SageMath Beispiele aus dem CrypTool-Buch

Dr. Doris Behrendt

FrOSCon 2022
Hochschule Bonn-Rhein-Sieg

21. August 2022

Inhaltsverzeichnis

1 Was das CrypTool-Buch? 3

2 Überblick Inhalt des CTB 4

3 Für heute ausgewählte Beispiele 6

4 Klassische Verfahren 7

4.1 Substitutionsverfahren “zu Fuss” . . . .

4.2 String Monoids . . . .

4.3 Substitutionsverfahren von SageMath . . . .

4.4 Spezialfall der Substitutionsverschlüsselung: zyklische Verschiebung . . . .

4.5 Die Hill-Verschlüsselung “zu Fuss” . . . .

4.6 Hill-Verschlüsselung von SageMath . . . .

4.7 Inverse Matrix . . . .

4.8 Attacke auf die Hill-Cipher .

5 XOR -Verschlüsselung 35

5.1 Spass mit Bits . . . .

5.2 Konversionsfunktionen aus dem CrypTool-Buch . . . . . .

5.3 XOR aus dem CrypTool-Buch . .

5.4 XOR mit Pythonmitteln: das “Dach” ^ . . . .

6 RSA 53

6.1 Theorie ganz kurz . . . . . .

6.2 kleines RSA-Beispiel . . . . .

6.3 großes RSA-Beispiel . . . . .

6.4 Faktorisierung . . . .

1 Was das CrypTool-Buch?

- Das CrypTool-Buch (CTB) ist ein Buch über Kryptografie. Es ist Teil des Open-Source-Projekts CrypTool (CT).
  Projektleitung: Prof. Esslinger, Universität Siegen.
- Titel: Kryptografie lernen und anwenden mit CrypTool und SageMath.
- Zur Hauptseite des CrypTool-Projektes: https://www.cryptool.org/de/
- Zum CTB gelangt man von dort über den Tab "Dokumentation" oder direkt: https://www.cryptool.org/de/documentation/ctbook/
- Einige - noch nicht alle !!! - SageMath-Beispiele findet man hier: https://www.cryptool.org/de/documentation/ctbook/sagemath
- Derzeit noch 12. Auflage 2018; neue Auflage 2022 fast fertig, aber noch nicht final.

  Die SageMath-Programme und Screenshots in dieser Slideshow sind aus der neuen Auflage und können deshalb nicht mit Seitenzahl referenziert werden.

2 Überblick Inhalt des CTB

1. Sicherheits-Definitionen und Verschlüsselungsverfahren
2. Papier- und Bleistift-Verschlüsselungsverfahren
3. Primzahlen
4. Einführung in die elementare Zahlentheorie mit Beispielen
5. Die mathematischen Ideen hinter der modernen Kryptografie
6. Hashfunktionen, Digitale Signaturen und PKIs
7. Elliptische Kurven
8. Einführung in die Bitblock- und Bitstrom-Verschlüsselung
9. Homomorphe Chiffren
10. Aktuelle Erkenntnisse zu diskreten Logarithmen, Faktorisierung und deren Praxisbezug
11. Krypto 202x Perspektiven für langfristige kryptografische Sicherheit
12. Einführung in die Gitterkryptografie

- Anhänge:
  - SageMath, Jupyter
  - CT1, CT2 etc.
  - openssl
  - ...

3 Für heute ausgewählte Beispiele

Kurze Beispiele, damit Interessierte an ihrem eigenen Rechner mitmachen können während des Vortrags.

- klassische
- Umwandlung von Bitblöcken, XOR
- RSA

4 Klassische Verfahren

https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/classical.html

Classical Cryptosystems

A convenient user interface to various classical ciphers. These include:

- affine cipher; see `AffineCryptosystem`
- Hill or matrix cipher; see `HillCryptosystem`
- shift cipher; see `ShiftCryptosystem`
- substitution cipher; see `SubstitutionCryptosystem`
- transposition cipher; see `TranspositionCryptosystem`
- Vigenere cipher; see `VigenereCryptosystem`

These classical cryptosystems support alphabets such as:

- the capital letters of the English alphabet; see `AlphabeticStrings()`
- the hexadecimal number system; see `HexadecimalStrings()`
- the binary number system; see `BinaryStrings()`
- the octal number system; see `OctalStrings()`
- the radix-64 number system; see `Radix64Strings()`

Screenshot vom 17.08.22

4.1 Substitutionsverfahren “zu Fuss”

Gegeben eine Nachricht, geschrieben mit “Buchstaben” aus einer (endlichen) Menge Ω.

Vorerst nehmen wir

$$\Omega = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$$

mit $|\Omega| = 26$.

```
sage: Buchstaben='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: Omega=[i for i in Buchstaben]
sage: Omega
['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']
sage: len(Omega)
26
```

Bei einem Substitutionsverfahren werden die Buchstaben aus Ω einfach nur vertauscht. Mathematisch präziser: Die Verschlüsselung ist gegeben durch eine Permutation auf Ω.

```
sage: k=Permutations(26).random_element()
sage: k
[13, 8, 6, 4, 11, 16, 9, 14, 26, 18, 7, 22, 12, 3, 21, 2, 23, 24, 20, 5, 19, 10, 1, 25, 17, 15]
```

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
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
↓	↓	↓	↓	...																					↓
13	8	6	4	11	16	9	14	26	18	7	22	12	3	21	2	23	24	20	5	19	10	1	25	17	15
M	H	F	D	K	P	I	N	Z	R	G	V	L	C	U	B	W	X	T	E	S	J	A	Y	Q	O

HALLOWELT ⟶ NMVVUAKVE

```
sage: key=[Omega[k(i+1)-1] for i in range(len(Omega))]
#keep in mind that range(len(Omega))=range(26)=0,1,2,...,24,25; k(0) is not defined;
#Omega[k(i)] instead of Omega[k(i+1)-1] returns an error!!
sage: key
['M','H','F','D','K','P','I','N','Z','R','G','V','L','C','U','B','W','X','T','E','S','J','A','Y','Q','O']
```

“Der Schlüssel” ist also obige Liste. Übersichtlicher ist es, statt einer Liste ein Dictionary zu verwenden,
Syntax: {key1:value1,key2:value2,...}

```
sage: keydict={Omega[i]:key[i] for i in range(len(Omega))}
sage: keydict
{'A':'M', 'B':'H', 'C':'F', 'D':'D', 'E':'K', 'F':'P', 'G':'I', 'H':'N
 ', 'I':'Z', 'J':'R', 'K':'G', 'L':'V', 'M':'L', 'N':'C', 'O':'U', '
 P':'B', 'Q':'W', 'R':'X', 'S':'T', 'T':'E', 'U':'S', 'V':'J', 'W':'
 A', 'X':'Y', 'Y':'Q', 'Z':'O'}
```

Zugriff auf die den Keys zugeordneten Values:

```
sage: keydict['A']
'M'
```

Das dann iteriert und - ohne Leerzeichen per ''.join - zu einem String gemacht:

```
sage: msg='HALLOWELT'                # msg for message
sage: c=''.join([keydict[i] for i in msg])
sage: c                              # c for ciphertext
'NMVVUAKVE'
```

