Anmerkung 1: Eine lange Liste, teilweise auch kommentierter Beispiele für Kryptologie in der belletristischen Literatur finden sich auf der folgenden Webseite:

`http://www.staff.uni-mainz.de/pommeren/Kryptologie/Klassisch/0_Unterhaltung/`

Für die ältere Literatur (z.B. von Jules Verne, Karl May, Arthur Conan Doyle, Edgar Allen Poe) sind darauf sogar die Links zu den eigentlichen Textstellen enthalten.

Anmerkung 2: Abgebildet und beschrieben finden Sie einige dieser Buchtitelseiten auf der Webseite von Tobias Schrödel, der sich vor allem mit antiken Büchern zur Kryptographie beschäftigt:

`http://tobiasschroedel.com/crypto_books.php`

Anmerkung 3: Wenn Sie weitere Literatur und Filme wissen, wo Kryptographie eine wesentliche Rolle spielt, dann würden wir uns sehr freuen, wenn Sie uns den genauen Titel und eine kurze Erläuterung zum Film-/Buchinhalt zusenden würden. Wir werden Ihre evtl. Begeisterung für einen Titel anklingen lassen. Herzlichen Dank.

A.5.2 Für Kinder und Jugendliche

Die folgende Auflistung enthält Filme und „Kinderbücher“. Die Kinderbücher enthalten nicht nur „Geschichten“, sondern auch Sammlungen von einfachen Verschlüsselungen, didaktisch und spannend aufbereitet:

[Mosesxxxx] [Ohne Autorenangabe],
Streng geheim – Das Buch für Detektive und Agenten, Edition moses, [ohne Angabe der Jahreszahl].
Ein dünnes Buch für kleinere Kinder mit Inspektor Fox und Dr. Chicken.

[Arthur196x] Robert Arthur,
Die 3 ???: Der geheime Schlüssel nach Alfred Hitchcock (Band 119), Kosmos-Verlag (ab 1960)
Darin müssen die drei Detektive Justus, Peter und Bob verdeckte und verschlüsselte Botschaften entschlüsseln, um herauszufinden, was es mit dem Spielzeug der Firma Kopperschmidt auf sich hat.

[Para1988] Para,
Geheimschriften, Ravensburger Taschenbuch Verlag, 1988 (erste Auflage 1977).
Auf 125 eng beschriebenen Seiten werden in dem klein-formatigen Buch viele verschiedene Verfahren erläutert, die Kinder selbst anwenden können, um ihre Nachrichten zu verschlüsseln oder unlesbar zu machen. Ein kleines Fachwörter-Verzeichnis und eine kurze Geschichte der Geheimschriften runden das Büchlein ab.

Gleich auf S. 6 steht in einem old-fashioned Style für den Anfänger „Das Wichtigste zuerst“ über Papier- und Bleistiftverfahren (vergleiche Kapitel 2): „Wollte man Goldene Regeln für Geheimschriften aufstellen, so könnten sie lauten:

- Deine geheimen Botschaften müssen sich an jedem Ort zu jeder Zeit mit einfachsten Mitteln bei geringstem Aufwand sofort anfertigen lassen.
- Deine Geheimschrift muss für Deine Partner leicht zu merken und zu lesen sein. Fremde hingegen sollen sie nicht entziffern können.
  Merke: Schnelligkeit geht vor Raffinesse, Sicherheit vor Sorglosigkeit.
- Deine Botschaft muss immer knapp und präzise sein wie ein Telegramm. Kürze geht vor Grammatik und Rechtschreibung. Alles Überflüssige weglassen (Anrede, Satzzeichen). Möglichst immer nur Groß- oder immer nur Kleinbuchstaben verwenden.“

[Müller-Michaelis2002] Matthias Müller-Michaelis,
Das Handbuch für Detektive. Alles über Geheimsprachen, Codes, Spurenlesen und die großen Detektive dieser Welt, Südwest-Verlag, 2002.
Broschiert, 96 Seiten.

[Kippenhahn2002] Rudolf Kippenhahn,
Streng geheim! – Wie man Botschaften verschlüsselt und Zahlencodes knackt, rororo, 2002.
In dieser Geschichte bringt ein Großvater, ein Geheimschriftexperte, seinen 4 Enkeln und deren Freunden bei, wie man Botschaften verschlüsselt, die niemand lesen soll. Da es

jemand gibt, der die Geheimnisse knackt, muss der Großvater den Kindern immer komplizierte Verfahren beibringen.
In dieser puren Rahmenhandlung werden die wichtigsten klassischen Verfahren und ihre Analyse kindgerecht und spannend erläutert.

[**Harder2003**] Corinna Harder und Jens Schumacher,
Streng geheim. Das große Buch der Detektive, Moses, 2003.
Gebundene Ausgabe: 118 Seiten.

[**Talke-Baisch2003**] Helga Talke und Milena Baisch,
Dein Auftrag in der unheimlichen Villa. Kennwort Rätselkrimi, Loewe, 2003.
Ab Klasse 4, `http://www.antolin.de`
Junge Rätseldetektive lösen bei ihren Einsätzen auch einfache Geheimsprachen und Codes.

[**Flessner2004**] Bernd Flessner,
Die 3 ???: Handbuch Geheimbotschaften, Kosmos-Verlag, 2004.
Auf 127 Seiten wird sehr verständlich, spannend, und strukturiert nach Verfahrenstyp geschildert, welche Geheimsprachen (Navajo-Indianer oder Dialekte) und welche Geheimschriften (echte Verschlüsselung, aber auch technische und linguistische Steganographie) es gab und wie man einfache Verfahren entschlüsseln kann.
Bei jedem Verfahren steht, wenn es in der Geschichte oder in Geschichten Verwendung fand [wie in Edgar Allan Poe's „Der Goldkäfer", wie von Jules Verne's Held Mathias Sandorf oder von Astrid Lindgren's Meisterdetektiv Kalle Blomquist, der die ROR-Sprache verwendet (ähnliche Einfüge-Chiffren sind auch die Löffel- oder die B-Sprache)].
Dieses Buch ist ein didaktisch hervorragender Einstieg für jüngere Jugendliche.

[**Zübert2005**] Regie Christian Zübert,
Der Schatz der weißen Falken, 2005.
Dieser spannende Kinder-Abenteuer-Film knüpft an die Tradition von Klassikern wie „Tom Sawyer und Huckleberry Finn" oder Enid Blytons „Fünf Freunde" an. Er spielt im Sommer des Jahres 1981. In einer halb verfallenen Villa finden 3 Jugendliche die Schatzkarte der „Weißen Falken", die sie mit Hilfe eines Computer entschlüsseln. Verfolgt von einer anderen Jugendbande machen sie sich auf zu einer alten Burgruine.

[**Dittert2011**] Christoph Dittert,
Die 3 ???: Geheimnisvolle Botschaften (Band 160), Kosmos-Verlag, 2011
Im Haus von Professor Mathewson wurde ein altes, handgefertigtes Buch gestohlen. Justus, Peter und Bob werden in ihren Ermittlungen von einem skrupellosen Gegner behindert, der ihnen stets einen Schritt voraus zu sein scheint. Eine große Rolle spielt ein Palimpsest, eine antike Manuskriptseite, die neu beschrieben wurde. Mit Röntgenstrahlen kann der alte Text darunter wieder sichtbar gemacht werden. Nicht nur die Geschichte ist spannend, sondern auch die Art, wie die Anleitung für die Schatzsuche verschlüsselt ist. Obwohl sie das einfache Gartenzaun-Verfahren nutzt, ist es nicht trivial, es zu lösen, denn die Botschaft ist auf zwei Zettel verteilt und die Druckzeichen stehen nicht für einzelne Buchstaben.

Anmerkung 1: Abgebildet und beschrieben finden Sie viele dieser Kinderbuch-Titelseiten auf der Webseite von Tobias Schrödel, der sich vor allem mit antiken Büchern zur Kryptographie beschäftigt:

`http://tobiasschroedel.com/crypto_books.php`

Anmerkung 2: Wenn Sie weitere Bücher kennen, die Kryptographie didaktisch kindergerecht aufbereiten, dann würden wir uns sehr freuen, wenn Sie uns den genauen Buchtitel und eine kurze Erläuterung zum Buchinhalt zusenden würden. Herzlichen Dank.

A.5.3 Code zur den Büchern der Unterhaltungsliteratur

Im Kapitel A.5.1 wurde als erstes E.A. Poes „Der Goldkäfer“ aufgeführt.

Mit dem folgenden Python-Code[10] kann man das Kryptogramm von Captain Kidd (Originaltext „The Gold Bug“ in `http://pinkmonkey.com/dl/library1/gold.pdf`, Seite 21) mit Python oder mit SageMath entschlüsseln.

Im Code sind die ASCII-Zeichen des Geheimtextes und die einander entsprechenden Alphabete dieses monoalphabetischen Verfahrens schon enthalten.

Am einfachsten kann man die Entschlüsselung mit dem SageMathCell-Server (`http://sagecell.sagemath.org/`) in einem Browser durchführen: Dort kann man zwischen den beiden Sprachen Sage und Python umschalten. Ausführen kann man den Code ganz leicht, indem man ihn mit „copy and paste“ einfügt und dann „Evaluate“ drückt.

SageMath-Beispiel A.1 Entschlüsselung des Gold-Bug-Geheimtextes in der Geschichte von E.A. Poe (mit Python)

```
# Hier geht's los ... Start here
import string
PA = 'ETHSONAIRDGLBVPFYMUC'
print 'Plaintext alphabet  PA: ', PA, '   Length of PA', len(PA)
CA = "8;4)+*56(!302'.1:9?-"
print 'Ciphertext alphabet CA: ', CA, '   Length of CA', len(CA)
codetableC2P = string.maketrans(CA,PA)
C = '''53++!305))6*;4826)4+.)4+);806*;48!8'60))85;1+(;:+*8!83(88)5*!;46
(;88*96*?;8)*+(;485);5*!2:*+(;4956*2(5*-4)8'8*;4069285);)6!8)4++;1(+9;4
8081;8:8+1;48!85;4)485!528806*81(+9;48;(88;4(+?34;48)4+;161;:188;+?;'''
P = string.translate(C, codetableC2P);
print 'Kidd decrypted:', P
# ... und hier sind wir schon fertig ... and here we are done!
```

Anmerkung 1: Pinkmonkey hat beim Drucken des Geheimtextes von Poe so ähnlich „geschummelt“ wie der Code-Autor, indem er nur ASCII-Zeichen verwendete.

In der Reproduktion einer Original-Publikation (z.B. unter `https://archive.org/details/goldbug00poegoog` auf Seite 95) kann man sehen, dass Poe Zeichen genommen hat, die im Buchdruck üblich waren, aber nur zum (allerdings größeren) Teil auch im ASCII-Zeichensatz vorkommen. Es ist zudem mehr als unwahrscheinlich, dass ein ungebildeter Seeräuber gerade diese Zeichen für sein Geheimtextalphabet verwenden würde ...

Anmerkung 2: Im Code werden die Python-Funktionen „maketrans“ und „translate“ aus dem String-Paket (siehe z.B. die Python 2.7-Dokumentation, Abschnitt „functions“) verwendet.

Hierbei gibt man das Klartext- und Geheimtext-Alphabet jeweils als einfache Strings ein und erzeugt daraus mit „maketrans“ eine Übersetzungstabelle. Die eigentliche Verschlüsselung erledigt dann „translate“. Für die Dechiffrierung müssen bei „maketrans“ nur die Argumente für die beiden Alphabete vertauscht werden. Die sonst üblichen Hin-und-Her-Umwandlungen zwischen Zeichen und ASCII-Nummern mittels „str“ und „ord“ entfallen. Das ist ideal für monoalphabetische Chiffrierungen – gerade im Unterricht in der Mittelstufe.

[10] Siehe [WLS98] und die Datei `2_MonoKidd.zip` unter `http://bscw.schule.de/pub/bscw.cgi/159132`.

Anmerkung 3: Hier ist ein Screenshot, der die Ausführung des Python-Codes auf dem SageMathCell-Server (`http://sagecell.sagemath.org/`) zeigt.

SageMathCell ist das Browser-Interface für Sage. Seine Funktionen werden auf dem Sage-Cell-Server kostenlos zur Verfügung gestellt.

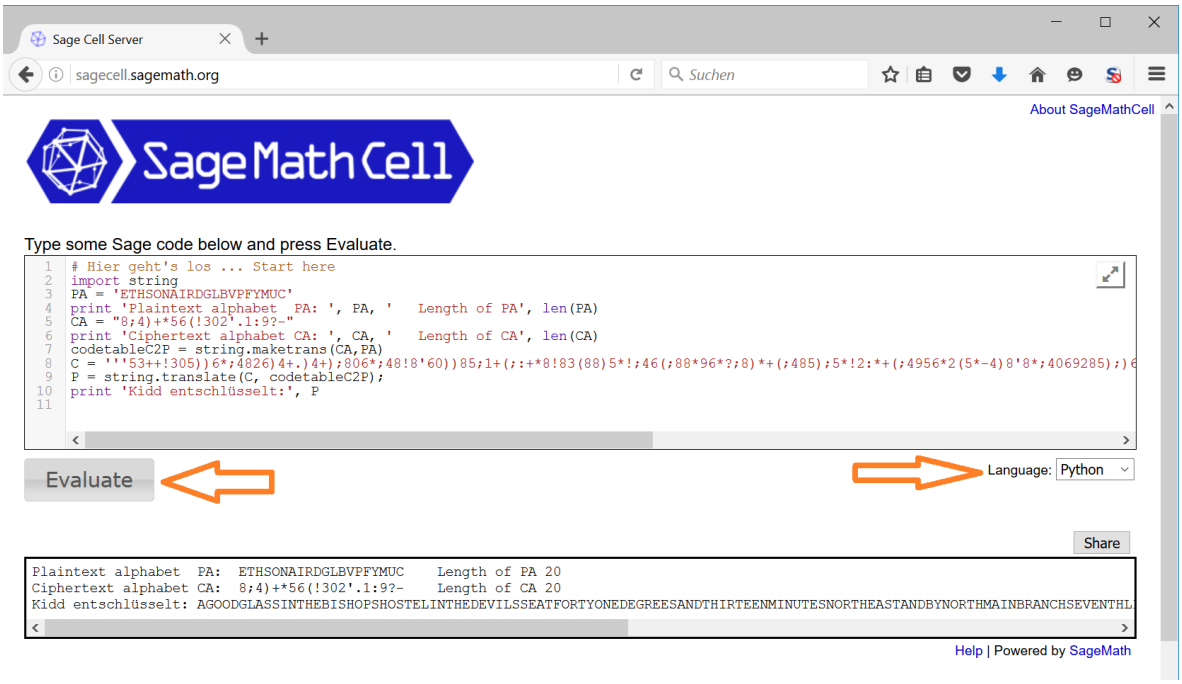


Abbildung A.8: Benutzung von SageMathCell, um Poes Goldkäfer zu entschlüsseln (mit Python)

Als Ergebnis erhalten wir:

```
Plaintext alphabet  PA:  ETHSONAIRDGLBVPFYMUC    Length of PA 20
Ciphertext alphabet CA:  8;4)+*56(!302'.1:9?-    Length of CA 20
Kidd decrypted: AGOODGLASSINTHEBISHOPSHOSTELINTHEDEVILSSEATFORTYONEDEGREES
                ANDTHIRTEENMINUTESNORTHEASTANDBYNORTHMAINBRANCHSEVENTHLIMB
                EASTSIDESHOOTFROMTHELEFTEYEOFTHEDEATHSHEADABEELINEFROMTHE
                TREETHROUGHTHESHOTFIFTYFEETOUT
```

Man sieht gut, wie wenig Code man mit Python oder Sage für solche Aufgaben benötigt. Im obigen Beispiel waren es 2 überflüssige Kommentarzeilen, 3 Zeilen Eingabe, 3 Zeilen echter Code und 3 Zeilen Ausgabe; effektiv nötig waren 7 Zeilen Code.

A.6 Lernprogramm Elementare Zahlentheorie

In CT1 ist ein interaktives Lernprogramm zur elementaren *Zahlentheorie*, genannt „ZT“, enthalten.[11]

Das Lernprogramm „NT“ (Zahlentheorie) von Martin Ramberger führt in die Zahlentheorie ein und visualisiert viele der Verfahren und Konzepte. Wo nötig zeigt es auch die entsprechenden mathematischen Formeln. Oftmals können die mathematischen Verfahren dynamisch mit eigenen kleinen Zahlenbeispielen ausprobiert werden.

Die Inhalte basieren vor allem auf den Büchern von [Buc16] und [Sch06].

Dieses visualisierte Lernprogramm wurde mit Authorware 4 erstellt.[12]

Bitte um Erweiterung/Upgrade: Ein Update auf die neueste Version von Authorware oder auf eine andere Entwicklungsplattform wäre wünschenswert. Wenn sich hierzu Interessenten finden, würde ich mich sehr freuen (bitte E-Mail an den Autor des CrypTool-Skriptes).

Abbildungen: Die Abbildungen A.9 bis A.16 vermitteln einen Eindruck des Lernprogramms „ZT“:

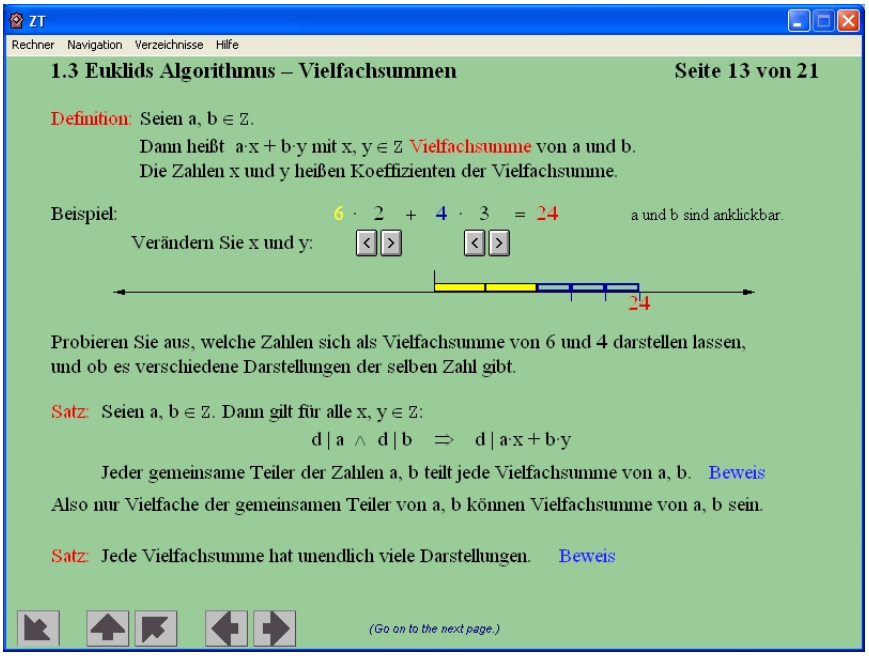


Abbildung A.9: Jeder gemeinsame Teiler zweier Zahlen teilt auch alle ihre Linearkombinationen

[11] ZT können Sie in CT1 über das Menü **Einzelverfahren \ Zahlentheorie interaktiv \ Lernprogramm für Zahlentheorie** aufrufen.

