Kapitel 3
Grundlagen der Kryptographie

1. Einführung
2. Einfache Symmetrische Verfahren
3. Moderne symmetrische Verfahren
4. Asymmetrische Verfahren („Public Key“)
5. Digitale Signaturen und kryptographische Hashfunktionen

Was ist Kryptographie?

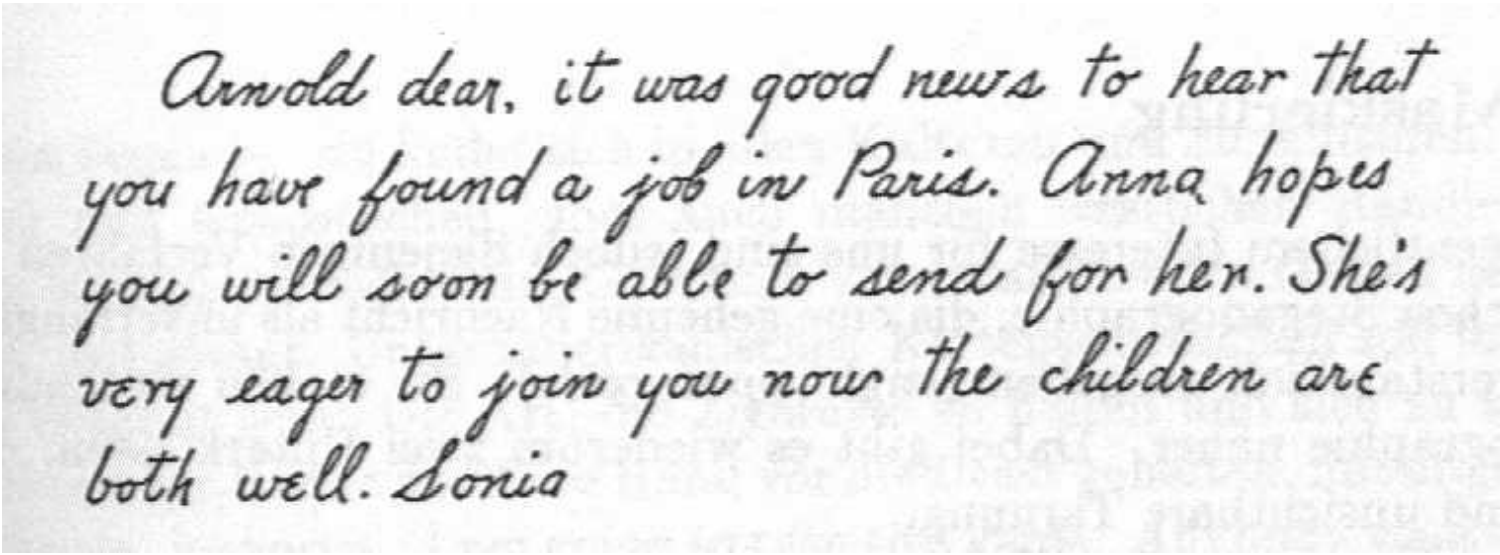
- **Ziele**
  - Geheimhaltung (Vertraulichkeit)
  - Schutz vor Veränderung (Integrität)

  von Nachrichten (Daten)

- **Techniken**
  - Verheimlichen der Existenz einer (geheimen) Nachricht: "Steganographie" (gedeckte Geheimschriften)
  - Verschlüsseln / Entschlüsseln von Nachrichten: "**Kryptographie**" (offene Geheimschriften)

Linguistische Steganografie

- Die Nachricht wird offen übertragen, ist aber nicht als geheim erkennbar (Open Code)
  - Jargon Codes (z.B. bei Kriminiellen: „Koks“, „Schnee“)
  - Vereinbarte Stichworte (z.B. in Radiosendungen im 2. Weltkrieg)
  - Mimik und Gestik (z.B. beim Kartenspiel)
- Die geheime Nachricht kann auch in grafischen Details einer Schrift oder Zeichnung enthalten sein (Semagramm)
  - Beispiel: Zahlencode = Anzahl der Buchstaben, die einem Aufschwung vorangehen (3 3 5 1 5 1 4 1 2 3 …)



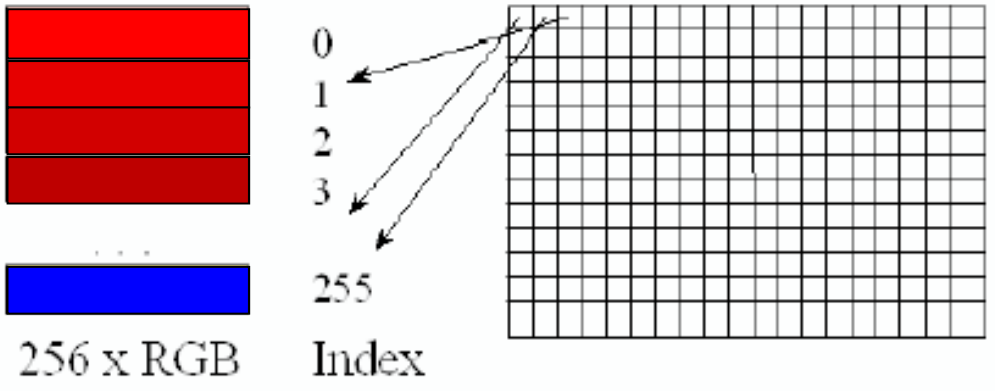
Technische Steganographie

- Geheimtinten: Zitronensaft, Milch, …
- Verstecke: Doppelte Böden, hohle Absätze, Geheimfächer, …
- Microdots: Mikrofotografie von Ø=0,5 mm verbirgt 1 DIN A4-Seite
- Heute: Verstecken von geheimen Informationen in
  - digitalen Bildern
  - Musikdateien

Steganographie in digitalen Bildern

- Repräsentation der Bildinformation durch einzelne Bildpunkte („Pixel“)
- Pro Pixel werden 1 – 32 Bit für Farbinformationen gespeichert

Beispiel: Farbpalette mit 8 Bit pro Pixel



Idee: Die Veränderung eines einzelnen Pixel-Bits ist optisch nicht zu erkennen, wenn eine benachbarte Farbe gewählt wird
(z.B. bei 24 Bit pro Pixel = 2^{24} Farben)

➔ Eine geheime Nachricht kann im Bild versteckt werden, indem z.B. das niederwertigste Bit jedes Pixels entsprechend umcodiert wird!

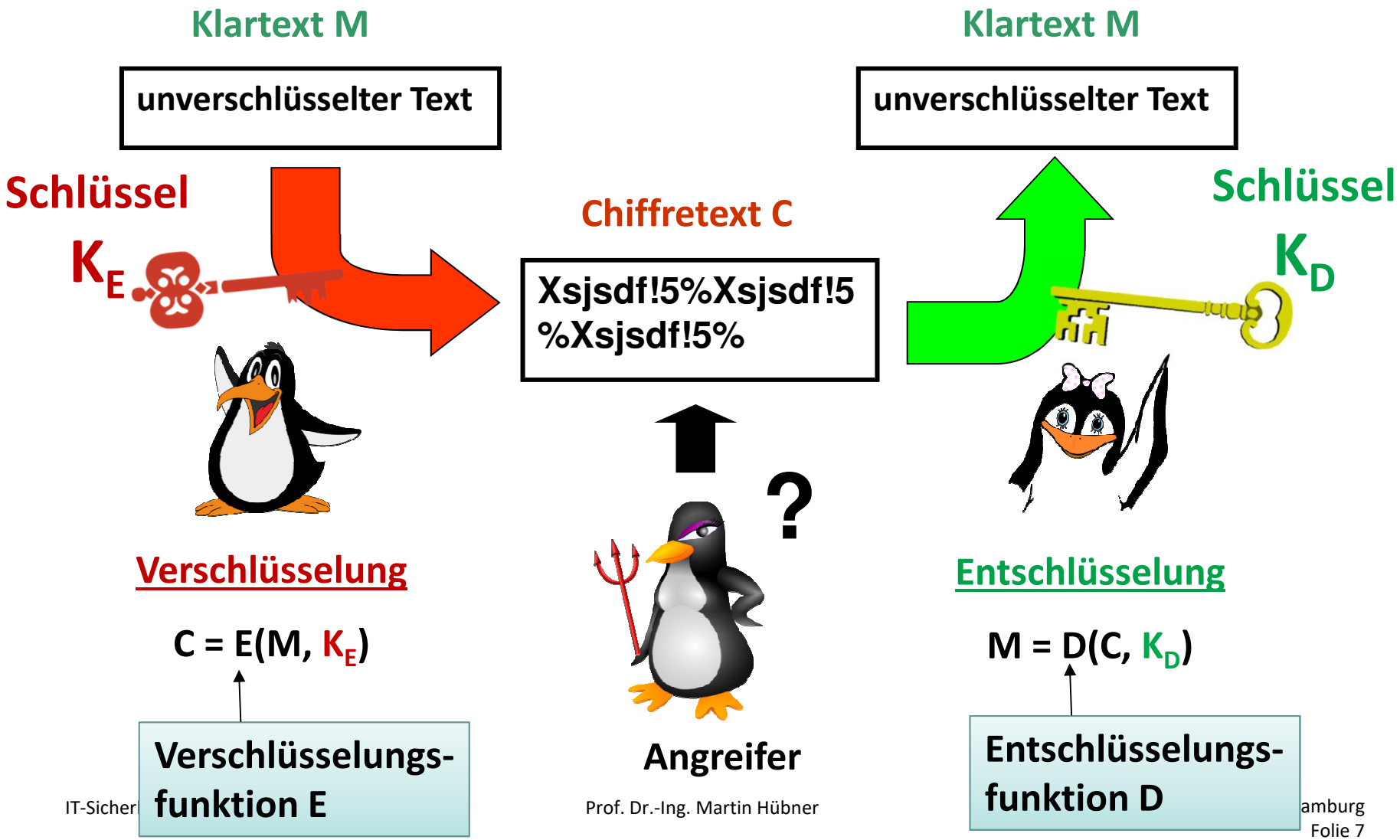
(Voraussetzung: Verlustfreie Komprimierung!)

Steganografie in digitalen Bildern

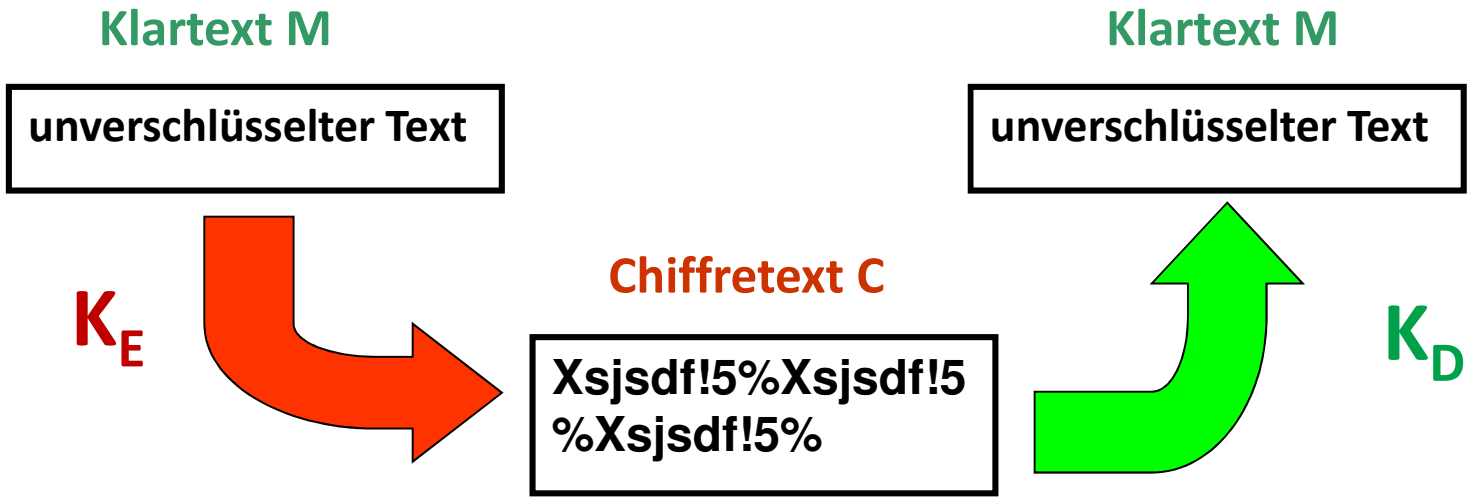
In einem der beiden Bilder ist der Text der kompletten Bibel enthalten!

(→ Tool und Bilddateien liegen im Pub-Verzeichnis)

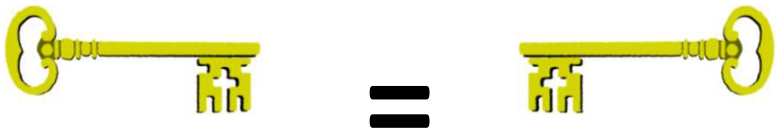
Prinzip der Verschlüsselung



Klassifikation von Verschlüsselungsverfahren

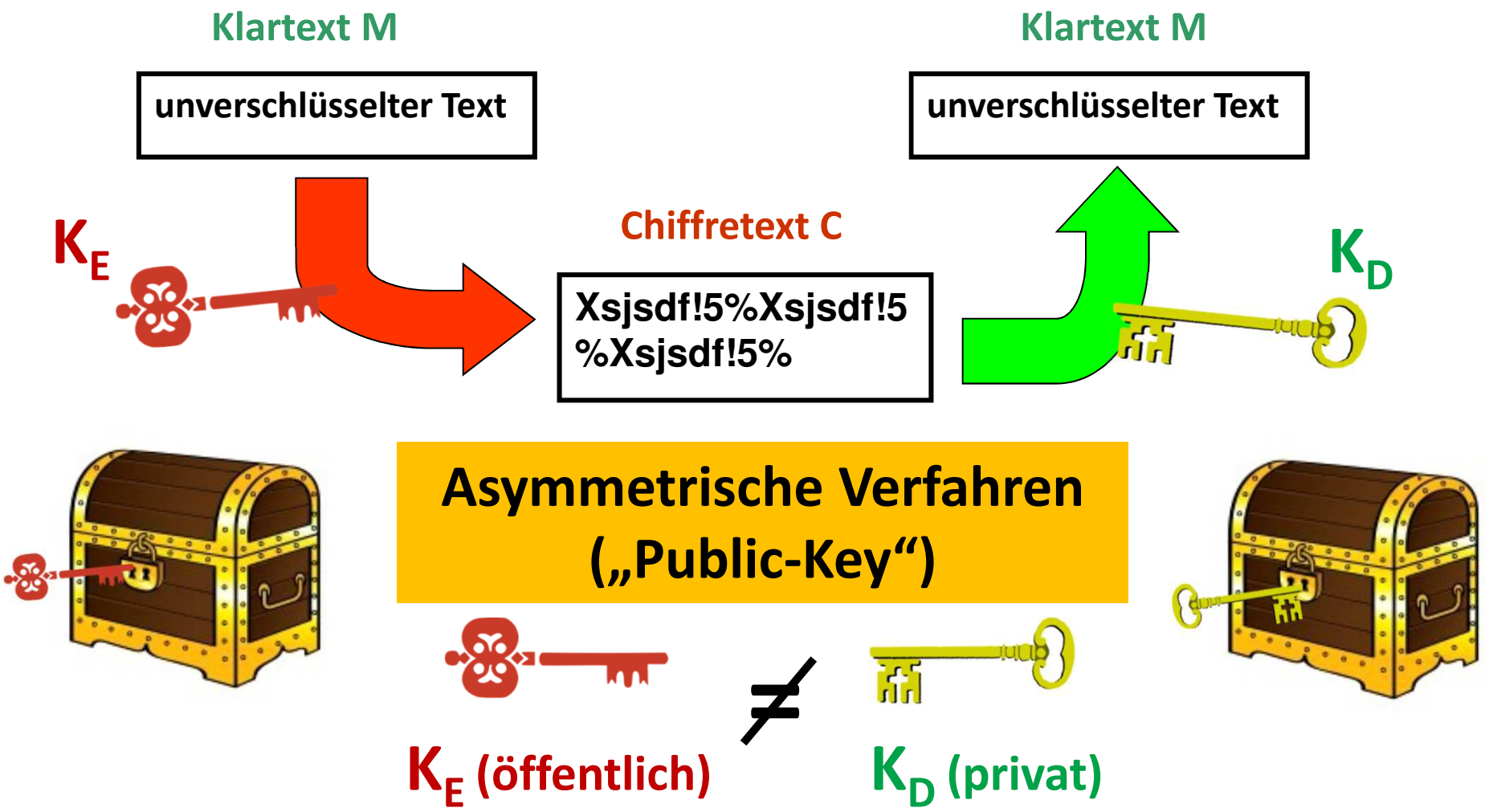


Symmetrische Verfahren („Secret Key“)



K_E (geheim) = K_D (geheim)

Klassifikation von Verschlüsselungsverfahren



Klassifikation kryptographischer Verfahren

- Symmetrische Verfahren („Secret-Key-Verfahren“)
  - Beide Schlüssel (für Ver- und Entschlüsselung) sind gleich: $K_E = K_D$
  - Beide Kommunikationspartner müssen den gemeinsamen Schlüssel kennen (vorher geheimer Austausch nötig!)
  - „Schnelle“ Verfahren
- Asymmetrische Verfahren („Public-Key-Verfahren“)
  - Erzeugung von Schlüsselpaaren (K_E, K_D)
    - Schlüssel K_E zur Verschlüsselung sind öffentlich bekannt
    - Schlüssel K_D zur Entschlüsselung sind geheim („privat“)
  - Kein Austausch nötig
  - „Langsame“ Verfahren

Angriffe auf kryptographische Verfahren

- Ciphertext-Only
  - Dem Angreifer ist nur Chiffretext (Ciphertext) bekannt
  - Ziel: zugehörigen Klartext ermitteln
- Known-Plaintext
  - Der Angreifer kennt mehrere Klartext (Plaintext) / Chiffretext-Paare
  - Ziel: Algorithmus und Schlüssel K_D ermitteln
- Chosen-Plaintext
  - Der Angreifer kann den Klartext vorgeben und erhält dazu den Chiffretext
  - Ziel: Algorithmus und Schlüssel K_D ermitteln

Technik: **Kryptoanalyse** (Statistik, Ausprobieren, …)

Angriffe auf Verschlüsselungsverfahren

Kerckhoff-Prinzip:

Die Sicherheit eines Systems darf nicht von der Geheimhaltung der Ver- und Entschlüsselungsfunktion abhängen,

sondern nur von der ***Geheimhaltung des Schlüssels****!*

Annahme: Ein Angreifer kennt alles außer dem Schlüssel!

