Kryptographie

1 Einleitung

Die Kryptographie (*kryptós*: verborgen; *gráphein*: schreiben) war lange Zeit die Disziplin, die sich mit dem Ver- und Entschlüsseln von Nachrichten befasst hat.

Mit dem Aufkommen der digitalen Kommunikation sind weitere Tätigkeitsbereiche hinzugekommen, weshalb sich die Kryptographie heute mit folgenden Aufgaben befasst:

Die Hauptaufgaben der Kryptographie

- *Vertraulichkeit (Confidentiality):* Nur berechtigte Personen sollen die Nachricht lesen können.
- *Authentifizierung (Authentication):* Der Urheber der Nachricht soll eindeutig identifizierbar sein.
- *Integrität (Integrity):* Die Nachricht soll vollständig und unverändert sein.
- *Verbindlichkeit (Non-Repudiation):* Der Empfänger kann beweisen, dass der Sender die Nachricht mit identischem Inhalt verschickt hat.

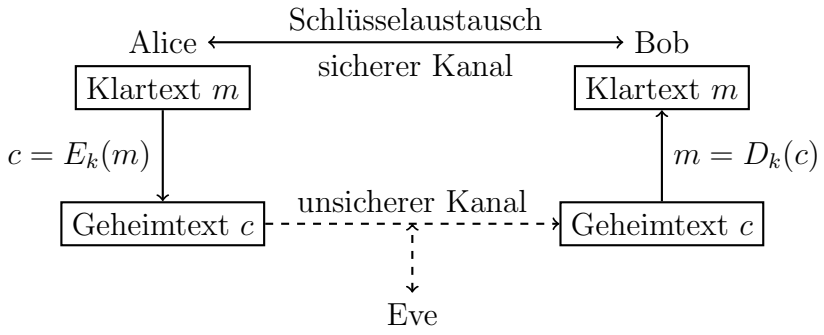
Zutaten für die Verschlüsselung

- Ein endliches Alphabet A
- Der Nachrichtenraum $M = A^*$, bestehend aus allen Nachrichten endlicher Länge über dem Alphabet A.
- Ein Verschlüsselungsverfahren, bestehend aus den Nachrichtenräumen M und C, einem Schlüsselraum K und zwei Funktionen $E_k \colon M \to C$ sowie $D_k \colon C \to M$ für die Ver- bzw. Entschlüsselung.

  E_k und D_k müssen aus nahe liegenden Gründen für jedes $m \in M$ die Bedingung $D_k(E_k(m)) = m$ erfüllen.

  Die Nachrichten $m \in M$ und $c \in C$ werden *Klartext* (*plaintext*) bzw. *Geheimtext* (*ciphertext*) genannt.

Verschlüsselte Kommunikation



Das Prinzip von Kerckhoffs

Die Sicherheit eines Verfahrens sollte nur auf der Geheimhaltung des Schlüssels und nicht auf der Geheimhaltung des Verfahrens beruhen.

Kryptoanalyse

Untersuchung eines kryptographischen Verfahrens mit dem Ziel, allfällige Schwächen zu entecken oder ihn zu brechen.

Ciphertext-only: Der Angreifer hat nur Zugriff auf einen Geheimtext oder Teile davon.

Known-plaintext: Der Angreifer besitzt einen Geheimtext und den zugehörigen Klartext oder Teile davon.

Chosen-plaintext: Der Angreifer kann die zu verschlüsselnden Klartexte frei wählen und kann daraus die entsprechenden Geheimtexte erzeugen, ohne schon den Schlüssel zu kennen.

Chosen-ciphertext: Der Angreifer kann beliebigen verschlüsselten Text in Klartext umwandeln, ohne schon den Schlüssel zu kennen.

2 Symmetrische kryptographische Verfahren

Viele klassische Verschlüsselungsverfahren haben die Eigenschaft, dass Sender und Empfänger ein gemeinsames Geheimnis teilen, das ihnen die Ver- und Entschlüsselung ermöglicht.

Ein Verschlüsselungsverfahren wird *symmetrisch* genannt, wenn das Verschlüsseln und Entschlüsseln mit demselben Schlüssel erfolgt.

Die Verschiebechiffre (Cäsar-Chiffrierung)

Das Alphabet wird um eine feste Anzahl Zeichen k zyklisch verschoben.

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

Diese Verschlüsselung ist absolut unsicher.

Python-Code

```
alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def encrypt(msg, key, alph):
    out = ''
    n = len(alph)
    for z in msg:
        out += alph[(alph.index(z)+key) % n]
    return out

def decrypt(msg, key, alph):
    return encrypt(msg, -key, alph)
```

Monoalphabetische Substitutionschiffre

Das Alphabet wird permutiert:

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

Ein Angriff mit der Brute-force-Methode ist wegen der hohen Schlüsselzahl (hier: 26^{26}) nicht durchführbar.

Die Verschlüsselung ist dennoch einfach zu brechen, weil die relativen Häufigkeiten der einzelnen Zeichen durch die Permutation erhalten bleiben.

relative Häufigkeiten der acht häufigsten Buchstaben in deutschen Texten:

E	N	I	S	R	A	T	D
17.4%	9.78%	7.55%	7.27%	7.00%	6.51%	6.15%	5.08%

Python-Code

```
alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
key  = 'HKZMBCLOWVXUNDQSJEPBRATIGF'

def encrypt(msg, key, alph):
    out = ''
    for z in msg:
        out += key[alph.index(z)]
    return out

def decrypt(msg, key, alph):
    out = ''
    for z in msg:
        out += alph[key.index(z)]
    return out
```

Das Verfahren von de Vigenère (1523–1596)

Der Index des i-ten Zeichens eines Schlüsselworts gibt an, um wie viele Positionen das i-te Klartextzeichen im Alphabet zyklisch verschoben wird. Das Schlüsselwort wird wiederholt, falls es kürzer als der Klartext ist.

```
    DASISTEINGEHEIMERTEXT
 +  KRYPTOKRYPTOKRYPTOKRY
 =  NRQXLHOZLVXVOZKTKHOOR
```

`D+K=N`, denn die Zeichen `D` und `K` stehen an den Positionen 3 und 10. Somit steht das Geheimtextzeichen an der Position 13 → `N`.

