1 Praktischer Teil

Nachfolgend wird der Aufbau und die Funktionsweise von RSA anhand meines selbst programmierten Python Codes erklärt.

Der Algorithmus kann in die drei folgenden beschriebenen Bereiche abgegrenzt werden:

- Schlüsselerzeugung
- Verschlüsselung
- Entschlüsselung

1.1 Schlüsselerzeugung

In meinem Code werden der öffentliche sowie der private Schlüssel eines jeden Teilnehmers mit der Funktion $schluesselerzeugung()$ berechnet.

1.1.1 Öffentlicher Schlüssel

Der öffentliche Schlüssel E_T jedes Teilnehmers, welcher nachfolgend generiert wird, besteht aus einem Teil n und einem zweiten Teil e:

n berechnen Der erste Teil n ist das Produkt aus zwei grossen Primzahlen p und q:

$$\boxed{n = p \cdot q}$$

Zum Thema Primzahlensuche stellen sich vorab gleich mehrere zu klärende Fragen: [Albrecht Beutelspacher, 2010, S. 117 ff.]

Gibt es überhaupt 300-stellige Primzahlen?
Der *Satz des Euklid* besagt, dass es zu jeder abgeschlossenen Primzahlenliste sicherlich noch eine weitere Primzahl gibt, welche noch nicht in der Liste aufgeführt ist. Somit gibt es unendlich viele Primzahlen. Dass es mehr als genug Primzahlen in der gewünschten Grössenordnung von etwa 300 Stellen gibt, zeigt der *Primzahlsatz*. Mit diesem kann die

ungefähre Anzahl an Primzahlen in einem gewissen Zahlenbereich berechnet werden, bei 300-stelligen Zahlen sind es ca. 2^{1013}.

Wie finde ich heraus, ob eine Zahl eine Primzahl ist?
Laut Definition hat eine Primzahl genau zwei Teiler, nämlich 1 und sich selbst. Um Primzahlen zu finden, gibt es leider keine Formel, sondern man wählt einfach eine Zahl aus und überprüft, ob es sich um eine Primzahl handelt.

```
faktor=2.0
primzahlen=[]
testzahl=2
while (testzahl<100000):
    if (testzahl/faktor)==int(testzahl/faktor) and faktor<testzahl:
        testzahl=testzahl+1
        faktor=2.0
    else:
        if(faktor>=testzahl**0.5):
            primzahlen.append(testzahl)
            testzahl=testzahl+1
            faktor=2.0
        else:
            faktor=faktor+1

import random
p= random.choice(primzahlen)
q= random.choice(primzahlen)
```

Abbildung 1: Erstellen einer Primzahlenliste und Aussuchen von random-Elementen daraus

Um in Python eine Liste mit allen Primzahlen bis zu einer gewissen Zahl zu erstellen, wird (vgl. Abb.7) bei allen *Testzahlen* überprüft, ob der, nach jeder Überprüfungsrechnung um 1 steigende *Faktor* ein Teiler der *Testzahl* ist. Falls dies eintrifft und der Teiler nicht gleich der *Testzahl* ist, kann diese aktuelle *Testzahl* laut Definition keine Primzahl sein und erhöht sich um 1. So geht die Suche immer weiter, bis eine *Testzahl* gefunden wird, bei der kein *Faktor*, welcher kleiner als die zweite Wurzel der *Testzahl* ist, ein Teiler ist. Alle auf diesem Weg gefundenen Primzahlen werden zur *primzahlen* - Liste hinzugefügt. Zum Schluss werden den Variablen p und q zwei random Primzahlen

aus der Liste zugeordnet.

Welche Eigenschaften sollten die Primzahlen für eine möglichst grosse Sicherheit haben?
Neben der möglichst vielstelligen Länge der zwei Primzahlen p und q, ist auch ihr Abstand zueinander wichtig. Eine ungünstige Wahl würde einem Hacker die Faktorisierung des Primzahlprodukts um einiges leichter machen.