Kerckhoff-Prinzip - Warum?

- Geheimhaltung kurzer Schlüssel ist einfacher
- Auswechseln der Schlüssel ist einfacher als Austausch des Algorithmus
- Gruppenkommunikation ist sicherer
  - Jeder benutzt gleiche Algorithmen aber verschiedene Schlüssel
- Sicherheit der Algorithmen kann von Experten analysiert und überprüft werden
- Algorithmen werden ohnehin irgendwann bekannt
  - Reverse Engineering
  - Interne Angreifer

Kapitel 3
Grundlagen der Kryptographie

1. Einführung
2. Einfache Symmetrische Verfahren
3. Moderne symmetrische Verfahren
4. Asymmetrische Verfahren („Public Key“)
5. Digitale Signaturen und kryptographische Hashfunktionen

Substitutionschiffren-Beispiel: „Caesar“-Chiffre

Algorithmus: Jeder Buchstabe wird im Alphabet um 3 Stellen nach rechts verschoben.

Schlüssel: K=3

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
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

Caesar sprach:

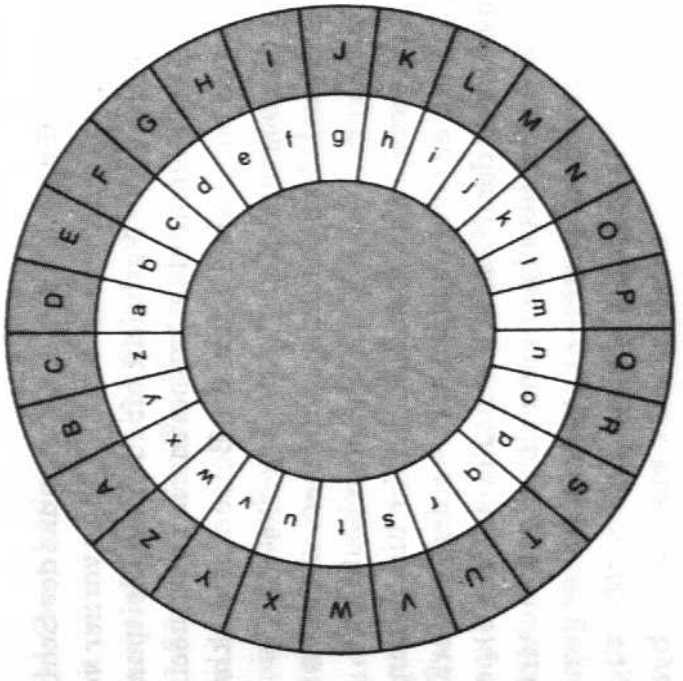
QRPHQ HVW RPHQ

???

Caesar meinte:

nomen est omen

Verallgemeinerung: Es gibt 25 sinnvolle Substitutionschiffren durch Verschiebung über dem Alphabet



„Caesar“-Chiffre: Verallgemeinerung

Klartext M: Buchstaben $m_1,\ldots, m_z$ *(Codezahlen)*

Schlüssel: K ϵ {1, .., 25}:

Chiffretext C: Buchstaben $c_1,\ldots, c_z$ *(Codezahlen)*

Algorithmus: Jeder Buchstabe wird um K Stellen im Alphabet nach rechts verschoben durch Addition von K zum Buchstaben-Code modulo 26

Code:
a = 0
b =1
.
.
.
z = 25

Verschlüsselungsfunktion:
$E(M, K) = (m_1+K,\ldots,m_z+K)$

Entschlüsselungsfunktion:
$D(C,K) = (c_1-K,\ldots,c_z-K)$

Nur 25 sinnvolle Schlüssel existieren („Schlüsselraum“)

➔ Chiffre durch Ausprobieren knackbar („Brute Force“)!!

Allgemeine monoalphabetische Substitutions-Chiffren

- **Algorithmus**: Jeder Buchstabe wird durch einen **beliebigen** anderen Buchstaben ersetzt
- Definition des Schlüssel über eine **Tabelle π**
- Beispiel:

π =

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

- **Klartext** = hitherehowareyou
- **Chiffretext** = π(h)…π(u)
  = YOGYULUYFCQLUWFH
- **Klartext** = π^{-1}(Y)…π^{-1}(H)
  = hitherehowareyou

Jeder Chiffretextbuchstabe repräsentiert immer denselben Klartextbuchstaben!

Allgemeine (monoalphabetische) Substitutions-Chiffren - Definition

Schlüsselraum K = $\{\pi_1, \ldots, \pi_n\}$ = alle möglichen Permutationstabellen von 26 Buchstaben.

Jede Permutationstabelle $\pi \in K$ ist ein möglicher Schlüssel

Verschlüsselungsfunktion:

$$\mathbf{E(M, \pi) = \pi(M)}$$

Entschlüsselungsfunktion:

$$\mathbf{D(C, \pi) = \pi^{-1}(M)}$$

wobei π^{-1} invers zu π ist.

Allgemeine (monoalphabetische) Substitutions-Chiffren - Schlüsselraum

π_i =

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

Wie viele solche Permutationen (Schlüssel) gibt es?

$$26! \approx 2^{88}$$

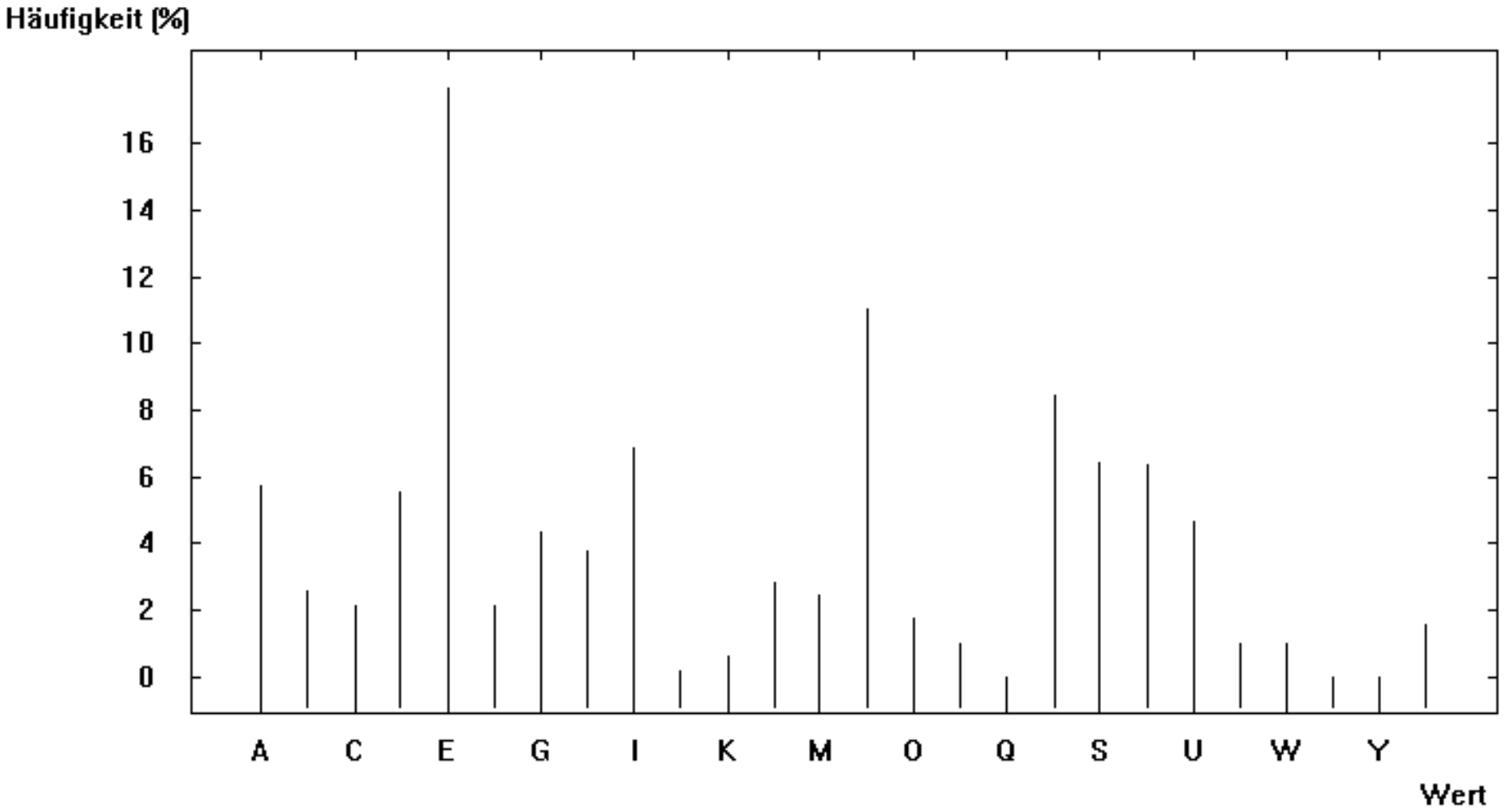
➔ Brute-force ist nicht möglich!

Ist das ausreichend für die Sicherheit?

Angriffstechnik: Häufigkeitsanalyse

Welcher Buchstabe kommt in **deutschsprachigen** Texten statistisch gesehen wie häufig vor?

→ Rückschluss vom Chiffretext auf Klartext möglich!



Permutations-Chiffren

Algorithmus: Die **Plätze** der Buchstaben nach einem geheimen Schema (Schlüssel) **vertauschen** („permutieren“)

Krebs: Rückwärtslesen

```
Retupmoc red. – Red retupmoc.
```

Würfel: Waagerechtes Einlesen und senkrechtes Auslesen

Gruppierung: je 5 Buchstaben

```
so einfach geht das
```

Schlüssel: Neue Positionen (4,1,2,5,3)

```
ISONEHFAGCDEHATS
```

```
so einfach
geht das
```

1	2	3	4
S	O	E	I
N	F	A	C
H	G	E	H
T	D	A	S

Spaltenreihen-folge: 2,1,4,3

```
OFGDSNHTICHSEAEA
```

Polyalphabetische Verfahren

- **Ziel**: Substitutions-Chiffren, die einen Angriff durch Häufigkeitsanalyse verhindern!
- Bei **polyalphabetischen** Verschlüsselungen wird **nicht** stets derselbe Klartextbuchstabe auf denselben Chiffretextbuchstaben abgebildet (*viele Alphabete*)
- **Algorithmus**: Verknüpfung jedes Klartextbuchstabens mit zusätzlichen Schlüsselinformationen
- Beispiel:
  - Schlüsselinformationen: Wort oder Text
  - Verknüpfung: Addition/Subtraktion der Buchstaben-Codes modulo 26 (Alphabet)

Vigenère-Chiffre - Definition

- **Algorithmus:** Verknüpfung jedes Klartextbuchstabens mit zusätzlichen Schlüsselinformationen (polyalphabetisch)
- **Klartext M:** Buchstaben m_1,…., m_z
- **Schlüsselinformationen**: **Schlüssel-Wort K** aus Buchstaben k_1, …,k_n (wiederholt, falls n < z)
- **Verknüpfung**: Addition/Subtraktion der Buchstaben-Codes modulo 26

Verschlüsselungsfunktion:

$$E(M, K) = (m_1+k_1,\ldots,m_n+k_n,m_{n+1}+k_1,\ldots)$$

Entschlüsselungsfunktion:

$$D(C,K) = (c_1-k_1,\ldots,c_n-k_n,c_{n+1}-k_1,\ldots)$$

Vigenère-Chiffre

- **Beispiel**:
  - **Schlüsselwort**: alice
  - **Klartext**: „Alles ist eine Folge von Bits“
  - **Chiffretext**:

Code:
a = 0
b =1
.
z = 25

```
  alles ist eine folge von bits
+ alice ali ceal iceal ice alic
  ----------------------------
  AWTGW IDB GMNP NQPGP DQR BTBU
```

➔ Problem:

Leicht knackbar, wenn die Schlüssellänge bekannt ist!

Angriffstechnik:
Ermittlung der Schlüssellänge

- Suche Buchstabenfolgen, die im Chiffretext doppelt vorkommen
  → wahrscheinlich dasselbe Klartextwort
- Bestimme den Abstand zwischen je zwei doppelten Buchstabenfolgen
  → alle Teiler sind mögliche Schlüssellängen
- Teste im weiteren alle Zahlen, die in allen doppelten Buchstabenfolgen mögliche Teiler sind (= mögliche Schlüssellängen)
- Jeder einzelne Buchstabe des Schlüsselworts erzeugt eine monalphabetische Cäsar-Chiffre (Periode = Schlüssellänge) und ist durch Häufigkeitsanalyse knackbar!
- Beispiel:

Wiederholte Folge	Abstand	Mögliche Schlüssellänge (Teiler)																	
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
E-F-I-Q	95				✓														✓	
P-S-D-L-P	5				✓															
W-C-X-Y-M	20	✓		✓	✓					✓										✓
E-T-R-L	120	✓	✓	✓	✓	✓		✓		✓		✓			✓					✓

Vernam-Chiffre

- **Klartext M:** Buchstaben m_1,…., m_z
- **Algorithmus:** wie bei Vigenère-Chiffre
- **Schlüsselinformationen**:
  Schlüssel-Text, der genauso lang ist wie der Klartext
- *Alle Probleme beseitigt?*

➔ **Rest-Problem** bleibt:
Die Verwendung einer natürlichen Sprache ermöglicht einem Angreifer immer noch einen Informationsgewinn durch Häufigkeitsanalyse, da Buchstabenfolgen Wörter mit spezieller Buchstabenhäufigkeit sind!

One-Time-Pad

- **Schlüsselinformationen**:
  Schlüssel-Text, der genauso lang ist wie der Klartext und eine rein **zufällige** Buchstabenfolge darstellt (daher auch nur ein einziges Mal verwendbar ist)!
- **Algorithmus:** wie bei Vernam/Vigenère-Chiffre
- Funktioniert auch mit **Bits** statt Buchstaben
  (→ Bitweise Addition ohne Übertrag = XOR-Verknüpfung)

- **Theoretisch ein absolut sicheres Verfahren!**
- **Nötig für die Anwendung** (seufz!):
  - Sicherer Austausch der (geheimen) Schlüsselinformationen
  - Erzeugung statistisch „reiner“ Zufallszahlen ohne Wiederholungen

Kapitel 3
Grundlagen der Kryptographie

1. Einführung
2. Einfache Symmetrische Verfahren
3. Moderne symmetrische Verfahren
4. Asymmetrische Verfahren („Public Key“)
5. Digitale Signaturen und kryptographische Hashfunktionen

Moderne Ansätze für symmetrische Verschlüsselungsverfahren

Stromchiffren

- **Idee:** Implementierung des **One-Time-Pad**-Verfahrens auf Bit-Basis mit Hilfe von Pseudo-Zufallszahlengeneratoren
- **Beispiele**: **RC4** (WEP), **A5** (GSM), **E_0** (Bluetooth)

Blockchiffren

- **Idee**: **Ersetzung** von (großen) Bit-Blöcken statt Buchstaben
  - Monoalphabetische Substitutionschiffren mit großem Alphabet (z.B. alle 2^{64} 64-Bitfolgen)
  - Die Verschlüsselungs-/Entschlüsselungsfunktion wird definiert durch speziellen Algorithmus mit Verarbeitung einer Schlüsselbitfolge (statt Tabelle)
  - Keine Häufigkeitsanalyse möglich, da praktisch keine Blockwiederholung im Chiffretext vorkommt
- **Beispiele**: **DES**, **3DES**, **AES** (alle für SSL/TLS u.a. nutzbar)

Exkurs: Zufallszahlen

- Wichtigste Anforderung an Folgen von Zufallszahlen (Bits): absolute **Unvorhersagbarkeit**!
- **Zufallszahlengeneratoren:**
  Erzeugen Folgen von Zufallszahlen
  - **Echter** Zufallszahlengenerator:
    - Verwendet „zufällige“ physikalische Ereignisse
      - Werfen einer Münze, Würfeln
      - Spannungsschwankungen an Widerständen oder Dioden
      - Zeit zwischen zwei Festplattenzugriffen, Mausbewegungen, Tastatureingaben, System-Statusinformationen
  - **Pseudo**-Zufallszahlengenerator:
    - Verwendet mathematischen Algorithmus
    - Pseudo-Zufallsfolgen sollten sich bei statistischen Tests wie 'echte' Zufallszahlen verhalten

Prinzip eines Pseudo-Zufallszahlengenerators

- Aus einem Startwert („seed“) wird mittels einer „Fortschaltfunktion“ eine Zufallszahl berechnet
- Diese dient dann ebenfalls als neue Eingabe für die Fortschaltfunktion

Folgerung:

- Gleicher Startwert → gleiche „Zufallszahlen“
- Der Startwert muss durch ein „echtes“ Zufallsereignis bestimmt werden!