Das Vigenère-Quadrat

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

Python-Code

```
alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def encrypt(msg, key, alph):
    out = ''
    n = len(alph)
    m = len(key)
    for i in range(0, len(msg)):
        p = alph.index(msg[i])
        q = alph.index(key[i%m])
        out += alph[(p+q)%n]
    return out

c = encrypt('DASISTEINGEHEIMERTEXT', 'KRYPTO', alph)
print(c, m)
```

Ist genügend Geheimtext vorhanden, kann eine Vigenère-chiffrierte Nachricht in zwei Schritten angegriffen werden:

(a) Wenn dieselben Klartext-Zeichenfolgen mit denselben Buchstaben des Schlüsselworts verschlüsselt werden, entstehen dieselben Geheimtext-Zeichenfolgen. Aus dem ggT ihrer Abstände kann die Schlüsselwortlänge n erraten werden.

(b) Die Zeichen an den Positionen

- $0, 0+n, 0+2n, \ldots,$
- $1, 1+n, 1+2n, \ldots,$
- $2, 2+n, 2+2n, \ldots,$
- $\ldots$
- $n-1, 1\ , n-1, 2n-1, \ldots,$

sind jeweils um die gleiche Anzahl Stellen verschoben und lassen sich mit einer Häufigkeitsanlyse entschlüsseln.

One Time Pad

Es wird ein zufälliges Passwort erzeugt, das so viele Zeichen wie der Klartext hat. Damit wird wie oben zeichenweise eine monalphabetische Verschlüsselung durchgeführt, die jedoch keine Rückschlüsse auf die Schlüsselwortlänge mehr erlaubt.

Mit „guten“ Zufallszahlen und einmaligem Gebrauch des Schlüssels ist das Verfahren bewiesenermassen sicher.

Welche praktischen Problem stellen sich hier dennoch?

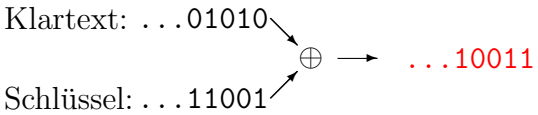
- Der Schlüssel muss auf einem sicheren Kanal transportiert werden.
- Es müssen auschreichend Schlüssel vorhanden sein.

Python-Code

```
import random

alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def randkey(msg, alph):
    n = len(alph)
    key = ''
    for c in msg:
        key += alph[random.randint(0,n-1)]
    return key

msg = "DASISTEINEGEHIMEBOTSCHAFT"
key = randkey(msg, alph)
print(key)
```

Stromchiffren

Beim One-Time-Pad-Verfahren können wir den Klartext und den Schlüssel in binärer Form darstellen und die Klartext- und Schlüssezeichen bitweise (mit XOR - ohne Übertrag) addieren:



Die Entschlüsselung funktioniert gleich: $10011 \oplus 11001 = 01010$

Diese Realisierung eines One-Time-Pad wird auch *Vernam-Chiffre* genannt.

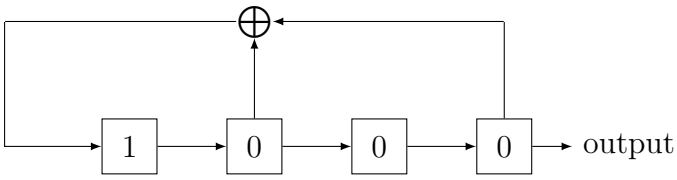
Anstatt echte Zufallszahlen zu verwenden, kann man auch eine Folge pseudozufälliger Zahlen erzeugen, die sich statistisch wie echte Zufallszahlen verhalten aber durch kürzere Schlüssel bestimmt sind. Solche Pseudozufallszahlen lassen sich durch *Schieberegister* erzeugen.

Schieberegister

Ein binäres Schieberegister der Länge n besteht aus n Zellen b_1, b_2, ..., b_n, die mit einem Startbit initialisiert sind und dann taktgesteuert jeweils eine Position nach rechts verschoben werden.

Bei jedem Takt wird das ganz rechts stehende Bit als Output ausgegeben. Die dadurch leer werdende Stelle ganz links wird durch eine XOR-Linearkombination der rückgekoppelten Zellen wieder aufgefüllt.

Beispiel



b_1	b_2	b_3	b_4	output
1	0	0	0	–
0	1	0	0	0
1	0	1	0	00
0	1	0	1	000
0	0	1	0	1000
0	0	0	1	01000
1	0	0	0	101000

Von jetzt an wiederholt sich die Folge.

Die Outputfolge eines linearen Schieberegisters wird irgendwann periodisch. Wenn das Register die Länge n hat, so ist die Periodenlänge höchstens $2^n - 1$.

Lineare Schieberegister sind in der eben dargestellten Form sehr unsicher, da man schon aus $2n$ aufeinanderfolgenden Outputbits eines linearen Schieberegisters der Länge n die gesamte Outputfolge rekonstruieren kann.

Um dennoch brauchbare Schlüsselbitgeneratoren für Stromchiffren zu erhalten, kann man unter anderem mehrere lineare Schieberegister auf *nichtlineare* Weise koppeln.

Das sich Schieberegister hardwaremässig leicht realisieren lassen und schnell arbeiten, eignen sie sich gut für Anwendungen, die Daten in Echtzeit verschlüsseln müssen (z. B. Telefongespräche).

Beispiele von Stromchiffren

- *A5/x* symmetrische Stromchiffren für GSM-Mobilfunknetze
  - A5/1: kann beinahe in Echtzeit gebrochen werden
  - A5/2: noch schwächer als A5/1
  - A5/3: Schlüssellänge von 64 Bit; mit Brute-Force angreifbar
  - A5/4: A5/3-Algorithmus mit einer Schlüssellänge von 128 Bit

Blockchiffren

Wird ein Klartextblock fester Länge durch Verschlüsselung auf einen Geheimtextblock fester Länge abgebildet, so spricht man von einer *Blockchiffre*

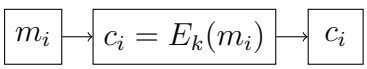
Die oben vorgestellten Verfahren waren im Grunde Blockchiffren. Wegen der Verwendung des lateinischen Alphabets betragen die Blocklängen jeweils 1 Zeichen.

Stellt man die 26 Grossbuchstaben des lateinischen Alphabets im binären Alphabet dar ($A = 00000$, $B = 00001$, $\ldots$, $Z = 11010$) so beträgt die Blocklänge $n = 5$.