Kompakter definiert als Funktion ver für Verschlüsselung:

```
sage: def ver(z):       # verschlüssele Zeichenkette/string z
....:     return(''.join([keydict[i] for i in z]))
....:
sage: ver(msg)
'NMVVUAKVE'
```

Erstes Problem: Leerzeichen

```
sage: m2='HALLO WELT'
sage: ver(m2)
------------------------------------------------------
...
KeyError: ' '
sage:
```

Zweites Problem: Kleinbuchstaben

```
sage: m3="HalloWelt"
sage: ver(m3)
----------------------------------------------------
KeyError           Traceback (most recent call last)
...
KeyError: 'a'
sage:
```

4.2 String Monoids

Ausweg: Free String Monoids. Ein Monoid ist in der Algebra eine Menge von “Dingen”, die man assoziativ “verknüpfen” kann. Bei Buchstaben ist die Verknüpfung die Aneinanderreihung. https://doc.sagemath.org/html/en/reference/monoids/sage/monoids/string_monoid.html

```
sage: Buchstabenx=AlphabeticStrings()
sage: Buchstabenx
Free alphabetic string monoid on A-Z
sage: alph=Buchstabenx.alphabet()
sage: alph
('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')
sage: alph[0]
'A'
sage: type(alph)
<class 'tuple'>
```

Tupel (runde Klammern) haben, im Gegensatz zu Listen (eckige Klammern), keine überschreibbaren Einzelkomponenten:

```
sage: alph[0]='B'
---------------------------------------------------------
TypeError                 Traceback (most recent call last)
...
TypeError: 'tuple' object does not support item assignment
sage: type(Omega)
<class 'list'>
sage: Omega[0]
'A'
sage: Omega[0]='B'
sage: Omega[0]
'B'
```

Alternativ zu Buchstabenx.alphabet():

```
sage: Buchstabenx.gens()
(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)
sage: type(Buchstabenx.gens())
<class 'tuple'>
sage: type(Buchstabenx.gen(0))
<class 'sage.monoids.string_monoid_element.StringMonoidElement'>
```

Im Gegensatz zur Variante ('A','B',...,'Z') von weiter vorne, bei der die Buchstaben Strings der Länge 1 sind, sind die Buchstaben im obigen Listing erzeugende Elemente ("generators") des Monoids, also algebraisch abstrakte Elemente und keine Strings. Das merkt man auch, wenn man die Python built-in Funktion ord() aufruft, die für einen ASCII-Buchstaben seinen dezimalen ASCII-Wert zurückgibt:

```
sage: ord(Buchstabenx.gen(0))
---------------------------------------------
...
TypeError: ord() expected string of length 1, but StringMonoidElement found
sage: ord(Buchstaben[0])
65
sage: ord('A')
65
```

Die Klasse AlphabeticStrings hat eine Methode namens encoding(). Damit müssen wir das Problem mit Satzzeichen, Leerzeichen, Kleinbuchstaben etc. nicht händisch lösen, denn:

doc.sagemath.org/html/en/reference/monoids/sage/monoids/string_monoid.html

Amaz... Spons... Sage... Home... magm... PARI/... https:/... cross... begin{... Winte... https:/...

encoding(*s*)

The encoding of the string s in the alphabetic string monoid, obtained by the monoid homomorphism

```
A -> A, ..., Z -> Z, a -> A, ..., z -> Z
```

and stripping away all other characters. It should be noted that this is a non-injective monoid homomorphism.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: s = S.encoding("The cat in the hat."); s
THECATINTHEHAT
sage: s.decoding()
'THECATINTHEHAT'
```

Wirklich “stripping away all other characters”?

```
sage: P="Hello Wörld!"
sage: msg=Buchstabenx.encoding(P)
---------------------------------------------------------
...
ValueError: ’Ö’ is not in list
...
During handling of the above exception, another exception occurred:
...
TypeError: Argument x (= HELLOWÖRLD) is not a valid string.
sage:
```

4.3 Substitutionsverfahren von SageMath

Nicht nur für die Buchstabenmenge, sondern auch für die Substitutions-verschlüsselung gibt es “a convenient user interface” in SageMath:

```
sage: reset()   # deletes all user defined variables
sage: S=SubstitutionCryptosystem(AlphabeticStrings())
sage: key=S.random_key()
sage: key
ZGVBULOKFHTDRASYECNJXWMPIQ
sage: P="Das ist eine geheime Nachricht, die keine Umlaute und scharfe
 ss enthalten darf!" # this is a secret message with no umlauts
sage: msg=S.encoding(P)
sage: msg
DASISTEINEGEHEIMENACHRICHTDIEKEINEUMLAUTEUNDSCHARFESSENTHALTENDARF
sage: C=S.enciphering(key,msg)
sage: C
BZNFNJUFAUOUKUFRUAZVKCFVKJBFUTUFAUXRDZXJUXABNVKZCLUNNUAJKZDJUABZCL
sage: DC=S.deciphering(key,C)
sage: DC
DASISTEINEGEHEIMENACHRICHTDIEKEINEUMLAUTEUNDSCHARFESSENTHALTENDARF
```

Konvention im CrypTool-Buch:

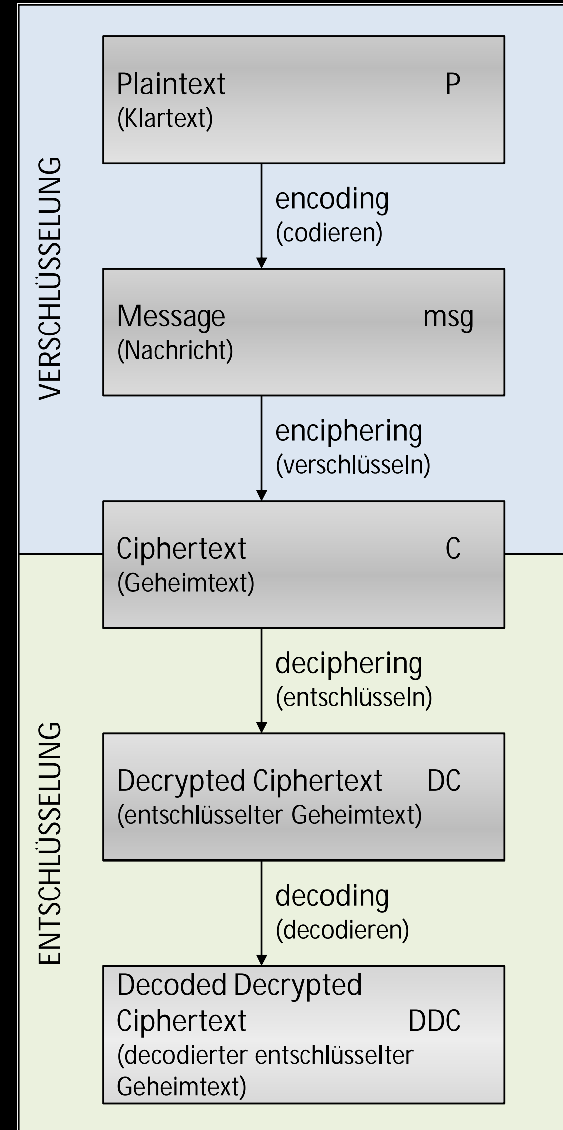
1. Kodieren: $P \longrightarrow msg$
2. Verschlüsseln: $msg \longrightarrow C$
3. Entschlüsseln: $C \longrightarrow DC$

   Der Schritt von C zu DC (wobei msg$\stackrel{!}{=}$DC), nennt man Entschlüsselung, engl. decryption oder deciphering. Der Begriff decryption wird manchmal vermieden, da es an Exhumierung erinnern könnte.

4. Dekodieren: $DC \longrightarrow DDC$

   Der Schritt von DC zurück zu DDC$\stackrel{!}{=}$P ist im obigen Beispiel weggelassen. Er ist z.B. nötig, wenn man msg, C und DC binär mit Nullen und Einsen darstellt.

Bemerkung: Bei einer (De-)Kodierung ist die Abbildung, die aus “vorher” das “nachher” macht, bekannt. Bei einer Verschlüsselung ist diese Abbildung geheim. Oft sind nicht alle Informationen über diese Abbildung geheim, aber zumindest Teilinformationen.



4.4 Spezialfall der Substitutionsverschlüsselung: zyklische Verschiebung

Wenn man die Buchstaben nicht beliebig permutiert, sondern (modulo 26) um 3 nach rechts schiebt: Caesar-Verschlüsselung

Wenn man um $k < 26$ nach rechts schiebt (auch modulo 26): Verschiebe-Chiffre, Shift Cipher