Pseudo-Zufallszahlengenerator-Beispiel: Lineare Kongruenzmethode

- Weitverbreitete Methode zur Generierung von gleichverteilten Pseudo-Zufallszahlen
- a, b, m, x_0 ϵ N, m > 0 und x_0 ϵ {1, .., m-1}
- **Fortschaltfunktion** (lineare Rekursion):

$$\mathbf{x_0 = Startwert}$$

$$\mathbf{x_{k+1} = (a * x_k + b) \bmod m}$$

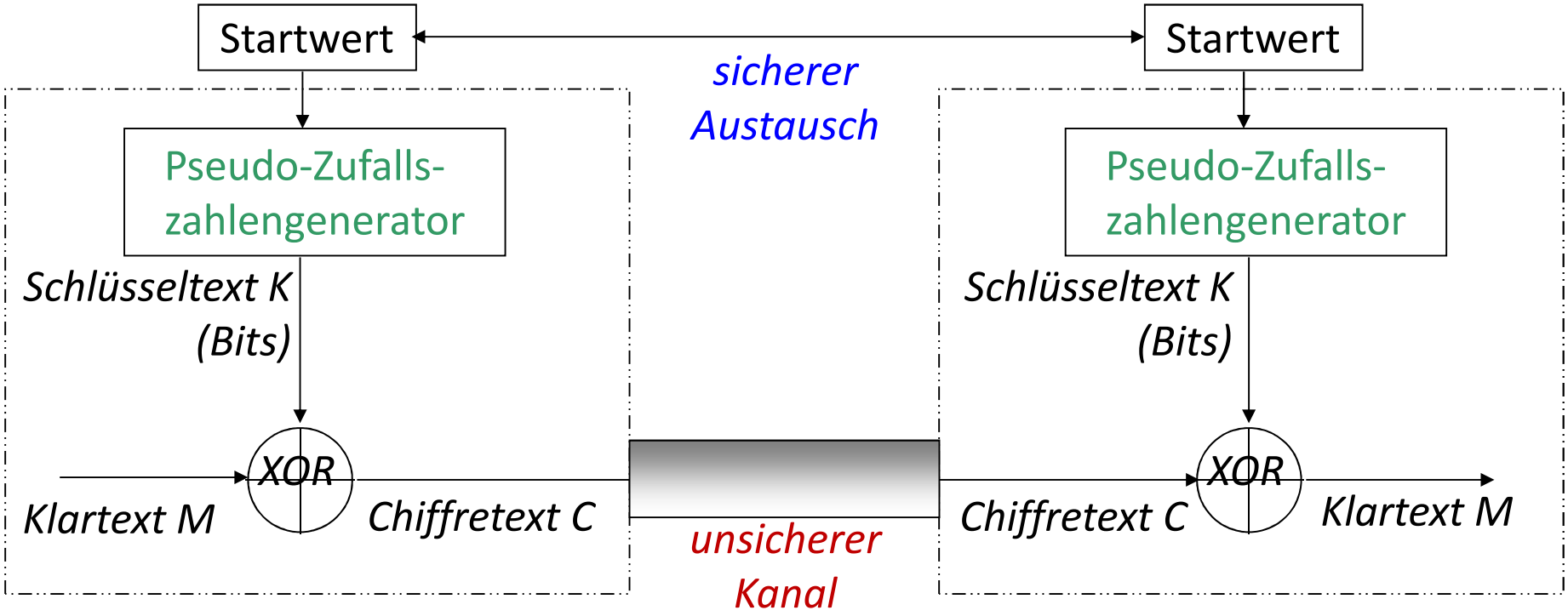
- Qualität ist abhängig von der Wahl der Parameter a und b (meist groß)
  - Analyse durch Zahlentheorie und statistische Tests
  - Ziel: **Periodisches Verhalten vermeiden** *(große „Streuung“)*

Kryptographische Anforderungen an Pseudo-Zufallszahlengeneratoren

- *Kryptografisch sichere Pseudozufallsgeneratoren* (*CSPRNG, „cryptographically secure pseudo random number generator“*) müssen eine spezielle Eigenschaft besitzen:
  Die Zufallszahlen sind *nicht vorhersagbar*!
- Es muss rechentechnisch unmöglich ein, aus den gegebenen n Bits eines Schlüsselstroms s_i, s_{i+1}, ... s_{i+n-1} die folgenden Bits s_{i+n} , s_{i+n+1} ... zu berechnen
- Die genauere Definition besagt, dass es keinen Algorithmus mit polynomieller Laufzeit gibt, der das Bit s_{i+n} mit einer Wahrscheinlichkeit von mehr als 50% plus einer vernachlässigbar kleinen Abweichung berechnen kann
- Die Lineare Kongruenzmethode ist daher nicht für kryptographische Zwecke geeignet!
  (Anwendungen: Simulation, Spiele)

Stromchiffren-Prinzip

Anwendung eines **Pseudo-Zufallszahlengenerators**, um einen kontinuierlichen, zufälligen Schlüsseltext („Strom“) zu erzeugen, der genauso lang ist wie der Klartext (→ **One-Time-Pad**)!



$E(M,K) = M \oplus K = C$

$D(C,K) = C \oplus K = M$

Stromchiffren: Sicherheit?

- Bei „zufälligem“ Startwert und „zufälligem“ Schlüsseltext: absolut sicheres Verfahren!
- Was passiert, wenn derselbe Startwert wiederverwendet wird?
  → dieselbe Schlüsselbitfolge K wird erzeugt!
  - Wenn ein Angreifer ein Chiffretext/Klartext-Paar (C_1,M_1) kennt, kann er die Schlüsselbitfolge K ermitteln (Known-Plaintext-Angriff):
    $C_1 = M_1 \oplus K$ ➔ $K = C_1 \oplus M_1$
  - Damit sind alle folgenden Nachrichten, die mit der Schlüsselbitfolge K verschlüsselt werden, für den Angreifer lesbar!
  - Möglicher Angriff: Anfrage an Server senden, deren Antwort bekannt ist (z.B. ICMP-Nachricht) und Leitung mit verschlüsselter Antwort abhören

RC4-Chiffre („Rivest Cipher No. 4“, 1987)

- **Stromchiffre** mit speziellem Pseudo-Zufallszahlengenerator
- Datenstrukturen:
  - array `int s[256]`, in dem jede Zahl zwischen 0 und 255 als Wert genau einmal vorkommt (Permutation)
  - `int i, j, k` Hilfsvariablen (Initialwert: 0)
- Startwert wird aus vorgegebener Schlüssel-Bytefolge (mindestens 5, maximal 256 Byte) berechnet → Initialisierung des Arrays s
- Fortschaltfunktion:

```
i = i + 1 (mod 256)
j = j + s[i] (mod 256)
Vertausche s[i] und s[j]
k = s[i] + s[j] (mod 256)
return s[k]
```

RC4 Bewertung

- Sehr schnelles Verfahren
- Mittlerweile für kryptographische Zwecke **nicht** mehr geeignet (mehrere gravierende Schwachstellen entdeckt → WEP!)
- Verbesserte Nachfolger-Version:
  Spritz (Rivest/Schuldt 2014)
- Spritz-Fortschaltfunktion:

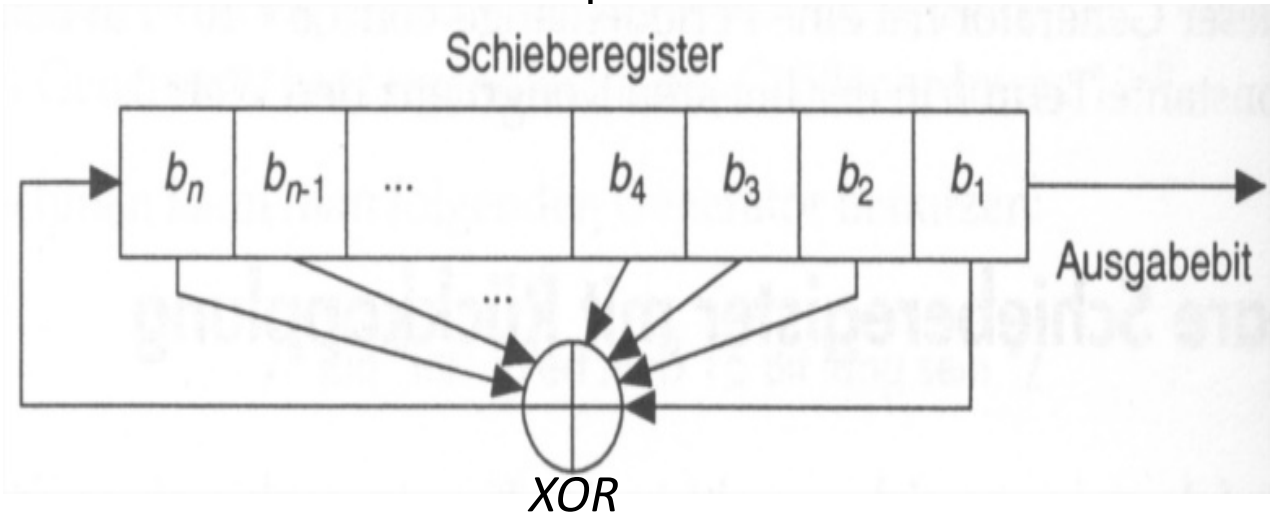
```
i := i + w (mod 256)   // w feste Primzahl
j := k + s[j + s[i]] (mod 256)
k := k + i + s[j] (mod 256)
Vertausche s[i] und s[j]
return z := s[j + s[i + s[z + k]]] (mod 256)
```

- Leider auch schon Schwachstellen in Spritz entdeckt (Banik/Isobe 2016)!

A5 (GSM) und E_0 (Bluetooth)

Stromchiffren mit speziellem Pseudo-Zufallszahlengenerator

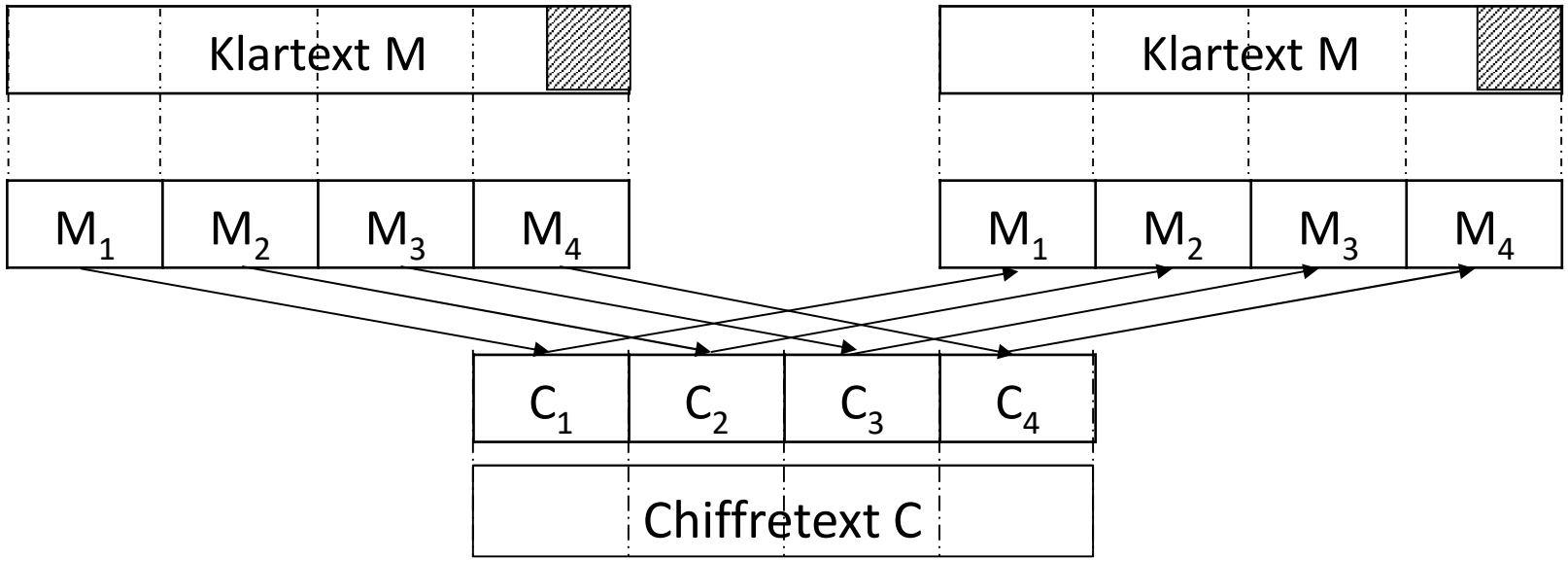
- Einsatz mehrerer **linearer Schieberegister** mit Rückkopplung als Pseudo-Zufallszahlengenerator
- Nur einige spezielle Bitstellen werden rückgekoppelt
- Sehr effizient in Hardware implementierbar!



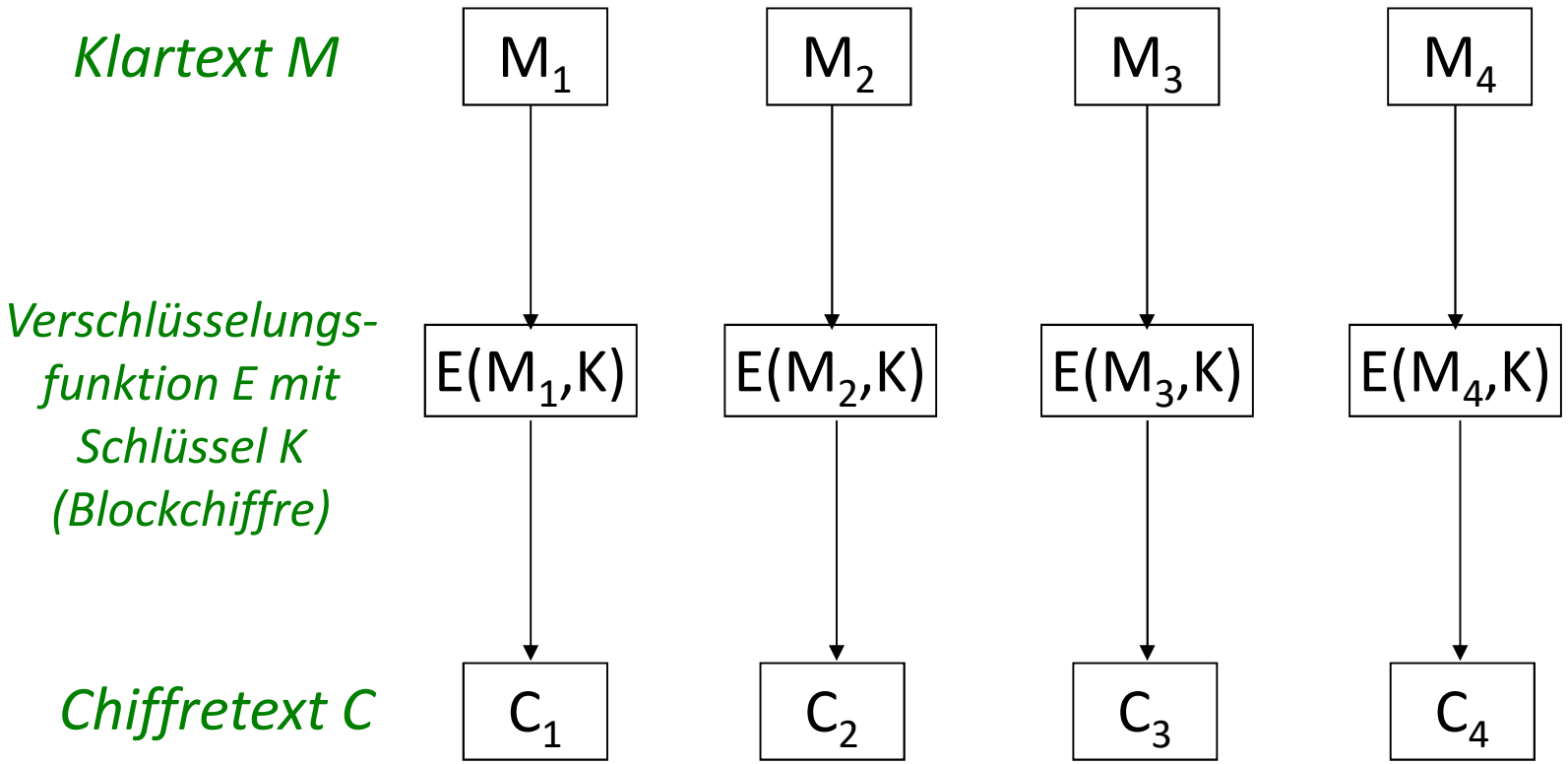
- A5: Gravierende Schwachstellen → unsicher
- E_0 : Schwachstellen zwar vorhanden, aber (noch) nicht praxisrelevant

Blockchiffren

- Blockchiffren teilen den Klartext in **Blöcke fester Länge** auf (z.B. 64 oder 128 Bit)
- Der letzte Klartextblock muss ggf. aufgefüllt werden
- Länge eines Chiffretextblocks = Länge eines Klartextblocks *(„Ersetzungs“-Verfahren)*



Standard: ECB (Electronic Code Book) - Modus



Verschlüsselung durch Verschlüsselungsfunktion E: $C_i = E(M_i,K)$

Entschlüsselung durch Entschlüsselungsfunktion D: $M_i = D(C_i,K)$

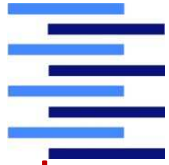
ECB-Modus: Bewertung

Welche Angriffe könnte ein „Man in the Middle“ im Rahmen einer verschlüsselten Kommunikationsverbindung unbemerkt durchführen?

