Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung

Seminar
Internet-Dienste
Sommersemester 2004

Kryptographie

Jens Kühnle
02. Juli 2004

Betreuung Prof. Dr. Franz Schweiggert

Inhaltsverzeichnis

1 Einleitung **1**

2 Verschlüsselungsverfahren – Grundlagen **2**

3 Symmetrische Kryptosysteme **3**
3.1 Die Transformationsmethode 3
3.2 Die Substitutionsmethode 3
3.3 Weitere Unterscheidungsmerkmale 4

4 Die Kryptoanalyse **4**

5 Perfekte Sicherheit **5**

6 Weitere symmetrische Algorithmen **5**
6.1 Das One-Time-Pad 5
6.2 Data Encryption Standard – DES 7
6.3 International Data Encrytion Algorithm – IDEA 7
6.4 Vor- und Nachteile symmetrischer Kryptosysteme 7

7 Asymmetrische Kryptosysteme **8**
7.1 Mathematische Basics 9
7.1.1 Komplexitätstheorie 9
7.1.2 Zahlentheorie 9
7.2 Schlüsselaustausch nach Diffie und Hellman 13
7.3 Der RSA-Algorithmus 13
7.3.1 Schlüsselerzeugung 14
7.3.2 Verschlüsselung 14
7.3.3 Entschlüsselung 14
7.3.4 Ein kleines Zahlenbeispiel 15
7.3.5 Zur Sicherheit von RSA 15
7.3.6 RSA als Signaturverfahren 15
7.4 Schlüsselmanagement durch Trusted Third Parties 16
7.5 Vor- und Nachteile asymmetrischer Kryptosysteme 17
7.6 Fazit 18

8 Hybride Kryptosysteme **18**

9 Anwendung: Pretty Good Privacy (PGP) 19
9.1 Funktionalitäten von PGP 19
9.2 Schlüsselverwaltung bei PGP 20
9.3 Zur Erzeugung von random session keys 22
9.4 Zur Sicherheit von PGP 22
9.5 Einige PGP-Befehle 23
9.5.1 Schlüsselerzeugung 23
9.5.2 Schlüsselverwaltung 24
9.5.3 Erste Schritte mit PGP 25
9.5.4 Weiterführendes zu PGP 26
9.6 Ein kleines Beispiel 27
9.7 Abschließende Bemerkungen zu PGP 31

10 Quellen 34

Abbildungsverzeichnis

1 Verschlüsselungsschema des One-Time-Pad 6
2 Standardsituation der Public-Key Kryptographie 8
3 Diffie-Hellman Schlüsselaustausch 13
4 Prinzip der Trusted Third Parties (TTP) 17
5 Prinzip hybrider Verfahren 18
6 Unser kleines PGP-Szenario 27

1 Einleitung

„Manche Menschen benützen
ihre Intelligenz zum Vereinfachen,
manche zum Komplizieren.“
(Erich Kästner)

Der Begriff „**Kryptologie**“ stammt aus dem Griechischen und setzt sich aus „kryptós“ (geheim, verborgen, versteckt) und „lógos“ (Wissen) zusammen. Die Kryptologie als *Teilgebiet der Mathematik* bezeichnet die Wissenschaft der *Methoden und Verfahren zur Verheimlichung von Informationen*. Sie umfasst die „**Kryptographie**“ (Wissenschaft der Entwicklung von Kryptosystemen) und die „**Kryptoanalyse**“ (Kunst, diese zu kompromittieren) – zwei Disziplinen im ständigen Wettstreit.

Die Geschichte der Kryptologie „*ist die Geschichte des jahrhundertealten Kampfes zwischen Verschlüsslern und Entschlüsslern, eines geistigen Rüstungswettlaufs, der dramatische Auswirkungen auf den Gang der Geschichte hat*“ [Singh 1999, S. 9][1]. Erste Spuren der Kryptologie finden sich bereits ca. 2000 v. Chr. in Ägypten, aber auch bei anderen frühen Kulturen. Historisch überwiegend militärisch motiviert, hat die Kryptologie insbesonders auch zur technischen Entwicklung des modernen Computers beigetragen.
Durch die weite Verbreitung und Vernetzung von Computern rückt in Zeiten mobiler Kommunikation, E-M@il, digitaler Signatur, Homebanking, elektronischem Geld, Chipkarten usw. die Kryptologie immer mehr in das öffentliche Interesse. Der geschichtlich interessierte Leser sei auf das Buch von [Singh, 1999] verwiesen.
Über den historischen Hauch des Abenteuers und der Romantik hinaus ist Kryptologie heute vor allem eine *mathematische Disziplin*. Die Mathematik liefert dabei *theoretische Rechtfertigung für die Stärken eines Verfahrens* – Beweise garantieren unter Umständen ein bestimmtes Sicherheitsniveau unabhängig von der gegenwärtigen Technologie. Auch wenn solche Beweise nur selten geführt werden konnten, scheint die Mathematik trotzdem *vertrauenswürdig* beim Test von Kryptosystemen.

Im Folgenden werden wir uns auf die **Kryptographie** konzentrieren.

---

[1] Erinnert sei beispielsweise an die *Enigma*-Verschlüsselung (griechisch „Rätsel“) der Deutschen ab ca. 1926. Lange Zeit galt diese als die sicherste der Welt, obgleich bereits Anfang der 30er Jahre der Pole *Marian Rejewski* die Enigma brach. Nach einer Modifikation derselben kurz vor Kriegsbeginn 1939 stellte Polen seine gesamten Ergebnisse Frankreich und England zur Verfügung. Aber erst zu Beginn der 40er Jahre gelang den Briten im *Bletchley Park* der Durchbruch. Ganz entscheidend war jedoch, dass diese Tatsache bis Kriegsende 1945 geheim gehalten werden konnte. Einige Historiker meinen, dass die Kryptoanalyse im zweiten Weltkrieg einige Jahre Krieg erspart hat.
Interessant ist auch, dass die US-Regierung nach Kriegsende Enigma-Maschinen an Regierungen von Entwicklungsländern verkaufte, ohne diesen zu sagen, dass die Verschlüsselung bereits geknackt wurde. Warum wohl?

2 Verschlüsselungsverfahren – Grundlagen

Bevor wir die Wissenschaft eine Nachricht so zu transformieren, dass niemand – außer dem berechtigten Empfänger – sie lesen kann, kennenlernen können, benötigen wir einige grundlegende Begriffe:

- **Klartext**:
  Text, Nachricht, Buchstaben-/Zeichenfolge, die wir übermitteln wollen;
- **Geheimtext**, **Chiffre** oder **Chiffrat**:
  verschlüsselte Nachricht, die tatsächlich übermittelt wird;
- **Chiffrieren** bzw. **Dechiffrieren**:
  Verschlüsselungs- bzw. Entschlüsselungsvorgang;
- **Algorithmus**:
  Regel zur Erzeugung eines Chiffres (de facto öffentlich);
- **Schlüssel** oder **Key**:
  beschreibt die Verwendung des Algorithmus in einer konkreten Situation;
- **Kryptosystem**, **Chiffriersystem**:
  System, welches die Encryption und Decrytion von Nachrichten ermöglicht.

Da der Chiffrieralgorithmus im Allgemeinen öffentlich ist, *beruht die gesamte Sicherheit eines Kryptosystems auf der Geheimhaltung des Schlüssels*!

Mathematisch gesehen ist ein **Kryptosystem** ein Tupel (M, C, K, E, D) mit:

- $M = \{messages\}$ = *Plaintext Space*: Elemente sind mögliche Klartexte;
- $C = \{chiffres\}$ = *Cipher text Space*: Elemente sind mögliche Chiffrate;
- $K = \{keys\}$ = *Key Space*: Elemente sind mögliche Schlüssel;
- E ist eine Familie von Transformationen $E_e : M \to C$;
- D ist eine Familie von Transformationen $D_d : C \to M$;
- $\forall\, e \in K \quad \exists\, d \in K : \; D_d(E_e(m)) = m, \; \forall\, m \in M$.
  Für praktische Anwendungen fordert man meist: $E_e(D_d(c)) = c, \; \forall\, c \in C$.

Grundvoraussetzungen an ein „sinnvolles“ Kryptosystem sind:

- **Maxime von Kerkhoffs**: *„Die Sicherheit eines Kryptosystems darf nicht von der Geheimhaltung des Algorithmus abhängen. Die Sicherheit gründet sich nur auf die Geheimhaltung des Schlüssels“* [Beutelspacher [6]2002, S. 15]. Die Veröffentlichung eines Algorithmus wird in der Kryptographie als wichtigste Voraussetzung angesehen, um ihn als sicher einschätzen zu können. Denn nur eine Überprüfung durch die gesamte wissenschaftliche Gemeinschaft und durch ausgiebige Kryptoanalyse kann Schwächen eines Algorithmus hinreichend zuverlässig aufdecken.

- **Geheimhaltungsforderung**: Es darf nicht einfach möglich sein, dass ein potentieller Angreifer die Entschlüsselungstransformation D_d aus dem Chiffrat $c = E_e(m)$ selbst bei bekanntem Klartext m findet oder systematisch m aus c bestimmen kann.

- **Authentizitätsforderung**: Es darf nicht einfach möglich sein, dass ein Angreifer die Verschlüsselungstransformation E_e aus dem Chiffrat c selbst bei bekanntem Klartext m findet oder systematisch Chiffrate $c^\star \in C$ bestimmen kann, so dass $D_d(c^\star)$ ein gültiger Klartext aus M ist.

3 Symmetrische Kryptosysteme

In Abgrenzung zu den später diskutierten asymmetrischen Verschlüsselungsverfahren gehen wir in diesem Abschnitt auf symmetrische Systeme ein, die zur Ver- und Entschlüsselung *einen gemeinsamen Schlüssel* verwenden, der *vor Beginn* der eigentlichen Kommunikation *ausgetauscht* werden muss. Natürlich müssen die Kommunikationspartner – wie auch bei jeder Art des Informationsaustausch – einen Chiffrieralgorithmus vereinbart haben.

3.1 Die Transformationsmethode

Das Chiffrat entsteht durch *Permutation der einzelnen Zeichen oder ganzer Blöcke des Klartextes.*
Sei dazu $m = m_1 \dots m_N$ eine Zerlegung des Klartextes $m \in M$ in einzelne Blöcke $m_i = m_{i1} \dots m_{in}$ der Länge n, und π eine Permutation der Menge $\{1 \dots n\}$, σ eine Permutation von $\{1 \dots N\}$. Definiere $m_{\pi(i)} := m_{i\pi(1)} \dots m_{i\pi(n)}$. Das Chiffrat c ergibt sich als $c := m_{\pi(\sigma(1))} \dots m_{\pi(\sigma(N))}$. Schlüssel ist dann das Tupel (π, σ). Dechiffriert wird mit den zu π und σ Inversen π^{-1} und σ^{-1}.

