9 Kryptographie

9.1 Einführung

Literatur

A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter
Moderne Verfahren der Kryptographie,
Vieweg+Teubner, Mai 2010.

C. Damm
Kryptographie,
Skript zur Vorlesung, 2004.

B. Schneier, N. Ferguson
Practical Cryptography,
John Wiley & Sons, 2003.

A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone
Handbook of Applied Cryptography,
Crc Press, Oktober 1996.

Geheimhaltung

Wie kann man mit jemanden vertraulich kommunizieren, d.h. kein Unbeteiligter soll Kenntnis von der übermittelten Nachricht erhalten?

Das Problem der Übermittlung (und Speicherung) geheimer Nachrichten kann man durch verschiedene Maßnahmen lösen.

- Organisatorische Maßnahmen
- Physikalische Maßnahmen
- Kryptographische Maßnahmen

Organisatorische Maßnahmen

Beispiel

- Ein Gespräch während eines einsamen Waldspaziergangs.

- Übermittlung einer Nachricht durch einen vertrauenswürdigen Boten.
- Einstufung vertraulicher Dokumente als Verschlusssache.

Physikalische Maßnahmen

Beispiel

- Verstecken der Informationen in einem Tresor.
- Übermitteln der Nachricht in einem versiegelten Brief.
- Verheimlichen der Existenz der Nachricht, z.B. durch Geheimtinte.

Kryptographische Maßnahmen

Kryptographische Maßnahmen verändern bzw. entstellen (**verschlüsseln**, **chiffrieren**, *encrypt*) die Nachricht bzw. den **Klartext** (*plaintext*).

Dadurch ist die Nachricht für einen Außenstehenden nicht mehr erkennbar und die übertragene Information, der **Geheimtext** (*ciphertext*), erscheint diesem (meist) völlig unsinnig.

Ein berechtigter Empfänger kann den Klartext aber (leicht) wieder herstellen (**entschlüsseln**, **dechiffrieren**, *decrypt*).

Der älteste Zweig der klassischen Kryptographie beschäftigt sich mit der **Geheimhaltung** von Nachrichten **durch Verschlüsselung**.

Codierung

Klassische Verschlüsselungsalgorithmus arbeiten meistens mit Nachrichten einer natürlichen, d.h. von Menschen gesprochenen, Sprache.

Diese Nachrichten enthalten Formatierungen (z.B. Groß- und Kleinschreibung) und zusätzliche Zeichen (z.B. Interpunktionszeichen), die für das Verständnis der Nachricht nicht notwendig sind.

Unter der **Codierung** (*encoding*) einer Nachricht versteht man ihre Umwandlung von einem Alphabet in ein anderes Alphabet.

Der Aufwand für die Ver- und Entschlüsselung steigt mit der Größe des Alphabets, deshalb wird bei den meisten klassischen Verschlüsselungsalgorithmus die Nachricht zu einem Klartext über einem kleineren Alphabet codiert.

Bemerkung

Moderne Verschlüsselungsalgorithmen arbeiten mit Bit-Blöcken/Streams. Digitale Daten sind i.d.R. schon in dieser Form codiert. Z.B. ergibt sich die zu einem Text gehörige Bitfolge, aus der verwendeten Zeichencodierung (ASCII, UTF-8, etc.).

Beispiel

Die Funktion **encode** wandelt alle Buchstaben in Großbuchstaben um und entfernt alle nicht in **alphabet** enthaltenden Zeichen.

```
1 alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3 def encode(text):
4     e = ""
5     for t in text:
6         if (t.upper() in alphabet):
7             e = e + t.upper()
8     return e
```

Die Nachricht

```
XV. Maerz Caesar treffen, Dolche nicht vergessen
```

wird so zum Klartext

```
XVMAERZCAESARTREFFENDOLCHENICHTVERGESSEN
```

codiert.

9.2 Symmetrische Verschlüsselung

Bei der **symmetrischen Verschlüsselung** besitzen der Sender und die berechtigten Empfänger eine gemeinsame geheime Zusatzinformation (den **Schlüssel**), darin unterscheiden sie sich von den Außenstehenden.

Derselbe Schlüssel wird sowohl vom Sender zum Verschlüsseln des Klartext verwendet, als auch vom Empfänger für das Entschlüsseln des Geheimtext benötigt.

Beim Sender ist das kein Problem, da er den Schlüssel gewählt/erzeugt hat.

Dem Empfänger fehlt dieser Schlüssel erstmal, deswegen ist es bei der symmetrischen Verschlüsselung sehr wichtig, dass der Schlüssel auf einem sicheren Übertragungsweg an den Empfänger weitervermittelt wird.

Anschauung

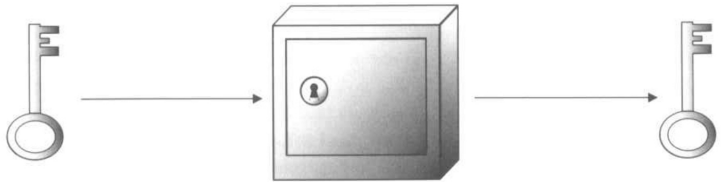


Abbildung 5: Symmetrische Verschlüsselung, Anschauung
(Quelle: A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter; Moderne Verfahren der Kryptographie)

Verschlüsseln schützt die Nachricht davor gelesen zu werden.

Man kann sich vorstellen, dass der Sender die Nachricht in einen Tresor legt und mit Hilfe seines Schlüssels abschließt.

Der Tresor wird samt Inhalt an den Empfänger geschickt. Dieser hat einen identischen Schlüssel, um den Tresor zu öffnen und die Nachricht zu lesen.

In der Kryptographie werden die Nachrichten nicht durch physikalische Maßnahmen geschützt, sondern durch mathematische Methoden.

Verschlüsselungsalgorithmus

Definition 9.1. Ein **symmetrischer Verschlüsselungsalgorithmus** besteht aus einer Funktion f mit zwei Eingabewerten, dem **Schlüssel** k und dem **Klartext** m, die Ausgabe ist der **Geheimtext** c, der sich aus k und m ergibt.

Hinweis

Da Kryptographie/Kryptoanalyse mit Wahrscheinlichkeitrechnung verzahnt ist, wo das Symbol p (*probability*) häufig benutzt wird, wird für Klartext das Symbol m (*plaintext message*) verwendet.