[12] Da Authorware veraltet ist und der Hersteller keine Portierungswerkzeuge auf seine Nachfolgeprodukte zur Verfügung stellte, wird das ZT-Programm nicht mehr weiter entwickelt.

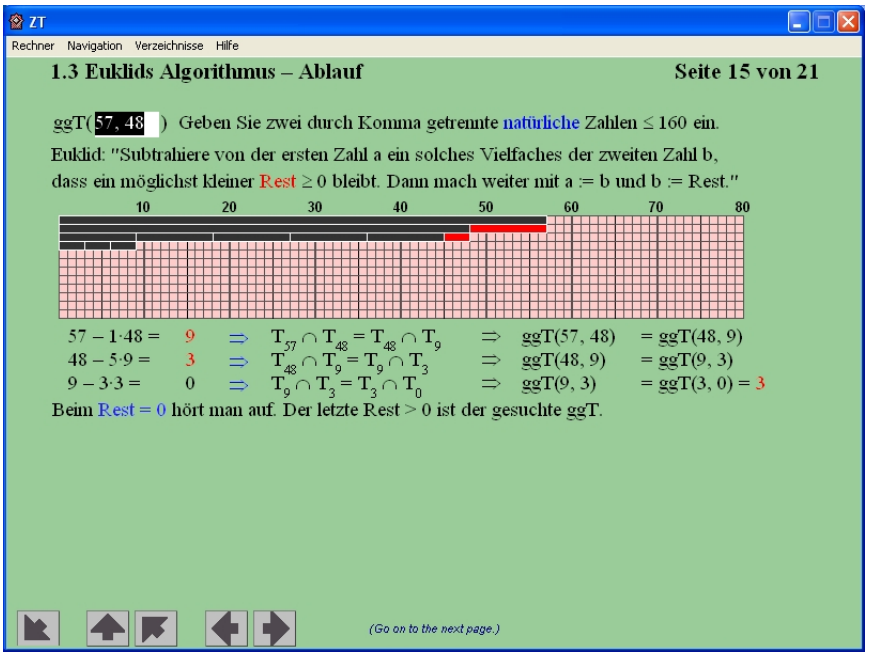


Abbildung A.10: Euklids Algorithmus zur Bestimmung des ggT

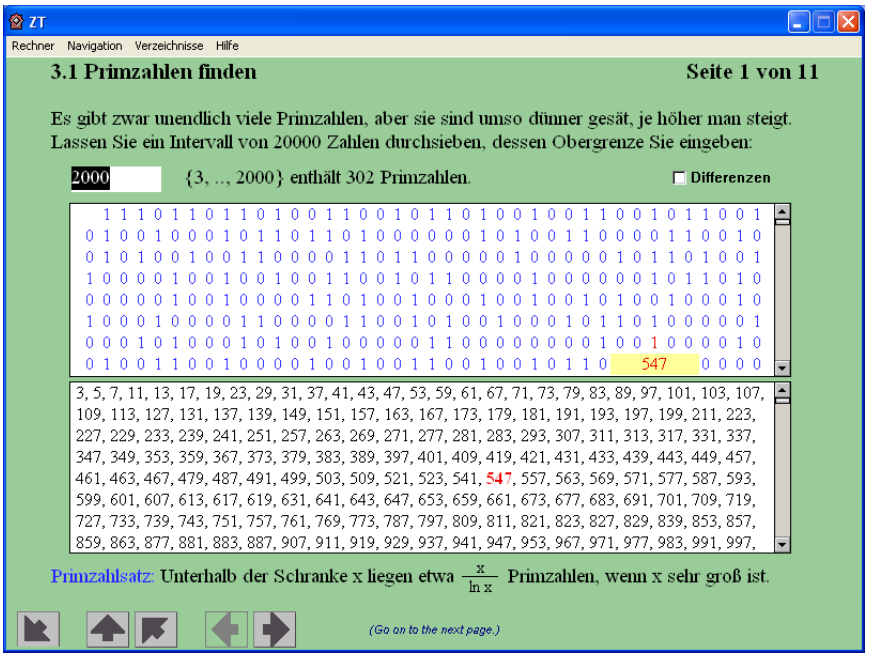


Abbildung A.11: Verteilung der Primzahlen und ihrer Differenzen

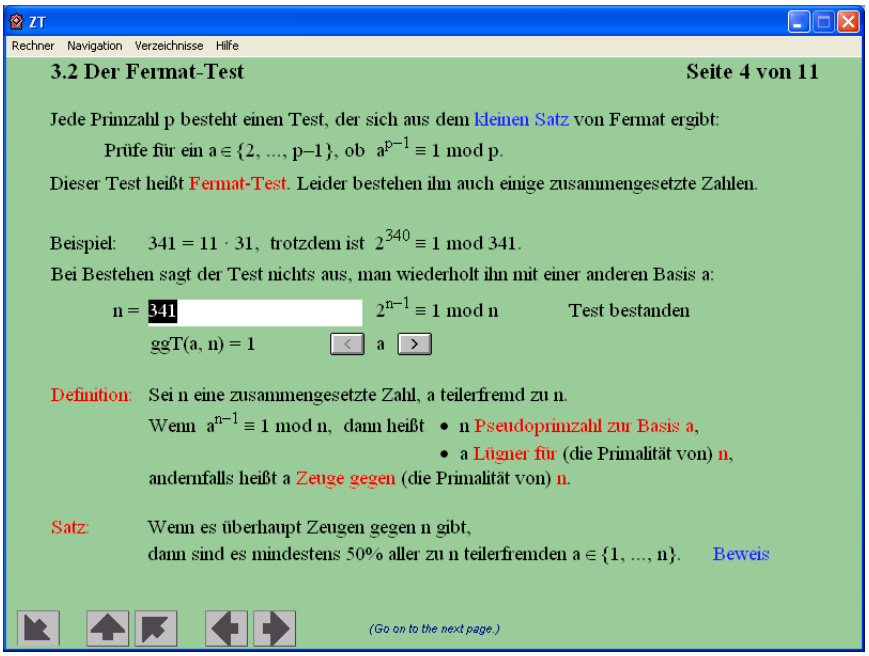


Abbildung A.12: Primzahlen finden mit dem Primzahltest nach Fermat

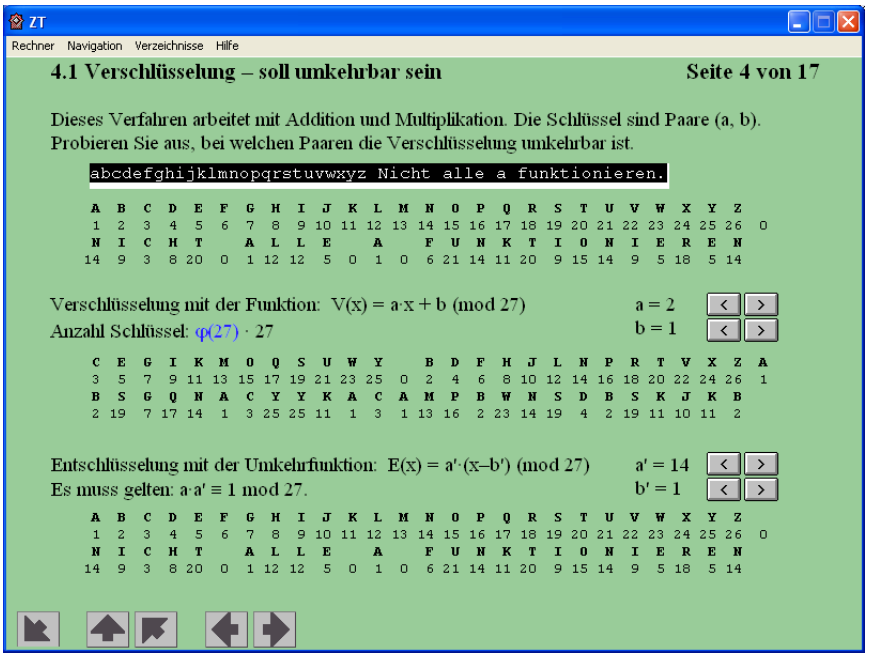


Abbildung A.13: Umkehrbarkeit von Verschlüsselungsalgoritmen am Beispiel additiver Chiffren

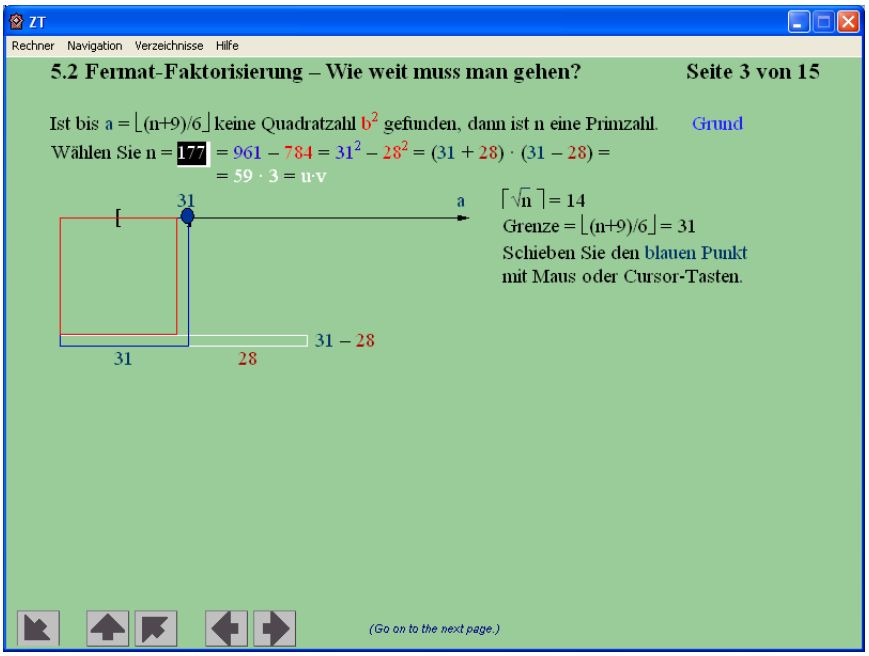


Abbildung A.14: Fermat-Faktorisierung m.H. der 3. Binomischen Formel

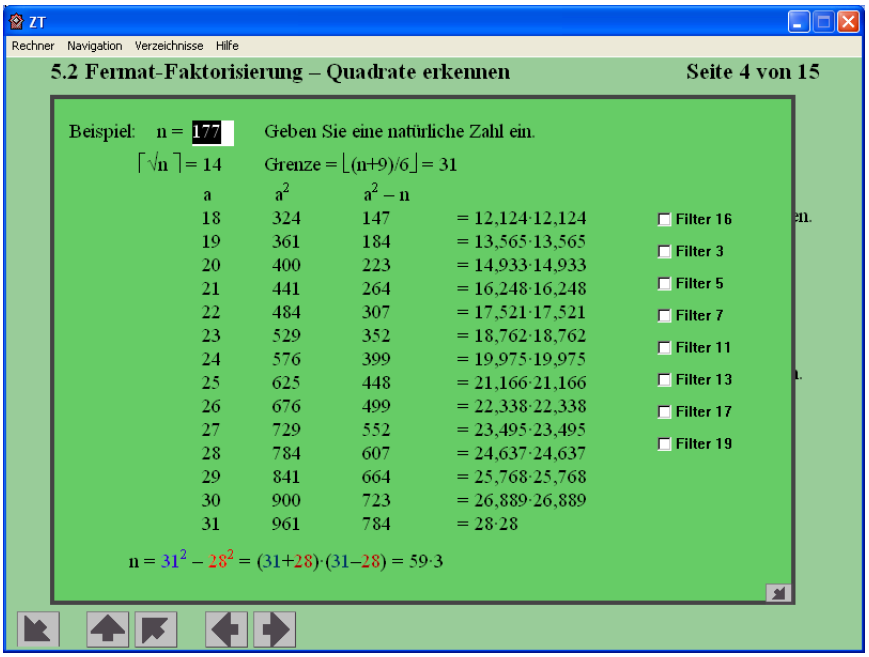


Abbildung A.15: Fermat-Faktorisierung: Quadrate erkennen

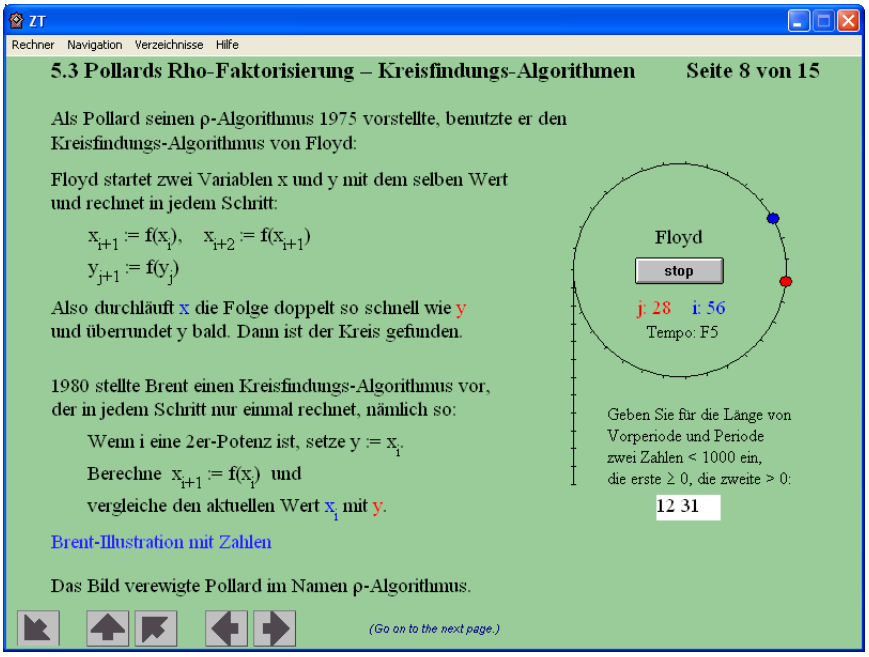


Abbildung A.16: Pollards Rho-Faktorisierung: Kreisfindungsalgorithmus nach Floyd

Literaturverzeichnis (Appendix_LearnTool)

[Buc16] Buchmann, Johannes: *Einführung in die Kryptographie*. Springer, 6. Auflage, 2016. Paperback.

[Kip97] Kippenhahn, Rudolf: *Verschlüsselte Botschaften: Geheimschrift, Enigma und Chipkarte*. rowohlt, 1. Auflage, 1997. New edition 2012, Paperback, *Verschlüsselte Botschaften: Geheimschrift, Enigma und digitale Codes*.

[Sch06] Scheid, Harald: *Zahlentheorie*. Spektrum Akademischer Verlag, 4. Auflage, 2006.

[WLS98] Witten, Helmut, Irmgard Letzner und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle, Teil 1: Sprache und Statistik*. LOG IN, 1998(3/4):57–65, 1998.
`http://bscw.schule.de/pub/bscw.cgi/d637160/RSA_u_Co_T1.pdf`.

A.7 Kurzeinführung in das CAS SageMath

Dieses Buch enthält zahlreiche mit SageMath erstellte Programmbeispiele. SageMath ist ein Open-Source Computer-Algebra-System (CAS), das für Lehre, Studium und Forschung eingesetzt wird. SageMath kombiniert viele hochwertige Open-Source-Packete[13] und liefert den Zugang zu deren Funktionalität über ein gemeinsames, auf der Programmiersprache Python basierendes Interface[14].

SageMath kann man auf vielfältige Weise nutzen: als mächtigen Taschenrechner; als Tool für das Mathematikstudium; oder als Programmier-Umgebung, um Algorithmen zu prototypen oder um Forschung im Bereich der algorithmischen Aspekte der Mathematik zu betreiben.

Einen schnellen Einstieg bieten z.B. die Referenzen in dieser Fußnote[15].

Die offizielle SageMath Online-Dokumentation[16] finden Sie unter: `http://www.sagemath.org`.

Es gibt inzwischen viele PDF- und HTML-Dokumente über SageMath, so dass wir als guten Startpunkt nur einige wenige nennen[17].

Auch beim Studium der Kryptologie können fertige SageMath-Module genutzt werden[18].

Umfangreiche Kryptographie-Einführungen finden sich in der folgenden Fußnote[19].

SageMath-Benutzerschnittstellen

SageMath ist kostenlos und kann von folgender Webseite herunter geladen werden:

`http://www.sagemath.org`

[13] Einen Eindruck von der Größe von SageMath erhält man, wenn man es selbst compiliert: Die heruntergeladenen Sourcen von SageMath 4.1 brauchten zur Compilierung auf einem durchschnittlichen Linux-PC rund 5 h (inklusive aller Bibliotheken). Danach nahm es 1,8 GB Plattenplatz ein.

[14] Es gibt auch ein relativ einfaches Interface für die Sprache C, genannt Cython, mit der man eigene Funktionen in SageMath stark beschleunigen kann.
Siehe `http://openwetware.org/wiki/Open_writing_projects/Sage_and_cython_a_brief_introduction`.

[15] - „Einladung zu Sage“ von David Joyner, letztes Update 2009,
`http://sage.math.washington.edu/home/wdj/teaching/calc1-sage/an-invitation-to-sage.pdf`
- „The SDSU Sage Tutorial“,
`http://www-rohan.sdsu.edu/~mosulliv/sagetutorial/`
`http://www-rohan.sdsu.edu/~mosulliv/sagetutorial/sagecalc.html`
- „SAGE For Newbies“ von Ted Kosan, 2007,
`http://sage.math.washington.edu/home/tkosan/newbies_book/sage_for_newbies_v1.23.pdf`

[16] Die entsprechenden offiziellen PDF-Dokuments können herunter geladen werden von
`http://www.sagemath.org/help.html`, `http://www.sagemath.org/doc` und `http://planet.sagemath.org`.

[17] - „Bibliothek“: `http://www.sagemath.org/library/index.html`,
- „Dokumentationsprojekt“: `http://wiki.sagemath.org/DocumentationProject`,
- „Lehrmaterial“: `http://wiki.sagemath.org/Teaching_with_SAGE`.

[18] - Sourcen der Module im Verzeichnis `SAGE_ROOT/devel/sage-main/sage/crypto`.
- Überblick, welche Kryptographie momentan in SageMath enthalten ist:
`http://www.sagemath.org/doc/reference/sage/crypto/`
- Diskussionen über Lernaspekte beim Entwickeln weiterer Krypto-Module in SageMath:
`http://groups.google.com/group/sage-devel/browse_thread/thread/c5572c4d8d42d081`

[19] - Ein fertiger Kryptographie-Kurs von David Kohel, der SageMath nutzt, aus 2008:
`http://www.sagemath.org/library/crypto.pdf`
bzw. derselbe Kurs in einer eventuell neueren Fassung
`http://sage.math.washington.edu/home/wdj/teaching/kohel-crypto.pdf` .
- „Introduction to Cryptography with Open-Source Software“, ein hervorragendes Buch von Alasdair McAndrew, CRC, 2011

Standardmäßig nutzt man die SageMath-**Kommandozeile** als Interface, wie im folgenden Bild A.17 zu sehen ist. Es gibt jedoch auch ein grafisches Benutzerinterface für diese Software in Form des SageMath-Notebooks (siehe Bild A.18). Und schließlich kann man SageMath-**Notebooks**[20] auch online auf verschiedenen Servern nutzen, ohne SageMath lokal zu installieren, z.B.:

`http://www.sagenb.org` oder
`http://sage.mathematik.uni-siegen.de:8000`

SageMath läuft unter den Betriebssystemen Linux, Mac OS X und Windows. Auf der Windows-Plattform läuft die komplette SageMath-Distribution momentan nur als ein VMware-Image.