3.2 Die Substitutionsmethode

Durch *Ersetzung* der Zeichen des Klartextalphabets (KTA) durch Zeichen eines (*monoalphabetisch*) oder mehrerer (*polyalphabetisch*) Geheimtextalphabete (GTA) wird die Nachricht in den Geheimtext überführt.
Die Cäsar-Verschiebung – benannt nach seinem Erfinder – ist wohl die älteste *monoalphabetische* Substitution. Das Chiffrat erhält man, indem jeder Klartextbuchstabe durch das in diesem Alphabet drei Positionen weiter liegende Zeichen ersetzt wird. Die 3 als Verschiebedistanz ist natürlich völlig willkürlich. Bei Verwendung des 26 Zeichen umfassenden Alphabets $\{a \dots z\}$ erhalten wir eine Verschiebungstabelle der folgenden Form:

KTA	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
GTA	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

Mathematisch bedeutet die Verschlüsselung des Klartextes m die Transformation $c = E_3(m) = (m + 3)\, mod\, 26$, wenn wir $a, b, \dots, z$ mit $\{0, 1, \dots, 25\}$

identifizieren. Die Entschlüsselung erfolgt durch $D_3(c) = (c - 3)\, mod\, 26$. Es sollte klar sein, dass dieses System nur wenig Sicherheit bietet – nach maximal 26 Versuchen ist es gebrochen.
Sicherer erscheint dagegen, als GTA eine beliebige Permutation der Zeichen des KTA zu verwenden und jedes Zeichen im Klartext entsprechend seiner Position im KTA zu ersetzen durch das Zeichen an eben dieser Position im GTA. Schlüssel ist dann das gesamte GTA, von dem es immerhin 26! verschiedene gibt. Durch eine *statistische Analyse* der Häufigkeiten von Buchstaben und Buchstabengruppen in der zugrundeliegenden natürlichen Sprache stellt der vermeintliche Zugewinn an Sicherheit durch die größere Anzahl an möglichen Schlüsseln aber kein Problem für die Kryptoanalyse dar[2].
Um eine solche Analyse zu erschweren, verwenden *polyalphabetische* Substitutionsmethoden mehrere voneinander unabhängige GTAs. Als Verallgemeinerung des Cäsar-Codes sei kurz auf die Vigenére-Verschlüsselung hingewiesen. Dabei verwendet man zum Chiffrieren reihum insgesamt N gegen das KTA verschobene GTAs. Schlüssel ist dann der geordnete N-Tupel der verwendeten GTAs. Aber auch hier kann bei relativ zu N langem Chiffre statistisch durch Analyse von Buchstabengruppen auf die Anzahl N der GTAs geschlossen, und danach das System analog zur monoalphabetischen Substitution gebrochen werden. Die Sicherheit des Verfahrens steigt also, je mehr GTAs benützt werden.

3.3 Weitere Unterscheidungsmerkmale

Der Vollständigkeit halber sei noch auf folgende Vorgehensweisen hingewiesen:

- **Blockchiffren** setzten ganze Blöcke des Klartextes in entsprechende Blöcke des Geheimtextes um;
- **Stromchiffren** arbeiten hingegen kontinuierlich zeichenweise;
- **Produktchiffren** entstehen durch mehrfache Verschlüsselung.

4 Die Kryptoanalyse

Kryptoanalyse ist die Lehre vom Brechen von Kryptosystemen. *Ohne Kenntnis des Schlüssels sollen Schwachstellen eines Verfahrens aufgespürt werden.* Wir wollen abhängig von den situationsbedingt zur Verfügung stehenden Informationen **mögliche Angriffsformen** grob klassifizieren:

- **Known ciphertext attack** – schwächster kryptoanalytischer Angriff: Dem Kryptoanalytiker ist *nur Geheimtext bekannt.* Darauf basierend versucht er Rückschlüsse auf den Klartext oder die Struktur des Schlüssel zu ziehen (beispielsweise mit Hilfe statistischer Verfahren).

[2] Grundidee: In der deutschen Sprache ist das „e“ der häufigste Buchstabe. Im Chiffrat wird „e“ bei monoalphabetischer Chiffrierung aber durch genau ein anderes Zeichen dargestellt. Der häufigste Geheimtextbuchstabe ist damit höchstwahrscheinlich das „e“. Usw.

- **Brute force attack** – Spezialfall der known ciphertext attack:
  Dieser Angriff beruht auf dem *Durchprobieren aller in Frage kommender Schlüssel*, bis ein sinnvoller Klartext entsteht. Zwar ist dies der denkbar schwächste Angriff, aber theoretisch gegen jedes kryptographische System anwendbar. Je höher die Anzahl der möglichen Schlüssel jedoch ist, desto länger dauert im Allgemeinen die Suche nach dem richtigen Schlüssel.

- **Known plaintext attack**:
  Der Kryptoanalytiker *kennt zusammengehörige Klar-/Geheimtextpassagen.* Dies ist realistischer, als es auf den ersten Blick scheint – standardisierte Formate und Kodierungen von Protokolleinheiten sind meist öffentlich bekannt und leicht von der eigentlichen Nachricht trennbar.

- **Chosen plaintext attack**:
  Hat der Kryptoanalytiker *Zugang zum Verschlüsselungsalgorithmus mit aktuellem Schlüssel*, so kann er *wiederholt selbstgewählten Klartext chiffrieren* und versuchen Rückschlüsse auf den Schlüssel zu ziehen.

Gute kryptographische Verfahren müssen *all diesen Angriffsarten möglichst viel Wiederstand entgegensetzen*!

5 Perfekte Sicherheit

„Intuitiv gesagt bedeutet ***perfekte Sicherheit****, dass der Kryptoanalytiker (...) keine Chance hat, seine Kenntsnisse über das System zu vergrößern, auch wenn ihm alles Wissen und alle Rechenkapazität der Welt zur Verfügung steht“* [Beutelspacher ^{6}2002, S. 47]. Ein sicheres System (in diesem Sinne) weist also die Eigenschaft perfekter Geheimhaltung auf.
Wir wollen darauf verzichten eine mathematisch exakte Definition des Begriffs der perfekten Sicherheit und entsprechende Kriterien zu geben. Viel wichtiger erscheint die Frage, ob es ein solch perfektes Kryptosystem überhaupt gibt?

6 Weitere symmetrische Algorithmen

6.1 Das One-Time-Pad

Das **One-Time-Pad** ist das einzige *absolut sichere Kryptoverfahren.* Es ist sogar gegen eine brute force attack vollständig immun.
Dieses System wurde 1917 von *Gilbert S. Vernam* entwickelt und patentiert. Aber erst 1949 bewies *Claude E. Shannon*, dass es perfekt sicher ist.
Verschlüsselt werden Bitfolgen[3] der Länge N. Jedes Bit-Zeichen im Klartext

[3] OBdA verwenden wir das One-Time-Pad zur Verschlüsselung von Bitfolgen. Dazu wird Klartext zunächst beispielsweise mit dem *American Standard Code for Information Interchange* (**ASCII**) in eine Bitfolge verwandelt. Das Verfahren könnte natürlich auch auf Buch-

wird dabei mit einem zufällig erzeugten Bit verknüpft (Stromchiffre). Das Ergebnis dieser Verknüpfung ist das Chiffrat, die N-stellige Folge der Zufall-Bits ist der Schlüssel.
Dazu sei $M = C = K = \{0,1\}^N$. Verwendung findet die *Addition modulo 2*:

$$\text{exclusive-OR} \oplus : \{0,1\} \to \{0,1\}, \text{ vermöge } \quad \begin{array}{c|cc} \oplus & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$$

Der Geheimtext wird nach folgendem Schema erzeugt:

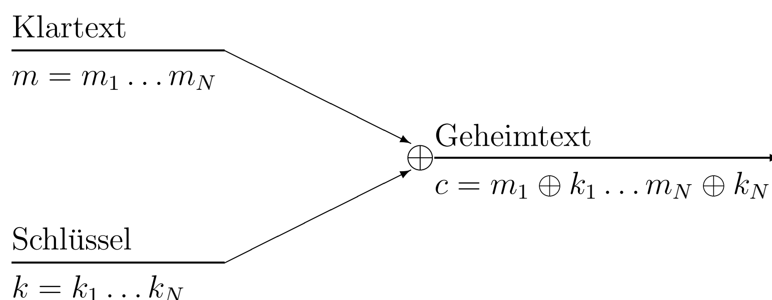

Abbildung 1: Verschlüsselungsschema des One-Time-Pad

Entschlüsselt wird durch wiederholtes Aufaddieren der Schlüsselbitfolge.
Für die Sicherheit dieses System ist es entscheidend, dass alle Schlüsselbitfolgen der Länge N gleich wahrscheinlich sind, d.h. die Schlüsselbits müssen zufällig generiert werden[4]. Damit ergibt sich, dass ein *Geheimtext* einer bestimmten Länge *jeden beliebigen Klartext* der gleichen Länge *repräsentieren kann* – und zwar *mit der selben Wahrscheinlichkeit*. Ohne Kenntnis des Schlüssels ist es *unmöglich*, aus dem Chiffrat auf den Klartext zu schließen, selbst mit beliebigem technischen Aufwand.
„The caveat, and this is a big one, is that the key (...) [has] to be generated randomly. Any attack against this scheme will be against the method used to generate the key. (...) The other important point is that you can never use the key sequence again, ever. (...) [Because] if a cryptoanalyst has multiple ciphertexts whose keys overlap, he can reconstruct the plaintext. (...) Even if you solve the key distribution and storage problem, you have to make sure the sender and receiver are perfectly synchronized. If the receiver is off by a bit (or if some bits are dropped during the transmission), the message won't make any sense“ [Schneier 1996, S. 16f].

stabenebene verwendet werden (Verknüpfung wäre dann die *Addition modulo 26*). Ein interessanter Aspekt der Verschlüsselung auf Bitebene ist, dass diese innerhalb eines Buchstabens erfolgen kann. In den weiteren Ausführungen werden wir stets auf Bits arbeiten.

[4]In der Praxis werden Pseudo-Zufallsgeneratoren (beispielsweise Schieberegister) zur Erzeugung der Schlüsselbitfolge verwendet.

6.2 Data Encryption Standard – DES

Der amerikanische **Data Encryption Standard** (**DES**) von 1977 beruht auf dem von IBM seit Beginn der 70er Jahre entwickelten Algorithmus *Lucifer*. Es handelt sich um eine symmetrische Blockchiffre mit einer Blocklänge von 64 Bit und einer Schlüssellänge von 56 Bit (die verbleibenden 8 Bit finden Verwendung zur *error detection*). DES wurde im Rahmen einer öffentlichen Ausschreibung ausgewählt und von dem mit der Überwachung internationaler Nachrichtenstrecken betrauten Geheimdienst **NSA** (**National Security Agency**) untersucht und teilweise verändert[5]. DES gilt als relativ sicher, obwohl eine brute force attack (insgesamt 2^{56} mögliche DES-Schlüssel) bei heutiger Technologie keine Probleme bereitet[6]. Man beachte jedoch, dass eine brute force attack auch die einzige öffentlich bekannt gewordene Möglichkeit ist, DES zu brechen – der Algorithmus an sich ist also sehr gut. Triple DES (3DES) ist nur eine Variante des klassischen DES, bei der der DES-Algorithmus dreifach mit verschiedenen Schlüsseln angewandt wird. Im Jahr 2000 wurde der symmetrische Blockchiffre **Advanced Encryption Standard** (**AES**, von Rijndael) zum Nachfolger des DES erklärt.

6.3 International Data Encrytion Algorithm – IDEA

Der **International Data Encrytion Algorithm** (**IDEA**) von *Xuejia Lai* und *James Massey* aus dem Jahre 1991 ist ein blockorientiertes symmetrisches Verschlüsselungsverfahren und gilt als einer der sichersten verfügbaren Algorithmen. Er ist nicht nur etwa doppelt so schnell wie DES, sondern auch sicherer. Öffentlich sind bislang keine erfolgreichen Angriffe gegen IDEA bekannt geworden und eine brute force attack bleibt bei der benützten Schlüssellänge von 128 Bit heute wirkungslos. Leider unterliegt IDEA dem Patentschutz und ist deshalb nur eingeschränkt einsetzbar. Da er bei **Pretty Good Privacy** (**PGP**) Verwendung findet, gilt er trotzdem als weit verbreitet.

6.4 Vor- und Nachteile symmetrischer Kryptosysteme

Vorteile:

- Hohe Geschwindigkeit der Ver- und Entschlüsselung;
- Grundlage: Wiederholte Anwendung einfacher Methoden.

Nachteile:

- Geheimer Schlüsseltausch (nicht-kryptographische Methoden) nötig;
- Spontane Kommunikation nicht realisierbar;
- Digitale Signatur nicht möglich (Authentizitätsproblem).

---

[5] Beispielsweise die Verkürzung der ursprünglichen Schlüssellänge von 128 Bit auf 56 Bit

[6] Im Jahr 1999 wurde ein DES-verschlüsselter Text durch eine brute force attack in nur 22 Stunden entschlüsselt [Beutelspacher ^{6}2002, S. 19].