Die beiden Primzahlen p und q dürfen einerseits nicht zu weit auseinander liegen. Sonst könnte das sogenannte *Zahlkörpersieb* eventuell zu einer schnellen Faktorisierung führen. In der vorher erstellten Primzahlenliste tauchen nur Zahlen 2 bis höchstens 99991 auf. Somit liegen alle möglichen random Primzahlenkombinationen höchstens 99′989 Zahlen auseinander. Im Vergleich zu den sonst bei RSA üblichen 300-stelligen Zahlen, liegt jede Primzahlkombination der Liste ziemlich nahe beisammen. Diese Sicherheitsmassnahme wurde im Code deshalb nicht berücksichtigt.

Andererseits darf man p und q aber auch nicht zu nahe beieinander wählen. Unter diesen Umständen könnte nämlich ein Hackerangriff mit der *Fermat-Faktorisierung* gelingen. Bei dieser Methode [*Exkurs: Die Fermatsche Faktorisierungsmethode fur RSA-Zahlen* o. D.] will man das bekannte Primzahlenprodukt n, als

$$\boxed{n = a^2 - b^2 \rightarrow n = (a+b) \cdot (a-b)} \tag{1}$$

schreiben. Die gesuchten zwei Primzahlen sind somit $p = (a+b)$ und $q = (a-b)$.

Um n wie in (1) als Differenz zweier Quadratzahlen zu schreiben, führt man folgende Umformungen durch.

$$\begin{aligned} n &= a^2 - b^2 && | -n + b^2 \\ a^2 - n &= b^2 \\ \rightarrow a^2 &> n && |\sqrt{*} \\ a &> \sqrt{n} \end{aligned} \tag{2}$$

Man berechnet $\sqrt{n}$ und setzt a als nächst grössere natürliche Zahl an.

Beispiel:

$$\begin{aligned} n &= 989 \\ \sqrt{n} &\approx 31.45 \\ \rightarrow a_1 &= 32 \end{aligned}$$

$a^2 - n$ muss laut (2) eine Quadratzahl b^2 ergeben. Ist dies mit a_1 nicht der Fall, testet man weiter mit $a_2 = a_1 + 1$:

$$\begin{aligned} 32^2 - 989 &= 35 \\ 35 &\neq Quadratzahl \\ \rightarrow a_2 &= 32 + 1 = 33 \\ 33^2 - 989 &= 100 \\ 100 &= Quadratzahl \\ b &= 10 \end{aligned}$$

Sobald man eine Zahl a gefunden hat, welche die Gleichung (2) erfüllt, kann man die beiden Primzahlen p und q ganz einfach berechnen:

$$\begin{aligned} p &= (a + b) = 33 + 10 = 43 \\ q &= (a - b) = 33 - 10 = 23 \\ \rightarrow n &= 989 = 43 \cdot 23 \end{aligned}$$

Diese Fermat-Faktorisierungsmethode arbeitet also besonders schnell, wenn die Primzahlen p oder q nahe an $\sqrt{n}$ sind. In meinem Code (vgl. Abb. 8) habe ich durch die Bedingung

```
while (abs(p-((q*p)**0.5))<300) or (abs(q-((q*p)**0.5))<300) or (p==q):
    q=random.choice(primzahlen)
    p=random.choice(primzahlen)
```

Abbildung 2: Bedingung an Primzahlen wegen Fermat-Faktorisierung

sichergestellt, dass keine der beiden Primzahlen näher als 300 an $\sqrt{n}$ liegt.

e berechnen Der zweite Teil e kann grundsätzlich frei aus den natürlichen Zahlen gewählt werden. Die Zahl muss nur teilerfremd zur *Eulerschen Phi-Funktion* von n

$$\phi(n) = (p-1) \cdot (q-1)$$

sein. Die Eulersche Phi-Funktion $\phi(n)$ berechnet die Anzahl aller zu n teilerfremden Zahlen, welche kleiner sind als n. [Albrecht Beutelspacher, 2010, S. 115 f.]