```
sage: reset()
sage: S=ShiftCryptosystem(AlphabeticStrings())
sage: key=3
sage: P="Gallia est omnis divisa in partes tres, quarum unam incolunt ▸
 ▸Belgae ..."
sage: msg=S.encoding(P)
sage: msg
GALLIAESTOMNISDIVISAINPARTESTRESQUARUMUNAMINCOLUNTBELGAE
sage: C=S.enciphering(key,msg)
sage: C
JDOOLDHVWRPQLVGLYLVDLQSDUWHVWUHVTXDUXPXQDPLQFROXQWEHOJDH
sage: DC=S.deciphering(key,C)
sage: DC
GALLIAESTOMNISDIVISAINPARTESTRESQUARUMUNAMINCOLUNTBELGAE
```

4.5 Die Hill-Verschlüsselung “zu Fuss”

Erst einmal der Weg “zu Fuss”:

Wir beginnen mit der Bijektion zwischen den 26 ASCII-Buchstaben und den Zahlen von 0 bis 25. Dieses Mal geht das A also nicht auf die 1, wie bei der Substitution von vorher.

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
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

```
sage: Buchstaben='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: bij={Buchstaben[i]:i for i in range(len(Buchstaben))}
sage: bij
{'A':0,'B':1,'C':2,'D':3,'E':4,'F':5,'G':6,'H':7,'I':8,'J':9,'K':10,'L
 ':11,'M':12,'N':13,'O':14,'P':15,'Q':16,'R':17,'S':18,'T':19,'U':20
 ,'V':21,'W':22,'X':23,'Y':24,'Z':25}
```

Der Schlüssel ist eine quadratische Matrix $K = \text{keymat}$ mit Einträgen aus $\mathbb{Z}_{26}$ (also modulo 26):

```
sage: R=Zmod(26)
sage: keymat=matrix(R,[[1,0,1],[0,1,1],[2,2,3]])
# this is the matrix from the sagemath doc webpage: https://doc.
  sagemath.org/html/en/reference/cryptography/sage/crypto/classical.
  html#sage.crypto.classical.HillCryptosystem
# not every matrix is a good choice, see later
sage: keymat
[1 0 1]
[0 1 1]
[2 2 3]
```

Die Nachricht - in ASCII-Großbuchstaben ohne Umlaute etc. - wird durch die über bij zugeordneten Zahlen ersetzt und zeilenweise in eine (auch 3×3-) Matrix geschrieben. Die Nachrichtenlänge muss durch 3 teilbar sein.

```
sage: msg='HALLOWELT'
sage: len(msg)%3==0
True
sage: msgzahl=[bij[c] for c in msg]
sage: msgzahl           # zahl (German) = number
[7, 0, 11, 11, 14, 22, 4, 11, 19]
sage: msgrowmat=matrix(R,3,msgzahl)
sage: msgrowmat
[ 7  0 11]
[11 14 22]
[ 4 11 19]
```

Nun wird die Nachrichtenmatrix msgrowmat von links an die Schlüsselmatrix heranmultipliziert. Vorsicht: Die Beispiele auf Wikipedia (https://de.wikipedia.org/wiki/Hill-Chiffre sowie https://en.wikipedia.org/wiki/Hill_cipher) schreiben die Nachricht sukzessive in die Spalten einer Matrix und multiplizieren diese von rechts an die Schlüsselmatrix.

```
sage: Cmat=msgrowmat*keymat
sage: Cmat
[ 3 22 14]
[ 3  6 13]
[16 23 20]
sage: Cmat.list()
[3, 22, 14, 3, 6, 13, 16, 23, 20]
```

Um nachher mit dem Ergebnis der in Sage implementierten Fertiglösung der Hillcipher zu vergleichen, brauchen wir statt der Liste von ganzen Zahlen < 26 wieder Buchstaben. Dazu drehen wir das Dictionary vom Anfang um und machen mit dessen Hilfe aus Cmat einen String:

```
sage: jib={value:key for (key,value) in bij.items()}
sage: jib
{0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11
  :'L',12:'M',13:'N',14:'O',15:'P',16:'Q',17:'R',18:'S',19:'T',20:'U
  ',21:'V',22:'W',23:'X',24:'Y',25: 'Z'}
sage: C=''.join([jib[i] for i in C.list()])
sage: C
'DWODGNQXU'
```

4.6 Hill-Verschlüsselung von SageMath

Nach dieser Lösung “zu Fuß” nun die wesentlich kürzere Variante von https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/classical.html#sage.crypto.classical.HillCryptosystem:

```
sage: keylen=3
sage: A=AlphabeticStrings()
sage: H=HillCryptosystem(A,keylen)
sage: HKS=H.key_space()
sage: key=HKS([[1,0,1],[0,1,1],[2,2,3]])
sage: key
[1 0 1]
[0 1 1]
[2 2 3]
sage: P="Hallo Welt!"
sage: msg=H.encoding(P)
sage: msg
HALLOWELT
sage: len(msg)%keylen==0
True
sage: C=H.enciphering(key,msg)
sage: C
DWODGNQXU # As expected the ciphertext $C$ from here and from before are the same.
sage:
```

4.7 Inverse Matrix

Der Weg zurück ist dann nur eine Zeile:

```
sage: DC=H.deciphering(key,C)
sage: DC
HALLOWELT
```

Die Entschlüsselung zu Fuß ist auch recht einfach: Man muss die inverse Matrix der Schlüsselmatrix (modulo 26 natürlich) berechnen und sie von rechts an die Matrix Cmat heranmultiplizieren.