7 Asymmetrische Kryptosysteme (Public-Key Kryptographie)

Im Jahre 1976 veröffentlichten *Whitfield Diffie* und *Martin Hellman* das Prinzip der **Public-Key Kryptographie**. Mit ihrer bahnbrechenden Arbeit „*New Directions in Cryptography*“ (und dem zwei Jahre später veröffentlichten RSA-Algorithmus) wurde ein bisher „unlösbares“ Problem denkbar elegant gelöst: Während bei der symmetrischen Kryptographie je zwei Teilnehmer, die geheim miteinander kommunizieren wollen, schon *vorher* einen gemeinsamen geheimen Schlüssel vereinbaren müssen[7], ist dies in der Public-Key Kryptographie nicht mehr nötig. Bei asymmetrischen Algorithmen verfügt jeder Beteiligte über *zwei zusammengehörige Schlüssel*[8]: Einen, mit dem Nachrichten an ihn *verschlüsselt* werden können, und einen zweiten, mit dem er derart verschlüsselte Nachrichten *entschlüsseln* kann. Es ist nicht möglich, den Entschlüsselungs- aus dem Verschlüsselungsschlüssel herzuleiten. Letzterer kann daher veröffentlicht werden (**öffentlicher Schlüssel, public key**). Geheim gehalten werden muss hingegen der Entschlüsselungsschlüssel (**privater Schlüssel, private key**).
Der Vorteil dieses Systems liegt auf der Hand: Um verschlüsselte Nachrichten erhalten zu können, muss man nur ein Schlüsselpaar erzeugen und den öffentlichen Schlüssel möglichst vielen anderen Personen zugänglich machen, etwa durch Hinterlegen in einer frei einsehbaren Datenbank (**key server**).
Zur Veranschaulichung verwenden wir die in der Literatur übliche Situation:

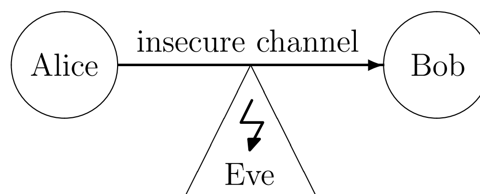

Abbildung 2: Standardsituation der Public-Key Kryptographie

Alice will Bob eine Mitteilung schicken, Eve (vom Englischen „eavesdropper“) versucht diese Nachricht zu belauschen. *„This ist how Alice can send a message to Bob using public-key cryptography [Schneier 1996, S. 32]:*

1. *Alice gets Bob's public key from the database;*
2. *Alice encrypts her message using Bob's public key an sends it to Bob;*
3. *Bob then decrypts Alice's message using his private key.“*

[7]Bei x potentiellen Kommunikationspartnern muss man bei symmetrischen Verfahren dazu $\binom{x}{2} = \frac{x(x-1)}{2} = O(x^2)$ Schlüssel verwalten – das sind bei $x = 1000$ immerhin 499500.

[8]Man benötigt hier für $x = 1000$ nur noch 2000 Schlüssel. Allerdings ist die Schlüsselverwaltung nicht mehr so einfach (**man-in-the-middle attack**, ...).

Die *Sicherheit* asymmetrischer Kryptosysteme *beruht* letztlich *auf einer vermuteten algorithmischen Schwierigkeit eines mathematischen Problems.* Verwendung finden sogenannte **Einwegfunktionen mit Hintertür** (**trapdoor one-way functions**) – diese sind leicht zu berechnen, aber sehr schwer wieder umzukehren, wenn keine Zusatzinformation gegeben ist (**secret trapdoor**). Der private Schlüssel dient also als eine Art „Hintertür", um die Verschlüsselung doch noch effizient umkehren zu können. Aus mathematischer Sicht gibt es weder einen Beweis, dass Einwegfunktionen (geschweige denn solche mit trapdoors) überhaupt existieren, noch eine echte Spur, wie solche konstruiert werden können. Im Bereich der Modul-Arithmetik werden jedoch derartige Funktionen vermutet.

7.1 Mathematische Basics

7.1.1 Komplexitätstheorie

Erfahrungsgemäß ist ein minimaler Zuwachs der Komplexität eines Kryptoalgorithmus mit einem sehr viel größeren Anwachsen der Ressourcen zum Brechen des Systems verbunden. Die Grundfrage der Komplexitätstheorie ist die nach dem Aufwand zur Lösung eines Problems. *„Complexity theory provides a methodology for analyzing the* ***computational complexity*** *of different cryptographic techniques and algorithms. It compares cryptographic algorithms and techniques and determines their security. (...) [It] tells us whether they can be broken before the heat death of the universe"* [Schneier 1996, S. 237].
Die entscheidende Frage dabei ist, wie stark der Aufwand zum Brechen eines Kryptosystems im *„worst case"* von den Eingangsparametern (beispielsweise der Schlüssellänge N) abhängt. Konstante Faktoren bleiben dabei unberücksichtigt (Maschinenunabhängigkeit).
Wir beschränken uns auf folgende Komplexitätsklassen:

- **P**: Probleme, die mit *polynomialem Aufwand lösbar* sind – $O(N^x), x \in \mathbb{N}$;
- **NP**: Probleme, bei denen die *Verifizierung einer gegebenen Lösung* mit *polynomialem Aufwand möglich* ist (*nichtdeterministisch polynomial*);
- **EXP**: *exponential* – $O(c^{f(N)})$, wobei $c = const$, f ein Polynom $\neq const$. Eine brute force attack gegen ein Kryptosystem mit Bit-Schlüssel der Länge N ist von der Komplexität $O(2^N)$.

Eine interessante bislang ungelöste Frage ist: $\mathbf{P} \neq \mathbf{NP}$?
Hingegen konnte gezeigt werden, dass $\mathbf{P} \neq \mathbf{EXP}$ gilt [Schneier 1996, S. 241].
Ein sinnvolles Kryptosystem sollte den Kryptoanalysten *„fast immer"* vor ein Problem aus $\mathbf{NP}-\mathbf{P}$ stellen (nur ein *„worst case"*-Instanz genügt nicht!).

7.1.2 Zahlentheorie

Die Grundlage für fast alle Public-Key Algorithmen ist die Modulo-Rechnung. Wir geben deshalb einige zahlentheoretische Anmerkungen: ($n \in \mathbb{N}$)

1. Mit $\mathbb{Z}/n\mathbb{Z}$ sei wie üblich die **Restklassenmenge modulo n** bezeichnet. Darauf kann wie gewohnt addiert und multipliziert werden $(mod\, n)$. $\mathbb{Z}/n\mathbb{Z}$ bildet bezüglich der Addition eine kommutative Gruppe.

2. Für das **modulare Inverse** von $a \in \mathbb{Z}$ gilt der wichtige **Satz**:
   Die Kongruenz $a \cdot x \equiv 1\, mod\, n$ besitzt genau dann eine eindeutige Lösung $x \in \{1, \ldots, n-1\}$, falls $ggT(a, n) = 1$ ist. Die Lösung x heißt multiplikative Inverse modulo n von a und wird mit $a^{-1} mod\, n$ bezeichnet.
   BEWEIS
   Der Beweis wird sich konstruktiv aus der später diskutierten Vielfachsummendarstellung ergeben. $\sharp$
   Beispielsweise haben 5 und 11 kein multiplikatives Inverses $mod\, 55$.
   Mit $(\mathbb{Z}/n\mathbb{Z})^\times$ sei die Menge der Restklassen modulo n bezeichnet, deren Elemente teilerfremd zu n sind, also modulare Inverse besitzen.
   $(\mathbb{Z}/n\mathbb{Z})^\times$ bildet bezüglich der Multiplikation eine kommutative Gruppe.

3. **Satz (*)**
   Für eine Primzahl $p \in \mathbb{P}$ ist $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$ ein Körper, und die Einheitengruppe $\mathbb{F}_p^\times := \mathbb{F}_p - \{0\}$ ist zyklisch, d.h. wird von den Potenzen eines Elements $\omega \in \mathbb{F}_p$ erzeugt (i.e. $\mathbb{F}_p^\times = \langle \omega \rangle$)[9].

4. Die **Eulersche φ-Funktion** ist definiert als

   $$\varphi(n) := \sharp \{1 \leq a \leq n-1 : \ ggT(a, n) = 1\} = |\ (\mathbb{Z}/n\mathbb{Z})^\times\ | .$$

   Man überlegt sich leicht:

   $$\begin{aligned} p \in \mathbb{P} \quad &\Rightarrow \quad \varphi(p) = p - 1; \\ p, q \in \mathbb{P},\ p \neq q \quad &\Rightarrow \quad \varphi(pq) = (p-1)(q-1). \end{aligned}$$

5. **Satz von Euler**[10]
   Sind $a \in \mathbb{Z}$ und $n \in \mathbb{N}$ teilerfremd, so gilt:

   $$a^{\varphi(n)} \equiv 1 mod\, n.$$

6. Als Spezialfall ergibt sich der **Kleine Satz von Fermat**
   Sind $a \in \mathbb{Z}$ und $p \in \mathbb{P}$ teilerfremd, so gilt:

   $$a^{p-1} \equiv 1 mod\, p.$$

7. **Folgerung aus dem Satz von Euler**
   Sind $p, q \in \mathbb{P}$ zwei verschiedene Primzahlen, und $m \in \mathbb{N}$ eine natürliche Zahl $m \leq p \cdot q =: n$, dann gilt für jedes natürliche $k \in \mathbb{N}$:

   $$m^{1+k\varphi(n)} \equiv m\, mod\, n.$$

[9] Insbesondere ist $a \in \{1, \ldots, p-1\}$ modulo p invertierbar.

[10] Mit Hilfe des Satzes von Euler können unter anderem **modulare Inverse** berechnet werden: Sind $a \in \mathbb{Z}$ und $n \in \mathbb{N}$ teilerfremd, so ist $a^{-1} mod\, n$ gegeben durch $a^{\varphi(n)-1} mod\, n$, denn $(a \cdot a^{\varphi(n)-1}) = a^{\varphi(n)} \equiv 1 mod\, n$.

Beweis
Wir setzen $h := 1 + k\varphi(n)$. Entweder teilt p die Zahl m (und damit $m^h \equiv m \, mod \, p \equiv 0 \, mod \, p$) oder es folgt mit dem Kleinen Fermat:

$$\begin{aligned} m^h = m^{1+k(p-1)(q-1)} &= m \cdot (m^{(p-1)})^{k(q-1)} \\ &\equiv (m \cdot 1^{k(q-1)}) \, mod \, p \\ &\equiv m \, mod \, p. \end{aligned}$$

Und analog natürlich auch $m^h \equiv m \, mod \, q$.
Also teilen sowohl p als auch q die Zahl $m^h - m$. Da p und q *verschiedene* Primzahlen sind, muss auch das Produkt $p \cdot q$ die Zahl $m^h - m$ teilen:

$$(m^h - m) \equiv 0 \, mod \, (pq) \quad \Longleftrightarrow \quad m^h \equiv m \, mod \, (pq). \sharp$$

8. **Euklidischer Algorithmus**
Der Euklidische Algorithmus liefert den $ggT(a, b)$ zweier natürlicher Zahlen a und b. Kurz:

$$ggT(a, b) = \begin{cases} b, & \text{falls } a \, mod \, b = 0 \\ ggT(b, a \, mod \, b), & \text{sonst (etwa } a = q \cdot b + r \; [1 \leq r \leq b - 1]). \end{cases}$$

Dieser Algorithmus läßt sich effizient realisieren.

