Die Überlegenheit von Quantenalgorithmen: Lektion 5 – RSA: Der unknackbare Code?

In dieser Lektion beschäftigen wir uns mit dem RSA-Verschlüsselungsalgorithmus und klassischen Methoden, ihn zu brechen. Der RSA-Algorithmus wurde 1979 von Rivest, Shamir und Adleman entwickelt und ist bis heute eine der erfolgreichsten Formen der Verschlüsselung. Wenn du jemals eine Finanztransaktion im Internet durchgeführt hast, ist es sehr wahrscheinlich, dass die dabei verwendeten Sicherheitscodes mithilfe des RSA-Systems verschlüsselt und deine Identität darüber verifiziert wurde.

Diese Lektion untersucht die Methodik des RSA-Verfahrens und zeigt, warum es in der klassischen Informatik praktisch unknackbar ist.

Fermats kleiner Satz

Im Zentrum des RSA-Verfahrens steht ein schönes mathematisches Resultat, bekannt als Fermats kleiner Satz:

$$\boldsymbol{x^{p-1} \equiv 1\ (mod\ p)}$$

Das bedeutet: Wenn man eine natürliche Zahl **x** mit einer um eins verminderten Primzahl p (also mit p−1) potenziert, dann ergibt der Rest bei Division durch **p** immer 1. Die **mod**-Funktion („modulo“) gibt den Rest nach der Division an.

Zum Beispiel nehmen wir x = 3 und p = 5:

$$3^{p-1} = 3^4 = 81$$

$$81\ /\ p = 81\ /\ 5 = 16\ Rest\ 1$$

Aufgabe 1

Überprüfe Fermats kleinen Satz für folgende Werte von x und p:

(a) x = 2, p = 3; (b) x = 2, p = 5; (c) x = 4, p = 5; (d) x = 4, p = 7; (e) x = 5, p = 11.

Hinweis zur Implementierung

Das letzte Beispiel in Übung 1 zeigt ein potenzielles Problem bei der Implementierung von RSA: Die Potenzfunktion wächst extrem schnell. Selbst mit kleinen Primzahlen kann es schnell zu Überlauf-Fehlern kommen, wenn man die Potenz direkt berechnet. Das lässt sich vermeiden, indem man ausnutzt, dass eine Potenz denselben Rest besitzt, egal ob man als Basis die Zahl oder das Ergebnis verwendet. Man kann also Zwischenergebnisse bei jeder Multiplikation modulo p reduzieren.

Nehmen wir zum Beispiel wieder x = 3 und p = 5, sehen wir:

$$3^4 = 9^2$$

Aber:

$$9 \equiv 4\ (mod\ 5)$$

ENTNOMMEN AUS QUANTEN-COMPUTING IM MINT-UNTERRICHT
Ein Projekt von Science on Stage Deutschland e.V.

Statt 9^2 mit der Modulo-Operation zu berechnen, können wir 4^2 verwenden und finden wie gewünscht $4^2 = 16 \equiv 1\ (mod\ 5)$.

Daher können wir zur Berechnung der modularen Potenz in Python eine Schleife schreiben, die x mehrfach mit sich selbst multipliziert und dabei jedes Mal den Rest modulo p berechnet.

Aufgabe 2

Implementiere den folgenden Python Code (**%** ist in Python die Notation für die Modulo-Operation):

```
def powerMod(a, b, c): # raises a to the power of b modulo c
    res = a % c
    for i in range(0,b-1):
        res = a * res % c
    return res

x=5
p=11
print(powerMod(x,p-1,p))
```

Nutze den Code, um herauszufinden, welche der folgenden Zahlen keine Primzahlen sind (weil Fermats kleiner Satz für sie nicht gilt):

(a) 1367; (b) 1369; (c) 7527; (d) 7531; (e) 7537; (f) 15723; (g) 15727; (h) 18791.

Der RSA-Algorithmus

RSA beruht auf einer Variante von Fermats kleinem Satz:

Wenn **p** und **q** Primzahlen sind, gilt: $\boldsymbol{x^{(p-1)(q-1)} \equiv 1\ (mod\ pq)}$.