Bei einer Blockchiffre besteht das Problem, dass jeweils genügend Zeichen für einen vollständigen Block vorhanden sein müssen (Stichwort: Padding), so dass dieser Chiffriertyp für Echtzeitanwendungen eher ungeeignet ist.

Electronic Code Book Mode (ECBM)

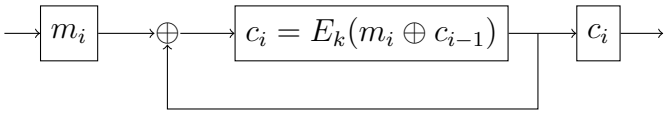
Bei dieser Betriebsart von Blockchiffren wird der Klartext in Blöcke m_1, m_2, $\ldots$ gleicher Länge unterteilt und jeder Block wir mit dem vereinbarten Schlüssel verschlüsselt.



Bei dieser Betriebsart kann man der Nachricht nicht ansehen, ob sie durch Hinzfügen, Entfernen oder Vertauschen von Blöcken verändert wurde.

Cipher Block Chaining Mode (CBCM)

Um Manipulationen am Chiffrat erkennen zu können, betreibt man Blockchiffre im *Cipher Block Chaining Mode*. Dann wird für jeder Klartextblock m_i vor der Verschlüsselung mit seinem bereits verschlüsselten Vorgänger durch ein XOR verkettet. Für die Verkettung mit dem ersten Block kann ein Zufallsblock verwendet werden,



Auf diese Weise hängt jeder Geheimtextblock von allen vorhergehenden Klartextblöcken ab und Manipulationen am Geheimtext werden so erkennbar.

Beispiele von Blockchiffren

- *Data Encryption Standard (DES)*

  Blocklänge: 64 Bit; Schlüssellänge: 56 Bits; Entwicklung in den 1970er Jahren; gilt heute nicht mehr als sicher

- *Triple-DES (3DES)*

  Erweiterung von DES; Schlüssellänge: 168 Bits

- *Advanced Encryption Standard (AES)*

  Blocklänge: 128 Bit; variablen Schlüssellängen von 128, 160, 192, 224 oder 256 Bit; in Gebrauch seit dem Jahr 2000; Anwendung: Drahtlosnetzwerke (WPA2), Secure Shell (verschlüsselte Netzwerkverbindungen zwischen entfernten Rechnern), IP-Telefonie

3 Modulare Arithmetik

Wenn zwei Zahlen a und b bei der Division durch n den gleichen Rest haben, so werden sie *kongruent modulo* n genannt. Formal:

$a \equiv b \pmod{n} \quad \Leftrightarrow \quad n$ teilt $(a - b)$ ohne Rest

Beispiel 3.12

(a) $18 \equiv 3 \pmod{5}$ [denn 5 teilt $18 - 3 = 15$]

(b) $3 \equiv 18 \pmod{5}$ [denn 5 teilt $3 - 18 = -15$]

(c) $4 \equiv 4 \pmod{7}$ [denn 7 teilt $4 - 4 = 0$]

(d) $-3 \equiv 5 \pmod{2}$ [denn 2 teilt $5 - (-3) = 8$]

Rechenregeln

Für alle c, $p \in \mathbb{Z}$ und $p > 0$ gilt:

(1) $a \equiv b \pmod{n} \quad \Rightarrow \quad a + c \equiv b + c \pmod{n}$

(2) $a \equiv b \pmod{n} \quad \Rightarrow \quad ac \equiv bc \pmod{n}$

(3) $a \equiv b \pmod{n} \quad \Rightarrow \quad a^p \equiv b^p \pmod{n}$

Restklassen

Für $n \in \mathbb{N}$ und $a \in \mathbb{Z}$ mit $0 \leq a \leq n$ ist die *Restklasse* $[a]$ (oder $\overline{a}$) definiert als die Menge aller ganzen Zahlen, die bei der Division durch n den Rest a ergeben. Für $n = 3$ gibt es drei Resklassen:

$[0] = \{\ldots, -6, -3, 0, 3, 6, \ldots\}$
$[1] = \{\ldots, -5, -2, 1, 4, 7, \ldots\}$
$[2] = \{\ldots, -4, -1, 2, 5, 8, \ldots\}$

Mit Restklassen rechnet man wie folgt. Um $[a] + [b]$ oder $[a] \cdot [b]$ zu berechnen berechnet man $s = a + b$ bzw. $p = a \cdot b$. Das Ergebnis ist dann die Restklasse, in der s bzw. p liegt.

Beispiel 3.2

Für $n = 6$ gilt:

(a) $[5] + [3] = [2]$, denn $5 + 3 = 8 \in [2]$

(b) $[2] + [4] = [0]$, denn $2 + 4 = 6 \in [0]$

(c) $[3] \cdot [3] = [3]$, denn $3 \cdot 3 = 9 \in [3]$

(d) $[2] \cdot [3] = [0]$, denn $2 \cdot 3 = 6 \in [0]$

Die Restklassenmenge $\mathbb{Z}_n$

Die Menge

$$\mathbb{Z}_n = \big\{[a] \colon a \in \mathbb{Z} \text{ und } 0 \leq a < n\big\}$$

wird Restklassenmenge modulo n genannt.

Bildet man alle möglichen Summen und Produkte in $\mathbb{Z}_n$, erhält man die Verknüpfungstafeln $(\mathbb{Z}_n, +)$ und $(\mathbb{Z}_n, \times)$.

Beispiel 3.3

$(\mathbb{Z}_5, +)$

$+$	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Eigenschaften von $(\mathbb{Z}_n, +)$

- Die Operation $+$ ist *abgeschlossen* in $\mathbb{Z}_n$.
- Die Operationen $+$ ist *assoziativ*, d. h. für alle a, b, $c \in \mathbb{Z}_n$ gilt $(a+b)+c = a+(b+c)$.
- Es gibt ein Element e, so dass für jedes $a \in \mathbb{Z}$ die Eigenschaft $a + e = e + a = a$ erfüllt ist. Ein solches Element heisst *neutrales Element*. Natürlich ist hier $e = 0$.
- Zu jedem Element $a \in \mathbb{Z}$ gibt es ein Element a' mit $a + a' = 0$. Dieses Element a' wird *inverses Element* von a genannt und in der additiven Schreibeweise mit $-a$ bezeichnet.
- Die Operationen $+$ ist *kommutativ*, d. h. für alle a, $b \in \mathbb{Z}_n$ gilt $a + b = b + a$.