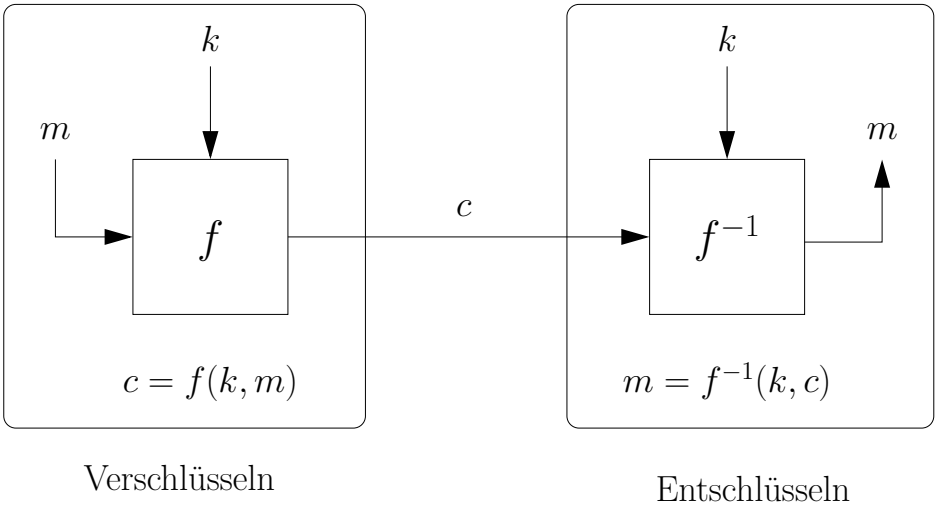
Die Verschlüsselungsfunktion f muss **umkehrbar** sein, d.h. es muss eine Funktion f^{-1} geben, die die Wirkung von f rückgängig macht.

Mit dem Schlüssel k und dem Gemeintext c kann man mit f^{-1} den Klartext m rekonstruieren.

Angenommen Sender und Empfänger benutzen den gemeinsamen (geheimen) Schlüssel k.

- Der Sender verschlüsselt einen Klartext m, indem er den Gemeintext c berechnet, $c = f(k, m)$ (oft auch $f_k(m)$).
- Der Empfänger rekonstruiert den Klartext m, indem er den Geheimtext c entschlüsselt, $m = f^{-1}(k, c) = f_k^{-1}(c)$

Funktionsschema



Verschlüsseln und Entschlüsseln müssen in einer sicheren Umgebung stattfinden, das wird im Bild durch die Kästen symbolisiert.

9.3 Blockchiffren und Stromchiffren

Verschlüsselungsverfahren

In der Regel sollen große Nachrichten oder Nachrichtenströme, d.h. eine kontinuierliche Abfolge von Zeichen, verschlüsselt werden, was mit einer einmalige Anwendung der Verschlüsselungsfunktion nicht umsetztbar ist.

Blockchiffren und **Stromchiffren** sind Verfahren für die Anwendung von Verschlüsselungsalgorithmen auf große Nachrichten oder Nachrichtenströme.

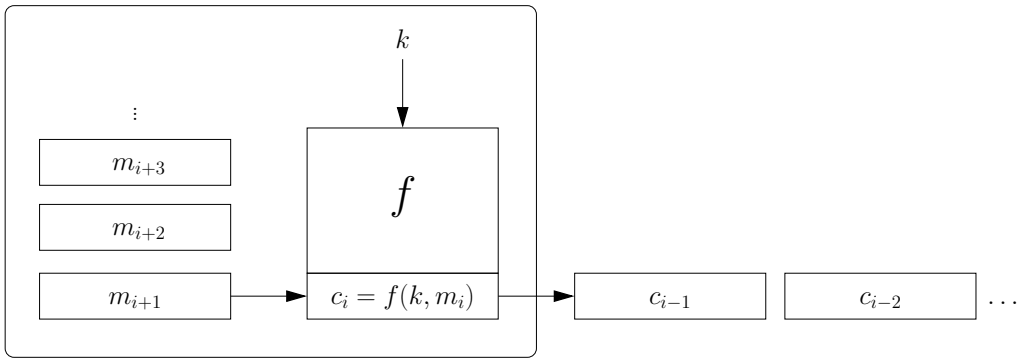
Beide Verschlüsselungsverfahren teilen den Klartext in Blöcke gleicher Größe $m_1, m_2, \ldots$ auf.

Durch die Verschlüsselung entstehen Geheimtextblöcke $c_1, c_2, \ldots$ gleicher Größe.

Blockchiffre

- Die Klartextblöcke werden unabhängig voreinander mit der Verschlüsselungsfunktion und **demselben Schlüssel** einzeln zu Geheimtextblöcken chiffriert.
- Die Geheimtextblöcke werden entsprechend einzeln dechiffriert.

Blockchiffre mit Verschlüsselungsfunktion f und Schlüssel k.



Verschlüsseln

Caesar-Verschlüsselung

Die **Caesar-Verschlüsselung** ist eine Blockchiffren mit symmetrischem Verschlüsselungsalgorithmus.

Ein Klartext m kann eine beliebige Zeichenfolge über dem Alphabet $\Sigma = \{A, B, \ldots, Z\}$ der 26 Großbuchstaben sein, $m \in \Sigma^*$.

Ein Klartext wird in Blöcke der Größe ein Zeichen aufgeteilt.

Der Schlüssel k ist ein Zeichen aus dem Alphabet Σ.

Die symmetrische Verschlüsselungsfunktion $f_k : \Sigma \to \Sigma$ verschiebt das übergebene Zeichen im Alphabet zyklisch nach **rechts**, dabei wir die Anzahl der zu verschiebenden Stellen von der Position des Schlüssels k im Alphabet bestimmt.

Die symmetrische Entschlüsselungsfunktion $f_k^{-1} : \Sigma \to \Sigma$ verschiebt entsprechend, abhängig von k, das übergebene Zeichen im Alphabet zyklisch nach **links**.

Beispiel

Caesar-Verschlüsselung mit Schlüssel C.

Die Position von Schlüssel C im Alphabt ist 3, d.h f_C (f_C^{-1}) verschiebt ein übergebenes Zeichen um 3 Stellen im Alphabet zyklisch nach rechts (links).

$$
\begin{array}{ll}
f_C(A) = D & f_C^{-1}(D) = A \\
f_C(B) = E & f_C^{-1}(E) = B \\
\ldots & \ldots \\
f_C(W) = Z & f_C^{-1}(Z) = W \\
f_C(X) = A & f_C^{-1}(A) = X \\
f_C(Y) = B & f_C^{-1}(B) = Y \\
f_C(Z) = C & f_C^{-1}(C) = Z
\end{array}
$$

CAESAR wird zeichenweise mit C verschlüsselt zu *FDHVDU*, das wird zeichenweise mit C entschlüsselt wieder zu *CAESAR*.

Beispiel - Python

Caesar-Verschlüsselung in Python

```
1  alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3  def char_to_index(c):
4      return (alphabet.index(c.upper()))
5
6  def index_to_char(n):
7      return alphabet[n % len(alphabet)]
8
9  def caesar(key, plain, deciphering=False):
10     k = char_to_index(key) + 1
11     if (deciphering): k = -k
12     c = ""
13     for p in plain:
14         if (p not in alphabet):
```

```
15              raise ValueError('"' + p + '" is not an alphabet charater')
16          c = c + index_to_char(char_to_index(p) + k)
17      return c
```

Funktionen können mit Parametern definiert werde, für die ein Standardwert (*default value*) festgelegt wird.