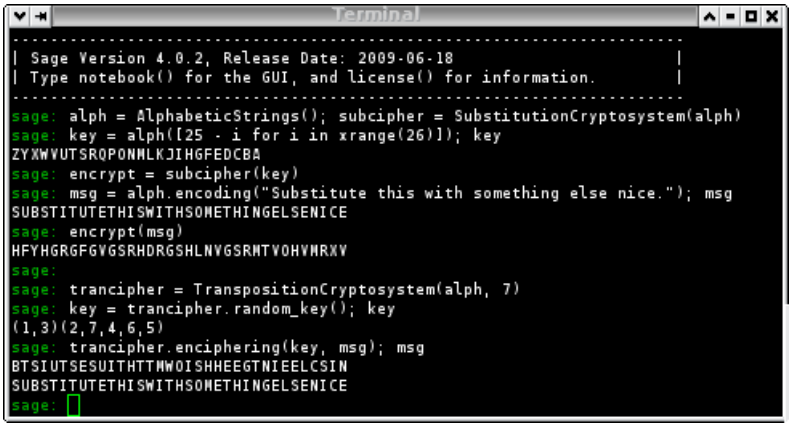


Abbildung A.17: SageMath-Kommandozeilen-Interface

[20] Weitere Details zu SageMath-Notebooks finden Sie in Kapitel 7.9.2 („Implementierung Elliptischer Kurven zu Lehrzwecken“⇒ „SageMath“).

[21] Um das grafische SageMath-Interface lokal zu starten, muss man auf der SageMath-Kommandozeile `notebook()` eingeben. Danach startet der eingestellte Browser (Iceweasel, Firefox, IE, ...) z.B. mit der URL `http://localhost:8000`.

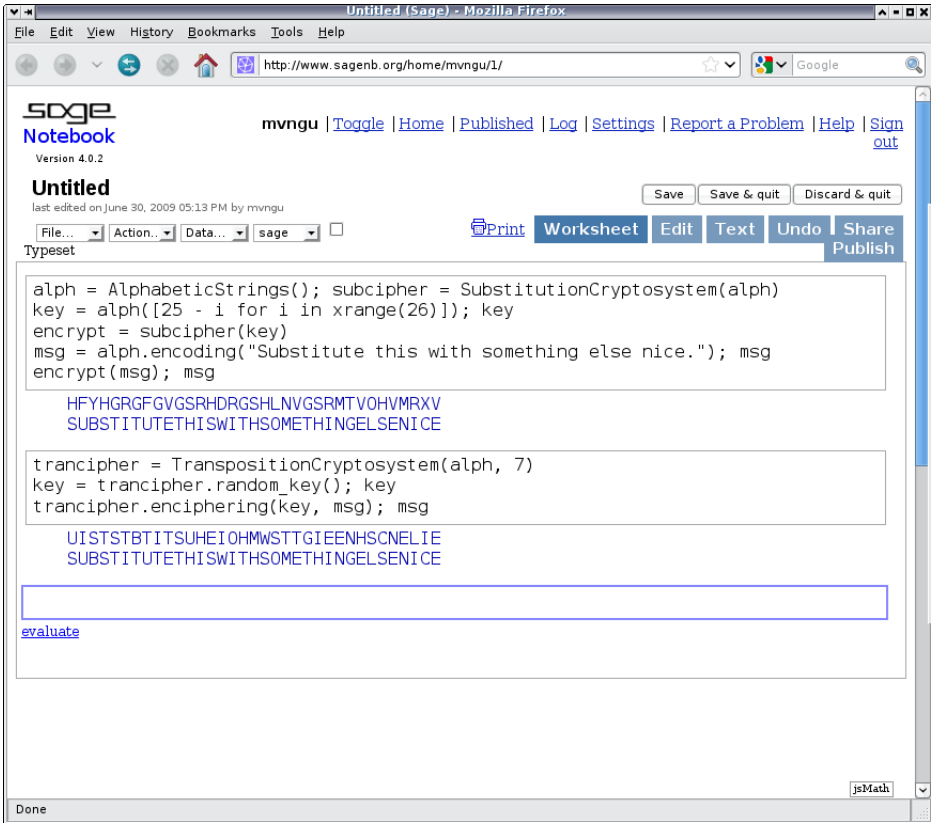


Abbildung A.18: SageMath-Notebook-Interface[21]

Hilfe beim Benutzen von SageMath

Wenn man SageMath auf der Kommandozeile startet, erhält etwas wie die folgenden Zeilen:

```
mnemonic:~$ sage
----------------------------------------------------------------------
| Sage Version 4.1, Release Date: 2009-07-09                         |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

sage: help
Type help() for interactive help, or help(object) for help about object.
sage:
sage:
sage: help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
```

```
such as "spam", type "modules spam".
```

Viele weitere Hilfen gibt es als offizielle SageMath-Dokumentation, die mit jedem Release von SageMath verteilt wird (siehe Bild A.19). Zur offiziellen SageMath-Standard-Dokumentation

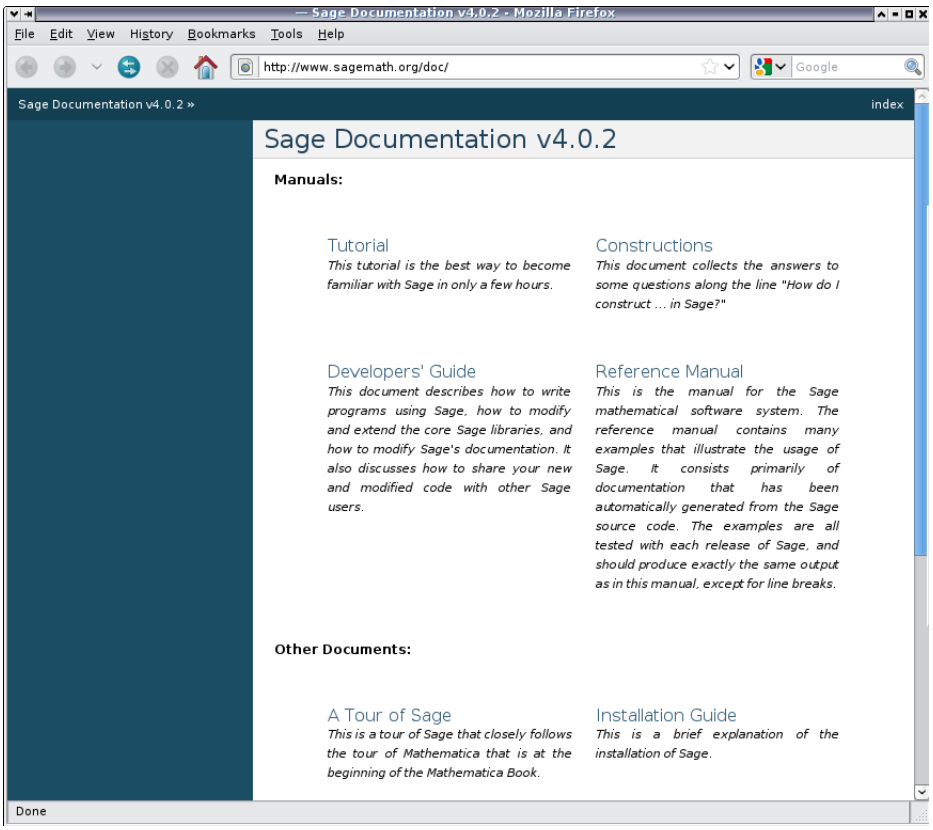


Abbildung A.19: Die Standard-Dokumentation von SageMath

gehören folgende Dokumente:

- Tutorial — Das Tutorial ist für SageMath-Einsteiger. Es ist dafür gedacht, sich in ein bis drei Stunden mit den wichtigsten Funktionen vertraut zu machen.
- Constructions — Dieses Dokument ist im Stil eines „Kochbuchs“ mit einer Sammlung von Antworten auf Fragen zur Konstruktion von SageMath-Objekten.
- Developers' Guide — Dieser Führer ist für Entwickler, die selbst SageMath mit weiter entwickeln wollen. Enthalten sind darin z.B. Hinweise zum Stil und zu Konventionen beim Programmieren, zur Modifikation von SageMath-Kern-Bibliotheken oder von SageMath-Standard-Dokumentation, und zum Code-Review und zur Software-Verteilung.
- Reference Manual — Dieses Handbuch enthält die komplette Dokumentation aller wichtigen SageMath-Funktionen. Zu jeder Klassen-Beschreibung gibt es mehrere Code-Beispiele. Alle Code-Beispiele im Referenz-Handbuch werden bei jedem neuen SageMath-Release getestet.
- Installation Guide — Dieser Führer erklärt, wie man SageMath auf verschiedenen Plattformen installiert.

- A Tour of Sage — Diese Tour durch SageMath zeigt exemplarisch verschiedene Funktionen, die für Einsteiger sinnvoll sind.
- Numerical Sage — Dieses Dokument führt Werkzeuge auf, die in SageMath für numerische Mathematik verfügbar sind.
- Three Lectures about Explicit Methods in Number Theory Using Sage — Drei Vorlesungen über Methoden der Zahlentheorie, die explizit SageMath nutzen. Dieses Dokument zeigt wie man mit SageMath Berechnungen in fortgeschrittener Zahlentheorie durchführt.

Von der SageMath-Kommandozeile erhält man eine Liste aller verfügbaren Kommandos (Funktionsnamen etc.), die ein bestimmtes Muster haben, wenn man die ersten Zeichen tippt, und dann die „Tab“-Taste drückt:

```
sage: Su[TAB]
Subsets                    Subwords                   SuzukiGroup
SubstitutionCryptosystem   SupersingularModule
```

Wenn man den genauen Namen eines Kommandos kennt, kann man die `help`-Funktion nutzen oder das Fragezeichen „?“ anfügen, um weitere Informationen zu diesem Kommando zu erhalten. Zum Beispiel liefert das Kommando `help(SubstitutionCryptosystem)` die Dokumentation zur der eingebauten Klasse `SubstitutionCryptosystem`. Mit dem Fragezeichen erhalten wir die Dokumentation zu dieser Klasse auf folgende Weise:

```
sage: SubstitutionCryptosystem?
Type:type
Base Class:<type 'type'>
String Form:<class 'sage.crypto.classical.SubstitutionCryptosystem'>
Namespace:Interactive
File:/home/mvngu/usr/bin/sage-3.4.1/local/lib/python2.5/site-packages/sage/crypto/classical.py
Docstring:

        Create a substitution cryptosystem.

        INPUT:

        - ``S`` - a string monoid over some alphabet

        OUTPUT:

        - A substitution cryptosystem over the alphabet ``S``.

        EXAMPLES::

            sage: M = AlphabeticStrings()
            sage: E = SubstitutionCryptosystem(M)
            sage: E
            Substitution cryptosystem on Free alphabetic string monoid
            on A-Z
            sage: K = M([ 25-i for i in range(26) ])
            sage: K
            ZYXWVUTSRQPONMLKJIHGFEDCBA
            sage: e = E(K)
            sage: m = M(''THECATINTHEHAT'')
            sage: e(m)
            GSVXZGRMGSVSZG
```

```
TESTS::

    sage: M = AlphabeticStrings()
    sage: E = SubstitutionCryptosystem(M)
    sage: E == loads(dumps(E))
    True
```

Weitere Unterstützung für spezifische Probleme gibt es in den Archiven der `sage-support` Mailing-Liste unter

`http://groups.google.com/group/sage-support`

Beispiele für in SageMath eingebaute mathematische Funktionen

Hier sind ein paar kleine Beispiele[22] (alle für das Kommandozeilen-Interface – zur einfacheren Nutzung), um zu sehen, was man mit SageMath machen kann:

SageMath-Beispiel A.2 Einige kleine Beispiele in SageMath aus verschiedenen Gebieten der Mathematik

```
# * Analysis (Infinitesimalrechnung):
    sage: x=var('x')
    sage: p=diff(exp(x^2),x,10)*exp(-x^2)
    sage: p.simplify_exp()
     1024 x^10 + 23040 x^8 + 161280 x^6 + 403200 x^4 + 302400 x^2 + 30240

# * Lineare Algebra:
    sage: M=matrix([[1,2,3],[4,5,6],[7,8,10]])
    sage: c=random_matrix(ZZ,3,1);c
     [ 7 ]
     [-2 ]
     [-2 ]
    sage: b=M*c
    sage: M^-1*b
     [ 7 ]
     [-2 ]
     [-2 ]

# * Zahlentheorie:
    sage: p=next_prime(randint(2^49,2^50));p
      1022095718672689
    sage: r=primitive_root(p);r
      7
    sage: pl=log(mod(10^15,p),r);pl
      1004868498084144
    sage: mod(r,p)^pl
      1000000000000000

# * Endliche Körper (\url{http://de.wikipedia.org/wiki/Endlicher_K%C3%B6rper}):
    sage: F.<x>=GF(2)[]
    sage: G.<a>=GF(2^4,name='a',modulus=x^4+x+1)
    sage: a^2/(a^2+1)
      a^3 + a
    sage: a^100
      a^2 + a + 1
    sage: log(a^2,a^3+1)
      13
    sage: (a^3+1)^13
      a^2
```

[22] Diese Beispiele stammen aus dem Blog von Dr. Alasdair McAndrew, Victoria University, http://amca01.wordpress.com/2008/12/19/sage-an-open-source-mathematics-software-system

Programmieren mit SageMath

Wenn man ein CAS (Computer-Algebra-System) nutzt, schreibt man zu Beginn einzelne Befehle in die Kommandozeile wie im obigen Beispiel[23].

Wenn man eigene Funktionen entwickelt, sie ändert und aufruft, dann ist es viel einfacher, die Entwicklung in einem eigenen Editor vorzunehmen, den Code als SageMath-Skriptdatei zu speichern und die Funktionen nicht-interaktiv auf der Kommandozeile auszuführen. Beide Arten, Code zu entwickeln, wurden in Kapitel 1.8 („Anhang: Beispiele mit SageMath“), Kapitel 2.5 („Anhang: Beispiele mit SageMath“), Kapitel 3.14 („Anhang: Beispiele mit SageMath“) und in Kapitel 4.19 („Anhang: Beispiele mit SageMath“) angewandt.

Um SageMath-Code in einem eigenen Editor zu entwickeln und zu testen, gibt es zwei nützliche Befehle: `load()` und `attach()`[24].
Angenommen Sie haben die folgende Funktions-Definition:

```
def function(var1):
    r"""
    DocText.
    """
    ...
    return (L)
```

die in der Datei `primroots.sage` gespeichert wurde.

Um diese Funktion in SageMath zu laden (und syntaktisch gleich zu testen), wird der Befehl `load()` benutzt:

```
sage: load primroots.sage
```

Danach kann man auf der Kommandozeile alle Variablen und Funktionen nutzen, die im SageMath-Skript definiert wurden[25].

Normalerweise editiert man ein eigenes SageMath-Skript wieder und möchte dann den Inhalt des geänderten Skripts wieder in SageMath laden. Dafür kann man den Befehl `attach()` nutzen (man kann auch direkt nach dem `load()` das `attach()` aufrufen, und nicht erst, wenn man das

[23] Standardmäßig wird SageMath-Code auch so präsentiert: Dabei beginnen die Zeilen mit „sage:“ und „...“.

```
sage: m = 11
sage: for a in xrange(1, m):
....:     print [power_mod(a, i, m) for i in xrange(1, m)]
....:
```

Auch dieses Skript benutzt normalerweise die obige Konvention, um SageMath-Code zu präsentieren, solange der Code nicht aus einer SageMath-Skriptdatei kommt. Wenn man den SageMath-Code aus diesem Skript kopiert und per Paste auf der SageMath-Kommandozeile wieder einfügt, sollte man „sage:“ und „...“ weglassen (obwohl das Kommandozeilen-Interface in den meisten Fällen mit diesen Präfixen korrekt umgehen kann).

[24] Vergleiche das SageMath-Tutorial über Programmierung, Kapitel „Loading and Attaching Sage files“, `http://www.sagemath.org/doc/tutorial/programming.html#loading-and-attaching-sage-files`.

[25] Anmerkungen:
- Bitte keine Leerzeichen oder White Spaces im Dateinamen.
- Es empfiehlt sich, der SageMath-Skriptdatei die Datei-Extension „.sage“ statt „.py“ zu geben. Hat ein SageMath-Skript die Dateinamens-Endung „.sage“, dann wird beim Laden der Datei in SageMath auch gleich die normale SageMath-Umgebung mit geladen, um die Syntax zu prüfen. Genauso funktioniert es, wenn man ein SageMath-Skript direkt von einer Bash-Shell aufruft mit `$ sage primroots.sage`.
- Beim Laden des obigen SageMath-Skripts wird es von SageMath zuerst geparst, und dann in eine andere Datei namens „primroots.py“ kopiert. SageMath ergänzt dann alle notwendigen Variablen in „primroots.py“ und alle Import-Statements. Somit wird das SageMath-Skript genauso ausgeführt, als hätte man die Befehle einzeln auf der Kommandozeile eingetippt. Ein bedeutender Unterschied ist, dass alle Ausgaben ein `print` benötigen.

Skript ändert; man kann `load()` sogar weglassen, da dies in `attach()` enthalten ist):

```
sage: attach primroots.sage
```

Nun kann man das SageMath-Skript ändern und die geänderte Funktionsdefinition wird – solange man die SageMath-Session nicht beendet – beim nächsten Enter in SageMath geladen (und syntaktisch gleich geprüft). Diese Neuladen passiert vollkommen automatisch. Der Befehl attach() lässt SageMath also permanent die genannte Datei auf Änderungen überwachen. Damit spart man sich das Kopieren und Pasten zwischen dem eigenen Texteditor und dem SageMath-Kommandozeilen-Interface.

Hier ist ein Bild, das SageMath-Code im Editor GVIM zeigt – mit aktiviertem Syntax-Highlighting (siehe Bild A.20).