In meinem Code (vgl. Abb. 9) werden Zahlen von 2 ausgehend getestet, bis eine Zahl e gefunden wird, welche teilerfremd zu $\phi(n)$ ist:

```
phi = (p-1)*(q-1)

e=2
faktor=2.0
while(faktor<=e):
    if(phi/faktor)==int(phi/faktor) and (e/faktor)==int(e/faktor):
        e=e+1
        faktor=2.0
    else:
        faktor=faktor+1
```

Abbildung 3: Zweiten Teil e des privaten Schlüssels berechnen

Nur diese zwei Zahlen n und e werden für die Verschlüsselung gebraucht und somit veröffentlicht. Die Primzahlen p und q sowie $\phi(n)$, welche für die Berechnung des öffentlichen Schlüssels E_T verwendet werden, bleiben jedoch geheim.

1.1.2 Privater Schlüssel

Der private Schlüssel D_T jedes Teilnehmers besteht aus der Zahl d und wird zur Entschlüsselung einer Nachricht benötigt. Dieser wird durch das geheime $\phi(n)$ und dem Teil e des öffentlichen Schlüssels berechnet.

Der *Euklidische Algorithmus* findet den grössten gemeinsamen Teiler von zwei natürlichen Zahlen. [ebd., S. 120]

Das *Lemma von Bézout* zeigt, dass man zu jedem ggT zweier Zahlen, eine Vielfachsummendarstellung finden kann.[Albrecht Beutelspacher, 2010, S. 121] Dazu ein Beispiel:

$$\begin{aligned} a = 25, b = 10 \\ ggT(a,b) = 5 \\ 5 = 1 \cdot 25 - 2 \cdot 20 \end{aligned}$$

Diese letzte Gleichung ist dann eine Vielfachsummendarstellung.

Die Kombination aus Euklidischem Algorithmus (also das Suchen des ggT) und dem Lemma von Bézout (die Vielfachsummendarstellung) nennt man den *erweiterten Euklidischen Algorithmus*. Diesen wendet man nun auf die Zahlen $\phi(n)$ und e an.

Da man e extra teilerfremd zu $\phi(n)$ gewählt hat, ist der ggT der beiden Zahlen 1. Für die Vielfachsummendarstellung muss man also zwei natürliche Zahlen d und k finden, für die gilt [ebd., S. 121]:

$$e \cdot d + k \cdot \phi(n) = 1 \tag{3}$$

Wie die gesuchten Zahlen d und k mit dem erweiterten Euklidischen Algorithmus ermittelt werden, zeigt folgendes Beispiel:

$$\begin{aligned} p = 5, q = 13 \rightarrow n = 65 \\ \phi(n) = 48 \rightarrow e = 5 \\ ggT(48,5) = 1 \end{aligned}$$

$$\begin{aligned} 48 = 9 \cdot 5 + 3 \\ 5 = 1 \cdot 3 + 2 \\ 3 = 1 \cdot 2 + 1 \end{aligned}$$

Abbildung 4: erweiterter Euklidischer Algorithmus an konkretem Zahlenbeispiel 1

Diese Schritte werden wiederholt, bis nur noch eine 1 als Rest addiert (vgl. Abb. 10) wird. Die Gleichungen werden jeweils auf die ganz rechts stehenden Reste aufgelöst (vgl. Abb. 11):

$$3 = 48 - 9 \cdot 5$$
$$2 = 5 - 1 \cdot 3$$
$$1 = 3 - 1 \cdot 2$$

Abbildung 5: erweiterter Euklidischer Algorithmus an konkretem Zahlenbeispiel 2

Nun werden die Zahlen der oberen Gleichung in die untere Gleichung (vgl. Abb. 11) wie folgt eingesetzt (vgl. Abb. 12):

$$1 = 3-1\cdot(5-1\cdot3) = 3-1\cdot5+1\cdot3 = -1\cdot5+2\cdot3 = -1\cdot5+2\cdot(48-9\cdot5) = -1\cdot5+2\cdot48-18\cdot5 = 2\cdot48-19\cdot5$$

Abbildung 6: erweiterter Euklidischer Algorithmus an konkretem Zahlenbeispiel 3

Die gesuchten Zahlen d und k konnten also ermittelt werden:

$$e \cdot d + k \cdot \phi(n) = 1 \quad \leftrightarrow \quad 5 \cdot d + k \cdot 48 = 1 \quad \leftrightarrow \quad 5 \cdot (-19) + 2 \cdot 48 = 1$$
$$\rightarrow d = -19, k = 2$$

Der private Schlüssel d lautet -19.