Wird für diese Parameter beim Aufruf der Funktion ein Argument übergeben, referenziert der Parameter in der Funktion dieses Argument. Wird kein Argument übergeben, referenziert der Parameter den Standardwert.

```
def caesar(key, plain, deciphering=False):
```

Wenn ein Block, der durch einen Doppelpunkt eingeleitet wird, nur aus einer Ausweisung besteht, kann diese Anweisung direkt in die Zeile mit dem Doppelpunkt geschrieben werden.

```
if (deciphering): k = -k
```

Python unterstützt **Exceptions**.

Auch wenn ein Ausdruck syntaktisch korrekt ist, kann bei dessen Ausführung einen Fehler auftreten. Ein Exception ist so ein Fehler, der vom Laufzeitsystem erkannt wurde.

Exception können im Code selbst behandelt werden, passiert das nicht behandelt sie das Laufzeitsystem, üblicherweise mit einer Fehlermeldungen, inklusive nützlicher Zusatzinformationen, z.B. der Codezeile, die die Exception verursacht hat.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Es gibt *built-in exceptions*, z.B. **`ValueError`**, das sind Klassen von den man, mit einem Konstruktor gleichen Namens, Objekte erzeugen kann.

Dem Konstruktor von **`ValueError`** kann man als Argument noch eine String, z.B. einer Fehlermeldung, übergeben

```
>>> e=ValueError("unknown reason")
>>> e
ValueError('unknown reason')
>>> type(e)
<class 'ValueError'>
>>> print(e)
unknown reason
```

Mit der *built-in function* **raise** kann man eine Exception auslösen.

```
>>> e=ValueError("unknown reason")
>>> raise(e)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: unknown reason
```

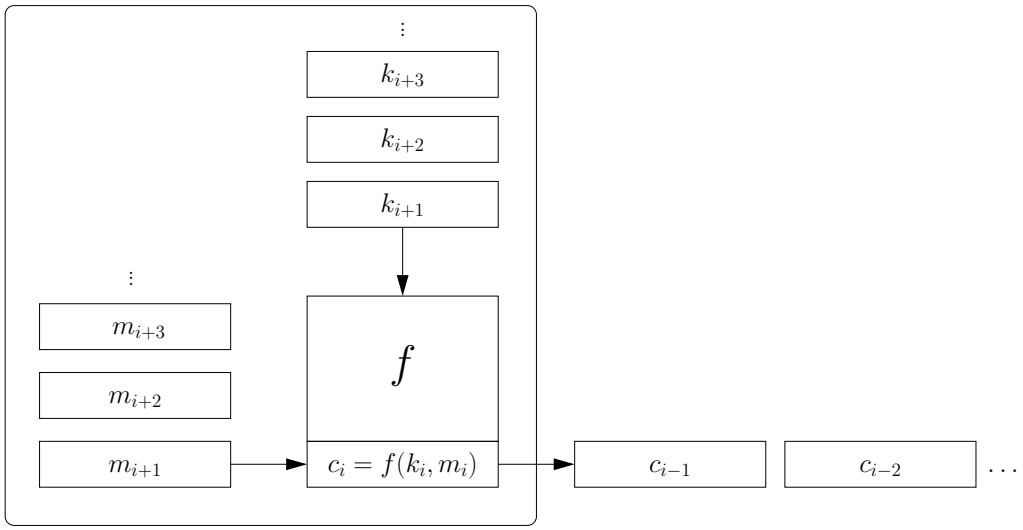
Bemerkung

Wie man Exceptions behandelt wird hier nicht besprochen, siehe dazu *Python Documentation, 8. Errors and Exceptions* `https://docs.python.org/3/tutorial/errors.html`.

Stromchiffre

- Eine **Schlüsselstrom** $k_1, k_2, \ldots$ wird erzeugt, sodass für jeden Klartextblock ein eigener Schlüssel vorliegt.
- Ein Klartextblock wird jeweils mit der Verschlüsselungsfunktion und dem zugehörigen Schlüssel des Schlüsselstroms zu einem Geheimtextblock chiffriert.
- Ein Geheimtextblock kann nur mit den zugehörigen Schlüssel des Schlüsselstroms dechiffriert werden.

Stromchiffre mit Verschlüsselungsfunktion f und Schlüsselstrom $k_1, k_2, \ldots$.



Verschlüsseln

One-Time-Pad

One-Time-Pad (wörtlich *Einmal-Block*) ist eine Stromchiffre für (große) Nachrichten bekannter Länge mit symmetrischem Verschlüsselungsalgorithmus.

Es werden **zufällig** genauso viele Schlüssel erzeugt, wie Klartextblöcke zu verschlüsseln sind, dabei haben die Schlüssel mindestens die gleiche Länge wie die Klartextblöcke.

Jeder Schlüssel wird nur **einmal** verwendet, um einen Klartextblock mit der Verschlüsselungsfunktion in einen Geheimtextblock zu chiffrieren.

Bemerkung

Da es für jeden Klartextblock einen Schlüssel gibt, die Schlüssel zufällig gewählt und nur einmal verwendet werden, kann man zeigen, dass One-Time-Pad ein absolut sicheres Verfahren ist.

Bemerkung

One-Time-Password (*Einmalkennwort*) ist ein anderes Verfahren, das z.B. durch Transaktionsnummern-Listen (TAN-Listen) beim Online-Banking Anwendung findet.

Beispiel

Für ein Umsetzung von One-Time-Pad wird die Verschlüsselungsfunktion $f_k : \Sigma \to \Sigma$ des Caesar-Verschlüsselung mit Alphabet $\Sigma = \{A, B, \ldots, Z\}$ benutzt.

Um den, aus der Nachricht

`XV. Maerz Caesar treffen, Dolche nicht vergessen`

codierten, Klartext

`XVMAERZCAESARTREFFENDOLCHENICHTVERGESSEN`

zu verschlüsseln wird eine ebenso lange zufällige Schlüsselfolge

`CTOSHBFEXKKSSMPJMYEXOPFRBYNZASDTZUXTSAWP`

erzeugt, z.B. durch Beobachtung der Bewegung von Blättern auf dem *Forum Romanum*.

Die Zeichen des Klartext werden einzeln mit der Verschlüsselungsfunktion f_k und jeweils einem Zeichen k der Schlüsselfolge zum Geheimtext

`APBTMTFHYPDTKGHOSEJLSERUJDBIDAXPEMEYLTBD`

chiffriert.

Beispiel - Python

One-Time-Pad mit Caesar-Verschlüsselung in Python

```
1 def onetimepad(key, plain, deciphering=False):
2     if (len(key) != len(plain)):
3         raise IndexError("length of key and plaintext did not match")
4     c = ""
5     for k, p in zip(key, plain):
6         c = c + caesar(k, p, deciphering)
7     return c
```

Die *built-in function* **zip** nimmt als Argumente n iterierbare Objekte entgegen und fasst diese in einem neuen iterierbaren Objekte von n-Tupeln zusammen.