```
#sage: R=Zmod(26)       # should still be in namespace
#sage: keymat=matrix(R,[[1,0,1],[0,1,1],[2,2,3]]) # like above
sage: keymat.inverse()  # shorter alternative: ~keymat
[25 24  1]
[24 25  1]
[ 2  2 25]
#sage: Cmat=matrix(R,[[3,22,14],[3,6,13],[16,23,20]])   # like above
sage: DC=Cmat*keymat.inverse()
sage: DC
[ 7  0 11]
[11 14 22]
[ 4 11 19]
```

Nun aus den Zahlen wieder Buchstaben machen und aus der Matrix einen Einzeiler:

```
sage: DCchars=''.join([jib[i] for i in DC.list()])      # jib s.o.
sage: DCchars
'HALLOWELT'
sage:
```

Nur der Vollständigkeit halber: Über $\mathbb{Q}$ funktioniert das so nicht!

```
sage: E=matrix(QQ,[[1,0,1],[0,1,1],[2,2,3]])
sage: ~E
[-1 -2  1]
[-2 -1  1]
[ 2  2 -1]
sage: F=matrix(QQ,[[25,24,1],[24,25,1],[2,2,25]])
sage: F*E
[27 26 52]
[26 27 52]
[52 52 79]
```

Bei der Wahl der Schlüsselmatrix K ist zu berücksichtigen, dass sie eine Inverse in $\mathrm{Mat}(n \times n, \mathbb{Z}_{26})$ haben muss. Das ist nur dann der Fall, wenn $\det(K) \notin \{0, 2, \ldots, 12, 13, 14, \ldots, 22, 24\}$ ist, wenn also die Determinante von K über $\mathbb{Z}$ zu 26 teilerfremd ist.

```
sage: det(E)
-1
sage: type(keymat)
<class 'sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_float'>
sage: type(E)
<class 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
sage: det(keymat)
25
```

4.8 Attacke auf die Hill-Cipher

Die Hill-Cipher ist im Normalfall sehr leicht zu brechen, wenn man einen Ciphertext und den zugehörigen Klartext kennt, aber den Schlüssel nicht. Man spricht dann von einer “known-plaintext-”Attacke (KPA). Was ist das?

https://www.informatik.hu-berlin.de/de/forschung/gebiete/algorithmenII/Lehre/ws05/krypto1/skript/kap2.pdf:

> Angriff bei bekanntem Klartext (known-plaintext attack)
>
> Der Gegner ist im Besitz von einigen zusammengehörigen Klartext-Kryptotext-Paaren. Hierdurch wird erfahrungsgemäß die Entschlüsselung weiterer Kryptotexte oder die Bestimmung der benutzten Schlüssel wesentlich erleichtert.

Wir holen von vorher noch einmal die Nachrichtenmatrix msgrowmat und die Ciphertextmatrix Cmat:

```
sage: msgrowmat
[ 7  0 11]
[11 14 22]
[ 4 11 19]
sage: Cmat
[ 3 22 14]
[ 3  6 13]
[16 23 20]
# if you don’t have it in your SageMath namespace anymore:
# msgrowmat=matrix(Zmod(26),[[7,0,11],[11,14,22],[4,11,19]]), same
  syntax for Cmat
sage: P=msgrowmat        # short P for plaintext matrix
sage: C=Cmat             # short C for ciphertext matrix
```

Die Verschlüsselung erfolgte über Matrixmultiplikation $C = PA$. Wenn P über $\mathbb{Z}_{26}$ invertierbar ist, kann man die inverse Matrix P^{-1} von links an diese Matrixgleichung heranmultiplizieren und erhält $P^{-1}C = A$:

```
sage: A=~P*C
sage: A
[1 0 1]
[0 1 1]
[2 2 3]
```

Um zu testen, ob eine (in Großbuchstaben als String übergebene) Nachricht s eine invertierbare Matrix liefert, muss man die Determinante berechnen und schauen, ob sie im Ring $\mathbb{Z}_{26}$ invertierbar ist, was - wie schon vorher erwähnt - nur dann der Fall ist, wenn die Determinante mit 26 keine gemeinsamen Teiler hat. Und natürlich müssen die Buchstaben eine 3×3-Matrix voll machen, also muss die Nachricht 9 Buchstaben haben:

```
sage: def test(s):                # s a capital letter string
....:     if len(s)!=9:
....:         print('length not =9')
....:     P=matrix(R,3,[bij[c] for c in s])
....:     d=P.det()
....:     if gcd(26,d)!=1:        # gcd greatest common divisor
....:         return(False)
....:     else:
....:         return(True)
....:
sage: test('HALLOWELT')
True
sage: test('NEINDOCHO')
False
```

Fehlermeldung dann:

```
sage: reset()   # begin again from the start
sage: Buchstaben='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: bij={Buchstaben[i]:i for i in range(len(Buchstaben))}
sage: R=Zmod(26)
sage: string="NEINDOCHO"
sage: P=matrix(R,3,[bij[c] for c in string])
sage: A=matrix(R,3,[[1,0,1],[0,1,1],[2,2,3]])
# A is supposed to be unknown
sage: C=P*A
# but we need A in order to compute C
sage: C
[ 3 20 15]
[15  5  6]
[ 4  9 25]
# now we forget A again and assume that only C and P are known
#                        und A is what we are searching for
sage: ~P        # Reminder: ~P is the inverse of P
----------------------------------------------------------
TypeError                Traceback (most recent call last)
...
   5795             except (TypeError, ZeroDivisionError):
...
   1493             if not b.gcd(d).is_one():
...
TypeError: matrix denominator not coprime to modulus
...
ZeroDivisionError: input matrix must be nonsingular
```

Die nicht invertierbare (Zahlen-)Matrix P, die zum Textstring "NEINDOCHO" gehört, ist:

```
sage: P
[13  4  8]
[13  3 14]
[ 2  7 14]
```

(Den Code für Umwandlung des Textstrings in die Zahlenmatrix haben wir weiter vorne auf Seite 21 schon beschrieben.)

Dass diese Matrix über $\mathbb{Z}_{26}$ nicht invertierbar ist, kann man schnell per Hand nachrechnen:

NRs: $26 \to 52 \to 78 \to 104 \to 130 \to 156 \to 182$; $13 \cdot 14 = 182$

$$\begin{pmatrix} 13 & 4 & 8 \\ 13 & 3 & 14 \\ 2 & 7 & 14 \end{pmatrix} \begin{matrix} \cdot 2 \\ \cdot 2 \\ \cdot 13 \end{matrix} \sim \begin{pmatrix} 0 & 8 & 16 \\ 0 & 6 & 2 \\ 0 & 13 & 0 \end{pmatrix} \text{ nicht invertierbar}$$

NR: $13 \cdot 7 \bmod 26 \equiv 91 \bmod 26 \equiv 13 \bmod 26$;

Am Ende des zweiten Kapitels “Papier- und Bleistift-Verschlüsselungsverfahren” des CrypTool-Buches findet man ein längeres SageMath-Beispiel zur KPA gegen die Hill-Verschlüsselung. Damit kann man z.B. größere Matrizen und längere Klartext-Ciphertext-Paare untersuchen.

5 XOR -Verschlüsselung

Auszug aus dem CrypTool-Buch Kapitel 8.3 “Bitstrom-Chiffren”:

“Die einfachste und gängigste Art von Bitstrom-Chiffren ist die XOR-Verschlüsselung. Hierbei wird der Klartext als Folge von Bits aufgefasst. Auch der Schlüssel ist eine Folge von Bits, die **Schlüsselstrom** genannt wird. Verschlüsselt wird, indem das jeweils nächste Bit des Klartexts mit dem nächsten Bit des Schlüsselstroms binär addiert wird.”