9. **Satz von der Vielfachsummendarstellung**
Für $a, \, b \in \mathbb{Z}$ existieren $x, \, y \in \mathbb{Z}$ mit:

$$ggT(a, \, b) = x \cdot a + y \cdot b.$$

Beweis
Zum Beweis wendet man den Euklidischen Algorithmus „umgekehrt" an (sogenannter Erweiteter Euklidischer Algorithmus). $\sharp$
Zurück zur effizienten Berechnung des **modularen Inversen**:
Für teilerfremde Zahlen $a, \, n \in \mathbb{N}$ berechnet sich das multiplikative Inverse modulo n von a wie folgt: Wegen $ggT(a, n) = 1$ existieren $x, \, y \in \mathbb{Z}$ mit:

$$1 = a \cdot x + n \cdot y \quad \Longleftrightarrow \quad a \cdot x = 1 - n \cdot y.$$

Da ny durch n teilbar ist, ergibt ax bei Division durch n den Rest 1, also

$$a \cdot x \equiv 1 \, mod \, n.$$

x ist also die gesuchte modulare Inverse von a. $\sharp$

10. **Primzahlen** und **Primzahltests**
Gegeben sei $p \in \mathbb{N}$.
Problem PRIMES: $p \in \mathbb{P}$?
Das *„Sieb des Erathosthenes"* besagt, dass PRIMES entscheidbar ist.
Mit dem *Satz von Euler* kann man einen „negativen" Primzahltest formulieren, denn ist p eine Primzahl, so muss für beliebiges $a \in \{1, \ldots, p - 1\}$

stets $a^{p-1} \equiv 1\, mod\, p$ gelten. Finden wir also ein a, welches diese Eigenschaft nicht hat, so kann p keine Primzahl sein. Aber: Bleibt die Suche erfolglos, so kann man nicht folgern, dass p prim ist[11]!
Manindra Agrawal, *Neeraj Kayal* und *Nitin Saxena* konnten im Jahr 2002 folgenden Satz beweisen:
PRIMES is in **P**.
Der dabei entwickelte *AKS-Algorithmus* ist jedoch für Primzahlen der oft in Public-Key Systemen verwendeten Größenordnung von 4096 Bit deutlich langsamer als die in der Praxis eingesetzten Primzahltest. Laufzeitvorteile ergeben sich erst für sehr viel größere p, weshalb dieses theoretische Ergebnis bislang keine praktische Implementierung ermöglicht.
Der **Primzahlsatz** macht eine asymptotische Aussage über die Anzahl $\pi(x) := \sharp \{p \in \mathbb{P} :\, p \leq x\}$ der Primzahlen kleiner gleich $x \in \mathbb{R}^+$:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/log(x)} = 1.$$

In der Praxis verwendet man sogenannte **probabilistische Primzahltests** (beispielsweise den Miller-Rabin-Test). Man wählt eine Zufallszahl der gewünschten Größe und testet mit kleiner Fehlerwahrscheinlichkeit, ob es sich dabei um eine Primzahl handelt. Mit dem Primzahlsatz gelangt man so in erstaunlich kurzer Zeit zum Ziel.

11. Die **modulare Exponentiation** $a^b mod\, n$ kann mit Hilfe des *Square-and-Multiply* Algorithmus effizient (polynomial) gelöst werden. Beispielsweise geht man zur Berechnung von $a^9 mod\, n$ wie folgt vor:

$$a^9 mod\, n = (a \cdot ((a^2)^2)^2) mod\, n = (((a^2 mod\, n)^2 mod\, n)^2 \cdot a) mod\, n.$$

Man benötigt dabei nur 4 kleinere Multiplikationen und 3 Modulo-Operationen gegenüber 7 und einer großen Modulo-Reduktion beim naiven Ausmultiplizieren.

12. **Das Problem des diskreten Logarithmus**
Dieses Problem ist die *Umkehrung der modularen Exponentiation.* Zu finden ist also für a, b und n ein x, so dass $b \equiv a^x\, mod\, n$ ist. Heutige Algorithmen können diese Aufgabe nur sehr ineffizient bewältigen ($\notin$ **P**).
Die zugehörige Einwegfunktion ist somit die modulare Exponentiation. Das Problem des diskreten Logarithmus scheint ebenso schwer zu sein, wie das der Faktorisierung großer Zahlen.

13. **Das Problem der Faktorisierung**
Faktorisieren einer Zahl $n \in \mathbb{N}$ bedeutet, sie *in ihre Primfaktoren zu zerlegen.* Es wurde bisher noch kein guter (polynomialer $\in$ **P**) Faktorisierungsalgorithmus veröffentlicht, noch ist bekannt, ob es überhaupt einen solchen geben kann.

[11] Eine *zusammengesetzte* Zahl n, die $a^{n-1} \equiv 1\, mod\, n$ für alle $a \in \{1, \ldots, n-1\}$ erfüllt, heißt *Carmichael*-Zahl (diese Zahlen schlagen beim negativen „Euler-Test“ fehl). Die kleinste Carmichael-Zahl ist: $561 = 3 \cdot 11 \cdot 17$. Carmichael-Zahlen können nicht vernachlässigt werden!

7.2 Schlüsselaustausch nach Diffie und Hellman

Der **Diffie-Hellman Schlüsselaustausch** war der erste Algorithmus, der Public-Key benützt. *„[It] can be used for key distribution – Alice and Bob can use this algorithm to generate a secret key – but it cannot be used to encrypt and decrypt messages“ [Schneier 1996, S. 513].* Verwendung findet dabei der **Satz (*)** des letzten Abschnitts 7.1.2.
Das **Schlüsselaustauschprotokoll** lautet:

1. Alice und Bob vereinbaren eine große Primzahl $p \in \mathbb{P}$ und ein ω mit der Eigenschaft $\mathbb{F}_p^\times = \langle\omega\rangle$. Die Geheimhaltung von p und ω ist nicht nötig.

2. Alice wählt zufällig ein großes $a \in \mathbb{N}$ und sendet Bob $A := \omega^a mod\, p$. Auch Bob wählt seinerseits zufällig ein großes $b \in \mathbb{N}$ und sendet Alice entsprechend $B := \omega^b mod\, p$. Die Zahlen a und b sind jeweils geheime Schlüssel.

3. Beide potenzieren den erhaltenen Wert nochmals mit ihrem geheimen Schlüssel $mod\, p$ – Alice berechnet also $k := B^a mod\, p$ und Bob entsprechend $k' := A^b mod\, p$.

Dann ist $k^\star := k = k' = \omega^{ab} mod\, n$ der gemeinsame geheime Schlüssel.

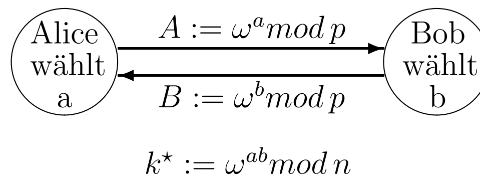

Abbildung 3: Diffie-Hellman Schlüsselaustausch

Eve kann aus p, ω, A und B den Schlüssel $k^\star$ nicht berechnen, solange sie nicht den *diskreten Logarithmus* und damit a oder b berechnen kann.
Verallgemeinert wird das Prinzip von Diffie-Hellman beim **ElGamal-Verschlüsselungsverfahren** in dem Sinne, dass das Senden von Teilschlüsseln zeitlich entkoppelt ist, und damit unmittelbar Nachrichten verschlüsselt werden können.

7.3 Der RSA-Algorithmus

1979 publizierten *Ronald Rivest*, *Adi Shamir* und *Leonard Adleman* vom Massachusetts Institute for Technology (MIT) das **RSA-Verfahren** zum sicheren Austausch von Nachrichten. Seine Sicherheit beruht auf dem Problem der *Faktorisierung großer Zahlen.* Sowohl die Schlüsselerzeugung als auch der Ver- und Entschlüsselungsvorgang sind effizient berechenbar.

7.3.1 Schlüsselerzeugung

Die **Schlüsselerzeugung** des RSA-Algorithmus funktioniert wie folgt:

- Man wählt zwei große Primzahlen $p,\, q \in \mathbb{P}$, $p \neq q$ und berechnet $n := pq$.
- Es ist dann $\varphi(n) = (p-1)(q-1)$.
- Man wählt ein $e \in \mathbb{N}$ teilerfremd zu $\varphi(n)$ und berechnet ein d, so dass $ed \equiv 1\, mod\, \varphi(n)$ gilt[12].
- **Privater Schlüssel** ist nun d (und n).
- **Öffentlicher Schlüssel** ist das Paar $(e,\, n)$.
- Geheime Parameter der Schlüsselerzeugung sind $p,\, q,\, \varphi(n)$. Diese sollten nach der Schlüsselerzeugung vernichtet werden!

7.3.2 Verschlüsselung

Der Klartext $m \in M$ wird zunächst in eine Bitfolge übersetzt (beispielsweise via ASCII). Die Nachricht – dargestellt als eine Folge von Nullen und Einsen – wird nun in Blöcke fester Größe zerlegt, und jeder dieser Blöcke einzeln chiffriert. OBdA können wir also $m \in \mathbb{N}$, $m \leq n-1$ annehmen.
Mit Hilfe des öffentlichen Schlüssels $(e,\, n)$ erhält man für m das Chiffrat mittels *Square-and-Multiply* als

$$c := E_{(e,\, n)}(m) = m^e mod\, n.$$

7.3.3 Entschlüsselung

Die Entschlüsselung verwendet das gleiche Prinzip, potenziert das Chiffrat c aber mit d:

$$m' := D_d(c) = c^d mod\, n.$$

Wir erhalten:

$$m = m',$$

denn zunächst gilt $\forall\, m \in M$ mit $c = m^e mod\, n$:

$$m' = c^d mod\, n \equiv (m^e)^d mod\, n \equiv m^{ed} mod\, n.$$

Aus der Definition von e und d folgt wegen $ed \equiv 1 mod\, \varphi(n)$:

$$ed = 1 + k\varphi(n), \text{ für ein } k\ \in\ \mathbb{N}.$$

Also nach dem Satz von Euler (bzw. der Folgerung):

$$m^{1+k\varphi(n)} mod\, n = m, \quad \text{also} \quad D_d \circ E_{(e,\, n)} = id.$$

[12] Da e und $\varphi(n)$ teilerfremd sind, ist e modulo $\varphi(n)$ invertierbar. Die Berechnung von d kann mit dem Erweiterten Euklidischen Algorithmus effizient gelöst werden (siehe oben).

7.3.4 Ein kleines Zahlenbeispiel

Alice will Bob die geheime Nachricht $m = 8$ zukommen lassen.
Bob erzeugt zunächst ein Schlüsselpaar mit $p = 29$ und $q = 37$, berechnet $n = 29 \cdot 37 = 1073$ und $\varphi(1073) = (29-1)\cdot(37-1) = 1008$. Er wählt $e = 5$ (wobei $ggT(5, 1008) = 1$ gilt) und löst die Kongruenz $5 \cdot d = 1 \, mod \, 1008$ mit $d = 605$ (denn $5 \cdot 605 = 3025 \equiv 1 \, mod \, 1008$). Öffentlicher Schlüssel von Bob ist das Paar $(e, n) = (5, 1073)$, geheim hält er den privaten Schlüssel $d = 605$.
Alice verschlüsselt den Klartext $m = 8$ durch $m^e = 8^5 = 32768 \equiv 578 \, mod \, 1073$ und sendet das Chiffrat $c = 578$ an Bob.
Bob verifiziert $c^d = 578^{605} \equiv 8 \, mod \, 1073$ und erhält die Nachricht $m = 8$ zurück.

7.3.5 Zur Sicherheit von RSA

Die Sicherheit des RSA-Algorithmus hängt von der Zeit ab, in der Unberechtigte eine *Primfaktorzerlegung* von $n = pq$ finden können. Ohne die Kenntnis von p und q (oder $\varphi(n)$[13]) ist es praktisch unmöglich (ineffizient), die Kongruenz $ed \equiv 1 \, mod \, \varphi(n)$ zu lösen.
Bei der heute erreichten Rechenleistung von Computern sollten p und q so gewählt werden, dass n eine Zahl zwischen 1024 und 4096 Bit Länge ist (p und q in möglichst gleicher Größenordnung). Auf die Formulierung geeigneter Voraussetzungen zur Wahl von p, q und e werden wir hier verzichten.
Eine andere bislang ungelöste Frage ist, ob es noch andere Wege außer der Faktorisierung von n gibt, um RSA zu brechen.

7.3.6 RSA als Signaturverfahren

Digitale Signaturverfahren verfolgen **zwei wesentliche Ziele**:

1. **Datenintegrität**: Schutz einer Nachricht vor Manipulation;
2. **Authentifikation**: Sicherstellung der Identität des Senders.

Grundanforderungen an eine digitale Unterschrift (*Prüfcode über einer Nachricht*) sind:

- **Authentizität**: Nur eine Person darf in der Lage sein, die Unterschrift zu erzeugen;
- **Verifizierbarkeit**: Alle Personen können die Echtheit der Unterschrift prüfen;
- **Verbindlichkeit**: Ein Unterzeichner darf später nicht abstreiten können, eine digitale Unterschrift geleistet und erzeugt zu haben.