Das i-te Tupel enthält das i-te Objekt aus jedem der n übergebenen Argumente. Die Anzahl der Tupel entspricht der Anzahl Elemente des kürzesten Arguments.

```
>>> z=zip("ABCD", "123", "ZYXW")
>>> list(z)
[('A', '1', 'Z'), ('B', '2', 'Y'), ('C', '3', 'X')]
```

Mehrere (gleichlange) iterierbare Objekte kann man mit **for**, Mehrfachbelegung (*multiple assignment*) und **zip** komfortable gleichzeitig durchlaufen.

```
>>> for c, x in zip("ABC", "123"):
...     print(f"{c} {x}")
...
A 1
B 2
C 3
```

Schlüsselerzeugung

Zum Austausch von verschlüsselten Nachrichten ist One-Time-Pad mit diesem Verfahren der Schlüsselerzeugung aber nur bedingt geeignet, denn zum Entschlüsseln braucht man die Schlüsselfolge, die genauso lang wie der Klartext ist.

Die Schlüsselfolge vorher, für einer später zu versendende Nachricht, auszutauschen oder parallel zum Geheimtext (z.B. durch zwei unabhängige Boten) auszutauschen ist denkbar, dadurch degeneriert das Verfahren aber zu einer Blockchiffre mit einem großen Block und einem großen Schlüssel.

Benötigt wird ein Algorithmus, der aus einem (kleinen) Initial-Schlüssel einen (großen) Schlüssel für One-Time-Pad berechnet, der möglichst nicht von einem wirklich zufällig erzeugten Schlüssel unterschieden werden kann.

9.4 Pseudozufall

In der Kryptographie spielen Zufallszahlen und Zufallsfolgen eine wichtige Rolle.

Dabei gibt es verschiedenen Aspekte.

- In vielen Kommunikationsprotokollen muss an einer bestimmten Stelle einen zufälligen Wert wählen. Dabei hängt die Sicherheit des Protokolls direkt davon ab **wie zufällig** der Wert gewählt wurde.

Im eigenen Interesse müssen die Kommunikationspartner darauf achten den Wert mit einem möglichst guten Zufallsgenerator zu wählen.

- Oft ist es nicht praktikabel oder sogar unmöglich echte Zufallszahlen und Zufallsfolgen zu verwenden. Das ist z.B. dann der Fall, wenn die Zahlen oder Folgen von mehreren erzeugt werden müssen.

  In diesen Fällen werden von Folgen von **Pseudozufallszahlen** verwendet, die mit Hilfe eines deterministischen Algorithmus berechnet werden, aber nur sehr schwer von wirklicher Zufälligkeit unterschieden werden können.

Echte Zufallszahlen und Zufallsfolgen werden mit Hilfe physikalischer Phänomene erzeugt, z.B. mit Hilfe des Rauschens elektronischer Bauelemente, dem radioaktiven Zerfall oder dem Konvektionsströmungen in einer Lavalampe.

Ein klassisches Beispiel für das Erzeugen echte Zufallsfolgen ist das Werfen einer *fairen Münze*.

Bei der Beurteilung der Güte eines Zufallsfolgengenerators ist ein Problem, dass man die Zufälligkeit einer Folge **nicht** beweisen kann.

Die nicht Zufälligkeit einer Folge lässt sich beweisen, durch Angabe eines Algorithmus zur Erzeugung der Folge.

Bei Pseudozufallsfolgen steht man prinzipiell vor dem gleichen Problem. Es gibt kein Kriterien um echte Zufallsfolgen zu bewerten, deshalb kann man nicht entscheinden wie zufällig eine Pseudozufallsfolge aussieht.

Wiederum kann man nur das Gegenteil nachweisen, nämlich dass eine Pseudozufallsfolge nicht zufällig aussieht.

Linearer Kongruenzgenerator

In den Bibliotheken vieler Programmiersprachen ist ein **linearer Kongruenzgenerator** (*linear congruential generator*) zur Erzeugung von Pseudozufallen implementiert.

Ein linearer Kongruenzgenerator berechnet eine Folge $x_1, x_2, x_3, \ldots$ von Pseudozufallen aus den Parametern $m, a, c, x \in \mathbb{N}$.

- Modul (*modulus*) $m > 0$

- Faktor (*multiplier*) $0 < a < m$
- Inkrement (*increment*) $0 \leq c < m$
- Startwert (*seed*) $0 \leq x < m$

Es gilt

$$x_i = ax_{i-1} + c \mod m$$

mit $x_0 = x$ für $i > 0$.

Bemerkung

Mathematisch korrekt handelt es sich mit $c > 0$ um einen affinen Abbildung, aber die Bezeichnung linearer Kongruenzgenerator hat sich etabliert.

Standardkonfiguration

Für die unterschiedlichen Implementierungen gibt es ebenso viele Standardkonfiguration, die gute Pseudozufallsfolgen produzieren sollen.

Die Funktionen `lrand48`/`nrand48` der C-Standardbibliothek (POSIX+glibc) und `java.util.Random` benutzen folgende Konfiguration.

- $m = 2^{48}$
- $a = 25214903917$
- $c = 11$
- berechnet werde jeweils 48-Bit, aber zurückgegeben werden nur die 32 höchstwertigen Bits

Beispiel - Python

Linearer Kongruenzgenerator in Python

```
1 def lcg(m, a, c, seed=0, cut=0, count=1):
2     k=[]
3     x=seed
4     for _ in range(count):
5         y=(a*x+c)%m
6         k.append(y//2**cut)
7         x=y
8     return k
```

Wenn man **lcg** mit der vorstehenden Konfiguration, sowie **seed=5**, **cut=16** ($2^{48}/2^{16} = 2^{32}$) und **count=10** aufruft

```
print(lcg(2**48, 25214903917, 11, 5, 16, 10))
```

bekommt man folgende 10 Pseudozufallszahlen.

```
[1923744, 2816743950, 832832900, 735168418, 2203812749,
1955956184, 776948653, 3341115479, 2902097218, 3472087760]
```

In der **for**-Schleife wird der Unterstrich _ als Platzhalter für die Schleifenvariable benutzt, weil diese im Rumpf der Schleife nicht benötigt wird.

Man kann mit Standwerten belegten Parametern

```
def lcg(m, a, c, seed=0, cut=0, count=1):
```

Argumente nicht nur über die Position in der Argumentliste (*positional argument*),

```
print(lcg(2**8, 37, 11, 7))
```

sondern auch über den Namen des Parameters (*keyword argument*)

```
print(lcg(2**8, 37, 11, seed=7, count=257))
```

zuordnen.

Nach dem ersten *keyword argument*, können nur noch solche Argumente folgen. Die Reihenfolge der *keyword arguments* ist beliebig.

```
print(lcg(2**8, 37, 11, count=257, seed=7, cut=2))
```

Bewertung

Linearen Kongruenzgeneratoren können Pseudozufallszahlen mit guten statistischen Eigenschaften erzeugten, eignen sich aber für die meisten kryptographischen Anwendungen nicht.

Man kann, mit vertretbarem Aufwand, aus wenigen Werten alle künftigen berechnen ohne den *seed* zu kennen.

Kennt man z.B. die aufeinanderfolgende Werte x, y, z und den Modul m, dann erhält man a und c durch Auflösen der beiden folgenden Gleichungen.

$$y = ax + c \mod m$$
$$z = ay + c \mod m$$

Mit a und c kann man alle auf z folgenden Werte berechnen.