In meinem Code habe ich den erweiterten Euklidischen Algorithmus mit Hilfe einer Art Tabelle (vgl. Tabelle 1) mit einer gewissen Ausfüll-Vorschrift durchgeführt. [**Dalwigk2021**] Die Tabelle besteht aus 5 Spalten und wird zeilenweise ausgefüllt.

Tabelle 1: Beispiel erweiterter Euklidischer Algorithmus 1

z (=Zeile)	a	b	qu (=Quotient)	r (=Rest)
1	a_1	b_1	qu_1	r_1
2	$a_2 = b_1$	$b_2 = r_1$	qu_2	r_1

In der Spalte qu wird jeweils eingetragen, wie viele ganze Male b in a Platz hat. In der Spalte r wird der übriggebliebene Rest notiert. Nachdem eine komplette Zeile ausgefüllt ist, wird das b und das r der ausgefüllten Zeile zum a und b der nächsten Zeile. Mit

dieser Ausfüll-Vorschrift wird fortgefahren, bis beim r die Zahl 0 steht. Nachfolgend ein Beispiel (vgl. Tabelle 2) mit den Zahlen $a = 48$ und $b = 5$:

Tabelle 2: Beispiel erweiterter Euklidischer Algorithmus 2

z (=Zeile)	a	b	qu (=Quotient)	r (=Rest)
1	48	5	9	3
2	5	3	1	2
3	3	2	1	1
4	2	1	2	0

Nach dem Ausfüllen der ganzen Tabelle bis zu $r = 0$, werden zwei neue Spalten x und y hinzugefügt. Diese werden von unten her mit den Startwerten $x = 0$ und $y = 1$ ausgefüllt. Die nun ergänzte Darstellung der Tabelle 2 präsentiert sich wie folgt (vgl. Tabelle 3):

Tabelle 3: Beispiel erweiterter Euklidischer Algorithmus 3

z (=Zeile)	a	b	qu (=Quotient)	r (=Rest)	x	y
1	48	5	9	3		
2	5	3	1	2		
3	3	2	1	1		
4	2	1	2	0	0	1

Die Spalten x und y werden nun mit der Ausfüll-Vorschrift

$$x_z = y_{z+1} \qquad y_z = x_{z+1} - q_z \cdot y_{z+1}$$

Zeile für Zeile ausgefüllt.

Die komplett ausgefüllte Beispiel-Tabelle sieht aus wie folgt (vgl. Tabelle 4):

Tabelle 4: Beispiel erweiterter Euklidischer Algorithmus 4

z (=Zeile)	a	b	qu (=Quotient)	r (=Rest)	x	y
1	48	5	9	3	2	−19
2	5	3	1	2	−1	2
3	3	2	1	1	1	−1
4	2	1	2	0	0	1

In jeder Zeile gilt nun:

$$x_z \cdot a_z + y_z \cdot b_z = ggT(a_1, b_1)$$

Die aus den Zahlen der Zeile 1 entstehende Gleichung ist die gesuchte Vielfachsummendarstellung des ggT von a_1 und b_1:

$$48 \cdot 2 + 5 \cdot (-19) = ggT(48, 5) = 1$$

In meinem Code (vgl. Abb 13) habe ich anstelle einer Tabelle eine Liste mit verschachtelten Klammern erstellt.

```
a= phi
b= e
qu =(phi//e)
r = phi
list = []
while (r!=0):
    while (r>=b):
        r=r-(b*qu)
    list.append([a,b,qu,r])
    if (r>0):
        a = b
        b = r
        qu = (a//b)
        r= a
```

Abbildung 7: Erster Teil des erweiterten Euklidischen Algorithmus

Jede Zeile ist dabei in einer eigenen Klammer gespeichert.

