Jörg Keller und Stefan Wohlfeil

Sicherheit im Internet

Kurseinheit 2:
Verschlüsselung und digitale Signaturen

mathematik
und
informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Diese Seite bleibt aus technischen Gründen frei!

Kapitel 2

Verschlüsselung und digitale Signaturen

2.1 Einführung

Der Zweck einer Verschlüsselung (engl. **encryption**) besteht darin, aus einer offenen, für jeden lesbaren und verständlichen Nachricht, auch Klartext (engl. **plain text**) genannt, mit Hilfe eines Verschlüsselungsalgorithmus eine verschlüsselte Nachricht, auch Geheimtext (engl. **cipher text**) genannt, zu erstellen. Der Inhalt der verschlüsselten Nachricht lässt dabei im Idealfall keine Rückschlüsse auf den Inhalt der unverschlüsselten Nachricht zu. Selbst wenn man die verschlüsselte Nachricht mitlesen kann, versteht man nichts vom eigentlichen Inhalt. Der Verschlüsselungsalgorithmus wird durch einen Parameter, genannt Schlüssel (engl. **key**), gesteuert.

Klartext

Geheimtext

Schlüssel

Mit Entschlüsselung (engl. **decryption**) bezeichnet man die umgekehrte Transformation. Aus einer verschlüsselten Nachricht entsteht mit Hilfe eines Entschlüsselungsalgorithmus, gesteuert durch einen Schlüssel, wieder die ursprüngliche Nachricht. Verschlüsselungs- und Entschlüsselungsalgorithmus können gleich sein, müssen es aber nicht. Dasselbe gilt für die Schlüssel. Von einem

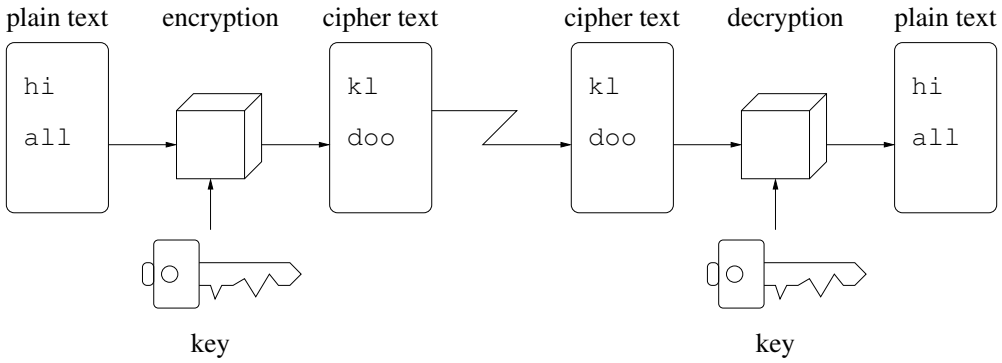

Abbildung 2.1: Prinzip der verschlüsselten Übertragung

vernünftigen Verschlüsselungsalgorithmus erwartet man, dass die Entschlüsselung *ohne* Kenntnis des Schlüssels nicht bzw. nur mit erheblichem Aufwand möglich ist. Unter erheblichem Aufwand versteht man Aufwand, der heute auch für Organisationen, denen sehr viele Ressourcen zur Verfügung stehen, nicht in vertretbarer Zeit machbar ist. Außerdem erwartet man, dass auch in mittelfristiger Zukunft niemand genug Ressourcen zur Verfügung haben wird,

um eine Nachricht ohne Kenntnis des Schlüssels zu entschlüsseln. Algorithmisch betrachtet erwartet man, dass die Entschlüsselung ohne Kenntnis des Schlüssels einen Aufwand erfordert, der exponentiell in der Länge des Schlüssels ist.

Abbildung 2.1 zeigt das Prinzip. Die Übertragung einer Nachricht erfolgt im Internet i. d. R. über mehrere Stationen. Man kann die Verschlüsselung daher unterschiedlich einsetzen:

Leitungsverschlüsselung: Zwischen je zwei Computern auf dem Weg wird die Nachricht verschlüsselt. Der Absender verschlüsselt für den ersten Empfänger, der entschlüsselt und verschlüsselt anschließend für den nächsten Empfänger usw.

Leitungsverschlüsselung

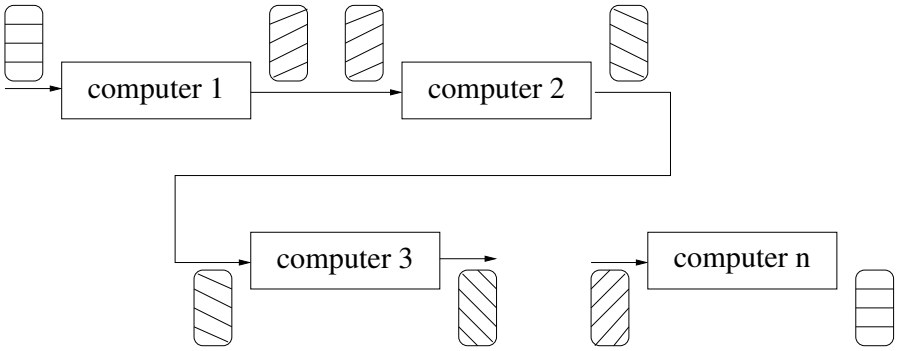

Abbildung 2.2: Leitungsverschlüsselung

Abbildung 2.2 zeigt, dass die Nachricht immer wieder entschlüsselt und mit einem neuen Schlüssel verschlüsselt wird.

Hauptvorteil dieser Methode ist, dass sich immer nur die beiden direkten „Nachbarn“ auf einen Verschlüsselungsalgorithmus und einen Schlüssel einigen müssen. Man kann diese Methode daher auch auf den niedrigen Ebenen der Protokollhierarchie einsetzen.

Ein Nachteil ist, dass *alle* Computer auf dem Weg sicher und vertrauenswürdig sein müssen.

Ende-zu-Ende-Verschlüsselung: Der Absender verschlüsselt bei der Ende-zu-Ende-Verschlüsselung die Nachricht und diese verschlüsselte Nachricht wird dann von Computer zu Computer unverändert[1] übertragen. Erst der Empfänger entschlüsselt die Nachricht wieder.

Ende-zu-Ende-Verschlüsselung

Abbildung 2.3 zeigt, dass nur der Absender verschlüsselt, alle Übertragungscomputer nur weiterleiten und erst der Empfänger wieder entschlüsselt.

Hauptvorteil dieser Methode ist, dass keiner der Computer auf dem Weg die Nachricht im Klartext sieht. Der Absender muss sich jetzt aber mit *jedem* möglichen Empfänger auf ein Verschlüsselungsverfahren und einen Schlüssel einigen.

Wegen der größeren Sicherheit wird man als Anwender lieber auf die Ende-zu-Ende-Verschlüsselung zurückgreifen wollen. Man muss sich dann nicht auf Leitungsbetreiber und deren Vertrauenswürdigkeit verlassen.

[1] Abgesehen von bestimmten Statistik- oder Debugging-Daten, die bei der Weiterleitung evtl. angehängt werden.

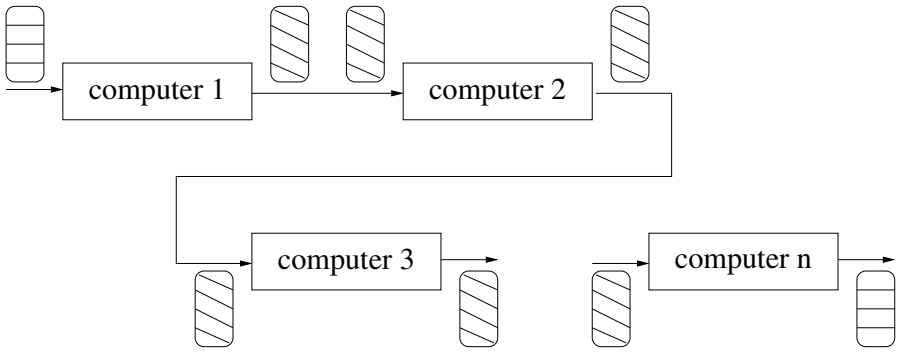

Abbildung 2.3: Ende-zu-Ende-Verschlüsselung

Verschlüsselungsverfahren können anhand der folgenden Kriterien klassifiziert werden:

- Verschlüsselungsoperationen
- Anzahl der Schlüssel
- Verarbeitung des Klartexts

Verschlüsselungsoperationen: Hier geht es darum, mit welchen Grundoperationen ein Verschlüsselungsalgorithmus die Abbildung vom Klartext auf den verschlüsselten Text vornimmt. Grundoperationen sind:

Verschlüsselungsoperationen

Ersetzung (engl. substitution): Bei diesem Verfahren wird jedes Zeichen (oder jede Zeichengruppe) des Klartexts auf ein bestimmtes Zeichen (bzw. eine Zeichengruppe) des verschlüsselten Textes abgebildet. Mathematisch betrachtet handelt es sich hierbei um eine bijektive Funktion. Eingabe- und Ausgabemenge können verschieden sein (z. B. beim Morse-Code). Bei Verschlüsselungsverfahren für Computer sind die beiden Mengen aber i. d. R. identisch.

Beispiel: Ersetze A durch B, B durch C, ..., Y durch Z und Z durch A.

Umordnung (engl. transposition): Hierbei werden die Zeichen des Klartexts neu angeordnet, d. h. ihre Reihenfolge wird verändert. Mathematisch betrachtet handelt es sich also um eine Permutation.

Beispiel: Nachrichten von hinten nach vorne aufschreiben.

Dies sind die Grundoperationen, die den typischen Verschlüsselungsverfahren zugrunde liegen. Ein Algorithmus kann diese Operationen kombinieren und dadurch eine bessere Verschlüsselung erreichen.

Anzahl der Schlüssel: Da man zum Verschlüsseln und Entschlüsseln jeweils einen Schlüssel braucht, stellt sich die Frage, ob das immer derselbe Schlüssel sein muss.

Anzahl der Schlüssel

Ein Schlüssel: Bei diesem Verfahren wird der Schlüssel, mit dem die Nachricht verschlüsselt wurde, auch zur Entschlüsselung benutzt. Man bezeichnet das Verfahren auch als **symmetrische Verschlüsselung** oder **Private-Key**-Verfahren. Da der Schlüssel geheim gehalten werden muss, nennt man ihn manchmal auch **Secret Key**. Auf diese Verfahren wird in Abschnitt 2.3 näher eingegangen.

symmetrische Verschlüsselung

Private Key

Secret Key

Mehrere Schlüssel: Bei diesem Verfahren werden verschiedene Schlüssel zum Verschlüsseln und Entschlüsseln eingesetzt. Da einer der beiden Schlüssel (i. d. R. der zum Verschlüsseln) problemlos öffentlich bekannt sein darf, nennt man dieses Verfahren auch **Public-Key**-Verfahren oder auch **asymmetrische Verschlüsselung**. Der andere Schlüssel (i. d. R. zum Entschlüsseln) wird auch **Private Key** genannt. Er ist nicht oder nur mit sehr hohem Aufwand aus dem öffentlichen Schlüssel berechenbar. Auf diese Verfahren wird in Abschnitt 2.4 näher eingegangen.

Public Key
asymmetrische Verschlüsselung
Private Key

Allgemein gilt, dass Secret-Key-Verfahren weniger Aufwand erfordern als Public-Key-Verfahren. Man kombiniert diese Verfahren daher gerne, indem man zunächst einen geheimen Schlüssel generiert und diesen Schlüssel mit einem Public-Key-Verfahren verschlüsselt an den Empfänger überträgt. Anschließend verschlüsselt man die Nachricht selbst symmetrisch mit dem gerade übertragenen geheimen Schlüssel. Solche Verfahren nennt man *hybride Verschlüsselungsverfahren.*

hybride Verschlüsselungsverfahren

Verarbeitung des Klartexts: Der zu verschickende Klartext kann normalerweise nicht „als Ganzes“ verschlüsselt werden, denn man weiß im voraus ja nicht, wie groß der Klartext sein wird. Für Verschlüsselungsverfahren gibt es daher die folgenden Verarbeitungsmodi:

Verarbeitungsmodi

Blockverschlüsselung: Bei der Blockverschlüsselung wird der zu verschlüsselnde Klartext in Blöcke fester Größe eingeteilt. Früher waren dies häufig 64 Bit große Blöcke. Eine heute gerne gewählte Blockgröße ist 128 Bit. Jeder der Blöcke wird für sich verschlüsselt und ergibt dann einen Geheimtextblock, i. d. R. mit derselben Größe (siehe dazu auch Abschnitt 2.3.6).

Ist der Klartext nicht ein Vielfaches der Blockgröße lang, so wird der Klartext durch bestimmte Zeichen aufgefüllt. Damit wird der letzte Block künstlich auch auf die erforderliche Größe gebracht. Man nennt das auch *padding.*

Stromverschlüsselung: In diesem Fall wird die Klartextnachricht als Folge oder Strom (engl. **stream**) von Bits oder auch Zeichen (also Bytes) betrachtet. Ein ankommendes Klartextzeichen wird verschlüsselt und ausgegeben, bevor das nächste Klartextzeichen ankommt.

Man bezeichnet einen solchen Algorithmus auch als **Online-Algorithmus**. Ein Online-Algorithmus erledigt seine Aufgabe *ohne* Kenntnis der Zukunft, in diesem Beispiel also ohne Kenntnis der nachfolgenden Zeichen.

Online-Algorithmus

Ein Vorteil dieser Methode ist es, dass eine Klartextnachricht nicht aufgefüllt werden muss, so dass tatsächlich nur Nutzdaten übertragen werden. Die Leitungskapazität wird besser ausgenutzt. Der Nachteil von Online-Algorithmen ist allerdings, dass sie ohne Kenntnis der Zukunft nicht immer optimale Ergebnisse berechnen.

2.2 Hardware- und Betriebssystemsicherheit

Hardware: Die Sicherheit eines Computers hängt neben dem Computer selbst auch von der Sicherheit der Umgebung ab. Eine sichere Umgebung

beginnt mit einem sicheren Gebäude, in dem der Computer untergebracht ist. Die zu betrachtenden Komponenten des Gebäudes sind

- das Bauwerk selbst,
- die Versorgungsleitungen.

Das Bauwerk sollte vor Brandgefahr, Blitzeinschlag, Vandalismus usw. schützen. Potentielle Gefahren bei Versorgungsleitungen sind Stromausfälle oder anderweitige Netzstörungen.

Innerhalb eines Gebäudes sind Computer in Räumen untergebracht. Die Räume müssen die Schutzfunktionen des Gebäudes ergänzen. Man unterscheidet

- Büroräume,
- Serverräume und
- Datenträgerräume (Datenarchiv).

Datenträgerräume und Serverräume sollten im Gegensatz zu Büroräumen *nicht* frei zugänglich sein. Beschädigungen oder Diebstahl von Computern oder Daten wird so vermieden bzw. erschwert.

Sicherheitsaspekte sind dann auch am Computer selbst, konkret am Gehäuse zu beachten. Abschließbare Gehäuse verhindern beispielsweise den Ausbau und damit den Diebstahl von Festplatten und deren Daten. Weiterhin sollte ein Gehäuse auch vor „Unfällen“ wie Wassereinbruch beim Blumengießen oder Reinigen schützen. Alle diese Maßnahmen stellen sicher, dass die Hardware für den bestimmungsgemäßen Gebrauch zur Verfügung steht.

Betriebssystem: Neben der Hardware muss als nächstes auch die Software, konkret das Betriebssystem, gewisse Anforderungen erfüllen. Diese werden im Kurs *(01802) Betriebssysteme* genauer diskutiert. Da sie im Kurs *(01801) Betriebssysteme und Rechnernetze* nicht behandelt werden, erfolgt hier eine kurze Einführung in die wichtigsten Anforderungen.

Benutzerkennungen

Die erste Anforderung ist, dass Benutzerkennungen (engl. **account**) vorgesehen sind. Damit ist keine anonyme Benutzung des Computers möglich. Gleichzeitig ist das die Grundlage für die Vergabe von Rollen wie Administrator, Entwickler, Anwender usw. Mit diesen Rollen werden dann Rechte verbunden.

Zugriffsrechte

Im Wesentlichen geht es dabei um Zugriffsrechte zu bestimmten Objekten, wie beispielsweise Daten oder Programmen. Die Rechte beziehen sich auf Operationen auf den Objekten. Typische Operationen sind: Lesen, Schreiben, Ausführen und Rechte vergeben. Moderne Betriebssysteme wie Linux, MS Windows oder Mac OS X enthalten diese Konzepte. In Kurs *(01868) Sicherheit im Internet 1 – Ergänzungen* werden Zugriffskontrollen genauer besprochen.

Speicherschutz

Neben persistenten Objekten, wie Dateien, sind auch transiente Objekte, wie Hauptspeicherbereiche, zu schützen. Persistente Objekte sind dauerhaft im System vorhanden und auch nach dem Aus- und Wiedereinschalten noch da. Transiente Objekte sind nur temporär vorhanden und überstehen einen Neustart des Systems nicht. Ohne diesen Schutz könnte ein Programm den Datenbereich aus dem Adressraum eines anderen Programms lesen und dort wichtige Informationen (z. B. Passwörter) ausspionieren. In modernen Betriebssystemen sind auch solche Schutzmechanismen vorgesehen.

kein direkter Hardwarezugriff

Weiterhin muss das Betriebssystem auch den direkten Zugriff auf die Hardware verhindern. Ein „normaler“ Anwender sollte beispielsweise keine eigenen Interrupt-Routinen schreiben und auf einem Computer installieren können. Ein „Spionage-Interrupthandler“ im Tastatur-Interrupt kann *jeden* Tastendruck registrieren. Beachten sie hierbei, dass Sie Ihr Passwort bei der Authentifikation über die Tastatur eingeben.

Sie können mehr zu diesen Anforderungen in nahezu jedem Buch über Betriebssysteme nachlesen, beispielsweise bei Tanenbaum [Tan02] oder Silberschatz et. al. [SGG08].

Diese Maßnahmen und Anforderungen sind eine Grundlage, ohne die auch die besten Verschlüsselungsverfahren sinnlos sind. Ihr geheimer Schlüssel muss schließlich auf irgend einem Weg in den Computer kommen. Das erfolgt i. d. R. über die Tastatur. Zusätzlich ist er (symmetrisch mit einem Passwort verschlüsselt) evtl. auch in einer Datei gespeichert. Wird der Schlüssel auf diesem Weg ausgespäht, so ist jede Verschlüsselung sinnlos. Falls die Programmierer eines Betriebssystems eine „Hintertür“ einprogrammiert haben, dann könnten Ihre geheimen Schlüssel möglicherweise an Dritte geschickt oder von Dritten abgerufen werden. Diese Dritten könnten dann alle Ihre verschlüsselten Nachrichten mitlesen.

2.3 Secret-Key-Verschlüsselung

2.3.1 Prinzip der Secret-Key-Verschlüsselung

symmetrische Verfahren

Secret-Key-Verschlüsselungsverfahren werden auch symmetrische Verfahren genannt. Sender und Empfänger müssen dabei (einmalig[2]) einen geheimen Schlüssel austauschen. Mit diesem Schlüssel können dann Nachrichten verschlüsselt *und* entschlüsselt werden. Jeder der Kommunikationspartner kann also senden und empfangen, ihre Rollen können beliebig vertauscht werden.

In Abschnitt 2.3.2 werden einige klassische Verfahren vorgestellt, die auch schon vor dem Computer-Zeitalter eingesetzt wurden. Diese Verfahren sind nicht nur von historischem Interesse, sondern einige der Grundprinzipien finden sich auch in den heute aktuellen Verfahren wieder. In Abschnitt 2.3.3 werden dann einige der aktuellen Verfahren kurz vorgestellt.

2.3.2 Klassische Verschlüsselungsalgorithmen

Cäsar-Chiffre: Eine einfache Substitutionschiffre wurde bereits vom römischen Kaiser Cäsar benutzt. Die Vertauschung beruht auf einer einfachen Verschiebung jedes Klartextzeichens um drei Zeichen. ‚a‘ wird zu ‚d‘ ‚b‘ zu ‚e‘ usw. Kommt man am Ende des Alphabets an, so beginnt die Ersetzung wieder von vorne, d. h. ‚x‘ wird zu ‚a‘, ‚y‘ zu ‚b‘ und ‚z‘ zu ‚c‘. Die folgende Tabelle zeigt die Zuordnung.

Klartextzeichen:	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
Chiffrezeichen:	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

[2] In der Praxis wird man diese Schlüssel trotzdem regelmäßig wechseln, um zu verhindern, dass der Schlüssel „geknackt“ werden kann. Außerdem reduziert man dadurch die Menge der Informationen, die einem Angreifer nach einem erfolgreichen Angriff zur Verfügung stehen.

Da Zeichen im Computer als Zahlen dargestellt werden, lässt sich die Verschiebung sehr einfach berechnen. Falls das ‚a' durch 0, ‚b' durch 1, ..., ‚z' durch 25 dargestellt werden, dann berechnet sich das Chiffrezeichen eines Klartextzeichens x nach folgender Formel:

$$\text{Chiffre}(x) = (x + 3) \bmod 26$$

Dabei bezeichnet *mod* den Rest bei der ganzzahligen Division. Die Verschiebedistanz (bei Cäsar war es drei) ist der Verschlüsselungsschlüssel k_v. Den Entschlüsselungsschlüssel k_e kann man aus der Zahl der Zeichen n (im Alphabet also 26) und dem Verschlüsselungsschlüssel berechnen. Es gilt $k_e = n - k_v$. Verschiebt man die einzelnen Zeichen nämlich um die Gesamtzahl der Zeichen, so steht man wieder am Anfang. Man kann das mit einer analogen Uhr vergleichen. Dreht man den Zeiger um 12 Stunden weiter, so hat man wieder die ursprüngliche Zeit.

Übungsaufgabe 2.1 *Entschlüsseln Sie folgenden Geheimtext, der mit einer Cäsar-Chiffre erzeugt wurde:*

`zrlvwehkoh`

Dieses einfache Verschlüsselungsverfahren wird heute noch im Internet eingesetzt. Möchte man einen Artikel in eine Newsgruppe schreiben, der möglicherweise einige Leser verletzen oder beleidigen könnte (typisches Beispiel sind Witze über bestimmte Personengruppen), so „verschlüsselt" man den Artikel mit **ROT13**. ROT13 ist eine Cäsar-Chiffre mit Verschlüsselungsschlüssel $k_v = 13$. Da das Alphabet aus 26 Zeichen besteht, entspricht dies genau einer paarweisen Vertauschung. Der Entschlüsselungsschlüssel ist $k_e = 26 - k_v = 13 = k_v$. Bei ROT13 werden nur die Großbuchstaben und die Kleinbuchstaben untereinander vertauscht. Alle anderen Zeichen bleiben unverändert. ROT13

Man kann diese Verschlüsselungsmethode nicht nur auf Buchstaben, sondern auf alle Zeichen anwenden. Zeichen werden im Computer durch ein Byte dargestellt, also durch eine Zahl zwischen 0 und 255. Man kann also allgemein jedes Zeichen wie folgt verschlüsseln

$$\text{Chiffre}_k(x) = (x + k) \bmod 256$$

Echte Sicherheit kann man durch diese Verschlüsselung allerdings nicht erreichen. Auch ohne die Kenntnis des Schlüssels kann ein Angreifer einfach alle 26 (bzw. 256) möglichen Verschiebungen ausprobieren. Den richtigen Schlüssel hat man dann sehr schnell gefunden.

Aber auch ohne das Ausprobieren aller Verschiebungen kann man die Verschlüsselung einfach rückgängig machen. Ist die verschlüsselte Nachricht beispielsweise auf Deutsch, so kann man diese Information benutzen. Bei der Cäsar-Chiffre wird jedes Klartextzeichen immer auf dasselbe Chiffrezeichen abgebildet. Da in deutschen Texten die einzelnen Buchstaben unterschiedlich häufig vorkommen (das ‚e' kommt am häufigsten vor), kann man vermuten, dass das Zeichen, das am häufigsten im Chiffretext vorkommt, das Ersatzzeichen für das ‚e' sein wird. Dann kann man die Verschiebelänge einfach ausrechnen und nur diese Verschiebung ausprobieren. Bei hinreichend langen Texten hat man damit i. d. R. Erfolg.

Übungsaufgabe 2.2 *Entschlüsseln Sie folgenden Geheimtext:*

`oxdcmrveoccovxcsopyvqoxnoxqoroswdohd`

Monoalphabetische Chiffre: Eine erste Verbesserung der Cäsar-Chiffre besteht darin, nicht mehr die Reihenfolge der Ersetzungszeichen vorzuschreiben.

Permutation

Man erlaubt in der Ersetzungstabelle jede Permutation der Buchstaben in der Zeile „Chiffrezeichen“. Der Schlüssel dieses Verfahrens ist nicht mehr die Länge der Verschiebung, sondern die zweite Zeile der Ersetzungstabelle. Ein Beispiel ist:

Klartextzeichen:	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
Chiffrezeichen:	x	e	o	i	h	u	j	l	m	k	n	f	p	a	r	c	t	g	v	w	d	y	z	q	b	s

Möchte man einen mit diesem Verfahren verschlüsselten Text *ohne* Kenntnis des Schlüssels entschlüsseln, kann man wieder mit Häufigkeitsanalysen arbeiten. Es reicht allerdings nicht mehr, den häufigsten Buchstaben zu finden, sondern man muss alle Buchstaben betrachten.

Weiterhin kann man die Analyse auch auf Buchstabenpaare oder -tripel ausdehnen. Dabei nutzt man aus, dass bestimmte Buchstabenfolgen, wie ‚ch‘ oder ‚ck‘, häufiger vorkommen als andere, wie ‚cq‘ oder ‚cw‘. Ein hinreichend langer Text kann auf diese Art ohne viel Aufwand entschlüsselt werden. Ursache hierfür ist, dass ein Klartextzeichen immer durch dasselbe Chiffrezeichen ersetzt wird, egal wo es im Klartext vorkommt.

Polyalphabetische Chiffre: Die nächste Verbesserung sorgt dafür, dass ein Klartextzeichen *nicht* immer durch dasselbe Chiffrezeichen ersetzt wird. Solche Verschlüsselungsverfahren nennt man polyalphabetische Chiffre. Ein klassisches

Vigenère-Chiffre

Beispiel ist die sogenannte **Vigenère-Chiffre**. Die Idee hierbei ist, dass auf jedes Klartextzeichen eine Verschiebung angewendet wird. Die Verschiebelänge ist allerdings nicht konstant, sondern sie wird durch ein Schlüsselwort definiert. Jeder Buchstabe des Schlüsselwortes wird als Zahl interpretiert. Für das Schlüsselwort *secret* ergibt sich folgende Verschiebetabelle:

s	e	c	r	e	t
18	4	2	17	4	19

Das erste Zeichen des Klartexts wird um 18 Zeichen verschoben, das zweite Zeichen um 4, das Dritte um 2 usw. Hat man alle Verschiebungen des Schlüssels durchgeführt und es sind noch weitere Klartextzeichen zu verschlüsseln, so beginnt man wieder mit dem Anfang des Schlüssels. Ein Beispiel:

Klartext	d	a	s	i	s	t	g	e	h	e	i	m
Schlüssel	s	e	c	r	e	t	s	e	c	r	e	t
Länge	18	4	2	17	4	19	18	4	2	17	4	19
Chiffre	v	e	u	z	w	m	y	i	j	v	m	f

Beachten Sie, dass die beiden ‚s‘ und ‚e‘ im Klartext auf verschiedene Zeichen im Chiffretext abgebildet werden. Ein Angriff auf diese Chiffre versucht zunächst die Länge des Schlüssels zu ermitteln. Im obigen Beispiel ist die Schlüssellänge 6. Anschließend muss man die 6 Verschiebelängen ermitteln. Dazu können wieder dieselben statistischen Techniken wie bei der Cäsar-Chiffre benutzt werden.

Enigma

In der im Zweiten Weltkrieg benutzten Verschlüsselungsmaschine *Enigma* wurde übrigens auch eine polyalphabetische Verschlüsselung benutzt. An Stelle des Schlüsselwortes in der Vigenère-Chiffre wurde bei der Enigma eine elektromechanische Vorrichtung aus Walzen benutzt, um jedes Zeichen durch ein anderes Zeichen zu ersetzen. Details zu dieser Maschine finden sich beispielsweise bei Bauer [Bau97].

Transpositionsverschlüsselung: Alle in diesem Abschnitt bisher beschriebenen Verfahren beruhen darauf, dass ein Klartextzeichen durch ein Geheimtextzeichen ersetzt wird. Bei den Transpositions- oder Vertauschungsverfahren bleiben die Klartextzeichen jedoch erhalten. Allerdings wird ihre Reihenfolge verändert. In der Mathematik bezeichnet man solche Vertauschungen als **Permutation**. Auf einer Zeichenfolge der Länge n gibt es $n! = n \times (n-1) \times (n-2) \times \ldots \times 2$ Permutationen.

Permutation

Vertauscht man beispielsweise immer das erste und dritte Zeichen sowie das zweite und vierte Zeichen, so notiert man diese Permutation wie folgt: (1 3)(2 4). Ein Beispiel dieser Permutation:

Klartext	d	a	s	i	s	t	g	e	h	e	i	m
Geheimtext	s	i	d	a	g	e	s	t	i	m	h	e

Der Geheimtext entsteht aus dem Klartext, indem man die Permutation erst auf die ersten vier Zeichen anwendet, dann auf die nächsten vier Zeichen usw. Sollten am Ende nicht genau vier Zeichen übrig bleiben, so füllt man mit beliebigen Zeichen, z. B. ‚a', ‚b', ‚c' auf.