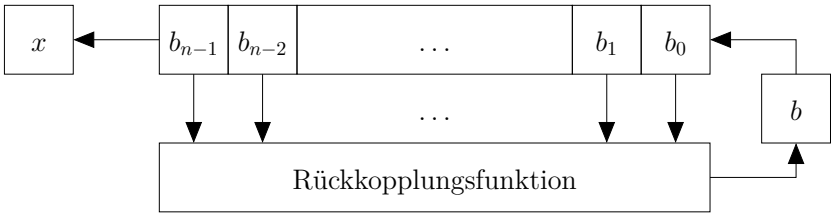
Bemerkungen

- Nicht alle Konfigurationen sind geeignet gute Pseudozufallszahlen zu erzeugen.
- Manche Konfigurationen erschweren das Auflösen der Gleichungen, aber prinzipiell bleibt das ein lösbares Problem.

Lineare Schieberegister mit Rückkopplung

Lineare Schieberegister werden häufig in der Kryptographie verwendet, da sie leicht in digitalen Schaltwerken zu realisieren sind.

Die Schieberegister mit Rückkopplung (*linear feedback shift register, LSR*) bestehen aus zwei Teilen, dem Schieberegister und der Rückkopplungsfunktion.



Die Funktion eines n-stelligen Schieberegisters.

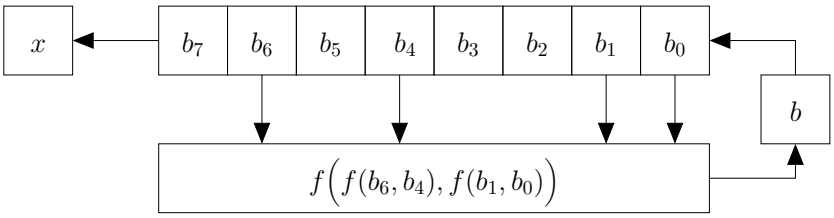
- Die Ausgabe des Schieberegister ist $x = b_{n-1}$.
- Der neue Wert b wird in Abhängigkeit von $b_{n-1}, \ldots, b_0$ und der Rückkopplungsfunktion berechnet.
- Die Einträge des Schieberegister werden mit folgender Regel nach links verschoben $b_{i+1} = b_i$ für $i = 0, \ldots, n-2$.
- Von links wird b in das Schieberegister hinein geschoben $b_0 = b$.

Eine Rückkopplungsfunktion kann mehr oder weniger kompliziert sein.

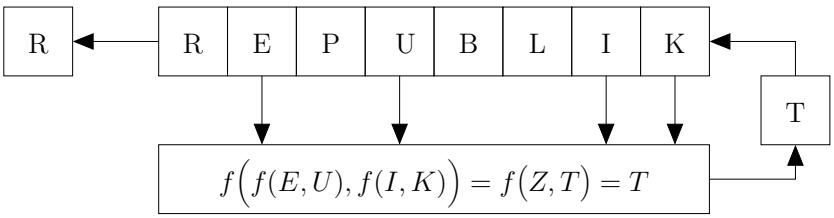
Beispiel

Sei, wie im One-Time-Pad-Beispiel,

- $\Sigma = \{A, B, \ldots, Z\}$ das Alphabet,
- $f(k, m) : \Sigma \times \Sigma \to \Sigma$ die Verschlüsselungsfunktion der Caesar-Verschlüsselung.



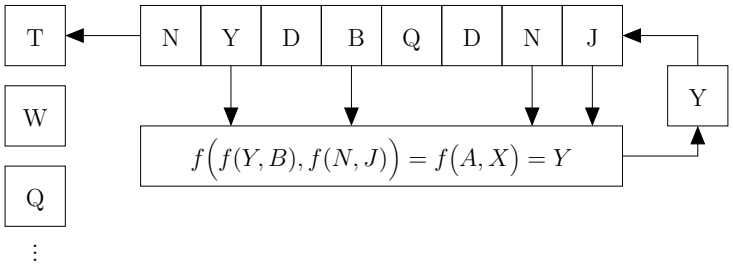
Initialisierung des Schieberegisters mit dem Schlüssel `REPUBLIK`.



Dann ergibt sich der Schlüsselstrom

```
REPUBLIKTWXFAJSUQKRWTAAKTQIUPFXGQWOZBQWT
```

und folgender Zustand des Schieberegisters.



Mit dem Schlüssel `REPUBLIK` können mit Hilfe eines lineare Schieberegisters mit (öffentlich) bekannter Rückkopplungsfunktion der Sender und jeder berechtigte Empfänger einen identischen Schlüsselstrom

```
REPUBLIKTWXFAJSUQKRWTAAKTQIUPFXGQWOZBQWT
```

erzeugen mit dem der Sender den Klartext

```
XVMAERZCAESARTREFFENDOLCHENICHTVERGESSEN
```

zum Geheimtext

```
PACVGDINUBQGSDKZWQWKXPMNBVWDSNRCVOVEUJBH
```

verschlüsseln kann und die berechtigten Empfänger den Klartext wiederherstellen können.

Um so besser die Pseudozufallszahlen, um schwerer fällt es einem Angreifer den Geheimtext zu entschlüsseln (siehe One-Time-Pad).

Linear/nicht linear

Folgen, die mit linearen Schieberegistern erzeugt werden haben sehr gute statistische Eigenschaften, d.h. die einzelnen Buchstaben, Bits, etc. sind in einer ausreichend großen Stichprobe gleichmäßig verteilt.

Trotzdem kann bei dieser einfachen Erzeugung des Schlüsselstroms schon aus relativ kurzen bekannten Schlüsselfolgen (nicht Geheimtexten) der Schlüssel mit vertretbarem Aufwand berechnet werden, weil ein **lineares** Verfahren benutzt wurde.

In der Praxis werden deshalb **nicht lineare** Verfahren verwendet, d.h. entweder werden nicht lineare Schieberegister (mit komplexerer Rückkopplungsfunktion) benutzt oder es werden mehrere lineare Schieberegister nicht linear gekoppelt.

9.5 Kryptosystem

Ein **Kryptosystem** besteht aus

- Endlichen Alphabeten
  - Γ_M der Klartextzeichen
  - $\Sigma_M \subseteq \Gamma_M^*$ der Klartextsymbole
  - Γ_C der Geheimtextzeichen
  - $\Sigma_C \subseteq \Gamma_C^*$ der Geheimtextsymbole
- Klartextmenge $M \subseteq \Sigma_M^*$
- Geheimtextmenge $C \subseteq \Sigma_C^*$
- Schlüsselmenge $K \neq \emptyset$
- Verschlüsselungsabbildungen $e : M \times K \to C$
- Entschlüsselungsabbildungen $d : C \times K \to M$
- Es gilt, dass für jedes $k' \in K$ (Schlüssel) ein $k'' \in K$ (Gegenschlüssel) existiert, sodass gilt $d(e(m, k'), k'') = m$ für alle $m \in M$.