Diese Eigenschaften werden nun im folgenden Begriff abstrahiert.

Gruppen

Eine *Gruppe* ist ein Paar $(G, *)$, bestehend aus einer Menge G und einer Verknüpfung $*$ mit folgenden Eigenschaften:

- Die Verknüpfung $*$ ist *abgeschlossen* in G, d. h. für zwei beliebige Elemente a, $b \in G$ gilt $a * b \in G$.
- Die Verknüpfung $*$ ist *assoziativ*, d. h. für drei beliebige Elemente a, b, $c \in G$ gilt: $a * (b * c) = (a * b) * c$.
- Es gibt ein *neutrales Element* $e \in G$, d. h. für ein beliebige Elemente $a \in G$ gilt: $a * e = e * a = a$.

- Zu jedem $a \in G$ gibt es ein *inverses Element* $a' \in G$, d. h. es gilt $a * a' = a' * a = e$.

Falls zudem alle a, $b \in G$ die Eigenschaft $a * b = b * a$ erfüllen, wird $(G, *)$ *kommutative* oder *abelsche* Gruppe genannt.

Beispiel 3.4

$(\mathbb{Z}_4, +)$ ist eine kommutative Gruppe.

$+$	**0**	**1**	**2**	**3**
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

- Die Addition ist abgeschlossen in $\mathbb{Z}_4$.
- Die Addition ist assoziativ und kommutativ.
- Das neutrale Element der Addition ist 0.
- Zu jedem $a \in \mathbb{Z}_4$ gibt es ein $a' \in \mathbb{Z}_4$ mit $a + a' = 0$.

Beispiel 3.5

$(\mathbb{Z}_4, \cdot)$ ist keine Gruppe.

$\times$	**0**	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

- Die Multiplikation ist abgeschlossen in $\mathbb{Z}_4$.
- Die Multiplikation ist assoziativ und kommutativ.
- Das neutrale Element der Multiplikation ist 1.
- Zu 0 und 2 $\in \mathbb{Z}_4$ gibt es kein multiplikatives Inverses.

Prime Restklassengruppe

Wählt man aus den Elementen von $(\mathbb{Z}_n, \times)$ diejenigen aus, die ein Inverses haben, so erhält man eine multiplikative Gruppe. Dies ist die die *Gruppe der primen Restklassen* oder *prime Restklassengruppe* und wird mit $\mathbb{Z}_n^*$ bezeichnet.

Die Elemente aus $\mathbb{Z}_n$, die ein multiplikatives Inverses haben, sind gerade die Elemente, die teilerfremd zu n sind.

Beispiel 3.6

$\mathbb{Z}_{10}^*$

$\times$	**1**	**3**	**7**	**9**
1	1	3	7	9
3	3	9	1	7
7	7	1	9	3
9	9	7	3	1

Beispiel 3.7

$\mathbb{Z}_5^*$

$\times$	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Ist p eine Primzahl, so ist jede ganze Zahl 1, 2, …, $p-1$ teilerfremd zu p und somit ist $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$.

Zyklische Gruppen

Eine Gruppe $(G, *)$ heisst *zyklisch*, wenn alle Elemente von G (in multiplikativer Schreibweise) als Potenz eines geeigneten Elements $a \in G$ dargestellt werden können, d. h. wenn $G = \{a^n : n \in \mathbb{Z}\}$ gilt.

Beispiel 3.8

$\mathbb{Z}_{10}^*$:

- $1^0 = 1$, $1^1 = 1$, … (nicht erzeugend)
- $3^0 = 1$, $3^1 = 3$, $3^2 = 9$, $3^3 = 7$, $3^4 = 1$, … (erzeugend)
- $7^0 = 1$, $7^1 = 7$, $7^2 = 9$, $7^3 = 3$, $7^4 = 1$, … (erzeugend)
- $9^0 = 1$, $9^1 = 9$, $9^2 = 1$, … (nicht erzeugend)

Beispiel 3.9

$\mathbb{Z}_5^*$:

- $1^0 = 1$, $1^1 = 1$, … (nicht erzeugend)

- $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 3$, $2^4 = 1$, ... (erzeugend)
- $3^0 = 1$, $3^1 = 3$, $3^2 = 4$, $3^3 = 2$, $3^4 = 1$, ... (erzeugend)
- $4^0 = 1$, $4^1 = 4$, $4^2 = 1$, ... (nicht erzeugend)

Die schlechte Nachricht

Ist p eine Primzahl, so ist nicht jedes Element in $\mathbb{Z}_p^*$ erzeugend.

Die gute Nachricht

Ist p eine Primzahl, so gibt es mindestens ein Element $g \in \mathbb{Z}_p^*$, das erzeugend ist.

In diesem Fall nennt man g *Primitivwurzel* zum Modul p.

Die eulersche φ-Funktion

Zwei natürliche Zahlen a und b sind *teilerfremd*, wenn ihr grösster gemeinsamer Teiler 1 ist.

Für eine natürliche Zahl n definiert $\varphi(n)$ die Anzahl der zu n teilerfremden natürliche Zahlen, die nicht grösser als n sind.

n	1	2	3	4	5	6	7	8	9	10	11	12
$\varphi(n)$	1	1	2	2	4	2	6	4	6	4	10	4

Satz

Ist p eine Primzahl und $\mathbb{Z}_p^*$ die zugehörige Gruppe der primen Restklassen, so gibt es $\varphi(p-1)$ erzeugende Elemente (Primitivwurzeln) in $\mathbb{Z}_p^*$.

Beispiel 3.10

In $\mathbb{Z}_5^*$ gibt es folglich $\varphi(5-1) = \varphi(4) = 2$ erzeugende Elemente, was das Beispiel 3. 8 bestätigt.

Der obige Satz sagt aber nicht, wie diese Primitivwurzeln gefunden werden können.

Man kann jedoch für jedes Element $g \in \mathbb{Z}_5^*$ überprüfen, ob es erzeugend ist; d. h. ob $g^i \neq 1$ für $1 \leq i \leq p-2$ und $g^{p-1} = 1$ gilt.