Man muss bei Permutationen nun nicht paarweise tauschen, sondern kann auch eine längere Permutation bilden. Die Permutation (1 4 2) (3) bedeutet: Das erste Zeichen wandert auf Position 4, das vierte Zeichen auf Position 2 und das zweite Zeichen auf Position 1. Das dritte Zeichen behält seinen Platz. Weiterhin können Permutationen auch auf größere Folgen als die gerade gezeigten 4-elementigen Folgen angewendet werden.

Der Schlüssel dieser Verschlüsselungsverfahren ist also die angewandte Permutation. Die mathematische Notation einer Permutation ist zwar knapp und eindeutig, jedoch ist sie nicht besonders anschaulich. Man hat daher früher Hilfsmittel wie Matrizen benutzt. Der Klartext wird zeilenweise in eine Matrix geschrieben. Der Geheimtext entsteht dann, wenn man die Matrix spaltenweise ausliest. Aus dem Klartext `dasistgeheim` entsteht beispielsweise folgende Matrix

d	a	s	i
s	t	g	e
h	e	i	m

und der Geheimtext

`dshatesgiiem`.

Der Schlüssel des Verfahrens ist dann die Zahl der Spalten der Matrix. Die Entschlüsselung erfolgt, indem der Geheimtext spaltenweise in eine Matrix eingetragen wird. Die Zahl der Zeilen ergibt sich aus der Länge des Geheimtexts und der Zahl der Spalten. Der Klartext kann dann zeilenweise ausgelesen werden.

Eine naheliegende Verbesserung dieses Verfahrens ergibt sich, wenn man die Spalten der Matrix vertauscht, bevor man den Geheimtext ausliest. Wendet man die Permutation (1 3)(2 4) auf die obige Matrix an, so ergibt sich der folgende Geheimtext

`sgiiemdshate`

Der Schlüssel des verbesserten Verfahrens besteht aus zwei Teilen, (1) der Zahl der Spalten und (2) der Permutation. Bei der Entschlüsselung geht man wieder umgekehrt vor. Zuerst wird der Geheimtext spaltenweise in die Matrix geschrieben, dann werden die Spalten zurück getauscht und dann kann der Klartext zeilenweise ausgelesen werden.

2.3.3 Moderne Verschlüsselungsalgorithmen

Grundoperationen: Moderne Verschlüsselungsverfahren bedienen sich derselben Techniken – Ersetzen und Vertauschen – wie die klassischen Verfahren. Die Realisierung der Ersetzen-Funktion erfolgt meistens durch die Funktion XOR (exklusiv-oder, mathematische Notation $\bigoplus$[3]). Sie ist auf x und y der Länge ein Bit durch die folgende Tabelle definiert.

$\bigoplus$	0	1
0	0	1
1	1	0

Zwei n Bit lange Wörter x und y werden bitweise verknüpft. Ein Beispiel:

$$\begin{array}{r} x = 1010 \\ y = 1100 \\ \hline x \bigoplus y = 0110 \end{array}$$

Die XOR-Funktion hat die schöne Eigenschaft, dass $((a \bigoplus b) \bigoplus b) = a$ gilt. Man kann also ein Wort a durch $a' = a \bigoplus b$ ersetzen und durch die gleiche Operation die Ersetzung rückgängig machen. Dazu wird $a' \bigoplus b$ berechnet.

Die Realisierung der Vertauschen-Funktion im Computer wird durch die Hardware direkt unterstützt. Die meisten Prozessoren erlauben das zirkuläre Verschieben (engl. **circular shift**) von Bits eines Wortes. Aus dem Wort $x = x_n x_{n-1} \ldots x_2 x_1$ der Länge n entsteht durch Verschieben um 1 Bit nach links das Wort $x' = x_{n-1} \ldots x_2 x_1 x_n$. Verschiebt man in einem Wort der Länge n (für gerades n) die Bits $n/2$ mal nach links oder rechts, so hat man gerade die vordere Hälfte mit der hinteren Hälfte vertauscht.

Feistel-Verfahren

Das Feistel-Verfahren: Horst Feistel hat schon 1973 ein Verschlüsselungsschema vorgeschlagen, das heute immer noch die Basis für viele aktuelle Verschlüsselungsalgorithmen ist. Die Idee dabei ist, die Operationen *Ersetzen* und *Vertauschen* mehrfach hintereinander zu benutzen. Abbildung 2.4 zeigt das Prinzip.

Ein Klartextwort der Länge g wird zunächst in ein linkes Teilwort L_0 und ein rechtes Teilwort R_0 jeweils der Länge $g/2$ aufgeteilt. Aus dem Schlüssel S (in Abbildung 2.4 nicht eingezeichnet) wird ein erster Teilschlüssel S_0 berechnet. Dieser Teilschlüssel steuert eine Funktion F, die auf R_0 angewandt wird. Das Ergebnis dieser Funktion R'_0 und das linke Teilwort L_0 werden dann mit XOR verknüpft. Zum Abschluss werden das Ergebnis von XOR und R_0 noch vertauscht.

Dieser Vorgang, genannt Runde, wird mehrmals hintereinander ausgeführt. Dabei wird für jede Runde i ein Teilschlüssel S_{i-1} generiert, der die Funktion F

[3] Die Schreibweise erinnert absichtlich an das Additionssymbol +, denn man kann XOR als „Addition ohne Übertrag“ auf einzelnen Bits interpretieren. Im binären Zahlensystem gilt ja $1 + 1 = 10$, das XOR liefert also die hintere Stelle des Additionsergebnis.

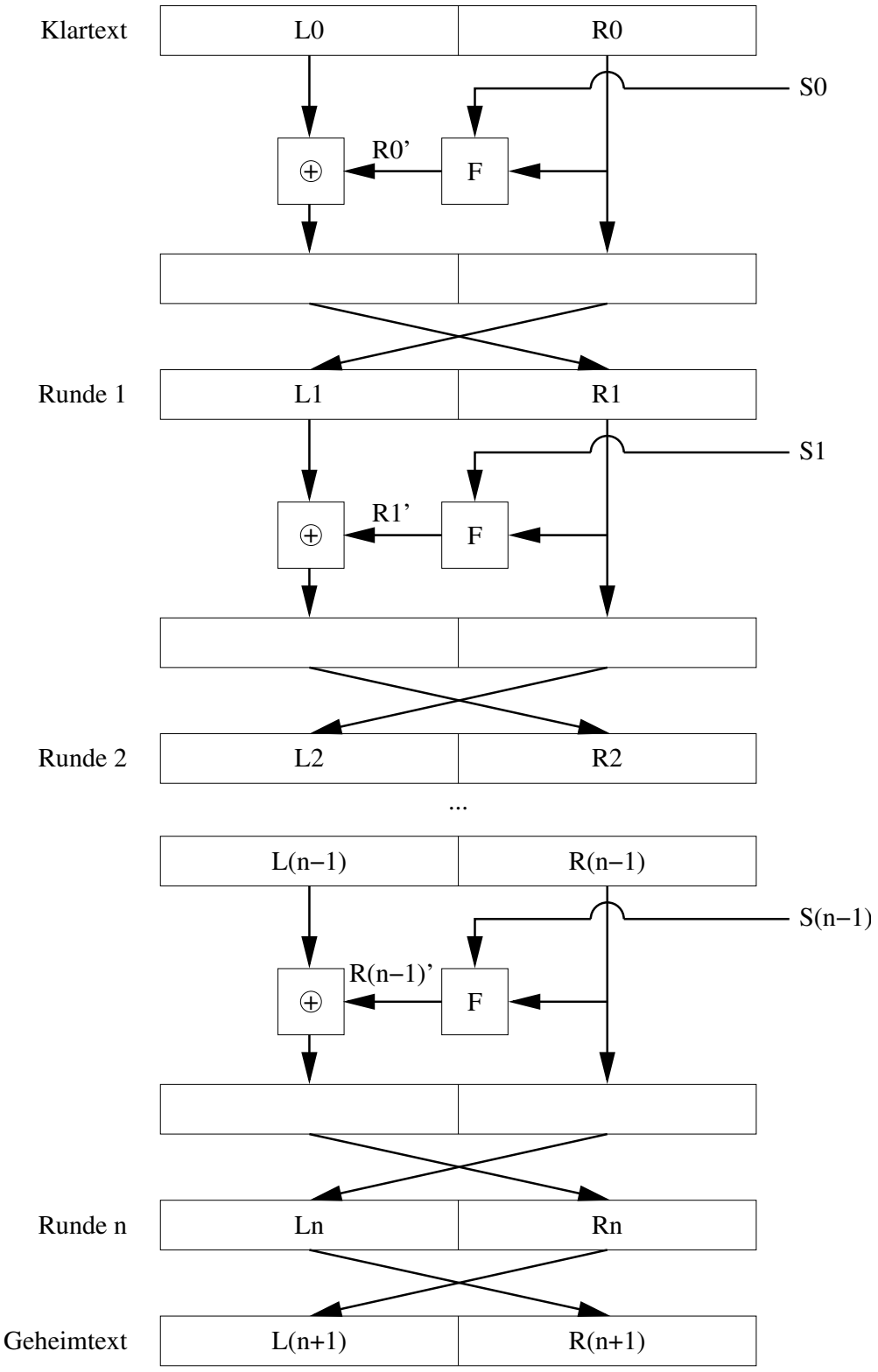

Abbildung 2.4: Verschlüsselungsprinzip von Feistel

steuert. Nach der letzten Runde werden L_n und R_n noch ein letztes Mal vertauscht. Damit ist die Verschlüsselung abgeschlossen.

Die Entschlüsselung erfolgt nach genau demselben Prinzip. Der einzige Unterschied zur Verschlüsselung besteht darin, die Teilschlüssel S_i in umgekehrter Reihenfolge zu benutzen. In der ersten Runde steuert also S_{n-1} die Funktion F, in der zweiten Runde S_{n-2} und in der letzten Runde steuert S_0 die Funktion F.

Man durchläuft die in Abbildung 2.4 gezeigten Runden also von unten nach oben. Warum kommt dabei aber wieder der Klartext heraus? Dies liegt im Wesentlichen daran, dass F eine Funktion ist (jede Eingabe also auf eine eindeutige Ausgabe abgebildet wird) und die Funktion XOR nach zweimaliger Anwendung wieder die ursprüngliche Eingabe liefert.

Bei der Entschlüsselung ist also $R_{n+1} = L_n = R_{n-1}$ die erste Eingabe für F und S_{n-1} steuert F. Da F also dieselben Eingaben wie in der letzten Verschlüsselungsrunde erhält, berechnet es auch wieder denselben Wert R'_{n-1}, der an XOR übergeben wird. L_{n+1} ist bei der Verschlüsselung aus L_{n-1} und R'_{n-1} entstanden. Wendet man die Funktion XOR also wieder auf L_{n+1} und R'_{n-1} an, so erhält man wieder L_{n-1}. Und so kann man Stufe für Stufe überprüfen, dass wieder dieselben Zwischenwerte wie bei der Verschlüsselung entstehen und am Ende wieder der Klartext entstanden ist. Die Parameter des Feistel-Verschlüsselungsverfahrens sind:

Parameter des Feistel-Verfahrens

1. Die Funktion F. Das Verfahren funktioniert mit jedem F, allerdings sollte F nicht zu einfach sein, damit Krypto-Analysen keinen Erfolg haben.
2. Die Art und Weise, wie aus dem Schlüssel S die Teilschlüssel S_i berechnet werden.
3. Die Zahl der Runden. Je mehr Runden, desto besser wird i. d. R. die Verschlüsselung. Allerdings dauert dann auch die Berechnung länger.
4. Die Blockgröße g. Große Blöcke machen statistische Analysen schwieriger.
5. Die Länge des Schlüssels S. Je länger der Schlüssel ist, desto mehr unterschiedliche Schlüssel gibt es und desto länger dauern Angriffe, die einfach alle möglichen Schlüssel nacheinander ausprobieren.

In den folgenden Abschnitten werden einige aktuelle Verschlüsselungsverfahren kurz vorgestellt. Dabei wird auf die Parameter und die generellen Eigenschaften der Verfahren eingegangen. Weitere Details zu den Verfahren finden sich bei Schneier [Sch96] und Stallings [Sta06].

Data Encryption Standard

DES: Der *Data Encryption Standard (DES)* wurde bereits 1977 in den USA vom National Bureau of Standards als Standardverfahren für US-Behörden festgelegt. Dieser Algorithmus wird auch heute noch oft eingesetzt, häufig auch im geschäftlichen Bereich. Es ist im Wesentlichen ein Feistel-Verfahren mit folgenden Parametern:

Funktion F	Kombination aus XOR und festen Ersetzungsboxen, (engl. **Substitution Box (S-BOX)**)
Teilschlüssel	Erzeugung von Teilschlüsseln der Länge 48 Bit durch Shift und Permutation
Runden	16
Blockgröße	64 Bit
Schlüssellänge	56 Bit

Die genaue Arbeitsweise von F und den S-Boxen wurde zunächst nicht offengelegt. Kritiker vermuteten dort Hintertüren, die es erlauben könnten, eine verschlüsselte Nachricht auch ohne Kenntnis des Schlüssels wieder zu entschlüsseln. Trotz vieler Analysen von DES hat bisher niemand eine solche Schwachstelle veröffentlicht.

Durch die gestiegene Rechenleistung ist der Schwachpunkt von DES eher in der relativ kurzen Schlüssellänge von 56 Bit zu sehen. 1997 konnte eine DES-verschlüsselte Nachricht durch Ausprobieren aller möglichen Schlüssel in 97 Tagen entschlüsselt werden. Ein Programm verteilte Intervalle von möglichen Schlüsseln an verschiedene Rechner im Internet (jedermann konnte sich beteiligen). Mitte 1998 wurde mit einer Spezialmaschine (Deep Crack) ein DES-Schlüssel in 56 Stunden gefunden, und im Januar 1999 brauchten parallel suchende Rechner und Deep Crack zusammen nur noch 22,25 Stunden. Hinweise auf den Aufbau von Deep Crack und ein Design für eine verbesserte Entzifferungsmaschine finden Sie bei Brazier [Bra99] oder in [Ele98].

Eine Verbesserung von DES besteht darin, es mit verschiedenen Schlüsseln mehrmals hintereinander auszuführen. Dadurch vergrößert sich der Suchraum für einen Angriff durch Ausprobieren aller Schlüssel erheblich. In der Praxis wird *Triple-DES (3DES)* häufig benutzt. Details zu diesem Verfahren finden Sie bei Stallings [Sta06].

Blowfish: Bruce Schneier hat 1993 einen sehr schnellen und kompakten Verschlüsselungsalgorithmus entworfen. Basierend auf dem Feistel-Verfahren werden bei Blowfish aber in jeder Runde *beide* Hälften des Worts verändert. Vergleichen Sie dazu Abbildung 2.4, in der immer nur das linke Teilwort verändert wird, während das rechte Teilwort einfach weitergeleitet wird. Blowfish

Die Verschlüsselung mit Blowfish findet in zwei Phasen statt. Zuerst werden anhand des Schlüssels die Teilschlüssel und der Inhalt von 4 S-Boxen berechnet. Anschließend findet die eigentliche Verschlüsselung des Klartextwortes statt. Die Parameter von Blowfish sind:

Funktion F	Kombination aus XOR, Ganzzahladdition modulo 2^{32} und S-Boxen
Teilschlüssel	18 Teilschlüssel der Länge 32 Bit werden ausgehend von Initialwerten durch Verknüpfung mit dem Schlüssel und Anwendung des Algorithmus auf sich selbst generiert.
Runden	16
Blockgröße	64 Bit
Schlüssellänge	32 ... 448 Bit (1 ... 14 Wörter der Länge 32 Bit)

Durch die variable Schlüssellänge kann man zwischen den Eigenschaften „Sichere Verschlüsselung“ und „Schnelle Berechnung“ abwägen. Das systematische Ausprobieren aller Schlüssel ist nicht nur wegen der großen Schlüssellänge schwierig, sondern auch wegen der aufwendigen Initialisierungsphase. Solange sich der Schlüssel nicht ändert, kann man die berechneten Teilschlüssel und S-Boxen auch speichern und so die Verarbeitungsgeschwindigkeit erhöhen. Der Speicherbedarf für die Teilschlüssel und die S-Boxen liegt in der Größenordnung von 5 KB.

Bruce Schneier hat einen „Nachfolge“-Algorithmus für Blowfish entwickelt. Er heißt *Twofish* und wird in einem eigenen Buch [Sch+99] vorgestellt und besprochen. *OpenVPN* ist eine freie Software, mit der man virtuelle private Twofish

Netze (VPN) realisieren kann. Dort ist Blowfish als Verschlüsselungsalgorithmus voreingestellt. VPNs werden in Kurs *(01867) Sicherheit im Internet 2* genauer behandelt.

CAST-128

CAST-128: Carlisle Adams und Stafford Tavares haben CAST-128 entwickelt. Es ist ein klassischer Feistel-Algorithmus und wurde von vielen Forschern analysiert. Schwachstellen wurden bisher nicht veröffentlicht. Die Parameter von CAST-128 sind:

Funktion F	Kombination aus einfacher Arithmetik ($+, -, \oplus$), zirkulären Shifts und fest definierten S-Boxen
Teilschlüssel	Aus dem Schlüssel mit Hilfe spezieller S-Boxen generiert
Runden	16
Blockgröße	64 Bit
Schlüssellänge	128 Bit

In *GnuPG* (siehe auch Abschnitt 3.2.1) ist CAST-128 als symmetrischer Verschlüsselungsalgorithmus voreingestellt. Dort wird zwar der Name CAST5 benutzt, es handelt sich aber um denselben Algorithmus.

Advanced Encryption Standard (AES): Aufgrund der inzwischen möglichen Angriffe auf mit DES verschlüsselte Nachrichten begann das National Institute of Standards and Technology (NIST) der USA im Jahre 1997 nach einem neuen Standard-Verschlüsselungsalgorithmus zu suchen. Jedermann konnte Algorithmen vorschlagen. Diese wurden dann vom NIST und von vielen Forschern auf dem Gebiet der Kryptografie analysiert. Alle Algorithmen waren komplett spezifiziert und standen zur Analyse zur Verfügung. 1999 wurden fünf Algorithmen in die engere Auswahl genommen:

Advanced Encryption Standard

Algorithmus	eingereicht von
MARS	IBM
RC6	RSA Laboratories
RIJNDAEL	Joan Daemen, Vincent Rijmen
Serpent	Ross Anderson, Eli Biham, Lars Knudsen
Twofish	Bruce Schneier, John Kelsey, Doug Whiting David Wagner, Chris Hall, Niels Ferguson

2.3.4 Der Advanced Encryption Standard AES

Nach eingehender Untersuchung der Algorithmen wurde eine Variante von Rijndael mit einer Blockgröße von 128 Bit [DR99; DR02] zum neuen Standard-Verschlüsselungsalgorithmus erklärt. Während das Feistel-Verfahren darauf basiert, dass ein Klartextblock in zwei Hälften zerlegt wird, findet bei Rijndael eine komplexere Verarbeitung statt. Zunächst einmal ist die Blockgröße für Klartexte nicht auf einen Wert festgelegt. Sie kann 128 Bit, 192 Bit oder 256 Bit betragen. Die gleichen Blockgrößen gelten für den Schlüssel. Während bei DES die Schlüsselgröße auf 64 Bit beschränkt ist (von denen aber 8 Bit Paritätsbits sind, so dass tatsächlich nur 56 Bit effektive Schlüssellänge übrig blieben), sind AES-Schlüssel mindestens 128 Bit groß.

Runden

Bei Rijndael wird ein Block ebenfalls in Runden verschlüsselt. Das heißt, eine bestimmte Gruppe von Operationen wird wiederholt auf den Block angewendet. Die Anzahl der Runden hängt von der Blockgröße und der Schlüsselgröße ab.

Die Rundenzahl liegt zwischen 10 und 14. Ganz am Anfang wird der Block mit dem ersten Rundenschlüssel XOR-verknüpft. Die letzte Runde weicht außerdem ein wenig von der Struktur der übrigen Runden ab. Auf diese „Spezialitäten“ wird im Folgenden nicht genauer eingegangen, sondern die Operationen einer „normalen“ Runde werden vorgestellt.

Für die Verarbeitung eines Blocks in einer Verschlüsselungsrunde stellt man sich den Block am besten als Matrix von Bytes vor. Auf dieser Matrix werden dann die einzelnen Operationen ausgeführt. In jeder Runde finden mehrere Operationen statt: Operationen

Linear-Mixing-Layer: Diese Schicht sorgt dafür, dass eine große Diffusion der Werte über mehrere Runden stattfindet.

Nonlinear-Layer: In dieser Schicht finden nicht-lineare Ersetzungen statt. Sie sind durch S-Boxen realisiert.

Key-Addition-Layer: In dieser Schicht wird ein Rundenschlüssel zum Block addiert. Mit Addition meint man hier die Operation *Exklusiv-Oder*.

Die Matrix hat unabhängig von der Blockgröße immer vier Zeilen. Die Anzahl der Spalten hängt von der Blockgröße ab. Bei einer Blockgröße von 128 Bit enthält die Matrix auch vier Spalten. Jedes Element der Matrix entspricht einem Byte des Blocks. Man kann nun leicht nachrechnen, dass vier Zeilen mal vier Spalten zu 16 Elementen (Bytes) der Matrix (des Blocks) führen. Und 16 Bytes entsprechen den 128 Bit der Blockgröße.

Wird die Blockgröße auf 192 Bit erhöht, so besteht die Matrix (als die man sich den Block vorstellt) aus sechs Spalten und vier Zeilen. Bei einer Blockgröße von 256 Bit hat die Matrix dann acht Spalten und vier Zeilen. Die folgenden Grafiken aus dem Bericht von Daemen und Rijmen [DR99] basieren auf einer Blockgröße von 192 Bit. Jedes Element der Matrix wird mit $x_{i,j}$ bezeichnet. Dabei ist x ein Name, i bezeichnet die Zeile und j bezeichnet die Spalte.

Als erstes findet in jeder Runde eine Ersetzung der Bytes des Blocks statt. Abbildung 2.5 zeigt das Prinzip. Ein Byte des Ursprungsblocks $a_{i,j}$ wird auf ein ByteSub-Transformation

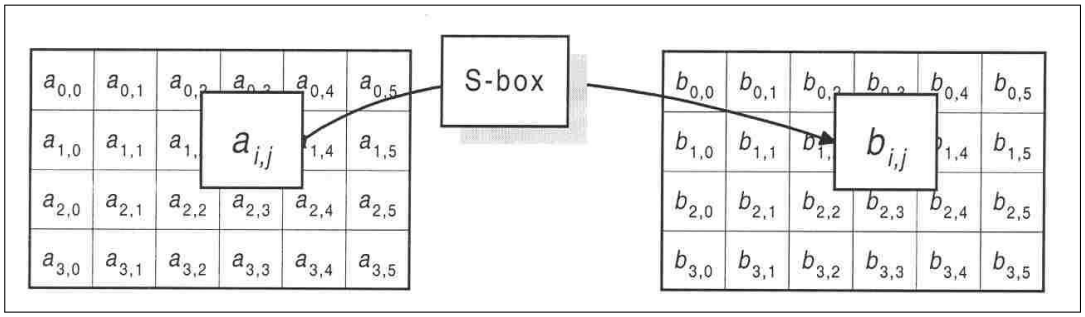

Abbildung 2.5: ByteSub-Transformation beim AES

anderes Byte $b_{i,j}$ abgebildet. Diese Funktion (in Abbildung 2.5 *S-Box* genannt) ist nicht linear, aber umkehrbar und in der Spezifikation von AES genau beschrieben. Letztlich steht dahinter eine Multiplikation eines Vektors mit einer Matrix und einer anschließenden Vektoraddition. Dabei wird das Byte als ein 8-elementiger Vektor mit den Elementwerten Null oder Eins betrachtet.

Im nächsten Schritt wird jede Zeile der Matrix im Kreis verschoben. Die Verschiebedistanzen sind fest definiert und sie hängen nur von der Blockgröße und der Schlüssellänge ab. Abbildung 2.6 zeigt das Prinzip. ShiftRow-Transformation

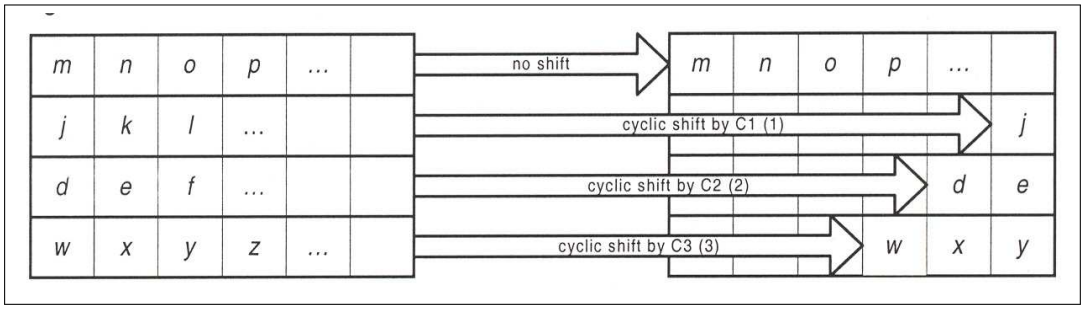

Abbildung 2.6: ShiftRows-Transformation beim AES

MixColumn-Transformation

Im dritten Schritt werden nun die Spalten der Matrix durcheinander gebracht. Dabei wird jede Spalte als vier-elementiger Vektor $a_{\star,j}$ von Bytes betrachtet. Dieser Vektor wird mit einer 4×4 Matrix $c(x)$ multipliziert und ergibt den neuen Vektor $b_{\star,j}$. Abbildung 2.7 zeigt das Prinzip. Die Matrix $c(x)$ ist unabhängig vom Schlüssel und wurde so gewählt, dass die Umkehrung dieser Operation möglich ist.

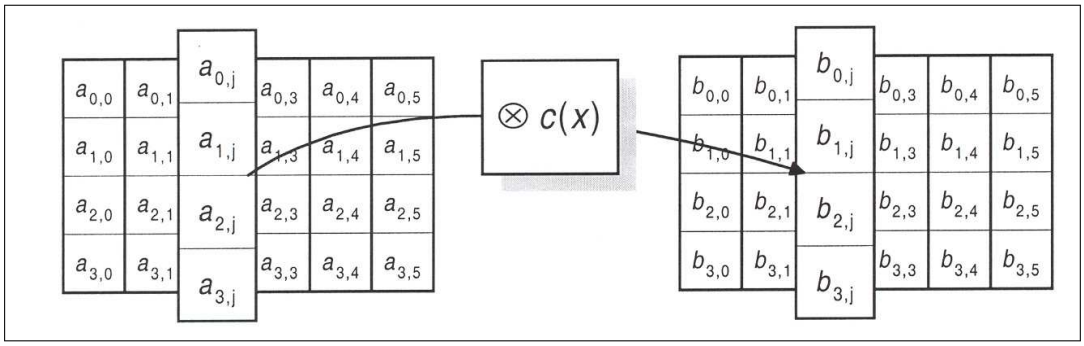

Abbildung 2.7: MixColumn-Transformation beim AES

Am Ende einer Runde wird aus dem Schlüssel ein spezieller Rundenschlüssel berechnet. Der Rundenschlüssel hat dabei immer dieselbe Größe wie ein Block. Er wird auch als Matrix betrachtet und auf den Bytes $a_{i,j}$ des Blocks und den Bytes $k_{i,j}$ des Rundenschlüssels wird die Exklusiv-Oder-Funktion ausgeführt (siehe Abbildung 2.8).

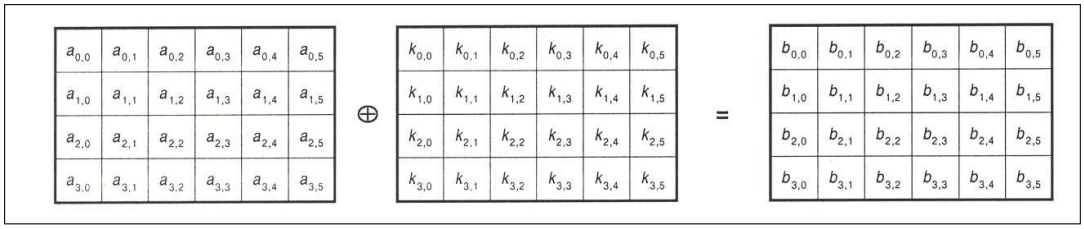

Abbildung 2.8: Add-Round-Key-Transformation beim AES

Schlüssel-Expansion

Die Berechnung der Rundenschlüssel aus dem eigentlichen Verschlüsselungsschlüssel erfolgt in zwei Schritten. Als erstes wird der Verschlüsselungsschlüssel verlängert. Dazu wird er an den Anfang eines langen Arrays geschrieben. Nach einem fest vorgegebenen Algorithmus werden dann die folgenden Felder des Arrays mit Werten gefüllt. Diese Werte ergeben sich aus den Inhalten der vorherigen Array-Elemente. Aus diesem Array werden dann im zweiten Schritt die Rundenschlüssel ausgelesen. Der erste Rundenschlüssel besteht aus den

ersten Elementen des Arrays, der zweite Rundenschlüssel aus den folgenden Elementen usw. Der Anfang des Verschlüsselungsschlüssels und der Anfang des ersten Rundenschlüssels sind also identisch, da der Verschlüsselungsschlüssel an den Anfang des Arrays geschrieben wurde.

Diese Operationen ergeben zusammen eine Verschlüsselungsrunde. Bei der Verschlüsselung eines Klartextblocks werden nun mehrere Runden nacheinander durchgeführt. Der Ausgabeblock der einen Runde ist immer der Eingabeblock der nachfolgenden Runde.

Da alle Operationen in diesem Verfahren umkehrbar sind, kann man einen Geheimtextblock auch wieder entschlüsseln. Dazu werden in jeder Runde immer die Umkehrfunktionen an Stelle der oben dargestellten Funktionen benutzt, sowie die Operationen einer Runde in umgekehrter Reihenfolge ausgeführt.