Als Beispiel nehmen wir x = 2, p = 3 und q = 5:

$2^{(p-1)(q-1)} = 2^{2 \cdot 4} = 2^8 = 256$

$256\ /\ (pq) = 256\ /\ 15 = 17\ Rest\ 1$

Multipliziert man beide Seiten der Gleichung mit **x**, erhält man:

$\boldsymbol{x^{(p-1)(q-1)} . x \equiv x\ (mod\ pq)}$

$\boldsymbol{x^{(p-1)(q-1)+1} \equiv x\ (mod\ pq)}$

Dies bedeutet, dass man x (mod pq) erhält, wenn man x mit (p−1)(q−1) + 1 potenziert. Der RSA Algorithmus nutzt diese Tatsache für eine Ver- und Entschlüsselung. Wenn wir zwei Zahlen **e** und **d** finden mit:

$\boldsymbol{ed \equiv 1\ (mod\ (p-1)(q-1))}$, dann gilt $\boldsymbol{x^{ed} \equiv x\ (mod\ pq)}$

Ist x die zu verschlüsselnde Nachricht, definieren wir **e** als Verschlüsselungsschlüssel und **d** als Entschlüsselungsschlüssel. Die beiden Schlüssel zum Ver- und Entschlüsseln sind also unterschiedlich (dies nennt man auch asymmetrische Verschlüsselung).

Schritt 1 – Verschlüsselung

Wir potenzieren die Nachricht **x** mit **e**, um die verschlüsselte Nachricht **y** zu erhalten:

$$x^e \equiv y \ (mod\ pq)$$

Die Sicherheit der RSA-Verfahrens beruht darauf, dass die Kenntnis von **y**, **e** und **pq** nicht ausreicht, um **x** zu berechnen. Es gibt keinen eindeutigen Weg, um die **e**-te Wurzel in modularer Arithmetik zu berechnen. Dies kann man zum Beispiel durch Fermats kleinen Satz selbst sehen:

$$1^4 \equiv 2^4 \equiv 3^4 \equiv 4^4 \equiv 1 \ (mod\ 5)$$

Falls wir also die vierte Wurzel von 1 (mod 5) bestimmen wollen, können wir nicht wissen, ob die ursprüngliche Zahl 1, 2, 3 oder 4 ist.

Schritt 2 – Entschlüsselung

Um die verschlüsselte Nachricht **y** wieder zu entschlüsseln, wird diese Nachricht mit **d (mod pq)** potenziert. Mithilfe der Potenzgesetze erkennt man:

$$y^d \equiv (x^e)^d \equiv x^{ed} \equiv x \ (mod\ pq)$$

Der RSA-Algorithmus: Zusammenfassung des Prozesses

Sender	Öffentlichkeit	Empfänger
	Was sie wissen	
x (zu verschlüsselnde Nachricht) **e** (Verschlüsselungsschlüssel) **pq** (zu verwendendes Modul) **p** oder **q** nicht einzeln bekannt	**y** (verschlüsselte Nachricht) **e** (Verschlüsselungsschlüssel) **pq** (zu verwendendes Modul) **p** oder **q** nicht einzeln bekannt	**y** (verschlüsselte Nachricht) **d** (Entschlüsselungsschlüssel) **pq** (zu verwendendes Modul) **p** und **q** sind einzeln bekannt, daher kann **d** errechnet werden
	Was sie tun	
Berechnen der verschlüsselten Nachricht **y = x^e (mod pq)**	Das Wissen über **e** und **pq** reicht nicht aus, um **y** zu entschlüsseln.	Berechnen der entschlüsselten Nachricht **x = y^d (mod pq)**

Beispiel

Wir verwenden p = 89, q = 151 und pq = 13439. Falls wir e = 7543 setzen, ist der zugehörige Entschlüsselungsschlüssel d = 7.

Wir verschlüsseln mit dem folgenden Python-Code die Beispielnachricht 8571:

```
def powerMod(a, b, c): # raises a to the power of b modulo c
    res = a % c
    for i in range(0,b-1):
        res = a * res % c
    return res

plainText=8571
pq=13439
e=7543

encryptedMessage=powerMod(plainText,e,pq)
print(encryptedMessage)
```

Als verschlüsselte Nachricht sollte sich 1134 ergeben.