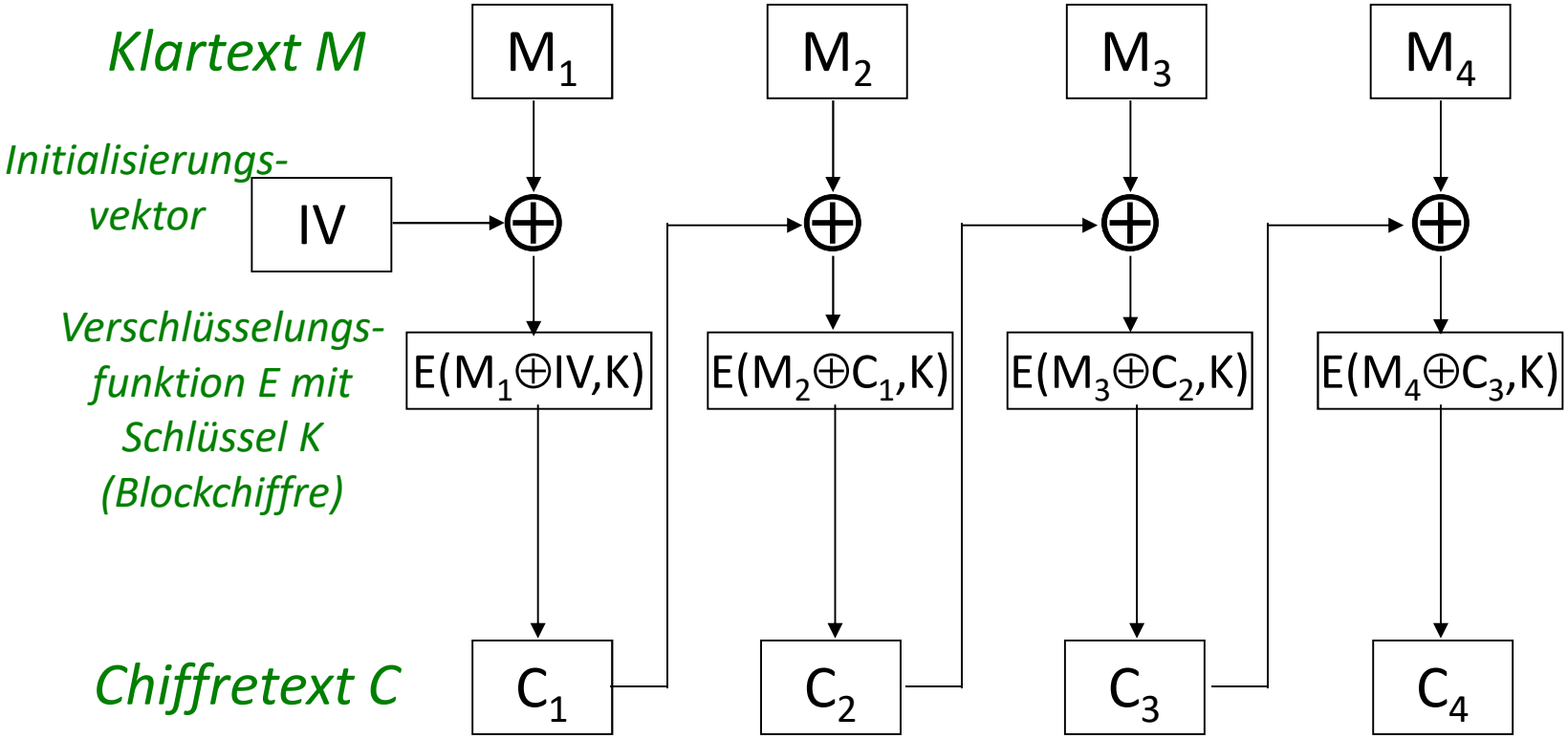
- Wiedereinspielen von Blöcken
- Vertauschen der Reihenfolge von Blöcken
- Entnahme von Blöcken

➔ **Verbesserungsvorschläge?**

CBC (Cipher Block Chaining) - Modus



Verknüpfen jedes Klartextblocks mit vorherigem Chiffretextblock

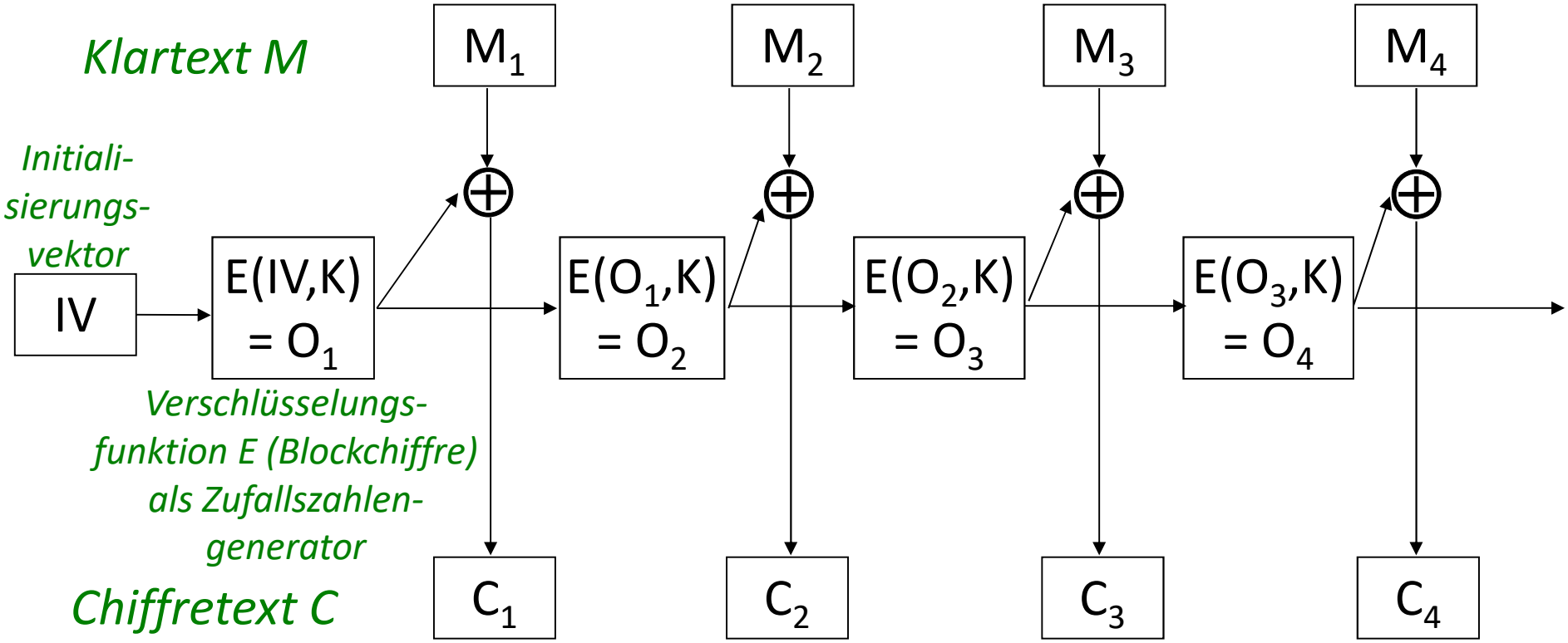


Berechnung eines Chiffretextblocks:

$C_i = E(M_i \oplus C_{i-1}, K)$ mit $C_0 = IV$

OFB (Output Feedback) - Modus

Die Blockchiffre wird als Zufallszahlengenerator für eine Stromchiffre verwendet

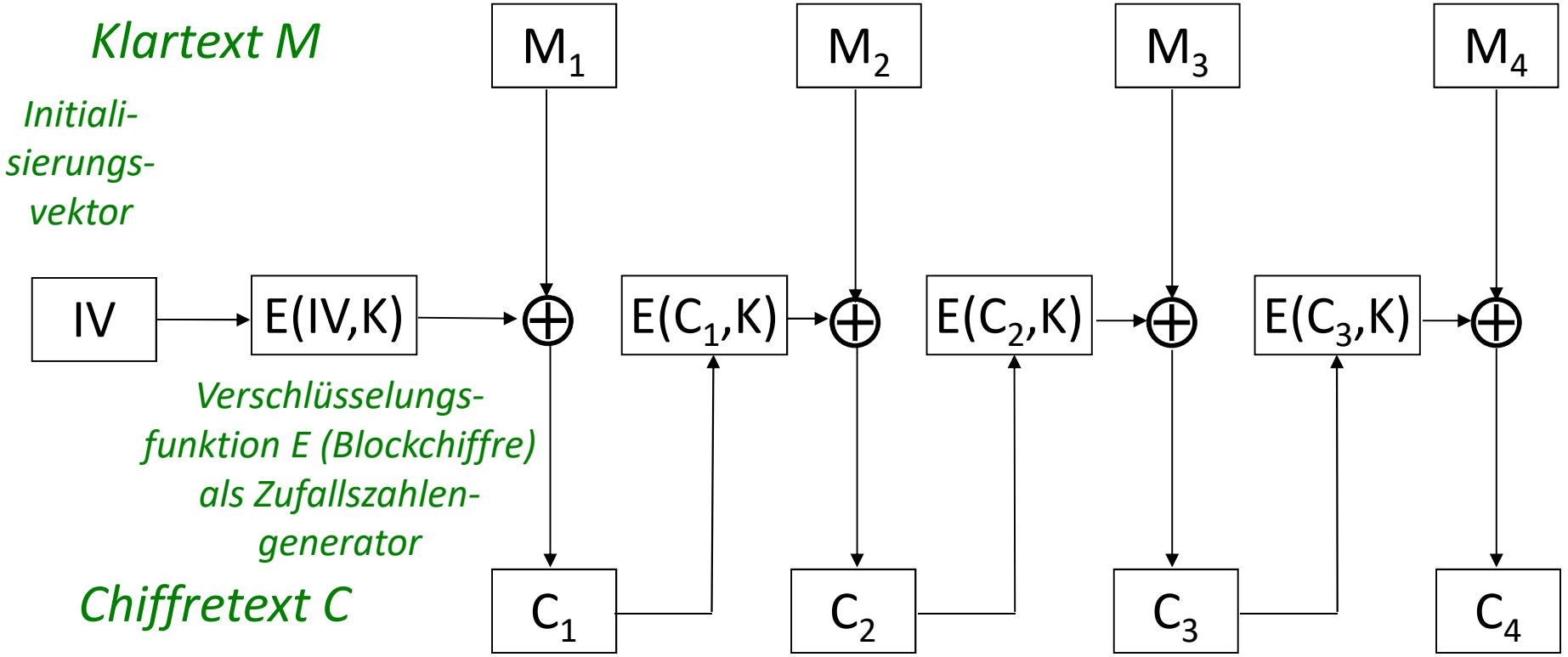


Berechnung eines Chiffretextblocks:

$C_i = M_i \oplus O_i$ und $O_i = E(O_{i-1}, K)$ mit $O_0 = IV$

CFB (Cipher Feedback) - Modus

Wie OFB, zusätzlich Rückkopplung des Chiffretextes zur Zufallszahlerzeugung

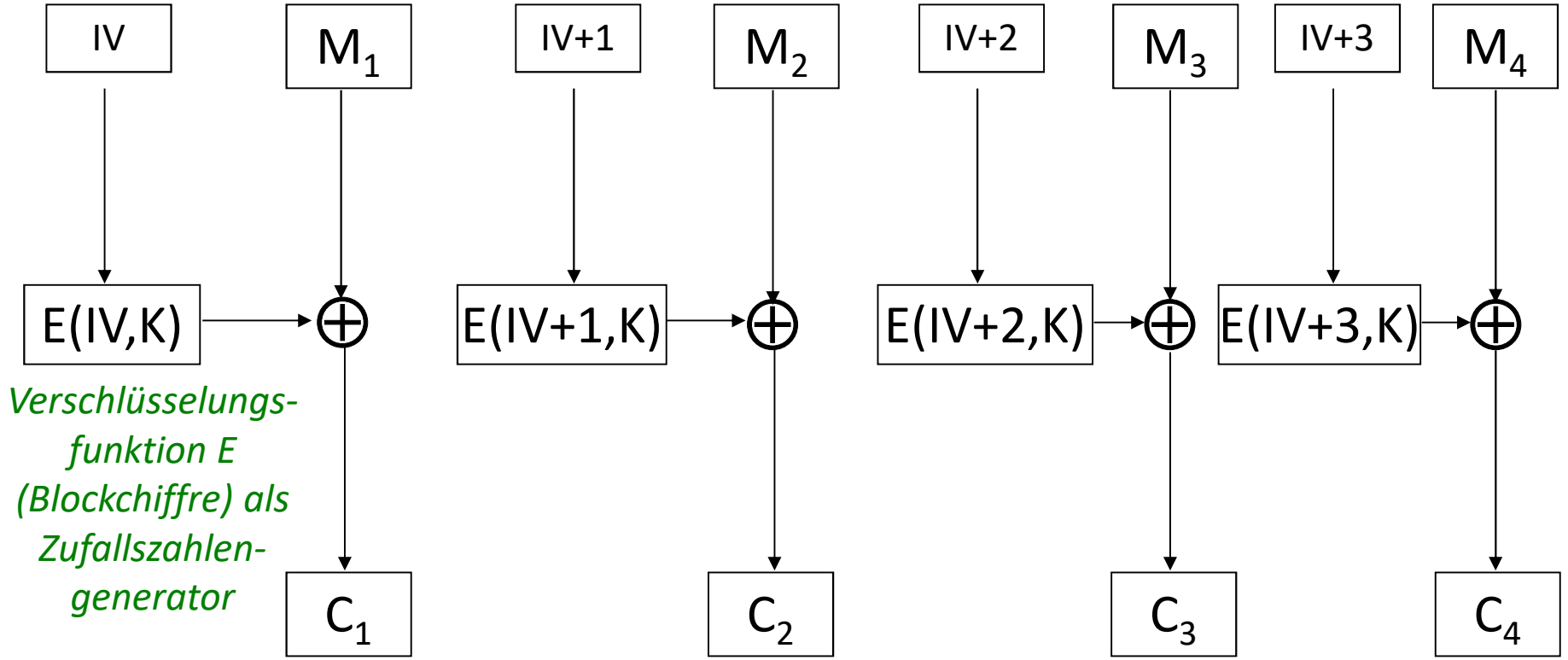


Berechnung eines Chiffretextblocks:

$C_i = M_i \oplus E(C_{i-1}, K)$ mit $C_0 = IV$

CTR (Counter) - Modus

Wie OFB, mit Verkettung der Chiffretextblöcke über einen Zähler



Verschlüsselungs-funktion E (Blockchiffre) als Zufallszahlen-generator

Berechnung eines Chiffretextblocks:

$C_i = M_i \oplus E(Counter_{i-1}, K)$

mit $Counter_0 = IV$ und $Counter_i = Counter_{i-1} + 1$

Zusammenfassung: Betriebsmodi von Blockchiffren (I)

- **ECB** (Electronic Code Book): Blockweise verschlüsseln
  - Gleiche Klartextblöcke ergeben gleiche Chiffretextblöcke („Codebuch“)
  - Auffüllen des letzten Blocks nötig („Padding“)
  - Probleme durch „Man in the Middle“: Wiedereinspielen von Blöcken, Vertauschen der Reihenfolge, Blockentnahmen möglich
- **CBC** (Cipher Block Chaining): Verknüpfen mit vorherigem Chiffretextblock
  - Gleiche Klartextblöcke ergeben ungleiche Chiffretextblöcke
  - Initialisierungsvektor IV nötig (Block mit Zufallszahlen)
  - Auffüllen des letzten Blocks nötig („Padding“)
  - Vorteil gegenüber ECB: Wiedereinspielungen / Blockentnahmen / Vertauschungen sind erkennbar: Fehler bei Entschlüsselung!

Zusammenfassung: Betriebsmodi von Blockchiffren (II)

- **OFB** (Output Feedback):
  - Blockchiffre wird als Zufallszahlengenerator für eine Stromchiffre verwendet
  - Startwert = Initialisierungsvektor
  - Vorteil gegenüber ECB/CBC: Blöcke müssen nicht aufgefüllt werden
- **CFB** (Cipher Feedback):
  - wie OFB, zusätzlich Rückkopplung des Chiffretextes zur Zufallszahlerzeugung
  - Vorteil gegenüber OFB: Bei Fehler (Einspielung eines falschen Blocks) können alle nachfolgenden Blöcke nicht mehr entschlüsselt werden
- **CTR** (Counter):
  - wie OFB, mit Verkettung der Chiffretextblöcke über einen Zähler
  - Vorteil gegenüber OFB/CFB: Zur Entschlüsselung hinten liegender Blöcke muss der Anfang nicht unbedingt entschlüsselt werden
    → Parallelverarbeitung möglich!