$$list = [[a_1, b_1, qu_1, r_1], [a_2, b_2, qu_2, r_2], ...]$$

Die while-Schleife wird wiederholt, solange der Rest r nicht gleich 0 ist. Sobald dieser Wert erreicht wird, berechnet der Algorithmus (vgl. Abb. 14) das jeweilige x und y jeder Zeile nach der oben erwähnten Ausfüll-Vorschrift.

```
x = 0
y = 1
elemente = len(list)
while (elemente>0):
    list.insert(elemente,[x,y])
    elemente =elemente-1
    if (elemente>0):
        xneu= y
        y= x-((list[elemente-1][2])*y)
        x = xneu
d = list[1][1]
```

Abbildung 8: Zweiter Teil des erweiterten Euklidischen Algorithmus

Der gesuchte private Schlüssel d ist das y der Zeile 1 (vgl. Tabelle 4):

$$e \cdot d + \phi(n) \cdot k = 1 \quad \leftrightarrow \quad b_1 \cdot y_1 + a_1 \cdot x_1 = 1 \quad \leftrightarrow \quad 5 \cdot (-19) + 48 \cdot 2 = 1$$

Hier also -19.

1.2 Verschlüsselungsverfahren

Sowohl der öffentliche als auch der private Teil des Schlüssels wurden berechnet. Doch wie kann eine Nachricht damit nun verschlüsselt werden?

Ein Klartext $k \in \mathbb{N}$ wird mit dem öffentlichen Schlüssel E_T eines Teilnehmers, bestehend aus n und e, chiffriert. So entsteht ein Geheimtext g. [Albrecht Beutelspacher, 2010, S. 116]

$$k^e \equiv g(mod n)$$

In meinem Code wird nach der Eingabe der drei Argumente k,e und n die Funktion $verschluesselung()$ ausgeführt. Als Ausgabe druckt diese den chiffrierten Geheimtext g.

Binäre Exponentiation Ein effizienter Weg des modularen Potenzierens ist die *binäre Exponentiation*. Man schreibt dabei den Exponenten in seiner Binärdarstellung, rechnet die Basen mit den einzelnen Exponenten separat modular und multipliziert diese zum Schluss. [Albrecht Beutelspacher, 2010, S.120] Hier folgt ein Beispiel in drei Schritten, wo die Gleichung $5^{23}\, mod 7$ möglichst von Hand gelöst wird.

1. Exponent in Binärdarstellung schreiben:

$$[23]_{10} = [10111]_2 = [2^4 + 2^2 + 2^1 + 2^0]_{10} = [16 + 4 + 2 + 1]_{10}$$

2. Basen mit schrittweise höheren Exponenten separat modulo rechnen.
Beachte dabei $a^b \equiv c(mod\, d) \quad \leftrightarrow \quad (a^b)^2 = a^{2b} \equiv c^2(mod\, d)$:

$$\begin{aligned} 5^1 &\equiv 5(mod 7) \\ 5^2 = 25 &\equiv 4(mod 7) \\ 5^4 = (5^2)^2 \equiv 4^2(mod 7) &\equiv 2(mod 7) \\ 5^8 = (5^4)^2 \equiv 2^2(mod 7) &\equiv 4(mod 7) \\ 5^16 = (5^8)^2 \equiv 4^2(mod 7) &\equiv 2(mod 7) \end{aligned}$$

3. Die benötigten Reste zusammen multiplizieren:
$5^{23} = 5^{16} \cdot 5^4 \cdot 5^2 \cdot 5 \equiv 2 \cdot 2 \cdot 4 \cdot 5(mod 7) \equiv 80(mod 7) \equiv 3(mod 7)$

In meinem Code habe ich die binäre Exponentiation in die oben aufgeführten drei Schritte unterteilt angewendet. Die Basis ist dabei k, der Exponent e und das Modul n. Im ersten Schritt wird e in seine Binärdarstellung zerlegt (vgl. Abb. 15).

```
binaer=[]
potenzen=1
wurzel = e**0.5

while(wurzel>potenzen):
    potenzen=potenzen+1
hoechste_potenz=potenzen-1

while(e>0):
    if((e-(2**hoechste_potenz))>=0):
        e=e-(2**hoechste_potenz)
        hoechste_potenz=hoechste_potenz-1
        binaer.append(1)
    else:
        hoechste_potenz=hoechste_potenz-1
        binaer.append(0)
```