Ein Kryptosystem heißt **symmetrisch**, wenn Schlüssel und Gegenschlüssel gleich sind, d.h. für alle $k \in K$ gilt $d(e(m, k), k) = m$ für alle $m \in M$.

Asymmetrische Kryptosystem, bei denen sich Schlüssel und Gegenschlüssel unterscheiden, sind erst seit Ende der siebziger Jahre bekannt.

Kryptosysteme werden auch nach der Wirkung von Verschlüsselungs- und Entschlüsselungsabbildungen unterschieden.

Nachfolgend werden Substitutions-Chiffren und Transpositions-Chiffren betrachtet.

Transposition

Bei einer **Transpositions-Chiffre** bleiben die Symbole *was* sie sind, aber nicht *wo* sie sind.

Abhängig vom Schlüssel wird die Position der Symbole mit der Permutation π vertauscht.

$$m_1 m_2 \dots m_t \mapsto m_{\pi(1)} m_{\pi(2)} \dots m_{\pi(t)} = c_1 c_2 \dots c_t$$

Beispiel. Matrixtransposition

Bei der **Matrixtransposition** wird der Klartext in Zeilen gleicher Länge angeordnet (unvollständige Zeilen werden aufgefüllt).

Zur Bildung des Geheimtextes wird der so angeordnete Text spaltenweise zurückgeschrieben.

Beispiel

Klartext zeilenweise (Zeilenlänge 5) anordnen.

```
XVMAE
RZCAE
SARTR
EFFEN
DOLCH
ENICH
TVERG
ESSEN
```

Geheimtext spaltenweise zurückschreiben.

```
XRSEDETEVZAFONVSMCRFLIESAATECCREEERNHHGN
```

Substitution

Bei einer **Substitutions-Chiffre** bleiben die Symbole *wo* sie sind, aber nicht *was* sie sind.

Substitutions-Chiffren erzeugen die Verschlüsselungsabbildungen mit Substitutionen (Ersetzungen).

Eine Substitution ist eine **injektive Abbildung**,

$$s_k : \Sigma_M \to \Sigma_C \quad \text{für } k \in K$$

die Symbole des Klartextalphabets auf Symbole des Geheimtextalphabets abbildet.

Es gilt Klartextmenge $M = \Sigma_M^*$ und Geheimtextmenge $C = \Sigma_M^*$.

Monoalphabetische Substitution

- Benutzung eine Alphabets (historisch), d.h. Benutzung genau einer Substitution, somit werden an jeder Stelle des Klartext gleiche Klartextsymbole durch dieselben Geheimtextsymbole ersetzt.
- Die Substitution

  $$s_k : \Sigma_M \to \Sigma_C \quad \text{für } k \in K$$

  induziert eine Verschlüsselungsabbildung

  $$e(m_1 m_2 \dots m_t, k) = s_k(m_1) s_k(m_2) \dots s_k(m_t) = c_1 c_2 \dots c_t$$

Polyalphabetische Substitution

- Benutzung mehrerer Alphabete (historisch), d.h. wechselnde Benutzung mehrere Substitutionen. Das kann man über die Anwendung einer Substitution, für die die verschiedenen Alphabete verschmolzen werden, mit unterschiedlichen Schlüsseln erreicht werden.
- Die Substitution

  $$s_k : \Sigma_M \to \Sigma_C \quad \text{für } k \in K$$

  und die Schlüsselmenge $k = \{k_1, \dots, k_p\} \subseteq K$ induziert eine Verschlüsselungsabbildung

  $$e(m_1 m_2 \dots m_t, k) = s_{\alpha(1)}(m_1) s_{\alpha(2)}(m_2) \dots s_{\alpha(t)}(m_t) = c_1 c_2 \dots c_t$$

  mit der Abbildung $\alpha : \{1, \dots, t\} \to \{k_1, \dots, k_p\}$.

Monographische Substitution

- Ein Klartextzeichen entspricht einem Klartextsymbol, d.h. $\Sigma_M = \Gamma_M$.

Beispiele

- Strom/Blockchiffren bei denen die Blöcke jeweils einzelne Zeichen sind

- Caesar-Chiffre ist eine monoalphabetische und monographische Substitution.

Polygraphische Substitution

- Die Klartextsymbole umfassen jeweils mehrere Zeichenpaare, Zeichentripel, etc. (allgemein **n-Gramme**), d.h. $\Sigma_M = \underbrace{\Gamma_M \times \ldots \times \Gamma_M}_{\text{n-mal}} = \Gamma_M^n$.

Beispiel. Vigenère, One-Time-Pad.

Die Vigenère-Chiffre stammt aus dem 16. Jahrhundert und wurde von dem französischen Kryptographen Blaise de Vigenère (1523-1596) entwickelt.

Basiert auf der Verwendung der Caesar-Chiffre, allerdings mit periodisch wechselnden Schlüsseln.

Galt lange Zeit als nicht zu knacken, insbesondere war das Ermitteln der Schlüssellänge problematisch, und konnte erst um 1850 entziffert werden.

Beispiel

Klartext: XVMAERZC AESARTRE FFENDOLC HENICHTV ERGESSEN
Schlüsselfolge: REPUBLIK REPUBLIK REPUBLIK REPUBLIK REPUBLIK
Geheimtext: PACVGDIN SJIVTFAP XKUIFAUN ZJDDETCG WWWZUENY

Beispiel - Python

Vigenère-Chiffre mit Caesar-Verschlüsselung in Python

```
1 def vigenere(key, plain, deciphering=False):
2     c=""
3     i=0
4     for p in plain:
5         c=c+caesar(key[i], p, d)
6         i=(i+1)%len(key)
7     return c
```

9.6 Kryptoanalyse

Die Dechiffrierung ohne Kenntnis der Geheiminformation, das *Brechen der Chiffre*, wird **Kryptoanalyse** genannt.

Grundannahme (**Kerckhoffs' Maxime**, 1883)

Die Sicherheit einer Chiffre darf nicht darauf beruhen, dass der Angreifer (Kryptoanalyst) das benutzte Verfahren nicht kennt.

Angriffsarten und Analysemethoden

Bei allen Angriffen ist die Verschlüsselungsfunktion e bekannt.

Das Klartextblöcke m' mit der Verschlüsselungsfunktion e und einen Schlüssel k zu Geheimtextblöcken c' verschlüsselt werden, d.h. $c' = e(m', k)$, ist ebenfalls bekannt.

ciphertext-only

- Beobachtet: Geheimtext c
- Gesucht:
  - Klartext m

  - Schlüssel k
  - Blocklänge des Geheimtexts, d.h. $c_1 \dots c_j = c$ (wenn nicht bekannt)
  - Algorithmus, um $m_{j+1}, \dots$ aus $c_{j+1}, \dots$ herzuleiten

Beispiel. Ein Lauscher (englisch *eavesdropper*), z.B. in einem Netzwerk.

known-plaintext

- Beobachtet: Klartext $m_1 \dots m_j$ und zugehöriger Geheimtext $c_1 \dots c_j$
- Gesucht
  - Schlüssel k
  - Algorithmus, um $m_{j+1}, \dots$ aus $c_{j+1}, \dots$ herzuleiten

Beispiel. Wiederkehrende Anfangs- und Schlußformeln.

chosen-plaintext

- Wähle Klartext $m_1 \dots m_j$, beobachte zugehörigen Geheimtext $c_1 \dots c_j$
- Gesucht
  - Schlüssel k
  - Algorithmus, um $m_{j+1}, \dots$ aus $c_{j+1}, \dots$ herzuleiten

adaptive chosen-plaintext

- Wähle m_1, beobachte c_1, wähle m_2, usw.