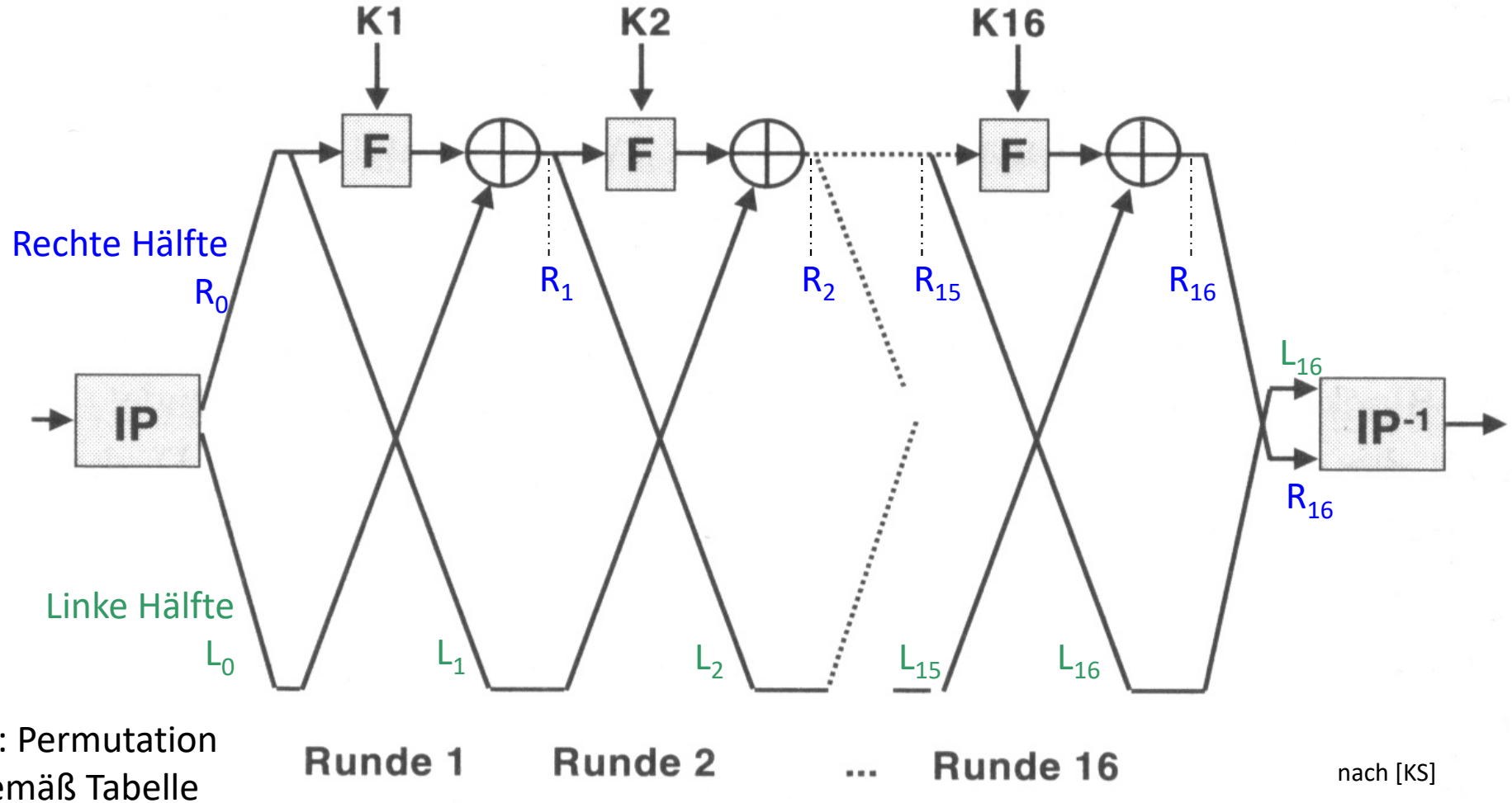
DES („Data Encryption Standard“, 1977)

- **Blockchiffre zur symmetrischen Verschlüsselung**
- Von IBM im Auftrag des NBS (National Bureau of Standards, heute NIST) entwickelt
- **Blocklänge**: 64 Bit *(8 Byte)*
- **Schlüssellänge**: 56 Bit *(+ 8 Prüfbits = 64 Bit)*
- **Ver- / Entschlüsselungsfunktion:**
  Kombiniert folgende Grundfunktionen:
  - **Substitution** (Ersetzen von Bits)
  - **Permutation** (Vertauschung von Bits)
  - **XOR**-Verknüpfung

DES – Verschlüsselung: Überblick

- **Anfangspermutation** IP (gemäß Tabelle)
- **Aufteilung des 64-Bit-Klartextblocks** in zwei Hälften zu je 32 Bit (L_0 und R_0)
- **16 identische Chiffrierschritte („Runden“):** i = 1,..,16
  - Auf R_{i-1} wird eine **Verschlüsselungsfunktion F** angewendet, die auch einen 48-Bit – Teil des Schlüssels (K_i) als zusätzliche Eingabe benutzt
  - Das Ergebnis von F wird mit L_{i-1} XOR-verknüpft zum neuen R_i: $R_i = F(R_{i-1}, K_i) \oplus L_{i-1}$
  - R_{i-1} wird zum neuen L_i: $L_i = R_{i-1}$
- **L_{16} und R_{16} werden vertauscht und zusammengefasst**
- **Endpermutation** (Umkehrung vom Anfang)

DES – Verschlüsselung: Überblick



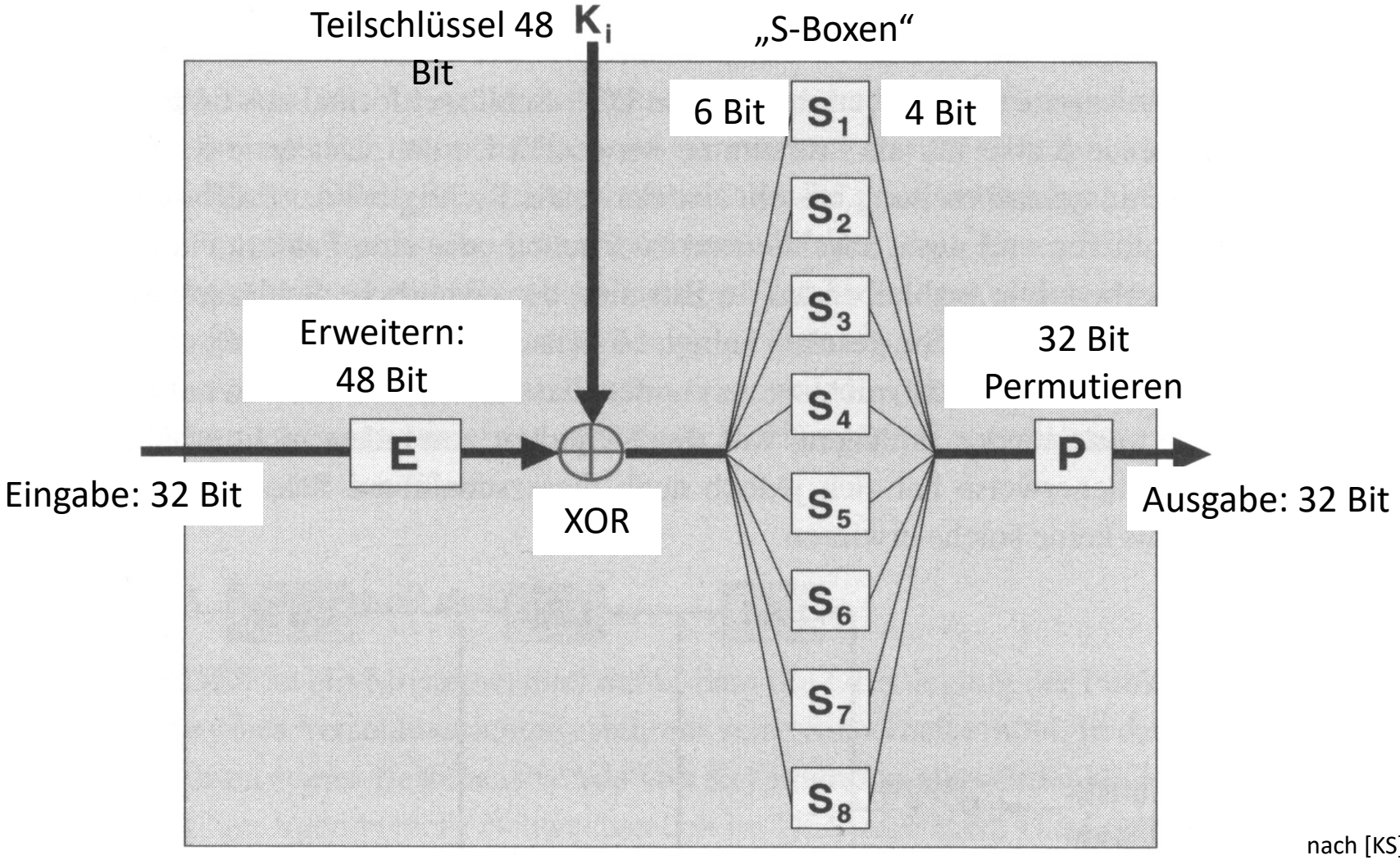
IP: Permutation gemäß Tabelle

nach [KS]

Die Verschlüsselungsfunktion F

- **Eingabe**: Bitblock der Länge 32 Bit
- **Erweitern** des Eingabeblocks auf **48 Bit**
  (Verdoppelung festgelegter Bitstellen)
- **XOR**-Verknüpfung mit **48-Bit-Teilschlüssel K_i**
- **Aufspaltung in 8 Blöcke zu je 6 Bit** und Anwendung je einer Substitutionsfunktion $s_1,..,s_8$
- **„S-Boxen“ $s_1,..,s_8$** sind definiert über Ersetzungstabellen aus 4 Zeilen und 16 Spalten (6 Bit Eingabe) mit 4-Bit-Zahlen als Eintrag (Ausgabe)
- **Zusammensetzen** der Ausgabe (4 * 8 = 32 Bit)
  aus den S-Boxen
- **Permutieren** des 32-Bit-Ausgabeblocks über Tabelle

Die Verschlüsselungsfunktion F



nach [KS]

DES - Entschlüsselung

- Einziger Unterschied zur Verschlüsselung:
  Anwendung der 16 Teilschlüssel **in umgekehrter Reihenfolge**
  → liefert den Klartext!
- Beweis:
  - Klartext: L_0R_0 , Chiffretext: $R_{16}L_{16}$ (wegen Vertauschung)
  - Ein Schritt i, i = 16,..,1 erzeugt aus $\mathbf{R_iL_i}$ den Wert $\mathbf{lr}$ (linke Hälfte und rechte Hälfte) mit

$$\mathbf{l = L_i = R_{i-1}} \text{ und}$$

$$\mathbf{r = F(L_i, K_i) \oplus R_i}$$

$$\mathbf{= F(L_i, K_i) \oplus (F(R_{i-1}, K_i) \oplus L_{i-1})}$$

$$\mathbf{= L_{i-1}}$$

also: $\mathbf{lr = R_{i-1}L_{i-1}}$

da $\mathbf{R_i = F(R_{i-1}, K_i) \oplus L_{i-1}}$ gemäß Verschlüsselung

da $\mathbf{L_i = R_{i-1}}$ gemäß Verschlüsselung

  - Die Endvertauschung liefert für R_0L_0 den Klartext L_0R_0

DES - Eigenschaften

- **Kryptographische Sicherheit**: sehr gut
  - große Streuung
  - Änderung eines Klartextbits hat starke Auswirkungen
- **Problem**: geringe Schlüssellänge von 56 Bit!
  - 1997: DES-Schlüssel erstmals durch vollständige Schlüsselraumsuche in 4 Monaten „geknackt“ (Known-Plaintext-Attacke)
  - 1999: Rekord bei 22 Stunden (mit Unterstützung von 100.000 Internet-Nutzern)
  - 2008: Die Weiterentwicklung des FPGA-basierten Parallelrechners COPACOBANA, die RIVYERA, bricht DES erstmals in weniger als einem Tag
    - 292 Millionen DES-Schlüssel pro Sekunde testbar
    - Benutzt 128 FPGAs
    - Kosten: < 10.000 €

Aufwand für vollständige Schlüsselraumsuche?

- **Pessimistische Annahme**:
  - 56-Bit-Schlüssel in 1 Sekunde durch vollständige Schlüsselraumsuche (2^{56} mögliche Schlüssel!) knackbar
- Pro zusätzlichem Schlüsselbit verdoppelt sich der Suchaufwand!

Schlüssellänge	Zeitaufwand
56 Bit	1 Sekunde
64 Bit	4 Minuten
80 Bit	194 Tage
112 Bit	10^9 Jahre
128 Bit	10^{14} Jahre
192 Bit	10^{33} Jahre
256 Bit	10^{52} Jahre

1 Milliarde!

nach [KS]

DES-Nachfolger

- **Triple-DES (3-DES)**
  - Dreifache Anwendung des einfachen DES-Algorithmus („Encrypt-Decrypt-Encrypt“)
  - Schlüssellänge: 168 Bit
  - **Sicherheit entsprechend 112 Bit** (2-DES-Angriff möglich)
- **IDEA („International Data Encryption Algorithm“)**
  - Weiterentwicklung des DES
    - Schlüssellänge 128 Bit
    - SW-Implementierung doppelt so schnell wie DES
  - Bestandteil von PGP *(frühe Versionen)*
- **AES („Advanced Encryption Standard“)**
  - **Algorithmus „Rijndael“ ist offizieller DES-Nachfolger seit 2000**
  - Aufgrund eines vom NIST ausgeschriebenen Wettbewerbs ermittelt

AES-Überblick

- **Blockchiffre zur symmetrischen Verschlüsselung**
- Blocklänge: 128 Bit *(16 Byte)*
- Schlüssellänge: 128, 192 oder 256 Bit
- Anzahl an Runden: 10 – 14 (abhängig von der Schlüssellänge)
- Blöcke werden als 4x4-Matrix verarbeitet (Eintrag: 1 Byte)
- In jeder Runde werden vier verschiedene Funktionen angewendet:
  - ByteSub (Substitution durch Ersetzungstabelle)
  - ShiftRow (Permutation durch Zeilenshift)
  - MixColumn (Permutation durch Spaltenshift und XOR)
  - AddRoundKey (XOR-Verknüpfung mit Teilschlüssel)
- Alle Funktionen sind invertierbar (→ Entschlüsselung)!

Bisher **keine** relevanten Schwachstellen entdeckt → Standardverfahren!

Blockchiffren nach AES (Auswahl)

- **Clefia (Sony 2007)**
  - Struktur wie DES
  - Blocklänge: 128 Bit *(16 Byte)*
  - Schlüssellänge: 128, 192 oder 256 Bit
  - Anzahl an Runden: 18 – 26 (abhängig von der Schlüssellänge)
  - Speziell für DRM (Digital Rights Management) entwickelt
- **Chiasmus (BSI 2014)**
  - Verschlüsselung von Dateien unter Windows / Linux
  - Chiasmus darf grundsätzlich nur dort eingesetzt werden, wo ein öffentliches Interesse für die Nutzung besteht
  - Blocklänge: 64 Bit *(8 Byte)*
  - Schlüssellänge: 160 Bit

Kapitel 3
Grundlagen der Kryptographie

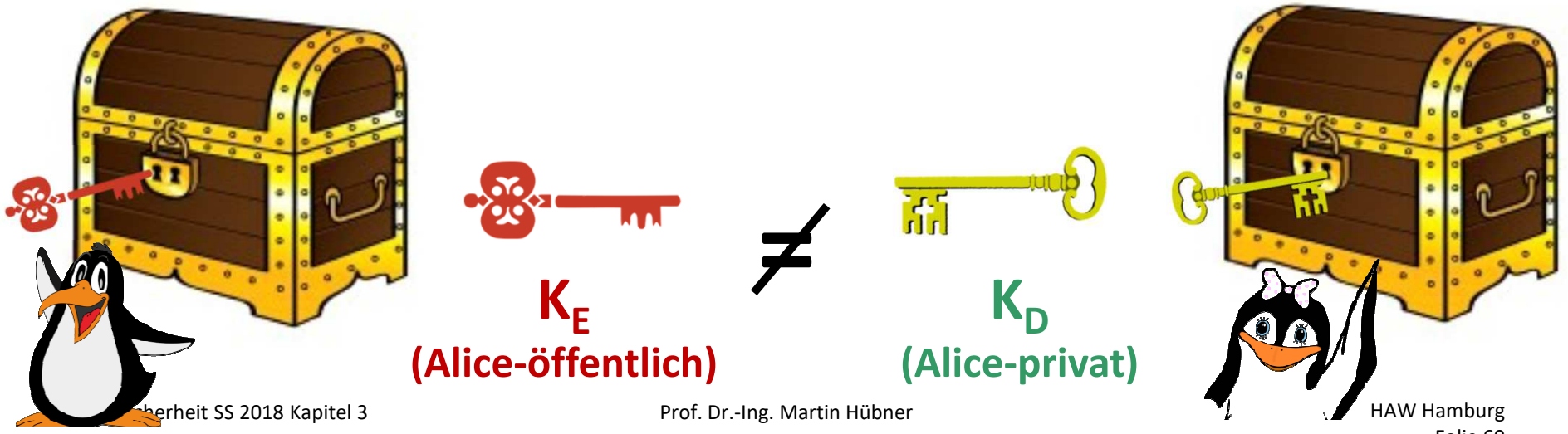
1. Einführung
2. Einfache Symmetrische Verfahren
3. Moderne symmetrische Verfahren
4. Asymmetrische Verfahren („Public Key“)
5. Digitale Signaturen und kryptographische Hashfunktionen

Asymmetrische Verfahren („Public-Key“)

Idee: Verwende ein Schloss mit zwei verschiedenen Schlüsseln 1 und 2

Wenn mit Schlüssel 1 abgeschlossen wurde, kann später nur mit Schlüssel 2 aufgeschlossen werden (und umgekehrt)

- Bob will an Alice eine verschlüsselte Nachricht senden
- Alice besitzt ein Schlüsselpaar (Schlüssel 1 , Schlüssel 2):
- Alice **gibt Bob Schlüssel 1** ("öffentlicher" Schlüssel von Alice) und **behält Schlüssel 2** ("privater" Schlüssel von Alice)



Anwendung von Public-Key-Verfahren

- **Verschlüsselung** von Nachrichten
  (→ Geheimhaltung)
  - Der Sender verwendet den **öffentlichen Schlüssel des Empfängers** zur Verschlüsselung
  - Nur der **Besitzer des zugehörigen privaten Schlüssels** kann die Nachricht entschlüsseln
- **Digitale Unterschrift ("Signierung")** von Nachrichten
  (→ Authentifikation, Verbindlichkeit, Datenintegrität,)
  - Der Sender verwendet **seinen privaten Schlüssel zur Verschlüsselung**
  - Wenn der Empfänger die Nachricht mit dem **öffentlichen Schlüssel des Senders entschlüsseln** kann, **muss** der Sender im Besitz des privaten Schlüssels sein (ist so eindeutig identifiziert) und die übertragenen Daten wurden nicht verändert

Grundidee der mathematischen Realisierung: Einwegfunktionen

- Eigenschaften einer **Einwegfunktion** f: X→ Y
  - f ist injektiv (umkehrbar)
  - für alle x ϵ X ist f(x) effizient („einfach“) berechenbar
  - es gibt aber kein effizientes Verfahren, um aus einem Funktionswert y = f(x) das Urbild x zu berechnen!
- Beispiele für Einwegfunktionen:
  - **Modulo-Exponentiation** (Problem: Diskreter Logarithmus)
  - **Produkt zweier großer Primzahlen** (Problem: Faktorisierung großer Zahlen)
- Aber: Konstruktion von Einwegfunktionen mit **„Falltür“** ist evtl. möglich
  - Falltür = geheime Zusatzinformation, mit der für einen Funktionswert y das Urbild x effizient berechnet werden kann

Diskreter Logarithmus

Seien g, p ϵ $\mathbb{N}$ gegeben. Dann ist f: {0,..,p-1} → {0,..,p-1} mit

$$\mathbf{f(x) = g^x \ (mod\ p) = y}$$

eine Einwegfunktion!