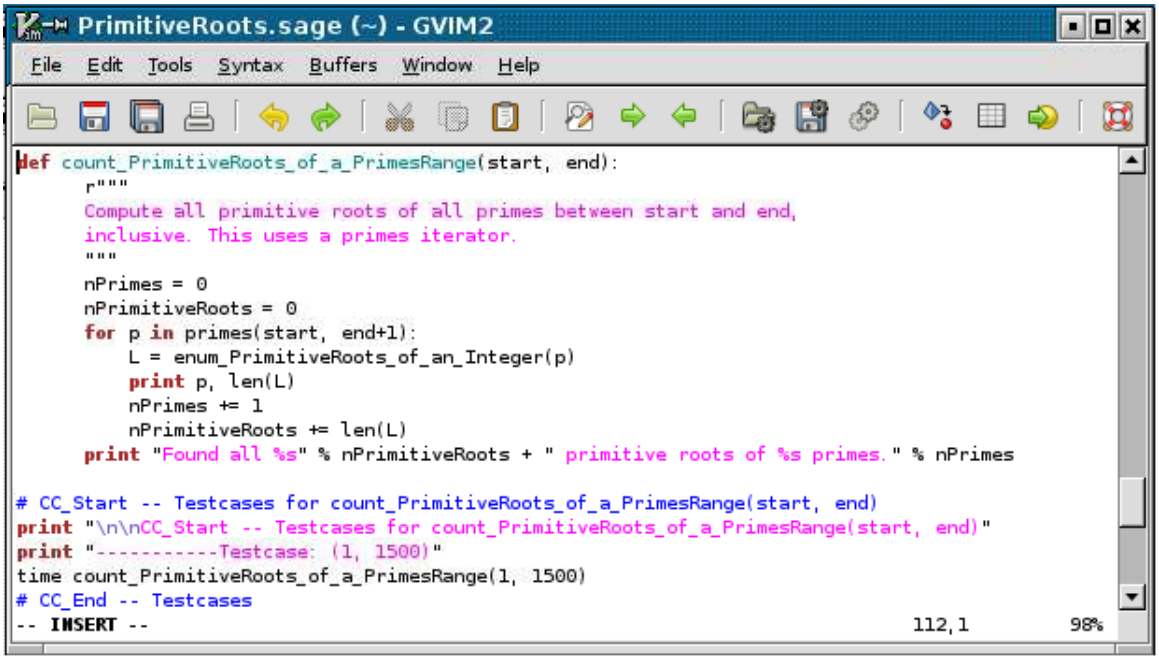


Abbildung A.20: SageMath-Beispiel in einem Editor mit aktiviertem Code-Highlighting

Falls man die Ausgabe einer attachten Datei so angezeigt haben möchte, wie wenn man die Einzelbefehle direkt auf der Kommandozeile eingibt (also nicht nur das, was per `print` ausgegeben wird), kann man den Befehl `iload()` verwenden: Jede Zeile wird dann einzeln geladen. Um die nächste Zeile zu laden, muss man die `Enter`-Taste drücken. Das muss man so lange wiederholen, bis alle Zeilen des SageMath-Skripts in die SageMath-Session geladen sind.

```
sage: iload primroots.sage
```

Weitere Hinweise:

- Abfrage der Version Ihrer SageMath-Umgebung mit: `version()`
- Um sich schnell die SageMath-Programmbeispiele in diesem Skript anzusehen, können Sie
  - im Index nach `SageMath -> Programmbeispiele` schauen, oder
  - sich im Anhang das „Verzeichnis der SageMath-Programmbeispiele“ ansehen.
- Die SageMath-Beispiele in diesem Skript werden mit CrypTool ausgeliefert.
  Weitere Details am Ende der Übersicht „Verzeichnis der SageMath-Programmbeispiele“.

A.8 Autoren des CrypTool-Buchs

Dieser Anhang führt die Autoren dieses Dokuments auf.
Die Autoren sind namentlich am Anfang jedes Kapitels aufgeführt, zu dem sie beigetragen haben.

Bernhard Esslinger
Initiator des CrypTool-Projekts, Editor und Hauptautor dieses Buchs. Professor für IT-Sicherheit und Kryptologie an der Universität Siegen. Ehemals: CISO der SAP AG, und Leiter IT-Sicherheit und Leiter Crypto Competence Center bei der Deutschen Bank.
E-Mail: bernhard.esslinger@gmail.com, bernhard.esslinger@uni-siegen.de

---

Matthias Büger
Mitautor des Kapitels 7 („Elliptische Kurven“), Research Analyst bei der Deutschen Bank.

Bartol Filipovic
Ursprünglicher Autor der Elliptische-Kurven-Implementierung in CT1 und des entsprechenden Kapitels in diesem Buch.

Martin Franz
Autor des Kapitels 9 („Homomorphe Chiffren“). Forscht und arbeitet im Bereich der angewandten Kryptographie.

Henrik Koy
Hauptentwickler und Koordinator der CT1-Entwicklung der Versionen 1.3 und 1.4, Reviewer des Buchs und TEX-Guru, Projektleiter IT und Kryptologe bei der Deutschen Bank.

Roger Oyono
Implementierer des Faktorisierungs-Dialogs in CT1 und ursprünglicher Autor des Kapitels 5 („Die mathematischen Ideen hinter der modernen Kryptographie“).

Klaus Pommerening
Autor des Kapitels 8 („Einführung in die Bitblock- und Bitstrom-Verschlüsselung“), Professor für Mathematik und Medizinische Informatik an der Johannes-Gutenberg-Universität. Im Ruhestand.

Jörg Cornelius Schneider
Design und Long-term-Support von CrypTool, Kryptographie-Enthusiast, IT-Architekt und Senior-Projektleiter IT bei der Deutschen Bank.

Christine Stötzel
Diplom Wirtschaftsinformatikerin an der Universität Siegen.

Johannes Buchmann
Mitautor des Kapitels 11 („Krypto 2020 — Perspektiven für langfristige kryptographische Sicherheit“), Vizepräsident der TU Darmstadt (TUD) und Professor an den Fachbereichen für Informatik und Mathematik der TUD. Dort hat er den Lehrstuhl für Theoretische Informatik (Kryptographie und Computer-Algebra) inne.

Alexander May
Mitautor des Kapitels 11 („Krypto 2020 — Perspektiven für langfristige kryptographische Sicherheit“) und des Kapitel 10 (“Resultate zur Widerstandskraft diskreter Logarithmen und zur Faktorisierung”). Ordentlicher Professor am Fachbereich Mathematik (Lehrstuhl für Kryptologie und IT-Sicherheit) der Ruhr-Universität Bochum, und zur Zeit (2014) Leiter des Horst-Görtz Instituts für IT-Sicherheit. Sein Forschungsschwerpunkt liegt bei Algorithmen für die Kryptoanalyse, insbesondere auf Methoden für Angriffe auf das RSA-Kryptoverfahren.

Erik Dahmen
Mitautor des Kapitels 11 („Krypto 2020 — Perspektiven für langfristige kryptographische Sicherheit“), Mitarbeiter am Lehrstuhl Theoretische Informatik (Kryptographie und Computeralgebra) der TU Darmstadt.

Ulrich Vollmer
Mitautor des Kapitels 11 („Krypto 2020 — Perspektiven für langfristige kryptographische Sicherheit“), Mitarbeiter am Lehrstuhl Theoretische Informatik (Kryptographie und Computeralgebra) der TU Darmstadt.

Antoine Joux
Mitautor des Kapitels 10 (“Resultate zur Widerstandskraft diskreter Logarithmen und zur Faktorisierung”). Antoine Joux ist der Inhaber des Cryptology Chair der Stiftung der Universität Pierre et Marie Curie (Paris 6) und ein Senior Sicherheitsexperte bei CryptoExperts, Paris. Er arbeitete in verschiedenen Gebieten der Kryptanalyse und ist die Schlüsselfigur bei den aktuellen Fortschritten in der Berechnung diskreter Logarithmen in Körpern mit kleiner Charakteristik.

Arjen Lenstra
Mitautor des Kapitels 10 (“Resultate zur Widerstandskraft diskreter Logarithmen und zur Faktorisierung”). Arjen Lenstra ist ordentlicher Professor an der École Polytechnique Fédérale de Lausanne (EPFL) und Leiter der Forschungsabteilung für kryptologische Algorithmen. Er ist einer der Erfinder des derzeit besten Algorithmus für ganzzahlige Faktorisierung, des Zahlkörpersiebs (Number Field Sieve). Außerdem war er an vielen praktischen Faktorisierungsrekorden beteiligt.

Minh Van Nguyen
SageMath-Entwickler und Reviewer des Dokuments.

GNU Free Documentation License

Version 1.3, 3 November 2008

`http://fsf.org/`

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to

the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Krypto-Verfahren mit Pseudocode

Verzeichnis der Zitate

Verzeichnis der OpenSSL-Beispiele

Verzeichnis der SageMath-Programmbeispiele

- Der Quellcode der SageMath-Beispiele in diesem Skript wird in Form von SageMath-Programmdateien mit dem CT1-Setup-Programm ausgeliefert. Die Beispiele in einem Kapitel sind jeweils in einer Datei zusammen gefasst.
  Nach der Installation von CrypTool 1 finden Sie die SageMath-Beispiele im Unterverzeichnis `sagemath` in den folgenden 4 Dateien:

  - SageMath-Samples-in-Chap01.sage
  - SageMath-Samples-in-Chap02.sage
  - SageMath-Samples-in-Chap03.sage
  - SageMath-Samples-in-Chap04.sage

- Alle Beispiel wurden mit der SageMath-Version 5.3 (Release Date 2012-09-08) getestet.

Literaturverzeichnis über alle Kapitel (nummeriert by babplain)

[1] Aaronson, Scott: *The Prime Facts: From Euclid to AKS*, 2003. `http://www.scottaaronson.com/writings/prime.pdf`.

[2] ACA: *Length and Standards for all ACA Ciphers*. Technischer Bericht, American Cryptogram Association, 2002. `http://www.cryptogram.org/cdb/aca.info/aca.and.you/chap08.html#`, `http://www.und.edu/org/crypto/crypto/.chap08.html`.

[3] Adleman, L.: *On breaking the iterated Merkle-Hellman public-key Cryptosystem*. In: *Advances in Cryptologie, Proceedings of Crypto 82*, Seiten 303–308. Plenum Press, 1983.

[4] Adleman, Leonard M.: *A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography (Abstract)*. In: *FOCS*, Seiten 55–60, 1979.

[5] Agence nationale de la sécurité des systèmes d'information: *Référentiel général de sécurité Version 2.02*, 2013. `http://www.ssi.gouv.fr/administration/reglementation/`.

[6] Agrawal, M., N. Kayal und N. Saxena: *PRIMES in P*, August 2002. Corrected version. `http://www.cse.iitk.ac.in/~manindra/algebra/primality_v6.pdf`, `http://fatphil.org/maths/AKS/`.

[7] Bach, Eric: *Discrete Logarithms and Factoring*. Technischer Bericht UCB/CSD-84-186, EECS Department, University of California, Berkeley, Juni 1984. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5973.html`, `http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-186.pdf`.

[8] Balcazar, J. L., J. Daaz und J. Gabarr: *Structural Complexity I*. Springer, 1998.

[9] Barbulescu, Razvan, Pierrick Gaudry, Antoine Joux und Emmanuel Thomé: *A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic*. CoRR, abs/1306.4244, 2013.

[10] Bard, Gregory V.: *Algebraic Cryptanalysis*. Springer, Dordrecht, 2009.

[11] Bartholomé, A., J. Rung und H. Kern: *Zahlentheorie für Einsteiger*. Vieweg, 2. Auflage, 1996.

[12] Bauer, Friedrich L.: *Entzifferte Geheimnisse*. Springer, 1995.

[13] Bauer, Friedrich L.: *Decrypted Secrets*. Springer, 2. Auflage, 2000.

[14] Bennett, Charles H. und Gilles Brassard: *An Update on Quantum Cryptography.* In: Blakley, G. R. und David Chaum (Herausgeber): *Advances in Cryptology – CRYPTO '84*, Band 196 der Reihe *Lecture Notes in Computer Science*, Seiten 475–480. Springer-Verlag, 1985.

[15] Bernstein, Daniel und Tanja Lange: *SafeCurves: choosing safe curves for elliptic-curve cryptography.* `http://safecurves.cr.yp.to/`, 2014.

[16] Bernstein, Daniel J.: *Circuits for integer factorization: a proposal.* `http://cr.yp.to/papers/nfscircuit.ps`, `http://cr.yp.to/djb.html`, 2001.

[17] Bernstein, Daniel J.: *Factoring into coprimes in essentially linear time.* Journal of Algorithms, 54, 2005. `http://cr.yp.to/lineartime/dcba-20040404.pdf`.

[18] Beutelspacher, Albrecht: *Kryptologie.* Vieweg, 5. Auflage, 1996.

[19] Beutelspacher, Albrecht, Jörg Schwenk und Klaus Dieter Wolfenstetter: *Moderne Verfahren in der Kryptographie.* Vieweg, 2. Auflage, 1998.

[20] Biryukov, Alex und Dmitry Khovratovich: *Related-key Cryptanalysis of the Full AES-192 and AES-256.* Cryptology ePrint Archive, 2009. `http://eprint.iacr.org/2009/317`.

[21] Blum, W.: *Die Grammatik der Logik.* dtv, 1999.

[22] Blömer, J.: *Vorlesungsskript Algorithmische Zahlentheorie*, 1999. Ruhr-University Bochum. `http://www.math.uni-frankfurt.de/~dmst/teaching/lecture_notes/bloemer.algorithmische_zahlentheorie.ps.gz`.

[23] Bos, Joppe W., Craig Costello, Patrick Longa und Michael Naehrig: *Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis*, 2014. Microsoft Research. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/selecting.pdf`.

[24] Bourseau, F., D. Fox und C. Thiel: *Vorzüge und Grenzen des RSA-Verfahrens.* Datenschutz und Datensicherheit (DuD), 26:84–89, 2002. `http://www.secorvo.de/publikationen/rsa-grenzen-fox-2002.pdf`.

[25] Brands, Gilbert: *Verschlüsselungsalgorithmen – Angewandte Zahlentheorie rund um Sicherheitsprotokolle.* Vieweg, 2002.

[26] Brickell, E. F.: *Breaking Iterated Knapsacks.* In: *Advances in Cryptology: Proc. CRYPTO'84, Lecture Notes in Computer Science, vol. 196*, Seiten 342–358. Springer, 1985.

[27] Brickenstein, Michael: *Boolean Gröbner Bases – Theory, Algorithms and Applications.* Dissertation, TU Kaiserslautern, department of Mathematics, 2010. See also "PolyBoRi – Polynomials over Boolean Rings", online: `http://polybori.sourceforge.net/`".

[28] BSI: *Angaben des BSI für die Algorithmenkataloge der Vorjahre, Empfehlungen zur Wahl der Schlüssellängen.* Technischer Bericht, BSI (Bundesamt für Sicherheit in der Informationstechnik), 2016.

https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/ElektronischeSignatur/TechnischeRealisierung/Kryptoalgorithmen/kryptoalg.html
- Vgl.: BNetzA (Bundesnetzagentur): Jährlich erscheinende Veröffentlichung zu Algorithmen und Parametern im Umfeld elektronischer Signaturen:
http://www.bundesnetzagentur.de/cln_1411/DE/Service-Funktionen/ElektronischeVertrauensdienste/QES/WelcheAufgabenhatdieBundesnetzagentur/GeeigneteAlgorithmenfestlegen/geeignetealgorithmenfestlegen_node.html
- Vgl.: Eine Stellungnahme zu diesen Empfehlungen:
http://www.secorvo.de/publikationen/stellungnahme-algorithmenempfehlung-020307.pdf .

[29] BSI (Bundesamt für Sicherheit in der Informationstechnik): *BSI TR-02102 Kryptographische Verfahren: Empfehlungen und Schlüssellängen*, 2012.
https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm.

[30] Buchmann, Johannes: *Einführung in die Kryptographie*. Springer, 6. Auflage, 2016. Paperback.

[31] Buchmann, Johannes, Luis Carlos Coronado García, Erik Dahmen, Martin Döring und Elena Klintsevich: *CMSS – an improved Merkle signature scheme*. In: Barua, Rana und Tanja Lange (Herausgeber): *7th International Conference on Cryptology in India - Indocrypt'06*, Nummer 4392 in *Lecture Notes in Computer Science*, Seiten 349–363. Springer-Verlag, 2006.

[32] Buhler, J. P., H. W. Lenstra und C. Pomerance: *Factoring integers with the number field sieve*. In: Lenstra, K. und H.W. Lenstra (Herausgeber): *The Development of the Number Field Sieve, Lecture Notes in Mathematics, Vol. 1554*, Seiten 50–94. Springer, 1993.

[33] Bundschuh, Peter: *Einführung in die Zahlentheorie*. Springer, 4. Auflage, 1998.

[34] Cassels, J. W. S.: *Lectures on Elliptic Curves*. Cambridge University Press, 1991.

[35] Castro, D., M. Giusti, J. Heintz, G. Matera und L. M. Pardo: *The hardness of polynomial equation solving*. Found. Comput. Math., 3:347–420, 2003.

[36] Coppersmith, Don: *Evaluating Logarithms in $GF(2^n)$*. In: *STOC*, Seiten 201–207, 1984.

[37] Coppersmith, Don: *Re: Impact of Courtois and Pieprzyk results*. Journal unknown, 2002.
http://csrc.nist.gov/archive/aes/ Former link from the AES Discussion Groups.

[38] Coppersmith, Don, Andrew M. Odlyzko und Richard Schroeppel: *Discrete Logarithms in GF(p)*. Algorithmica, 1(1):1–15, 1986.

[39] Costello, Craig, Patrick Longa und Michael Naehrig: *A brief discussion on selecting new elliptic curves*, 2015. Microsoft Research.
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/NIST.pdf.

[40] Courtois, Nicolas und Josef Pieprzyk: *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations*. Cryptology ePrint Archive, 2002.
A different, so called compact version of the first XSL attack, was published in the proceedings for Asiacrypt Dec 2002. http://eprint.iacr.org/2002/044.

[41] Cox, David, John Little und Donal O'Shea: *Ideals, Varieties, and Algorithms*. Springer, 3. Auflage, 2007.

[42] Crama, Yves und Peter L. Hammer (Herausgeber): *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*. Cambridge University Press, 2010.

[43] Crandell, Richard und Carl Pomerance: *Prime Numbers. A Computational Perspective*. Springer, 2001.

[44] Crowley, Paul: *Mirdek: A card cipher inspired by "Solitaire"*, 2000. `http://www.ciphergoth.org/crypto/mirdek/`.

[45] Cusick, Thomas W. und Pantelimon Stănică: *Cryptographic Boolean Functions and Applications*. Elsevier Academic Press, 2009.

[46] Daemen, Joan und Vincent Rijmen: *The Design of Rijndael. AES – The Advanced Encryption Standard*. Springer, 2002.

[47] Doxiadis, Apostolos: *Onkel Petros und die Goldbachsche Vermutung*. Lübbe, 2001.

[48] Drobick, Jörg: *Abriss DDR-Chiffriergeschichte: SAS- und Chiffrierdienst*, 2015. `http://scz.bplaced.net/m.html#dwa`.

[49] Eckert, Claudia: *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. De Gruyter Oldenbourg, 9. Auflage, 2014. Paperback.

[50] Enge, Andreas, Pierrick Gaudry und Emmanuel Thomé: *An L(1/3) Discrete Logarithm Algorithm for Low Degree Curves*. J. Cryptology, 24(1):24–41, 2011.