Beispiel.

- Freund-Feind-Erkennung (*challenge response protocol*)
- Public-Key-Systeme

Sichere/unsichere Kryptosystem

System heißt **sicher** gegen bestimmten Angriffstyp, wenn ein potentieller Angreifer die erforderlichen Berechnungen nicht mit **vertretbarem Aufwand** durchführen kann.

Nachweis der Sicherheit ist schwierig ⇒ Rückführung auf anerkannt schwierige Probleme

- Faktorisierung großer Zahlen
- NP-vollständige Probleme
- ...

Vorsicht. Schwierige Probleme können auch einfache Instanzen haben!

In der Regel ist es einfacher, die Unsicherheit eines Systems gegen bestimmte Angriffe nachzuweisen.

Offenbar unsicher gegen *chosen-plaintext*-Angriffe sind

- Caesar,
- Vigenère,
- Matrixtransposition.

Diese Verfahren sind ebenfalls unsicher gegen *ciphertext-only*-Angriffe, wenn **genügend viel** Geheimtext vorhanden und der Klartextraum hinreichend gut bekannt ist.

Klartextraum

Kenntnisse des Angreifers über den Klartextraum.

- Sprache (natürliche Sprache, Programmiersprache, ...)
- häufige Wörter (Kontext!)
- Sprachstatistik (Symbol-, Bigramm-, Trigramm-, ..., -Häufigkeiten)
- Buchstabenmuster (z.B. 1221 im Englischen: cabbage, ballast, apparant, ...)
- Randinformationen

- ...

Häufigkeitsmerkmale prägen natürliche Sprachen sehr stark ⇒ wichtigster Einstiegspunkt für Kryptoanalyse

Nützlich sind **Häufigkeitsreihenfolgen**, i.d.R. aber allein nicht ausreichend.

deutsch (verschiedene Quellen):

`enrisdutaghlobmfzkcwvjpqxy` (1840)
`enirsahtudlcgmwfbozkpjvqxy` (1863)
...
`enisratduhglcmwobfzkvpjqxy` (1955)

englisch (verschiedene Quellen):

`etaoinshrdlucmfwypvbgkqjxz` (1884)
`etoanirshdlcufmpywgbvkxjqz` (1893)
...
`etaoinsrhldcumfpgwybvkxjqz` (1982)

9.7 Asymmetrische Verschlüsselung

Die **asymmetrische Verschlüsselung** verwendet zwei unterschiedliche Schlüssel.

Es wird eine **Öffentlicher** und ein **privater** Schlüssel verwendet, die zueinander komplementär sind.

Daten, die mit dem Öffentlichen Schlüssel verschlüsselt werden (verschlüsseln von Nachrichten), können mit dem private Schlüssel entschlüsselt werden.

Daten, die mit dem privaten Schlüssel verschlüsselt werden (signieren von Nachrichten), können mit dem öffentlichen Schlüssel entschlüsselt werden.

Entscheidend bei der asymmetrischen Verschlüsselung ist, dass der private Schlüssel nicht aus dem öffentlichen Schlüssel abgeleitet werden kann.

Bei der asymmetrischen Verschlüsselung erzeugt jeder Teilnehmer ein Schlüsselpaar, behält den privaten Schlüssel und machten den öffentlichen Schlüssel den anderen Teilnehmern zugänglich.

Für den Austausch von verschlüsselten Nachrichten chiffriert der Sender die Daten mit dem öffentlichen Schlüssel des Empfängers und schickt sie diesem. Der Empfänger kann die Daten anschließend mit Hilfe seines privaten Schlüssels dechiffrieren.

Anschauung

Man kann einem Empfänger eine verschlüsselte Nachricht schicken, ohne eine Geheiminformation zu besitzen.

Das kann man sich wie das Einwerfen einer Nachricht in einen Briefkasten vorstellen.

Jeder, der Zugang zum Briefkasten (dem öffentlichen Schlüssel) hat, kann eine Nachricht einwerfen (Verschlüsseln).

Nur der Empfänger kann mit seinem privaten Schlüssel den Briefkasten öffnen und die Nachricht entnehmen (Entschlüsseln).

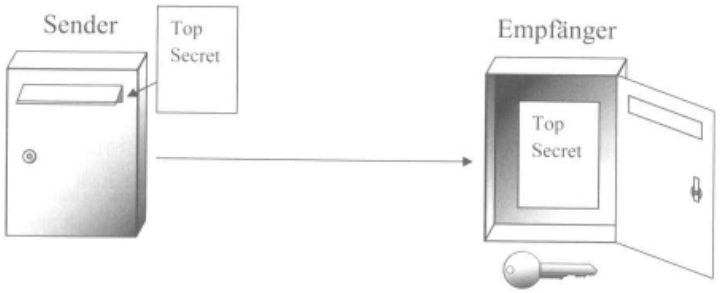


Abbildung 6: Asymmetrische Verschlüsselung, Anschauung
(Quelle: A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter; Moderne Verfahren der Kryptographie)

Verschlüsselungsalgorithmus

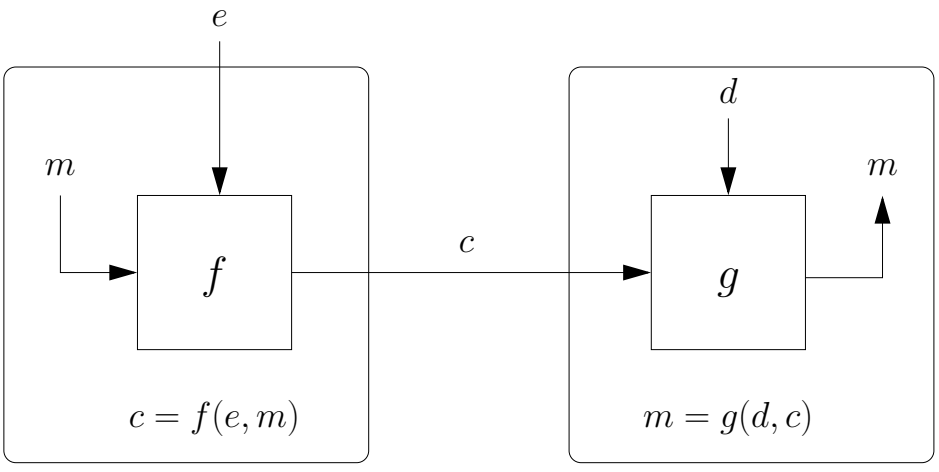
Definition 9.2. Ein **asymmetrischer Verschlüsselungsalgorithmus** besteht aus

- einer Funktion f mit zwei Eingabewerten, dem **öffentlichen Schlüssel** e und einem Text m, die Ausgabe ist der Text $f(e, m)$,
- einer Funktion g mit zwei Eingabewerten, dem **privaten Schlüssel** d und einem Text m, die Ausgabe ist der Text $g(d, m)$.