2.3.5 Das One Time Pad

Es gibt ein symmetrisches Verschlüsselungsverfahren, das sehr einfach berechnet werden kann und beweisbar nicht zu „knacken“ ist. Es heißt **One Time Pad** oder auch *Vernam Chiffre*. Der Klartext wird dabei mit dem Schlüssel XOR-verknüpft.

One Time Pad

$$\text{Geheimtext} = \text{Klartext} \oplus \text{Schlüssel}$$

Die Entschlüsselung erfolgt auch durch XOR. Man verknüpft den Geheimtext und den Schlüssel wieder mit XOR und die zweimalige Anwendung von XOR liefert die ursprüngliche Eingabe. Der Verschlüsselungsalgorithmus ist gleich dem Entschlüsselungsalgorithmus.

Damit dieses Verfahren wirklich sicher ist sind allerdings eine Reihe von Bedingungen zu erfüllen:

- Der Schlüssel muss genauso lang wie der Klartext sein. Das macht dieses Verfahren sehr unpraktisch. Muss man bei AES für die vertrauliche Übertragung von 100 MB vorab nur einen Schlüssel von 16 Bytes sicher übertragen, so hat man beim One Time Pad das Problem *sichere Übertragung von 100 MB Daten* auf das Problem, *sichere Übertragung eines Schlüssels von 100 MB* „reduziert“.
- Der Schlüssel muss absolut zufällig gewählt werden.
- Jeder Schlüssel darf *nur ein einziges Mal* benutzt werden. Auch Teile eines Schlüssels dürfen nicht wiederverwendet werden.
- Der Schlüssel muss natürlich geheim bleiben.

Unter den genannten Bedingungen kann man einen mit einem One-Time-Pad verschlüsselten Geheimtext nicht entschlüsseln. Da der Schlüssel zufällig gewählt wurde ist für einen Angreifer jeder Schlüssel gleich wahrscheinlich der Richtige. Nehmen wir an, der Klartext K wurde mit dem Schlüssel S_K verschlüsselt und ergibt den Geheimtext $G = K \oplus S_K$. Nun kann man zu jedem anderen Klartext K_i (mit derselben Länge wie K) einen Schlüssel S_i finden, so dass gilt: $K_i = G \oplus S_i$.

Da aber alle Schlüssel S_i gleich wahrscheinlich möglich wären, sind auch alle potentiellen Klartexte gleich wahrscheinlich möglich. Auch sind keine statistischen Angriffe mehr möglich, denn jedes Klartext-Zeichen wurde mit

einem anderen Schlüssel-Zeichen verschlüsselt. Im Prinzip ist das One-Time-Pad wie eine Vigenère-Chiffre mit einem sehr langen Schlüssel.

Dieses Verschlüsselungsverfahren wird trotz seines „Problems“ mit den sehr langen Schlüsseln praktisch eingesetzt. Das „rote Telefon“ mit dem die Präsidenten der USA und Rußlands direkt miteinander kommunizieren können ist tatsächlich ein Fernschreiber (Fax) und benutzt ein One-Time-Pad als Verschlüsselungsverfahren.

2.3.6 Verschlüsselungsmodi

Alle bisher besprochenen Verfahren arbeiten auf Blöcken. In der Praxis verschlüsselt man aber nur selten einzelne Blöcke, sondern größere Nachrichten. Diese werden dazu in Blöcke der passenden Größe aufgeteilt. Dabei kann es natürlich vorkommen, dass mehrere identische Klartextblöcke entstehen. Ist die Nachricht beispielsweise ein Java-Quelltext, dann könnte dort öfter der Block ******** vorkommen, weil Zeilen aus Sternen gerne als Kommentar zur optischen Gliederung benutzt werden.

Jedes in diesem Abschnitt bisher beschriebene Verfahren ist auch auf Blockebene deterministisch, d. h. aus einem Klartextblock wird durch Verschlüsselung mit demselben Schlüssel immer der gleiche Geheimtextblock. Man kann also wieder mit Hilfe von statistischen Methoden versuchen, den Geheimtext ohne Kenntnis des Schlüssels zu entschlüsseln. Den Modus, bei dem aus einem Klartextblock immer derselbe Geheimtextblock wird, nennt man auch elektronisches
ECB Codebuch (engl. **Electronic Code Book (ECB)**). Dieser Modus bietet sich nur für kurze Nachrichten an.

In diesem Modus kann ein Angreifer evtl. Nachrichten wiedererkennen und
Replay-Angriff wiederholt einspielen (sog. Replay-Angriff). Dazu bringt man den Absender A dazu, zwei Mal dieselbe Nachricht zu verschlüsseln und an den Empfänger B zu versenden. Der Angreifer protokolliert nun alle Nachrichten von A an B. Diese sind zwar verschlüsselt, aber er kann seine Nachricht daran erkennen, dass sie zwei Mal vorkommt. Später kann der Angreifer die erkannte und aufgezeichnete Nachricht noch einmal an B schicken und B glaubt, dass sie wieder von A stammt.

Übungsaufgabe 2.3 *Warum ist es überhaupt wichtig, einen Replay-Angriff zu verhindern?*

Eine einfache Verbesserung des ECB-Modus führt zum **Cipher-Block-**
CBC **Chaining (CBC)**. Dabei wird jeder Klartextblock vor seiner Verschlüsselung mit dem vorhergehenden Geheimtextblock verknüpft. Dazu benutzt man wieder die Funktion XOR.

Abbildung 2.9 zeigt das CBC Prinzip. Der erste Block wird vor der Verschlüsselung mit einem beliebigen, zufällig gewählten und nicht vorhersagbaren Initialisierungsvektor IV verknüpft. Dabei stellt + die XOR-Operation dar. Dieser Initialisierungsvektor kann beispielsweise im ECB-Modus verschlüsselt vorab an den Empfänger übertragen werden. Das Ergebnis des XOR wird dann verschlüsselt, in Abbildung 2.9 mit E wie *encrypt* bezeichnet. Der Schlüssel K ist der „normale“ geheime Schlüssel. Das Ergebnis ist dann der erste Geheimtextblock G_1, der ausgegeben wird und gleichzeitig mit dem zweiten Klartextblock K_2 verknüpft wird. Dadurch ist sichergestellt, dass zwei identische Klartextblöcke i. d. R. durch zwei unterschiedliche Geheimtextblöcke

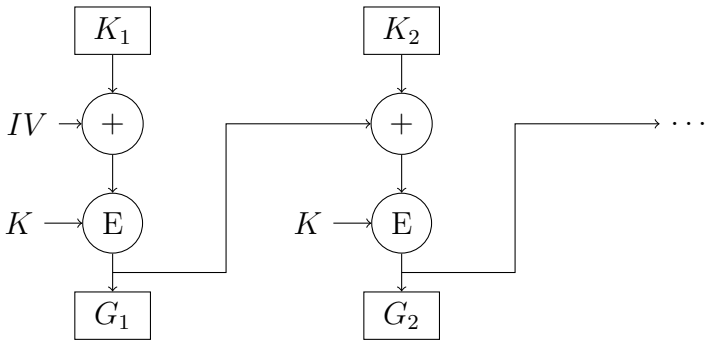

Abbildung 2.9: Prinzip des Cipher-Block-Chaining-Mode

ersetzt werden. Die Entschlüsselung erfolgt im Prinzip genauso. Zunächst wird G_i mit K entschlüsselt und das Ergebnis mit der vorherigen Geheimtextblock G_{i-1} mit XOR verknüpft. Beim ersten Geheimtextblock wird stattdessen der IV für das XOR benutzt. Die zweimalige Anwendung von XOR liefert dann wieder den ursprünglichen Eingabewert.

Insgesamt, also für eine Nachricht als Ganzes, ist die Verschlüsselung natürlich wieder deterministisch. Ein *kompletter* Klartext wird auch im CBC-Modus immer in denselben Geheimtext umgewandelt. Durch den CBC-Modus kann man allerdings keine statistischen Angriffe auf *einzelne Blöcke* mehr durchführen, denn gleiche Klartextblöcke werden nicht mehr zwangsläufig auf gleiche Geheimtextblöcke abgebildet.

Ein offensichtlicher Nachteil des CBC-Modus ist, dass man zwei Klartextblöcke nicht mehr parallel (gleichzeitig) verschlüsseln kann. Bei Mehr-Kern-CPUs könnte der zweite CPU-Kern nicht schon den zweiten Klartextblock verschlüsseln, während der erste CPU-Kern den ersten Klartextblock verschlüsselt. Der zweite CPU-Kern muss warten, bis der erste Kern fertig ist, denn der erste Geheimtextblock geht ja in die Verschlüsselung des zweiten Klartextblocks ein.

Counter Mode

Eine weitere populäre Methode ist der **Counter Mode**. Hierbei können Klartextblöcke parallel verschlüsselt werden und es wird auch erreicht, dass zwei identische Klartextblöcke *nicht* zu zwei identischen Geheimtextblöcken werden. Die Idee dieses Verfahrens basiert auf einem One-Time-Pad (siehe Abschnitt 2.3.5). Der lange, zufällige, niemals wiederverwendete Schlüssel wird erzeugt, indem ein zufällig gewählter Zähler Ct mit dem Schlüssel verschlüsselt wird. Der Klartextblock wird mit diesem Verschlüsselungsergebnis XOR-verknüpft. Für den nächsten Klartextblock wird der Zähler inkrementiert.

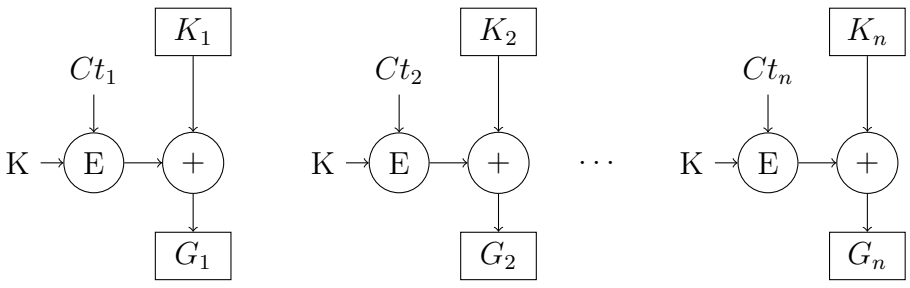

Abbildung 2.10: Prinzip des Counter-Mode

Abbildung 2.10 zeigt den Ablauf einer Verschlüsselung im Counter Mode. Zunächst wird ein Zähler Ct_1 mit einem Zufallswert initialisiert. Bei einer Block-

länge von n Bits ist dieser Zähler eine Zahl aus dem Wertbereich $0, 1, \ldots, 2^n - 1$. Man wählt $n \geq 128$, damit ausreichend viele verschiedene Werte möglich sind. Dieser Zähler Ct_1 wird nun mit dem Schlüssel K verschlüsselt und das Ergebnis wird mit dem ersten Klartextblock K_1 XOR-verknüpft und ergibt den ersten Geheimtextblock G_1. Die folgenden Zähler werden inkrementiert, z. B.: $Ct_i = (Ct_{i-1} + 1) \bmod 2^n$. Der Zähler wird um eins erhöht und falls ein Überlauf bei der Addition auftreten würde, dann geht es bei 0 weiter.

inkrementiert

Wichtig für die Sicherheit des Counter Mode ist, dass immer wieder andere Counter benutzt werden. Auch bei unterschiedlichen Nachrichten sollten nicht dieselben Counter vorkommen. Daher macht man es häufig so, dass die ersten 64 Bit des Counters mit einem sogenannte *message nonce* gefüllt werden. Das ist ein Zufallswert der speziell für diese Nachricht erzeugt wurde also eine Art kurzer Hashwert. Die zweiten 64 Bit sind dann der eigentliche Zähler. Sie werden auch mit einer Zufallszahl initialisiert und nur die hinteren 64 Bit werden beim Inkrementieren hochgezählt.

Wie der IV im Cipher Block Chaining Mode muss der Empfänger den ersten Counter Ct_1 kennen, wenn er die Nachricht entschlüsseln will. Dazu überträgt man den ersten Counter Ct_1 einfach unverschlüsselt vor dem ersten Geheimtextblock.

Übungsaufgabe 2.4 *Warum wird der erste Counter nicht wie der IV im CBC-Modus mit der Schlüssel verschlüsselt vor dem ersten Geheimtextblock an den Empfänger geschickt?*

Neben diesen dreien beschriebenen Modi gibt es auch weitere Modi. Beispielsweise sind im DES auch ein Cipher-Feedback-Modus (CFB) und ein Output-Feedback-Modus (OFB) definiert. Auch diese Modi stellen sicher, dass ein Klartextblock nicht immer durch denselben Geheimtextblock ersetzt wird.

CFB
OFB

2.3.7 Zusammenfassung: Secret-Key-Verschlüsselung

Es gibt heute eine ganze Reihe klassischer Verschlüsselungsverfahren. Sie sind schnell und effizient berechenbar und bei ausreichender Schlüssellänge auch hinreichend sicher. Insbesondere AES mit seinen variablen Schlüssellängen empfiehlt sich hier. Die zugehörigen Programme sind bereits implementiert und getestet. Sie können sie kaufen oder teilweise auch frei aus dem Internet laden.

Dann brauchen Sie sich mit einem potentiellen Kommunikationspartner „nur“ noch auf einen geheimen Schlüssel zu einigen und schon können Sie sicher kommunizieren. Leider ist es nicht immer einfach sich auf einen gemeinsamen Schlüssel zu verständigen, insbesondere weil der Schlüssel wie zufällig gewählt sein sollte und nicht zu erraten sein darf. Denn jeder, der im Besitz des Schlüssels ist, kann alle damit verschlüsselten Nachrichten entschlüsseln.

Durch die Enthüllungen von Edward Snowden im Jahr 2013 wurde häufig berichtet, dass die NSA symmetrische Verschlüsselungsverfahren knacken könnte. Vorstellbar wäre es natürlich, dass in AES solche ausnutzbaren Schwachstellen existieren. Es ist allerdings sehr unwahrscheinlich, dass keiner der nicht bei der NSA beschäftigten Kryptografieexperten bisher auf solche Schwachstellen aufmerksam geworden ist. Statt dessen stellt es sich eher so dar, dass die NSA nicht den Verschlüsselungsalgorithmus selbst angreift, sondern den verwendeten Schlüssel. Schlecht implementierte Verschlüsselungsprogramme bauen vielleicht

einen Hinweis auf den benutzen Schlüssel in den Geheimtext ein. Dieser Hinweis ist dann nur für die NSA verständlich. Oder die NSA greift die Implementierung der Verfahren an, mit denen Programme die symmetrischen Schlüssel erzeugen. Details hierzu sind allerdings noch nicht bekannt (Stand: September 2013).

2.4 Public-Key-Verschlüsselung

2.4.1 Prinzip der Public-Key-Verschlüsselung

asymmetrische Verfahren

Public-Key-Verschlüsselungsverfahren werden auch asymmetrische Verfahren genannt. Jeder Teilnehmer braucht zwei zusammengehörige Schlüssel, einen zum Verschlüsseln und einem zum Entschlüsseln. Der Sender einer Nachricht verschlüsselt die Nachricht mit dem Verschlüsselungsschlüssel des Empfängers und schickt die Nachricht ab. Nur der Empfänger besitzt den zugehörigen Entschlüsselungsschlüssel und kann die Nachricht lesen.

öffentlichen Schlüssel

private Schlüssel

Den Schlüssel zur Verschlüsselung bezeichnet man auch als **öffentlichen Schlüssel** (engl. **public key**). Diesen Schlüssel veröffentlicht jeder Teilnehmer, damit jedermann Nachrichten an ihn verschlüsseln kann. Der Entschlüsselungsschlüssel ist der **private Schlüssel** (engl. **private key**). Diesen hält der Empfänger geheim, denn nur er selbst soll an ihn gerichtete Nachrichten entschlüsseln können. Abbildung 2.11 zeigt das Prinzip.

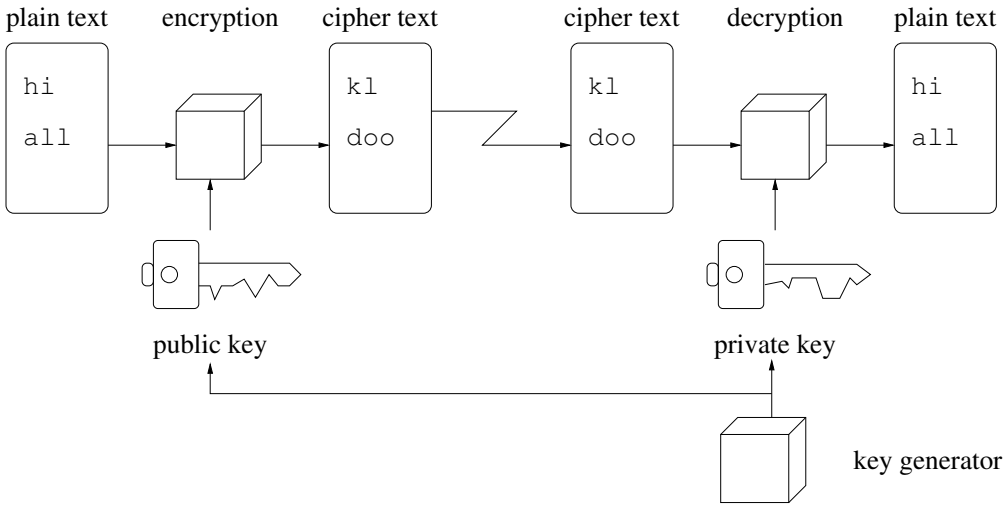

Abbildung 2.11: Ablauf bei Nachrichtenübertragung mit Public-Key-Verschlüsselung

Der Empfänger hat einmal mit einem Schlüsselgenerator sein Schlüsselpaar erzeugt. Der öffentliche Schlüssel wird jedermann zugänglich gemacht, also auch potentiellen Gegnern. Der Sender verschlüsselt eine Nachricht nun mit dem öffentlichen Schlüssel. Der Empfänger kann die Nachricht mit seinem geheimen privaten Schlüssel entschlüsseln.

Damit dieses Verfahren funktioniert, müssen die folgenden Bedingungen erfüllt sein:

- Aus dem öffentlichen Schlüssel kann man den privaten Schlüssel *nicht* ableiten, bzw. diese Berechnung ist so schwierig, dass der zu erwartende Zeitaufwand nicht lohnt oder größer ist als die Lebensdauer der verwen-

deten Schlüssel oder größer als die Zeitspanne, in der die verschlüsselten Daten vertraulich bleiben müssen.

- Alleine mit Hilfe des öffentlichen Schlüssels kann eine verschlüsselte Nachricht nicht entschlüsselt werden, bzw. das ist genauso aufwendig wie das Ausprobieren aller möglichen privaten Schlüssel.

Bei jedem Public-Key-Algorithmus gilt für jede Nachricht m

$$\text{DeCrypt}_d(\text{Crypt}_e(m)) = m$$

wobei e und d die öffentlichen und privaten Schlüssel sind.

Gilt auch $\text{Crypt}_e(\text{DeCrypt}_d(m)) = m$, was stets der Fall ist, wenn die Mengen der möglichen Klartexte und Geheimtexte gleich sind, dann kann man mit dem Algorithmus auch eine digitale Unterschrift leisten. Kann der Empfänger die verschlüsselte Nachricht des Absenders mit dessen öffentlichem Schlüssel entschlüsseln, so muss die Nachricht von diesem Absender stammen. Niemand sonst kennt den privaten Schlüssel. Damit ist die Authentizität der Nachricht sichergestellt. Allerdings ist keine Vertraulichkeit mehr gegeben, da jedermann die Nachricht mit dem öffentlichen Schlüssel entschlüsseln und lesen kann. Es sei denn, man verschlüsselt die Nachricht noch zusätzlich mit dem öffentlichen Schlüssel des Empfängers. Mehr zu digitalen Unterschriften finden Sie in Abschnitt 2.6.

2.4.2 Verschlüsselungsalgorithmen

RSA

RSA: Ron Rivest, Adi Shamir und Leonhard Adleman haben 1978 einen Public-Key-Verschlüsselungsalgorithmus entwickelt. Dieser Algorithmus kann zur Verschlüsselung und zur Erzeugung digitaler Signaturen benutzt werden. Viele Forscher haben inzwischen RSA analysiert und keine grundlegenden Schwachstellen gefunden. Allerdings ist es auch nicht gelungen, die Sicherheit von RSA zu beweisen. Die Sicherheit von RSA basiert darauf, dass es schwer ist eine große Zahl (d. h. mit mehreren hundert Dezimalstellen) zu faktorisieren, also in ihre Primfaktoren zu zerlegen.

Prinzip von RSA

Um ein Schlüsselpaar zu erzeugen, wählt man zunächst zwei große Primzahlen p und q. Dann wird das Produkt $n = p \cdot q$ berechnet. Für jede Zahl $m < n$ gilt nun die folgende Gleichung

$$m^{k(p-1)(q-1)+1} \bmod n = m$$

für ein beliebiges k.

Anschließend wählt man eine natürliche Zahl e, so dass e teilerfremd zu $(p-1)(q-1)$ ist. Der öffentliche Schlüssel besteht aus e und n. Der geheime Schlüssel d wird so berechnet, dass

$$e \cdot d \bmod (p-1)(q-1) = 1 \tag{2.1}$$

gilt. Daraus folgt dann, dass auch

$$e \cdot d = k(p-1)(q-1) + 1$$

für ein k gilt.

Verschlüsselung

Eine Nachricht m wird mit dem öffentlichen Schlüssel wie folgt verschlüsselt:

$$\text{Crypt}_e(m) = m^e \bmod n$$

Zur Entschlüsselung wird $c = \text{Crypt}_e(m)$ in folgende Formel eingesetzt:

$$\text{DeCrypt}_d(c) = c^d \bmod n$$

Insgesamt gilt nun:

$$\begin{aligned} \text{DeCrypt}_d(\text{Crypt}_e(m)) &= \text{Crypt}_e(m)^d \bmod n \\ &= (m^e)^d \bmod n \\ &= m^{e \cdot d} \bmod n \\ &= m^{k(p-1)(q-1)+1} \bmod n \\ &= m \end{aligned}$$

Eigenschaften von RSA

Mit RSA kann man nur Nachrichten verschlüsseln, die kleiner als n sind, d. h. deren Länge begrenzt ist. Größere Nachrichten werden vor der Verschlüsselung in Blöcke passender Länge zerlegt.

Da die Multiplikation kommutativ ist, kann man auch mit dem geheimen Schlüssel d eine Nachricht verschlüsseln. Diese Nachricht kann dann mit dem öffentlichen Schlüssel wieder entschlüsselt werden. Somit kann man mit RSA auch eine digitale Unterschrift leisten.

Probleme

Die Sicherheit von RSA beruht darauf, dass man aus dem öffentlichen Schlüssel (e, n) nicht so ohne weiteres auf den privaten Schlüssel (d, n) schließen kann. Alle heute bekannten Verfahren basieren darauf, zunächst n in seine Faktoren p und q zu zerlegen und dann mittels des erweiterten euklidischen Algorithmus eine Lösung d für die Gleichung 2.1 zu finden. Solche Faktorisierungsalgorithmen werden intensiv erforscht. Der Aufwand der Faktorisierung steigt mit der Größe von n. Sie sollten bei der Erzeugung Ihres Schlüsselpaars auf eine möglichst große Schlüssellänge achten. 1024 Bit ist eine untere Grenze, besser sind 2048 Bit Schlüssellänge. Sollte in Zukunft ein neuer, effizienter Algorithmus zur Faktorisierung großer Zahlen gefunden werden, so ist RSA mit einem Mal hinfällig, da dann alle Nachrichten einfach entschlüsselt werden könnten. Möglicherweise können Quantencomputer (siehe Kurs *(01867) Sicherheit im Internet 2*) demnächst große Zahlen effizient faktorisieren.

Die Multiplikation großer ganzer Zahlen ist eine aufwendige Operation die im RSA-Verfahren als Realisierung der Potenzierung benötigt wird. Die Verschlüsselung mit RSA dauert also spürbar länger als die Verschlüsselung mit einem symmetrischen Verschlüsselungsalgorithmus. Eine Daumenregel besagt, dass RSA-Verschlüsselung ca. 100-mal langsamer ist als DES-Verschlüsselung. Dieser Wert variiert natürlich und hängt von der verwendeten Hard- und Software ab. In der Praxis benutzt man RSA deshalb häufig nur dazu, einen zufällig generierten geheimen Schlüssel zu übertragen. Die eigentliche Kommunikation wird dann symmetrisch mit dem geheimen, sogenannten **Session Key** verschlüsselt. Dabei muss man darauf achten, dass dieser Session Key groß genug ist. Ist der Session Key beispielsweise nur eine Zahl zwischen 0 und 999, so kann ein Angreifer einfach auch alle diese Session Keys mit dem Public Key des Empfängers verschlüsseln und speichern. Fängt der Angreifer dann die erste Nachricht ab, so muss er diese Nachricht *nicht* entschlüsseln, sondern nur mit seinen 1000 gespeicherten Nachrichten vergleichen. Damit wäre der Session Key schnell gefunden.

Session Key

Aber auch wenn der Session Key groß genug ist, sind weitere Vorsichtsmaßnahmen erforderlich. Ein Session Key wird i. d. R. von einem Programm generiert. Einem Angreifer darf es also nicht möglich sein, anhand des ihm evtl. vorliegenden Programmcodes zu erkennen, welchen Session Key das Programm als nächstes erzeugen wird. Gute Zufallszahlengeneratoren und eine echte Zufallsquelle zur Initialisierung des Generators sind daher Grundbestandteile eines guten Verschlüsselungsprogramms.

Der öffentliche Schlüssel eines Teilnehmers ist bei Public-Key-Verfahren jedermann bekannt. Es bleibt aber das Problem, dass man nicht sicher sein kann, tatsächlich den öffentlichen Schlüssel des geplanten Empfängers zu benutzen. Kann sich ein Angreifer zwischen Sender und Empfänger schalten, kann die sogenannte **Man-in-the-Middle-Attack** erfolgen.

Man-in-the-Middle-Attack

Abbildung 2.12: „Man in the middle“-Angriff

Der Ablauf ist dabei wie folgt:

1. Der Sender schickt eine unverschlüsselte Nachricht an den Empfänger und bittet um Übersendung des Public Keys des Empfängers. In dieser Nachricht schickt der Sender seinen öffentlichen Schlüssel mit.

2. Der Dritte fängt die Nachricht ab und erstellt sich zwei Schlüsselpaare P_S, G_S für den Sender und P_E, G_E für den Empfänger. Dann schickt der Dritte eine Nachricht an den Sender, in der er den gerade generierten öffentlichen Schlüssel P_S als öffentlichen Schlüssel des Empfängers einträgt. Weiterhin verändert er die Nachricht des Senders und trägt P_E als öffentlichen Schlüssel ein. Diese Nachricht schickt er an den Empfänger.

3. Der Sender hält nun P_S für den öffentlichen Schlüssel des Empfängers und schickt eine mit P_S verschlüsselte Nachricht. Der Empfänger hält P_E für den öffentlichen Schlüssel des Senders und verschickt seinen öffentlichen Schlüssel damit verschlüsselt ab.

4. Der Dritte kann nun diese Nachrichten entschlüsseln und lesen. Weiterhin kann er sie mit den echten öffentlichen Schlüsseln wieder verschlüsseln und weiterleiten.

5. Sender und Empfänger können mit ihren echten privaten Schlüsseln die Nachrichten korrekt entschlüsseln. Sie merken gar nicht, dass sie immer die falschen öffentlichen Schlüssel zur Verschlüsselung benutzen.

Es ist also wichtig, einen festen und überprüfbaren Zusammenhang zwischen einem öffentlichen Schlüssel und der dazugehörigen Person herzustellen. Diese Aufgabe sollen sogenannte Trustcenter (siehe auch Abschnitt 2.7.1) übernehmen.

Trustcenter

El-Gamal: Während die Sicherheit des RSA-Verfahrens darauf basiert, dass keine effizienten Algorithmen zur Faktorisierung großer Zahlen bekannt sind, basiert das Verfahren von El-Gamal [El 85] auf einem anderen Problem. Die Berechnung von *diskreten Logarithmen* über endlichen Körpern ist ein ähnlich schwieriges Problem.

El-Gamal

endliche Körper

Ein Beispiel für einen endlichen Körper ist Z_p, die Menge aller Zahlen, die als Rest bei der Division durch die Primzahl p entstehen. Es gilt $Z_p = \{0, 1, 2, \ldots, p-1\}$; für die Primzahl 7 besteht Z_7 aus den Ziffern 0 bis 6. Auf diesen Elementen sind die Addition und die Multiplikation wie üblich definiert. Entsteht bei einer Operation allerdings eine größere Zahl, so wird wieder der Rest bei der Division durch p ermittelt.

$$
\begin{aligned}
&(1+3) \bmod 7 = 4 \\
&(4+5) \bmod 7 = 9 \bmod 7 = 2 \quad \text{da } 9/7 = 1 \text{ Rest } 2 \text{ ist.} \\
&(3 \cdot 5) \bmod 7 = 15 \bmod 7 = 1 \quad \text{da } 15/7 = 2 \text{ Rest } 1 \text{ ist.}
\end{aligned}
$$

Weiterhin ist auch die Exponentiation definiert.

$$
\begin{aligned}
&3^2 \bmod 7 = 9 \bmod 7 = 2 \\
&3^3 \bmod 7 = 27 \bmod 7 = 6
\end{aligned}
$$

diskreter Logarithmus

Der diskrete Logarithmus von y zur Basis g ist nun die kleinste Zahl x, für die gilt: $y = g^x \bmod p$. Bei gegebenem g und x ist die Berechnung von $g^x \bmod p$ recht einfach. Nicht für alle g kann man zu jedem y ein x finden, so dass $y = g^x$ gilt. Es gibt aber stets solche Zahlen g und man nennt sie *primitiv modulo p*. Das heißt, dass zu jedem Element y aus Z_p außer der 0 ein x existiert mit $g^x \bmod p = y$. Man nennt g in diesem Fall *Generator*. 3 ist beispielsweise primitiv modulo 7, denn es gilt:

$$
\begin{aligned}
3^0 &= 1 \quad &&\bmod 7 = 1 \\
3^1 &= 3 \quad &&\bmod 7 = 3 \\
3^2 &= 9 \quad &&\bmod 7 = 2 \\
3^3 &= 27 \quad &&\bmod 7 = 6 \\
3^4 &= 81 \quad &&\bmod 7 = 4 \\
3^5 &= 243 \quad &&\bmod 7 = 5
\end{aligned}
$$

Hat man nun nur g und y gegeben und sucht die Zahl x für die $g^x = y \bmod p$ gilt, so ist diese Berechnung sehr schwierig.