Falls die Primzahlzerlegung von $p-1 = p_1 \cdot p_2 \cdot \ldots \cdot p_k$ bekannt ist, kann man sich bei der Überprüfung auf die Fälle $g^{(p-1)/p_i} \neq 1$ für $i = 1, 2, \ldots k$ beschränken.

Ist $\mathbb{Z}_p^*$ die prime Restklassengruppe für eine Primzahl p und g ein Primitivwurzel in $\mathbb{Z}_p^*$, so hat die Gleichung

$$g^x \mod p = b$$

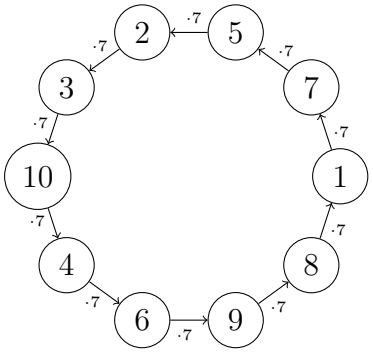
für jedes $b \in \mathbb{Z}_p^*$ eine Lösung.

Die dadurch eindeutig bestimmte Zahl x wird *diskreter Logarithmus* genannt.

Die Berechnung des diskreten Logarithmus' ist für grosse Primzahlen p, ein erzeugendes Element g und einen zufällig gewählten Exponenten x eine im Allgemeinen nicht effizient lösbare Aufgabe.

Beispiel 3.11

Das Element $g = 7$ erzeugt $\mathbb{Z}_{11}^*$:

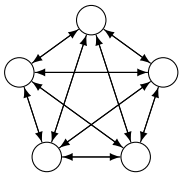


4 Der Diffie-Hellman-Merkle Schlüsselaustausch

Einleitung

Ein Problem, dass sich bei symmetrischen Verschlüsselungsverfahren stellt ist, dass der Absender einer verschlüsselten Nachricht jedem berechtigten Empfänger den Schlüssel (auf einem sicheren Kanal) zukommen lassen muss.

Wie viele Schlüssel sind nötig bzw. müssen ausgetauscht werden, wenn n Teilnehmer miteinander geheim kommunizieren wollen, und jeweils zwei Teilnehmer einen eigenen geheimen Schlüssel teilen?



$$\frac{n(n-1)}{2}$$

Public-Key-Kryptographie

Im Jahr 1976 stellten die Kryptographen W. Diffie und M. E. Hellman folgende provokante Frage:

> Kann man jemandem spontan eine geheime Nachricht schicken, ohne dass man zuvor einen gemeinsamen Schlüssel vereinbart hat?

Diffie und Hellman haben sich dieser Frage angenommen und herausgefunden, dass prinzipiell eine geheime Kommunikation *ohne einen gemeinsamen Schlüssel* möglich ist.

Das Verfahren

1. Alice und Bob einigen sich „öffentlich“ auf eine gemeinsame grosse Primzahl p und ein erzeugendes Element g von $\mathbb{Z}_p*$.
2. Alice und Bob wählen jeweils eine geheime „gute“ Zufallszahl a bzw. b aus der Menge $\{1, 2, \ldots, p-1\}$.
3. Alice berechnet $A = g^a \mod p$ und sendet diese Zahl an Bob. Umgekehrt berechnet Bob $B = g^b \mod p$ und sendet diese Zahl an Alice.
4. Alice berechnet für sich $K_a = B^a = (g^b)^a = g^{ab}$ und Bob für sich $K_b = A^b = (g^a)^b = g^{ba}$. Da die Multiplikation modulo p kommutativ ist, gilt $K_a = g^{ba} = g^{ab} = K_b$.

   Nun können beide Teilnehmer ihre Kommunikation verschlüsseln, ohne vorher einen geheimen Schlüssel ausgetauscht zu haben.

Um den gemeinsamen Sitzungsschlüssel zu erlangen, müsste ein Angreifer aus den abgehörten Zahlen p, g, A und B die modularen Gleichungen

$$g^x = A \mod p \quad \text{und} \quad g^y = B \mod p$$

nach x und y lösen. Dies ist jedoch nach heutigem Wissensstand für ausreichend grosse Primzahlen p und gut gewählte Exponenten nicht innert vernünftiger Zeit möglich.

Beispiel 4.1

1. Alice und Bob einigen sich auf $p = 11$ und $g = 7$.

2. Alice wählt als geheimen Exponenten $a = 8$ und Bob als geheimen Exponenten $b = 4$.

3. Alice sendet $A = 7^8 \mod 11 = 9$ an Bob.

   Bob sendet $B = 7^4 \mod 11 = 3$ an Alice.

4. Alice berechnet $K_a = B^a \mod p = 3^8 \mod 11 = 5$.

   Bob berechnet $K_b = A^b \mod p = 9^4 \mod 11 = 5$.

Alice und Bob können nun ein symmetrisches kryptografisches Verfahren mit dem gemeinsamen Schlüssel $k = 5$ verwenden.

Sicherheit

Die Primzahl p sollte möglichst gross sein. Derzeit (2018) wird eine Primzahl mit mindestens 3000 Bit (etwa 900 Dezimalstellen) empfohlen.

Darüber hinaus sollte vermieden werden, dass $p - 1$ nur kleine Primfaktoren hat, was einen Angriff erleichtern würde.

Als Basis g kann eine Primitivwurzel (erzeugendes Element) verwendet werden. Als Alternative kann auch ein Element g verwendet werden, das eine Untergruppe von $\mathbb{Z}_p^*$ mit einer hinreichend grossen Primzahlordnung q erzeugt.

Es gibt noch weitere spezielle Einschränkungen, die jedoch anspruchsvolleres zahlentheoretisches Wissen verlangen, um sie zu verstehen.

Man-in-the-Middle-Attack

Die folgende Angriffsmethode ist dann wirkungsvoll, wenn Alice und Bob ihre Identität nicht gegenseitig überprüfen können.

Wenn es der Angreiferin Eve gelingt, den gesamten Datenverkehr zwischen Alice und Bob über sich zu leiten, ist folgendes Szenario möglich:

Nachdem Alice und Bob sich auf eine Primzahl p und eine Zahl g geeinigt haben, fängt Eve die Zahlen $A = g^a \mod p$ und $B = g^b \mod p$ ab und berechnet mit einer von ihr gewählten Zahl z den Wert $E = g^e \mod p$ und leitet ihn sowohl an Alice als auch an Bob weiter.