[51] Ertel, Wolfgang: *Angewandte Kryptographie*. Fachbuchverlag Leipzig FV, 2001.

[52] Esslinger, B., J. Schneider und V. Simon: *RSA – Sicherheit in der Praxis*. KES Zeitschrift für Informationssicherheit, 2012(2):22–27, April 2012.
`https://www.cryptool.org/images/ctp/documents/kes_2012_RSA_Sicherheit.pdf`.

[53] Esslinger, B., J. Schneider und V. Simon: *Krypto + NSA = ? – Kryptografische Folgerungen aus der NSA-Affäre*. KES Zeitschrift für Informationssicherheit, 2014(1):70–77, März 2014.
`https://www.cryptool.org/images/ctp/documents/krypto_nsa.pdf`.

[54] Ferguson, Niels, Richard Schroeppel und Doug Whiting: *A simple algebraic representation of Rijndael*, 2001.
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.4921`.

[55] Fouque, Pierre Alain, Antoine Joux und Chrysanthi Mavromati: *Multi-user collisions: Applications to Discrete Logarithm, Even-Mansour and Prince*. Cryptology ePrint Archive, 2014. `https://eprint.iacr.org/2013/761`.

[56] Galbraith, Steven D.: *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012, ISBN 9781107013926.

[57] Garey, Michael R. und David S. Johnson: *Computers and Intractability*. Freeman, 1979.

[58] Gathen, Joachim von zur und Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 1999.

[59] Gaudry, Pierrick: *Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem.* J. Symb. Comput., 44(12):1690–1702, 2009.

[60] Gaudry, Pierrick, Florian Hess und Nigel P. Smart: *Constructive and Destructive Facets of Weil Descent on Elliptic Curves.* J. Cryptology, 15(1):19–46, 2002.

[61] Gentry, Craig: *Fully Homomorphic Encryption Using Ideal Lattices.* In: *41st ACM Symposium on Theory of Computing (STOC)*, 2009.

[62] Goebel, Greg: *Codes, Ciphers and Codebreaking*, 2014. Version 2.3.2. `http://www.vectorsite.net/ttcode.html`.

[63] Goebel, Greg: *A Codes & Ciphers Primer*, 2015. Version 1.0.5. `http://www.vectorsite.net/ttcode.html`.

[64] Göloglu, Faruk, Robert Granger, Gary McGuire und Jens Zumbrägel: *On the Function Field Sieve and the Impact of Higher Splitting Probabilities – Application to Discrete Logarithms.* In: *CRYPTO (2)*, Seiten 109–128, 2013.

[65] Golomb, Solomon W.: *Shift Register Sequences.* Aegean Park Press, 1982. Revised Edition.

[66] Graham, R. E., D. E. Knuth und O. Patashnik: *Concrete Mathemathics, a Foundation of Computer Science.* Addison Wesley, 6. Auflage, 1994.

[67] Haan, Kristian Laurent: *Advanced Encryption Standard (AES)*, 2008. `http://www.codeplanet.eu/tutorials/cpp/51-advanced-encryption-standard.html`.

[68] Heninger, Nadia, Zakir Durumeric, Eric Wustrow und J. Alex Halderman: *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.* In: *Proceedings of the 21st USENIX Security Symposium*, August 2012. `https://factorable.net/paper.html`.

[69] Hesselink, Wim H.: *The borderline between P and NP*, Februar 2001. `http://www.cs.rug.nl/~wim/pub/whh237.pdf`.

[70] Hill, Lester S.: *Cryptography in an Algebraic Alphabet.* The American Mathematical Monthly, 36(6):306–312, 1929.

[71] Hill, Lester S.: *Concerning Certain Linear Transformation Apparatus of Cryptography.* The American Mathematical Monthly, 38(3):135–154, 1931.

[72] Hoffman, Nick: *A SIMPLIFIED IDEA ALGORITHM*, 2006. `http://www.nku.edu/~christensen/simplified%20IDEA%20algorithm.pdf`.

[73] Homeister, Matthias: *Quantum Computer Science: An Introduction.* Vieweg+Teubner Verlag, 2007, ISBN 9780521876582.

[74] ITU-T: *X.509 (1993) Amendment 1: Certificate Extensions, The Directory Authentication Framework.* Technischer Bericht, International Telecommunication Union ITU-T, Juli 1995. (equivalent to amendment 1 to ISO/IEC 9594-8).

[75] ITU-T: *ITU-T Recommendation X.509 (1997 E): Information Technology – Open Systems Interconnection – The Directory: Authentication Framework.* Technischer Bericht, International Telecommunication Union ITU-T, Juni 1997.

[76] Joux, Antoine: *Algorithmic Cryptanalysis.* CRC Cryptography and Network Security Series. Chapman & Hall, 2009, ISBN 1420070029.

[77] Joux, Antoine: *A new index calculus algorithm with complexity L(1/4+o(1)) in very small characteristic.* IACR Cryptology ePrint Archive, 2013:95, 2013.

[78] Joux, Antoine: *Faster Index Calculus for the Medium Prime Case Application to 1175-bit and 1425-bit Finite Fields.* In: *EUROCRYPT*, Seiten 177–193, 2013.

[79] Joux, Antoine und Reynald Lercier: *The Function Field Sieve in the Medium Prime Case.* In: *EUROCRYPT*, Seiten 254–270, 2006.

[80] Joux, Antoine und Vanessa Vitse: *Cover and Decomposition Index Calculus on Elliptic Curves made practical. Application to a seemingly secure curve over F_{p^6}.* IACR Cryptology ePrint Archive, 2011:20, 2011.

[81] Katzenbeisser, Stefan: *Recent Advances in RSA Cryptography.* Springer, 2001.

[82] Kippenhahn, Rudolf: *Verschlüsselte Botschaften: Geheimschrift, Enigma und Chipkarte.* rowohlt, 1. Auflage, 1997. New edition 2012, Paperback, *Verschlüsselte Botschaften: Geheimschrift, Enigma und digitale Codes.*

[83] Kippenhahn, Rudolph: *Code Breaking – A History and Exploration.* Constable, 1999.

[84] Klee, V. und S. Wagon: *Ungelöste Probleme in der Zahlentheorie und der Geometrie der Ebene.* Birkhäuser Verlag, 1997.

[85] Kleinjung, Thorsten, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev und Paul Zimmermann: *Factorization of a 768-Bit RSA Modulus.* In: *CRYPTO*, Seiten 333–350, 2010.

[86] Kleinjung, Thorsten et al.: *Factorization of a 768-bit RSA modulus, version 1.4*, 2010. `http://eprint.iacr.org/2010/006.pdf`.

[87] Knuth, Donald E.: *The Art of Computer Programming, vol 2: Seminumerical Algorithms.* Addison-Wesley, 3. Auflage, 1998.

[88] Koblitz, N.: *Introduction to Elliptic Curves and Modular Forms.* Graduate Texts in Mathemathics, Springer, 1984.

[89] Koblitz, N.: *Algebraic Aspects of Cryptography. With an appendix on Hyperelleptic Curves by Alfred J. Menezes, Yi Hong Wu, and Robert J. Zuccherato.* Springer, 1998.

[90] Labs, RSA: *PKCS #1 v2.1 Draft 3: RSA Cryptography Standard.* Technischer Bericht, RSA Laboratories, April 2002.

[91] Lagarias, J. C.: *Knapsack public key Cryptosystems and diophantine Approximation.* In: *Advances in Cryptology, Proceedings of Crypto 83.* Plenum Press, 1983.

[92] Lazard, Daniel: *Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations.* In: *Lecture Notes in Computer Science 162*, Seiten 146–156. Springer, 1983. EUROCAL '83.

[93] Lenstra, A. und H. Lenstra: *The development of the Number Field Sieve.* Lecture Notes in Mathematics 1554. Springer, 1993.

[94] Lenstra, A. K. und H. W. Jr. Lenstra: *The Development of the Number Field Sieve*. Lecture Notes in Mathematics. Springer Verlag, 1993, ISBN 0387570136.

[95] Lenstra, Arjen K., James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung und Christophe Wachter: *Public Keys*. In: *CRYPTO*, Seiten 626–642, 2012.

[96] Lenstra, Arjen K., James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung und Christophe Wachter: *Ron was wrong, Whit is right, A Sanity Check of Public Keys Collected on the Web*. Cryptology ePrint Archive, Februar 2012.
`http://eprint.iacr.org/2012/064.pdf`.

[97] Lenstra, Arjen K., Adi Shamir, Jim Tomlinson und Eran Tromer: *Analysis of Bernstein's Factorization Circuit*, 2002.
`http://tau.ac.il/~tromer/papers/meshc.pdf`.

[98] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes*. In: *Lecture Notes in Computer Science 558*, Seiten 446–465, 2000. PKC2000.
See also "BlueKrypt Cryptographic Key Length Recommendation", last update 2015, online: `http://www.keylength.com/en/2/`.

[99] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes (1999 + 2001)*. Journal of Cryptology, 14:255–293, 2001.
`http://www.cs.ru.nl/E.Verheul/papers/Joc2001/joc2001.pdf`,
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.69&rep=rep1&type=pdf`.

[100] Lenstra, H. W.: *Factoring Integers with Elliptic Curves*. Annals of Mathematics, 126:649–673, 1987.

[101] Lochter, Manfred und Johannes Merkle: *ECC Brainpool Standard Curves and Curve Generation v. 1.0*, 2005.
`www.ecc-brainpool.org/download/Domain-parameters.pdf`.

[102] Lochter, Manfred und Johannes Merkle: *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, 2010. RFC 5639.
`http://www.rfc-base.org/txt/rfc-5639.txt`.

[103] Lochter, Manfred und Johannes Merkle: *Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)*, 2013. RFC 7027.
`http://tools.ietf.org/search/rfc7027`.

[104] Lorenz, F.: *Algebraische Zahlentheorie*. BI Wissenschaftsverlag, 1993.

[105] Lucks, Stefan und Rüdiger Weis: *Neue Ergebnisse zur Sicherheit des Verschlüsselungsstandards AES*. DuD, Dezember 2002.

[106] Mansoori, S. Davod und H. Khaleghei Bizaki: *On the vulnerability of Simplified AES Algorithm Against Linear Cryptanalysis*. IJCSNS International Journal of Computer Science and Network Security, 7(7):257–263, 2007.
`http://paper.ijcsns.org/07_book/200707/20070735.pdf`.

[107] Maseberg, Jan Sönke: *Fail-Safe-Konzept für Public-Key-Infrastrukturen*. Dissertation, TU Darmstadt, 2002.

[108] Massacci, Fabio und Laura Marraro: *Logical Cryptanalysis as a SAT Problem: Encoding and Analysis*. Journal of Automated Reasoning Security, 24:165–203, 2000.

[109] May, Alexander: *Vorlesungsskript Kryptanalyse 1*, 2008. Ruhr-University Bochum. http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/pkk08/skript.pdf.

[110] May, Alexander: *Vorlesungsskript Kryptanalyse 2*, 2012. Ruhr-University Bochum. http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/12/ws1213/kryptanal12/kryptanalyse_2013.pdf.

[111] May, Alexander: *Vorlesungsskript Zahlentheorie*, 2013. Ruhr-University Bochum. http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/13/ss13/zahlenss13/zahlentheorie.pdf.

[112] McDonald, Cameron, Philip Hawkes und Josef Pieprzyk: *Differential Path for SHA-1 with complexity* $O(2^{52})$. Cryptology ePrint Archive, 2012. http://eprint.iacr.org/2009/259.

[113] McEliece, Robert J.: *A public key cryptosystem based on algebraic coding theory*. DSN progress report, 42–44:114–116, 1978.

[114] Meier, W. und O. Staffelbach: *Fast correlation attacks on certain stream ciphers*. Journal of Cryptology, 1:159–176, 1989.

[115] Menezes, A. J.: *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[116] Menezes, Alfred J., Paul C. van Oorschot und Scott A. Vanstone: *Handbook of Applied Cryptography*. Series on Discrete Mathematics and Its Application. CRC Press, 5. Auflage, 2001, ISBN 0-8493-8523-7. (Errata last update Jan 22, 2014).
http://cacr.uwaterloo.ca/hac/,
http://www.cacr.math.uwaterloo.ca/hac/.

[117] Merkle, R. und M. Hellman: *Hiding information and signatures in trapdoor knapsacks*. IEEE Trans. Information Theory, IT-24, 24, 1978.

[118] Merkle, Ralph C.: *Secrecy, authentication, and public key systems*. Dissertation, Department of Electrical Engineering, Stanford University, 1979.

[119] Mermin, David N.: *Quantum Computing verstehen*. Cambridge University Press, 2008, ISBN 3834804363.

[120] Mironov, Ilya und Lintao Zhang: *Applications of SAT Solvers to Cryptanalysis of Hash Functions*. Springer, 2006.

[121] Murphy, S. P. und M. J. B. Robshaw: *Comments on the Security of the AES and the XSL Technique*, September 2002. http://crypto.rd.francetelecom.com/people/Robshaw/rijndael/rijndael.html.

[122] Murphy, S. P. und M. J. B. Robshaw: *Essential Algebraic Structure within the AES*. Technischer Bericht, Crypto 2002, 2002. http://crypto.rd.francetelecom.com/people/Robshaw/rijndael/rijndael.html.

[123] Musa, Mohammad A., Edward F. Schaefer und Stephen Wedig: *A simplified AES algorithm and its linear and differential cryptanalyses.* Cryptologia, 17(2):148–177, April 2003.
http://www.rose-hulman.edu/~holden/Preprints/s-aes.pdf,
http://math.scu.edu/eschaefe/ Ed Schaefer's homepage.

[124] Müller-Stach und Piontkowski: *Elementare und Algebraische Zahlentheorie.* Vieweg Studium, 2011, ISBN 3834882631.

[125] National Institute of Standards and Technology (NIST): *Federal Information Processing Standards Publication 197: Advanced Encyption Standard*, 2002.

[126] Nguyen, Minh Van: *Exploring Cryptography Using the Sage Computer Algebra System.* Diplomarbeit, Victoria University, 2009.
http://www.sagemath.org/files/thesis/nguyen-thesis-2009.pdf,
http://www.sagemath.org/library-publications.html.

[127] Nguyen, Minh Van: *Number Theory and the RSA Public Key Cryptosystem – An introductory tutorial on using SageMath to study elementary number theory and public key cryptography*, 2009. http://faculty.washington.edu/moishe/hanoiex/Number%20Theory%20Applications/numtheory-crypto.pdf.

[128] Nichols, Randall K.: *Classical Cryptography Course, Volume 1 and 2.* Technischer Bericht, Aegean Park Press 1996, 1996. 12 lessons.
www.apprendre-en-ligne.net/crypto/bibliotheque/lanaki/lesson1.htm.

[129] NIST: *Entity authentication using public key cryptography.* Technischer Bericht, NIST (U.S. Department of Commerce), 1997. No more valid (withdrawal date: October 2015).

[130] NIST: *Digital Signature Standard (DSS).* Technischer Bericht, NIST (U.S. Department of Commerce), 2013. Change note 4.
http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf,
http://csrc.nist.gov/publications/PubsFIPS.html.

[131] NIST: *Secure Hash Standard (SHS).* Technischer Bericht, NIST (U.S. Department of Commerce), August 2015. FIPS 180-4 supersedes FIPS 180-2.
http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf,
http://csrc.nist.gov/publications/PubsFIPS.html.

[132] Oechslin, Philippe: *Making a Faster Cryptanalytic Time-Memory Trade-Off.* Technischer Bericht, Crypto 2003, 2003.
http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf.

[133] Oppliger, Rolf: *Contemporary Cryptography, Second Edition.* Artech House, 2. Auflage, 2011. http://books.esecurity.ch/cryptography2e.html.

[134] Paar, Christof und Jan Pelzl: *Understanding Cryptography – A Textbook for Students and Practioners.* Springer, 2009.

[135] Padberg, Friedhelm: *Elementare Zahlentheorie.* Spektrum Akademischer Verlag, 2. Auflage, 1996.

[136] Paillier, Pascal: *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes.* In: *Advances in Cryptology – EUROCRYPT'99*, 1999.

[137] Pfleeger, Charles P.: *Security in Computing.* Prentice-Hall, 2. Auflage, 1997.

[138] Phan, Raphael Chung Wei: *Mini Advanced Encryption Standard (Mini-AES): A Testbed for Cryptanalysis Students.* Cryptologia, 26(4):283–306, 2002.

[139] Phan, Raphael Chung Wei: *Impossible differential cryptanalysis of Mini-AES.* Cryptologia, 2003.
http://www.tandfonline.com/doi/abs/10.1080/0161-110391891964.

[140] Pieper, H.: *Zahlen aus Primzahlen.* Verlag Harri Deutsch, 3. Auflage, 1983.

[141] Pollard, John M.: *A Monte Carlo method for factorization.* BIT Numerical Mathematics 15, 3:331–334, 1975.

[142] Pollard, John M.: *Kangaroos, Monopoly and Discrete Logarithms.* J. Cryptology, 13(4):437–447, 2000.

[143] Pomerance, Carl: *The Quadratic Sieve Factoring Algorithm.* In: Blakley, G.R. und D. Chaum (Herausgeber): *Proceedings of Crypto '84, LNCS 196*, Seiten 169–182. Springer, 1984.

[144] Pomerance, Carl: *A tale of two sieves.* Notices Amer. Math. Soc, 43:1473–1485, 1996.

[145] Pommerening, Klaus: *Linearitätsmaße für Boolesche Abbildungen*, 2008. Manuskript, 30. Mai 2000. Letzte Revision 4. Juli 2008.
English equivalent: *Fourier Analysis of Boolean Maps – A Tutorial.*
http://www.staff.uni-mainz.de/pommeren/Kryptologie/Bitblock/A_Nonlin/nonlin.pdf.

[146] Pommerening, Klaus: *Fourier Analysis of Boolean Maps – A Tutorial*, 2014. Manuscript: May 30, 2000. Last revision August 11, 2014.
German aequivalent: *Linearitätsmaße für Boolesche Abbildungen.*
http://www.staff.uni-mainz.de/pommeren/Cryptology/Bitblock/Fourier/Fourier.pdf.

[147] Pommerening, Klaus: *Cryptanalysis of nonlinear shift registers.* Cryptologia, 30, 2016.
http://www.tandfonline.com/doi/abs/10.1080/01611194.2015.1055385.

[148] Ptacek, Thomas, Tom Ritter, Javed Samuel und Alex Stamos: *The Factoring Dead – Preparing for the Cryptopocalypse.* Black Hat Conference, 2013.

[149] Rempe, L. und R. Waldecker: *Primzahltests für Einsteiger.* Vieweg+Teubner, 2009.
Dieses Buch entstand aus einem Kurs an der „Deutschen Schülerakademie". Es stellt den AKS-Beweis vollständig dar, ohne viel mathematisches Vorwissen zu erwarten.

[150] Richstein, J.: *Verifying the Goldbach Conjecture up to* $4 * 10^{14}$. Mathematics of Computation, 70:1745–1749, 2001.