Die Verschlüsselung einer Nachricht nach El-Gamal erfolgt nun wie folgt: Der Sender wählt eine zufällige Zahl k zwischen 0 und $p-1$ und berechnet aus dem „öffentlichen Schlüssel“ $y = g^x \bmod p$ des Empfängers und der Nachricht M die Werte

$$
\begin{aligned}
a &= g^k \bmod p \\
b &= y^k \cdot M \bmod p
\end{aligned}
$$

und schickt sie an den Empfänger. Der berechnet nun (wieder $\bmod p$)

$$
\frac{b}{a^x} = \frac{y^k \cdot M}{a^x} = \frac{g^{(x \cdot k)} \cdot M}{g^{(x \cdot k)}} = M
$$

An Stelle der Division durch a^x multipliziert man mit a^{-x} und nutzt zusätzlich aus, dass $a^{-x} = a^{p-1-x}$ gilt. Das liegt daran, dass $a^{p-1} = 1 \bmod p$ ist. Und da $p - 1 - x > 0$ ist, werden vom Prozessor nur Multiplikationen durchgeführt.

Diffie-Hellman

Diffie-Hellman: Bereits 1976 haben Diffie und Hellman [DH76] ein ähnliches Verfahren vorgestellt, mit dem sich zwei Parteien auf einen geheimen Schlüssel verständigen können. Das Verfahren ist *kein* Public-Key-Verfahren wie RSA oder El-Gamal, sondern ein Schlüsselaustauschverfahren. Den sicher ausgetauschten Schlüssel kann man dann als Schlüssel für einen symmetrischen Verschlüsselungsalgorithmus benutzen.

Ablauf

Die Parteien A und B einigen sich zuerst auf eine große Primzahl p und eine Zahl g, die primitiv modulo p ist. Diese Zahlen dürfen bekannt sein. A wählt nun zufällig eine Zahl a_x und schickt $a_y = g^{a_x} \bmod p$ an B. B wählt eine Zahl b_x und schickt $b_y = g^{b_x} \bmod p$ an A. Jetzt berechnet A aus dem empfangenen b_y und seinem geheimen zufällig gewählten Wert a_x den folgenden Wert: ${b_y}^{a_x} \bmod p$. Genauso berechnet B den Wert ${a_y}^{b_x} \bmod p$. A und B haben nun beide den gleichen Wert berechnet, da die folgende Umformung (immer $\bmod p$) gilt:

$$ {b_y}^{a_x} = (g^{b_x})^{a_x} = g^{(b_x \cdot a_x)} = g^{(a_x \cdot b_x)} = (g^{a_x})^{b_x} = {a_y}^{b_x} $$

Jemand, der den Kommunikationskanal abhört, kennt aber nur g, p, a_y und b_y und kann den berechneten Wert nicht rekonstruieren.

MITM

Dieses Schlüsselaustauschverfahren ist allerdings anfällig gegen Man-In-The-Middle-Angriffe wie in Abbildung 2.12 dargestellt. Ein Angreifer H, der den Nachrichtenaustausch zwischen A und B kontrolliert kann A gegenüber so tun als sei er B und B gegenüber so tun als sei er A. Der Angreifer handelt also mit A einen Schlüssel aus und auch mit B. Die Nachrichten von A an B muss H dann zunächst entschlüsseln, mitlesen und anschliessend für B erneut verschlüsseln. Die Antworten von B werden analog behandelt. Das Problem beim Diffie-Hellman-Schlüsselaustausch ist, dass die Kommunikationspartner *nicht* authentisiert werden.

authentisiertes Diffie-Hellman

Sollten A und B bereits ein gemeinsames Geheimnis gg kennen, dann könnten sie die Nachrichten beim Schlüsselaustausch damit verschlüsseln. Allerdings könnten Sie auch direkt gg als symmetrischen Schlüssel benutzen. Welchen Vorteil hätte es, wenn auch in diesem Fall ein Diffie-Hellman-Schlüsselaustausch gemacht wird? Man reduziert den entstehenden Schaden falls das gemeinsame Geheimnis gg doch bekannt würde. Gelangt ein Angreifer irgendwann in den Besitz von gg, dann kann er alle aufgezeichneten Nachrichten, die mit gg symmetrisch verschlüsselt wurden, entschlüsseln. Hätte man also gg direkt als Schlüssel benutzt, dann wäre der Inhalt aller vertraulichen Nachrichten doch bekannt. Hat man gg aber nur benutzt, um den Diffie-Hellmann-Schlüsselautausch zu verschlüsseln, dann kann der Angreifer nur die Austausch-Nachrichten entschlüsseln. Er kann daraus aber *nicht* die ausgehandelten Schlüssel rekonstruieren. Nachdem das Geheimnis gg kompromittiert wurde, darf damit natürlich kein Schlüsselaustausch mehr gemacht werden. Ein Angreifer kann den oben beschriebenen MITM-Angriff erfolgreich durchführen wenn er gg kennt.

Sollten A und B beide ein asymmetrisches Schlüsselpaar besitzen, dann könnte man das Diffie-Hellman-Protokoll so erweitern, dass A und B ihre Nachrichten während des Schlüsselaustauschs digital signieren (siehe Abschnitt 2.6.2). Dann können A und B sicher sein, dass sie den Schlüssel mit dem richtigen Partner ausgehandelt haben. Alternativ kann A seine Nachrichten an B auch mit dem öffentlichen Schlüssel von B verschlüsseln und B verschlüsselt die Nachrichten an A mit dem öffentlichen Schlüssel von A.

2.4.3 Zusammenfassung: Public-Key-Verschlüsselung

Jeder Anwender in einem Public-Key-Verschlüsselungverfahren benötigt zwei Schlüssel. Ein öffentlicher Schlüssel ist zum Verschlüsseln von Nachrichten an den Teilnehmer nötig. Die Entschlüsselung der Nachricht ist dann nur mit Hilfe des geheimen Schlüssels möglich. Daher stehen der öffentliche und der geheime Schlüssel in einem mathematischen Zusammenhang. Im Prinzip kann aus dem öffentlichen Schlüssel auch der geheime Schlüssel berechnet werden. Allerdings sind bisher keine effizienten Algorithmen für diese Berechnung bekannt.

Die vorgestellten Verfahren von Rivest, Shamir und Adleman sowie von El-Gamal stützen sich auf unterschiedliche mathematische Probleme. RSA nutzt aus, dass große Zahlen nur schwer in ihre Primfaktoren zerlegt werden können. El-Gamal und der Diffie-Hellman-Schlüsselaustausch basieren auf dem Problem der Berechnung von diskreten Logarithmen in endlichen Körpern. Es gibt weitere mathematische Probleme, die in Public-Key-Verschlüsselungsverfahren zum Einsatz kommen können. An dieser Stelle soll nur erwähnt werden, dass ein Zusammenhang zwischen Public-Key-Verschlüsselung und der Theorie der *elliptischen Kurven* besteht.

elliptische Kurven

Aufgrund der komplexen Berechnungsvorschriften sind die Public-Key-Verfahren deutlich langsamer als Private-Key-Verfahren. Trotzdem sollte man als Anwender möglichst große Schlüssel wählen. Die Verschlüsselung dauert dann zwar etwas länger, allerdings ist es für einen Gegner auch deutlich schwieriger, den Schlüssel zu „knacken“.

2.5 Hashfunktionen

Ursprünglich waren Hashfunktionen als Hilfsmittel zum Speichern und Wiederfinden von Datensätzen (Elementen) konzipiert. Datenstrukturen wie Listen oder Bäume enthalten i. d. R. explizit die Adressen der Elemente in Form von Zeigern (engl. **pointer**). Die Idee des Hashing ist es nun, aus den Datensätzen selbst die Adressen zu berechnen. Diese Adressberechnung soll eine sogenannte Hashfunktion ausführen.

2.5.1 Prinzip von Hashfunktionen

Eine Hashfunktion erhält das zu speichernde Element als Eingabe und berechnet daraus eine Adresse, also eine Zahl aus einem gegebenen Intervall. Die Zielmenge der Hashfunktion ist i. d. R. deutlich kleiner als die Ursprungsmenge. Ein Beispiel:

Ursprungsmenge:	Zeichenketten
Zielmenge:	Zahlen zwischen 0 und 25
Hashfunktion:	Zahlenwert des ersten Zeichens der Zeichenkette dabei sei ‚a‘ = 0, ‚b‘ = 1 usw.

Die Hashfunktion H betrachtet also nur das erste Zeichen der Zeichenkette und berechnet daraus eine Adresse, beispielsweise den Index in einer Tabelle. Einige konkrete Hashwerte sind:

$$\begin{aligned} &\text{H(‚andreas‘)} = 0 \\ &\text{H(‚caesar‘)} = 2 \\ &\text{H(‚amadeus‘)} = 0 \end{aligned}$$

Da die Zielmenge (hier $\{0, \ldots, 25\}$) kleiner ist als die Ursprungsmenge, müssen einige Elemente der Ursprungsmenge auf denselben Hashwert abgebildet werden.

Kollision

Dies nennt man *Kollision*. In Kurs *(01661) Datenstrukturen I* und Kurs *(01662) Datenstrukturen II* wird das Thema Hashing zum Speichern und Wiederfinden von Daten und die Behandlung von Kollisionen genauer behandelt.

Zusammenfassend kann man sagen, dass eine Hashfunktion eine Abbildung ist, die Elemente einer großen Ursprungsmenge auf Elemente einer kleineren Zielmenge abbildet. Eine gute Hashfunktion besitzt die folgenden Eigenschaften:

- Sie ist schnell und einfach zu berechnen.
- Sie „streut“ möglichst gut auf die Elemente der Zielmenge.

In der Kryptografie benutzt man Hashfunktionen, um die Authentizität einer Nachricht zu prüfen. Dazu wird von einer Nachricht M beliebiger Länge ein „Fingerabdruck“ $H(M)$ fester Länge berechnet. Man bezeichnet den Hashwert auch als *kryptografische Prüfsumme* (oder engl. **message digest**). Dieser Wert wird zusammen mit der Nachricht verschickt. Der Empfänger kann nun selbst die Hashfunktion auf die empfangene Nachricht M' anwenden und dann $H(M')$ mit dem mitgeschickten $H(M)$ vergleichen. Stimmen diese Werte nicht überein, so wurde die Nachricht unterwegs verändert. Es darf hierbei einem Angreifer aber nicht möglich sein, während der Übertragung M *und* $H(M)$ gleichzeitig zu ändern.

kryptografische Prüfsumme
message digest

Damit eine Hashfunktion in diesem Sinne zuverlässig (und sicher) funktioniert, sind allerdings weitere Eigenschaften erforderlich:

Eigenschaften

Einwegfunktion (engl. preimage resistance): Zu einem vorgegebenen Hashwert h ist es praktisch unmöglich eine Nachricht M zu finden, für die $H(M) = h$ gilt.

Schwache Kollisionsresistenz (engl. second preimage resistance): Zu einer vorgegebenen Nachricht M_1 ist es praktisch unmöglich, eine Nachricht $M_2 \neq M_1$ zu finden, für die $H(M_1) = H(M_2)$ gilt.

Starke Kollisionsresistenz (engl. collision resistance): Es ist praktisch unmöglich, zwei verschiedene Nachrichten M_1 und M_2 zu finden, für die $H(M_1) = H(M_2)$ gilt.

Hierbei bedeutet praktisch unmöglich, dass man auch mit aller zur Verfügung stehenden Rechenzeit das Problem nicht in realistischer Zeit lösen kann.

Aus diesen allgemeinen Eigenschaften lassen sich konkrete Anforderungen an Hashfunktionen ableiten. Zunächst muss die Zielmenge ausreichend groß sein. Bildet H wie im obigen Beispiel jede Zeichenkette nur auf eine Zahl aus dem Intervall 0 bis 25 ab, so ist es trivial, zwei Nachrichten mit dem gleichen Hashwert zu finden. Dazu braucht man bloß 27 Nachrichten zu generieren. Dann muss mindestens ein Hashwert doppelt vorkommen.

Heute werden Hashfunktionen mit einem 128 Bit breiten Wertebereich bereits als zu unsicher betrachtet. Gute Hashfunktionen haben einen Wertebereich, der mindestens 200 Bit breit ist und damit Hashwerte aus dem Intervall 0 bis $2^{200} - 1$ zulässt.

Weiterhin muss jedes Bit aus der Nachricht M in die Berechnung von $H(M)$ eingehen. Andernfalls kann man wieder sehr einfach eine Nachricht $M' \neq M$

mit $H(M') = H(M)$ generieren. Dazu ändert man einfach die Bits, die nicht in die Berechnung eingehen. In obigem Beispiel kann man alle Zeichen außer dem ersten verändern und erhält wieder denselben Hashwert.

Die starke Kollisionsresistenz scheint sich nur durch die Wortwahl von der schwachen Kollisionsresistenz zu unterscheiden. Wir verdeutlichen den Unterschied am sogenannten Geburtstagsparadoxon. Schätzen Sie einmal, wie viele Gäste auf Ihre Geburtstagsfeier kommen müssen, damit mit einer Wahrscheinlichkeit größer als 0.5 jemand am selben Tag wie Sie Geburtstag hat[4]. Die Antwort berechnet man mit Hilfe der Wahrscheinlichkeit, dass niemand am selben Tag wie Sie Geburtstag hat. Bei nur einem Gast liegt diese Wahrscheinlichkeit bei $364/365 = 0.99726$, bei zwei Gästen entspricht sie $(364/365)^2 = 0.99542$ etc. (wenn mehrere unabhängige Ereignisse eintreten, dann multiplizieren sich die Wahrscheinlichkeiten). Gesucht ist nun das n mit $(364/365)^n < 0.5$. Dann ist die Wahrscheinlichkeit, dass niemand am selben Tag Geburtstag hat wie Sie, kleiner als 0.5, also hat mit mehr als 50% Wahrscheinlichkeit mindestens einer am selben Tag Geburtstag. Für $n = 253$ gilt nun $(364/365)^n = 0.4995$. Es müssen also 253 Gäste kommen, damit die Wahrscheinlichkeit, jemanden mit demselben Geburtstag wie Sie dabei zu haben, größer als 0.5 wird.

Geburtstagsparadoxon

Fragt man stattdessen danach, wieviele Personen anwesend sein müssen, damit die Wahrscheinlichkeit, dass mindestens zwei *beliebige* Teilnehmer am selben Tag Geburtstag haben, größer als 0.5 ist, so ist die Antwort überraschend niedrig. Bei n Gästen entspricht die (umgekehrte) Wahrscheinlichkeit, dass nämlich keine zwei Gäste am gleichen Tag Geburtstag haben, der Wahrscheinlichkeit, aus 365 Zahlen n verschiedene auszuwählen, also $365 \cdot 364 \cdot \ldots \cdot (365-n+1)/365^n$. Schon für $n = 23$ sinkt diese Wahrscheinlichkeit unter 0.5, d.h. bereits bei 23 Teilnehmern ist die Wahrscheinlichkeit, dass zwei von ihnen am selben Tag Geburtstag haben, größer als 50%.

Man kann sich das auch anschaulich so erklären: Bei 23 Besuchern kann man $23 * 22/2 = 253$ mögliche Paare bilden. Allgemein kann man bei n Personen $n * (n-1)/2$ Paare bilden. Hat nun eine Hashfunktion einen Wertebereich der Größe $0 \ldots 2^n - 1$, so reicht es für den Angreifer aus, $O(2^{n/2})$ viele Nachrichten zu erzeugen. Damit lassen sich dann $O(2^{n/2}) * O(2^{n/2}) = O(2^n)$ viele Paare bilden. Die Kollisionswahrscheinlichkeit ist dann statistisch bereits größer als 50%. Salopp gesagt, „bietet eine Hashfunktion mit n Bit breitem Wertebereich nur eine Sicherheit von $n/2$ Bits“.

Die starke Kollisionsresistenz fordert also, dass eine Hashfunktion auch dann keine Kollisionen erzeugt, wenn ein Angreifer eine Menge von Nachrichten erzeugt.

Man fordert starke Kollisionsresistenz, um der folgenden Art Angriff zu begegnen: Nehmen wir an, dass ein Angreifer beispielsweise seinem Chef ein Dokument D, z. B. sein Arbeitszeugnis, mit dem Hashwert $H(D)$ zur digitalen Unterschrift vorlegen darf. Der Angreifer schreibt nun ein realistisches Zeugnis D und erzeugt davon sehr viele unterschiedliche Varianten $D_1, D_2, \ldots, D_n$, alle mit derselben Bedeutung. Das kann er beispielsweise durch Einfügung von zusätzlichen Leerzeichen in den Text erreichen oder indem er einzelne Worte durch Synonyme ersetzt. Hat der Angreifer i Stellen im Dokument, wo er jeweils zwei Möglichkeiten hat (Leerzeichen oder keine Leerzeichen, bzw. Wort oder

Angriffsbeispiel

[4] Wir gehen hier davon aus, dass die Geburtstage gleichverteilt sind, d. h. dass jeder Tag als Geburtstag gleich wahrscheinlich ist. In der Praxis ist das nicht der Fall, denn in einigen Monaten kommen mehr Kinder zur Welt als in anderen.

Synonym), dann kann er 2^i viele Dokumente mit „demselben“ Inhalt erzeugen.

Nun erzeugt er nach demselben Prinzip eine zweite Menge von Dokumenten $K_1, \ldots, K_n$, in denen er sich als Supermann darstellt. Dann werden alle Hashwerte berechnet und Dokumente D_i und K_j gesucht, die denselben Hashwert haben. Wenn n groß genug ist, gibt es genug Dokumentpaare (nämlich $O(n^2)$), so dass eine Kollision sehr wahrscheinlich wird.

Nun kann der Angreifer seinem Chef das Dokument D_i zur Unterschrift vorlegen. Wenn der Chef es unterschreibt, dann sieht es genauso aus, als hätte der Chef K_j unterschrieben. Warum das so ist, wird später in Abschnitt 2.6.2 erklärt.

Übungsaufgabe 2.5 *Gegeben ist die folgende Hashfunktion:*

```
hashwert := 0;
for i := 1 to laenge do
   hashwert := (nachricht[i] + hashwert) mod (2^32);
return hashwert;
```

Dabei ist ***hashwert*** *ein 32 Bit unsigned integer,* ***laenge*** *die Anzahl der Bytes, aus denen die Nachricht besteht, und* ***nachricht[i]*** *das i-te Byte der Nachricht als 8 Bit integer. Ist diese Hashfunktion sicher? Begründen Sie Ihre Antwort.*

Übungsaufgabe 2.6 *Gegeben sei eine Hashfunktion H, die* 1 000 000 *unterschiedliche Hashwerte erzeugen kann, z. B. Zahlen aus dem Intervall von 0 bis 999 999.*

a) Weiterhin sei eine Nachricht N mit Hashwert $H(N)$ gegeben. Wieviele Nachrichten müssen Sie erzeugen, um mit einer Wahrscheinlichkeit größer als $1/2$ *eine Nachricht mit demselben Hashwert $H(N)$ zu erhalten?*

b) Wie viele zufällige verschiedene Nachrichten müssen Sie erzeugen, damit mit einer Wahrscheinlichkeit größer als $1/2$ *mindestens zwei (beliebige) dieser Nachrichten denselben Hashwert haben?*

2.5.2 Hash-Algorithmen

MD5: Dieser Algorithmus wurde von Ron Rivest 1991 entwickelt und stellt eine Verbesserung der vorher entwickelten Hashfunktionen MD2 und MD4 dar. Der Algorithmus verarbeitet Nachrichten bis zu einer maximalen Länge von knapp unter 2^{64} Bits und erzeugt einen Hashwert mit 128 Bit Länge. Die Nachricht wird dazu in Blöcke der Länge 512 Bit zerlegt. Damit dies immer möglich ist, wird die Nachricht am Ende immer nach folgenden Schema aufgefüllt (engl. **padding**).

MD5

1. Es wird ein 1-Bit und dann so viele 0-Bits angehängt, dass die Nachrichtenlänge in Bits bei der Division durch 512 den Rest 448 ergibt.

2. Die Länge der ursprünglichen Nachricht wird als Binärzahl der Länge 64 Bit an die aufgefüllte Nachricht gehängt. Damit ist die Nachricht jetzt ein Vielfaches von 512 Bit lang.

Als nächstes wird ein 128 Bit breiter Puffer mit fest vorgegebenen Werten initialisiert. Dieser Puffer ist in Abbildung 2.13 mit IV bezeichnet. Auf den

Puffer wird nun eine Komprimierungsfunktion F (engl. **compression function**) angewendet. Diese Funktion wird durch den 512 Bit breiten Block der Nachricht gesteuert. Das Resultat der Funktion F ist wieder ein 128 Bit breiter Wert, der weitergereicht wird. Am Ende dieser Kette steht der Hashwert der Nachricht. Das in Abbildung 2.13 gezeigt Prinzip nennt man auch *Merkle-Damgård-Konstruktion.*

Merkle-Damgård-Konstruktion

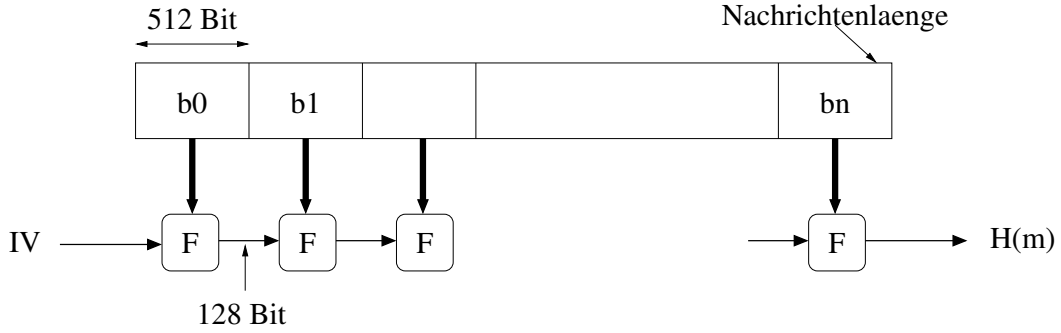

Abbildung 2.13: Prinzip des MD5-Algorithmus

Die Funktion F besteht wiederum aus 4 Runden, die hintereinander ausgeführt werden. Jede der 4 Runden besteht aus 16 Schritten. In den Schritten werden die Boole'schen Operationen UND, ODER, NICHT und XOR benutzt. Weiterhin wird an bestimmten Stellen ein Wert aus einer vorher festgelegten Tabelle zu einzelnen Zwischenwerten addiert. Die Tabelle enthält dabei die Ergebnisse der Sinusfunktion und ist dadurch mit zufällig aussehenden Bitmustern gefüllt. Details zu den Runden und Schritten findet man bei Schneier [Sch96] und Stallings [Sta06].

Im Jahr 2004 haben Xiaoyun Wang und Hongbo Yu [WY05] gezeigt, wie man eine Kollision in MD5 erzeugen kann. Ende 2008 haben dann Sotirov et al. [Sot+08] gezeigt, wie man ein X.509-Zertifikat, das MD5 benutzt, fälschen kann. Damit ist MD5 nicht mehr sicher und sollte durch andere Hash-Algorithmen abgelöst werden.

SHA-1: Dieser Algorithmus wurde vom National Institute of Standards and Technology (NIST) in den USA 1993 als sicherer Hash-Algorithmus entwickelt und standardisiert. Sein Design ist dem von MD5 sehr ähnlich. Der Algorithmus verarbeitet Nachrichten bis zu einer Länge von knapp unter 2^{64} Bit und erzeugt einen Hashwert mit 160 Bit Länge. Wie MD5, so füllt auch SHA-1 zunächst die Nachricht auf und hängt die Länge der ursprünglichen Nachricht an. Die so erzeugte Nachricht ist wieder ein Vielfaches von 512 Bit lang.

SHA-1

Nun wird ein 160 Bit breiter Puffer initialisiert. Die Berechnung des Hashwertes erfolgt nach dem in Abbildung 2.13 gezeigten Verfahren. Die Funktion F besteht bei SHA-1 allerdings aus 4 Runden mit jeweils 20 Schritten. Auch die in den Runden und Schritten ausgeführten Operationen sind bei SHA-1 andere als bei MD5. Die Details findet man bei Schneier [Sch96] und Stallings [Sta06]. Neben SHA-1 gibt es auch noch die als SHA-2-Familie bekannten Varianten SHA-224, SHA-256, SHA-384 und SHA-512. Sie sind Varianten des SHA-1-Prinzips, die sich im Wesentlichen durch den größeren Wertebereich unterscheiden. Die Zahl am Ende des Namens sagt, wie viel Bit breit ein Hashwert der jeweiligen Funktion ist. Vom NIST wurden diese Funktionen zuerst

2001 veröffentlicht und im Jahr 2004 zuletzt aktualisiert.

Auch für die Hashfunktion SHA-1 existieren einige Angriffsmöglichkeiten, die derzeit erforscht werden. Allerdings wurden noch keine Kollisionen gezielt produziert. Bis zur Fertigstellung von SHA-3 sollte man also besser die Varianten mit den Wertebereichen größer 200 Bit verwenden.

SHA-3: Das NIST in den USA hat, vergleichbar mit der Suche nach dem Advanced Encryption Standard (AES), im Jahr 2007 einen Prozess angestoßen, bei dem eine neue Hashstandardfunktion gefunden werden soll. Jedermann durfte bis 31.10.2008 seinen Vorschlag einreichen, 64 Vorschläge trafen ein. Im Februar 2009 wurden die Vorschläge auf einer Konferenz vorgestellt, 14 Kandidaten wurden im Sommer 2009 in die zweite Runde übernommen, und ein Jahr lang untersucht. Die Ergebnisse wurden auf einer Konferenz im August 2010 vorgestellt, und Ende 2010 wurden fünf Finalisten benannt, also Hashfunktionen, die weiter analysiert werden sollen. Im Frühjahr 2012 fand die *Final SHA-3 Candidate Conference* statt. Dort wurden die vertieften Analyseergebnisse der Finalisten vorgestellt und besprochen. Danach hat das NIST den Gewinner zum neuen Hashstandardalgorithmus erklärt.

Dieser Algorithmus soll Hashwerte der Breite 224 Bit, 256 Bit, 384 Bit oder 512 Bit berechnen. Sie sollen stark kollisionsresistent, effizient zu berechnen, sicher gegen die bisher bekannten Angriffe und zufällig verteilt sein.

2.5.3 Der neue Hashstandard SHA-3

Keccak

Der neue Standardalgorithmus heißt **Keccak**. Die vollständige Spezifikation findet sich bei Bertoni et. al. [Ber+11a]. Der Algorithmus ist an mehreren Stellen parametrisierbar. Eine konkrete Parametrisierung ist als SHA-3 Standard [Ber+11b] definiert.

sponge construction

Sponge Konstruktion: Das Keccak zugrunde liegende Verfahren nennt sich **sponge construction** [Ber+07]. Aus einer Permutationsfunktion f, die auf einem Bitvektor mit fester Länge $b = c + r$ arbeitet, wird durch die sponge construction ein Verfahren, das beliebig lange Eingaben M in beliebig lange Ergebnisse *result* umrechnen kann. Diese Konstruktion eignet sich nicht nur für Hash-Funktionen, sondern auch zur Erstellung von Stromverschlüsselungsverfahren oder Zufallszahlengeneratoren.

Abbildung 2.14 zeigt wie die Konstruktion funktioniert. Ein Bitvektor der Länge $c+r$ wird mit Nullen initialisiert. Diese $c+r$ Bits sind der Zustand, der sich im Laufe der Berechnung immer wieder ändert. Die Zahl r heißt **Bitrate** (engl. **bitrate**). Sie gibt an, in welcher Blockgröße die Eingabe M verarbeitet wird. Je größer r ist, desto schneller kann eine Eingabenachricht M verarbeitet werden.

Bitrate

Die Zahl c steht für die Länge des „inneren Zustands“ und wird **Kapazität** (engl. **capacity**) genannt. Je größer c ist, desto „sicherer“ ist Keccak. Für ein gegebenes c gilt: „Es gibt keinen allgemeinen Angriff (engl. **generic attack**) mit einer erwarteten Komplexität kleiner als $2^{c/2}$“ (vgl. [Ber+11b]). Soll ein Ergebnis-Hashwert also n Bit lang sein und Angriffe eine erwartete Komplexität von mindestens 2^n haben, dann wählt man $c = 2n$.

Kapazität

Falls M kein Vielfaches von r Bit lang ist, wird es aufgefüllt, bzw. auf die passende Länge gebracht (engl. **to pad**). Die Verarbeitung besteht nun aus zwei Phasen.