Wenn Alice eine Nachricht an Bob sendet, verwendet sie den Schlüssel $K_{AE} = \left(g^e\right)^a \mod p = \left(g^a\right)^e \mod p$ den auch Eve kennt. Eve entschlüsselt damit die Nachricht, verschlüsselt sie mit dem Schlüssel $K_{BE} = \left(g^e\right)^b \mod p = \left(g^b\right)^e \mod p$ und sendet sie an Bob. In der umgekehrten Richtung geht Eve analog vor.

Auf diese Weise kann Eve die gesamte Kommunikation zwischen Alice und Bob abhören. Sie kann sogar Nachrichten verfälschen oder den Teilnehmenden neue Nachrichten zukommen lassen.

Daher ist es nötig, dass Alice und Bob sich vorgängig authentifzieren, d. h. sicherstellen, dass sie unmittelbar mit der jeweils anderen Person kommunzieren.

5 Das RSA-Verfahren

1977 haben Rivest, Shamir und Adleman folgendes Verschlüsselungssystem vorgeschlagen, dessen Sicherheit auf abstrakter Algebra und auf der Schwierigkeit basiert, grosse ganze Zahlen zu faktorisieren.

Beim RSA-Verfahren hat jeder Teilnehmer einen öffentlichen Schlüssel e und einen geheimen Schlüssel d.

Dieses Schlüssel-Paar $k = (e, d)$ hat folgende Eigenschaften:

- *Public-Key-Eigenschaft*: Es ist praktisch unmöglich, aus dem öffentlichen Schlüssel e den privaten Schlüssel d zu berechnen.
- *Entschlüsselungseigenschaft:* Für jede Nachricht m gilt: $D_d(E_e(m)) = m$
- *Signatureigenschaft:* Für jede Nachricht m gilt: $E_e(D_d(m)) = m$

Das Verfahren läuft so ab, dass sich Alice den öffentlichen Schlüssel e von Bob beschafft und damit die Nachricht m verschlüsselt, indem sie $c = E_e(m)$ bildet.

Die verschlüsselte Nachricht c sendet sie an Bob.

Bob wendet seinen privaten Schlüssel d auf den verschlüsselten Text c an und erhält so $D_d(c) = D_d(E_e(m)) = m$.

Authentizität

Wie kann Bob überprüfen, ob eine Nachricht auch tatsächlich von Alice stammt und nicht von Eve, die sich für Alice ausgibt?

Wenn nun Alice beweisen möchte, dass sie die Absenderin einer Nachricht m ist, dann verschlüsselt sie diese Nachricht mit ihrem privaten Schlüssel: $sig = D_d(m)$. Anschliessend veröffentlicht sie m zusammen mit sig und ihrem öffentlichen Schlüssel e.

Wer die Autentizität der Nachricht verifzieren will, muss Alices' öffentlichen Schlüssel e auf sig anwenden und prüfen, ob sich dabei m ergibt, d. h. ob $E_e(sig) = E_e(D_d(m)) = m$ gilt.

Die Realisierung

Rivest, Shamir und Adleman haben ausgenützt, dass das Multiplizieren zweier Primzahlen eine Operation ist, deren Aufwand quadratisch mit der Anzahl der Stellen dieser Primzahlen wächst.

Ist umgekehrt das Produkt aus zwei grossen unbekannten Primzahlen gegeben, so ist es im Allgemeinen schwierig, die Zerlegung der Zahl in die beiden Faktoren innert nützlicher Zeit zu finden. Konzeptionell müssen alle Möglichkeiten durchprobiert werden und dafür wächst der Aufwand exponentiell mit der Anzahl der Ziffern der zu zerlegenden Zahl.

Bis heute konnte noch niemand eine Verfahren angeben, mit dem diese Aufgabe um Grössenordnungen schneller erledigt werden kann – zumindest nicht für herkömmliche Computer.

Modulares Potenzieren

Aufgabe: Berechne $11^{5\,210\,768} \mod 13$

(a)
```
11**5210768 % 13
```

(b)
```
def modExpNaive(base, exponent, mod):
    result = 1
    for i in range(0, exponent):
        result = (result*base) % mod
    return result

print(modExpNaive(11, 5210768, 13))
```

(c)
```
def modExp(base, exp, mod):
    if exp == 0:
        return 1
    t = modExp((base*base) % mod, exp//2, mod)
    if exp%2 != 0:
        t = (t * base) % mod
    return t

print(modExp(11, 5210768, 13))
```

Der euklidsche Algorithmus

Ein Verfahren, um den grössten gemeinsamen Teiler zweier natürlicher Zahlen a und b zu berechnen:

setze $a_0 = a$ und $b_0 = b$ und berechne mit

$a_{i+1} = b_i$
$b_{i+1} = a_i \mod b_i$

schrittweise (a_1, b_1), (a_2, b_2), $\ldots$, (a_n, b_n) bis $b_n = 0$.

Dann ist $a_n = \mathrm{ggT}(a, b)$

Beispiel 5.1

a	b
39	21
21	18
18	3
3	0

Also ist 3 ein Teiler von 3 und 18

und damit auch ein Teiler von 18 und 21

und damit auch ein Teiler von 21 und 39

Python Code

```
# rekursive Funktion zur Berechung des ggTs:
def ggt(a, b):
    if b == 0:
        return a
    else:
        return ggt(b, a%b)
```

Der erweiterte euklidsche Algorithmus

Für a, $b \in \mathbb{N}$ und $d = \text{ggT}(a, b)$ ist es immer möglich, zwei Zahlen x, $y \in \mathbb{Z}$ zu finden, welche die Gleichung $d = x \cdot a + y \cdot b$ erfüllen.