[151] Rivest, Ron L., Adi Shamir und Leonard Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21(2):120–126, April 1978.

[152] Satoh, T. und K. Araki: *Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves.* Commentarii Mathematici Universitatis Sancti Pauli 47, 1998.

[153] Sautoy, Marcus du: *Die Musik der Primzahlen: Auf den Spuren des größten Rätsels der Mathematik.* Beck, 4. Auflage, 2005.

[154] Savard, John J. G.: *A Cryptographic Compendium*, 1999.
`http://www.quadibloc.com/crypto/jscrypt.htm`.

[155] Schaefer, Edward F.: *A Simplified Data Encryption Standard Algorithm.* Cryptologia, 20(1):77–84, 1996.

[156] Scheid, Harald: *Zahlentheorie.* Spektrum Akademischer Verlag, 4. Auflage, 2006.

[157] Schmeh, Klaus: *Cryptography and Public Key Infrastructure on the Internet.* John Wiley, 2003. In German, the 6th edition was published in 2016.

[158] Schmeh, Klaus: *Codeknacker gegen Codemacher. Die faszinierende Geschichte der Verschlüsselung.* W3L Verlag Bochum, 2. Auflage, 2007.
Dieses Buch ist bis dato das aktuellste unter denen, die sich mit der Geschichte der Kryptographie beschäftigen.
Das Buch enthält auch eine kleine Sammlung gelöster und ungelöster Krypto-Rätsel. Eine der Challenges nutzt den „Doppelwürfel“ (doppelte Spaltentransposition) mit zwei langen Schlüsseln, die unterschiedlich sind.
Siehe die Challenge auf MTC3: `https://www.mysterytwisterc3.org/de/challenges/level-x-kryptographie-challenges/doppelwuerfel` .

[159] Schmeh, Klaus: *Kryptographie – Verfahren, Protokolle, Infrastrukturen.* dpunkt.verlag, 6. Auflage, 2016. Sehr gut lesbares, aktuelles und umfangreiches Buch über Kryptographie. Geht auch auf praktische Probleme (wie Standardisierung oder real existierende Software) ein.

[160] Schmidt, Jürgen: *Kryptographie in der IT – Empfehlungen zu Verschlüsselung und Verfahren.* c't, 2016(1), 2016.
Dieser Artikel erschien ursprünglich in c't 01/2016, Seite 174. Danach veröffentlicht am 17.06.2016 in:
`http://www.heise.de/security/artikel/Kryptographie-in-der-IT-Empfehlungen-zu-Verschluesselung-und-Verfahren-3221002.html`.

[161] Schneider, Matthias: *Analyse der Sicherheit des RSA-Algorithmus. Mögliche Angriffe, deren Einfluss auf sichere Implementierungen und ökonomische Konsequenzen.* Diplomarbeit, Universität Siegen, 2004.

[162] Schneier, Bruce: *Applied Cryptography, Protocols, Algorithms, and Source Code in C.* Wiley, 2. Auflage, 1996.

[163] Schneier, Bruce: *The Solitaire Encryption Algorithm*, 1999. v. 1.2.
`https://www.schneier.com/academic/solitaire/`.

[164] Schneier, Bruce: *A Self-Study Course in Block-Cipher Cryptanalysis.* Cryptologia, 24:18–34, 2000. `www.schneier.com/paper-self-study.pdf`.

[165] Schroeder, M. R.: *Number Theory in Science and Communication.* Springer, 3. Auflage, 1999.

[166] Schulz, Ralph Hardo und Helmut Witten: *Zeitexperimente zur Faktorisierung. Ein Beitrag zur Didaktik der Kryptographie.* LOG IN, 166/167:113–120, 2010. http://bscw.schule.de/pub/bscw.cgi/d864899/Schulz_Witten_Zeit-Experiment e.pdf.

[167] Schulz, Ralph Hardo, Helmut Witten und Bernhard Esslinger: *Rechnen mit Punkten einer elliptischen Kurve.* LOG IN, 2015(181/182):103–115, 2015. Geschrieben für Lehrer; didaktisch aufbereitet, leicht verständlich, mit vielen SageMath-Beispielen. http://bscw.schule.de/pub/nj_bscw.cgi/d1024028/Schulz_Witten_Esslinger-Rechnen_mit_Punkten_einer_elliptischen_Kurve.pdf.

[168] Schwenk, Jörg: *Conditional Access.* taschenbuch der telekom praxis. B. Seiler, Verlag Schiele und Schön, 1996.

[169] Schwenk, Jörg: *Sicherheit und Kryptographie im Internet.* Vieweg, 2002.

[170] Security, RSA: *Has the RSA algorithm been compromised as a result of Bernstein's Paper?* Technischer Bericht, RSA Security, April 2002. http://www.emc.com/emc-plus/rsa-labs/historical/has-the-rsa-algorithm-been-compromised.htm.

[171] Sedgewick, Robert: *Algorithms in C.* Addison-Wesley, 1990.

[172] Segers, A. J. M.: *Algebraic Attacks from a Gröbner Basis Perspective.* Diplomarbeit, Technische Universiteit Eindhoven, 2004. http://www.win.tue.nl/~henkvt/images/ReportSegersGB2-11-04.pdf.

[173] Semaev, I.: *Evaluation of Discrete Logarithms on Some Elliptic Curves.* Mathematics of Computation 67, 1998.

[174] Semaev, Igor: *Summation polynomials and the discrete logarithm problem on elliptic curves.* IACR Cryptology ePrint Archive, 2004:31, 2004.

[175] Shamir, A.: *A polynomial time algorithm for breaking the basic Merkle-Hellman Cryptosystem.* In: *Symposium on Foundations of Computer Science*, Seiten 145–152, 1982.

[176] Shamir, Adi und Eran Tromer: *Factoring Large Numbers with the TWIRL Device*, 2003. http://www.tau.ac.il/~tromer/papers/twirl.pdf.

[177] Shamir, Adi und Eran Tromer: *On the Cost of Factoring RSA-1024.* RSA Laboratories CryptoBytes, 6(2):11–20, 2003. http://www.tau.ac.il/~tromer/papers/cbtwirl.pdf.

[178] Shor, Peter W.: *Algorithms for Quantum Computation: Discrete Logarithms and Factoring.* In: *FOCS*, Seiten 124–134, 1994.

[179] Shor, Peter W.: *Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer.* SIAM Journal on Computing, 26(5):1484–1509, 1997.

[180] Shoup, Victor: *Lower Bounds for Discrete Logarithms and Related Problems.* In: *EUROCRYPT*, Seiten 256–266, 1997.

[181] Shoup, Victor: *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, 2. Auflage, 2008. http://shoup.net/ntb/.

[182] Silverman, J. und J. Tate: *Rational Points on Elliptic Curves.* Springer, 1992.

[183] Silverman, Joe: *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics 106. Springer, 2. Auflage, 2009, ISBN 978-0-387-09493-9.

[184] Silverman, Joseph H.: *The Xedni Calculus And The Elliptic Curve Discrete Logarithm Problem*. Designs, Codes and Cryptography, 20:5–40, 1999.

[185] Silverman, Robert D.: *A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths*. RSA Laboratories Bulletin, 13:1–22, April 2000.

[186] Singh, Simon: *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor, 1999.

[187] Singh, Simon: *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet*. dtv, 2001.

[188] Smart, N.: *The Discrete Logarithm Problem on Elliptic Curves of Trace One*. Journal of Cryptology 12, 1999.

[189] Stallings, William: *Cryptography and Network Security*. Pearson, 2014. `http://williamstallings.com/Cryptography/`.

[190] Stamp, Mark: *Information Security: Principles and Practice*. Wiley, 2. Auflage, 2011.

[191] Stamp, Mark und Richard M. Low: *Applied Cryptanalysis: Breaking Ciphers in the Real World*. Wiley-IEEE Press, 2007. `http://cs.sjsu.edu/faculty/stamp/crypto/`.

[192] Stinson, Douglas R.: *Cryptography – Theory and Practice*. Chapman & Hall/CRC, 3. Auflage, 2006.

[193] Swenson, Christopher: *Modern Cryptanalysis: Techniques for Advanced Code Breaking*. Wiley, 2008.

[194] ThinkQuest Team 27158: *Data Encryption*, 1999.

[195] Tietze, H.: *Gelöste und ungelöste mathematische Probleme*. C.H. Beck, 6. Auflage, 1973.

[196] U.S. Department of Commerce, National Bureau of Standards, National Technical Information Service, Springfield, Virginia: *Federal Information Processing Standards Publication 46: Data Encryption Standard*, 1977.

[197] Wang, Xiaoyun, Andrew Yao und Frances Yao: *New Collision Search for SHA-1*. Technischer Bericht, Crypto 2005, Rump Session, 2005. `http://www.iacr.org/conferences/crypto2005/rumpSchedule.html`.

[198] Wang, Xiaoyun, Yiqun Yin und Hongbo Yu: *Finding Collisions in the Full SHA-1*. Advances in Cryptology-Crypto, LNCS 3621, Seiten 17–36, 2005.

[199] Washington, Lawrence C.: *Elliptic Curves: Number Theory and Cryptography*. Discrete Mathematics and its Applications. Chapman and Hall/CRC, 2008, ISBN 9781420071467.

[200] Weis, Rüdiger, Stefan Lucks und Andreas Bogk: *Sicherheit von 1024 bit RSA-Schlüsseln gefährdet*. Datenschutz und Datensicherheit (DuD), 27(6):360–362, 2003.

[201] Welschenbach, Michael: *Kryptographie in C und C++*. Springer, 2001.

[202] Wikipedia: *Homomorphic Encryption & Homomorphismus.*
`https://en.wikipedia.org/wiki/Homomorphic_encryption`,
`https://de.wikipedia.org/wiki/Homomorphismus`.

[203] Wikipedia: *Secure Multiparty Computation.*
`http://en.wikipedia.org/wiki/Secure_multi-party_computation`.

[204] Wiles, Andrew: *Modular elliptic curves and fermat's last theorem.* Annals of Mathematics, 141, 1995.

[205] Witten, Helmut, Irmgard Letzner und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle, Teil 1: Sprache und Statistik.* LOG IN, 1998(3/4):57–65, 1998.
`http://bscw.schule.de/pub/bscw.cgi/d637160/RSA_u_Co_T1.pdf`.

[206] Witten, Helmut, Irmgard Letzner und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. Teil 3: Flusschiffren, perfekte Sicherheit und Zufall per Computer.* LOG IN, 1999(2):50–57, 1999. `http://bscw.schule.de/pub/nj_bscw.cgi/d637156/RSA_u_Co_T3.pdf`.

[207] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 2: RSA für große Zahlen.* LOG IN, 2006(143):50–58, 2006.
`http://bscw.schule.de/pub/bscw.cgi/d404410/RSA_u_Co_NF2.pdf`.

[208] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 3: RSA und die elementare Zahlentheorie.* LOG IN, 2008(152):60–70, 2008.
`http://bscw.schule.de/pub/nj_bscw.cgi/d533821/RSA_u_Co_NF3.pdf`.

[209] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 4: Gibt es genügend Primzahlen für RSA?* LOG IN, 2010(163):97–103, 2010.
`http://bscw.schule.de/pub/nj_bscw.cgi/d864891/RSA_u_Co_NF4.pdf`.

[210] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 5: Der Miller-Rabin-Primzahltest oder: Falltüren für RSA mit Primzahlen aus Monte Carlo.* LOG IN, 2010(166/167):92–106, 2010.
`http://bscw.schule.de/pub/nj_bscw.cgi/d864895/RSA_u_Co_NF5.pdf`.

[211] Witten, Helmut, Ralph Hardo Schulz und Bernhard Esslinger: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle, NF Teil 7: Alternativen zu RSA oder Diskreter Logarithmus statt Faktorisierung.* LOG IN, 2010(181-182):85–102, 2015.
Hierin werden u.a. DH und Elgamal in einem breiteren Kontext behandelt. Die Verfahren werden mit Codebeispielen in Python und SageMath erläutert.
`http://bscw.schule.de/pub/nj_bscw.cgi/d1024013/RSA_u_Co_NF7.pdf`,
`http://www.log-in-verlag.de/wp-content/uploads/2015/07/Internetquellen-LOG-IN-Heft-Nr.181-182.doc`.

[212] Wobst, Reinhard: *Angekratzt – Kryptoanalyse von AES schreitet voran.* iX, Dezember 2002. (Und der Leserbrief dazu von Johannes Merkle in der iX 2/2003).

[213] Wobst, Reinhard: *New Attacks Against Hash Functions.* Information Security Bulletin, April 2005.

[214] Yan, Song Y.: *Number Theory for Computing.* Springer, 2000.

[215] Young, Adam L. und Moti Yung: *The Dark Side of Black-Box Cryptography, or: Should We Trust Capstone?* In: *CRYPTO*, Seiten 89–103, 1996.

[216] Young, Adam L. und Moti Yung: *Kleptography: Using Cryptography against Cryptography.* In: *EUROCRYPT*, Seiten 62–74, 1997.

Literaturverzeichnis über alle Kapitel (sortiert by babalpha)

[Aar03] Aaronson, Scott: *The Prime Facts: From Euclid to AKS*, 2003. http://www.scottaaronson.com/writings/prime.pdf.

[ACA02] ACA: *Length and Standards for all ACA Ciphers*. Technischer Bericht, American Cryptogram Association, 2002. http://www.cryptogram.org/cdb/aca.info/aca.and.you/chap08.html#, http://www.und.edu/org/crypto/crypto/.chap08.html.

[Adl79] Adleman, Leonard M.: *A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography (Abstract)*. In: *FOCS*, Seiten 55–60, 1979.

[Adl83] Adleman, L.: *On breaking the iterated Merkle-Hellman public-key Cryptosystem*. In: *Advances in Cryptologie, Proceedings of Crypto 82*, Seiten 303–308. Plenum Press, 1983.

[AES02] National Institute of Standards and Technology (NIST): *Federal Information Processing Standards Publication 197: Advanced Encyption Standard*, 2002.

[Age13] Agence nationale de la sécurité des systèmes d'information: *Référentiel général de sécurité Version 2.02*, 2013. http://www.ssi.gouv.fr/administration/reglementation/.

[AKS02] Agrawal, M., N. Kayal und N. Saxena: *PRIMES in P*, August 2002. Corrected version. http://www.cse.iitk.ac.in/~manindra/algebra/primality_v6.pdf, http://fatphil.org/maths/AKS/.

[Bac84] Bach, Eric: *Discrete Logarithms and Factoring*. Technischer Bericht UCB/CSD-84-186, EECS Department, University of California, Berkeley, Juni 1984. http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5973.html, http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-186.pdf.

[Bar09] Bard, Gregory V.: *Algebraic Cryptanalysis*. Springer, Dordrecht, 2009.

[Bau95] Bauer, Friedrich L.: *Entzifferte Geheimnisse*. Springer, 1995.

[Bau00] Bauer, Friedrich L.: *Decrypted Secrets*. Springer, 2. Auflage, 2000.

[BB85] Bennett, Charles H. und Gilles Brassard: *An Update on Quantum Cryptography.* In: Blakley, G. R. und David Chaum (Herausgeber): *Advances in Cryptology – CRYPTO '84*, Band 196 der Reihe *Lecture Notes in Computer Science*, Seiten 475–480. Springer-Verlag, 1985.

[BCD+06] Buchmann, Johannes, Luis Carlos Coronado García, Erik Dahmen, Martin Döring und Elena Klintsevich: *CMSS – an improved Merkle signature scheme.* In: Barua, Rana und Tanja Lange (Herausgeber): *7th International Conference on Cryptology in India - Indocrypt'06*, Nummer 4392 in *Lecture Notes in Computer Science*, Seiten 349–363. Springer-Verlag, 2006.

[BCLN14] Bos, Joppe W., Craig Costello, Patrick Longa und Michael Naehrig: *Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis*, 2014. Microsoft Research.
`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/selecting.pdf`.

[BDG98] Balcazar, J. L., J. Daaz und J. Gabarr: *Structural Complexity I.* Springer, 1998.

[Ber01] Bernstein, Daniel J.: *Circuits for integer factorization: a proposal.*
`http://cr.yp.to/papers/nfscircuit.ps`,
`http://cr.yp.to/djb.html`, 2001.

[Ber05] Bernstein, Daniel J.: *Factoring into coprimes in essentially linear time.* Journal of Algorithms, 54, 2005. `http://cr.yp.to/lineartime/dcba-20040404.pdf`.

[Beu96] Beutelspacher, Albrecht: *Kryptologie.* Vieweg, 5. Auflage, 1996.

[BFT02] Bourseau, F., D. Fox und C. Thiel: *Vorzüge und Grenzen des RSA-Verfahrens.* Datenschutz und Datensicherheit (DuD), 26:84–89, 2002.
`http://www.secorvo.de/publikationen/rsa-grenzen-fox-2002.pdf`.

[BGJT13] Barbulescu, Razvan, Pierrick Gaudry, Antoine Joux und Emmanuel Thomé: *A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic.* CoRR, abs/1306.4244, 2013.

[BK09] Biryukov, Alex und Dmitry Khovratovich: *Related-key Cryptanalysis of the Full AES-192 and AES-256.* Cryptology ePrint Archive, 2009. `http://eprint.iacr.org/2009/317`.

[BL14] Bernstein, Daniel und Tanja Lange: *SafeCurves: choosing safe curves for elliptic-curve cryptography.* `http://safecurves.cr.yp.to/`, 2014.

[BLP93] Buhler, J. P., H. W. Lenstra und C. Pomerance: *Factoring integers with the number field sieve.* In: Lenstra, K. und H.W. Lenstra (Herausgeber): *The Development of the Number Field Sieve, Lecture Notes in Mathematics, Vol. 1554*, Seiten 50–94. Springer, 1993.

[Blu99] Blum, W.: *Die Grammatik der Logik.* dtv, 1999.

[Blö99] Blömer, J.: *Vorlesungsskript Algorithmische Zahlentheorie*, 1999. Ruhr-University Bochum.
`http://www.math.uni-frankfurt.de/~dmst/teaching/lecture_notes/bloemer.algorithmische_zahlentheorie.ps.gz`.

[Bra02] Brands, Gilbert: *Verschlüsselungsalgorithmen – Angewandte Zahlentheorie rund um Sicherheitsprotokolle.* Vieweg, 2002.

[Bri85] Brickell, E. F.: *Breaking Iterated Knapsacks.* In: *Advances in Cryptology: Proc. CRYPTO'84, Lecture Notes in Computer Science, vol. 196*, Seiten 342–358. Springer, 1985.

[Bri10] Brickenstein, Michael: *Boolean Gröbner Bases – Theory, Algorithms and Applications.* Dissertation, TU Kaiserslautern, department of Mathematics, 2010.
See also "PolyBoRi – Polynomials over Boolean Rings", online:
`http://polybori.sourceforge.net/`".

[BRK96] Bartholomé, A., J. Rung und H. Kern: *Zahlentheorie für Einsteiger.* Vieweg, 2. Auflage, 1996.