Zu einem beliebigen y ϵ {1, .., p-1} kann der **diskrete Logarithmus** $\mathbf{x = \log_g y \ (mod\ p)}$ auch für bekannte Werte g, p ϵ $\mathbb{N}$ **nicht effizient berechnet** werden, also ein
x ϵ {1, .., p-1} bestimmt werden, für das gilt: $y = g^x \ (mod\ p)$

Beispiel: g = 3, p = 17, $f(x) = 3^x \ (mod\ 17)$

$f(3) = 3^3 \ (mod\ 17) = 10$

Für welches x gilt f(x) = 4 ???

Generatoren

g = 3 ist „Generator“ einer Gruppe **Z(17,.)** bzgl. Modulo-Multiplikation
→ der diskrete Logarithmus existiert immer!

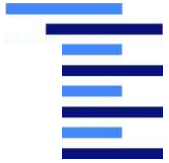
g = 4: Funktionswerte f(x) kommen mehrfach vor
→ Funktion ist nicht umkehrbar
→ der diskrete Logarithmus existiert nicht (keine Eindeutigkeit)!

x	f(x) 3^x mod 17
0	---
1	3
2	9
3	10
4	13
5	5
6	15
7	11
8	16
9	14
10	8
11	7
12	4
13	12
14	2
15	6
16	1

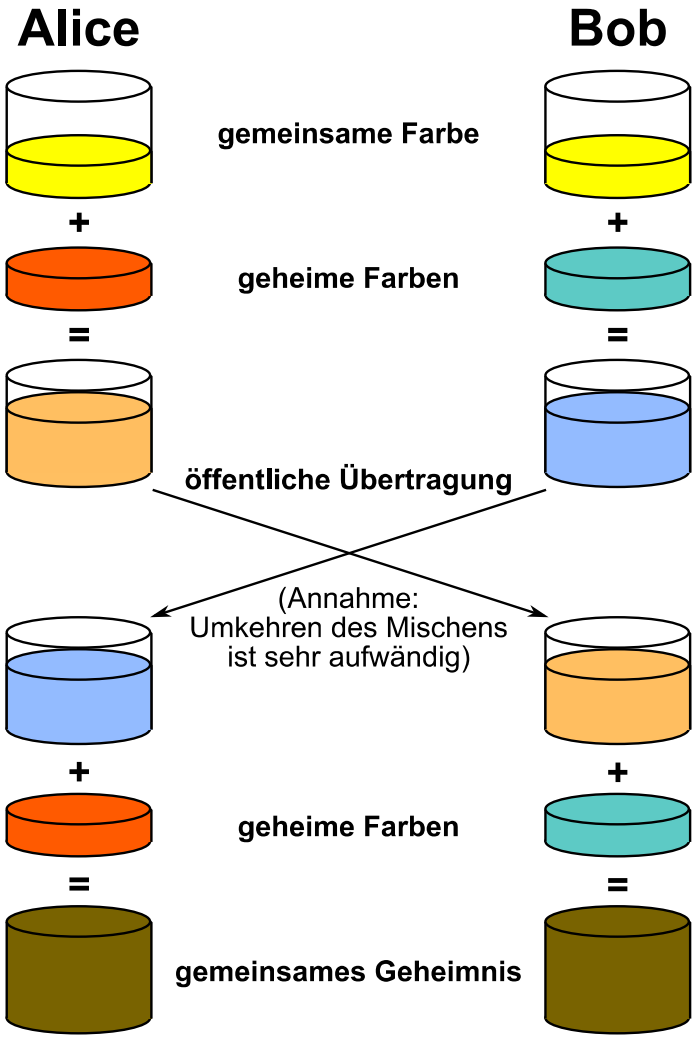
g heißt „Generator“, wenn jede Zahl aus {1, .., p-1} genau einmal als y = f(x) vorkommt *(für alle x > 0)*!

x	f(x) 4^x mod 17
0	1
1	4
2	16
3	13
4	1
5	4
6	16
7	13
8	1
9	4
10	16
11	13
12	1
13	4
14	16
15	13
16	1

Problem: Vereinbarung eines gemeinsamen (geheimen) Schlüssels über das (unsichere) Internet



- Verfahren zum Schlüsselaustausch von **Diffie-Hellmann** (1976) *(→ Bietet aber **keine** Verschlüsselung!)*
- Mathematische Basis: Problem des Diskreten Logarithmus
- Analogie: Alice und Bob wollen sich auf eine geheime Farbmischung (*Einwegfunktion!*) einigen



Schlüsselaustauschverfahren von Diffie-Hellmann

- Ziel: Austausch (Vereinbarung) eines geheimen Schlüssels zwischen Alice und Bob
- Alice und Bob wählen eine große Primzahl p und einen Generator g (öffentlich bekannt)
  - Alice wählt eine zufällige Zahl x < p *(x: privater Schlüssel)* und schickt Bob a = g^x (mod p) *(a: öffentlicher Schlüssel)*
  - Bob wählt eine zufällige Zahl y < p *(y: privater Schlüssel)* und schickt Alice b = g^y (mod p) *(b: öffentlicher Schlüssel)*
- Beide können nun den gemeinsamen geheimen Schlüssel K_{ab} berechnen:
  - Alice: $K_{ab} = b^x \pmod p = (g^y)^x \pmod p = g^{xy} \pmod p$
  - Bob: $K_{ab} = a^y \pmod p = (g^x)^y \pmod p = g^{yx} \pmod p = g^{xy} \pmod p$

Sicherheit des Diffie-Hellmann-Algorithmus

- Angriff: Berechnung des Sitzungsschlüssels K_{ab} aus den öffentlichen Schlüsseln a und b
  - Berechnung des diskreten Logarithmus nötig, z.B. $x = \log_g a \pmod p$
  - für Schlüssellängen ab 2048 Bit unmöglich!
- Aber: **Man-in-the-Middle-Attacke** möglich!

- Sicherheit nur bei langen Schlüsseln gegeben!

→ Verbesserungsvorschlag?

Elliptische Kurven (ECC)

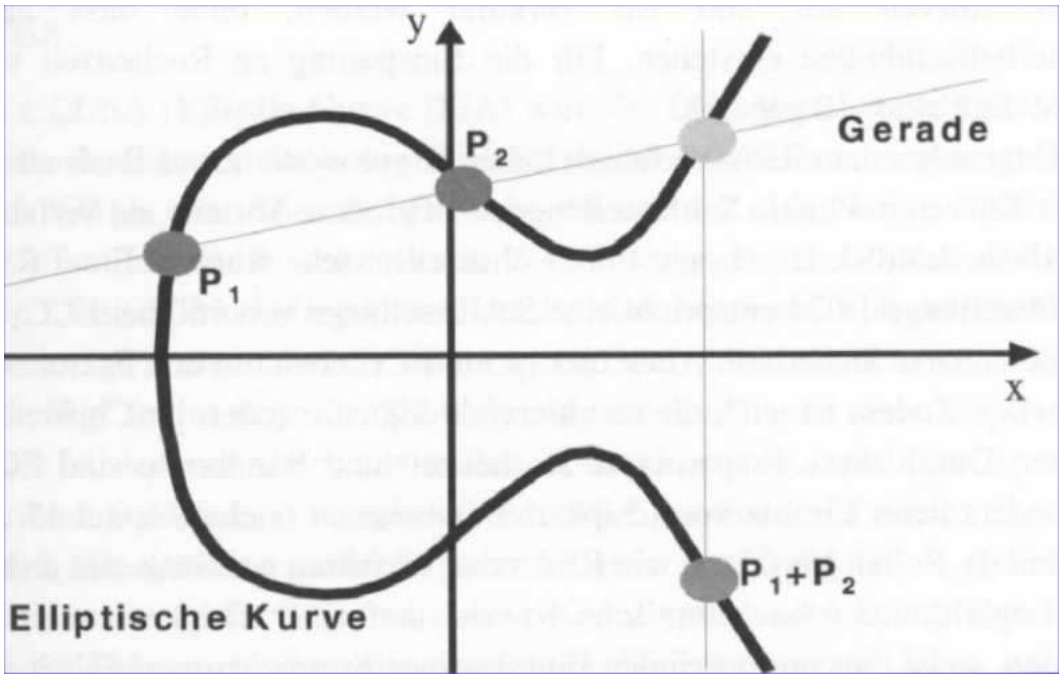
- Kurven auf einem endlichen Körper, die die folgende Gleichung erfüllen:

$$y^2 + a_1xy + a_2y = x^3 + a_3x^2 + a_4x + a_5 \pmod p$$

- mit zusätzlichem imaginären „neutralem Element“ O (Punkt im Unendlichen)
- Beispiel:
  - „Curve25519“: $y^2 = x^3 + 486662x^2 + x \pmod{2^{255} - 19}$

- Eigenschaft einer elliptischen Kurve:
  Jede Gerade, die die Kurve schneidet, hat genau **drei** Schnittpunkte!

Elliptische Kurven (ECC): Addition



- Zwei Punkte P_1 und P_2 einer elliptischen Kurve werden addiert, indem eine Gerade durch sie gezogen wird.
- Ergebnis P_1+P_2: Spiegelung des 3. Schnittpunkts der Gerade an der x-Achse

Multiplikation und Potenzbildung können abgeleitet werden → Berechnung von diskreten Logarithmen möglich, aber deutlich „aufwändiger“!

→ Als Basis für alle Krypto-Verfahren, die auf diskreten Logarithmen basieren, verwendbar (z.B. Diffie-Hellmann)

→ Liefert dieselbe Sicherheit bei kürzeren Schlüssellängen!

RSA-Verfahren

- **R**ivest, **S**hamir und **A**dleman 1978
- Public-Key-Verfahren
- Basis: Produkt zweier großer Primzahlen
- Blockchiffre: Bitfolgen werden als Zahlen interpretiert
- Schlüssellänge: variabel, 2048 – 4096 Bit
- Anwendung:
  - **Verschlüsselung (inkl. Schlüsselaustausch) und**
  - **digitale Signierung**

RSA: Mathematische Grundbegriffe

Inverses Element

- Seien a und b natürliche Zahlen < n. Dann heißt b (multiplikatives) inverses Element von a modulo n, wenn gilt: a*b (mod n) = 1 .
  - Ein inverses Element b existiert zu a, wenn a und n keine gemeinsamen Teiler haben, d.h. ggT(a,n) = 1
  - Ein inverses Element ist eindeutig bestimmt.

Eulersche ϕ -Funktion

- ϕ(n) gibt an, wie viele natürliche Zahlen a mit 0 < a < n zu n teilerfremd sind.
  - Für jede Primzahl n gilt: ϕ(n) = n-1
  - p und q seien Primzahlen:
    Wenn n = p*q, dann gilt: ϕ(n) = (p-1)(q-1)

RSA: Wahl der Schlüssel

1. Wähle zwei zufällige große Primzahlen **p** und **q**
2. Berechne "Modul" **n = p*q**
3. Wähle eine Zahl **e** (mit e < n), die keine gemeinsamen Teiler mit ϕ(n) = (p-1)*(q-1) hat.
4. Finde eine Zahl **d** so, dass (e*d -1) durch ϕ(n) teilbar ist
   d.h.: e*d (mod ϕ(n)) = 1 ,
   also ist d inverses Element von e modulo ϕ(n).
5. **Öffentlicher Schlüssel ist (n,e)**,
   Privater Schlüssel ist (n,d).

e ist meist öffentlich bekannte Primzahl!

p und q müssten hierfür bekannt sein!!

RSA: Verschlüsselung, Entschlüsselung

Öffentlicher Schlüssel K_E ist (n,e)
Privater Schlüssel K_D ist (n,d)

1. Verschlüsselung von Klartext M (Bitfolge ≅ Zahl < n):

$$E(M,K_E) = M^{e} \pmod{n} = C$$

2. Entschlüsselung des Chiffretextes C (Bitfolge ≅ Zahl < n):

$$D(C,K_D) = C^{d} \pmod{n} = M$$

➔ $M = (M^e)^d \pmod{n}$

Magie?

RSA Beispiel:

Bob wählt p = 5, q = 7 → n = 35, (p-1)*(q-1) = 24

e = 5 (e und (p-1)*(q-1) sind teilerfremd)

d = 29 (e*d -1 = 144 ist teilbar durch 24)

	Buchstabe	M	M^e	$C = M^e \pmod n$
Alice verschlüsselt:	1	12	248832	17

	C	C^d	$M = C^d \pmod n$	Buchstabe
Bob entschlüsselt:	17	481968572106750915091411825223072000	12	1

RSA: Warum $M = (M^e)^d \bmod n$?

Ergebnis der Zahlentheorie:
Wenn *p,q* Primzahlen sind und $n = p*q$, dann gilt

$$x^y \pmod n = x^{y \pmod{\varphi(n)}} \pmod n$$

$$(M^e)^d \pmod n = M^{e*d} \pmod n$$

$$= M^{(e*d \pmod{\phi(n)})} \pmod n$$

(Ergebnis der Zahlentheorie, siehe oben)

$$= M^1 \pmod n$$

(weil $e*d \pmod{\phi(n)} = 1$ aufgrund der Schlüsselwahl)

$$= M$$

(weil $M < n$ ist)

RSA: Parameterwahl in der Praxis

- **e**: Teil des öffentlichen Schlüssels
  - Kleine Primzahlen (z.B. 65537) können gewählt werden, da die Sicherheit von der Größe des Moduls **n** abhängt!
- **n**: Variable Schlüssellänge durch beliebige Wahl von **n**
  - 512 Bit – Schlüssellänge (155 Dezimalstellen) ist nicht ausreichend (1999 in 7,4 Monaten durch Faktorisierungsangriff mit Einsatz von 35,7 CPU-Jahren geknackt!)
  - **n** sollte mindestens 2048 Bit lang sein (~600 Dezimalstellen)
- **p** und **q**:
  - Sollten sich um einige Ziffern in der Länge unterscheiden (→ Faktorisierungsangriff!)

Sicherheit des RSA-Verfahrens

- Vollständige Schlüsselraumsuche
  - Im Mittel müssen $2^{\text{(Länge von n in Bit)-1}}$ Schlüssel getestet werden
  - Noch aussichtsloser als bei symmetrischen Verfahren aufgrund der üblichen großen Schlüssellängen!

- Faktorisierungsangriff
  - Vermutung (nicht bewiesen!): Außer der vollständigen Schlüsselraumsuche bleibt nur noch die Möglichkeit, n durch ein Faktorisierungsverfahren in die Primfaktoren p und q zu zerlegen, um einen Chiffretext zu entschlüsseln!

- Fazit: RSA ist bei großer Schlüssellänge ein **praktisch sicheres Verschlüsselungsverfahren**!

Beispiel für ein Faktorisierungsverfahren: Fermat´sche Faktorisierung

- Seien p und q Primzahlen und n = p * q
- Ziel: Ermittle p und q aus der Kenntnis von n
- Ergebnis der Zahlentheorie:
  - Für ein solches n existieren zwei Zahlen a *(„Mitte“ zwischen p und q)* und b *(Abstand von der Mitte zu p bzw. q)*, so dass gilt:
  - $n = p * q = (a + b) * (a - b) = a^2 - b^2$
- Algorithmus: Suche solche Zahlen a, b
  - $a = \lfloor \sqrt{n} + 1 \rfloor$
  - *while not* ($a^2 - n$) Quadratzahl *do* a++
- Folgerung für RSA-Anwendung:
  - p und q sollten sich in der Länge unterscheiden!