Abbildung 9: e in Binärdarstellung zerlegen

Die ganze Zahl e muss durch eine Summe von Zweierpotenzen ausgedrückt werden:

$$e = x_1 \cdot 2^0 + x_2 \cdot 2^1 + x_3 \cdot 2^2 + x_4 \cdot 2^3 + ... \qquad , wobei\ x_k = 1\ oder\ x_k = 0$$

In der ersten while-Schleife (vgl. Abb. 15) wird die höchste Zweierpotenz gesucht, die Summand der Summe e sein kann. Danach wird in der zweiten while-Schleife überprüft, ob $2^{hoechste\_Potenz}$ ein Summand der Summe e ist. Wenn dies eintritt, wird eine 1 zur Liste *binaer* hinzugefügt, andernfalls eine 0. Die Zahl *hoechste_Potenz* wird nach jedem Prüfdurchgang um 1 kleiner, bis die Zahl e 0 ist und nun komplett als Summe von Zweierpotenzen ausgedrückt werden kann. Die Liste *binaer* enthält nun eine Abfolge von 1 und 0, welche der Binärdarstellung von e entspricht.

In einem zweiten Schritt werden in einer Liste *reste* alle Modulo-Reste gespeichert (vgl. Abb. 16).

```
zaehler=0
rest= k%n
reste=[rest]
while(zaehler<(potenzen-1)):
    rest= (rest**2)%n
    reste.insert(0,rest)
    zaehler=zaehler+1
```

Abbildung 10: Liste mit Modulo-Resten

Als Start wird mit $rest = k\%n$ der *rest* der Gleichung $k \equiv rest(modn)$ berechnet und zur Liste hinzugefügt. In einer while-Schleife (vgl. Abb. 16) wird der Modulo-Rest der jeweilig vorherigen Rechnung quadriert, wiederum modulo n gerechnet und am Anfang der Liste ergänzt. Dieser Prozess läuft so lange, bis der *zaehler* und somit der Exponent die Grösse der im vorherigen Schritt berechneten höchst möglichen Potenz hat.

Im dritten und letzten Schritt müssen die benötigten Modulo-Reste multipliziert werden (vgl. Abb. 17).

```
g=1
zaehler=0
while(zaehler<=(potenzen-1)):
    if(binaer[zaehler]==1):
        g=g*(reste[zaehler])
        zaehler=zaehler+1
    else:
        zaehler=zaehler+1
g=g%n
```

Abbildung 11: Modulo-Reste multiplizieren, um g zu erhalten

Dazu benötigt man die zwei Listen *binaer* und *reste* (vgl. Abb. 17). Die beiden Listen werden parallel vom ersten Element an überprüft. Steht in der Liste *binaer* an erster Stelle eine 1, dann ist die erste Zahl der Liste *reste* ein Faktor des gesuchten Produkts g. Steht in *binaer* aber eine 0, ist die Zahl an der gleichen Stelle in *reste* kein Faktor von g. Beim vorher verwendeten Beispiel $5^{23}\, mod\, 7$ mit $e = 23$ und $n = 7$ würde die

Darstellung der Listen wie folgt aussehen:

$$\begin{aligned} binaer &= [1,0,1,1,1] \\ reste &= [2,4,2,4,5] \\ g &= 5 \cdot 2 \cdot 2 \cdot 4 \cdot 5 = 80 \end{aligned}$$

Zum Schluss wird g nochmals modulo n gerechnet und man erhält folgende endgültig verschlüsselte Nachricht.

$$g = g\%n \qquad \leftrightarrow \qquad 80 = 3(modn)$$

1.3 Entschlüsselungsverfahren

Einem Teilnehmer wurde eine mit seinem öffentlichen Schlüssel chiffrierte Nachricht gesendet. Wie kann diese von ihm wieder dechiffriert werden?

In meinem Code erledigt dies die Funktion $entschlschluesselung()$. Man gibt die Argumente g, d und n ein und die vorgängig verschlüsselte Nachricht g wird als Klartext k gedruckt.