[BSI12] BSI (Bundesamt für Sicherheit in der Informationstechnik): *BSI TR-02102 Kryptographische Verfahren: Empfehlungen und Schlüssellängen*, 2012.
`https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm`.

[BSI16] BSI: *Angaben des BSI für die Algorithmenkataloge der Vorjahre, Empfehlungen zur Wahl der Schlüssellängen.* Technischer Bericht, BSI (Bundesamt für Sicherheit in der Informationstechnik), 2016.
`https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/ElektronischeSignatur/TechnischeRealisierung/Kryptoalgorithmen/kryptoalg.html`
- Vgl.: BNetzA (Bundesnetzagentur): Jährlich erscheinende Veröffentlichung zu Algorithmen und Parametern im Umfeld elektronischer Signaturen:
`http://www.bundesnetzagentur.de/cln_1411/DE/Service-Funktionen/ElektronischeVertrauensdienste/QES/WelcheAufgabenhatdieBundesnetzagentur/GeeigneteAlgorithmenfestlegen/geeignetealgorithmenfestlegen_node.html`
- Vgl.: Eine Stellungnahme zu diesen Empfehlungen:
`http://www.secorvo.de/publikationen/stellungnahme-algorithmenempfehlung-020307.pdf` .

[BSW98] Beutelspacher, Albrecht, Jörg Schwenk und Klaus Dieter Wolfenstetter: *Moderne Verfahren in der Kryptographie.* Vieweg, 2. Auflage, 1998.

[Buc16] Buchmann, Johannes: *Einführung in die Kryptographie.* Springer, 6. Auflage, 2016. Paperback.

[Bun98] Bundschuh, Peter: *Einführung in die Zahlentheorie.* Springer, 4. Auflage, 1998.

[Cas91] Cassels, J. W. S.: *Lectures on Elliptic Curves.* Cambridge University Press, 1991.

[CGH+03] Castro, D., M. Giusti, J. Heintz, G. Matera und L. M. Pardo: *The hardness of polynomial equation solving.* Found. Comput. Math., 3:347–420, 2003.

[CH10] Crama, Yves und Peter L. Hammer (Herausgeber): *Boolean Models and Methods in Mathematics, Computer Science, and Engineering.* Cambridge University Press, 2010.

[CLN15] Costello, Craig, Patrick Longa und Michael Naehrig: *A brief discussion on selecting new elliptic curves*, 2015. Microsoft Research.
`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/NIST.pdf`.

[CLO07] Cox, David, John Little und Donal O'Shea: *Ideals, Varieties, and Algorithms*. Springer, 3. Auflage, 2007.

[Cop84] Coppersmith, Don: *Evaluating Logarithms in $GF(2^n)$*. In: *STOC*, Seiten 201–207, 1984.

[Cop02] Coppersmith, Don: *Re: Impact of Courtois and Pieprzyk results*. Journal unknown, 2002.
`http://csrc.nist.gov/archive/aes/` Former link from the AES Discussion Groups.

[COS86] Coppersmith, Don, Andrew M. Odlyzko und Richard Schroeppel: *Discrete Logarithms in GF(p)*. Algorithmica, 1(1):1–15, 1986.

[CP01] Crandell, Richard und Carl Pomerance: *Prime Numbers. A Computational Perspective*. Springer, 2001.

[CP02] Courtois, Nicolas und Josef Pieprzyk: *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations*. Cryptology ePrint Archive, 2002.
A different, so called compact version of the first XSL attack, was published in the proceedings for Asiacrypt Dec 2002. `http://eprint.iacr.org/2002/044`.

[Cro00] Crowley, Paul: *Mirdek: A card cipher inspired by "Solitaire"*, 2000. `http://www.ciphergoth.org/crypto/mirdek/`.

[CS09] Cusick, Thomas W. und Pantelimon Stănică: *Cryptographic Boolean Functions and Applications*. Elsevier Academic Press, 2009.

[DES77] U.S. Department of Commerce, National Bureau of Standards, National Technical Information Service, Springfield, Virginia: *Federal Information Processing Standards Publication 46: Data Encryption Standard*, 1977.

[Dox01] Doxiadis, Apostolos: *Onkel Petros und die Goldbachsche Vermutung*. Lübbe, 2001.

[DR02] Daemen, Joan und Vincent Rijmen: *The Design of Rijndael. AES – The Advanced Encryption Standard*. Springer, 2002.

[Dro15] Drobick, Jörg: *Abriss DDR-Chiffriergeschichte: SAS- und Chiffrierdienst*, 2015. `http://scz.bplaced.net/m.html#dwa`.

[dS05] Sautoy, Marcus du: *Die Musik der Primzahlen: Auf den Spuren des größten Rätsels der Mathematik*. Beck, 4. Auflage, 2005.

[Eck14] Eckert, Claudia: *IT-Sicherheit: Konzepte-Verfahren-Protokolle*. De Gruyter Oldenbourg, 9. Auflage, 2014. Paperback.

[EGT11] Enge, Andreas, Pierrick Gaudry und Emmanuel Thomé: *An L(1/3) Discrete Logarithm Algorithm for Low Degree Curves*. J. Cryptology, 24(1):24–41, 2011.

[Ert01] Ertel, Wolfgang: *Angewandte Kryptographie*. Fachbuchverlag Leipzig FV, 2001.

[ESS12] Esslinger, B., J. Schneider und V. Simon: *RSA – Sicherheit in der Praxis*. KES Zeitschrift für Informationssicherheit, 2012(2):22–27, April 2012. https://www.cryptool.org/images/ctp/documents/kes_2012_RSA_Sicherheit.pdf.

[ESS14] Esslinger, B., J. Schneider und V. Simon: *Krypto + NSA = ? – Kryptografische Folgerungen aus der NSA-Affäre*. KES Zeitschrift für Informationssicherheit, 2014(1):70–77, März 2014. https://www.cryptool.org/images/ctp/documents/krypto_nsa.pdf.

[FJM14] Fouque, Pierre Alain, Antoine Joux und Chrysanthi Mavromati: *Multi-user collisions: Applications to Discrete Logarithm, Even-Mansour and Prince*. Cryptology ePrint Archive, 2014. https://eprint.iacr.org/2013/761.

[FSW01] Ferguson, Niels, Richard Schroeppel und Doug Whiting: *A simple algebraic representation of Rijndael*, 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.4921.

[Gal12] Galbraith, Steven D.: *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012, ISBN 9781107013926.

[Gau09] Gaudry, Pierrick: *Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem*. J. Symb. Comput., 44(12):1690–1702, 2009.

[Gen09] Gentry, Craig: *Fully Homomorphic Encryption Using Ideal Lattices*. In: *41st ACM Symposium on Theory of Computing (STOC)*, 2009.

[GGMZ13] Gölоglu, Faruk, Robert Granger, Gary McGuire und Jens Zumbrägel: *On the Function Field Sieve and the Impact of Higher Splitting Probabilities – Application to Discrete Logarithms*. In: *CRYPTO (2)*, Seiten 109–128, 2013.

[GHS02] Gaudry, Pierrick, Florian Hess und Nigel P. Smart: *Constructive and Destructive Facets of Weil Descent on Elliptic Curves*. J. Cryptology, 15(1):19–46, 2002.

[GJ79] Garey, Michael R. und David S. Johnson: *Computers and Intractability*. Freeman, 1979.

[GKP94] Graham, R. E., D. E. Knuth und O. Patashnik: *Concrete Mathemathics, a Foundation of Computer Science*. Addison Wesley, 6. Auflage, 1994.

[Goe14] Goebel, Greg: *Codes, Ciphers and Codebreaking*, 2014. Version 2.3.2. http://www.vectorsite.net/ttcode.html.

[Goe15] Goebel, Greg: *A Codes & Ciphers Primer*, 2015. Version 1.0.5. http://www.vectorsite.net/ttcode.html.

[Gol82] Golomb, Solomon W.: *Shift Register Sequences*. Aegean Park Press, 1982. Revised Edition.

[Haa08] Haan, Kristian Laurent: *Advanced Encryption Standard (AES)*, 2008. http://www.codeplanet.eu/tutorials/cpp/51-advanced-encryption-standard.html.

[HDWH12] Heninger, Nadia, Zakir Durumeric, Eric Wustrow und J. Alex Halderman: *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.* In: *Proceedings of the 21st USENIX Security Symposium*, August 2012. `https://factorable.net/paper.html`.

[Hes01] Hesselink, Wim H.: *The borderline between P and NP*, Februar 2001. `http://www.cs.rug.nl/~wim/pub/whh237.pdf`.

[Hil29] Hill, Lester S.: *Cryptography in an Algebraic Alphabet.* The American Mathematical Monthly, 36(6):306–312, 1929.

[Hil31] Hill, Lester S.: *Concerning Certain Linear Transformation Apparatus of Cryptography.* The American Mathematical Monthly, 38(3):135–154, 1931.

[Hof06] Hoffman, Nick: *A SIMPLIFIED IDEA ALGORITHM*, 2006. `http://www.nku.edu/~christensen/simplified%20IDEA%20algorithm.pdf`.

[Hom07] Homeister, Matthias: *Quantum Computer Science: An Introduction.* Vieweg+Teubner Verlag, 2007, ISBN 9780521876582.

[IT95] ITU-T: *X.509 (1993) Amendment 1: Certificate Extensions, The Directory Authentication Framework.* Technischer Bericht, International Telecommunication Union ITU-T, Juli 1995. (equivalent to amendment 1 to ISO/IEC 9594-8).

[IT97] ITU-T: *ITU-T Recommendation X.509 (1997 E): Information Technology – Open Systems Interconnection – The Directory: Authentication Framework.* Technischer Bericht, International Telecommunication Union ITU-T, Juni 1997.

[JL06] Joux, Antoine und Reynald Lercier: *The Function Field Sieve in the Medium Prime Case.* In: *EUROCRYPT*, Seiten 254–270, 2006.

[Jou09] Joux, Antoine: *Algorithmic Cryptanalysis.* CRC Cryptography and Network Security Series. Chapman & Hall, 2009, ISBN 1420070029.

[Jou13a] Joux, Antoine: *A new index calculus algorithm with complexity L(1/4+o(1)) in very small characteristic.* IACR Cryptology ePrint Archive, 2013:95, 2013.

[Jou13b] Joux, Antoine: *Faster Index Calculus for the Medium Prime Case Application to 1175-bit and 1425-bit Finite Fields.* In: *EUROCRYPT*, Seiten 177–193, 2013.

[JV11] Joux, Antoine und Vanessa Vitse: *Cover and Decomposition Index Calculus on Elliptic Curves made practical. Application to a seemingly secure curve over* F_{p^6}. IACR Cryptology ePrint Archive, 2011:20, 2011.

[KAF+10] Kleinjung, Thorsten, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev und Paul Zimmermann: *Factorization of a 768-Bit RSA Modulus.* In: *CRYPTO*, Seiten 333–350, 2010.

[Kat01] Katzenbeisser, Stefan: *Recent Advances in RSA Cryptography.* Springer, 2001.

[Kip97] Kippenhahn, Rudolf: *Verschlüsselte Botschaften: Geheimschrift, Enigma und Chipkarte.* rowohlt, 1. Auflage, 1997. New edition 2012, Paperback, *Verschlüsselte Botschaften: Geheimschrift, Enigma und digitale Codes.*

[Kip99] Kippenhahn, Rudolph: *Code Breaking – A History and Exploration.* Constable, 1999.

[Kle10] Kleinjung, Thorsten et al.: *Factorization of a 768-bit RSA modulus, version 1.4*, 2010. `http://eprint.iacr.org/2010/006.pdf`.

[Knu98] Knuth, Donald E.: *The Art of Computer Programming, vol 2: Seminumerical Algorithms.* Addison-Wesley, 3. Auflage, 1998.

[Kob84] Koblitz, N.: *Introduction to Elliptic Curves and Modular Forms.* Graduate Texts in Mathemathics, Springer, 1984.

[Kob98] Koblitz, N.: *Algebraic Aspects of Cryptography. With an appendix on Hyperelleptic Curves by Alfred J. Menezes, Yi Hong Wu, and Robert J. Zuccherato.* Springer, 1998.

[KW97] Klee, V. und S. Wagon: *Ungelöste Probleme in der Zahlentheorie und der Geometrie der Ebene.* Birkhäuser Verlag, 1997.

[Lab02] Labs, RSA: *PKCS #1 v2.1 Draft 3: RSA Cryptography Standard.* Technischer Bericht, RSA Laboratories, April 2002.

[Lag83] Lagarias, J. C.: *Knapsack public key Cryptosystems and diophantine Approximation.* In: *Advances in Cryptology, Proceedings of Crypto 83.* Plenum Press, 1983.

[Laz83] Lazard, Daniel: *Gröbner bases, Gaussian elimination and resolution of systems of algebraic equations.* In: *Lecture Notes in Computer Science 162*, Seiten 146–156. Springer, 1983. EUROCAL '83.

[Len87] Lenstra, H. W.: *Factoring Integers with Elliptic Curves.* Annals of Mathematics, 126:649–673, 1987.

[LHA+12a] Lenstra, Arjen K., James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung und Christophe Wachter: *Public Keys.* In: *CRYPTO*, Seiten 626–642, 2012.

[LHA+12b] Lenstra, Arjen K., James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung und Christophe Wachter: *Ron was wrong, Whit is right, A Sanity Check of Public Keys Collected on the Web.* Cryptology ePrint Archive, Februar 2012. `http://eprint.iacr.org/2012/064.pdf`.

[LL93a] Lenstra, A. und H. Lenstra: *The development of the Number Field Sieve.* Lecture Notes in Mathematics 1554. Springer, 1993.

[LL93b] Lenstra, A. K. und H. W. Jr. Lenstra: *The Development of the Number Field Sieve.* Lecture Notes in Mathematics. Springer Verlag, 1993, ISBN 0387570136.

[LM05] Lochter, Manfred und Johannes Merkle: *ECC Brainpool Standard Curves and Curve Generation v. 1.0*, 2005. `www.ecc-brainpool.org/download/Domain-parameters.pdf`.

[LM10] Lochter, Manfred und Johannes Merkle: *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, 2010. RFC 5639. `http://www.rfc-base.org/txt/rfc-5639.txt`.

[LM13] Lochter, Manfred und Johannes Merkle: *Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)*, 2013. RFC 7027. http://tools.ietf.org/search/rfc7027.

[Lor93] Lorenz, F.: *Algebraische Zahlentheorie*. BI Wissenschaftsverlag, 1993.

[LSTT02] Lenstra, Arjen K., Adi Shamir, Jim Tomlinson und Eran Tromer: *Analysis of Bernstein's Factorization Circuit*, 2002. http://tau.ac.il/~tromer/papers/meshc.pdf.

[LV00] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes*. In: *Lecture Notes in Computer Science 558*, Seiten 446–465, 2000. PKC2000. See also "BlueKrypt Cryptographic Key Length Recommendation", last update 2015, online: http://www.keylength.com/en/2/.

[LV01] Lenstra, Arjen K. und Eric R. Verheul: *Selecting Cryptographic Key Sizes (1999 + 2001)*. Journal of Cryptology, 14:255–293, 2001. http://www.cs.ru.nl/E.Verheul/papers/Joc2001/joc2001.pdf, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.69&rep=rep1&type=pdf.

[LW02] Lucks, Stefan und Rüdiger Weis: *Neue Ergebnisse zur Sicherheit des Verschlüsselungsstandards AES*. DuD, Dezember 2002.

[Mas02] Maseberg, Jan Sönke: *Fail-Safe-Konzept für Public-Key-Infrastrukturen*. Dissertation, TU Darmstadt, 2002.

[May08] May, Alexander: *Vorlesungsskript Kryptanalyse 1*, 2008. Ruhr-University Bochum. http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/pkk08/skript.pdf.

[May12] May, Alexander: *Vorlesungsskript Kryptanalyse 2*, 2012. Ruhr-University Bochum. http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/12/ws1213/kryptanal12/kryptanalyse_2013.pdf.

[May13] May, Alexander: *Vorlesungsskript Zahlentheorie*, 2013. Ruhr-University Bochum. http://www.cits.ruhr-uni-bochum.de/imperia/md/content/may/13/ss13/zahlenss13/zahlentheorie.pdf.

[MB07] Mansoori, S. Davod und H. Khaleghei Bizaki: *On the vulnerability of Simplified AES Algorithm Against Linear Cryptanalysis*. IJCSNS International Journal of Computer Science and Network Security, 7(7):257–263, 2007. http://paper.ijcsns.org/07_book/200707/20070735.pdf.

[McE78] McEliece, Robert J.: *A public key cryptosystem based on algebraic coding theory*. DSN progress report, 42–44:114–116, 1978.

[Men93] Menezes, A. J.: *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[Mer79] Merkle, Ralph C.: *Secrecy, authentication, and public key systems*. Dissertation, Department of Electrical Engineering, Stanford University, 1979.

[Mer08] Mermin, David N.: *Quantum Computing verstehen*. Cambridge University Press, 2008, ISBN 3834804363.

[MH78] Merkle, R. und M. Hellman: *Hiding information and signatures in trapdoor knapsacks*. IEEE Trans. Information Theory, IT-24, 24, 1978.

[MHP12] McDonald, Cameron, Philip Hawkes und Josef Pieprzyk: *Differential Path for SHA-1 with complexity $O(2^{52})$*. Cryptology ePrint Archive, 2012. `http://eprint.iacr.org/2009/259`.

[MM00] Massacci, Fabio und Laura Marraro: *Logical Cryptanalysis as a SAT Problem: Encoding and Analysis*. Journal of Automated Reasoning Security, 24:165–203, 2000.

[MR02a] Murphy, S. P. und M. J. B. Robshaw: *Comments on the Security of the AES and the XSL Technique*, September 2002. `http://crypto.rd.francetelecom.com/people/Robshaw/rijndael/rijndael.html`.

[MR02b] Murphy, S. P. und M. J. B. Robshaw: *Essential Algebraic Structure within the AES*. Technischer Bericht, Crypto 2002, 2002. `http://crypto.rd.francetelecom.com/people/Robshaw/rijndael/rijndael.html`.

[MS89] Meier, W. und O. Staffelbach: *Fast correlation attacks on certain stream ciphers*. Journal of Cryptology, 1:159–176, 1989.

[MSP11] Müller-Stach und Piontkowski: *Elementare und Algebraische Zahlentheorie*. Vieweg Studium, 2011, ISBN 3834882631.

[MSW03] Musa, Mohammad A., Edward F. Schaefer und Stephen Wedig: *A simplified AES algorithm and its linear and differential cryptanalyses*. Cryptologia, 17(2):148–177, April 2003.
`http://www.rose-hulman.edu/~holden/Preprints/s-aes.pdf`,
`http://math.scu.edu/eschaefe/` Ed Schaefer's homepage.

[MvOV01] Menezes, Alfred J., Paul C. van Oorschot und Scott A. Vanstone: *Handbook of Applied Cryptography*. Series on Discrete Mathematics and Its Application. CRC Press, 5. Auflage, 2001, ISBN 0-8493-8523-7. (Errata last update Jan 22, 2014).
`http://cacr.uwaterloo.ca/hac/`,
`http://www.cacr.math.uwaterloo.ca/hac/`.