Achtung:
$(a^2 - n) = b^2$

Implementierung des RSA-Verfahrens

- **Algorithmen zur Schlüsselerzeugung**
  - Erzeugung großer Primzahlen
    → Primzahlgeneratoren
  - Berechnung der „multiplikativen Inversen“ d (Schritt 4)
    → Basis: modifizierter Euklidischer Algorithmus zur Berechnung des ggT
- **Verschlüsselung / Entschlüsselung**
  - Klartext (Bitfolge) muss in Folge von Zahlen transformiert werden → Blockchiffre
  - Potenzieren großer Zahlen
    → Basis: Wiederholtes Quadrieren
  - Geschwindigkeit im Verhältnis zu DES
    - Hardware-Realisierung: ~ 1000 mal langsamer
    - Software-Realisierung: ~ 100 mal langsamer

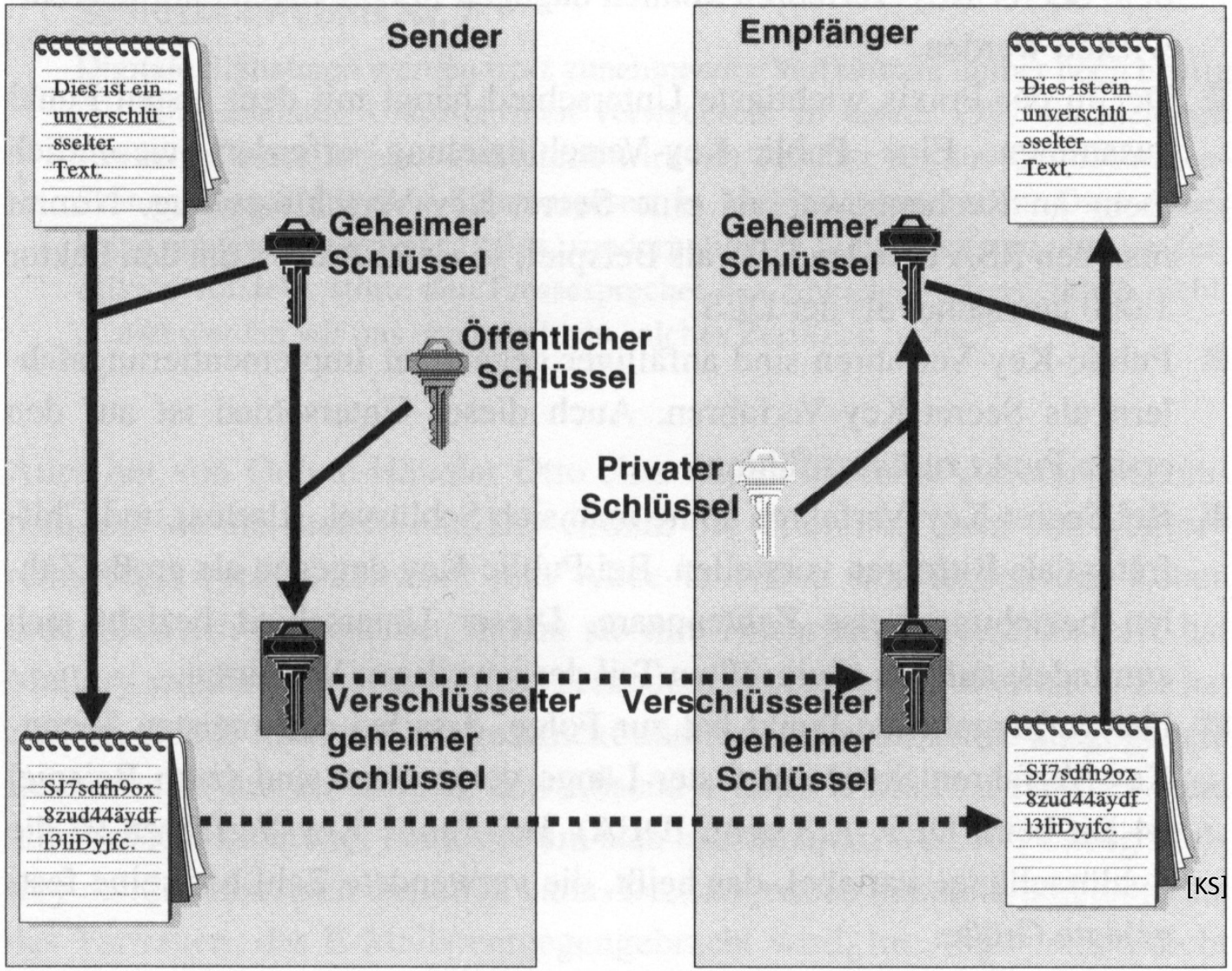
Primzahlgeneratoren

- **Wie kann man (große) Primzahlen erzeugen?**
  (z.B. 512, 1024 oder 2048 Bit Länge)
  1. Ungerade (Pseudo-)Zufallszahl p generieren
  2. Mit „probabilistischem“ Verfahren testen, ob p eine Primzahl ist
  3. Falls ja, fertig! Sonst: → Schritt 1

- Probabilistische Primzahltestverfahren
  - liefern nur mit einer hohen Wahrscheinlichkeit ein korrektes Ergebnis
  - sind aber sehr schnell
  - Standard: „Rabin-Miller“-Verfahren
  - Basis: Zahlentheorie

Kombination von Verfahren in der Praxis

Das Public-Key-Verfahren wird nur zum sicheren Austausch eines **"Sitzungs"-Schlüssels** für ein symmetrisches Verfahren verwendet!



Perfect Forward Secrecy

- Problemstellung (Beispiel SSL/TLS):
  - Alle Sitzungsschlüssel werden vom Client erzeugt und mit dem öffentlichen Schlüssel des Servers verschlüsselt übertragen
  - Angreifer (z.B. NSA) protokolliert alle verschlüsselten Übertragungen mit
  - Wenn der private Schlüssel des Servers kompromittiert wurde, sind nachträglich alle geheimen Sitzungs-Schlüssel und damit alle Nachrichten lesbar
- Perfect Forward Secrecy - Prinzip
  - Die Vertraulichkeit eines Sitzungs-Schlüssels sollte möglichst nicht von der Vertraulichkeit eines Langzeit-Schlüssels abhängen
  - Beispiel SSL/TLS-Empfehlung:
    Diffie-Hellmann mit "neuen" Parametern pro Sitzung zum Schlüsselaustausch verwenden statt RSA

PKCS-Standards

- PKCS: „Public Key Cryptography Standards“ (13 Dokumente)
- Entwickelt von der Firma RSA Security (ab 1991)
- **Ziel**: Festlegen von Implementierungsstandards und Datenformaten für kryptographische Verfahren
  Beispiele:
  - PKCS#1: RSA-Implementierungsstandards
  - PKCS#3: Diffie-Hellman-Protokoll
  - PKCS#5: Ableitung eines Schlüssels aus einem Passwort
  - PKCS#12: Dateiformat, das dazu benutzt wird, private Schlüssel mit dem zugehörigen Zertifikat passwortgeschützt zu speichern

Zusammenfassung: Eigenschaften Public Key – Secret Key-Verfahren

Kriterium	Symmetrische Verfahren („Secret Key“), z.B. AES	Asymmetrische Verfahren („Public Key“), z.B. RSA
Mathematische Grundlage	einfach	anspruchsvoll
Entwicklung neuer Verfahren	einfach	sehr schwierig
Geschwindigkeit	schnell	langsam
Form von Klartext, Chiffretext und Schlüsseln	Bitfolgen	Zahlen
Schlüssellänge	fest, 128 – 256 Bit	variabel, 2048 - 4096 Bit *(ECC: ~ Faktor 10 kleiner)*

Kapitel 3
Grundlagen der Kryptographie

1. Einführung
2. Einfache Symmetrische Verfahren
3. Moderne symmetrische Verfahren
4. Asymmetrische Verfahren („Public Key“)
5. Digitale Signaturen und kryptographische Hashfunktionen

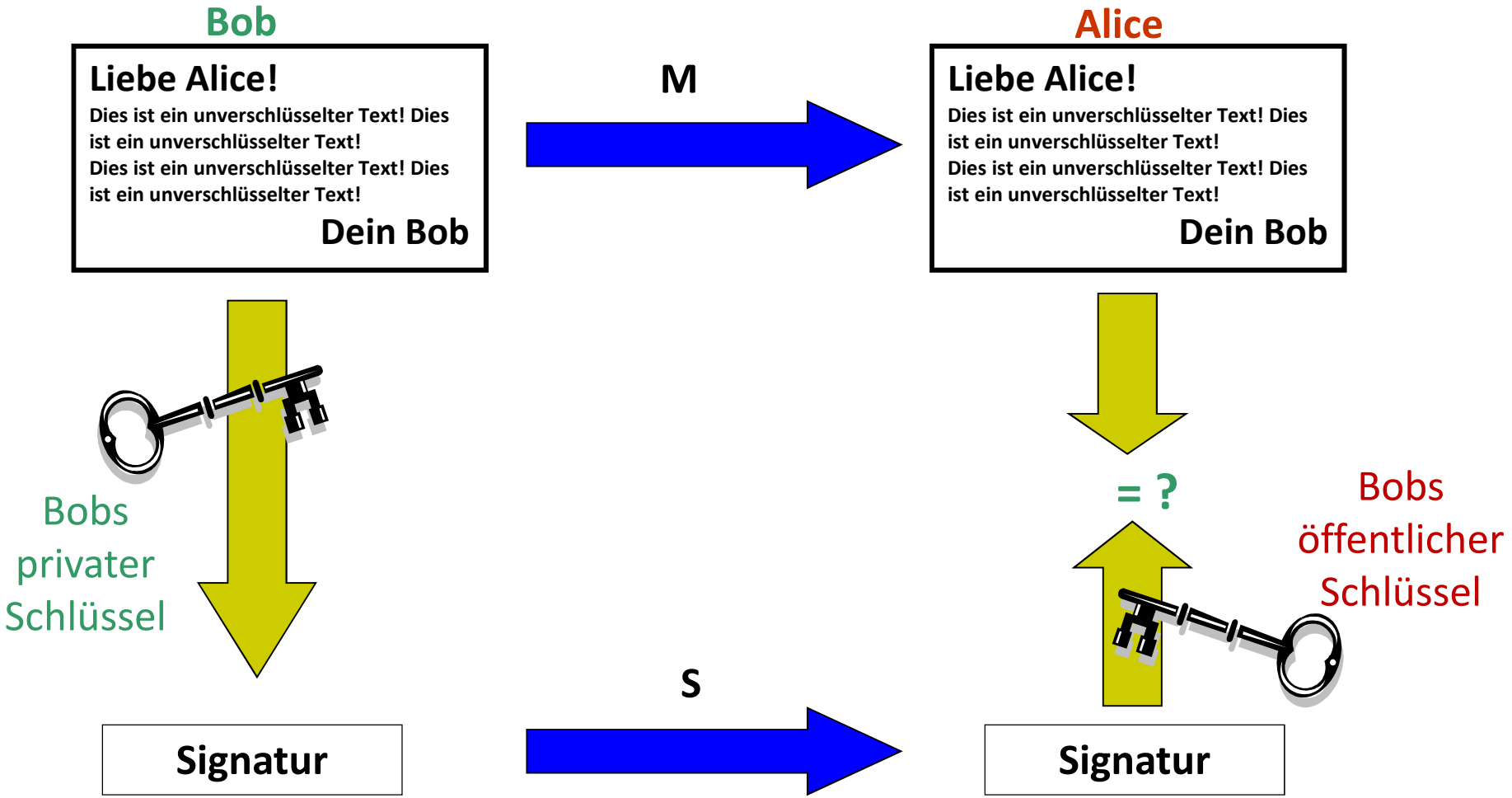
Digitale Signatur

- Kryptographisch erzeugte Bitfolge für elektronische Dokumente (Nachrichten) analog zur (handschriftlichen) Unterschrift
- Anforderungen an eine Digitale Signatur
  - Die Identität des Unterzeichners muss zweifelsfrei feststellbar sein (→ Authentifikation)
  - Die Digitale Signatur darf nur in Verbindung mit dem Originaldokument gültig sein (keine Wiederverwendbarkeit)
  - Ein signiertes Dokument darf nicht unbemerkt verändert werden können (→ Integrität)
  - Der Unterzeichner darf die Signierung nicht abstreiten können (Verbindlichkeit)

Einfache Digitale Signatur mittels Public-Key-Verfahren

- Bob erzeugt eine Digitale Signatur S für ein Dokument M, indem er es mit seinem privaten Schlüssel K_D verschlüsselt: $S = E(M, K_D)$
- Bob sendet M und die Signatur S an Alice
- Alice prüft die Signatur S, indem sie S mit Bobs öffentlichem Schlüssel K_E entschlüsselt: $D(S, K_E)$
- Alice vergleicht nun das übermittelte Dokument M mit $D(S, K_E)$.
- Wenn die Signatur korrekt ist, gilt: $M = D(S, K_E)$ und das Dokument als verifiziert.

Bild: Einfache Digitale Signatur mittels Public-Key-Verfahren



Werden die Anforderungen an eine Digitale Signatur durch ein Public-Key-Verfahren erfüllt?

- Zweifelsfreie Identität
  - √ Nur Bob kennt den notwendigen privaten Schlüssel
- Keine Wiederverwendbarkeit
  - √ Die Signatur ist Ergebnis einer Verschlüsselung eines bestimmten Dokuments
- Integrität
  - √ Bei einer Veränderung des Dokuments oder der Signatur ist keine Verifikation möglich → jede Veränderung wird erkannt
- Verbindlichkeit
  - √ Wer immer die Nachricht unterzeichnet hat, muss Bobs privaten Schlüssel benutzt haben

Signaturverfahren

- RSA
  - Signieren → Verschlüsseln mit privatem Schlüssel (n,d):
    $E(M,K_D) = \mathbf{M^d \ (mod\ n)}$
  - Verifizieren der Signatur
    → Entschlüsseln mit öffentlichem Schlüssel (n,e):
    $D(E(M,K_D), K_E) = \mathbf{(M^d)^e \ (mod\ n)} = M^{e*d} \ (mod\ n) = \mathbf{M}$
- DSA („Digital Signature Algorithm“)
  - Von der NSA entwickelt, seit 1994 US-Standard (NIST)
  - Basiert auf diskretem Logarithmus-Problem
  - Nur für Signierung einsetzbar
  - Langsamer als RSA
  - Frei verfügbar (nicht patentiert)

➔ Geeignet für ECC-Verfahren!

DSA-Basis: ElGamal-Signaturverfahren

- Anwendung: Alice möchte Bob signierte Nachrichten senden
- Vorbereitung *(analog zu Diffie-Hellmann)*:
  - Alice wählt eine große Primzahl **p** und einen passenden Generator **g** sowie eine zufällige Zahl **x** < p *(x: privater Schlüssel)* und berechnet **a** = g^x (mod p) *(a: öffentlicher Schlüssel)*
  - Alice gibt (p, g, a) öffentlich bekannt
- Signieren einer Nachricht m durch Alice
  - Erzeuge eine Zufallszahl **k** (0 < k < p-1), die keine gemeinsamen Teiler mit p-1 hat (→ multipl. inverses Element k^{-1} von k (mod p-1) existiert)
  - Berechne **r** = g^k (mod p)
  - Stelle eine Beziehung zur Nachricht **m** her:
    m = x * r + k *s (mod p-1) und löse nach der Unbekannten s auf:
    s = (m – x * r) * k^{-1} (mod p-1)
  - Die Signatur der Nachricht m *(die Nachrichtenbits werden als Zahl interpretiert!)* sind die Zahlen **r** und **s**

ElGamal: Verifizieren einer Signatur

- Verifizieren der Signatur (r,s) einer Nachricht m durch Bob
  1. Gilt $0 < r < p$ und $0 < s < p-1$?
  2. Gilt $g^m \pmod p = a^r * r^s \pmod p$?
  - Wenn 1. und 2. gelten, ist die Signatur erfolgreich verifiziert
- Beweis des Verfahrens:
  - $m = x * r + k * s \pmod{p-1}$ *Alice' Konstruktion*
    ➔ $g^m = g^{x*r+k*s} \pmod{p-1}$
    ➔ $g^m \pmod p = g^{x*r+k*s} \pmod p$ *analog Satz Folie RSA-Beweis, da* $p-1=\varphi(p)$
    ➔ $g^m \pmod p = a^r * r^s \pmod p$ *da* $a = g^x \pmod p$ *und* $r = g^k \pmod p$
  - Bob hat alle nötigen Informationen zur Verfügung: m, (p, g ,a) als öffentlicher Schlüssel von Alice, Signatur (r,s)
  - Zur Erzeugung der Signatur (r,s) war die Kenntnis des privaten Schlüssels x und der Nachrichten-spezifischen Zufallszahl k nötig
  - Durch ein jeweils neues k erhält jede Nachricht eine eigene Signatur *(→ aber: aufwändig!)*

Angriffe auf Digitale Signaturverfahren

- Annahmen
  - Zum Verschlüsseln und Signieren werden dieselben Schlüssel verwendet
  - Eine Signatur ist ein Chiffretext des kompletten Dokuments

- Angriff mit gewähltem Kryptotext
  - Angreifer Carl kann Bob einen Klartext zum Entschlüsseln unterschieben und erhält damit als Ergebnis eine gültige Signatur!
  - Angreifer Carl kann Bob einen Chiffretext zum Signieren unterschieben und erhält damit als Ergebnis den Klartext!
  - ➔ Zwei verschiedene Schlüsselpaare zum Verschlüsseln und Signieren verwenden!