```
msg: 010001000111101 ...
key: 100101101001101 ...
     ------------------
  c: 110100101110000 ...
```

5.1 Spass mit Bits

Man hat die Möglichkeit, den Monoid aus Nullen und Einsen zu verwenden, analog zu dem schon vorher benutzten Monoid aus den 26 Großbuchstaben. Statt AlphabeticStrings() heißt es dann BinaryStrings():

```
sage: S=BinaryStrings()
sage: S
Free binary string monoid
sage: S.encoding('A')
01000001
sage: S.encoding('A',padic=True)
10000010                        # 8-Bit ASCII reversed
sage: ord('A')                  # decimal ASCII code of capital A
65
sage: bin(65)                   # with Pyhton built-in functions we can
'0b1000001'                     # also get the binary representation, as a
  string
sage: bin(65)[2:].zfill(8)      # cut 0b on the left, fill with zeroes
  (zero-fill) up to length 8
'01000001'
```

Die Binärdarstellung als String bekommt man statt mit bin(65) auch über binary(), als Liste über bits(). Aber auch hier muss man auf die Datentypen aufpassen:

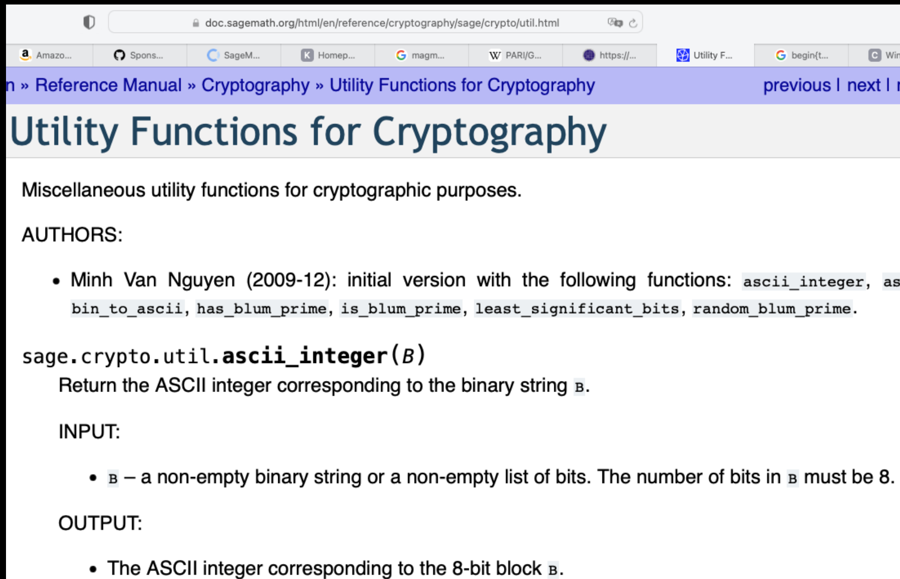
```
sage: 65.binary()
'1000001'                               # Alternative
sage: type(65)
<class 'sage.rings.integer.Integer'>
sage: type(ord('A'))
<class 'int'>
sage: 65.bits()
[1, 0, 0, 0, 0, 0, 1]
sage: ord('A').bits()                   # ord('A')=65
---------------------------------------------------------
AttributeError          Traceback (most recent call last)
...
AttributeError: 'int' object has no attribute 'bits'
sage: Integer(ord('A')).bits()
[1, 0, 0, 0, 0, 0, 1]
```

Wie man an dem AttributeError oben sieht, muss man aufpassen, ob man eine SageMath-Integer hat oder eine Python-Integer.

Leider ist der Weg zurück bei BinaryStrings() nicht implementiert:

```
sage: S.decoding(01000001)
---------------------------------------------------------
...
KeyError: 'decoding'
...
AttributeError: 'BinaryStringMonoid_with_category' object has no ▸
  ▸attribute 'decoding'
```

Es gibt aber den Befehl ascii_integer(), siehe https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/util.html:



```
sage: S=BinaryStrings()
sage: B=S.encoding('A')
sage: B
01000001
sage: ascii_integer(B)
---------------------------------------------------------
...     # don't forget to import :-)
NameError: name 'ascii_integer' is not defined
sage: from sage.crypto.util import ascii_integer
sage: ascii_integer(B)
65
sage: ascii_integer(01000001)
---------------------------------------------------------
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
sage: ascii_integer('01000001')
65
sage: type(B)
<class 'sage.monoids.string_monoid_element.StringMonoidElement'>
sage: type(01000001)
<class 'sage.rings.integer.Integer'>
sage: x=01000001
sage: x         # x is an int with base 10, not 2
1000001         # the leading zero is stripped away
```

Weitere Befehle aus cypto.util, die man importieren kann/muss:

- ascii_to_bin
- bin_to_ascii

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin('A')
01000001
sage: ascii_to_bin('Hallo, Welt')
0100100001100001011011000110110001101111001011000010000001010111011001▸
 ▸010110110001110100
sage: ascii_to_bin('HalloWelt')
0100100001100001011011000110110001101111010101110110010101101100011101▸
 ▸00
sage: ascii_to_bin(['H','a','lloWelt'])
0100100001100001011011000110110001101111010101110110010101101100011101▸
 ▸00
```

Nachteil: Darstellung der Bitketten als abstrakte Elemente eines Monoids, umständlich weiter zu verarbeiten. Wir vergleichen z.B. PLUS und MAL:

```
sage: from sage.crypto.util import ascii_to_bin
sage: ascii_to_bin('A')
01000001
sage: type(ascii_to_bin('A'))
<class 'sage.monoids.string_monoid_element.StringMonoidElement'>
sage: a=ascii_to_bin('A')
sage: b=ascii_to_bin('B')
sage: a+b                       # PLUS -> ERROR
------------------------------------------------------------
...
AttributeError: 'StringMonoidElement' object has no attribute '_add_'
...
TypeError: unsupported operand parent(s) for +: 'Free binary string monoid' and '
  Free binary string monoid'
sage: a*b                       # MAL  -> OK
0100000101000010
sage: a2=bin(ord('A'))[2:]
sage: b2=bin(ord('B'))[2:]
sage: a2+b2                     # PLUS -> OK
'10000011000010'
sage: a2*b2                     # MAL  -> EROOR
-------------------------------------------------------------
...
TypeError: can't multiply sequence by non-int of type 'str'
```

Umgekehrt von Binär zu ASCII:

```
sage: from sage.crypto.util import bin_to_ascii
sage: bin_to_ascii(01000001)            # like this it's a base 10 int
-----------------------------------------------------------
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
sage: bin_to_ascii('01000001')          # string: ok
'A'
sage: bin_to_ascii([0,1,0,0,0,0,0,1])   # list: also ok
'A'
sage: bin_to_ascii(3*[0,1,0,0,0,0,0,1])
'AAA'
```

Eine weitere Möglichkeit, von der Binärdarstellung zur ASCII-Darstellung zu kommen, findet man am Ende des RSA-Beispiels auf Seite 59.