Für alle Texte m gelten die folgenden Beziehungen zwischen f und g.

$$g\Big(d, f(e, m)\Big) = m$$
$$f\Big(e, g(d, m)\Big) = m$$

Funktionsschema



Verschlüsseln
öffentlicher Schlüssel

Entschlüsseln
privater Schlüssel

Einwegfunktion

Eine **Einwegfunktion** ist eine Funktion, die einfach auszuführen, aber schwer (nur mit sehr großem Aufwand) zu invertieren ist.

Etwas formaler ist eine Einwegfunktion $f : X \to Y$ eine Abbildung für zwei Mengen X, Y, sodass Folgendes gilt.

- Für alle $x \in X$ ist $f(x)$ leicht zu berechnen.
- Für (fast) jedes $y \in Y$ ist es schwer ein Urbild $x \in X$, d.h. ein $x \in X$ mit $y = f(x)$, zu finden.

Einwegfunktionen spielen in der theoretischen und der praktischen Kryptographie eine entscheidende Rolle.

Beispiel

Die Funktion, die über Nachschlagen in einem gedruckten Telefonbuch einem Namen eine Telefonnummer zuordnet ist leicht auszuführen, da die Namen im Telefonbuch alphabetisch geordnet sind.

Allerdings ist die Umkehrung, die Zuordnung einer Telefonnummer zu einem Namen, mit Hilfe eine gedruckten Telefonbuchs ein sehr schwieriges Unterfangen.

Trapdoor-Einwegfunktion

Einwegfunktionen finden in der Kryptographie Verwendung, wenn alle Beteiligten die Funktion anwenden dürfen und kein Beteiligter die Umkehrfunktion kennt bzw. ermitteln kann, z.B. zur Integritätsprüfung von Daten.

Für die asymmetrische Verschlüsselung benötigt man ein erweitertes Konzept, da für Verschlüsseln und Entschlüsseln, sowohl Funktion als auch Umkehrfunktion benötigt werden.

Eine **Trapdoor-Einwegfunktion** ist eine Einwegfunktion, also eine eigentlich schwer zu invertierende Funktion, es sei denn man kennt die Zusatzinformation (die *trapdoor* zu Deutsch *Falltür*, im Deutschen oft *Hintertür*), mit deren Hilfe man die Funktion leicht invertieren kann.

Beispiel

Ist $n \in \mathbb{N}$ (groß und) **keine** Primzahl, dann ist das Quadrieren modulo n

$$\begin{aligned} f : \mathbb{N} &\to \{0, \ldots, n-1\} \\ x &\mapsto x^2 \mod n \end{aligned}$$

eine Einwegfunktion.

Seien p, q zwei (große) Primzahlen und $n = pq$, dann ist das Quadrieren modulo n eine Trapdoor-Einwegfunktion.

Die Trapdoor ist in diesem Fall die Faktorisierung von n, also die Faktoren p, q. Mit dieser Information ist das Invertieren einfach möglich.

Allgemein kann man zeigen, dass für zwei (große) Primzahlen p, q und $n = pq$ die Potenzfunktion

$$\begin{aligned} f : \mathbb{N} &\to \{0, \ldots, n-1\} \\ x &\mapsto x^k \mod n \end{aligned}$$

für beliebiges $k > 1$ eine Trapdoor-Einwegfunktion ist, mit Trapdoor p, q.

Ein weitere Trapdoor-Einwegfunktion ist die Exponentialfunktion.

Für zwei (große) Primzahlen p, q und $n = pq$ ist

$$\begin{aligned} f : \mathbb{N} \times \mathbb{N} &\to \{0, \ldots, n-1\} \\ f(m, x) &= m^x \mod n \end{aligned}$$

eine Trapdoor-Einwegfunktion, mit Trapdoor p, q.

Grundlagen

Satz 9.3. *Sei $n = pq$ das Produkt zweier Primzahlen $p, q \in \mathbb{N}$.*

Dann gilt für alle natürlichen Zahl m, k mit $m < n$ folgende Gleichung.

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

Rechenregeln

Für natürliche Zahlen $x, i, j, n \in \mathbb{N}$ gilt Folgendes.

$$\left(x^i \mod n\right)^j \mod n = \left(x^i\right)^j \mod n = x^{i \cdot j} \mod n$$

RSA-Algorithmus

Der RSA-Algorithmus wurde 1978 von Rivest, Shamir and Adleman erfunden, der Algorithmus basiert auf folgendem Prinzip.

Sei $n = pq$ das Produkt zweier Primzahlen $p, q \in \mathbb{N}$.

Wähle bzw. berechne die Schlüssel $e, d \in \mathbb{N}$, sodass für ein $k \in \mathbb{N}$ gilt

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

Dann gilt das Folgende für jedes $m \in \mathbb{N}$ mit $m < n$.

$$\begin{aligned} \left(m^e \mod n\right)^d \mod n &= m^{e \cdot d} \mod n = m \\ \left(m^d \mod n\right)^e \mod n &= m^{e \cdot d} \mod n = m \end{aligned}$$

Somit sind die Funktionen f und g identisch.

$$f(r, m) = g(r, m) = m^r \mod n$$

RSA-Schlüssel

Der **erweiterte euklidische Algorithmus** liefert für zwei natürliche Zahlen $a, b \in \mathbb{N}$ den größten gemeinsamen Teiler und zwei weitere ganze Zahlen $x, y \in \mathbb{Z}$, sodass folgendes gilt.

$$\gcd(a, b) = xa + yb$$

Dieser Algorithmus wird bei der Berechnung der Schlüssel verwendet.

Gesucht

- (e, n) öffentlicher Schlüssel (*public key*)
- (d, n) privater Schlüssel (*private key*)

Wähle zwei (große) ungleiche Primzahlen p, q und berechne den RSA-Modul $n = pq$.

Wähle eine Zahl e für die gilt

- $1 < e < (p-1)(q-1) - 1$
- e und $(p-1)(q-1)$ sind teilerfremd
- $\gcd\Big(e, (p-1)(q-1)\Big) = xe + y(p-1)(q-1)$ mit $x > 0$

Bemerkung. Eulersche φ-Funktion $\varphi(n) = (p-1)(q-1)$

Aus $xe + y(p-1)(q-1) = 1$ mit $x > 0, y < 0$ folgt

$$\begin{aligned} & xe + y(p-1)(q-1) = 1 \\ \iff \quad & xe = (-y)(p-1)(q-1) + 1 \end{aligned}$$

d.h. mit $d = x \in \mathbb{N}$ und $k = -y \in \mathbb{N}$ gilt

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

somit ist ein Schlüsselpaar

- (e, n) öffentlicher Schlüssel (*public key*)
- (d, n) privater Schlüssel (*private key*)

gefunden.