[13] $\varphi(n) = (p-1)(q-1)$ ist genauso schwer zu berechnen, wie die Faktorisierung von n. Sind p und q bekannt, so kann $\varphi(n) = (p-1)(q-1)$ leicht berechnet werden. Umgekehrt gilt aber auch: Sind n und $\varphi(n)$ bekannt, so kann man n faktorisieren – p und q sind dann Lösungen des Gleichungssystems: $pq = n$ und $(p-1)(q-1) = \varphi(n)$.

Neben speziellen Signaturalgorithmen wie beispielsweise dem von der NSA entwickelten **Digital Signature Algorithm** (**DSA**) kann auch **RSA zur Signatur von Nachrichten** benützt werden, indem eine unfälschbare Unterschrift erzeugt wird. Wir wollen das Vorgehen kurz beschreiben:
Mit obigen Bezeichnungen gilt nicht nur $D_d \circ E_{(e,\,n)} = id$, sondern auch:

$$E_{(e,\,n)} \circ D_d = id.$$

Alice möchte eine Nachricht m an Bob signieren. Es sei hier d der private und das Paar $(e,\, n)$ der öffentliche Schlüssel von Alice. Das **Protokoll** lautet nun:

- **Erstellen der Signatur**: Um die Nachricht m zu signieren verschlüsselt Alice m mit ihrem *privaten* Schüssel d, berechnet also $sig := m^d mod\, n$. Es entsteht eine authentische Unterschrift sig.

- **Verifizieren der Signatur**: Eine von Alice erstellte Signatur wird verifiziert, indem ihr *öffentlicher* Schlüssel auf die Signatur angewandt, also $m' := sig^e mod\, n$ berechnet wird.

- Gilt $m = m'$, so wird die Signatur akzeptiert.

In der Praxis wird Alice aus Performancegründen[14] m zunächst mittels einer sogenannten **Einweg-Hash-Funktion** komprimieren (sie erzeugt eine sogenannte *„message digest"/Textauswahl*) und den *Signaturalgorithmus nur auf das Komprimat anwenden.*
Eve müsste die Nachricht entweder so ändern, dass die Prüfsumme gleich bleibt, was praktisch unmöglich ist, oder sie müsste mit der geänderten Prüfsumme eine neue digitale Unterschrift $sig_\star$ erzeugen, was ohne den geheimen privaten Schlüssel d von Alice ebenfalls nicht möglich ist.

7.4 Schlüsselmanagement durch Trusted Third Parties

Auch asymmetrische Kryptosysteme lösen folgende Probleme nur teilweise:

- Wie findet Alice den öffentlichen Schlüssel von Bob?

- Kann Alice sicher sein, dass der erhaltene Schlüssel wirklich Bob gehört?

Gerade bei großen Netzen, in denen es häufig zu *„Erstkontakten"* zwischen Kommunikationspartner kommt, die sich *nicht* kennen, sind solche Fragestellungen von entscheidender Bedeutung.
Eine Möglichkeit zur Lösung dieser Probleme ist die Einführung einer „**Trusted Third Party**" (**TTP**)[15], die die *Authentizität öffentlicher Schlüssel überprüft.* Eine solche TTP ist eine Institution, der alle Teilnehmer vertrauen [vgl. Beutelspacher/Schwenk/Wolfenstetter ^{3}1999, S. 92 f].

[14] Beim beschriebenen Vorgehen verdoppelt sich die Länge der zu übertragenden Information, da die Signatur ebenso lang ist, wie die Nachricht.

[15] Bei Public-Key Systemen spricht man auch von einer *„Certification Authority"* (CA).

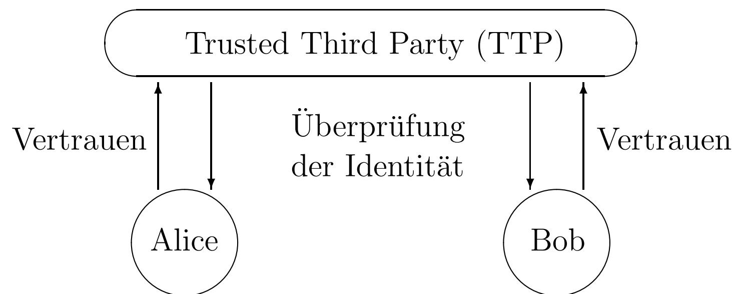

Abbildung 4: Prinzip der Trusted Third Parties (TTP)

Ein Teilnehmer muss sich zunächst bei der TTP anmelden und authentifizieren. Dabei erfolgt die Überprüfung der Personalien in mehreren Sicherheitsklassen[16]. Danach wird der Teilnehmer zusammen mit seinem public key und einem nach der Authentifikation von der TTP erstellten Zertifikat in das Register der TTP aufgenommen. Die TTP muss also für jeden Teilnehmer einmal aktiv werden, nämlich bei der Überprüfung seiner Identität und dem anschließenden Signieren seines öffentlichen Schlüssels.

Grundanforderungen an die TTP sind unter anderem:

- Sicherheit bei der **Authentifikation der Teilnehmer**;
- Gewährleistung der **Integrität der Daten**;
- Bereitstellung einer **sicheren Datenbankapplikation** (dezentral);
- Einbruchsicherheit;
- Einhaltung von Datenschutzrichtlinien;
- Unbeeinflussbarkeit und Vertrauenswürdigkeit.

In Deutschland agieren zum Beispiel *Telesec* (Telekom) und *Signtrust* (Deutsche Post) als TTP.

7.5 Vor- und Nachteile asymmetrischer Kryptosysteme

Vorteile:

- Wenige Schlüssel nötig;
- Schlüsselaustausch elegant gelöst;
- Digitale Unterschrift wird unterstützt.

[16] Von *„Ohne strenge Überprüfung der Personalien“* bis hin zum *persönlichen Erscheinen bei der TTP.*

Nachteile:

- Deutlich langsamer als symmetrische Verfahren (RSA ist ca. 1000 mal langsamer als DES und 4000 mal langsamer als IDEA);
- Lange Schlüssel nötig;
- Schlüsselverwaltung notwendig (Vertrauen);
- Grundlage sind Vermutungen über Schwierigkeiten bei der algorithmischen Lösung mathematischer Sachverhalte.

7.6 Fazit

Der *Reiz an Public-Key* ist nicht die Verschlüsselung selbst, sondern die *wesentlich bequemere und vielseitigere Handhabung der Schlüssel.*

8 Hybride Kryptosysteme

In der Praxis werden aus Performancegründen überwiegend **hybride Verfahren** eingesetzt. Diese *kombinieren symmetrische und asymmetrische Kryptosysteme*, um die jeweiligen Vorteile zu nutzen.
Schlüsselaustausch eines sogenannten *random session key* k (und *digitale Signatur*) werden *mittels asymmetrischen Verfahren* (E^{asym}, D^{asym}) realisiert, während die *eigentliche Nachrichtenverschlüsselung* ein *symmetrisches Verfahren* ($\tilde{E}^{sym}$, $\tilde{D}^{sym}$) mit dem ausgetauschten Schlüssel k benützt.
Verwendung findet somit folgendes Schema (d_{Bob} sei privater und e_{Bob} öffentlicher Schlüssel von Bob im asymmetrischen Verfahren):

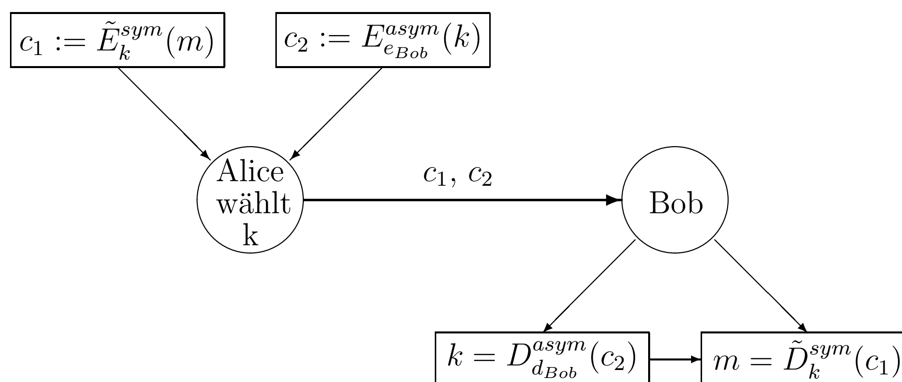

Abbildung 5: Prinzip hybrider Verfahren

Der Einsatz solcher Kryptosysteme erlaubt sowohl die *sichere*, als auch *effiziente* Übertragung geheimer Informationen.

9 Anwendung: Pretty Good Privacy (PGP)

Pretty Good Privacy (**PGP**) ist ein *Programm zur Verschlüsselung und digitalen Signatur* von E-M@ils und allgemeiner Dateien. Entwickelt im Jahre 1990 von *Philip Zimmermann („Public Key for the masses“)* steht es heute nicht nur für kommerzielle Zwecke, sondern auch als Freeware-Version beispielsweise unter `http://www.pgpi.org/` zum kostenlosen Download zur Verfügung. `Pgpi.org` ist die offizielle internationale PGP-Homepage, von der die amerikanische und internationale Version von PGP heruntergeladen werden kann. Es stehen mehrere Versionen für eine Vielzahl gängiger Plattformen bereit: *„Wer außerhalb der Vereinigten Staaten lebt, sollte die amerikanische Version von PGP nicht herunterladen, weil er damit amerikanische Exportgesetze verletzen würde. Die internationale Version von PGP unterliegt keinen Exportbeschränkungen“* [Singh 1999, S. 380 f].
Für die folgenden Ausführungen wurde

```
Pretty Good Privacy(tm) 2.6.3i - Public-key encryption for the masses.
(c) 1990-96 Philip Zimmermann, Phil's Pretty Good Software. 1996-03-04
International version - not for use in the USA. Does not use RSAREF.
```

unter Linux verwendet, dessen Source-Code über `http://www.pgpi.org/` bezogen werden kann. Unter der allgemeinen Bezeichnung PGP verstehen wir im Folgenden somit stets PGP 2.6.3i!
PGP verwendet ein *hybrides System auf Basis des RSA-Algorithmus – IDEA realisiert die konventionelle Verschlüsselung*[17], und das *Message-Digest-5* Verfahren (MD5)[18] die *Erzeugung elektronischer Unterschriften.*

9.1 Funktionalitäten von PGP

PGP bietet im Wesentlichen fünf Möglichkeiten der Dateibearbeitung:

1. **Die Verschlüsselung einer Nachricht**.
2. **Digitale Unterschrift**.
3. **Datenkomprimierung**.
   Normalerweise komprimiert PGP den Klartext, bevor er verschlüsselt wird. Verschlüsselte Daten lassen sich hingegen nicht mehr komprimieren[19]. Dies spart nicht nur Übertragungszeit und Festplattenkapazität,

[17] ⇒ Ist IDEA gebrochen, so auch PGP. Die Verwendung von IDEA ist aber <u>nicht</u> als Schwachstelle von PGP anzusehen, da eine brute force attack gegen IDEA mit den 2^{128} möglichen Schlüsseln vergleichbar ist mit dem Rechenaufwand der Faktorisierung eines RSA-Schlüssels n von ca. 3100 Bit.

[18] Dieses Verfahren erzeugt mit einer Einweg-Hash-Funktion aus der Nachricht eine 128-Bit-Zahl, welche die Nachricht „eindeutig genug“ identifiziert. Signiert wird darauf basierend entsprechend den Anmerkungen zur RSA-Signatur.

[19] Das liegt daran, dass gut verschlüsselte Daten „sehr zufällig“ aussehen, Kompressionsalgorithmen aber darauf basieren, eine bestimmte Systematik in den Daten zu erkennen und auf Grundlage solcher Muster eine kürzere Darstellung der Daten suchen.

sondern erhöht auch die Sicherheit der Verschlüsselung, da viele kryptoanalytische Techniken die Redundanz des Klartextes voraussetzen, die durch eine Datenkompression jedoch reduziert wird.

4. **Aufbereitung von Dateien für E-M@il** (mit dem Radix-64 Format). Wird ein Text mit einer digitalen Unterschrift versehen, komprimiert und verschlüsselt, so besteht das Ergebnis aus Buchstaben und nicht druckbaren Zeichen (Binärdaten). Die meisten M@il-Systeme können dies jedoch nicht verarbeiten. PGP konvertiert das erhaltene Ergebnis in Text (im sogenannten Radix-64 Format) und macht das Chiffrat damit transportfähig.