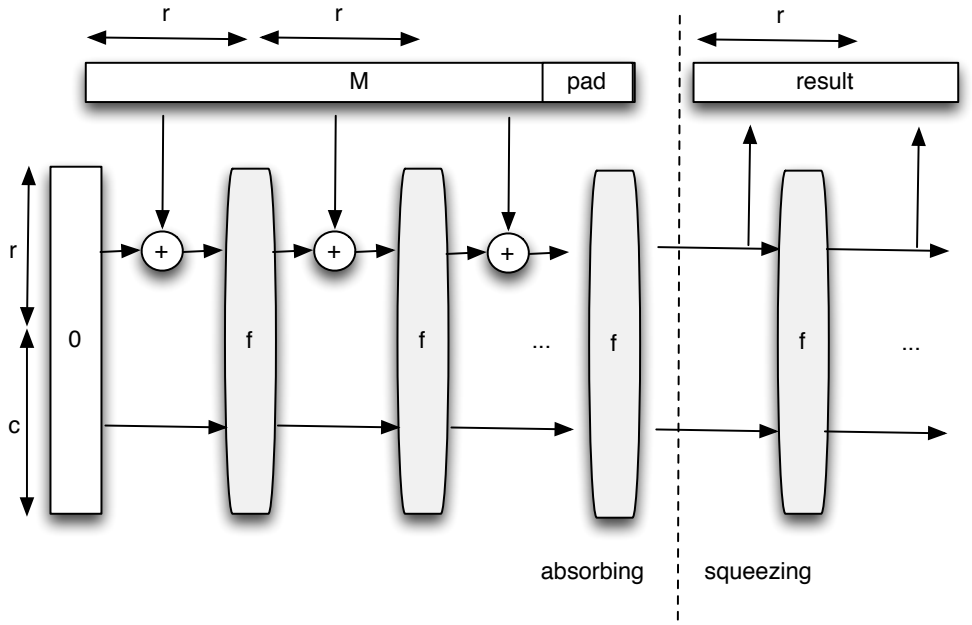

Abbildung 2.14: Übersicht über eine *sponge construction* nach [Ber+07].

1. **Absorbing:** Die ersten r Bits der Eingabe M werden mit den ersten r Bits des Zustands XOR-verknüpft. Zusammen mit den hinteren c Bits aus dem Zustand gehen diese $c + r$ Bits als Eingabe in die Funktion f. Die ersten r Bits der Ausgabe der Funktion f werden dann mit dem nächsten Eingabeblock XOR-verknüpft. Das Ergebnis des XOR und die hinteren c Bits des Funktionsergebnisses gehen wieder als Eingabe in die nächste Berechnung von f ein. Dieses Verfahren wiederholt sich nun für jeden Block aus der Eingabenachricht M.

2. **Squeezing:** Um ein Ergebnis (engl. **result**) der Gesamtlänge s Bits zu berechnen, werden die ersten r Bits des letzten Zustands in das Ergebnis kopiert. Ist $s > r$ so wird der Zustand wieder als Eingabe in die Funktion f benutzt. Die ersten r Bits des Funktionsergebnisses werden in das Ergebnis kopiert. Das wird so oft wiederholt, bis mindestens s Ergebnisbits berechnet wurden. Ist s kein Vielfaches von r, so wird das Resultat nach s Bits einfach abgeschnitten, d. h. die „überzähligen“ Bits werden einfach ignoriert.

Um eine Hashfunktion basierend auf der sponge construction zu definieren, muss man folgende Parameter festlegen:

1. Die Zustandlänge $b = c + r$. Sie ist auch die Eingabegröße für die Funktion f.

2. Eine Funktion $f : \{0,1\}^b \to \{0,1\}^b$.

3. Die Bitrate r und ein padding-Verfahren, wie Eingaben M auf eine Länge gebracht werden, die ein Vielfaches von r ist. Alternativ kann man natürlich auch die Kapazität c festlegen.

4. Die Ergebnislänge s.

Keccak Parameter: In Keccak sind sieben Zustandslängen definiert, 25×2^l, wobei l die Werte 0 bis 6 annimmt. Die kleinste Zustandslänge beträgt also 25 Bit, die größte 1600 Bit (also 200 Byte). Dementsprechend gibt es auch sieben Funktionen, die in den Spezifikationsdokumenten mit KECCAK-[b] bezeichnet werden. Jede der Funktionen KECCAK-[b] arbeitet auf einem Bitstring der Länge b. Dieser Bitstring ist der Zustand (engl. **state**) der Funktion und auf diesem Zustand werden nun bestimmte Runden-Operationen wiederholt angewendet. In jeder Runde werden fünf Operationen ausgeführt. In [Ber+11a] ist eine Runde wie folgt definiert.

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

In dieser Definition stellt man sich den Bitvektor als ein dreidimensionales Array $a[5][5][2^l]$ von Bits vor. Abbildung 2.15 zeigt das Prinzip. Dabei bezeichnet $a[x][y][z]$ mit $0 \leq x < 5$, $0 \leq y < 5$ und $0 \leq z < 2^l$ das Bit an der Koordinate (x, y, z) im Würfel. Die genaue Definition der fünf Operationen finden Sie in der Keccak Referenz [Ber+11a]. Die Zahl der Runden ist $n_r = 12 + 2l$.

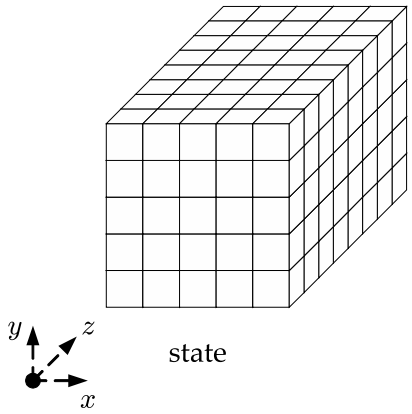

Abbildung 2.15: Übersicht über den Zustand der Funktion KECCAK-[200] nach [Ber+11a]. Der Zustand besteht aus 5×5 „Reihen“ von 8 Bit, die in der Grafik in die z-Dimension weisen.

Als default Wert für die Zustandslänge wird $b = 1600$ genommen und für die Bitrate $r = 1024$. Somit ergibt sich die Kapazität $c = 1600 - 1024 = 576$. Angriffe haben also eine erwartete Komplexität von $2^{576/2} = 2^{288}$. Wenn die Eingabelänge (in Bits) kein Vielfaches der Bitrate ist wird ein sogenanntes **Multi-rate** padding durchgeführt. Dabei wird zunächst ein 1-Bit angehängt, dann die kleinste Zahl an 0-Bits die man braucht um mit einem abschließenden 1-Bit auf die Bitrate zu kommen. Man nennt dieses padding auch *pad10*1*, abgeleitet vom regulären Ausdruck *10*1* der das Muster für die aufgefüllten Bits beschreibt. Bei diesem Verfahren werden mindestens 2 Bits und höchstens $r + 1$ Bits angehängt.

Multi-rate

Eine default Ausgabelänge s ist in der Definition nicht angegeben. Da für den SHA-3 Standard bestimmte Ausgabelängen gefordert wurden, hat man für diese geforderten Längen die entsprechenden Keccak-Parameter in [Ber+11b] definiert. Sie werden im folgenden Abschnitt vorgestellt.

SHA-3: Die als SHA-3 ausgewählte Keccak-Konfiguration benutzt 1600 Bit als Zustandslänge und somit die Funktion KECCAK-[1600] in der sponge con-

struction. Für die vom NIST geforderten Ergebnislängen wurden folgende Bitraten und Kapazitäten festgelegt:

Ausgabelänge	Bitrate	Kapazität
224 Bit	1152 Bit	448 Bit
256 Bit	1088 Bit	512 Bit
384 Bit	832 Bit	768 Bit
512 Bit	576 Bit	1024 Bit

Die Zahl der Runden in der Funktion Keccak-[1600] beträgt 24. Man hat bei dieser Festlegung die doppelte Ausgabelänge n als Kapazität genommen ($c = 2n$) und die Bitrate daraus dann als $r = 1600 - c$ berechnet. Somit geht man davon aus, dass es keine allgemeinen Angriffe (engl. **generic attacks**) mit einer Komplexität kleiner als 2^n gibt. Außerdem sind die 25 Reihen in der Würfel-Interpretation des Zustands (siehe Abbildung 2.15) dann alle 64 Bit lang und passen in ein Register einer 64 Bit CPU. Eine effiziente Berechnung von Keccak-[1600] ist dann auf diesen CPUs besonders einfach programmierbar.

Die Sponge-Konstruktion von Keccak kann auch für einen weiteren interessanten Anwendungsfall benutzt werden. War bei der Berechnung eines Hashwerts die Absorbing-Phase der umfangreichere Teil verglichen mit der Squeezing-Phase, so kann man das auch umdrehen. Benutzt man ein „kleines Geheimnis“ als Eingabe in Keccak, so kann man damit ein beliebig langes Ergebnis produzieren, indem die Squeezing-Kette entsprechend verlängert wird. Dieses lange Ergebnis kann dann als Schlüssel für das One-Time-Pad-Verfahren dienen.

2.5.4 Zusammenfassung: Hashfunktionen

Eine Hashfunktion H ist eine Abbildung, die eine beliebig lange Eingabenachricht m in einen eindeutigen Hashwert $H(m)$ mit einer festen Länge umwandelt. Eine gute Hashfunktion ist einfach zu berechnen, ist nicht umkehrbar und erfüllt die Bedingungen der schwachen und starken Kollisionsresistenz.

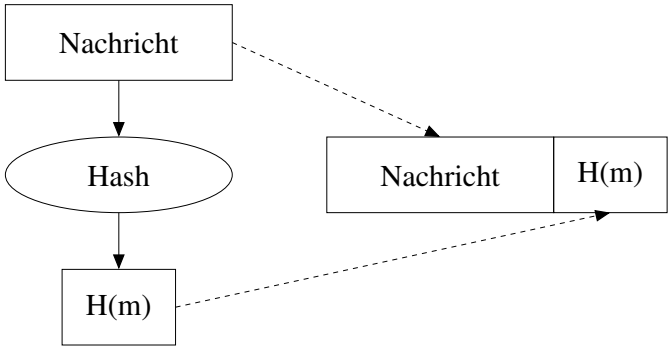

Abbildung 2.16: Anhängen eines Hashwertes an eine Nachricht

Die älteren Hashfunktionen wie MD5 oder SHA-1 werden heute (Stand: Juni 2013) nicht mehr als sicher genug angesehen. Die steigende Rechenleistung und das Geburtstagsparadoxon machen es möglich, Kollisionen zu provozieren.

Man sollte also entweder die SHA-2 Algorithmen mit mehr als 200 Bit langen Hashwerten benutzen oder besser gleich den neuen Standard SHA-3 benutzen.

Hashwerte mit einer Länge größer als 200 Bit machen Angriffe durch Ausnutzung des Geburtstagsparadoxons hinreichend schwierig. Das Geburtstagsparadoxon ist auch der Grund, warum die Länge von sicheren Hashwerten doppelt so groß ist wie die Länge von sicheren symmetrischen Schlüsseln.

Mit Hilfe von guten Hashfunktionen ist es möglich, eine Art „Fingerabdruck“ einer Nachricht zu erstellen. Da die Algorithmen öffentlich bekannt und verfügbar sind, kann man damit allerdings keine echte Verbindlichkeit erwarten. Schließlich kann jemand, der eine Nachricht verändert, auch einfach den neuen Hashwert berechnen und anhängen. Die nächste Erweiterung/Verbesserung kombiniert daher Verschlüsselung mit den Hashfunktionen und liefert die sogenannten „Message Authentication Codes“ oder „digitale Signaturen“.

2.6 Message Authentication Codes und digitale Signaturen

Eine Hashfunktion alleine kann einige Anforderungen an digitale Dokumente noch nicht erfüllen. Wie kann man zum Beispiel sicher sein, dass die Nachricht tatsächlich vom vorgegebenen Absender kommt und beim Transport nicht verändert wurde?

2.6.1 Message Authentication Code

Message Authentication Codes (MAC)

Die Idee eines **Message Authentication Codes (MAC)** ist, dass nicht nur die Nachricht selbst, sondern auch eine geheime Zusatzinformation in die Berechnung des Message Authentication Codes eingeht. Haben sich beispielsweise Sender und Empfänger auf einen geheimen Schlüssel geeinigt, so kann der Hashwert damit vor der Übertragung verschlüsselt werden.

Abbildung 2.17 zeigt das Prinzip des Message Authentication Codes. Der Sender einer Nachricht m, in der Abbildung in der linken Hälfte des Bildes dargestellt, berechnet den Hashwert $H(m)$ und verschlüsselt ihn mit einem geheimen Schlüssel. Dann werden die Nachricht und der MAC (also der verschlüsselte Hashwert) zusammen an den Empfänger geschickt. Der Empfänger der Nachricht m' berechnet wiederum den Hashwert $H(m')$. Außerdem entschlüsselt er den empfangenen MAC und vergleicht die beiden Werte. Sind sie identisch, so ist die Nachricht unverändert angekommen.

Ein Angreifer kann nun die Nachricht abfangen und verändern, er kann aber *nicht* den passenden MAC erzeugen. Dazu müsste auch der geheime Schlüssel bekannt sein.

alternatives Verfahren

Ein alternatives Verfahren zur Berechnung eines Message Authentication Codes wäre es, die geheime Zusatzinformation mit der Nachricht zu konkatenieren und dann davon den Hashwert zu berechnen:

$$MAC(M) = H(\text{Geheimnis}|\text{Nachricht})$$

Der Empfänger muss dann zur Überprüfung des Message Authentication Codes auch das Geheimnis mit der empfangenen Nachricht konkatenieren und davon den Hashwert berechnen. Stimmt diese Berechnung mit dem empfangenen MAC überein, dann wurde die Nachricht bei der Übertragung nicht verändert.

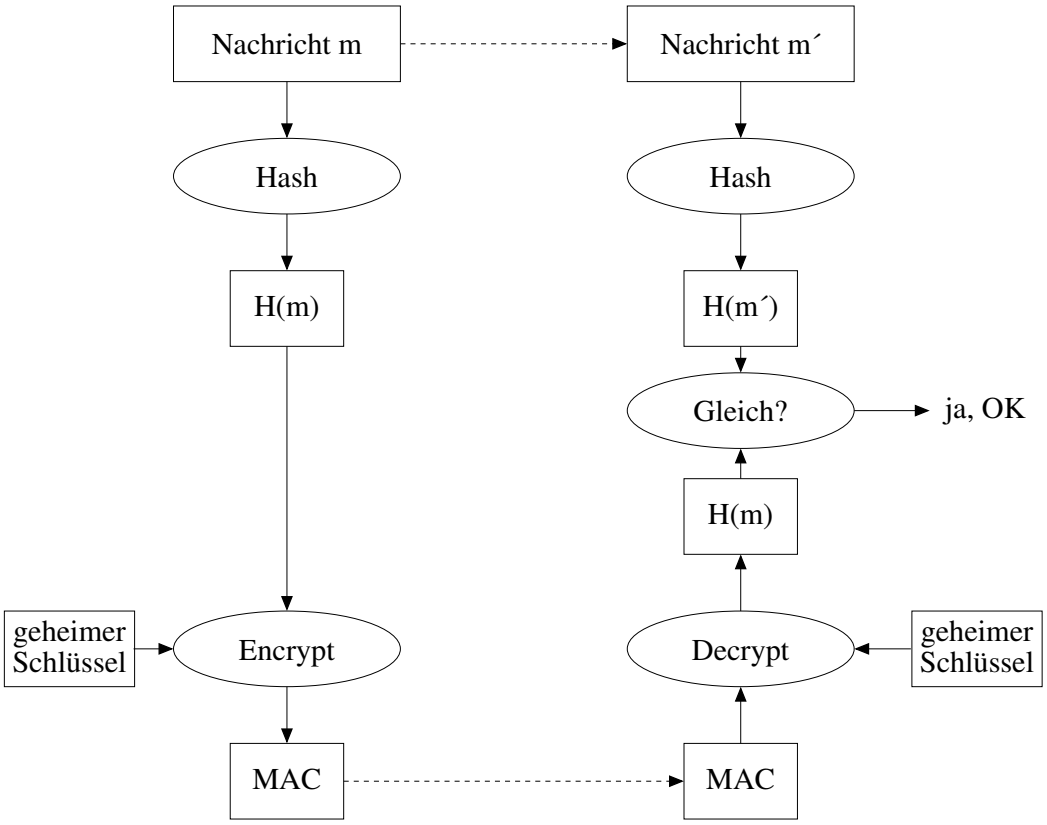

Abbildung 2.17: Prinzip der Message-Authentication

Dieses Verfahren ist allerdings anfällig gegen sogenannte Erweiterungsangriffe (engl. **length extension attacks**). Falls die Hashfunktion nach dem Merkle-Damgård-Prinzip arbeitet (Abbildung 2.13 zeigt dieses Prinzip), dann ist ja der Hashwert das Ergebnis des letzten Funktionsaufrufes. Ein Angreifer kann nun versuchen, die ursprüngliche Nachricht zu verlängern und den angehängten letzten Block mit dem abgefangenen MAC in die Funktion F einzugeben und einen neuen MAC für die verlängerte Nachricht zu berechnen. Im HMAC Standard (siehe RFC 2104) wurde daher dieses etwas erweiterte Berechnungsverfahren definiert:

HMAC Standard

$$HMAC(M) = H(\text{Geheimnis}|H(\text{Geheimnis}|\text{Nachricht}))$$

Der Hashwert der Konkatenation des Geheimnis mit der Nachricht wird noch einmal mit dem Geheimnis konkateniert und erneut ein Hashwert berechnet.

Beim neuen Hashstandard SHA-3 ist diese Erweiterung nicht mehr erforderlich. Da Keccak neben den r Bits der Bitrate auch die c Bits der Kapazität (auch „innerer Zustand“ genannt) besitzt kann ein Angreifer nicht mehr versuchen, die Berechnungskette mit den Funktionsaufrufen zu verlängern. Dazu müsste der Angreifer auch die inneren Zustandsbits rekonstruieren. Hier kann man also einfach den Schlüssel vor die Nachricht konkatenieren und den SHA-3 Hashwert als MAC benutzen.

Trotz eines korrekten Message Authentication Code kann ein Sender allerdings bestreiten, dass er die Nachricht tatsächlich geschickt hat. Da auch der Empfänger den geheimen Schlüssel kennt, könnte er die Nachricht und den passenden MAC auch selbst erzeugt haben.

2.6.2 Digitale Signatur

digitalen Signatur

Tauscht man den symmetrischen Verschlüsselungsalgorithmus bei der ersten Berechnungsvariante des MAC gegen einen asymmetrischen Algorithmus aus, so erhält man das Prinzip der **digitalen Signatur** (siehe Abbildung 2.18). Der Unterschied zu den Message Authentication Codes ist fett gedruckt.

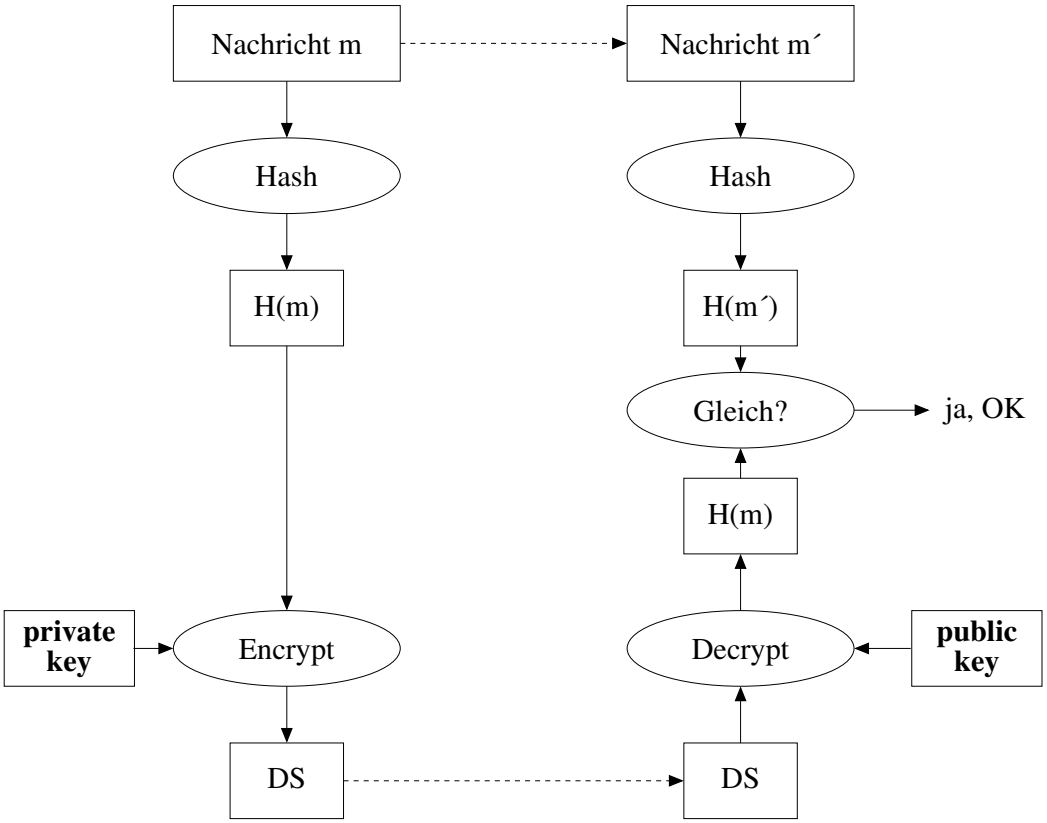

Abbildung 2.18: Prinzip der digitalen Signatur

Ein Angreifer kann wiederum zwar die übertragene Nachricht verändern, aber die passende digitale Signatur (DS) nicht selbst erzeugen. Aber auch der Empfänger kann jetzt keine passende DS zu einer Nachricht erstellen. Der Empfänger kann nur prüfen, ob die digitale Signatur zur empfangenen Nachricht passt. Nur der Absender, der den geheimen Schlüssel (Private Key) kennt, kann Nachrichten mit passender digitaler Signatur erstellen.

An dieser Stelle kann man sich fragen, warum man überhaupt den Hashwert der Nachricht bildet? Schließlich kann man auch direkt die ganze Nachricht erst mit dem eigenen privaten Schlüssel und dann mit dem öffentlichen Schlüssel des Empfängers verschlüsseln.

Man macht dies i. d. R. nicht, da es zu viel Rechenleistung erfordert. Es ist letztlich schneller, den Hashwert zu berechnen und dann nur diesen kurzen Wert mit den asymmetrischen Verfahren gesichert zu übertragen. Die Kollisionsresistenz der Hashfunktion stellt dabei sicher, dass kein Angreifer eine zweite Nachricht mit demselben Hashwert erzeugen kann. Würde das gelingen dann hätte der Angreifer eine digitale Signatur unter einer gefälschten Nachricht.

2.6.3 Algorithmen für digitale Signaturen

Zur Berechnung digitaler Signaturen kann man den bereits in Abschnitt 2.4.2 beschriebenen RSA-Algorithmus einsetzen. Daneben wurde in den USA auch ein eigener Standard, der *Digital Signature Standard (DSS)* , definiert. Dieses Verfahren beruht auf dem Public-Key-Verschlüsselungsverfahren von El-Gamal. Der zugehörige Algorithmus wird *Digital Signature Algorithm (DSA)* genannt und beispielsweise in PGP und GPG (siehe Abschnitt 3.2.1) eingesetzt.

DSS

DSA

2.7 Zertifikate und Schlüsselmanagement

2.7.1 Zertifikate

Die asymmetrischen Verschlüsselungsverfahren garantieren, dass eine mit einem öffentlichen Schlüssel p_i entschlüsselbare Nachricht nur vom Besitzer des zugehörigen privaten Schlüssels s_i stammen kann. Woher kann man aber wissen, welcher öffentliche Schlüssel zu einer bestimmten Person gehört?

Authentizität d. Schlüssels?

Ein äquivalentes Problem aus dem normalen Leben ist die Frage, ob eine Person tatsächlich diejenige ist, die sie vorgibt zu sein. Zur Lösung dieses Problems gibt es Personalausweise. Sie stellen einen Zusammenhang zwischen der Person (durch das Passbild und den Besitz des Dokumentes an sich) und dem Namen her. Wichtige Eigenschaften eines Personalausweises sind die Fälschungssicherheit und die Tatsache, dass sie von einer vertrauenswürdigen Instanz (dem Einwohnermeldeamt) ausgegeben werden.

Das elektronische Äquivalent zum Personalausweis nennt man **Zertifikat**. Ein Zertifikat erlaubt die eindeutige Zuordnung eines öffentlichen Schlüssels zu einer Person. Die Zuordnung wird von einer vertrauenswürdigen Instanz, der sogenannten **Zertifizierungsstelle** (engl. **Certification Authority (CA)**) oder (engl. **Trust Center**) hergestellt. Das Zertifikat ist ein Datensatz, der bestimmte Felder enthält und der mit dem privaten Schlüssel der Zertifizierungsstelle digital signiert ist. Der Datensatz enthält u. a. auch die folgenden Daten:

Zertifikat

Zertifizierungsstelle

- Name des Inhabers (evtl. auch Adresse, Telefonnr. usw.),
- öffentlicher Schlüssel des Inhabers,
- Seriennummer,
- Gültigkeitsdauer,
- Name der Zertifizierungsstelle

Dazu kommt natürlich die digitale Signatur der Zertifizierungsstelle unter dem Zertifikat. Letztlich ist ein Zertifikat also auch „nur“ eine digital signierte Nachricht. Abbildung 2.19 zeigt woraus ein Zertifikat im Prinzip besteht.
Der Sender einer Nachricht schickt nun

- die Nachricht,
- die digitale Signatur und
- sein Zertifikat

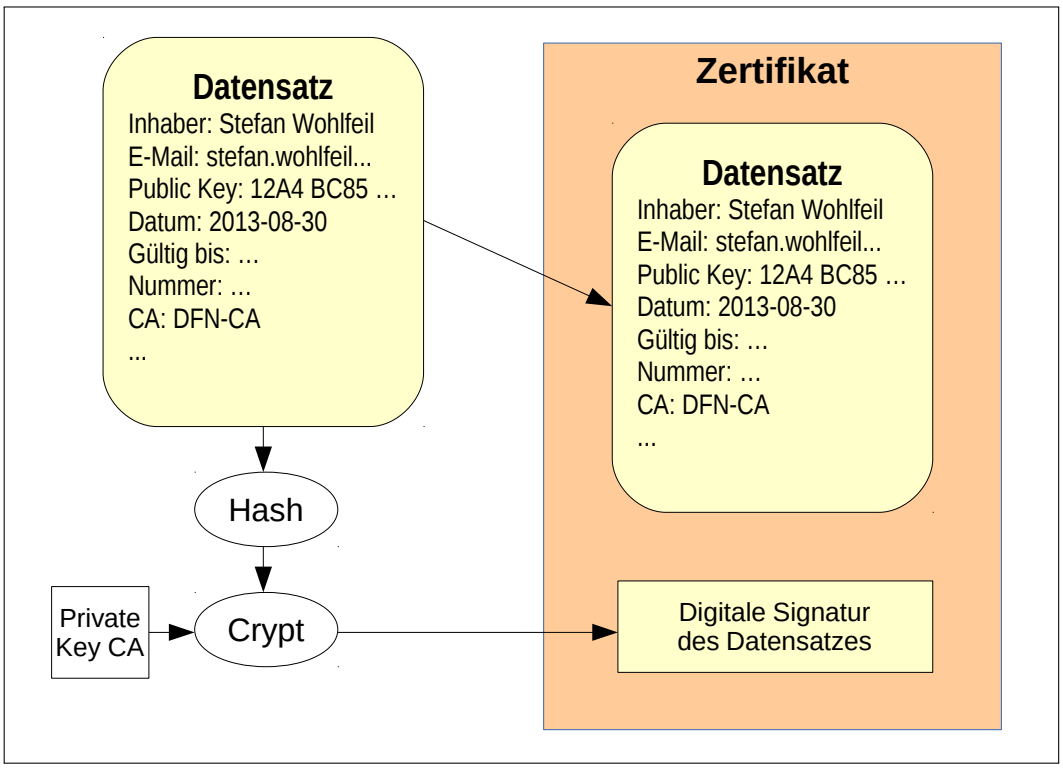

Abbildung 2.19: Bestandteile eines Zertifikats

an den Empfänger. Der Empfänger überprüft das Zertifikat mit Hilfe des öffentlichen Schlüssels der Zertifizierungsstelle. Somit erhält er den Namen und den öffentlichen Schlüssel aus dem Zertifikat. Mit diesem öffentlichen Schlüssel entschlüsselt der Empfänger die digitale Signatur der Nachricht. Nun kann er prüfen, ob der Absender in der Nachricht mit dem Inhaber des Zertifikates übereinstimmt und ob die Nachricht unverändert angekommen ist. Der Ablauf der Prüfung ist in Abbildung 2.20 dargestellt.

Woher weiß man nun aber, welches der öffentliche Schlüssel der Zertifizierungsstelle ist? Man hat das Problem der Zuordnung eines öffentlichen Schlüssels zu einer Person in ein neues Problem umgewandelt: Welches ist der öffentliche Schlüssel der Zertifizierungsstelle? Man kann dieses Problem dadurch „lösen“, dass auch eine Zertifizierungsstelle ein Zertifikat besitzt. Dieses Zertifikat ist dann von einer übergeordneten Zertifizierungsstelle ausgestellt. Dadurch erhält man eine Hierarchie von Zertifizierungsstellen. Der öffentliche Schlüssel der obersten Zertifizierungsstelle muss aber trotzdem irgendwie bekannt sein.

Das könnte man beispielsweise dadurch erreichen, dass dieser Schlüssel täglich in großen Zeitungen abgedruckt wird, immer wieder im Fernsehen eingeblendet wird und täglich in die verschiedensten Newsgroups geschrieben wird. Alle diese Medien gleichzeitig zu fälschen dürfte unmöglich sein. Allerdings ist es für Benutzer auch nicht gerade einfach, einen langen Schlüssel (2048 Bit) zu prüfen. Deshalb komprimiert man den Schlüssel mit einem Hashverfahren und erhält einen digitalen „Fingerabdruck“ (engl. **fingerprint**) des Schlüssels. Dieser wird statt des Schlüssels veröffentlicht und kann von Menschen einfacher überprüft werden. Mit diesem öffentlichen Schlüssel kann man dann die Zertifikate der Zertifizierungsstellen prüfen und dann mit deren öffentlichen Schlüssel die Zertifikate der Benutzer prüfen.