Das Verfahren ist beinahe identisch mit jenem der Verschlüsselung. Lediglich ist hier die Basis die geheime Nachricht g und der Exponent der private Schlüssel d. Das Modul bleibt weiterhin n. [Albrecht Beutelspacher, 2010, S. 116]

$$g^d \equiv k(modn)$$

Auch bei der Dechiffrierung wird die binäre Exponentiation mit den drei gleichen Schritten, wie bereits bei der Chiffrierung, verwendet.

1.3.1 Kernidee der RSA-Verschlüsselung

Doch warum funktioniert RSA überhaupt? Kommt man nach dem Durchlauf eines solchen Verfahrens wirklich jedes Mal wieder auf die ursprüngliche Nachricht k zurück?

Aus dem *Satz von Euler* und dem *kleinen Satz von Fermat* kann man folgern: [ebd., S. 122 ff.]

Für die Zahlen a, $k \in \mathbb{Z}$ und $n = p \cdot q$, wobei p und $q \in \mathbb{P}$ gilt:

$$\boxed{a^{k \cdot \phi(n)+1} \equiv a(modn)} \tag{4}$$

Die bei der Erzeugung des privaten Schlüssels d aufgestellte Gleichung (3), kann man auf $e \cdot d$ auflösen:

$$e \cdot d + k \cdot \phi(n) = 1 \quad \leftrightarrow \quad e \cdot d = -k \cdot \phi(n) + 1$$

Den Exponenten $k \cdot \phi(n) + 1$ der Gleichung (4) kann man durch $e \cdot d$ ersetzen. Da $k \in \mathbb{Z}$, spielt es keine Rolle, ob vor k ein Minus steht oder nicht:

$$a^{e \cdot d} \equiv a(modn) \tag{5}$$

Aus den zwei zentralen Gleichungen der Ver- und Entschlüsslung bei RSA

$$k^e \equiv g(modn)$$
$$g^d \equiv k(modn)$$

lässt sich dank Ersetzen von g durch k^e folgern:

$$g^d(modn) \equiv k^{e \cdot d}(modn) \tag{6}$$

Die Kombination aus Gleichung (5) und (6) beweist also:

$$\boxed{g^d \equiv k(modn)}$$

Nach anwenden des privaten Schlüssels d auf die verschlüsselte Nachricht g, muss somit zwingend wieder der Klartext k entstehen.

Abbildungsverzeichnis

Abbildungen ohne Quellenangaben wurden von der Autorin selbst erstellt.

Tabellenverzeichnis

Sämtliche Tabellen wurden vom Autor selbst erstellt.

Anhang

2 RSA-Algorithmus programmiert mit Python

Die Python-Dateien sind zusätzlich auf dem beigelegten USB-Stick verfügbar.

2.1 Schlüsselerzeugung

```
def schluesselerzeugung():


    faktor=2.0
    primzahlen=[]
    testzahl=2
    while (testzahl<100000):
        if (testzahl/faktor)==int(testzahl/faktor) and faktor<testzahl:
            testzahl=testzahl+1
            faktor=2.0
        else:
            if(faktor>=testzahl**0.5):
                primzahlen.append(testzahl)
                testzahl=testzahl+1
                faktor=2.0
            else:
                faktor=faktor+1

    import random
    p= random.choice(primzahlen)
    q= random.choice(primzahlen)

    while (abs(p-((q*p)**0.5))<300) or (abs(q-((q*p)**0.5))<300) or (p==q):
        q=random.choice(primzahlen)
        p=random.choice(primzahlen)

    n = p*q


    phi = (p-1)*(q-1)

    e=2
    faktor=2.0
    while(faktor<=e):
        if(phi/faktor)==int(phi/faktor) and (e/faktor)==int(e/faktor):
            e=e+1
            faktor=2.0
        else:
            faktor=faktor+1
```