Durch die Analyse eines ggT-Schritts lässt sich ein rekursiver Algorithmus zur Berechnung von x und y finden:

$$\begin{aligned} \text{ggT}(a, b) &= x \cdot a + y \cdot b \\ \text{ggT}(b, a\%b) &= x' \cdot b + y' \cdot (a\%b) \\ &= x' \cdot b + y' \cdot \big(a - \lfloor a/b \rfloor b\big) \\ &= y' \cdot a + \big(x' - \lfloor a/b \rfloor \cdot y'\big) \cdot b \end{aligned}$$

Also erhalten wir die neuen Koeffzienten x und y durch

$x = y'$ und $y = \big(x' - \lfloor a/b \rfloor \cdot y'\big)$

Beispiel 5.2

$x = y'$ und $y = \big(x' - \lfloor a/b \rfloor \cdot y'\big)$

a	b	$\lfloor a/b \rfloor$	x	y
39	21	1	-1	$1 - \lfloor 39/21 \rfloor \cdot (-1) = 2$
21	18	1	1	$0 - \lfloor 21/8 \rfloor \cdot 1 = -1$
18	3	6	0	$1 - \lfloor 18/3 \rfloor \cdot 0 = 1$
3	0		1	0

Also: $3 = (-1) \cdot 39 + 2 \cdot 21$

Python Code

```
# rekursive Funktion zur Berechung des erweiterten ggTs:
def egcd(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        (d, x, y) = egcd(b, a%b)
        return (d, y, x - a//b*y)
```

Satz von Euler-Fermat

Sind a und n zwei teilerfremde Zahlen, so gilt

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

zur Erinnerung: $\varphi(n)$ ist die Funktion, welche jeder natürlichen Zahl n die Anzahl der zu ihr teilerfremden Zahlen $k \leq n$ zuordnet. Teilefremd zu n sind alle Zahlen k mit $ggt(n, k) = 1$.

Beispiel 5.3

(a) $2^{\varphi(5)} \equiv 2^4 \equiv 16 \equiv 1 \mod 5$

(b) $7^{\varphi(3)} \equiv 7^2 \equiv 49 \equiv 1 \mod 3$

(c) $3^{\varphi(4)} \equiv 3^2 \equiv 9 \equiv 1 \mod 4$

(d) $4^{\varphi(9)} \equiv 4^4 \equiv 64 \equiv 1 \mod 9$

Beweis

Die Elemente $a \in \mathbb{Z}_n$ mit $\text{ggT}(a, n) = 1$ bilden $\mathbb{Z}_n^* \subset \mathbb{Z}_n$.

Als Beispiel $\mathbb{Z}_8^*$:

$\times$	**1**	**3**	**5**	**7**
1	1	3	5	7
3	3	1	7	5
5	5	7	1	3
7	7	5	3	1

Für ein $a \in \mathbb{Z}_n^*$ gilt $a \cdot \mathbb{Z}_n^* = \mathbb{Z}_n^*$.

Das bedeutet: Die Multiplikation der Elemente von $\mathbb{Z}_n^*$ mit einem beliebigen Element $a \in \mathbb{Z}_n^*$ ergibt wieder, bis auf eine eventuelle Vertauschung der Elemente, dieselbe Menge.

Denn nimmt man an, das Produkt von a mit zwei verschiedenen Restklassen x, $y \in \mathbb{Z}_n^*$ wäre identisch

$$ax \equiv ay \mod n,$$

so kann man beide Seiten mit a^{-1} multiplizieren ($\mathbb{Z}_n^*$ besteht schliesslich aus der Menge aller invertierbaren Restklassen). Dies aber ergibt

$$x \equiv y \mod n,$$

was im Widerspruch zur Annahme steht.

Für eine Aufzählung $x_1, x_2, \ldots, x_{\varphi(n)}$ aller Elemente von $\mathbb{Z}_n^*$ gilt:

$$ax_1 \cdot ax_2 \cdot \ldots \cdot ax_{\varphi(n)} \equiv x_1 \cdot x_2 \cdot \ldots \cdot x_{\varphi(n)} \mod n$$

$$a^{\varphi(n)} x_1 \cdot x_2 \cdot \ldots \cdot x_{\varphi(n)} \equiv x_1 \cdot x_2 \cdot \ldots \cdot x_{\varphi(n)} \mod n$$

Mulitpliziert man die Gleichung mit allen multiplikativen Inversen $x_1^{-1}, x_2^{-1}, \ldots, x_{\varphi(n)}^{-1}$, so erhält man $a^{\varphi(n)} \equiv 1 \mod n$

□

Die Schlüsselerzeugung

(1) Wähle zwei sehr grosse verschiedene Primzahlen p und q.

(2) Berechne $n = pq$. (für das Ver- und Entschlüsseln)

(3) Berechne $\varphi(n) = (p-1)(q-1)$.

(4) Wähle eine natürliche Zahl e mit $\text{ggT}(e, \varphi(n)) = 1$.

(5) Berechne mit dem erweiterten euklidschen Algorithmus die multiplikative Inverse von e modulo $\varphi(n)$, d. h. die Zahl d mit $e \cdot d \mod \varphi(n) = 1$.

Das Zahlenpaar (e, n) bildet den öffentlichen Schlüssel.

Das Zahlenpaar (d, n) bildet den privaten Schlüssel.

Die Primzahlen p und q können gelöscht werden; auf keinen Fall dürfen sie publik gemacht werden.

Das Verschlüsseln

Für das Verschlüsseln muss die Nachricht m in Form einer Zahl $m < n$ mit $\text{ggT}(m, n) = 1$ dargestellt werden. Nachrichten mit $m \geq n$ müssen in kleinere Blöcke $m = m_1 m_2 \dots m_k$ zerlegt werden.

Verschlüsseln: $m^e \mod n = c$

Entschlüsseln: $c^d \mod n = m$

Beweis der Entschlüsselungseigenschaft:

$$c^d = \left(m^e\right)^d \mod n = m^{ed} \mod n \overset{(5)}{=} m^{1+k\cdot\varphi(n)} \mod n$$

$$= m \cdot m^{k\varphi(n)} \mod n = m \cdot \left(m^{\varphi(n)}\right)^k \mod n$$

$$\overset{*}{=} m \cdot 1^k \mod n = m$$

* Satz von Euler-Fermat

Beispiel 5.4

Primzahlen: $p = 3$, $q = 11$

öffentlicher Schlüssel: $e = 3$

Nachricht: $m = 4$

$n = p \cdot q = 3 \cdot 11 = 33$

$\varphi(n) = \varphi(33) = (3-1)(11-1) = 20$

erweiterter euklidscher Algorithmus:

$\varphi(n)$	e	$\lfloor \varphi(n)/e \rfloor$	x	y
20	3	6	−1	7
3	2	1	1	−1
2	1	2	0	1
1	0	-	1	0

Also gilt: $1 = (-1 \cdot 20 + 7 \cdot 3) \mod 20 = 7 \cdot 3$

folglich ist $d = 7$ die multiplikative Inverse von $e = 3$ modulo $\varphi(33) = 20$ und damit der private Schlüssel.