Fingerprint eines Schlüssels

Bundesnetzagentur

In Deutschland hat die Bundesnetzagentur (früher Regulierungsbehörde

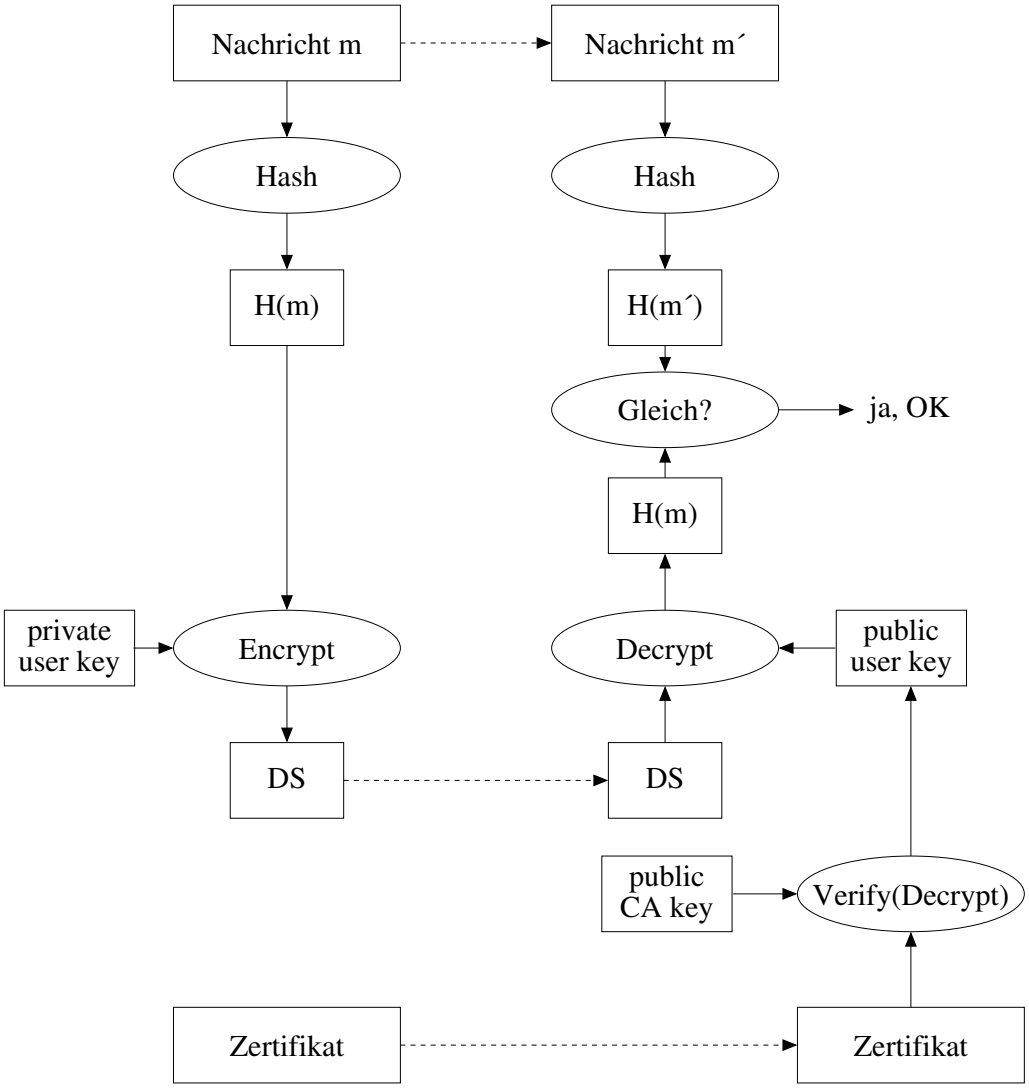

Abbildung 2.20: Zertifikate zur Authentizitätsprüfung

für Telekommunikation und Post (RegTP)) die Aufgabe der Wurzelzertifizierungsstelle übernommen. Ihr öffentlicher Schlüssel wird im Bundesanzeiger abgedruckt und parallel auch im Internet veröffentlicht. RegTP

```
Veröffentlicht im Bundesanzeiger Nr. 169 - Seite 3965 v. 10.11.2011

Root-CA-Schlüsselinformation 14:
Signaturalgorithmus: RSA, 2048 Bit

Seriennummer [dezimal]:
802

Issuer Distinguished Name und Subject Distinguished Name:
CN=14R-CA 1:PN
O=Bundesnetzagentur
C=DE

Modul n [hexadezimal]:
00a2 d867 6345 5ced b048 41dc 85c2 cc66 930a 0eb4 0a71
a07d 794a 3b45 f524 ff9f 78f1 60da 13b2 b64e 9539 b3d5
06a2 5734 016e 80f3 da1d 7289 0cf2 00a1 863f 0351 fa4b
89ba 76a8 e5ca 9548 adf4 ac92 df15 844f c124 7c1d 1caf
```

```
b90f bf63 1895 335c 99b5 6113 7ba0 c06e 955b 2dc4 9d30
e22b 71bd 88fb 7afa 6956 dbc0 c5a1 23a8 e3d9 a750 2bd7
1309 40e9 a6f2 e3fc 7d22 b8e5 545a 3511 aa6c 8fed 2e73
4fa6 2d75 e32c 0468 50eb ed30 d454 27f9 ac26 ae4f 0f82
7acf 58c8 d4ce 24c4 6606 41bb 1842 10e6 de48 4ba4 acd9
0923 2bbd 792b 8ea0 adb7 aa6a be99 87fd c477 e68f 5eb0
b1a7 db28 60c9 8f82 1d1b d5a8 71ae d76f dfc4 a4f7 fe2b
b837 f035 eb44 11cf 981b cedb 68ab 9f

Öffentlicher Exponent e [hexadezimal]:
4000 0081

Gültig von: 25.07.2011, 08:13:50 (UTC) d.h. 10:13:50 (MESZ)
Gültig bis: 24.07.2016, 08:08:08 (UTC) d.h. 10:08:08 (MESZ)
```

Im Signaturgesetz ist festgelegt, welche Anforderungen eine Zertifizierungsstelle (auch Zertifizierungsinstanz genannt) erfüllen muss, damit ihre Zertifikate Beweiskraft vor Gericht haben können. Hierzu gehören Zuverlässigkeit, Fachkunde und eine Deckungsvorsorge. Letzteres bedeutet, dass eine Zertifizierungsstelle bei Fehlern in der Lage sein muss, Schadensersatzansprüche zu befriedigen. Weiterhin werden im Signaturgesetz Maßnahmen beschrieben, mit denen eine Zertifizierungsstelle sicherstellen soll, dass keine falschen Zertifikate ausgegeben werden. Das sind technische (z. B. sichere Gebäude und sichere Systeme) und organisatorische (z. B. Vier-Augen-Prinzip) Maßnahmen. Diese Maßnahmen führen außerdem dazu, dass der Betrieb einer Zertifizierungsstelle mit einem bestimmten Aufwand verbunden ist.

Der Aufbau und Betrieb einer Zertifizierungsstelle scheint trotzdem ein neues und interessantes Geschäftsfeld. Daher versuchen verschiedene Firmen solche Zertifizierungsstellen aufzubauen. In Deutschland sind das beispielsweise die *Deutsche Telekom AG*, die *D-Trust GmbH* (eine Tochter der Bundesdruckerei), die *DATEV eG* (eine Genossenschaft die Datenverarbeitungsdienste anbietet) oder der *Deutsche Sparkassen Verlag*. Früher boten auch einige Rechtsanwaltskammern und Steuerberatungskammern digitale Zertifikate an. Sie haben den Betrieb aber inzwischen eingestellt. Auch das *TC Trust Center*, ursprünglich von privaten Banken in Deutschland gegründet hat nach mehrfachem Eigentümerwechsel angekündigt, den Betrieb der Zertifizierungsstelle einzustellen. Auf den Internetseiten der Bundesnetzagentur finden Sie eine aktuelle Liste mit den akkreditierten Zertifizierungsstellen in Deutschland. Es entsteht dann eine Hierarchie wie in Abbildung 2.21.

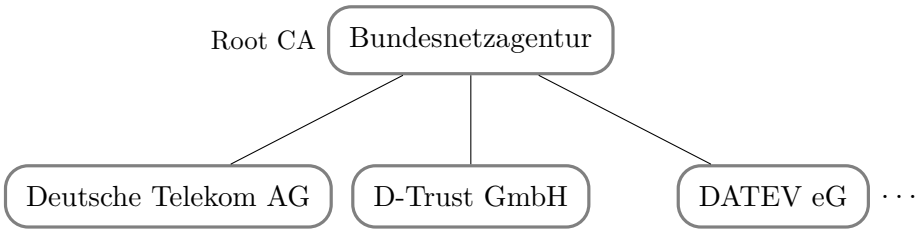

Abbildung 2.21: Hierarchie von Zertifizierungsstellen in Deutschland

Eine naheliegende Erweiterung dieser Hierarchie besteht darin, dass auch Firmen eine eigene Zertifizierungsstelle für ihre Mitarbeiter einrichten. Für große Firmen mit vielen Mitarbeitern ist das sinnvoll, denn die Mitarbeiter müssen der Firma ja genau bekannt sein. Die Identitätsprüfung findet also sowieso statt. Für die Firma könnte dieser Ansatz kostengünstiger sein als Zertifikate bei

anderen Anbietern für jeden Mitarbeiter zu kaufen. Beispielsweise betreibt auch das Rechenzentrum der FernUniversität eine Zertifizierungsstelle für Mitarbeiter und Studierende, siehe auch Abschnitt 2.7.3.

Allerdings scheuen viele Firmen den Aufwand, Zertifikate gemäß dem Signaturgesetz (sog. *qualifizierte Zertifikate*) zu erstellen. Eine digitale Signatur basierend auf einem qualifizierten Zertifikat (im Gesetz *qualifizierte elektronische Signatur* genannt) ist der eigenhändigen Unterschrift im Prinzip gleichgestellt. Das ist im Geschäftsverkehr aber oft gar nicht erforderlich. Firmen-CAs würden lieber weniger anspruchsvolle Zertifikate benutzen, die den praktischen Ansprüchen an Handhabung, Wirtschaftlichkeit und Sicherheit genügen. Somit würden Firmen-CAs außerhalb der Hierarchie in Abbildung 2.21 stehen. Das Zertifikat der Firmen-CA (mit dem öffentlichen Schlüssel der Firmen-CA werden ja die ausgestellten Zertifikate überprüft, mit dem privaten Schlüssel der Firmen-CA sind sie digital signiert) trägt also keine Unterschrift der Wurzel-CA.

qualifizierte Zertifikate
qualifizierte elektronische Signatur

Eine Bridge-CA signiert nun Zertifikate von CAs, die zwar nicht den Standards des Signaturgesetzes entsprechen, die dafür aber trotzdem bestimmte pragmatisch orientierte Standards erfüllen. Es entsteht eine Hierarchie wie in Abbildung 2.22. Weitere Informationen zur Bridge-CA, den Standards und Anforderungen finden Sie im Internet unter der URL

Bridge-CA

```
http://www.ebca.de/
```

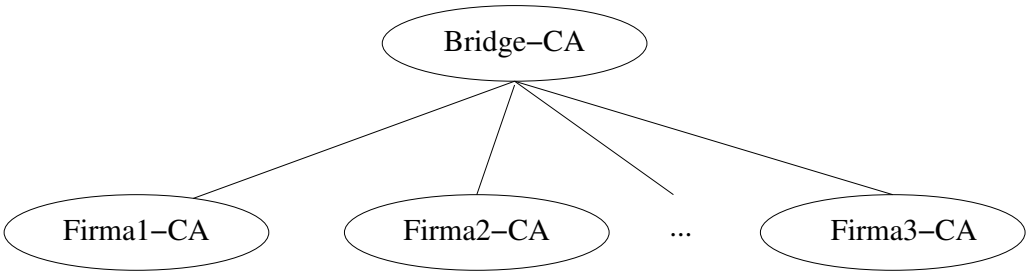

Abbildung 2.22: Hierarchie von Firmen-CAs mit Hilfe einer Bridge-CA

Wenn viele Firmen ihre Mitarbeiter mit Zertifikaten ausstatten, dann könnten die Mitarbeiter anschließend vertraulich und authentisch firmenübergreifend kommunizieren und dadurch die Geschäftsprozesse effizienter und kostengünstiger gestalten.

Die Zertifizierungsstellen aus anderen Ländern passen nicht direkt in diese Baumstruktur. Um Zertifikate solcher Zertifizierungsstellen auch benutzen zu können, gibt es die **Cross-Zertifikate**. Dabei signieren zwei Zertifizierungsstellen ihre öffentlichen Schlüssel gegenseitig.

Cross-Zertifikate

Erstellt eine Zertifizierungsstelle ein Zertifikat für einen Benutzer, so muss die Zertifizierungsstelle die Identität des Benutzers feststellen. Dabei kann man unterschiedlich viel Aufwand treiben. Das reicht von einem kurzen telefonischen Rückruf bis zum persönlichen Erscheinen des Benutzers bei der Zertifizierungsstelle. Dabei kann dann ein amtlicher Ausweis kontrolliert werden. Diese Unterschiede bei der Identitätsfeststellung führen dazu, dass verschiedene Zertifikatsklassen existieren.

Zertifikatsklassen

Die Firma *VeriSign* aus den USA vergibt drei Zertifikatsklassen. Zertifikate der Klasse 1 entsprechen der schwächsten Form der Identitätsprüfung. Ein

Benutzer kann das Zertifikat elektronisch beantragen. *VeriSign* prüft dann nur, ob die E-Mail-Adresse tatsächlich existiert und eindeutig ist. Da sich hierbei jedermann einen Alias-Namen geben kann, sagt ein Zertifikat dieser Klasse nicht viel aus. Man kann damit aber immerhin sicherstellen, dass eine Folge von E-Mails immer von derselben Person stammt.

Zertifikate der Klasse 2 sind bereits etwas aussagekräftiger. Die Zertifizierungsstelle prüft hierbei nicht nur die E-Mail-Adresse des Benutzers, sondern betrachtet auch die weiteren Angaben, wie Telefonnummer oder Adresse. Es wird allerdings nur geprüft, ob die Angaben des Benutzers mit denen in öffentlich zugänglichen Datenbanken übereinstimmen. Konkret heißt dies, dass die Telefonnummer beispielsweise im Telefonbuch steht oder dass die Straße in der angegebenen Ortschaft tatsächlich existiert. Da *VeriSign* eine amerikanische Firma ist, werden nur US-Datenbanken zur Überprüfung herangezogen. Daher können auch nur Amerikaner ein Zertifikat der Klasse 2 ausgestellt bekommen.

Die höchste Zertifizierungsstufe ist Klasse 3. Der Benutzer oder die Organisation müssen persönlich erscheinen oder von einem vertrauenswürdigen Dritten (z. B. Notar) vertreten werden. Zertifikate der Klasse 3 werden auch an Organisationen, wie Firmen, Universitäten, Regierungsstellen usw. vergeben. Die Identität von Personen wird durch Ausweise, die Existenz und Identität von Organisationen durch öffentliche Datenbanken (z. B. Handelsregister) und anderweitige Rückfragen geprüft. Anderweitige Rückfragen werden dabei nicht an den antragstellenden Repräsentanten, sondern an die öffentlich bekannten Anlaufstellen gestellt.

Die aufwendige Identitätsprüfung eines Zertifikats der Klasse 3 hat natürlich ihren Preis. Für ein Zertifikat Klasse 3 muss ein Benutzer also mehr bezahlen als für die weniger aussagekräftigen Zertifikate. Im E-Commerce sollte man diese Anforderung allerdings schon stellen. Wer bei einer Internetbank, die sich nicht durch ein Zertifikat der höchsten Stufe ausweist, Aufträge mit PIN und TAN eingibt, handelt fahrlässig. Schließlich könnte sich jemand für die Bank ausgeben und mit den gestohlenen Daten dann bei der echten Bank Aufträge einreichen. Wie man als „Websurfer“ Zertifikate prüft, wird in Abschnitt 3.3 angesprochen.

2.7.2 Schlüsselmanagement

Bei den in Abschnitt 2.3 und Abschnitt 2.4 vorgestellten Verschlüsselungsverfahren ist es für einen Angreifer normalerweise einfacher einen Schlüssel zu stehlen als eine verschlüsselte Nachricht ohne Kenntnis des Schlüssels zu „knacken“. Deshalb müssen die Anwender bei der Schlüsselhandhabung große Sorgfalt walten lassen. Das beginnt bei der Erstellung eines Schlüssels und setzt sich beim Speichern, Wechseln und Löschen eines Schlüssels fort.

Management öffentlicher Schlüssel: Wie oben bereits erwähnt, haben öffentliche Schlüssel und damit auch Zertifikate nur eine begrenzte Gültigkeit. Grund dafür ist, dass es mit intensivem Rechnereinsatz vielleicht doch möglich ist, aus einem öffentlichen Schlüssel den zugehörigen privaten Schlüssel zu berechnen. Außerdem könnte der private Schlüssel abhanden kommen oder gestohlen werden. Dann muss ein Benutzer die Möglichkeit haben, seinen öffentichen Schlüssel sperren zu lassen.

Je länger man einen Schlüssel benutzt, desto größer wird die Wahrscheinlichkeit, dass ein Brute-Force-Angriff erfolgreich ist. Daher ist es sinnvoll, regelmäßig neue Schlüssel zu erstellen und zu benutzen. Die Dauer dieser Aktualisierungsintervalle hängt von der Größe/Länge des Schlüssels ab und liegt i. d. R. im Bereich von Jahren. Beim Austausch eines Schlüssels darf man den alten Schlüssel allerdings nicht einfach löschen. Alte Schlüssel sind erforderlich, um beispielsweise auch in 10 Jahren noch eine digitale Unterschrift unter einem Dokument überprüfen zu können. Dazu muss also ein 10 Jahre alter Schlüssel noch vorhanden sein und dem damaligen Besitzer eindeutig zuzuordnen sein. Bei *PGP* (siehe auch Abschnitt 3.2.1) kann man auf den Schlüsselservern daher keinen Schlüssel löschen, sondern alte Schlüssel nur als „veraltet“ markieren.

Die Menge der vorhandenen Schlüssel wird also nur größer, nie kleiner. Auf Dauer wird das eine Herausforderung für die Verwaltung dieser Schlüsselmengen.

Management privater Schlüssel: Nicht nur die öffentlichen Schlüssel der Kommunikationspartner müssen verwaltet werden. Auch der private Schlüssel muss sicher aufbewahrt werden. Im folgenden geht es darum, dass (1) manchmal nicht nur der Besitzer des privaten Schlüssels diesen kennen sollte und (2) ein privater Schlüssel immer sicher aufbewahrt werden muss.

Zu (1): Verschlüsselte Kommunikation ist für staatliche Ermittlungsstellen ein schwieriges Problem. Auch Verbrecher können damit abhörsicher kommunizieren. Deshalb gibt es Bestrebungen, eine Schlüsselhinterlegung (engl. **key escrow**) gesetzlich zu verlangen. Diese Idee wurde beispielsweise beim *Clipper-Chip* in den USA verwirklicht. Der (vollständige) private Schlüssel war dabei nur dem Besitzer selbst bekannt. Allerdings wurde der private Schlüssel in zwei Hälften zerlegt und bei zwei staatlichen Agenturen wurde je eine Hälfte gespeichert. Jede der Hälften ist für sich alleine unbrauchbar. Eine Agentur kann also alleine keine Abhörmaßnahmen durchführen. Auf richterliche Anordnung stellen jedoch beide Agenturen ihre Hälften zur Verfügung. Die Ermittlungsbehörden können dann den privaten Schlüssel rekonstruieren.

Clipper

In diesem Fall kann die gesamte Kommunikation, also auch die in der Vergangenheit erfolgte Kommunikation, gelesen werden. Die Kenntnis des privaten Schlüssels erlaubt es dann auch, in Zukunft jede Nachricht mitzulesen, die mit diesem Schlüssel verschlüsselt wird. Da der private Schlüssel fest in den Chip „eingebrannt“ ist, kann der Anwender den Schlüssel nicht einfach ändern. Kritiker der Schlüsselhinterlegung haben weitere Argumente:

- Man kann vor der Verschlüsselung mit dem Clipper-Chip eine andere Verschlüsselung ausführen. Die Algorithmen sind öffentlich bekannt und es ist im Prinzip kein Problem, sie zu implementieren.
- Die bereits kurz erwähnte *Steganografie* erlaubt es, geheime Nachrichten in unverfänglichen Nachrichten zu verstecken. Binäre Bilddateien sind als unverfängliche Nachricht gut zu gebrauchen. Kleine Veränderungen an der Datei verändern den optischen Eindruck oft nur unwesentlich.

Der Clipper-Chip hat sich am Markt nicht durchsetzen können.

Es gibt allerdings auch gute und legitime Gründe für eine Schlüsselhinterlegung. Für eine Firma wäre es unter Umständen katastrophal, wenn ein Mitarbeiter seinen Schlüssel verliert oder er die Firma verlässt und es dann unmöglich würde, seine dienstliche Korrespondenz nachlesen zu können.

Um in solchen Fällen die Dokumente, beispielsweise alte Verträge, lesen zu können, muss der Schlüssel rekonstruierbar sein. Daher würde man auch hier den Schlüssel spalten und die Einzelteile bei separaten Autoritäten speichern.

Wichtig ist in diesem Fall allerdings, dass ein Mitarbeiter mehrere Schlüssel hat. Der Schlüssel zum Signieren von Dokumenten sollte verschieden vom Schlüssel zum Verschlüsseln sein. Schließlich soll ein Arbeitgeber zwar immer die alte Post eines Mitarbeiters lesen können, es sollte aber unmöglich sein, im Nachhinein die Unterschrift des Mitarbeiters zu wiederholen, bzw. zu fälschen.

Für den Fall, dass ein (privater) Schlüssel verloren geht, muss auch Vorsorge getroffen werden. Ein Benutzer kann in diesem Fall sein Zertifikat von der Zertifizierungsstelle sperren lassen. Es wird in eine Sperrliste (engl. **Certificate Revocation List (CRL)**) eingetragen. Die komplette Überprüfung eines Zertifikats ist also etwas aufwendiger, als in Abbildung 2.20 unten rechts dargestellt. Sie besteht aus den folgenden Schritten:

Certificate Revocation List

1. Überprüfen des Zertifikats der Zertifizierungsstelle. Dies kann bedeuten, dass man sich die Hierarchie der Zertifizierungsstellen von unten nach oben „durchhangeln“ muss, um dann evtl. noch einem Cross-Zertifikat zu folgen.

2. Überprüfen, ob das Benutzerzertifikat oder das Zertifikat einer der Zertifizierungsstellen vielleicht gesperrt ist. Dazu werden die Certificate-Revocation-Lists der Zertifizierungsstellen benutzt.

Zu (2): Da private Schlüssel recht groß sind, kann sich ein Benutzer sie nicht einfach merken und bei Bedarf eintippen. Daher wird man den privaten Schlüssel speichern. Es reicht allerdings nicht, den privaten Schlüssel einfach in einer Datei auf dem eigenen Rechner zu speichern. Die Gefahr, dass ein Dritter die Datei stehlen könnte, ist viel zu groß. Man speichert den privaten Schlüssel daher in einer „sicheren persönlichen Umgebung“ (engl. **Personal Security Environment (PSE)**).

Personal Security Environment

Chipkarte

Die sichere Umgebung kann beispielsweise eine Chipkarte sein. So kann man seinen Schlüssel immer bei sich haben. Weiterhin wird der Schlüssel auf der Chipkarte durch eine persönliche Identifikationsnummer (engl. **Personal Identification Number (PIN)**) geschützt. Eine Chipkarte hat außerdem den Vorteil, dass sie auch einen Prozessor enthält. Der Schlüssel muss die Karte also nicht verlassen, wenn ein Dokument signiert oder eine Datei entschlüsselt werden muss. Diese Operationen können vom Chipkartenprozessor ausgeführt werden. Der Ablauf ist dabei wie folgt:

1. Die Software des Rechners berechnet den Hashwert des zu signierenden Dokumentes.

2. Der Hashwert wird an die Chipkarte übertragen.

3. Der Benutzer muss am Chipkartenleser seine korrekte PIN eingeben.

4. Die Chipkarte verschlüsselt den Hashwert mit dem auf ihr gespeicherten privaten Schlüssel.

5. Das Ergebnis wird wieder auf den Rechner übertragen und von der Software an die Nachricht angehängt.

Heute (August 2013) sind Chipkartenleser allerdings immer noch wenig verbreitet. Durch den neuen Personalauweis wird demnächst allerdings jeder in Deutschland eine signaturfähige Chipkarte besitzen (siehe auch Abschnitt 2.7.4). Zusammen mit einem Kartenleser können dann signaturgesetzkonforme digitale Signaturen (im Gesetz qualifizierte elektronische Signatur genannt) erstellt werden.

Bis dahin wird man seinen privaten Schlüssel noch in einer Datei *symmetrisch verschlüsselt* abgelegen müssen. Bei PGP heißt dieser symmetrische Schlüssel Passwortsatz (engl. **pass phrase**) oder auch „Mantra“. Diesen kann sich der Benutzer besser merken. Bevor man nun Nachrichten signieren oder entschlüsseln kann, muss man den symmetrischen Schlüssel eingeben.

USB-Stick

Wenn man den verschlüsselten Private Key nun auf einem USB-Stick speichert, dann kann man seinen Schlüssel im Prinzip immer dabei haben. Bevor man den Schlüssel aber in einer „unsicheren Umgebung“, beispielsweise einem Internetcafé benutzt, sollte man bedenken, dass Angreifer den Schlüssel möglicherweise kopieren könnten. Falls dann auch das Mantra, beispielsweise durch Mitlesen der Tastatureingaben, nicht mehr geheim ist, kennt der Angreifer dann Ihren privaten Schlüssel. Er kann sich nun für Sie ausgeben oder alle Ihre privaten Nachrichten entschlüsseln.

Zum Entschlüsseln müssen Sie zuerst das Mantra eingeben, danach wird damit der private Schlüssel entschlüsselt. Der Entschlüsselungsalgorithmus hat nun den privaten Schlüssel im Hauptspeicher vorliegen und entschlüsselt damit die Nachricht. Das Betriebssystem muss nun sicher stellen, dass kein anderer Benutzer den Hauptspeicherbereich mit dem privaten Schlüssel lesen kann. Falls das Betriebssystem Teile des Hauptspeichers bei Bedarf auf die Festplatte auslagert (engl. **paging**), dann liegt der private Schlüssel dort sogar im Klartext vor. Das Betriebssystem muss auch hier sicher stellen, dass kein Unbefugter diese Datei lesen kann.

2.7.3 Public-Key-Infrastrukturen (PKI)

Mit den in Abschnitt 2.7.1 vorgestellten Techniken kann man nun theoretisch jeden Benutzer eindeutig identifizieren. Somit kann man auch sicher, d. h. vertraulich und authentisch, verschiedene Dienste nutzen. Dazu gehören:

- elektronischer Handel jeder Art,
- Teilnahme an elektronischer Verwaltung jeder Art, wie beispielsweise elektronische Steuererklärung, oder
- Internetbanking und -broking

Tatsächlich sind diese Dienste heute zwar möglich, sie benutzen heute zur Authentisierung der Benutzer aber i. d. R. keine Zertifikate. Stattdessen werden Passwörter (Benutzerkennungen im E-Commerce beispielsweise bei *eBay* oder *Amazon*) oder PIN/TAN-Verfahren (beim Internetbanking und -broking) eingesetzt. In der elektronischen Verwaltung, beispielsweise bei der Steuererklärung, verlangt der Gesetzgeber eine eigenhändige Unterschrift oder das elektronische Äquivalent. Deshalb wird die Steuererklärung zwar elektronisch übermittelt, man muss parallel aber auch einen unterschriebenen Mantelbogen an das Finanzamt schicken. Woran liegt es, dass Zertifikate noch nicht in größerem Umfang eingesetzt werden?

Technische Standards: Damit sich Public-Key-Infrastrukturen durchsetzten können, muss sichergestellt sein, dass jeder Benutzer die Zertifikate auch einsetzen kann. Das bedeutet, dass die Anwendungsprogramme (die ja stellvertretend für den Benutzer agieren) zunächst einmal Zertifikate lesen können müssen. Am besten sollte ein einheitliches Datenformat für Zertifikate existieren.

X.509

Zur Zeit existiert mit **X.509** ein internationaler Standard für die Syntax und Semantik von Zertifikaten. In RFC 3280 [Hou+02] wird das Grundprinzip erklärt sowie die Syntax der Zertifikate festgelegt. Ein wesentliches Problem besteht darin, wie die Gültigkeit von Zertifikaten überprüft werden kann. Der ursprünglich vorgesehene, streng hierarchische Aufbau einer Zertifizierungshierarchie ist unpraktisch. Einerseits wäre dann einfach zu prüfen, ob ein Zertifikat gültig ist: Man muss nur das Wurzelzertifikat kennen und kann dann die Zertifikate der Zertifizierungsstellen prüfen. Das entspricht einem Top-Down-Durchlauf durch die Zertifizierungshierarchie. Man setzt hierbei voraus, dass alle Zertifizierungsstellen dieselben Maßstäbe bei der Überprüfung desjenigen anlegen, der das Zertifikat bekommen möchte. Andererseits wären ihre Zertifikate nicht mehr sinnvoll überprüfbar, falls Firmen davon abweichen wollen und eigene, weniger strenge Zertifizierungsstellen einrichten wollen. Im RFC 3280 ist daher auch ein Algorithmus definiert, wie Zertifikate überprüft werden können. Neben dem oben genannten Top-Down-Durchlauf können auch andere Pfade durchlaufen werden.