5. **Aufteilen und Zusammensetzen von Dateien**.
   Die Größe der versandfähigen Dateien ist in vielen M@il-Programmen limitiert. PGP zerlegt zu große Dateien automatisch in kleinere, die dann einzeln verschickt werden können. Beim Entschlüsseln werden die einzelnen Dateien wieder zu der ursprünglichen zusammengefügt.

9.2 Schlüsselverwaltung bei PGP

PGP erstellt benutzerfreundlich die RSA-Schlüssel automatisch. Diese Schlüssel werden in sogenannten *„key certificates"* aufbewahrt, die neben dem Schlüssel selbst noch einige andere Informationen enthalten – „**public key certificates**" für öffentliche und „**secret key certificates**" entsprechend für private Schlüssel werden in Dateien (meist `pubring.pgp` bzw. `secring.pgp`, **Schlüsselbund**) hinterlegt. Diese Dateien sollte der Benutzer auf seinem eigenen PC speichern (und nicht über Netzwerk darauf zugreifen). Sicherheitskopien der Schlüsseldateien sind ebenfalls erforderlich, da bei einem Verlust der Schlüssel (beispielsweise durch einen Defekt des Speichermediums) das Schlüsselpaar nicht mehr rekonstruiert werden kann. Private Schlüssel sind mit einem **Mantra** (**pass phrase**) geschützt, einem potentiell beliebig langen Passwort(-satz). Fallen sowohl Bobs privater Schlüssel als auch sein Mantra in Eves Hände, sollte er ein sogenanntes „**key compromise/revocation certificate**" (*„Schlüssel-Rückrufurkunde"*) ausstellen – eine Urkunde, die besagt, dass sein Schlüssel auf keinen Fall mehr verwendet werden darf. Da es *keine zentrale TTP* gibt, stellt dies ein organisatorisches Problem für Bob dar: Jeder, der Bob eine vertrauliche Nachricht übermitteln will, muss davon in Kenntnis gesetzt werden, dass sein alter Schlüssel nicht mehr verwendet werden darf. Da Bob jedoch im Allgemeinen potentielle Kommunikationspartner nicht kennt (= Erstkontakt) und auch nicht nachvollziehen kann, wie diese an seinen öffentlichen Schlüssel gelangt sind (beispielsweise über Dritte oder key server), kann er nicht ausschließen, dass mit dem alten Schlüssel chiffrierte Nachrichten über den insecure channel gesendet werden, welche Eve nicht nur abfangen, sondern nun auch entschlüsseln kann[20].

[20] *„After creating a „key compromise certificate" Bob must somehow send this to everyone else on the planet, or at least to all his friends. Their own PGP software will install this key*

Wie schon erwähnt, gibt es bei PGP keine zentrale TTP zur Schlüsselverwaltung. Vielmehr speichert jeder User in seinem öffentlichen Schlüsselbund die public keys der Kommunikationspartner ab, mit denen er mittels PGP kommunizieren möchte. Die Authentizität der öffentlichen Schlüssel wir durch einen trickreichen Zertifikatsmechanismus gewährleistet – einem sogenannten **Vertrauensnetz** (**Web-of-Trust**). *„Distributed key management, used in PGP, (...) works with **introducers**. Introducers are other users of the system who sign their friends' public keys. For example, when Bob generates his public key, he gives copies to his friends: Carol and Dave. They know Bob, so they each sign Bob's key and give Bob a copy of the signature. Now, when Bob presents his key to a stranger, Alice, he presents it with the signatures of these two introducers. If Alice also knows and trusts Carol, she has reason to believe that Bob's key is valid. If she knows and trusts Carol and Dave a little, she has reason to believe that Bob's key is valid. If she doesn't know either Carol or Dave, she has no reason to trust Bob's key*[21]*.*
Over time, Bob will collect many more introducers. If Alice and Bob travel in similar circles, the odds are good that Alice will know one of Bob's introducers. To prevent against Eve's substituting one key for another, an introducer must be sure that Bob's key belongs to Bob before he signs it. Perhaps the introducer should require the key be given face-to-face or verified over the telephone.
The benefit of this mechanism is that there is no TTP that everyone has to trust. The down side is that when Alice receives Bob's public key, she has no guarantee that she will know any of the introducers and therefore no guarantee that she will trust the validity of the key“ [Schneier 1996, S. 187].
„PGP does not specify a policy for establishing trust; users are free to decide who they trust and who they do not. (...) Each key [in `pubring.pgp`*] has a key legitimacy field that indicates the degree to which the particular user trusts the validity of the key. The higher the trust level, the more the user believes the key is legitimate. A signature trust field measures how far the user trusts the signer to certify the public key of other users. And finally, an owner trust field indicates the degree to which the particular user trusts the key's owner to sign other public keys“* [Schneier 1996, S. 585].
*„There are **PGP public key servers** which allow one to exchange public keys. (...) NOTE! It is here only to help transfer keys between PGP users. It does NOT attempt to guarantee that a key is a valid key; use the introducers on a key for that kind of security.“* [PGP 2.6.3i documentation, vol. I].
Zur Vereinfachung der Adressierung von key servern wurde die Domain `pgp.net` eingerichtet, unter der alle key server einheitlich erreichbar sein sollten. *„The* `pgp.net` *domain services run on a number of replicated servers. Using the name* `pgp.net` *for Web access will get you a random one“* [`http://www.pgp.net`].

compromise certificate on their public key rings and will automatically prevent them from accidentally using Bob's public key ever again“ [PGP 2.6.3i documentation, vol. I].

[21] *„Trust is not necessarily transferable; I have a friend who I trust not to lie. He's a gullible person who trusts the President not to lie. That doesn't mean I trust the President not to lie“* [PGP 2.6.3i documentation, vol. I].

9.3 Zur Erzeugung von random session keys

PGP verwendet einen *kryptographisch zuverlässigen Pseudozufallsgenerator*, um random session keys für die konventionelle Verschlüsselung mittels IDEA zu erzeugen. Die Datei, die benötigte Informationen für den Zufallsgenerator enthält, heißt `randseed.bin`. Vor und nach der Erzeugung einer Zufallszahl liegt die Datei in verschlüsselter Form auf der Festplatte. Falls die Datei nicht vorhanden ist, wird sie automatisch erzeugt. Sie erhält einen Startwert aus Zufallszahlen, die aus „*Umgebungslärm*“ (beispielsweise Schwankungen der Festplattengeschwindigkeit, zeitlichen Abständen von Tastatureingaben, Mausbewegungen, ...) abgeleitet werden.

9.4 Zur Sicherheit von PGP

Aus Sicht der Kryptoanalyse läßt sich all das anführen, was bereits zur Sicherheit von IDEA und RSA gesagt wurde. Die automatische Datenkomprimierung vor der PGP-Verschlüsselung stellt dabei eine weitere Erschwernis für mögliche Angriffe dar.
Neben meist schwierigen und kostspieligen kryptoanalytischen Angriffen auf PGP sei auch auf andere Angriffstechniken kurz hingewiesen, die insbesondere bei PGP bedeutsam sind [vgl. PGP 2.6.3i documentation, vol. I]:

- Bekannt gewordene Mantras und private Schlüssel.
- Fälschung des öffentlichen Schlüssels (Problem der Authentifikation).
- Schutz gegen gefälschte Zeitangaben.
  Nichts hindert den Benutzer daran, die Einstellungen von Datum und Zeit an seinem Computer zu ändern und danach Schlüssel und Unterschriften zu erzeugen. Daraus entstehende Probleme, sowie mögliche Abhilfe (beispielsweise durch „notariell“ beglaubigte Unterschriften und glaubwürdiger Zeitmarken) wollen wir hier nicht weiter diskutieren.
- Nicht richtig (= physikalisch) gelöschte Dateien.
- Viren, Trojanische Pferde, ...
- Lücken in der physischen Sicherheit.
  Unbefugter Zugriff auf fremde Daten/Computer, Bespitzelung (beispielsweise mittels Auswertung elektromagnetischer Strahlung, die ein Computer aussendet – *„tempest attack“*. Abwehren läßt sich ein solcher Angriff durch eine geeignete Abschirmung des Computers, dessen Peripherie und gegebenenfalls der Netzwerk-Verkabelung. Solche Maßnahmen sind jedoch genehmigungspflichtig.), ...

9.5 Einige PGP-Befehle

PGP kann über einige Parameter, die in der Konfigurationsdatei `config.txt` zu definieren sind, anwenderspezifisch konfiguriert werden. Durch diese Einstellungen kann PGP bequem auf individuelle Bedürfnisse angepasst werden, so dass es nicht erforderlich ist, bei jedem Aufruf von PGP mühsam eine größere Anzahl von Parametern in die Kommandozeile einzutippen. Es sei dazu auf die Dokumentation von PGP 2.6.3i verwiesen!
Im Folgenden wollen wir einige nützliche Dinge im Umgang mit PGP 2.6.3i kennenlernen.

9.5.1 Schlüsselerzeugung

Ein Schlüsselpaar wir durch
`jk@linux:> pgp -kg`
erzeugt. Nach der Eingabe fragt PGP nach der gewünschten Schlüsselgröße (Bit-Größe), einer User-ID und einem Mantra. PGP erzeugt dann die Schlüssel und speichert diese in Dateien (meist `pubring.pgp` bzw. `secring.pgp`) ab.

```
Pick your RSA key size:
   1)  512 bits - Low commercial grade, fast but less secure
   2)  768 bits - High commercial grade, medium speed, good security
   3) 1024 bits - 'Military' grade, slow, highest security
Choose 1, 2, or 3, or enter desired number of bits: 3
Generating an RSA key with a 1024-bit modulus.

You need a user ID for your public key. The desired form for this
user ID is your name, followed by your E-mail address enclosed in
<angle brackets>, if you have an E-mail address. Enter a user ID
for your public key:
Jens Kuehnle <jens.kuehnle@mathematik.uni-ulm.de>

You need a pass phrase to protect your RSA secret key.
Your pass phrase can be any sentence or phrase and may have many
words, spaces, punctuation, or any other printable characters.
Enter pass phrase: ******************************
Enter same pass phrase again: ******************************

Note that key generation is a lengthy process.
We need to generate 993 random bits. This is done by measuring the
time intervals between your keystrokes. Please enter some random text
on your keyboard until you hear the beep:
   0 * -Enough, thank you. ..........**** .........****
Pass phrase is good. Just a moment....
Key generation completed.
```

```
jk@linux:> ls -l ~/.pgp/ # ~/.pgp/ bezeichne das PGP-Verzeichnis
-r-------- 1 jk users 5028 2004-06-05 15:38 config.txt
-rw------- 1 jk users  186 2004-06-05 16:16 pubring.bak
-rw------- 1 jk users  341 2004-06-05 16:16 pubring.pgp
-rw------- 1 jk users  408 2004-06-05 15:52 randseed.bin
-rw------- 1 jk users  519 2004-06-05 16:16 secring.pgp
jk@linux:>
```

9.5.2 Schlüsselverwaltung

To see a quick summary of PGP's key-management commands:
`pgp -k`
To add a key file's contents to your public or secret key ring:
`pgp -ka keyfile [keyring]`[22]
To remove a key or a user ID from your public or secret key ring:
`pgp -kr userid [keyring]`
To edit your user ID or pass phrase:
`pgp -ke your_userid [keyring]`
To edit the trust parameters for a public key:
`pgp -ke userid [keyring]`
To extract (copy) a key from your public or secret key ring:
`pgp -kx[a] userid keyfile [keyring]`[23]
To view the contents of your public key ring:
`pgp -kv[v] [userid] [keyring]`[24]
To view the fingerprint of a public key, to help verify it over the telephone with its owner[25]:
`pgp -kvc [userid] [keyring]`
This will display the key with the 16-byte digest of the public key components. It's best to use a different channel than the one that was used to send the key itself. A good combination is to send the key via E-mail, and the key fingerprint via a voice telephone conversation.
To check (signatures on) your public key ring:
`pgp -kc [userid] [keyring]`

[22][keyring] ist stets optional mit Default-Wert `pubring.pgp` bzw. `secring.pgp`.
PGP prüft zunächst, ob der Schlüssel schon bekannt ist. In diesem Fall wird er nicht wiederholt eingebunden, sondern auf neue Unterschriften und/oder User-IDs untersucht, die gegebenenfalls in den Schlüsselbund aufgenommen werden.