Angriffe auf Digitale Signaturverfahren

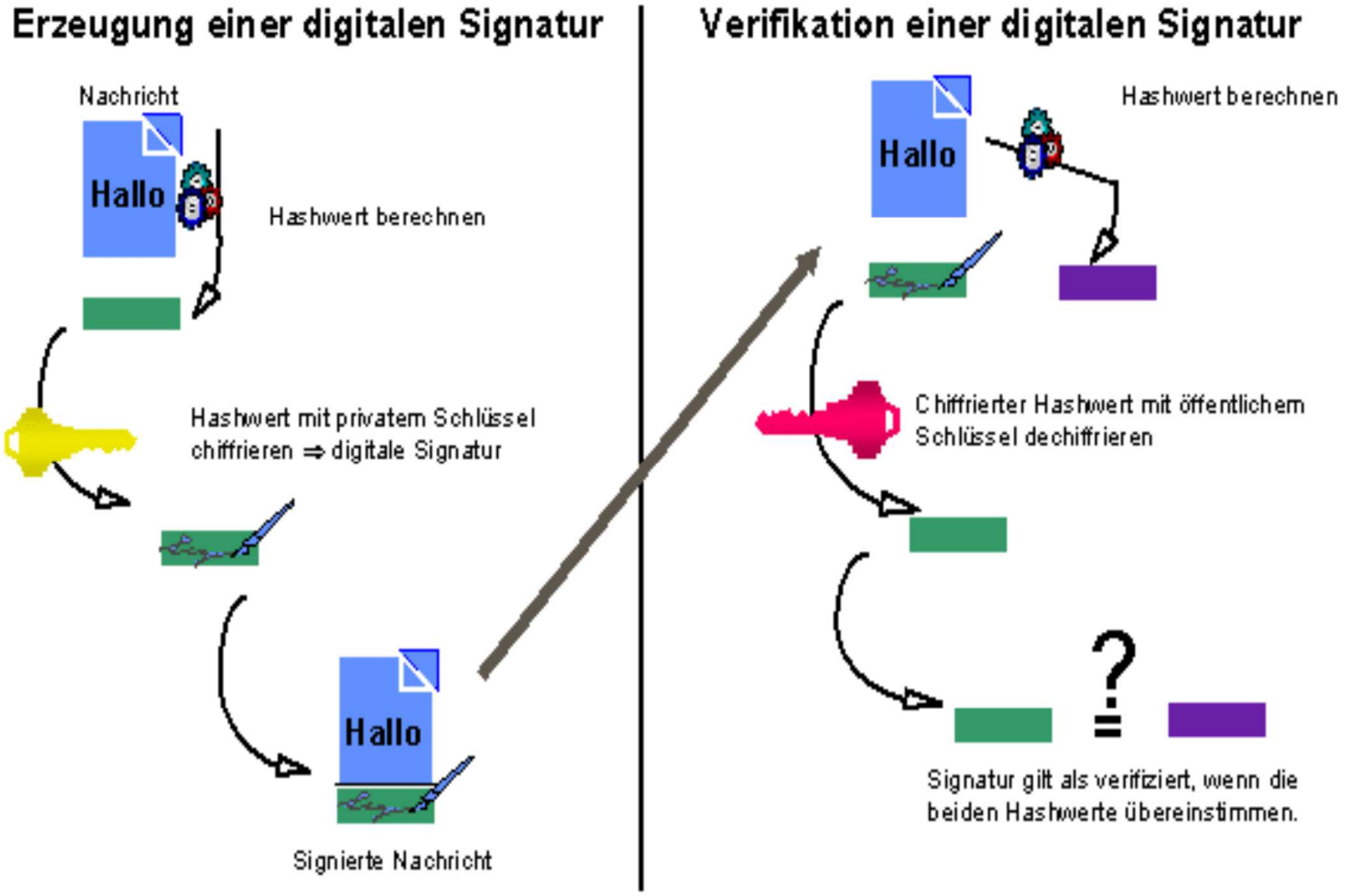
- Multiplikativität von RSA
  - $M_1^d * M_2^d \pmod n = (M_1 * M_2)^d \pmod n$
  - Aus zwei Signaturen kann durch Multiplikation ohne Schlüssel eine dritte Signatur erzeugt werden!
  - ➔ Kryptographische Hashfunktion auf das Dokument anwenden!

- Darstellungsproblem
  - Signatur-Software wird modifiziert, so dass zu signierende Dokumente falsch dargestellt werden (→ „Trojanisches Pferd“)
  - Keine generelle Gegenmaßnahme
    - Schutz vor Viren und Trojanern
    - Schlüssel und Digitale Signaturen nicht auf fremden Rechnern erzeugen lassen!

Kryptographische Hashfunktionen

- **Idee**:
  - Anstatt das komplette Dokument zu signieren, besser kurzen Hashwert („*Message Digest*“, „*Fingerprint*“) für das Dokument berechnen und nur diesen Hashwert signieren!
- **Vorteile**:
  - schneller
  - sicherer
- **Problem**:
  - Hohe „kryptographische“ Anforderungen an die Hashfunktion

Digitale Signatur eines Hashwertes



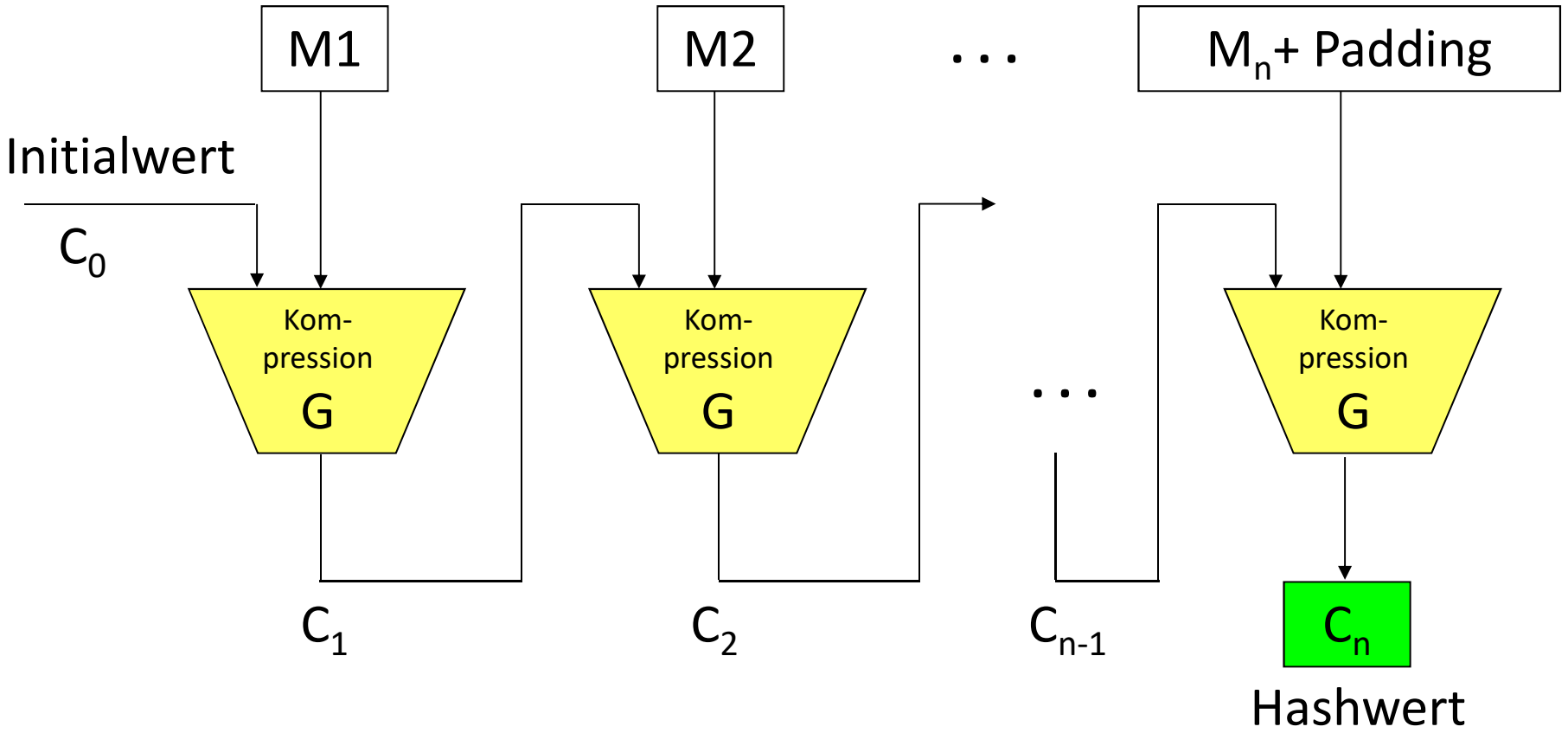
Anforderungen an kryptographische Hashfunktionen

- Allgemeine Eigenschaften
  - Nicht-injektive Funktion H: $A_1^* \rightarrow A_2^k$
  - Alle Funktionswerte haben eine feste Länge k
  - Wertebereich ist wesentlich kleiner als Definitionsbereich ("Kollisionen" sind möglich)
- Kryptographische Anforderungen
  - H ist Einwegfunktion
    - Für ein gegebenes M ist H(M) leicht zu berechnen
    - Aus gegebenem H(M) kann M nicht effizient bestimmt werden
  - Bei gegebenem M und H(M) ist es praktisch unmöglich, ein dazu passendes M' zu konstruieren, so dass H(M)=H(M')
  - Wenn es zusätzlich praktisch unmöglich ist, zwei beliebige verschiedene Eingabewerte M und M' zu finden, so dass H(M)=H(M'), heißt die Hashfunktion „starke Hashfunktion"

Prinzip kryptographischer Hashfunktionen

- Aufteilung der Eingabebitfolge M in Blöcke $M_1,..,M_n$ (ggf. Auffüllen des letzten Blocks)
- n-malige Anwendung einer „Kompressionsfunktion“ G auf jeden Block M_i und das letzte Ergebnis C_i: Ergebnis $C_{i+1} = G(M_{i+1}, C_i)$
- C_0 ist fest vorgegeben (Initialwert)
- Berechneter (Gesamt-)Hashwert ist C_n

Bild: Prinzip kryptographischer Hashfunktionen

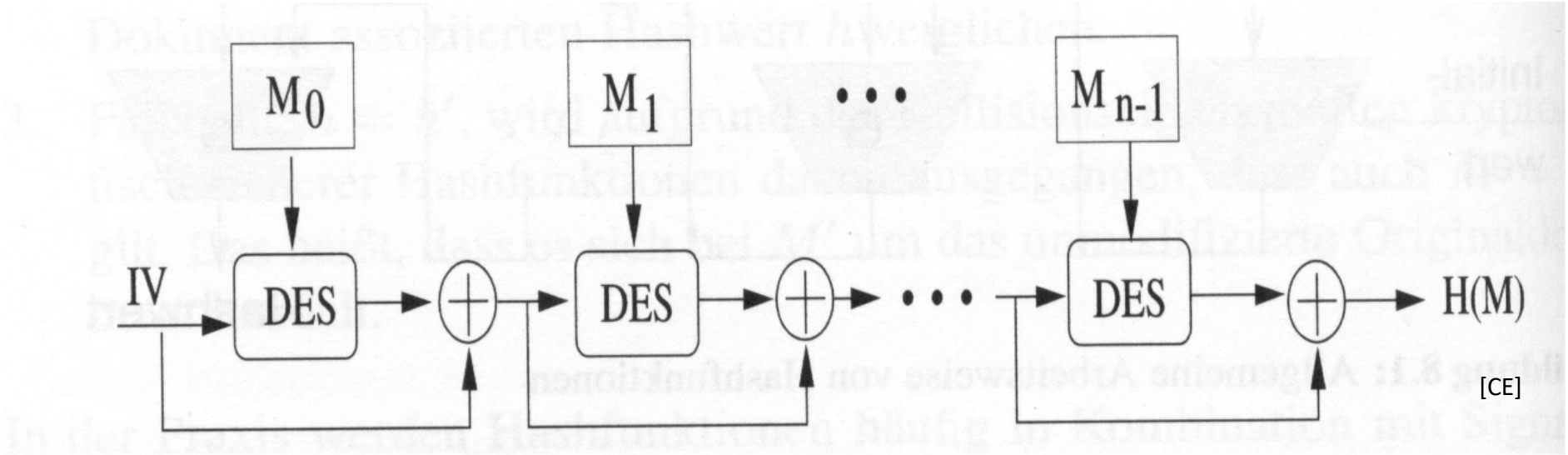


Beispiel: DES als Kompressionsfunktion G

$C_0 := IV$

$C_{i+1} := DES(C_i, M_i) \oplus C_i$ (für i = 0, …, n-1)

$H(M) := C_n$



[CE]

- M_i: 56 Bit-Blöcke als DES-Schlüssel, C_i: 64 Bit-Blöcke als DES-Eingabe/Ausgabe
- „Meyer-Hashfunktion“ mit 64 Bit-Hashwerten (schwache Hashfunktion)

Kryptographische Hashfunktionen

- MD5 *("Message Digest 5")*
  - Entwickelt von Ron Rivest 1992 (RFC 1321)
  - Blockgröße 512 Bit, Länge des Hashwerts 128 Bit
  - Die Kompressionsfunktion verwendet 4 Runden mit jeweils 16 Operationen (XOR, Linksrotation in 4 "Kettenvariablen" à 32 Bit)
  - Kollisionen wurden entdeckt (1996, 2004)
    → "schwache" kryptographische Hashfunktion!
- SHA-1 ("*Secure Hash Algorithm*")
  - Entwickelt von der NSA 1991, NIST-Standard seit 1993
  - Blockgröße 512 Bit, Länge des Hashwerts 160 Bit
  - Die Kompressionsfunktion verwendet 4 Runden mit jeweils 20 Operationen (XOR, Linksrotation in 5 "Kettenvariablen" à 32 Bit)
  - Kollisionen wurden entdeckt (Februar 2017)
    → "schwache" kryptographische Hashfunktion!

Kryptographische Hashfunktionen

- RIPEMD-160
  - Weiterentwicklung von MD4 (u.a. durch BSI)
  - Nicht so verbreitet wie SHA-1
  - Kryptographische Stärke entspricht SHA-1 *(also nicht mehr verwenden!)*
- SHA-224, SHA-256, SHA-384, SHA-512
  - Erweiterungen von SHA-1 (NIST 2002)
  - angefügte Zahl gibt jeweils die Länge des Hash-Werts (in Bit) an
  - z.T. größere Blocklänge (1024), mehr Runden, mehr Variablen
- SHA-3 (*KECCAK*)
  - Neuer Algorithmus (Gewinner des NIST-Wettbewerbs 2012)
  - Länge des Hashwerts 224 – 512 Bit

Schlüsselabhängige Hashfunktionen

- Weitere Bezeichnungen:
  - Message Authentication Code (MAC)
  - Message Integrity Code (MIC)
- Idee: Anwendung einer kryptographischen Hashfunktion mit einem zusätzlichen geheimen Schlüssel
- Der gemeinsame geheime Schlüssel muss Sender und Empfänger bekannt sein
- Der Schlüssel **K** wird vor Anwendung der Hashfunktion **H** an den Klartext **M** angehängt und geht in die Hashwertberechnung ein:

$$MAC(M,K) = H(M|K)$$

- Ziele:
  - Authentifikation des Absenders
  - Integrität der Nachricht

→ *Wieso gewährleistet?*

| bedeutet „Konkatenation“

Message Authentication Code (MAC) - Nutzen

- **Alternative zur Digitalen Signatur**
  - Der Empfänger berechnet den MAC aus M und K ebenfalls und vergleicht mit dem übermittelten MAC
  - Wenn $MAC_{Sender} = MAC_{Empfänger}$
    - Authentifikation des Senders: Nur mit Kenntnis des geheimen Schlüssels K konnte der Sender den MAC korrekt berechnen
    - Integrität der Nachricht: Kein Angreifer konnte die Nachricht verfälschen, weil dazu die Neuberechnung des korrekten MAC notwendig gewesen werden wäre!
- Vorteil gegenüber einer digitalen Signatur: schnellere Berechnung
- Nachteil gegenüber einer digitalen Signatur: keine Verbindlichkeit
  - Der Empfänger kann die Nachricht selbst erzeugt haben, da er ja auch den Schlüssel K kennt!
- *Typische Anwendung: Bestandteil von Netzwerk-Sicherheitsprotokollen (SSL/TLS, IPSec, …)*

Höhere Sicherheit durch HMAC (RFC 2104)

- Zweifache Anwendung der Hashfunktion H mit Blocklänge b Byte
- Ggf. Auffüllen des Schlüssels K auf die Länge b *(“Padding“)*
- Verwendung von zwei konstanten Bytes, die b-mal wiederholt werden:
  - opad = 00110110 00110110 00110110 ….
  - ipad = 01011100 01011100 01011100 ….
- Berechnungsvorschrift:
  HMAC(M,K) = H(K⊕opad | H(K⊕ipad | M))
- Bewertung:
  - performant
  - sicher

  → häufig eingesetzt

Zusammenfassung Kapitel 3

3. Grundlagen der Kryptographie

3.1. Einführung

3.2. Einfache Symmetrische Verfahren

3.3. Moderne symmetrische Verfahren

3.4. Asymmetrische Verfahren („Public Key“)

3.5. Digitale Signaturen und kryptographische Hashfunktionen