Zusätzlich wird in RFC 3279 [BPH02] erklärt, welche Verschlüsselungsalgorithmen in X.509-Zertifikaten eingesetzt werden dürfen. Außerdem wird festgelegt, wie ein Verschlüsselungsalgorithmus denn genau bezeichnet wird, also welcher Name mit welcher genauen Schreibweise (vergleiche Triple-DES vs. 3DES) verwendet wird.

Zur Zeit werden Zertifikate im E-Commerce häufig eingesetzt. Webserver (z. B. der Ihrer Bank) identifizieren sich damit sicher gegenüber den Benutzern im Internet. Außerdem kann durch diese Technik (genannt SSL, siehe Abschnitt 3.3.1) auch eine vertrauliche Kommunikation sichergestellt werden. An der FernUniversität können Sie als Studierende auch ein SSL-Zertifikat vom Rechenzentrum bekommen. Damit können Sie sich dann gegenüber einigen Systemen der FernUniversität authentisieren und beispielsweise Ihre eigenen Klausurergebnisse abrufen. Details finden Sie unter der URL

`http://ca.fernuni-hagen.de/`

Interoperabilität

Neben Browsern müssen aber auch E-Mail-Programme den Umgang mit Zertifikaten beherrschen. Besonders wichtig ist hier die Interoperabilität, d. h. Benutzer A hat mit Programm P_1 eine signierte E-Mail an Benutzer B geschickt, der Programm P_2 benutzt. Trotzdem kann Benutzer B das Zertifikat von Benutzer A lesen und prüfen, sowie die Signatur der E-Mail überprüfen.

Für Firmen ist es weiterhin interessant, dass sich Mitarbeiter auch beim Betreten der Firma authentisieren. In vielen großen Firmen existieren hierfür bereits chipkartenbasierte Lösungen. Am besten wäre es, wenn diese Chipkarten auch für die Authentisierung am Rechner (bereits beim Login) und die Authentisierung an Web-Anwendungen eingesetzt werden können. Man möchte also eine Technik, die transparent in diesen Anwendungsfeldern funktioniert:

- Zutrittssysteme
- Computer allgemein

- speziellen Anwendungen gegenüber

In all diesen Anwendungsfeldern existieren aber schon inkompatible Techniken, die geändert werden müssten. Es ist also damit zu rechnen, dass der Einsatz von Zertifikaten nur nach und nach zunimmt.

Gesetzliche Anforderungen: Im Signaturgesetz sind die Anforderungen festgelegt, die Zertifikate und Zertifizierungsstellen erfüllen müssen, damit die digitale Signatur vor Gericht der handschriftlichen Unterschrift gleichgestellt ist. Solche digitalen Signaturen nennt der Gesetzgeber **qualifizierte elektronische Signatur**. Der Gesetzgeber hat die Anforderungen sehr hoch gelegt, damit Missbrauch nicht möglich ist. Das hat zur Folge, dass die Erfüllung dieser Anforderungen mit vergleichsweise großem Aufwand verbunden ist.

qualifizierte elektronische Signatur

Es stellt sich daher natürlich die Frage: „Braucht man in der Praxis denn tatsächlich qualifizierte Zertifikate oder reichen auch einfachere?“ Für Sie als Studierende der FernUniversität reichen auch „einfachere“ Zertifikate. Ausgestellt werden diese Zertifikate vom Rechenzentrum der FernUniversität - eine Zertifizierungsstelle, die nicht den strengen Anforderungen des Signaturgesetzes entspricht. Für den sicheren Zugriff auf Ihre Studentendaten ist es aber in Ordnung, da alle Beteiligten zu derselben Organisation gehören.

Wollen Sie sich aber Dritten (also *nicht* der FernUniversität) gegenüber authentisieren, dann müsste dieser Dritte (1) dem Rechenzentrum der FernUniversität hinreichend viel Vertrauen entgegen bringen und (2) den öffentlichen Schlüssel des Rechenzentrums kennen. Punkt (2) bedeutet, dass der Dritte dann auch das Zertifikat des Rechenzentrums der FernUniversität bei sich installieren müsste. Benutzen nun viele Hochschulen oder viele Firmen eigene, nicht mit dem „Signaturgesetz konforme“ Zertifikate, dann ist das Verfahren nicht mehr praktikabel. Es müssten zu viele „einfache“ Zertifikate auf vielen Systemen i. d. R. manuell installiert und gepflegt werden.

Wirtschaftliche Aspekte: Bevor Zertifikate weiträumig eingesetzt werden können, sind verschiedene Vorarbeiten erforderlich. Hierzu gehören unter anderem:

- Aufbau und Betrieb von vertrauenswürdigen Zertifizierungsstellen
- Entwicklung und Verbreitung von Anwendungsprogrammen, die mit Zertifikaten umgehen können und die unter einander Daten austauschen können
- Entwicklung und Verbreitung der erforderlichen zusätzlichen Hardware

All diese Aufgaben sind mit Kosten verbunden. Privatwirtschaftliche Firmen investieren jedoch nur, wenn ein zukünftiger Gewinn absehbar ist. Dazu ist eine entsprechende Nachfrage nach Zertifikaten, Software und Hardware erforderlich. Diese Nachfrage entsteht aber erst dann, wenn auch entsprechende Angebote zur Verfügung stehen. Man hat es hier also mit dem klassischen Henne-Ei-Problem zu tun. Ohne Nachfrage wird kein Angebot erstellt. Ohne die Angebote entsteht keine Nachfrage. Verschärft wird dieses Problem durch die oben genannten technischen und gesetzlichen Unwägbarkeiten.

Henne-Ei-Problem

2.7.4 Der neue Personalausweis

Seit November 2010 gibt es in Deutschland den neuen Personalausweis (nPA). Abbildung 2.23 zeigt die etwa scheckkartengroße Vorderseite. Er ist mit einem kontaktlosen RF[5]-Chip ausgestattet und bietet dadurch zusätzliche Funktionen [Ben+08]. Dieser Chip enthält einen Speicher für Daten und Prozessoren, die

Abbildung 2.23: Muster des neuen Personalausweis. Quelle: Bundesministerium des Inneren

auch kryptografische Operationen durchführen können. Der Speicher besteht aus einem auslesbaren Teil, in dem u. a. auch die aufgedruckten Informationen noch einmal elektronisch gespeichert sind. Dazu kommen weitere Daten wie Fingerabdrücke. Daneben gibt es auch einen geschützten Speicherbereich. In ihm stehen z. B. die privaten Schlüssel des Inhabers, die den Chip niemals verlassen sollen, oder die Wurzelzertifikate der PKI. Der geschützte Speicherbereich kann nicht ausgelesen und somit auch nicht kopiert werden. Nur der Prozessor des Chips kann auf diesen Speicher zugreifen und damit digitale Signaturen erstellen oder vorgelegte Zertifikate überprüfen.

Funktionen: Der neue Personalausweis dient wie der alte auch als Authentisierungsmittel bei Kontrollen aller Art. Zusätzlich bietet er folgende Funktionen:

ePass

1. Die sogenannte *ePass*-Anwendung. Dabei können die im Chip gespeicherten biometrischen Daten des Inhabers (Lichtbild, Fingerabdruck) beispielsweise bei Grenzkontrollen ausgelesen werden.

Authentisierungs-Funktion

2. Mit der *Authentisierungs-Funktion* (auch *eID* oder *eAuthentisierung* genannt) kann sich er Inhaber online bestimmten Diensten gegenüber authentisieren. Außerdem kann eine anonyme Altersüberprüfung durchgeführt werden.

qualifizierten elektronischen Signaturen

3. Die dritte Funktion ist optional. Sie erlaubt die Erstellung von *qualifizierten elektronischen Signaturen*, also digitale Signaturen die die Anforderungen aus dem Signaturgesetz erfüllen. Dazu muss der Inhaber allerdings ein Zertifikat bei einer zugelassenen Zertifizierungsstelle erwerben und auf dem Chip im Ausweis speichern.

Im Chip werden die personenbezogenen Daten parallel zum Aufdruck auf dem Ausweisdokument gespeichert. Der Name des Inhabers steht also an drei Stellen: (1) Ausgedruckt auf der Vorderseite des Ausweises, (2) ausgedruckt in der

MRZ

Machine Readable Zone (MRZ) unten auf der Rückseite wie in Abbildung 2.24

[5] RF = radio frequency

gezeigt und (3) unsichtbar im Chip im Inneren des Ausweises. Diese perso-

Abbildung 2.24: Rückseite des neuen Personalausweis. Quelle: Bundesamt für Sicherheit in der Informationstechnik

nenbezogenen Daten werden bei der Herstellung des Ausweises auf den Chip kopiert und digital signiert. Sie können später nicht mehr verändert werden.

Bedrohungen: Eine Reihe von unerwünschten Benutzungen des Ausweises müssen ausgeschlossen sein. Zu den unerwünschten Einsatzformen gehören:

Fälschungen erstellen: Niemand soll in der Lage sein, die Daten des Ausweises unbemerkt zu verändern oder eine Kopie des Ausweises zu erstellen.

Unerlaubt Daten auslesen: Niemand soll unerlaubt Daten des Ausweises auslesen können. Wenn man den Ausweis bei sich trägt, z. B. in der Hosentasche, dann soll trotzdem niemand, der hinter einem steht, die Daten mit einem RFID-Kartenleser auslesen können.

Zu viele Daten auslesen: Auch wenn autorisierte Lesegeräte auf die Daten grundsätzlich zugreifen dürfen, soll der Inhaber des Ausweises kontrollieren können, welche Daten des Ausweises denn an ein Lesegerät weitergegeben werden sollen. Die Fingerabdruckdaten sollen bei der Grenzkontrolle ausgelesen werden (dann kann der Personalausweis den Reisepass ersetzen), aber nicht von anderen Geräten.

Sicherheitsmechanismen

Um diesen Bedrohungen zu begegnen, sind verschiedene Sicherheitsmechanismen vorgesehen.

Sicherheitsmechanismen: Wie schon gesagt sind die gespeicherten Daten digital vom Hersteller signiert. In jedem Land gibt es eine zweistufige PKI mit der sogenannten *Country Signing Certification Authority (CSCA)* als Wurzel. Sie signiert die Zertifikate der *Document Signer (DS)*, also der Hersteller von Ausweisen. Lesegeräte an Grenzkontrollstellen prüfen diese digitalen Signaturen.

Hacker könnten allerdings den kompletten Chip kopieren und das Lesegerät sollte daher auch die Authentizität des Chips selbst prüfen können. Dazu hat der Ausweishersteller ein Schlüsselpaar für den Chip selbst erstellt. Der private Teil des Schlüssels steht dabei im nicht lesbaren Speicherbereich, der nicht kopiert oder ausgelesen werden kann. Der öffentliche Schlüssel des Chips steht digital signiert im lesbaren Teil des Speichers und kann somit vom Lesegerät benutzt werden.

Damit nur autorisierte Lesegeräte die Daten des Ausweises auslesen können, gibt es einen sogenannten Zugriffsschutz auf den Ausweis (vgl. Kügler und Naumann [KN07]). Man spricht hier auch vom **Basic Access Control (BAC)**. Aus einigen Daten in der MRZ (Passnummer, Geburtsdatum und Ablaufdatum) kann das Lesegerät einen Schlüssel berechnen und sich mit diesem dann dem

Basic Access Control (BAC)

Ausweis gegenüber authentisieren. Dazu ist also optischer Zugriff auf die MRZ des Ausweises erforderlich. Kann ein Angreifer aber vermuten, dass diese Daten keine Zufallsdaten sind, sondern bestimmten Mustern genügen, dann kann er einen Brute Force Angriff auf diesen Authentisierungsschlüssel durchführen. Bereits 2006 wurde in den Niederlanden ein solcher Angriff erfolgreich demonstriert. Glücklicherweise erlaubt BAC nur den Zugriff auf eher unkritische personenbezogene Daten im Chip des Ausweises.

Extended Access Control (EAC)

Ein erweiterter Zugriffsschutz (**Extended Access Control (EAC)**) ist für den Zugriff auf sensitivere Daten erforderlich. Hierbei muss sich ein Lesegerät dem Chip gegenüber authentisieren und auch autorisieren. Dieses Verfahren wird *Terminal-Authentisierung* genannt und basiert auch auf einer PKI. In jedem Land gibt es eine *Country Verifying Certificate Authority (CVCA)* als Wurzel und dann sogenannte *Document Verifier (DV)*. Die CVCA signiert dabei nicht den Schlüssel des DV, sondern trägt im Zertifikat auch ein, auf welche Daten des Chips dieser DV denn Zugriff erhalten soll. Der Chip im Ausweis kann mit Hilfe dieses Zertifikats also das Lesegerät authentisieren und gleichzeitig auch feststellen, welche Daten es diesem Lesegerät denn nun senden darf. Diese beiden Verfahren, Basic Access Control und Extended Access Control, sind auch im elektronischen Reisepass implementiert.

Terminal-Authentisierung

CVCA

DV

Password Authenticated Connection Establishment (PACE)

Im elektronischen Personalausweis gibt es ein weiteres Zugriffsschutzverfahren. Das **Password Authenticated Connection Establishment (PACE)** erweitert BAC. Während bei BAC nur ein aus dem Inhalt der MRZ abgeleiteter symmetrischer Schlüssel benutzt wird, werden bei PACE mehrere Passwörter benutzt. Auf dem Ausweis selbst wird eine Karten-PIN (engl. **Card Access Number (CAN)**) gedruckt und sie muss am Lesegerät eingegeben werden. Alternativ kann auch der Inhaber eine eID-PIN selbst wählen. Sie hat sechs Stellen und muss beim PACE-Verbindungsaufbau am Lesegerät eingegeben werden.

Der Chip im Ausweis erzeugt nun eine Zufallszahl, verschlüsselt sie mit der o. g. PIN und sendet das Ergebnis an das Lesegerät. Das entschlüsselt die Zufallszahl, und Chip und Lesegerät berechnen nun aus der Zufallszahl einen Generator für die Diffie-Hellman Gruppe (vergleiche Abschnitt 2.4.1, authentisiertes Diffie-Hellman mit einem gemeinsamen Geheimnis). Danach benutzen das Lesegerät und der Chip im Ausweis ein Elliptic Curve Diffie-Hellman (ECDH) Protokoll mit dem berechneten Generator, um sich zu authentisieren und einen symmetrischen Sitzungsschlüssel zu berechnen. Danach können Lesegerät und Chip sicher miteinander kommunizieren. Die Sicherheit des Schlüssels hängt bei diesem Verfahren nicht von der Sicherheit der PIN ab.

Für die eID-Funktion möchte man sich aber auch gegenüber *entfernten Systemen* authentisieren. Den Ausweis selbst kann man aber nur vor ein lokales Lesegerät halten. Man braucht also eine Erweiterung, so dass auch ein entferntes System/ein entfernter Dienst mit dem lokalen Lesegerät und dem lokal vorhandenen Ausweis kommunizieren kann. Nachdem Ausweis und Lesegerät den oben beschriebenen symmetrischen Sitzungsschlüssel berechnet haben, wird das Extended Access Control Verfahren zwischen entferntem System/Dienst und dem Chip auf dem Ausweis durchgeführt. Das entfernte System braucht dazu ein Zertifikat, mit dem nicht nur die Identität des entfernten Systems geprüft wird, sondern auch die Berechtigungen des entfernten Systems. Im Prinzip findet wieder ein authentisiertes Diffie-Hellman-Verfahren, diesmal mit öffentlichen Schlüsselpaaren der Beteiligten, statt.

Anhand des Zertifikats kann der Chip im Ausweis außerdem auch prüfen, welche Daten das entfernte System lesen darf. Bevor der Chip diese Daten tatsächlich ausliefert, wird der Ausweisinhaber auch explizit um seine Zustimmung gefragt. Ein Altersüberprüfungsdienst bekommt vom Chip im Ausweis also nur mitgeteilt, ob der Ausweisinhaber älter als in der Anfrage des Prüfungsdienstes ist oder nicht. Das genaue Alter erfährt der Prüfungsdienst somit nicht. Auf dem Weg zwischen Chip und entferntem System sind die Daten dann symmetrisch mit dem ausgehandelten Schlüssel verschlüsselt.

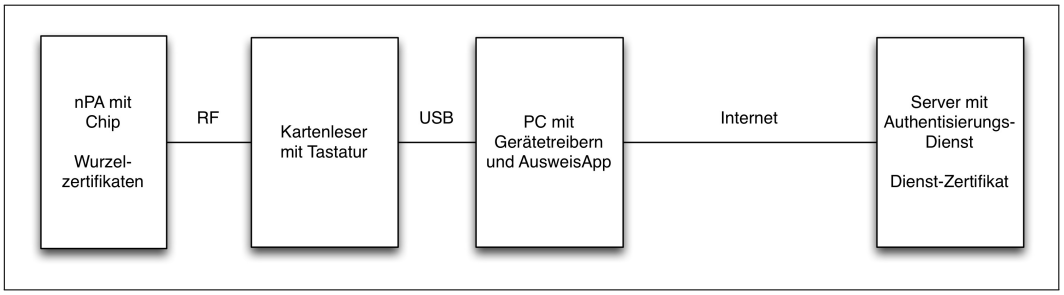

Abbildung 2.25: Beteiligte Systeme beim Einsatz des nPA

In Abbildung 2.25 sind die beteiligten Systeme dargestellt. Der komplette Ablauf bei der Nutzung von PACE bei der Authentisierung ist nun wie folgt:

1. Der Chip im Ausweis und der Kartenleser handeln einen authentisierten Diffie-Hellman Schlüssel aus. Die PIN ist das dabei benutzte gemeinsame Geheimnis. Falls der Kartenleser keine eigene Tastatur für die PIN-Eingabe besitzt, dann muss die PIN am PC eingegeben werden. Dazu gibt es eine AusweisApp.

   Nun können die Daten über die RF-Strecke verschlüsselt übertragen werden. Ein Angreifer, der die Funkwellen mitschneidet kann keine Informationen mitlesen.

2. Der Chip authentisiert das Terminal. Dazu sendet das Terminal sein Zertifikat und das des Document Verifier an den Chip. Der Chip prüft das Zertifikat mit Hilfe der Country Verifying Certificate Authority, deren Zertifikat im Chip gespeichert ist. Nun kennt der Chip den öffentlichen Schlüssel des Terminals.

3. Als nächstes authentisiert sich der Chip gegenüber dem Terminal. Dazu wird der öffentliche Schlüssel des Chips vom Terminal ausgelesen.

4. Im letzte Schritt wird auf Secure Messaging, also verschlüsselte Kommunikation zwischen Chip und Terminal, umgeschaltet. Dazu wird symmetrische Verschlüsselung eingesetzt. Der Schlüssel dazu wurde durch authentisiertes Diffie-Hellman berechnet. Hierbei wurden die Schlüsselpaare aus den Schritten 2. und 3., also dem Zertifikat des Terminals und die Schlüssel des Chips, benutzt.

Der Chip hat sich für diese Verbindung auch gemerkt, welche Daten aus dem Chip das Terminal denn nun lesen darf. Entsprechende Anfragen beantwortet der Chip.

Die Details des Verfahrens stehen in den Technical Guidelines des Bundesamts für Sicherheit in der Informationstechnik [BSI12a; BSI12b; BSI12c].

Digitale Signaturen sind mit dem neuen Personalausweis auch möglich. Dazu muss man sich aber eine Zertifizierungsstelle suchen und dort ein persönliches Zertifikat kaufen. Man muss sich also ein Schlüsselpaar erstellen, den öffentlichen Schlüssel und seine persönlichen Daten in einen sogenannten Certificate Signing Request (CRS) packen und den bei der Zertifizierungsstelle einreichen. Wurde dort dann die Authentizität überprüft, wird das entsprechende Zertifikat erstellt und auf den Chip im neuen Personalausweis kopiert.

Mit dem neuen Personalausweis besitzt bald jeder in Deutschland ein Chipkartensystem, mit dem man sich im Internet authentisieren kann. Diese Funktion muss man allerdings explizit freischalten lassen. Wie schnell entsprechende Dienste angeboten werden bleibt abzuwarten.

Außerdem kann man mit dem nPA auch digitale Signaturen erstellen. Hierfür muss man sich zusätzlich ein Zertifikat besorgen und auf den Chip im Ausweis kopieren.

2.7.5 Identity Based Encryption

In den Public-Key-Verfahren, die wir bisher kennengelernt haben, war es so, dass sowohl der öffentliche als auch der private Schlüssel aus einem Geheimnis abgeleitet waren. Beim RSA-Verfahren bestand dieses Geheimnis aus den Primzahlen p und q, deren Produkt $n = pq$ einen Teil des öffentlichen Schlüssels bildete. Beim El-Gamal-Verfahren bestand das Geheimnis aus dem Wert x, mittels dessen der öffentliche Schlüssel $y = g^x$ gebildet wurde.

Wenn man hingegen bei einem Public-Key-Verfahren den öffentlichen Schlüssel eines Benutzers aus einer bereits über ihn verlässlich und öffentlich bekannten Information ableiten könnte, dann könnte man auf Techniken wie Zertifikate verzichten. Könnte man zum Beispiel aus der E-Mail-Adresse eines Benutzers A seinen öffentlichen Schlüssel ableiten, dann könnte man eine E-Mail an A verfassen, die man mit diesem abgeleiteten öffentlichen Schlüssel verschlüsselt und an seine E-Mail-Adresse schickt. Wird die E-Mail von jemand anders empfangen, kann dieser sie nicht entziffern, da er nicht den passenden privaten Schlüssel hat.

Genauso könnte man eine von A signierte E-Mail dadurch auf Echtheit prüfen, dass man aus der E-Mail-Adresse den öffentlichen Schlüssel bestimmt und dann die Signatur prüft wie in Abschnitt 2.6.2 beschrieben. Ein Angreifer könnte zwar durch Fälschung die E-Mail-Adresse des Absenders benutzen, hätte allerdings nicht seinen privaten Schlüssel und könnte deshalb keine gültige Signatur erzeugen.

Natürlich darf der private Schlüssel eines Benutzers *nicht* alleine aus der öffentlich bekannten Information (wie der E-Mail-Adresse) ableitbar sein. Hierfür braucht man eine vertrauenswürdige Instanz, die ein Geheimnis einfließen lässt. Allerdings muss jeder Nutzer diese Instanz nur kontaktieren, wenn er eine E-Mail empfangen oder signieren will. Wenn man eine E-Mail an jemand anderen schicken oder dessen Signatur überprüfen will, braucht man den vertrauenswürdigen Dritten nicht, wie die obigen Ausführungen zeigen.

Adi Shamir hat bereits 1984 die Vision eines solchen Systems vorgestellt [Sha84], und auch ein Verfahren zur Identitäts-basierten Signaturprüfung entwickelt. Allerdings fand er kein Verfahren zur Identitäts-basierten Verschlüsselung. Das von ihm mitentworfene RSA-Verfahren eignet sich zum Beispiel dafür nicht. Der öffentliche RSA-Schlüssel eines Nutzers besteht aus dem Modulus $n = pq$

und dem Exponenten e. Der Modulus lässt sich aber nicht aus öffentlichen Informationen herleiten, da man dazu die zwei Primzahlen p und q aus den öffentlichen Informationen herleiten können müsste. Damit könnte man aber auch den privaten Schlüssel herleiten. Für alle Benutzer den gleichen Modulus zu nutzen, stellt leider auch keinen Ausweg dar, da dann der Common-Modulus-Angriff durchgeführt werden kann, und so aus dem eigenen privaten Schlüssel und dem öffentlichen Exponenten eines anderen Benutzers auch dessen privater Schlüssel berechenbar ist.

Gäbe es dieses Problem nicht, dann könnte das RSA-Verfahren tatsächlich genutzt werden: zwar ist im Allgemeinen nicht jede Zahl e, die aus einer öffentlich bekannten Information generiert werden kann, teilerfremd zu $(p-1)(q-1)$ und damit als Exponent nutzbar. Wenn allerdings die Primzahlen p und q die Form $p = 2p' + 1$ und $q = 2q' + 1$, wobei p' und q' wiederum Primzahlen sind, dann ist $(p-1)(q-1) = 4p'q'$ und alle ungeraden Zahlen, die keine Vielfachen von p' oder q' sind, d.h. fast alle, sind teilerfremd zu $(p-1)(q-1)$. Die Wahrscheinlichkeit, eine nicht als Exponent nutzbare Zahl e aus einer E-Mail-Adresse zu generieren, wäre damit vernachlässigbar gering.

Tatsächlich hat es fast 20 Jahre gedauert, bis 2001 Boneh und Franklin [BF01] und unabhängig Cocks [Coc01] zwei Verfahren zur Identitäts-basierten Verschlüsselung vorgestellt haben. Beide Verfahren sind mathematisch sehr komplex, sie basieren auf Weill-Pairings bzw. auf quadratischen Resten.

2.8 Erzeugung zufälliger Zahlen und Primzahlen

2.8.1 Erzeugung zufälliger Primzahlen

In vielen kryptografischen Anwendungen, zum Beispiel bei der Schlüsselerzeugung im RSA-Verfahren, benötigt man eine Primzahl in einem bestimmten Zahlenbereich $N_n = \{1, \ldots, n\}$. In der Regel soll eine solche Primzahl zufällig aus dem Bereich erzeugt werden, so dass jede Primzahl die gleiche Wahrscheinlichkeit hat, gezogen zu werden. Hierzu geht man wie folgt vor: Man erzeugt eine beliebige ungerade Zahl aus dem gewünschten Zahlenbereich zufällig mit Gleichverteilung (siehe Abschnitt 2.8.2), und testet dann, ob sie eine Primzahl ist. Falls ja, gibt man sie aus. Falls nein, wiederholt man das Verfahren, und zwar so lange, bis man eine Primzahl erhält.

zufällige Primzahl

Zunächst stellt sich dabei die Frage, wie oft man im Mittel die Erzeugung wiederholen muss, bis man eine Primzahl erhält. Dies ist wichtig um festzustellen, ob das Verfahren überhaupt praktikabel ist. Hierauf gibt der *Primzahlsatz* eine Antwort. Ist $\pi(n)$ die Anzahl der Primzahlen zwischen 1 und n, dann gilt:

Primzahlsatz

$$\frac{\pi(n)}{n} \approx \frac{1}{\ln n} \,.$$

Die Wahrscheinlichkeit, beim Würfeln einer Zahl aus N_n eine Primzahl zu erwischen, ist also etwa $q = 1/\ln n$. Man muss im Mittel also etwa $\ln n$ Zahlen würfeln, um eine Primzahl zu erhalten. Bei $n = 2^{1024}$ — wenn man einen 2048 Bit langen RSA-Schlüssel will, müssen die Primzahlen jeweils 1024 Bit lang sein — sind das $\ln n = \ln 2^{1024} = 1024 \cdot \ln 2 \approx 707$ Versuche. Dies ist praktikabel, falls der Primzahltest nicht zu lange dauert.

Hat man eine Zahl als Kandidaten generiert, so muss man entscheiden, ob es sich um eine Primzahl handelt oder nicht. Hierzu kann man prinzipiell eine Primfaktorzerlegung vornehmen (siehe Kurs *(01867) Sicherheit im Internet 2*). Da diese aber für Zahlen der verwendeten Größenordnungen (z. B. 1024 Bit Primzahlen, um einen 2048 Bit Schlüssel bei RSA zu erhalten) zu aufwändig ist (sonst wäre RSA auch nicht sicher), behilft man sich mit probabilistischen *Primzahltests*.

probabilistischer Primzahltest

Ein solcher Test erhält als Eingabe eine Zahl x, und gibt als Ergebnis entweder „Primzahl“ oder „Keine Primzahl“ aus. Gibt er das Ergebnis „Keine Primzahl“ aus, dann ist die Zahl x wirklich keine Primzahl. Gibt er als Ergebnis „Primzahl“ aus, dann ist die Zahl x mit großer Wahrscheinlichkeit tatsächlich eine Primzahl. Es gibt aber eine kleine Wahrscheinlichkeit f, dass sie trotzdem keine Primzahl ist. Diese kleine Wahrscheinlichkeit nennt man *Fehlerwahrscheinlichkeit*. Um diese Fehlerwahrscheinlichkeit klein zu machen, führt man mehrere verschiedene Primzahltests aus. Haben s dieser Tests alle die Fehlerwahrscheinlichkeit f, und nimmt man an, dass die Fehler in den verschiedenen Tests unabhängig voneinander auftreten, dann beträgt die Wahrscheinlichkeit, dass alle Tests gleichzeitig einen Fehler machen, f^s. Ist zum Beispiel $f = 10^{-4}$ und wir benutzen $s = 5$ Tests, dann ist die resultierende Fehlerwahrscheinlichkeit 10^{-20}. Zum Vergleich beträgt die Wahrscheinlichkeit, im Lottospiel „6 aus 49“ die sechs Gewinnzahlen samt Superzahl zu ziehen, etwa $7 \cdot 10^{-8}$, was bereits als ziemlich selten angesehen wird, allerdings etwa 7 Billionen mal wahrscheinlicher ist als die obige Fehlerwahrscheinlichkeit aller Tests zusammen.

Probabilistische Primzahltests basieren in der Regel auf zahlentheoretischen Zusammenhängen der folgenden Form:

Wenn x eine Primzahl ist, dann gilt die Gleichung $g(x) = 0$.