5.2 Konversionsfunktionen aus dem CrypTool-Buch

Im CrypTool-Buch im Abschnitt 8.4. “Anhang: Boolesche Abbildungen in SageMath” werden nützliche Konversionsfunktionen definiert. Wenn dort von einem Bitblock die Rede ist, ist das immer eine Liste.

Unter dem Link https://www.cryptool.org/de/documentation/ctbook/sagemath findet man eine Datei namens “bitciphers.sage”. Der folgende Code ist von dort:

```
################################################################
# Sage module 'bitciphers.sage'                                #
# ------------------------------------------------------------ #
# Klaus Pommerening (Johannes-Gutenberg-Universitaet Mainz)    #
#   2014-Dec-31, last version 2015-Aug-11                      #
# ...
# We refrain from defining a class "Bitblock", avoiding object ▸
  ▸oriented
# overhead and type conversion struggles.
# ...
################################################################
##### Conversion routines for bitblocks ########################
################################################################
```

```
def int2bbl(number,dim):                       # example ctd.
  """Converts number to bitblock of length dim via base-2 ▸
   ▸representation."""
  n = number                                 # catch input
  b = []                                     # initialize output
  for i in range(0,dim):
    bit = n % 2                              # next base-2 bit
    b = [bit] + b                            # prepend
    n = (n - bit)//2
  return b

def bbl2int(bbl):
  """Converts bitblock to number via base-2 representation."""
  ll = len(bbl)
  nn = 0                                     # initialize output
  for i in range(0,ll):
    nn = nn + bbl[i]*(2**(ll-1-i))   # build base-2 representation
  return nn                                  # return base-10 int
```

```
def str2bbl(bitstr):                        # example ctd.
  """Converts bitstring to bitblock."""
  ll = len(bitstr)
  xbl = []
  for k in range(0,ll):
    xbl.append(int(bitstr[k]))
  return xbl

def bbl2str(bbl):
  """Converts bitblock to bitstring."""
  bitstr = ""
  for i in range(0,len(bbl)):
    bitstr += str(bbl[i])
  return bitstr

def txt2bbl(text):
  """Converts ASCII-text to bitblock."""
  ll = len(text)
  xbl = []
  for k in range(0,ll):
    n = ord(text[k])
    by = int2bbl(n,8)
    xbl.extend(by)
  return xbl
```

Test: Nochmal der Link, dort ganz unten: https://www.cryptool.org/de/documentation/ctbook/sagemath

```
sage: load('./bitciphers.sage') # download
sage: thebeast=int2bbl(6,66); thebeast
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0▶
 ▶,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0]
sage: type(thebeast)
<class 'list'>
sage: type(thebeast[0])
<class 'sage.rings.integer.Integer'>
sage: number(thebeast[0])
---------------------------------------------------
...
NameError: name 'number' is not defined
sage: txt2bbl('the number of the beast')
[0,1,1,1,0,1,0,0,0,1,1,0,1,0,0,0,0,1,1,0,0,1,0,1,0,0,1,0,0,0,0,0,0,1,1▶
 ▶,0,1,1,1,0,0,1,1,1,0,1,0,1,0,1,1,0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1,0,▶
 ▶0,1,0,1,0,1,1,1,0,0,1,0,0,0,1,0,0,0,0,0,0,1,1,0,1,1,1,1,0,1,1,0,0,1▶
 ▶,1,0,0,0,1,0,0,0,0,0,0,1,1,1,0,1,0,0,0,1,1,0,1,0,0,0,0,1,1,0,0,1,0,▶
 ▶1,0,0,1,0,0,0,0,0,0,1,1,0,0,0,1,0,0,1,1,0,0,1,0,1,0,1,1,0,0,0,0,1,0▶
 ▶,1,1,1,0,0,1,1,0,1,1,1,0,1,0,0]
sage: len(txt2bbl('the number of the beast'))
184
```

5.3 XOR aus dem CrypTool-Buch

Für die XOR-Verschlüsselung kann man damit dann folgende Funktion - entnommen aus derselben Datei wie oben - verwenden:

```
#########################################################
##### XOR of bitblocks ##################################
#########################################################
def xor(plain,key):     #input two lists of 0’s and 1’s
  """Binary addition of bitblocks.
  Crops key if longer than plain.
  Repeats key if shorter than plain.
  """
  lk = len(key)
  lp = len(plain)
  ciph = []
  i = 0
  for k in range(0,lp):
    cbit = (plain[k] + key[i]) % 2
    ciph.append(cbit)
    i += 1
    if i >= lk:
      i = i-lk
  return ciph
```

Bei diesem Codebeispiel wiederholt sich also der Schlüssel, falls man eine "sehr" lange Nachricht mit einem vergleichsweise "kurzen" Schlüssel verschlüsseln will. Dann ist das Verfahren angreifbar.

Ist die Nachricht kürzer als der Schlüssel und wird der Schlüssel nach einmaliger Verwendung vernichtet, ist das Verfahren ein Beispiel für perfekte Sicherheit nach Shannon, Stichwort One-Time-Pad.

Wenn das One-Time-Pad so perfekt ist – warum wird es denn nicht generell verwendet?

- Unhandliches Schlüssel-Management: Die Vereinbarung eines Schlüssels wird zum Problem – er ist ja ebenso lang wie der Klartext und schwer zu merken. Die Kommunikationspartner müssen also im Voraus den Schlüsselstrom vereinbaren und aufzeichnen. Wollen sie die Schlüssel erst bei Bedarf vereinbaren, benötigen sie dazu einen sicheren Kommunikationsweg, aber dann können sie den (auch nicht längeren) Klartext gleich direkt verschicken.

- Keine Eignung zur Massenanwendung: Das Verfahren ist bestenfalls zur Kommunikation zwischen *zwei* Partnern geeignet, wegen des Aufwands bei der Schlüsselverwaltung nicht für eine Mehr-Parteien-Kommunikation.

Screenshot aus dem CrypTool-Buch, Kapitel Bitstrom-Chiffren.

Beispiel der Verwendung eines periodischen Schlüssels:

Beispiel: Schlüsselfolge der Periode 8 mit $k = 10010110$. Die Buchstaben werden nach dem ISO-Zeichensatz durch Bytes repräsentiert.

```
       D    |   u    |        |   b    |   i    |   s    |
a: 01000100|01110101|00100000|01100010|01101001|01110011|
k: 10010110|10010110|10010110|10010110|10010110|10010110|
   -------- -------- -------- -------- -------- --------
c: 11010010|11100011|10110110|11110100|11111111|11100101|

       t    |        |   d    |   o    |   o    |   f
   01110100|00100000|01100100|01101111|01101111|01100110
   10010110|10010110|10010110|10010110|10010110|10010110
   -------- -------- -------- -------- -------- --------
   11100010|10110110|11110010|11111001|11111001|11110000
```

Das kann man leicht per Hand ausführen, aber auch mit dem SageMath-Beispiel 8.3.1 nachvollziehen.