[23]Mit der Option `-kxa` kann ein Schlüssel als Text (beispielsweise als Anhang für E-M@ils) extrahiert werden (armored im Radix-64 Format).

[24]Die Option `-kvv` gibt neben den Schlüsseln zusätzlich alle Unterschriften aus.

[25] *„If you get a public key from someone that is not certified by anyone you trust, how can you tell if it's really their key? The best way to verify an uncertified key is to verify it over some independent channel other than the one you received the key through. One convenient way to tell, if you know this person and would recognize them on the phone, is to call them and verify their key over the telephone. Rather than reading their whole tiresome (ASCII-armored) key to them over the phone, you can just read their key's fingerprint to them“* [PGP 2.6.3i documentation, vol. II].

To sign and certify someone else's public key on your public key ring:
`pgp -ks her_userid [-u your_userid] [keyring]`
To remove selected signatures from a userid on a keyring:
`pgp -krs userid [keyring]`
To generate a key compromise certificate to revoke your own key:
`pgp -kd your_userid`
To disable a foreign public key on your own public key rings:
`pgp -kd her_userid`
To display the web-of-trust:
`pgp -km`

9.5.3 Erste Schritte mit PGP

To see a quick command usage summary for PGP:
`pgp -h`
To decrypt a message:
`pgp message`
To encrypt a plaintext file with recipent's public key:
`pgp -e textfile her_userid [other userids] # produces textfile.pgp`
To sign a plaintext file with your secret key:
`pgp -s textfile [-u your_userid] # produces textfile.pgp`
To sign a plaintext file with your secret key, and then encrypt it with recipent's public key, producing a `.pgp` file:
`pgp -es textfile her_userid [other userids] [-u your_userid]`
To make PGP assume the plaintext is text that should be converted to canonical text[26] before encryption, just add the `-t` option. For example:
`pgp -est textfile her_userid [other userids] [-u your_userid]`
If you need to use the `-t` option a lot, you can just turn on the TEXTMODE flag in the PGP configuration file `config.txt`.
Normally, signature certificates are physically attached to the text they sign. To generate a signature certificate that is detached from the text it signs, combine the `-b` (break) option with the `-s` (sign) option:
`pgp -sb textfile # produces textfile.sig`[27]

[26] *„ASCII text is sometimes represented differently on different machines. For example, on an MSDOS system, all lines of ASCII text are terminated with a carriage return followed by a linefeed. On a Unix system, all lines end with just a linefeed. Normal unencrypted ASCII text messages are often automatically translated to some common canonical form when they are transmitted from one machine to another. Canonical text has a carriage return and a linefeed at the end of each line of text. But encrypted text cannot be automatically converted by a communication protocol, because the plaintext is hidden by encipherment. To remedy this inconvenience, PGP lets you specify that the plaintext should be treated as ASCII text (not binary data) and should be converted to canonical text form before it gets encrypted. At the receiving end, the decrypted plaintext is automatically converted back to whatever text form is appropriate for the local environment.“* [PGP 2.6.3i documentation, vol. II].

[27] *„After creating the signature certificate file, send it along with the original text file to the recipient. The recipient must have both files to check the signature integrity. When the recipient attempts to process the signature file, PGP notices that there is no text in the same*

To encrypt with conventional encryption only[28]:

```
pgp -c textfile
```

To produce a ciphertext file or key in ASCII radix-64 format (suitable for transporting it through E-mail channels), just add the -a option when encrypting or signing a message or extracting a key – Output files are named with the `.asc` extension:

```
pgp -sea textfile her_userid
pgp -kxa userid keyfile [keyring]
```

The configuration parameter ARMOR is equivalent to the -a command line option. If you need to use the `-a` option a lot, you can just turn on the ARMOR flag in `config.txt`.

To decrypt or check a signature for a ciphertext (.pgp) file:

```
pgp ciphertextfile.pgp [-o plaintextfile]
```
[29]

To view the decrypted plaintext output on your screen (like the Unix-style `more` command), without writing it to a file, use the `-m` (more) option while decrypting:

```
pgp -m ciphertextfile.pgp
```

To specify that the recipient's decrypted plaintext will be shown only on her screen and cannot be saved to disk, add the more option while encrypting:

```
pgp -em textfile her_userid
```

To wipe out a file (or the plaintext file after producing the cipher text file), add the `-w` (wipe out) option (while enrypting or signing a message):

```
pgp -w textfile
pgp -esw textfile her_userid
```

To use a Unix-style filter mode, reading from standard input and writing to standard output, add the `-f` option. For example:

```
pgp -fes her_userid <inputfile >outputfile
```

9.5.4 Weiterführendes zu PGP

Wir verweisen wieder auf die ausführlichen Dokumentationen zu PGP 2.6.3i.

---

file with the signature and prompts the user for the filename of the text. Only then can PGP properly check the signature integrity" [PGP 2.6.3i documentation, vol. II].

[28] *„Sometimes you just need to encrypt a file the old-fashioned way, with conventional single-key cryptography. This approach is useful for protecting archive files that will be stored but will not be sent to anyone else. Since the same person that encrypted the file will also decrypt the file, public key cryptography is not really necessary. (...)*
A pass phrase [is needed] as a conventional key to encipher the file. This pass phrase SHOULD not be the same pass phrase that you use to protect your own secret key. (...) PGP will not encrypt the same plaintext the same way twice, even if you used the same pass phrase every time" [PGP 2.6.3i documentation, vol. I].

[29] Default bei fehlender Option `-o` ist der Dateiname ohne Suffix `.pgp` bzw. `.asc`.

9.6 Ein kleines Beispiel

In diesem Abschnitt wollen wir nun PGP anwenden. Dazu dient die schon oft verwendete Situation:

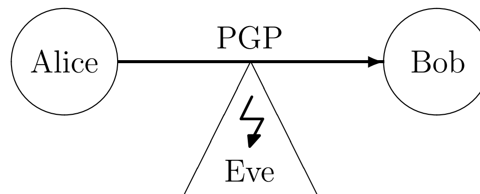

Abbildung 6: Unser kleines PGP-Szenario

Alice will Bob mittels PGP eine Mitteilung schicken. Alice ist schon PGP-Benutzerin.
Bob bislang noch nicht. Er muss deshalb zunächst ein Schlüsselpaar erzeugen
`BOB@linux:> pgp -kg`
Nachdem er sein Schlüsselpaar getestet hat (Test der Ver- und Entschlüsselung), schickt er seinen öffentlichen Schlüssel (armored) via E-M@il an Alice
`BOB@linux:> pgp -kxaf BOB .pgp/pubring.pgp | mail -s 'BOBsKEY' ALICE`
Alice erhält die E-M@il

```
Date: Sat, 5 Jun 2004 17:56:03 +0200
From: BOB <BOB@linux.local>
To: ALICE@linux.local
Subject: BOBsKEY

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.3ia

mQBtA0DLiVUAAAEDA0EvuH0ojAqlGQ5JhN34azRhLsHNawX9JSEIFuRt45caNhgQ
4DFyDBlm8GhuSLvfd7uFcom8qbKx9KMoR8IgV4Os58uXUeTAC4Bk3C5mUPFHTXGX
Di6iibF3PCzzNIsGUQAFEbQDQk9CiQB1AwUQQMuJVXc8LPM0iwZRAQFUUAMAvHC8
7bITUZF2pqnT2TTz/APc7r1v9mhXRGewZbguThgKmDKcze4gN12WlifxMKTZK7ol
g32v/UWQAd0mPkCGANk04OILEUyf9cAUe6HZktQYwwduUCzfh6bg3/fQEuoB
=emVe
-----END PGP PUBLIC KEY BLOCK-----
```

und speichert deren Inhalt in die Datei `BOBsKEY.txt`. Bislang ist Bobs öffentlicher Schlüssel noch nicht in Alice öffentlichem Schlüsselbund

```
ALICE@linux:> pgp -kv .pgp/pubring.pgp
Key ring: '.pgp/pubring.pgp'
Type Bits/KeyID    Date       User ID
pub  1024/F331C029 2004/06/05 ALICE
```

Nun fügt sie Bob hinzu

```
ALICE@linux:> pgp -ka BOBsKEY.txt .pgp/pubring.pgp
Looking for new keys...
pub 768/348B0651 2004/06/05 BOB

Checking signatures...
pub 768/348B0651 2004/06/05 BOB
sig!    348B0651 2004/06/05 BOB

Keyfile contains:    1 new key(s)

One or more of the new keys are not fully certified.
Do you want to certify any of these keys yourself (y/N)? n
```

Nun ist Bob in Alice öffentlichem Schlüsselbund aufgenommen

```
ALICE@linux:> pgp -kvv
Key ring: '.pgp/pubring.pgp'
Type Bits/KeyID    Date       User ID
pub   768/348B0651 2004/06/05 BOB
sig       348B0651             BOB
pub  1024/F331C029 2004/06/05 ALICE
sig       F331C029             ALICE
```

Alice möchte nun ihre Nachricht

```
ALICE@linux:> cat testfile.txt
* * * * *
geheime nachricht von ALICE an BOB
* * * * *
```

via E-M@il an Bob schicken (verschlüsselt und signiert)

```
ALICE@linux:> pgp -esa testfile.txt BOB
A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key.
Key for user ID: ALICE
1024-bit key, key ID F331C029, created 2004/06/05

Enter pass phrase: Pass phrase is good. Just a moment....

Recipients' public key(s) will be used to encrypt.
Key for user ID: BOB
768-bit key, key ID 348B0651, created 2004/06/05

WARNING: Because this public key is not certified with a trusted
signature, it is not known with high confidence that this public key
actually belongs to: 'BOB'.

Are you sure you want to use this public key (y/N)? n
```

PGP warnt Alice also, dass Bobs Schlüssel nicht zertifiziert ist.

Um sich von der Echtheit von Bobs Schlüssel zu überzeugen, erzeugt sie dessen fingerprint

```
ALICE@linux:> pgp -kvc BOB .pgp/pubring.pgp
Key ring: '.pgp/pubring.pgp', looking for user ID 'BOB'.
Type Bits/KeyID    Date       User ID
pub   768/348B0651 2004/06/05 BOB
Key fingerprint = FD 9D DB 6B 40 77 8E 25 03 4C B6 71 FD CA 2E 1A
```

und vergleicht diesen telefonisch mit Bob.
Da die fingerprints übereinstimmen, signiert Alice Bobs Schlüssel

```
ALICE@linux:> pgp -ks BOB .pgp/pubring.pgp -u ALICE
Looking for key for user 'BOB':

Key for user ID: BOB
768-bit key, key ID 348B0651, created 2004/06/05
Key fingerprint = FD 9D DB 6B 40 77 8E 25 03 4C B6 71 FD CA 2E 1A

READ CAREFULLY: Based on your own direct first-hand knowledge, are
you absolutely certain that you are prepared to solemnly certify that
the above public key actually belongs to the user specified by the
above user ID (y/N)? y

You need a pass phrase to unlock your RSA secret key.
Key for user ID: ALICE
1024-bit key, key ID F331C029, created 2004/06/05

Enter pass phrase: Pass phrase is good. Just a moment....
Key signature certificate added.

Make a determination in your own mind whether this key actually
belongs to the person whom you think it belongs to, based on available
evidence. If you think it does, then based on your estimate of
that person's integrity and competence in key management, answer
the following question:
Would you trust 'BOB' to act as an introducer and certify other people's
public keys to you?
(1=I don't know. 2=No. 3=Usually. 4=Yes, always.) ? 4
```

Damit hat Alice Bobs Schlüssel zertifiziert/signiert

```
ALICE@linux:> pgp -kvv
Key ring: '.pgp/pubring.pgp'
Type Bits/KeyID    Date       User ID
pub   768/348B0651 2004/06/05 BOB
sig       F331C029             ALICE
sig       348B0651             BOB
pub  1024/F331C029 2004/06/05 ALICE
sig       F331C029             ALICE
```