Hierbei ist die Funktion g spezifisch für den jeweiligen Test. Damit ist zunächst klar, dass, wenn die Gleichung nicht gilt, x auch keine Primzahl sein kann. Da allerdings nur eine Folgerung (Wenn ... dann ...) und keine Äquivalenz (x ist Primzahl genau dann wenn $g(x) = 0$ gilt) besteht, kann es Fälle geben, in denen x keine Primzahl ist, aber die Gleichung trotzdem gilt. Hieraus resultiert der Fehler. Damit der Test sinnvoll ist, sollten diese Fälle selten, d. h. die Fehlerwahrscheinlichkeit klein sein.

Miller-Rabin-Test

Einer der bekanntesten probabilistischen Primzahltests ist der *Miller-Rabin-Test*. Bei diesem würfelt man zunächst eine zufällige Zahl a zwischen 2 und $x - 1$, die sogenannte *Basis*. Dann bestimmt man die eindeutige Zerlegung der geraden Zahl $x - 1$ (die Zahl x war ja ungerade) als $x - 1 = d \cdot 2^j$, wobei d ungerade ist. Wenn man $x-1$ als Binärzahl darstellt, ist j gerade die Anzahl der Nullen am Ende der Darstellung, und d die Zahl, die durch die vorangehenden Stellen dargestellt wird. Wenn x nun eine Primzahl ist, dann gilt entweder

$$a^d \equiv 1 \bmod x$$

oder

$$a^{d \cdot 2^r} \equiv -1 \bmod x$$

für ein r zwischen 0 und $j - 1$.

Als Beispiel verwenden wir die Primzahl $x = 7$ und die Zahl $a = 2$. Es gilt $x - 1 = 6 = 3 \cdot 2^1$, also ist $d = 3$ und $j = 1$. Es gilt $a^d = 2^3 \equiv 1 \bmod 7$ und der Miller-Rabin-Test gibt „Primzahl“ aus.

Die einzigen Zahlen, die keine Primzahlen sind und für die der Test trotzdem erfolgreich ist, sind die sogenannten *starken Pseudoprimzahlen* zur Basis a. Diese sind hinreichend selten, so dass der Test praktikabel ist. Genauer gilt, dass die Fehlerwahrscheinlichkeit $f \leq 0,25$ beträgt. Da der Test den Vorteil hat, das man ihn mit verschiedenen a wiederholen kann, lässt sich durch k-fache Anwendung schon dieses Tests die Fehlerwahrscheinlichkeit auf $f \leq 2^{-2k}$ reduzieren[6].

starken Pseudoprimzahlen

Als Beispiel für eine starke Pseudoprimzahl sollen uns die Zahl $x = 91 = 7 \cdot 13$ und die Basis $a = 9$ dienen. Es gilt $x - 1 = 90 = 45 \cdot 2^1$, also ist $d = 45$ und $j = 1$. Außerdem gilt $a^d = 9^{45} \equiv 1 \bmod 91$.

Übungsaufgabe 2.7 *Bestimmen Sie drei weitere Basen a, bezüglich derer $x = 91$ eine Pseudoprimzahl ist.*

Die probabilistischen Primzahltests sind ein Beispiel für *Monte-Carlo-Verfahren*. Dies sind Algorithmen, die Zufallszahlen benutzen und mit einer gewissen Wahrscheinlichkeit einen Fehler machen, den man aber durch Wiederholung eingrenzen kann. Im vorliegenden Beispiel ist der Fehler einseitig, denn die Ausgabe „Keine Primzahl“ stimmt immer. Von den Monte-Carlo-Verfahren abzugrenzen sind die Las-Vegas-Verfahren. Diese benutzen keine Zufallszahlen, sondern ihr Ressourcenverbrauch (Laufzeit, Speicherplatz) schwankt abhängig von der Eingabe, wie zum Beispiel beim Quicksort-Algorithmus, dessen Laufzeit zwar für viele Eingabesequenzen aus n Zahlen $O(n \log n)$ beträgt, dessen Laufzeit im schlechtesten Fall aber $O(n^2)$ sein kann.

Man hat lange geglaubt, dass sich die Primzahleigenschaft in polynomieller Zeit nur probabilistisch testen lässt. Für die ersten effizienten deterministischen Verfahren konnte man die polynomielle Laufzeit nur unter der Bedingung beweisen, dass gewisse Hypothesen der Mathematik (wie die verallgemeinerte Riemann-Hypothese) wahr sind. Im Jahr 2002 wurde der AKS-Test [AKS02] publiziert, der deterministisch, effizient und unabhängig von Hypothesen ist.

Übungsaufgabe 2.8 *Warum erzeugt man nur ungerade Zufallszahlen, wenn man Primzahlen erzeugen will?*

Wie erzeugt man ungerade Zufallszahlen mit Gleichverteilung in N_n, wenn man einen Zufallszahlengenerator für beliebige Zahlen mit Gleichverteilung in N_n hat?

Übungsaufgabe 2.9 *Gegeben seien Primzahltests mit einer Fehlerwahrscheinlichkeit von 10^{-2}. Wieviele solcher Tests muss man ausführen, um die Fehlerwahrscheinlichkeit auf 10^{-8} zu reduzieren?*

2.8.2 Erzeugung von Zufallszahlen

Viele kryptografische Protokolle benötigen zur Durchführung zufällige Zahlen. Als Beispiel haben wir im vorigen Abschnitt die Erzeugung von Primzahlen

[6] Die Berechnung der Fehlerwahrscheinlichkeit im Miller-Rabin-Test und die Berechnung der mittleren Anzahl von Durchläufen bei der Erzeugung von Primzahlen durch Würfeln von Zahlen und Ausführen des Miller-Rabin-Tests erfordert eigentlich eine gemeinsame Betrachtung, da beide Ereignisse aufeinander folgen und sich immer wieder abwechseln. Wir haben der Anschaulichkeit halber hierauf verzichtet und verweisen auf die Diskussion bei Note 4.47 von [MOV96].

besprochen, die zum Beispiel bei der Schlüsselgenerierung des RSA-Verfahrens eine Rolle spielen.

Bei der Erzeugung zufälliger Zahlen unterscheiden wir die Erzeugung *echter* Zufallszahlen und die Erzeugung von *Pseudo*-Zufallszahlen. Um echte Zufallszahlen zu erzeugen, braucht man eine physikalische Quelle von Zufall. Zum Beispiel verwendet man hier Detektoren für Teilchen, die in einer gewissen Dichte aus dem Weltall auf die Erde „regnen“. Die gemessene Häufigkeit oder daraus abgeleitete Werte dient dann als Ausgangspunkt für die erzeugte Zufallszahl. Wir werden uns mit solchen Generatoren nicht weiter befassen und verweisen auf [MOV96, Kap. 5.2, 5.6] und [Gut98].

Wenn wir von der Erzeugung von Pseudozufallszahlen (PZZ) sprechen, meinen wir in der Regel die Erzeugung von Zahlen aus einem festen Zahlenbereich, zum Beispiel dem der `unsigned int` mit 32 Bit Länge. Dabei erfolgt die Erzeugung ausgehend von einem vorgegebenen Startwert (engl. **seed**) mit einem deterministischen Algorithmus, der eventuell ebenfalls vom Startwert abhängt. Teilweise werden zur Erzeugung des Seeds echte Zufallszahlen benutzt, man spricht dann von einem *hybriden* Generator.

Die determistisch erzeugten Zahlen sollen von einer zufälligen Folge nicht unterscheidbar sein. Zum Beispiel soll, wenn man eine sehr große Menge solcher erzeugten Zahlen betrachtet, die Häufigkeit jedes Zahlenwertes gleichgroß sein. Zum Test der statistischen Eigenschaften von Pseudo-Zufallszahlen-Generatoren (PZZG) gibt es daher eine ganze Reihe von sogenannten Testsuiten, also Programmen, die statistische Eigenschaften Überprüfen.

Die bekannteste dieser Testsuites ist *Diehard* von G. Marsaglia [Mar]. Daneben hat auch D. Knuth eine Reihe von Tests beschrieben [Knu97] und es gibt die Suite *Test01* von P. L'Ecuyer [LS07]. Schließlich hat das US-amerikanische Standardisierungsinstitut NIST eine Anleitung zum Test von PZZG für *kryptografische* Anwendungen sowie eine Testsuite erstellt [Ruk01].

Ein PZZG in kryptografischen Anwendungen soll nämlich zusätzlich zu den statistischen Eigenschaften seiner Ausgaben noch weitere Eigenschaften aufweisen. So soll er nicht vorhersagbar sein, d. h. man soll aus der Beobachtung einer Reihe von erzeugten Zahlen nicht die nächsten Zahlen vorhersagen oder den inneren Zustand des PZZG ableiten können. Andernfalls wären die nächsten Zufallszahlen, die in einem kryptografischen Protokoll benötigt werden, für einen Angreifer nicht mehr unbekannt und zufällig, sondern bekannt und deterministisch. Weiterhin sollen aus einem (teilweise) kompromittierten Zustand nicht die Vorgängerzustände berechenbar sein.

Ein PZZG ist eigentlich ein endlicher Automat mit Zustand, Ausgabefunktion und Zustandsübergangsfunktion, der zu bestimmten Zeiten durch den Seed in einen bestimmten Zustand versetzt wird und wobei durch den Seed eventuell die Zustandsübergangsfunktion verändert wird. Außerhalb dieser Zeiten erfährt er keine weitere Eingabe. Bei jedem Aufruf, eine PZZ zu erzeugen, ruft der endliche Automat seine Ausgabefunktion auf, die die PZZ erzeugt, und er ruft seine Zustandsübergangsfunktion auf, die den inneren Zustand aktualisiert.

Damit ergeben sich für einen kryptografischen PZZG die Anforderungen, dass die Anzahl der Bits der Ausgabe deutlich geringer sein muss als die Anzahl der Bits des Zustands. Ansonsten könnte aus der Ausgabe auf den Zustand rückgeschlossen werden. Weiterhin muss die Anzahl der Zustände hinreichend groß sein, wobei „hinreichend“ von den Berechnungsmöglichkeiten eines Angreifers abhängt (heute hält man 200 Bit lange Zustände für hinreichend). Denn

bei zu wenigen Zuständen könnte ein Angreifer sich vorab für *jeden* Zustand i die nächsten 10 Ausgaben o_i und den dann erreichten Zustand n_i als Tripel (i, o_i, n_i) speichern. Die Liste dieser Tripel sortiert er nach o_i. Beobachtet der Angreifer nun eine Folge o von 10 Ausgaben, dann sucht er in der Liste alle Tripel bei denen $o_i = o$ ist (geht mit Binärsuche, die Tripel stehen wegen der Sortierung direkt hintereinander). Nun hat er über die n_i der vorigen Tripel die Menge M aller möglichen Zustände, in denen sich der PZZG nun befinden kann. Beobachtet er eine weitere Ausgabe, dann kann er alle Zustände aus M eliminieren, die diese Ausgabe nicht erzeugen. Dies macht er solange, bis die Menge nur noch ein Element umfasst.

Außerdem soll bei jedem PZZG die *Periode*, d. h. die Anzahl von Zuständen, die durchlaufen wird, bevor ein Zustand erneut erreicht wird, möglichst groß sein, um die statistischen Eigenschaften zu verbessern. Bei n-Bit Darstellung hat man 2^n Zustände, so lange könnte die Periode maximal sein. Allerdings führen lange Perioden auch zu Vorhersagbarkeit. Bei komplexen Übergangsfunktionen verkürzt sich die Periode meist auf $\Theta(\sqrt{2^n})$. Das heißt, dass man die Anzahl der Bits für den Zustand doppelt so groß wie die Anzahl der Bits für die gewünschte Periode wählen muss.

Zufallszahlengeneratoren für kryptografische Aufgaben werden normalerweise in drei Klassen unterteilt. Entweder (1) sie basieren auf einem kryptografischen Primitiv, oder (2) auf einem zahlentheoretischen Problem, oder (3) stellen einen speziellen Entwurf dar.

1. Zur ersten Klasse zählen Stromchiffren wie A5/1, da diese nach Definition eine pseudozufällige Bitfolge erzeugen, und desweiteren Zähler, deren Ausgabefunktion entweder durch einen symmetrischen Blockverschlüsselungs-Algorithmus oder eine kryptografische Hashfunktion gebildet wird. Die dahinterliegende Annahme ist, dass eine Verschlüsselung oder die Anwendung einer Hashfunktion aus einem Klartext einen zufällig aussehenden Chiffretext bzw. Hashwert erzeugen. Als Klartexte, d. h. Zustände des Generators, nimmt man x, $x + 1$, $x + 2$,..., wobei x der Startzustand ist. In beiden Fällen werden der Startzustand und ggf. der benutzte Schlüssel aus dem Seed gebildet und müssen geheim gehalten werden.

2. Zur zweiten Klasse zählen Verfahren, deren Sicherheit wie bei der Public-Key-Verschlüsselung auf der Schwierigkeit der Primfaktorzerlegung oder des diskreten Logarithmus basieren. Beispiele sind die Generatoren nach Blum-Blum-Shub und Blum-Micali.

3. Zur dritten Klasse zählt zum Beispiel die `/dev/urandom`-Funktion in UNIX und Linux-Betriebssystemen, und der ANSI X9.17 Standard [MOV96; Gut98]. Dieser wird wohl, obwohl es Nachfolger gibt, immer noch benutzt, obwohl beide genannte Quellen Kritik bezüglich seiner Sicherheit vortragen. Übrigens benutzt der X9.17-Algorithmus den Triple-DES, allerdings nicht in einer vorgegebenen Weise wie in der ersten Klasse von Verfahren beschrieben, sondern in einer eher ad-hoc zu nennenden Weise.

Die US-amerikanische Standardisierungsbehörde NIST hat eine gut sortierte Webseite zum Thema kryptografisch sichere PZZG:

`http://csrc.nist.gov/groups/ST/toolkit/random_number.html`

2.9 Zusammenfassung

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie Folgendes gelernt haben:

- Welche Klassifikations- und Betriebsmöglichkeiten existieren für Verschlüsselungsverfahren?
- Welche weiteren Voraussetzungen müssen erfüllt sein, damit Verschlüsselung nicht hinfällig wird?
- Die Prinzipien und Eigenschaften der Secret-Key-Verschlüsselung sowie einige der aktuellen Verfahren.
- Die Prinzipien und Eigenschaften der Public-Key-Verschlüsselung sowie einige der aktuellen Verfahren.
- Die Prinzipien und Eigenschaften von Hashfunktionen und digitalen Signaturen.
- Welche administrativen Maßnahmen (Zertifikate, Schlüsselmanagement etc.) sind für den sicheren Einsatz von Public-Key-Verfahren erforderlich?
- Wie kann man sich mit dem neuen Personalausweis im Internet authentisieren oder digitale Signaturen mit dem Personalausweis erstellen?
- Wie kann man Zufallszahlen erzeugen und wie findet man zufällig eine große Primzahl?

Lösungen der Übungsaufgaben

Übungsaufgabe 2.1 Der Klartext zu dieser Nachricht lautet:

```
woistbehle
```

Übungsaufgabe 2.2 Der häufigste Buchstabe im Geheimtext ist das ‚o', dieser entspricht vermutlich dem Buchstaben ‚e' im Klartext. Vermutlich liegt also eine Cäsar-Verschlüsselung mit Schlüssel 10 vor. Die Überprüfung dieser Vermutung liefert folgenden Klartext:

```
entschluesselnsiefolgendengeheimtext
```

Übungsaufgabe 2.3 Die Nachricht könnte eine Aufforderung zum Handeln sein, beispielsweise Geld zu überweisen. Die Gefahr der Replay-Angriffe liegt darin, dass diese Handlungen automatisch veranlasst werden können, sobald die Nachricht eintrifft. Später lässt sich die Handlung evtl. nur mit großem Aufwand wieder rückgängig machen.

Übungsaufgabe 2.4 Würde man den Counter im ECB-Modus mit K verschlüsseln und vorab an den Empfänger senden, dann könnte ein Angreifer den ersten Geheimtextblock G_1 entschlüsseln. Der Angreifer verknüpft den ersten Geheimtextblock und den verschlüsselten Counter dazu einfach mit XOR.

Übungsaufgabe 2.5 Diese Hashfunktion ist nicht sicher, denn man kann sehr einfach zwei Nachrichten erzeugen, die denselben Hashwert haben. Dazu nutzt man aus, dass die Addition (auch wenn sie modulo 2^{32} ausgeführt wird) kommutativ ist. Man kann also einfach verschiedene Bytes einer Nachricht vertauschen und erhält wieder denselben Hashwert.

Übungsaufgabe 2.6

a) Bei 1 000 000 verschiedenen Hashwerten ist die Wahrscheinlichkeit, dass eine weitere Nachricht N' einen anderen Hashwert hat, 0.999999. Gesucht ist nun wieder das n mit $0.999999^n < 0.5$. Das ist für $n = 693\,147$ der Fall. Man braucht also 693 147 neue Nachrichten, damit mit Wahrscheinlichkeit größer 1/2 eine davon denselben Hashwert wie die Nachricht N hat.

b) Hier ist nun das m gesucht, für das gilt

$$1 - \frac{1\,000\,000 \cdot 999\,999 \cdot \ldots \cdot (1\,000\,000 - m + 1)}{1\,000\,000^m} > 0.5.$$

Das ist für $m = 1178$ der Fall. Man kann das m beispielsweise durch ein selbstgeschriebenes Programm in Java oder C herausfinden. Unter so vielen zufällig erzeugten Nachrichten sind mit einer Wahrscheinlichkeit größer als 1/2 zwei Nachrichten mit demselben Hashwert.

Übungsaufgabe 2.7 Man wähle zum Beispiel $a = 10, 12, 16$.

Übungsaufgabe 2.8 Alle Primzahlen mit Ausnahme der 2 sind ungerade Zahlen.

Eine einfache Methode besteht darin, eine gewürfelte Zahl zu verwerfen, wenn sie gerade ist, und so lange Zahlen zu erzeugen, bis eine ungerade Zahl entsteht. Die Trefferwahrscheinlichkeit bei diesem Bernoulli-Experiment ist

50%. Man kann natürlich eine gewürfelte gerade Zahl um 1 erhöhen. Alternativ reicht es auch, einen Zufallszahlengenerator des Bereichs $N_{n/2}$ zu benutzen. Liefert dieser eine Zahl x, dann gibt man $2x + 1$ aus. Damit erhält man alle ungeraden Zahlen im Bereich N_n mit Gleichverteilung.

Übungsaufgabe 2.9 Man muss vier solcher Tests ausführen, da $10^{-8} = 10^{-2 \cdot 4} = (10^{-2})^4$.

Literatur

[AKS02] Manindra Agrawal, Neeraj Kayal und Nitin Saxena. „PRIMES is in P“. In: *Ann. of Math* 2 (2002), S. 781–793.

[And08] Ross J. Anderson. *Security Engineering*. 2. Aufl. Wiley und Sons, 2008.

[Ano03] Anonymous. *Hacker's Guide*. Übersetzung von Maximum Security, 4th ed. München, Germany: Markt+Technik Verlag, 2003.

[Ano98] Anonymous. *Maximum Security*. 2. Aufl. Indianapolis, Indiana: SAMS, 1998.

[Bau97] Friedrich L. Bauer. *Entzifferte Geheimnisse. Methoden und Maximen der Kryptologie*. Heidelberg, Germany: Springer-Verlag, 1997.

[Ben+08] Jens Bender, Dennis Kügler, Marian Margraf und Ingo Naumann. „Sicherheitsmechanismen für kontaktlose Chips im deutschen elektronischen Personalausweis - Ein Überblick über Sicherheitsmerkmale, Risiken und Gegenmaßnahmen“. In: *Datenschutz und Datensicherheit* 32.3 (2008), S. 173–177.

[Ber+07] G. Bertoni, J. Daemen, M. Peeters und G. Van Assche. *Sponge functions*. Ecrypt Hash Workshop 2007. Mai 2007.

[Ber+11a] G. Bertoni, J. Daemen, M. Peeters und G. Van Assche. *The* Keccak *reference*. `http://keccak.noekeon.org/`. Jan. 2011.

[Ber+11b] G. Bertoni, J. Daemen, M. Peeters und G. Van Assche. *The* Keccak *SHA-3 submission*. `http://keccak.noekeon.org/`. Jan. 2011.

[BF01] Dan Boneh und Matthew K. Franklin. „Identity-Based Encryption from the Weil Pairing“. In: *CRYPTO*. Hrsg. von Joe Kilian. Bd. 2139. Lecture Notes in Computer Science. Springer, 2001, S. 213–229. ISBN: 3-540-42456-3.

[BPH02] L. Bassham, W. Polk und R. Housley. „Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation Lists (CRL) Profile“. `ftp://ftp.rfc-editor.org/in-notes/rfc3280.txt`. Apr. 2002.

[Bra99] John R. T. Brazier. „Possible NSA Decryption Capabilities“. In: *DuD Datenschutz und Datensicherheit* 23.10 (Okt. 1999), S. 576–581.

[BSB05] Daniel J. Barrett, Richard Silverman und Robert G. Byrnes. *SSH The Secure Shell — The Definitive Guide*. 2. Aufl. O'Reilly, 2005.

[BSI08a] BSI. *IT-Grundschutz-Vorgehensweise*. BSI-Standard 100-2, Version 2.0. Mai 2008.

[BSI08b] BSI. *Managementsysteme für Informationssicherheit (ISMS)*. BSI-Standard 100-1, Version 1.5. Mai 2008.

[BSI08c] BSI. *Risikoanalyse auf der Basis von IT-Grundschutz*. BSI-Standard 100-3, Version 2.5. Mai 2008.

[BSI12a] BSI. *Advanced Security Mechanisms for Machine Readable Travel Documents – Part 1 - eMRTDs with BAC/PACEv2 and EACv1*. Techn. Ber. TR-03110-1. Bundesamt für Sicherheit in der Informationstechnik, 2012.

[BSI12b] BSI. *Advanced Security Mechanisms for Machine Readable Travel Documents – Part 2 - Extended Access Conrol Version 2 (EACv2), Password Authenticated Connection Establishment (PACE) and Restricted Indetification (RI)*. Techn. Ber. TR-03110-2. Bundesamt für Sicherheit in der Informationstechnik, 2012.

[BSI12c] BSI. *Advanced Security Mechanisms for Machine Readable Travel Documents – Part 3 - Common Specifications*. Techn. Ber. TR-03110-3. Bundesamt für Sicherheit in der Informationstechnik, 2012.

[Coc01] Clifford Cocks. „An Identity Based Encryption Scheme Based on Quadratic Residues“. In: *IMA Int. Conf.* Hrsg. von Bahram Honary. Bd. 2260. Lecture Notes in Computer Science. Springer, 2001, S. 360–363. ISBN: 3-540-43026-1.

[CZ02] D. Brent Chapman und Elizabeth D. Zwicky. *Einrichten von Internet Firewalls*. Sebastopol, CA: O'Reilly, 2002.

[Dem00] W. Edwards Deming. *Out of the Crisis*. 2. Aufl. MIT Press, Oktober 2000.

[DH76] W. Diffie und M. Hellman. „New directions in cryptography“. In: *IEEE Transactions on Information Theory* 22.6 (Sep. 1976), S. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: http://dx.doi.org/10.1109/TIT.1976.1055638.

[DR02] Joan Daemen und Vincent Rijmen. *The Design of Rijndael*. Berlin Heidelberg: Springer Verlag, 2002.

[DR99] Joan Daemen und Vincent Rijmen. *AES Proposal: Rijndael*. http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip. 1999.

[Eck13] Claudia Eckert. *IT-Sicherheit*. 8. Aufl. München: Oldenbourg Wissenschaftsverlag GmbH, 2013.

[El 85] Taher El Gamal. „A public key cryptosystem and a signature scheme based on discrete logarithms“. In: *Proceedings of CRYPTO 84 on Advances in Cryptology*. Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1985, S. 10–18. ISBN: 0-387-15658-5. URL: http://dl.acm.org/citation.cfm?id=19478.19480.

[Ele98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. Sebastopol, CA: O'Reilly & Associates Inc., 1998.

[Ert03] Wolfgang Ertel. *Angewandte Kryptographie*. München: Fachbuchverlag Leipzig im Carl Hanser Verlag, 2003.

[Ger+04] Helmar Gerloni, Barbara Oberhaitzinger, Helmut Reiser und Jürgen Plate. *Praxisbuch Sicherheit für Linux-Server und -Netze*. München: Hanser Verlag, 2004.

[Ges04] Alexander Geschonneck. *Computer Forensik*. Heidelberg: dpunkt.verlag, 2004.

[Gon99] Marcus Goncalves. *Firewalls: A Complete Guide*. McGraw-Hill, Okt. 1999.

[Gut98] Peter Gutmann. „Software Generation of Practically Strong Random Numbers“. In: *Proc. Usenix Security Symposium*. Eine wesentlich erweiterte Version des Artikels ist verfügbar unter `http://www.cypherpunks.to/~peter/06_random.pdf`. 1998.

[Hel80] Martin E. Hellman. „A cryptanalytic time-memory trade-off“. In: *IEEE Transactions on Information Theory* 26.4 (1980), S. 401–406.

[Hou+02] R. Housley, W. Polk, W. Ford und D. Solo. „Internet X.509 Public Key Infrastructure“. `ftp://ftp.rfc-editor.org/in-notes/rfc3280.txt`. Apr. 2002.

[KN07] Dennis Kügler und Ingo Naumann. „Sicherheitsmechanismen für kontaktlose Chips im deutschen Reisepass - Ein Überblick über Sicherheitsmerkmale, Risiken und Gegenmaßnahmen“. In: *Datenschutz und Datensicherheit* 31.3 (2007), S. 176–180.

[Knu97] Donald E. Knuth. *The Art of Computer Programming Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

[LS07] P. L'Ecuyer und R. Simard. „TestU01: A C Library for Empirical Testing of Random Number Generators“. In: *ACM Transactions on Mathematical Software* 33.4, Article 22 (Aug. 2007). `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`.

[Mar] G. Marsaglia. *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*. `http://www.stat.fsu.edu/pub/diehard/`.

[MOV96] A. Menezes, P. van Oorschot und S. Vanstone. *Handbook of Applied Cryptography*. `http://www.cacr.math.uwaterloo.ca/hac`. CRC Press, 1996.

[MR99] Günter Müller und Kai Rannenberg, Hrsg. *Multilateral Security in Communications*. München: Addison Wesley Longman GmbH, 1999.

[Oec03] Philippe Oechslin. „Making a Faster Cryptanalytic Time-Memory Trade-Off“. In: *CRYPTO*. Hrsg. von Dan Boneh. Bd. 2729. Lecture Notes in Computer Science. Springer, 2003, S. 617–630. ISBN: 3-540-40674-3.

[Ris05] Ivan Ristic. *Apache Security — The Complete Guide to Securing Your Apache Web Server*. O'Reilly, 2005.

[Ris10] Ivan Ristic. *ModSecurity Handbook: The Complete Guide to the Popular Open Source Web Application Firewall*. FeistyDuck, 2010.

[Roß99] Stephan Roßbach. *Der Apache Webserver*. Bonn, Germany: Addison-Wesley, 1999.

[Ruk01] A. Rukhin et al. *NIST Special Publication 800-22: A Statistical Test Suite for Random And Pseudorandom Number Generators for Cryptographic Applications*. `http://csrc.nist.gov/rng/SP800-22b.pdf`. Mai 2001.

[Sch+99] Bruce Schneier, John Kelsey, David Wagner, Chris Hall, Niels Ferguson und Doug Whiting. *Twofish Encryption Algorithm : A 128-Bit Block Cipher*. John Wiley and Sons, 1999.

[Sch00] Bruce Schneier. *Secrets & Lies. IT-Sicherheit in einer vernetzten Welt*. Heidelberg, Weinheim, Germany: dpunkt.Verlag / Wiley-VCH, 2000.

[Sch96] Bruce Schneier. *Angewandte Kryptographie. Protokolle, Algorithmen und Sourcecode in C*. Bonn, Germany: Addison-Wesley, 1996.

[SGG08] Abraham Silberschatz, Peter Galvin und Greg Gagne. *Applied Operating System Concepts*. 8. Aufl. New York, NY, USA: John Wiley, Inc., 2008.

[Sha84] Adi Shamir. „Identity-Based Cryptosystems and Signature Schemes“. In: *CRYPTO*. Hrsg. von G. R. Blakley und David Chaum. Bd. 196. Lecture Notes in Computer Science. Springer, 1984, S. 47–53. ISBN: 3-540-15658-5.

[Sot+08] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik und Benne de Weger. *MD5 considered harmful today*. `http://www.win.tue.nl/hashclash/rogue-ca/`. Dez. 2008.

[Spe11] Ralf Spenneberg. *Linux Firewalls – Sicherheit für Linux-Server und -Netzwerke mit IPv4 und IPv6*. 2. Aufl. München, Deutschland: Addison-Wesley, 2011.

[Sta00] William Stallings. *Network Security Essentials*. Uppser Saddle River, New Jersey: Prentice Hall, 2000.

[Sta06] William Stallings. *Cryptography and Network Security*. 4. Aufl. Upper Saddle River, New Jersey: Prentice Hall, 2006.

[Tan02] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 2002.

[Vie09] John Viega. *the myths of security*. Sebastopol, CA: O'Reilly, 2009.

[Wol07] Sebastian Wolfgarten. *Apache Webserver 2. Installation, Konfiguration, Programmierung*. Addison-Wesley, 2007.

[WWS02] Tobias Weltner, Kai Wilke und Björn Schneider. *Windows-Sicherheit, Das Praxisbuch*. Konrad-Zuse-Str. 1, D-85716 Unterschleißheim: Microsoft Press, 2002.

[WY05] Xiaoyun Wang und Hongbo Yu. „How to Break MD5 and Other Hash Functions“. In: *Advances in Cryptology - EUROCRYPT 2005*. Hrsg. von Ronald Cramer. Lecture Notes in Computer Science 3494. Berlin: Springer-Verlag, 2005, S. 19–35.

002456656
(04/17)

01866-5-02-S 1