Werden in diesem Beispiel die Geheimtext-Bytes in Zeichen des ISO-9960-1-Zeichensatzes zurückgewandelt, sieht der Geheimtext so aus

Ò ã ¶ ô œ å â ¶ ò ù ù ð

und kann Laien vielleicht beeindrucken. Einem Fachmann fällt sofort auf, dass alle Zeichen in der oberen Hälfte der möglichen 256 Bytes liegen. Das legt die Vermutung nahe, dass ein gewöhnlicher Text mit einem Schlüssel behandelt wurde, dessen Leitbit eine 1 ist. Versucht er, das auffällig wiederholte Zeichen ¶ = 10110110 als Leerzeichen 00100000 zu deuten, kann er sofort den Schlüssel als Differenz 10010110 bestimmen und hat die Verschlüsselung gebrochen.

Screenshot aus dem CrypTool-Buch, Kapitel Bitstrom-Chiffren.

Reproduktion der “merkwürdigen” ASCII-Zeichen vom Screenshot:

```
sage: load('./bitciphers.sage') # perhaps still in namespace: txt2bbl
  und bbl2str
sage: plaintext="Du bist doof"  # du bist doof = you're dumb
sage: bintext=txt2bbl(plaintext)
sage: binstr=bbl2str(bintext)
sage: binstr
'010001000111010100100000011000100110100101110011011101000010000001100
  10001101111011011110110 0110'
sage: testkey=[1,0,0,1,0,1,1,0]
sage: altkey=[0,0,0,1,0,1,1,0]
sage: keystr=bbl2str(testkey)
sage: altkeystr=bbl2str(altkey)
sage: keystr
'10010110'
sage: altkeystr          # for later comparison a key
'00010110'               # with leading bit 0 instead of 1
sage: ciphertext=xor(bintext,testkey)
sage: ciphstr=bbl2str(ciphertext)
sage: from sage.crypto.util import bin_to_ascii
sage: bin_to_ascii(ciphstr)
'Òãűôßåâűòùùð'
```

```
sage: altciphertext=xor(bintext,altkey)
sage: altciphstr=bbl2str(altciphertext)
sage: bin_to_ascii(altciphstr)
'Rc6t\x7feb6ryyp'
```

Nicht druckbare ASCII-Zeichen werden mit \x eingeleitet, gefolgt von zwei HEX-Ziffern. Sie haben in der Binärkodierung als Leitbit eine Null.

```
sage: x=bin_to_ascii(ciphstr); x
'Òãűôßåâűòùùð'
sage: x[2]
'ű'
sage: ord(x[2])
182             # not in the 7 bit ASCII set since  > 2^7=128
sage: bin(ord(x[2]))
'0b10110110'
sage: y=bin_to_ascii(altciphstr)
sage: y
'Rc6t\x7feb6ryyp'
sage: bin(ord(y[4]))
'0b1111111'
sage: ord(y[4])
127     # the ASCII character 127 stands for DELETE.
sage: y[4]
'\x7f'
```

5.4 XOR mit Pythonmitteln: das “Dach” ^

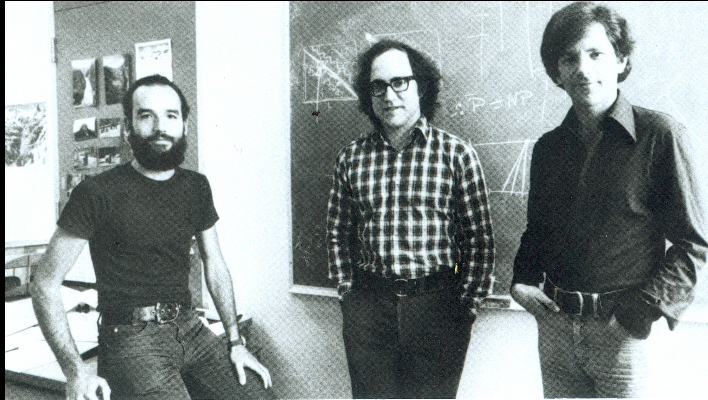
```
SageMath version 9.6,
Using Python 3.10.3.

sage: 2^2
4
sage: 2^1
2
```

```
dorisbehrendt@
Python 3.8.9 (
[Clang 13.1.6
Type "help", "
>>> 2^2
0
>>> 2^1
3
```

Ob und falls ja wie man aus Sagemath heraus auf dieses einfache Python-XOR zugreifen kann, weiß ich nicht. Eine Internetsuche lieferte auf die Schnelle jedenfalls nichts.

6 RSA



https://cryptologicfoundation.org/what-we-do/educate/bytes/this_day_in_history_calendar.html/event/2020/09/20/1600578000/1983-three-inventors-receive-patent-for-encryption-algorithm-rsa/78258
https://www.zib.de/node/2931

6.1 Theorie ganz kurz

1. die Person, die verschlüsselte Nachrichten empfangen will (Bob), wählt zwei verschiedene Primzahlen p, q und bildet ihr Produkt n,
2. berechnet die Anzahl r der zu n teilerfremden Zahlen mit Hilfe der Eulerschen ϕ-Funktion oder der Formel $r = \phi(pq) = (p-1)(q-1)$ für den Fall des Produktes zweier verschiedener Primzahlen,
3. sucht in $\mathbb{Z}_r$ ein multiplikativ invertierbares Element e; diese Bedingung ist erfüllt, wenn e und r teilerfremd sind,
4. Bob berechnet in $\mathbb{Z}_r$ das Inverse d zu e (also $\mathrm{mod} r$)
5. die Zahl n und die Zahl e werden veröffentlicht, p, q und d hält Bob geheim;
6. die Person, die Bob eine geheime Nachricht schreiben will (Alice), besorgt sich n und e;
7. Alice bereitet den Klartext so auf, dass er als natürliche Zahl $msg < n$ vorliegt,
8. berechnet $C = msg^e$ in $\mathbb{Z}_n$ (also $\mathrm{mod} n$); C ist Geheimtext, computes $C = msg^e$ in $\mathbb{Z}_n$ (also $\mathrm{mod} n$); C is the ciphertext,
9. Alice verschickt C an Bob
10. Bob berechnet $DC = C^d$ in $\mathbb{Z}_n$; DC ist der dechiffrierte Geheimtext, also wieder der Klartext, kodiert als natürliche Zahl $< n$

Das Verfahren funktioniert unter der Annahme, dass man n nicht “schnell” faktorisieren kann und deshalb r nicht kennt und deshalb das Inverse d zu e nicht “schnell” finden kann.