Abbildung 12: Python-Code Schlüsselerzeugung (Fortsetzung siehe Abb. 19)

```
    a= phi
    b= e
    qu =(phi//e)
    r = phi
    list = []
    while (r!=0):
        while (r>=b):
            r=r-(b*qu)
        list.append([a,b,qu,r])
        if (r>0):
            a = b
            b = r
            qu = (a//b)
            r= a


    x = 0
    y = 1
    elemente = len(list)
    while (elemente>0):
        list.insert(elemente,[x,y])
        elemente =elemente-1
        if (elemente>0):
            xneu= y
            y= x-((list[elemente-1][2])*y)
            x = xneu
    d = list[1][1]

    while d<0:
        d=d+phi

    e=str(e)
    n=str(n)
    d=str(d)

    print("Oeffentlicher Schluessel: e="+e+" , n="+n)
    print("Privater Schluessel: d="+d)

    return()
schluesselerzeugung()
```

Abbildung 13: Python-Code Schlüsselerzeugung (Fortsetzung)

2.2 Verschlüsselung

```
def verschluesselung(k,e,n):
    k= int(k)
    e=int(e)
    n=int(n)
    ebeginn=e

    binaer=[]                              #Dezimalzahl in Binaerzahl umrechnen
    potenzen=1
    wurzel = e**0.5

    while(wurzel>potenzen):
        potenzen=potenzen+1
    hoechste_potenz=potenzen-1

    while(e>0):
        if((e-(2**hoechste_potenz))>=0):
            e=e-(2**hoechste_potenz)
            hoechste_potenz=hoechste_potenz-1
            binaer.append(1)
        else:
            hoechste_potenz=hoechste_potenz-1
            binaer.append(0)
```

Abbildung 14: Python-Code Verschlüsselung (Fortsetzung siehe Abb. 21)

```
    zaehler=0                                           #Liste der Modulo-Reste
    rest= k%n
    reste=[rest]
    while(zaehler<(potenzen-1)):
        rest= (rest**2)%n
        reste.insert(0,rest)
        zaehler=zaehler+1

    g=1
    zaehler=0
    while(zaehler<=(potenzen-1)):
        if(binaer[zaehler]==1):
            g=g*(reste[zaehler])
            zaehler=zaehler+1
        else:
            zaehler=zaehler+1
    g=g%n
    g=str(g)
    print("verschluesselte Nachricht: g="+g)
verschluesselung(input("Gib deine geheime Nachricht ein.")
,input("Wie lautet der oeffentliche Schluessel e des Empfaengers?")
,input("Wie lautet der oeffentliche Schluessel n des Empfaengers?"))
```

Abbildung 15: Python-Code Verschlüsselung (Fortsetzung)

2.3 Entschlüsselung

```
def entschluesselung(g,d,n):
    g=int(g)
    d=int(d)
    n=int(n)

    binaer=[]                          #Dezimalzahl in Binaerzahl umrechnen
    potenzen=1
    wurzel = d**0.5

    while(wurzel>potenzen):
        potenzen=potenzen+1
    hoechste_potenz=potenzen-1

    while(d>0):
        if((d-(2**hoechste_potenz))>=0):
            d=d-(2**hoechste_potenz)
            hoechste_potenz=hoechste_potenz-1
            binaer.append(1)
        else:
            hoechste_potenz=hoechste_potenz-1
            binaer.append(0)
```

Abbildung 16: Python-Code Entschlüsselung (Fortsetzung siehe Abb. 23)

```
    zaehler=0                                   #Liste der Modulo-Reste
    rest= g%n
    reste=[rest]
    while(zaehler<(potenzen-1)):
        rest= (rest**2)%n
        reste.insert(0,rest)
        zaehler=zaehler+1

    k=1
    zaehler=0
    while(zaehler<=(potenzen-1)):
        if(binaer[zaehler]==1):
            k=k*(reste[zaehler])
            zaehler=zaehler+1
        else:
            zaehler=zaehler+1
    k=k%n

    return(k)

print(entschluesselung(input("Gib die verschluesselte Nachricht ein.")
 ,input("Wie lautet dein privater Schluessel?")
 ,input("Wie lautet der Teil n deines oeffentlichen Schluessels?")))
```

Abbildung 17: Python-Code Entschlüsselung (Fortsetzung)