Im zweiten Schritt entschlüsseln wir diese Nachricht wieder mit dem Entschlüsselungsschlüssel d=7:

```
def powerMod(a, b, c): # raises a to the power of b modulo c
    res = a % c
    for i in range(0,b-1):
        res = a * res % c
    return res

cipherText=1134
pq=13439
d=7

decryptedMessage=powerMod(cipherText,d,pq)
print(decryptedMessage)
```

Es ergibt sich wieder die ursprüngliche Nachricht 8571.

Aufgabe 3a

Verwende pq=13439 und e = 5077, um die Nachricht QUANTUM zu verschlüsseln. Wandle jeden Buchstaben in eine Zahl um (A = 1, B = 2, …) und verschlüssele jeden Buchstaben einzeln.

Aufgabe 3b

Entschlüssele die Nachricht wieder mit pq=13439 und d = 13.

Der RSA-Algorithmus: Zusammenfassung

Das RSA-Verfahren besitzt zwei Schlüssel: einen öffentlichen Schlüssel und einen privaten (geheimen) Schlüssel. Der öffentliche Schlüssel besteht aus **e** und **pq**; der private Schlüssel besteht aus **d** und den einzelnen Zahlen **p** und **q**.

Der öffentliche Schlüssel kann ohne Einschränkungen herausgegeben werden, da dieser nur zum Verschlüsseln, aber nicht zum Entschlüsseln verwendet werden kann. Möchte eine Bank zum Beispiel eine geheim zuhaltende Transaktion mit einem Kunden (z. B. eine PIN oder eine Kartenprüfnummer einer Kreditkarte) durchführen, kann die Bank den öffentlichen Schlüssel veröffentlichen.

Der Kunde kann seine Information über die Operation $x^e \equiv y \ (mod \ pq)$ verschlüsseln und **y** an die Bank übermitteln. Selbst wenn ein Hacker die Nachricht abfängt und ebenfalls im Besitz des öffentlichen Schlüssels der Bank ist, kann dieser nicht die ursprüngliche Nachricht **x** berechnen. Nur die Bank selber kann mit dem privaten Schlüssel **d** die Nachricht entschlüsseln, um die geheime Information x zu lesen.

Das Brechen von RSA auf einem klassischen Computer

Um RSA zu brechen, muss man **d** kennen. Dazu benötigt man jedoch **(p−1)(q−1)**. Obwohl **pq** öffentlich ist, ist das Produkt **(p−1)(q−1)** nicht einfach zu bestimmen. Dazu müsste man **pq** in seine Primfaktoren zerlegen. Das ist für große Primzahlen **p** und **q** extrem aufwendig. Die folgende Aufgabe macht diese Herausforderung deutlich:

Aufgabe 4

Schreibe ein Programm, um die Zahl 900 239 055 271 631 zu faktorisieren. Stoppe die Zeit, die dein Programm benötigt, um die Faktoren **p** und **q** zu finden.

Zusammenfassung

Faktorisieren ist ein Beispiel für ein nur schwierig lösbares Problem: Es gibt eine Methode zum Finden der beiden Faktoren (Trial and Error Division), aber die benötigte Zeit wächst exponentiell mit der Größe der Zahl. Wählt man die beiden Primzahlen groß genug, könnten selbst Supercomputer das Produkt dieser Zahlen nicht in realistischer Zeit faktorisieren. In unserem Beispiel waren die beiden Primzahlen 8-stellig. In heutigen Implementationen von RSA werden typischerweise Primzahlen mit etwa je 300 Stellen verwendet. Damit ist derzeit RSA der Goldstandard der Internetsicherheit für abhörsichere Kommunikation.

Doch Quantencomputing könnte dieses Prinzip bedrohen: Mit **Shors Algorithmus** ließe sich die Komplexität der Faktorisierung reduzieren – so weit, dass RSA in Echtzeit gebrochen werden könnte. In der nächsten Lektion wirst du lernen, wie der Shor-Algorithmus die Sicherheit von RSA bedrohen kann und warum dies eine Revolution in der Kryptographie auslösen könnte.