6.2 kleines RSA-Beispiel

```
sage: p,q=random_prime(2^10),random_prime(2^10)
sage: p==q
False
sage: n=p*q
sage: Zn=Zmod(n)
sage: r=euler_phi(n)
sage: Zr=Zmod(r)
sage: e=ZZ.random_element(r)    # e=exponent=key for enciphering
sage: while gcd(e,r)!=1:
....:     e=ZZ.random_element(r)
....:
sage: d=Zr(e)^-1 # d=inverse of e in Zmod(r), key for deciphering
sage: msg=Zn.random_element(); msg
13492
sage: C=msg^e; C        # C ciphertext
113247
sage: n,p,q,r,e,d
(371323, 449, 827, 370048, 75447, 56071)
sage: C^d       # C^d deciphered ciphertext must be = msg
13492
```

6.3 großes RSA-Beispiel

Ein Beispiel mit 512-Bit-Primzahlen zu RSA findet man im Kapitel 12.10 “Gitter und RSA” des CrypTool-Buches. Hier ein leicht verändertes Beispiel:

```
sage: P='6, 66, the number of the beast!'
sage: P_ascii=[ord(x) for x in P]
sage: P_ascii
[54,44,32,54,54,44,32,116,104,101,32,110,117,109,98,101,114,32,111,102
 ,32,116,104,101,32,98,101,97,115,116,33]
sage: P_bin=[bin(x)[2:].zfill(8) for x in P_ascii]
sage: P_bin
['00110110','00101100','00100000','00110110','00110110','00101100','00
 100000','01110100','01101000','01100101','00100000','01101110','011
 10101','01101101','01100010','01100101','01110010','00100000','0110
 1111','01100110','00100000','01110100','01101000','01100101','00100
 000','01100010','01100101','01100001','01110011','01110100','001000
 01']
sage: msg=Integer(''.join(P_bin),2)
sage: msg
95714286764733858046056695692827363370527019483767976502265996337803916
 6321
sage: b=512
sage: p=random_prime(2^b-1,lbound=2^(b-1)+2^(b-2))
```

```
sage: q=random_prime(2^b-1,lbound=2^(b-1)+2^(b-2))
sage: p==q
False
sage: p.nbits(),q.nbits()
(512, 512)
sage: n=p*q
sage: n.nbits()
1024
sage: e=2^16+1  # not chosen, but recommended
sage: e
65537
sage: r=(p-1)*(q-1)
sage: r
1078101730075069437368249624814042708209942270324089537039271223049820▸
  ▸9592853280628250065667233723248626529626825317507769243714647383959▸
  ▸6022567571336626924177327605367345646688142878478885417505711992109▸
  ▸6541659234522765019081713848515257102384986914185246863707336492867▸
  ▸48037129287663764457388728530764618700
sage: %time factor(r)   # better don't do it ;-) see last slide
sage: r.nbits()
1024
sage: d=inverse_mod(e,r)
sage: d
```

```
8623399116875233070827320991558498681214181874766787863517652132916469▸
  ▸2474016482874330026144324428401704580552635910870923527895010530623▸
  ▸3593087058766394554572179135596933967840935323976021555925191622805▸
  ▸4657722924900112160349500534510016107561133257321028707814255503395▸
  ▸9730447753003985483326587092958360573
sage: d<r
True
sage: e*d%r
1
sage: C=power_mod(msg,e,n)
sage: C                              # ciphertext
3246479944239379864634999053827420237289087458176767592603875320470692▸
  ▸7757392847655933668782406412580886579102151639046041304200199548850▸
  ▸7741696376750325385534196645153779011176347918610309707160185759293▸
  ▸8755969799410205040151415410793474190454305304887097766773210186375▸
  ▸4005218691768147036380446587935983872
sage: DC=power_mod(C,d,n)            # decrypted ciphertext
sage: DC
9571428676473385804605669569282736337052701948376797650226599633780391▸
  ▸6321
sage: DC_bin=bin(DC)[2:]
sage: while len(DC_bin)%8!=0:
....:     DC_bin='0'+DC_bin
....:
```

```
sage: DC_ascii=[chr(int(DC_bin[x*8:8*(x+1)],2)) for x in range(len(
DC_bin)/8)]
sage: StringDC=''.join(DC_ascii)
sage: StringDC
'6, 66, the number of the beast!'
```

6.4 Faktorisierung

Und nebebnei, wenn man die 1024 Bit lange Zahl r faktorisieren will, könnte das viele Jahre lang dauern, je nach r ;-)

4.12.4 Status regarding factorization of concrete large numbers

An exhaustive overview about the factoring records of composed integers using different methods can be found on the following web pages:

- http://primerecords.dk/consecutive_factorizations.htm
- https://en.wikipedia.org/wiki/Integer_factorization_records
- https://en.wikipedia.org/wiki/RSA_Factoring_Challenge

Screenshot aus dem CrypTool-Buch.

Natürlich habe ich es trotzdem probiert:

```
sage: P
'6, 66, the number of the beast!'
sage: r
1078101730075069437368249624814042708209942270324089537039271223049820959285328062825006566723372324
8626529626825317507769243714647383959602256757133662692417732760536734564668814287847888541750571199
2109654165923452276501908171384851525710238498691418524686370733649286748037129287663764457388728530
764618700
sage: % time factor(20)
UsageError: Line magic function `%` not found.
sage: %time factor(20)
CPU times: user 1.25 ms, sys: 2.21 ms, total: 3.45 ms
Wall time: 30.4 ms
2^2 * 5
sage: %time factor(r)
  ***   Warning: MPQS: number too big to be factored with MPQS,
        giving up.
```

```
Processes: 539 total, 3 running, 536 sleeping, 2064 threads                  08:28:2
Load Avg: 2.58, 2.59, 2.60  CPU usage: 18.69% user, 2.43% sys, 78.87% idle
SharedLibs: 884M resident, 129M data, 75M linkedit.
MemRegions: 90971 total, 4486M resident, 696M private, 4810M shared.
PhysMem: 28G used (1864M wired), 3139M unused.
VM: 206T vsize, 3823M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 678585/768M in, 340635/49M out. Disks: 500727/15G read, 812171/13G written.
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG	CMPR	PGRP	PPID	STATE
4951	python3.10	100.6	21:20:10	2/1	0	26	161M	224K	0B	4950	4950	running
106	systemstats	47.1	02:28.93	5	4	103-	4961K-	0B	0B	106	1	sleeping
162	WindowServer	23.2	02:05:48	26	4	2763-	625M-	1060M-	0B	162	1	sleeping
5679	com.apple.pr	15.2	01:46:42	15	3	342	105M	8896K	0B	5679	1	sleeping
212	coreaudiod	14.5	06:42.85	10	1	283	28M	0B	0B	212	1	sleeping

Leider habe ich die genaue Zeit nicht geloggt, aber bevor der Rechner aufgab, hat er es ca. einen halben Tag lang versucht.

But 330 bit is possible on a FroSCon afternoon on my MBP:

```
sage: rsa100
1522605027922533360535618378132637429718068114961380688657908494580122
 9632589528976540003506920061339
sage: rsa100.factor(algorithm='qsieve')
37975227936943673922808872755445627854565536638199 * 40094690950920881
 030683735292761468389214899724061
```

Es hat länger als 8 Stunden gedauert, aber kürzer als einen Tag ;-)

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro18,2
Chip:	Apple M1 Max
Total Number of Cores:	10 (8 performance and 2 efficiency)
Memory:	32 GB