[MZ06] Mironov, Ilya und Lintao Zhang: *Applications of SAT Solvers to Cryptanalysis of Hash Functions*. Springer, 2006.

[Ngu09a] Nguyen, Minh Van: *Exploring Cryptography Using the Sage Computer Algebra System*. Diplomarbeit, Victoria University, 2009.
`http://www.sagemath.org/files/thesis/nguyen-thesis-2009.pdf`,
`http://www.sagemath.org/library-publications.html`.

[Ngu09b] Nguyen, Minh Van: *Number Theory and the RSA Public Key Cryptosystem – An introductory tutorial on using SageMath to study elementary number theory and public key cryptography*, 2009. `http://faculty.washington.edu/moishe/hanoiex/Number%20Theory%20Applications/numtheory-crypto.pdf`.

[Nic96] Nichols, Randall K.: *Classical Cryptography Course, Volume 1 and 2*. Technischer Bericht, Aegean Park Press 1996, 1996. 12 lessons.
`www.apprendre-en-ligne.net/crypto/bibliotheque/lanaki/lesson1.htm`.

[NIS97] NIST: *Entity authentication using public key cryptography.* Technischer Bericht, NIST (U.S. Department of Commerce), 1997. No more valid (withdrawal date: October 2015).

[NIS13] NIST: *Digital Signature Standard (DSS).* Technischer Bericht, NIST (U.S. Department of Commerce), 2013. Change note 4.
`http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`,
`http://csrc.nist.gov/publications/PubsFIPS.html`.

[NIS15] NIST: *Secure Hash Standard (SHS).* Technischer Bericht, NIST (U.S. Department of Commerce), August 2015. FIPS 180-4 supersedes FIPS 180-2.
`http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`,
`http://csrc.nist.gov/publications/PubsFIPS.html`.

[Oec03] Oechslin, Philippe: *Making a Faster Cryptanalytic Time-Memory Trade-Off.* Technischer Bericht, Crypto 2003, 2003.
`http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf`.

[Opp11] Oppliger, Rolf: *Contemporary Cryptography, Second Edition.* Artech House, 2. Auflage, 2011. `http://books.esecurity.ch/cryptography2e.html`.

[Pad96] Padberg, Friedhelm: *Elementare Zahlentheorie.* Spektrum Akademischer Verlag, 2. Auflage, 1996.

[Pai99] Paillier, Pascal: *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes.* In: *Advances in Cryptology – EUROCRYPT'99*, 1999.

[Pfl97] Pfleeger, Charles P.: *Security in Computing.* Prentice-Hall, 2. Auflage, 1997.

[Pha02] Phan, Raphael Chung Wei: *Mini Advanced Encryption Standard (Mini-AES): A Testbed for Cryptanalysis Students.* Cryptologia, 26(4):283–306, 2002.

[Pha03] Phan, Raphael Chung Wei: *Impossible differential cryptanalysis of Mini-AES.* Cryptologia, 2003.
`http://www.tandfonline.com/doi/abs/10.1080/0161-110391891964`.

[Pie83] Pieper, H.: *Zahlen aus Primzahlen.* Verlag Harri Deutsch, 3. Auflage, 1983.

[Pol75] Pollard, John M.: *A Monte Carlo method for factorization.* BIT Numerical Mathematics 15, 3:331–334, 1975.

[Pol00] Pollard, John M.: *Kangaroos, Monopoly and Discrete Logarithms.* J. Cryptology, 13(4):437–447, 2000.

[Pom84] Pomerance, Carl: *The Quadratic Sieve Factoring Algorithm.* In: Blakley, G.R. und D. Chaum (Herausgeber): *Proceedings of Crypto '84, LNCS 196*, Seiten 169–182. Springer, 1984.

[Pom96] Pomerance, Carl: *A tale of two sieves.* Notices Amer. Math. Soc, 43:1473–1485, 1996.

[Pom08] Pommerening, Klaus: *Linearitätsmaße für Boolesche Abbildungen*, 2008. Manuskript, 30. Mai 2000. Letzte Revision 4. Juli 2008.
English equivalent: *Fourier Analysis of Boolean Maps – A Tutorial.*
`http://www.staff.uni-mainz.de/pommeren/Kryptologie/Bitblock/A_Nonlin/nonlin.pdf`.

[Pom14] Pommerening, Klaus: *Fourier Analysis of Boolean Maps – A Tutorial*, 2014. Manuscript: May 30, 2000. Last revision August 11, 2014.
German aequivalent: *Linearitätsmaße für Boolesche Abbildungen.*
`http://www.staff.uni-mainz.de/pommeren/Cryptology/Bitblock/Fourier/Fourier.pdf`.

[Pom16] Pommerening, Klaus: *Cryptanalysis of nonlinear shift registers.* Cryptologia, 30, 2016.
`http://www.tandfonline.com/doi/abs/10.1080/01611194.2015.1055385`.

[PP09] Paar, Christof und Jan Pelzl: *Understanding Cryptography – A Textbook for Students and Practioners.* Springer, 2009.

[PRSS13] Ptacek, Thomas, Tom Ritter, Javed Samuel und Alex Stamos: *The Factoring Dead – Preparing for the Cryptopocalypse.* Black Hat Conference, 2013.

[Ric01] Richstein, J.: *Verifying the Goldbach Conjecture up to* $4 * 10^{14}$. Mathematics of Computation, 70:1745–1749, 2001.

[RSA78] Rivest, Ron L., Adi Shamir und Leonard Adleman: *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21(2):120–126, April 1978.

[RW09] Rempe, L. und R. Waldecker: *Primzahltests für Einsteiger.* Vieweg+Teubner, 2009. Dieses Buch entstand aus einem Kurs an der „Deutschen Schülerakademie". Es stellt den AKS-Beweis vollständig dar, ohne viel mathematisches Vorwissen zu erwarten.

[SA98] Satoh, T. und K. Araki: *Fermat Quotients and the Polynomial Time Discrete Log Algorithm for Anomalous Elliptic Curves.* Commentarii Mathematici Universitatis Sancti Pauli 47, 1998.

[Sav99] Savard, John J. G.: *A Cryptographic Compendium*, 1999.
`http://www.quadibloc.com/crypto/jscrypt.htm`.

[Sch96a] Schaefer, Edward F.: *A Simplified Data Encryption Standard Algorithm.* Cryptologia, 20(1):77–84, 1996.

[Sch96b] Schneier, Bruce: *Applied Cryptography, Protocols, Algorithms, and Source Code in C.* Wiley, 2. Auflage, 1996.

[Sch96c] Schwenk, Jörg: *Conditional Access.* taschenbuch der telekom praxis. B. Seiler, Verlag Schiele und Schön, 1996.

[Sch99a] Schneier, Bruce: *The Solitaire Encryption Algorithm*, 1999. v. 1.2.
`https://www.schneier.com/academic/solitaire/`.

[Sch99b] Schroeder, M. R.: *Number Theory in Science and Communication.* Springer, 3. Auflage, 1999.

[Sch00] Schneier, Bruce: *A Self-Study Course in Block-Cipher Cryptanalysis.* Cryptologia, 24:18–34, 2000. `www.schneier.com/paper-self-study.pdf`.

[Sch02] Schwenk, Jörg: *Sicherheit und Kryptographie im Internet.* Vieweg, 2002.

[Sch03] Schmeh, Klaus: *Cryptography and Public Key Infrastructure on the Internet.* John Wiley, 2003. In German, the 6th edition was published in 2016.

[Sch04] Schneider, Matthias: *Analyse der Sicherheit des RSA-Algorithmus. Mögliche Angriffe, deren Einfluss auf sichere Implementierungen und ökonomische Konsequenzen.* Diplomarbeit, Universität Siegen, 2004.

[Sch06] Scheid, Harald: *Zahlentheorie.* Spektrum Akademischer Verlag, 4. Auflage, 2006.

[Sch07] Schmeh, Klaus: *Codeknacker gegen Codemacher. Die faszinierende Geschichte der Verschlüsselung.* W3L Verlag Bochum, 2. Auflage, 2007.
Dieses Buch ist bis dato das aktuellste unter denen, die sich mit der Geschichte der Kryptographie beschäftigen.
Das Buch enthält auch eine kleine Sammlung gelöster und ungelöster Krypto-Rätsel. Eine der Challenges nutzt den „Doppelwürfel“ (doppelte Spaltentransposition) mit zwei langen Schlüsseln, die unterschiedlich sind.
Siehe die Challenge auf MTC3: `https://www.mysterytwisterc3.org/de/challenges/level-x-kryptographie-challenges/doppelwuerfel` .

[Sch16a] Schmeh, Klaus: *Kryptographie – Verfahren, Protokolle, Infrastrukturen.* dpunkt.verlag, 6. Auflage, 2016. Sehr gut lesbares, aktuelles und umfangreiches Buch über Kryptographie. Geht auch auf praktische Probleme (wie Standardisierung oder real existierende Software) ein.

[Sch16b] Schmidt, Jürgen: *Kryptographie in der IT – Empfehlungen zu Verschlüsselung und Verfahren.* c't, 2016(1), 2016.
Dieser Artikel erschien ursprünglich in c't 01/2016, Seite 174. Danach veröffentlicht am 17.06.2016 in:
`http://www.heise.de/security/artikel/Kryptographie-in-der-IT-Empfehlungen-zu-Verschluesselung-und-Verfahren-3221002.html`.

[Sec02] Security, RSA: *Has the RSA algorithm been compromised as a result of Bernstein's Paper?* Technischer Bericht, RSA Security, April 2002. `http://www.emc.com/emc-plus/rsa-labs/historical/has-the-rsa-algorithm-been-compromised.htm`.

[Sed90] Sedgewick, Robert: *Algorithms in C.* Addison-Wesley, 1990.

[Seg04] Segers, A. J. M.: *Algebraic Attacks from a Gröbner Basis Perspective.* Diplomarbeit, Technische Universiteit Eindhoven, 2004.
`http://www.win.tue.nl/~henkvt/images/ReportSegersGB2-11-04.pdf`.

[Sem98] Semaev, I.: *Evaluation of Discrete Logarithms on Some Elliptic Curves.* Mathematics of Computation 67, 1998.

[Sem04] Semaev, Igor: *Summation polynomials and the discrete logarithm problem on elliptic curves.* IACR Cryptology ePrint Archive, 2004:31, 2004.

[Sha82] Shamir, A.: *A polynomial time algorithm for breaking the basic Merkle-Hellman Cryptosystem.* In: *Symposium on Foundations of Computer Science*, Seiten 145–152, 1982.

[Sho94] Shor, Peter W.: *Algorithms for Quantum Computation: Discrete Logarithms and Factoring.* In: *FOCS*, Seiten 124–134, 1994.

[Sho97a] Shor, Peter W.: *Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer.* SIAM Journal on Computing, 26(5):1484–1509, 1997.

[Sho97b] Shoup, Victor: *Lower Bounds for Discrete Logarithms and Related Problems.* In: *EUROCRYPT*, Seiten 256–266, 1997.

[Sho08] Shoup, Victor: *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, 2. Auflage, 2008. `http://shoup.net/ntb/`.

[Sil99] Silverman, Joseph H.: *The Xedni Calculus And The Elliptic Curve Discrete Logarithm Problem.* Designs, Codes and Cryptography, 20:5–40, 1999.

[Sil00] Silverman, Robert D.: *A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths.* RSA Laboratories Bulletin, 13:1–22, April 2000.

[Sil09] Silverman, Joe: *The Arithmetic of Elliptic Curves.* Graduate Texts in Mathematics 106. Springer, 2. Auflage, 2009, ISBN 978-0-387-09493-9.

[Sin99] Singh, Simon: *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography.* Anchor, 1999.

[Sin01] Singh, Simon: *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet.* dtv, 2001.

[SL07] Stamp, Mark und Richard M. Low: *Applied Cryptanalysis: Breaking Ciphers in the Real World.* Wiley-IEEE Press, 2007.
`http://cs.sjsu.edu/faculty/stamp/crypto/`.

[Sma99] Smart, N.: *The Discrete Logarithm Problem on Elliptic Curves of Trace One.* Journal of Cryptology 12, 1999.

[ST92] Silverman, J. und J. Tate: *Rational Points on Elliptic Curves.* Springer, 1992.

[ST03a] Shamir, Adi und Eran Tromer: *Factoring Large Numbers with the TWIRL Device*, 2003. `http://www.tau.ac.il/~tromer/papers/twirl.pdf`.

[ST03b] Shamir, Adi und Eran Tromer: *On the Cost of Factoring RSA-1024.* RSA Laboratories CryptoBytes, 6(2):11–20, 2003.
`http://www.tau.ac.il/~tromer/papers/cbtwirl.pdf`.

[Sta11] Stamp, Mark: *Information Security: Principles and Practice.* Wiley, 2. Auflage, 2011.

[Sta14] Stallings, William: *Cryptography and Network Security.* Pearson, 2014.
`http://williamstallings.com/Cryptography/`.

[Sti06] Stinson, Douglas R.: *Cryptography – Theory and Practice.* Chapman & Hall/CRC, 3. Auflage, 2006.

[SW10] Schulz, Ralph Hardo und Helmut Witten: *Zeitexperimente zur Faktorisierung. Ein Beitrag zur Didaktik der Kryptographie.* LOG IN, 166/167:113–120, 2010.
`http://bscw.schule.de/pub/bscw.cgi/d864899/Schulz_Witten_Zeit-Experimente.pdf`.

[Swe08] Swenson, Christopher: *Modern Cryptanalysis: Techniques for Advanced Code Breaking.* Wiley, 2008.

[SWE15] Schulz, Ralph Hardo, Helmut Witten und Bernhard Esslinger: *Rechnen mit Punkten einer elliptischen Kurve*. LOG IN, 2015(181/182):103–115, 2015. Geschrieben für Lehrer; didaktisch aufbereitet, leicht verständlich, mit vielen SageMath-Beispielen. `http://bscw.schule.de/pub/nj_bscw.cgi/d1024028/Schulz_Witten_Esslinger-Rechnen_mit_Punkten_einer_elliptischen_Kurve.pdf`.

[Thi99] ThinkQuest Team 27158: *Data Encryption*, 1999.

[Tie73] Tietze, H.: *Gelöste und ungelöste mathematische Probleme*. C.H. Beck, 6. Auflage, 1973.

[vzGG99] Gathen, Joachim von zur und Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 1999.

[Was08] Washington, Lawrence C.: *Elliptic Curves: Number Theory and Cryptography*. Discrete Mathematics and its Applications. Chapman and Hall/CRC, 2008, ISBN 9781420071467.

[Wel01] Welschenbach, Michael: *Kryptographie in C und C++*. Springer, 2001.

[Wika] Wikipedia: *Homomorphic Encryption & Homomorphismus*. `https://en.wikipedia.org/wiki/Homomorphic_encryption`, `https://de.wikipedia.org/wiki/Homomorphismus`.

[Wikb] Wikipedia: *Secure Multiparty Computation*. `http://en.wikipedia.org/wiki/Secure_multi-party_computation`.

[Wil95] Wiles, Andrew: *Modular elliptic curves and fermat's last theorem*. Annals of Mathematics, 141, 1995.

[WLB03] Weis, Rüdiger, Stefan Lucks und Andreas Bogk: *Sicherheit von 1024 bit RSA-Schlüsseln gefährdet*. Datenschutz und Datensicherheit (DuD), 27(6):360–362, 2003.

[WLS98] Witten, Helmut, Irmgard Letzner und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle, Teil 1: Sprache und Statistik*. LOG IN, 1998(3/4):57–65, 1998. `http://bscw.schule.de/pub/bscw.cgi/d637160/RSA_u_Co_T1.pdf`.

[WLS99] Witten, Helmut, Irmgard Letzner und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. Teil 3: Flusschiffren, perfekte Sicherheit und Zufall per Computer*. LOG IN, 1999(2):50–57, 1999. `http://bscw.schule.de/pub/nj_bscw.cgi/d637156/RSA_u_Co_T3.pdf`.

[Wob02] Wobst, Reinhard: *Angekratzt – Kryptoanalyse von AES schreitet voran*. iX, Dezember 2002. (Und der Leserbrief dazu von Johannes Merkle in der iX 2/2003).

[Wob05] Wobst, Reinhard: *New Attacks Against Hash Functions*. Information Security Bulletin, April 2005.

[WS06] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 2: RSA für große Zahlen*. LOG IN, 2006(143):50–58, 2006. `http://bscw.schule.de/pub/bscw.cgi/d404410/RSA_u_Co_NF2.pdf`.

[WS08] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 3: RSA und die elementare Zahlentheorie.* LOG IN, 2008(152):60–70, 2008.
http://bscw.schule.de/pub/nj_bscw.cgi/d533821/RSA_u_Co_NF3.pdf.

[WS10a] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 4: Gibt es genügend Primzahlen für RSA?* LOG IN, 2010(163):97–103, 2010.
http://bscw.schule.de/pub/nj_bscw.cgi/d864891/RSA_u_Co_NF4.pdf.

[WS10b] Witten, Helmut und Ralph Hardo Schulz: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. NF Teil 5: Der Miller-Rabin-Primzahltest oder: Falltüren für RSA mit Primzahlen aus Monte Carlo.* LOG IN, 2010(166/167):92–106, 2010.
http://bscw.schule.de/pub/nj_bscw.cgi/d864895/RSA_u_Co_NF5.pdf.

[WSE15] Witten, Helmut, Ralph Hardo Schulz und Bernhard Esslinger: *RSA & Co. in der Schule: Moderne Kryptologie, alte Mathematik, raffinierte Protokolle, NF Teil 7: Alternativen zu RSA oder Diskreter Logarithmus statt Faktorisierung.* LOG IN, 2010(181-182):85–102, 2015.
Hierin werden u.a. DH und Elgamal in einem breiteren Kontext behandelt. Die Verfahren werden mit Codebeispielen in Python und SageMath erläutert.
http://bscw.schule.de/pub/nj_bscw.cgi/d1024013/RSA_u_Co_NF7.pdf,
http://www.log-in-verlag.de/wp-content/uploads/2015/07/Internetquellen-LOG-IN-Heft-Nr.181-182.doc.

[WYY05a] Wang, Xiaoyun, Andrew Yao und Frances Yao: *New Collision Search for SHA-1.* Technischer Bericht, Crypto 2005, Rump Session, 2005.
http://www.iacr.org/conferences/crypto2005/rumpSchedule.html.

[WYY05b] Wang, Xiaoyun, Yiqun Yin und Hongbo Yu: *Finding Collisions in the Full SHA-1.* Advances in Cryptology-Crypto, LNCS 3621, Seiten 17–36, 2005.

[Yan00] Yan, Song Y.: *Number Theory for Computing.* Springer, 2000.

[YY96] Young, Adam L. und Moti Yung: *The Dark Side of Black-Box Cryptography, or: Should We Trust Capstone?* In: *CRYPTO*, Seiten 89–103, 1996.

[YY97] Young, Adam L. und Moti Yung: *Kleptography: Using Cryptography against Cryptography.* In: *EUROCRYPT*, Seiten 62–74, 1997.

Index