Verschlüsselung: $c = E_e(4) = 4^3 \mod 33 = 64 \mod 33 = 31$

Entschlüsselung: $c = D_d(31) = 31^7 \mod 33 = \dots$

Square-and-Multiply:

$7 = 2 \cdot 3 + 1 \qquad 31 \cdot 25^2 = 4$
$3 = 2 \cdot 1 + 1 \qquad 31 \cdot 31^2 = 25$
$1 = 2 \cdot 0 + 1 \qquad 31 \cdot 1^2 = 31$

$\dots = 4 = m$ (stimmt!)

6 Kryptographische Hash-Funktionen

Eine *kryptographische Hashfunktion* (Streuwertfunktion) bilden einen Eingabestring x beliebiger Länge auf einen Ausgabestring $y = h(x)$ fester Länge ab.

Man unterscheidet

- *schlüssellose* Hashfunktionen mit $y = h(x)$
- *schlüsselabhängige* Hashfunktionen mit $y = h_k(x)$, die neben dem Eingabestring x noch einen geheimen Schlüssel k als Eingabe benötigen.

Schlüssellose Hashfunktionen erfüllen idealerweise folgende Bedingungen:

(a) *Einweg-Eigenschaft:* Es ist praktisch unmöglich, zu einem gegebenen Ausgabewert y den Eingabewert x mit $h(x) = y$ zu finden.

(b) *Schwache Kollisionsresistenz:* Es ist praktisch unmöglich, zu einem x und dem zugehörigen $y = h(x)$ ein zweites, von x verschiedenes x' zu finden, mit $h(x') = y$.

(c) *Starke Kollisionsresistenz:* Es ist praktisch unmöglich, zwei verschiedene Eingaben x und x' zu finden, mit $h(x) = h(x') = y$.

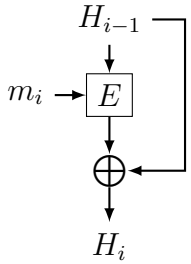
Eine schlüssellose Hashfunktion, die (a) und (b) erfüllt, wird *Einweg-Hashfunktion* genannt; erfüllt sich auch (c), so heisst sie *kollisionsresistente Hashfunktion*.

Kompressionsfunktionen

Die folgenden zwei Verfahren komprimieren einen Text m_i der Länge a und einen bereits komprimierten Block H_{i-1} der Länge b zu einem komprimierten Block H_i der Länge b. Aus Gründen, die gleich klar werden, setzen sie einen festen Initialiserungsblock H_0 der Länge b voraus.

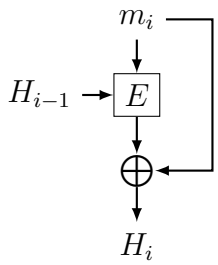
Davies-Meyer-Kompressionsfunktion

Ein Nachrichtenblock m_i der festen Länge a dient als Schlüssel bei einer Blockverschüsselung, in der das Resultat der vorangehenden Kompression H_{i-1} verschlüsselt wird. Danach wird der verschlüsselte Text mit der Eingabe H_{i-1} zu einem neuen Block H_i verknüpft (oft durch bitweises XOR).



Matyas-Meyer-Oseas-Kompressionsfunktion

Das Resultat der vorangehenden Kompression H_{i-1} dient als Schlüssel bei einer Blockverschüsselung, die den Nachrichtenblock m_i verschlüsselt. Danach wird der verschlüsselte Text mit der Eingabe m_i zu einem neuen Block H_i verknüpft (oft durch bitweises XOR).



Merkle-Damgård-Konstruktion

Von Ralph Merkle und Ivan Damgård stammt folgendes Konzept, mit dem aus einer kollisionssicheren Kompressionsfunktion

$f\colon \{0,1\}^{a+b} \to \{0,1\}^b$

eine kollisionssichere Hashfunktion

$h\colon \{0,1\}^* \to \{0,1\}^b$

konstruiert werden kann, die Zeichenketten beliebiger Länge auf einen Hashwert der Länge b abbildet.

Gegeben: Nachricht m, ein Initialisierungsvektor H_0 der Länge b (oft b Nullen) und eine kollisionsresistente Kompressionsfunktion f

1. Die Nachricht m wird mit Füllzeichen zu $\overline{m}$ erweitert, so dass die Länge von $\overline{m}$ ein Vielfaches von a ist (*Padding*).

2. $\overline{m}$ wird in n Blöcke der Länge a aufgeteilt:

   $\overline{m} = m_1|m_2|\ldots|m_n$

3. Die Kompressionsfunktion f wird auf H_0 und m_1 angewendet. Daraus erhält man die Ausgabe H_1, die f mit dem nächsten Block m_2 zu H_2 komprimiert. Dieser Vorgang wird fortgesetzt, bis alle Nachrichtenblöcke verarbeitet sind.

   $H_1 = f(H_0, m_1)$
   $H_2 = f(H_1, m_2)$
   $\ldots$
   $H_n = f(H_{n-1}, m_n)$

Implementationen

- MD5 (Message Digest Algorithm): Gilt nicht mehr als sicher.

- SHA-1 (Secure Hash Alogrithm): Gilt nicht mehr als sicher. Im Februar 2017 haben konnten Google-Mitarbeiter zwei unterschiedliche PDF-Dateien mit derselben SHA-1-Prüfsumme erzeugen.

- SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512): Nachfolger von SHA-1 mit unterschiedlichen Schlüssel- und Blocklängen. Gelten derzeit (2018) noch als sicher.

- SHA-3 (Keccak): Seit 2015 Nachfolger von SHA-2. Verwendet anstelle der Merkle-Damgård-Konstruktion eine „Sponge“-Konstruktion. In Python ab Version 3.6 verfügbar

Beispiel

```
import hashlib

# SHA-256 ist eines von 4 Verfahren des SHA-2-Standards
# und gilt noch als sicher (2018)

y1 = hashlib.sha256(b"Hello World")
y2 = hashlib.sha256(b"Hello World!")
print(y1.hexdigest());
print(y2.hexdigest());

print(y1.digest_size);
```