Gleichzeitig hat sie ihr Web-of-Trust erweitert

```
ALICE@linux:> pgp -km -u ALICE
Pass 1: Looking for the 'ultimately-trusted' keys...
* F331C029 ALICE

Pass 2: Tracing signature chains...
* ALICE
  > ALICE
  > BOB
    > BOB
KeyID      Trust     Validity    User ID
  348B0651 complete  complete    BOB
c          ultimate              ALICE
c          complete              BOB
* F331C029 ultimate  complete    ALICE
c          ultimate              ALICE
```

Nun kann Alice ihre Nachricht an Bob signieren, verschlüsseln

```
ALICE@linux:> pgp -esa testfile.txt BOB -u ALICE
A secret key is required to make a signature.
You need a pass phrase to unlock your RSA secret key.
Key for user ID: ALICE
1024-bit key, key ID F331C029, created 2004/06/05

Enter pass phrase: Pass phrase is good. Just a moment....

Recipients' public key(s) will be used to encrypt.
Key for user ID: BOB
768-bit key, key ID 348B0651, created 2004/06/05

Transport armor file: testfile.txt.asc
```

und verschicken

```
ALICE@linux:> mail -s 'ALICEsMESSAGE' BOB <testfile.txt.asc
```

Bob speichert die E-M@il von Alice in `ALICEsMESSAGE.asc` und entschlüsselt

```
BOB@linux:> pgp ALICEsMESSAGE.asc -u BOB
File is encrypted. Secret key is required to read it.
Key for user ID: BOB
768-bit key, key ID 348B0651, created 2004/06/05

You need a pass phrase to unlock your RSA secret key.
Enter pass phrase: Pass phrase is good. Just a moment......
File has signature. Public key is required to check signature.
Good signature from user 'ALICE'.
Signature made 2004/06/05 19:34 GMT using 1024-bit key, key ID F331C029

Plaintext filename: ALICEsMESSAGE
```

```
BOB@linux:> cat ALICEsMESSAGE
* * * * *
geheime nachricht von ALICE an BOB
* * * * *
```
Bob hat die Nachricht also sicher erhalten.
Nun möchte **Alice** Bob noch seinen von ihr signierten Schlüssel zur Verfügung stellen. Da sie sich beide morgen persönlich treffen werden, speichert sie ihren öffentlichen Schlüssel (`ALICEsKEY`)
```
ALICE@linux:> pgp -kxa ALICE ALICEsKEY >/dev/null 2>&1
```
und den von ihr signierten Schlüssel von Bob (`BOBsKEYsigned`)
```
ALICE@linux:> pgp -kxa BOB BOBsKEYsigned >/dev/null 2>&1
```
auf eine Diskette, die Bob erhalten wird.
Bob kann am nächsten Tag nicht nur Alice Schlüssel in seinen öffentlichen Schlüsselbund aufnehmen und gleich signieren, sondern auch Alice Signatur seines Schlüssels eintragen
```
BOB@linux:> pgp -ka BOBsKEYsigned .pgp/pubring.pgp -u BOB
Looking for new keys...
New signature from ALICE
on userid 'BOB'

Checking signatures...
pub 768/348B0651 2004/06/12 BOB
sig!    F331C029 2004/06/13 ALICE

Keyfile contains:
1 new signatures(s)
```

9.7 Abschließende Bemerkungen zu PGP

Die erste Version von **PGP** begann *Philip Zimmermann* unter dem Eindruck des US Senate Bill 266 zu schreiben, „*der 1991 im amerikanischen Senat verhandelt wurde und folgende Klausel enthielt: Der Kongress hält es für erforderlich, dass die Anbieter elektronischer Kommunikationsdienste (...) garantieren, ihre Kommunikationssysteme so auszustatten, dass die Klartextinhalte von (...) Mitteilungen den staatlichen Behörden zur Verfügung gestellt werden können, sofern hierzu eine gesetzliche Erlaubnis vorliegt*“ [Singh 1999, S. 363]. Zimmermanns Ansinnen war es, PGP 1.0 vor Inkrafttreten des Gesetztes so weit zu verbreiten, dass das Gesetz ins Leere laufen würde. Der Gesetzentwurf wurde niemals geltendes Recht.
Bereits ein Jahr später (1992) wurde ein deutlich erweitertes PGP 2.0 mit den Algorithmen RSA, IDEA und MD5 veröffentlicht. Ohne Zimmermanns Zutun verbreitete sich PGP um die ganze Welt.
1993 wurde es für Zimmermann unangenehm: Gegen ihn wurden Ermittlungen wegen des Vorwurfs des Patentmissbrauchs eingeleitet – da PGP das RSA-Verfahren verwendet, verletzt es die Lizenzrechte der Firma *Public Key Part-*

ners, die die alleinigen Rechte an der Vermarktung von RSA in den USA innehat[30]. In den USA war somit keine legale Anwendung (auch keine nicht-kommerzielle) ohne RSA-Lizenz möglich. Parallel dazu lief ein Verfahren zur Anklageerhebung gegen Zimmermann wegen illegalen Waffenexports. Weil die US-Regierung auch Krypto-Software zu den Rüstungsgütern zählte, durfte PGP nicht ohne staatliche Genehmigung exportiert werden – PGP wurde eine Frage der nationalen Sicherheit. Erst drei Jahre später (1996) wurden diese Verfahren eingestellt.
1994 entstand mit PGP 2.5 eine Version, die für den nicht-kommerziellen Gebrauch durch Verwendung von RSAREF in den USA legal eingesetzt werden konnte[31]. Außerhalb der USA war PGP noch nicht legal anwendbar, da die RSAREF-Lizenz eine Nutzung außerhalb der USA explizit verbot. Nur wenige später wurden PGP 2.6.2, die internationale Version 2.6.2i und schließlich die letzte rein kommandozeilen-orientierte PGP-Version 2.6.3i veröffentlicht. Die internationalen Versionen verwendeten eine Weiterentwicklung der RSA-Bibliothek, die Zimmermann ursprünglich für PGP entwickelt hatte und umgingen somit RSAREF.
In den USA vertrieb die Firma *ViaCrypt* eine kommerzielle Version von PGP unter den Versionsnummern 2.7, 4.0 und 4.5.
Nach der vollständigen Rehabilitation Zimmermanns 1996 gründete er die Firma *PGP, Inc.* und kaufte *ViaCrypt* auf. 1997 erschien PGP 5.0 – die erste Programmversion mit grafischer Benutzeroberfläche. PGP 5.0 bot erstmals auch andere Verfahren der konventionellen Verschlüsselung, Public-Key und der Signatur an. Die Versionen ab PGP 5.x sind aber nicht vollständig abwärtskompatibel zu den Vorgängerversionen. Der gesamte Quellcode von PGP 5.0 wurde in Buchform publiziert, da Bücher im Gegensatz zu Software legal aus den USA exportiert werden durften. Diese mehrere tausend Seiten umfassenden Werke wurden in einer speziellen Schrift gedruckt, die das spätere Einscannen vereinfachen sollte. Zudem waren auch Prüfsummen abgedruckt, die eine automatisierte Fehlerkorrektur der eingescannten Seiten ermöglichte. Damit gab es legale PGP-Versionen auch außerhalb der USA.
Im Jahre 1999 erhielt die Firma *NAI/McAfee*, die in der Zwischenzeit *PGP, Inc.* übernommen hatte, von den US-Behörden eine Export-Lizenz für PGP. Damit war der umständliche Export des Quellcodes in Buchform nicht mehr erforderlich. Zimmermanns Einfluss auf die weitere Entwicklung schwand unter *NAI* immer mehr. Gleichzeitig veröffentlichte *NAI* immer neuere Versionen

[30] In den USA und Europa ist außerdem der IDEA Algorithmus patentiert. Somit ist außerhalb der USA PGP nur für nicht-kommerzielle Zwecke kostenlos verwendbar. Für kommerzielle Anwendungen dagegen wird eine IDEA-Lizenz benötigt.

[31] „*RSAREF is a software library that implements the RSA cryptography routines. RSAREF is freeware, and is released by the patent holder of the RSA algorithm in the US. Everyone in the US who wants to make use of RSA in their programs and give it away for free (e.g. PGP), must use RSAREF. Since the RSA patent does not exist outside the USA and Canada, it seems reasonable to not encumber European users with the RSAREF subroutine library and its own additional copyright restrictions. In general, "international" means "non-US", i.e. it may be used by anyone except those who live in the US.*"

von PGP mit immer neuen Funktionalitäten. Auf Grund dessen wurde immer häufiger Kritik sowohl an PGP als auch an *NAI* laut – viele PGP-User weigerten sich wegen mangelnden Vertrauens, immer neuere Programmversionen zu installieren. Es erschienen die Versionen 6.0.x und 6.5.x – die letzten PGP-Versionen, die eine Vielzahl von verschiedenen Funktionalitäten in der Freeware-Version enthalten.
2000/2001 stellte *NAI* PGP 7.0 für Windows vor, welches aber nicht mehr als Source-Code, sondern nur als Binärversion herausgegeben wurde. Der PGP-Quellcode wurde und wird seitdem unter Verschluß gehalten.
Im Jahr 2002 stellte *NAI* die Produktion von PGP ein. Wiederbelebt wurde PGP von *PGP Corp.* mit der aktuellsten Windows-Version 8.0[32].
Heute ist die Idee von PGP mit **OpenPGP** (RFC 2440[33]) standardisiert, was unabhängige Implementierungen ermöglicht. Während PGP 6.x/7.x nicht hundertprozentig OpenPGP-kompatibel sind, ist es das 1999 veröffentlichte **GnuPG** 1.0.0 (**GNU Privacy Guard**). Das kommandozeilen-orientierte GnuPG wurde völlig neu programmiert, verwendet keine patentierten Algorithmen und erzeugt OpenPGP kompatible Dateien. IDEA steht nur über Plugins (erhältlich unter `http://www.gnupg.org/`) zur Verfügung. Mit dem Auslaufen des RSA-Patents wird RSA ab GnuPG 1.0.3 unterstützt. Damit ist GnuPG sogar PGP 2.6.3i kompatibel. Mit GnuPG steht erstmals eine PGP kompatible freie und rechtlich einwandfreie Implementierung zur Verfügung.

```
jk@linux:> gpg --version
gpg (GnuPG) 1.2.2
Copyright (C) 2003 Free Software Foundation, Inc.

Home:  /.gnupg
Unterstützte Verfahren:
Öff.Schlüssel: RSA, RSA-E, RSA-S, ELG-E, DSA, ELG
Verschlü.: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256
Komprimierung: Uncompressed, ZIP, ZLIB
```

Mit **Kgpg** steht zudem eine graphische Oberfläche von GnuPG zur Verfügung:

```
jk@linux:> kgpg --version
Kgpg - Einfache graphische Oberfläche für gpg

KDE: 3.1.4
kgpg: 1.0.0
```

[32]Zum Download unter `http://www.pgp.com/`.

[33]Die RFCs (Request for Comment) sind eine Sammlung der Definitionen und technischen Spezifikationen, nach denen unter anderem das Internet ablaufen – zu finden unter: `http://www.rfc-editor.org/`

10 Quellen

Literatur:

Beutelspacher, Albrecht:
Kryptologie. Braunschweig/Wiesbaden: Vieweg [6]2002

Beutelspacher, Albrecht; **Schwenk**, Jörg; **Wolfenstetter**, Klaus-Dieter:
Moderne Verfahren der Kryptographie. Braunschweig/Wiesbaden: Vieweg [3]1999

Salomaa, Arto:
Public-Key cryptography. New York/Berlin/Heidelberg: Springer 1990

Schneier, Bruce:
Applied cryptography. John Wiley & Sons, Inc. 1996

Singh, Simon:
Geheime Botschaften. München/Wien: Hanser 1999

Dokumentationen:

Zimmermann, Philip:
PGP(tm) User's Guide, Volume I: Essential Topics. 11 October 1994

Zimmermann, Philip:
PGP(tm) User's Guide, Volume II: Special Topics. 11 October 1994

Web-Quellen:

http://www.bsi.bund.de/

http://www.gnupg.org/

http://www.pgp.net/

http://www.pgpi.org/

http://www.rfc-editor.org/