

RSA in Haskell

Sven Eric Panitz*

8. März 2021

Inhaltsverzeichnis

1	RSA Verschlüsselung	2
1.1	Öffentlichen und privaten Schlüssels	3
1.2	Algorithmus zur Schlüsselpaargenerierung	4
2	Implementierung in Haskell	4
2.1	Generierung von Schlüsselpaar für zwei Primzahlen	5
2.2	Erweiterte Euklidischer Algorithmus	6
2.3	Verschlüsseln und Entschlüsseln mit Schlüsselpaar	7
2.4	Generierung großer Primzahlen	8
2.5	Texte Verschlüsseln	10
3	Lernzuwachs	12

1 RSA Verschlüsselung

In diesem Übungsblatt implementieren wir das RSA-Kryptosystem (Verschlüsselung und Entschlüsselung) in Haskell. Das RSA-Verschlüsselungsverfahren wurde 1977 von den Mathematikern Rivest, Shamir und Adleman entwickelt[RSA78]. Der Name leitet sich aus den Anfangsbuchstaben der drei Autoren ab.

Ziel einer Verschlüsselung ist es, eine Nachricht, die wir nur als natürliche Zahl darstellen werden, durch eine Verschlüsselungsfunktion bijektiv auf eine andere Zahl abzubilden, um dann nicht mehr die Originalnachricht zu versenden, sondern das Bild der Nachricht. Der Empfänger wendet dann die Inverse der Funktion an, um die verschlüsselte Nachricht wieder zu entschlüsseln.

Wenn wir als Nachricht die Zahl 42 versenden wollen, können wir sie zum Beispiel mit folgender Funktion verschlüsseln:

*Dank für seine wertvollen Hinweise gebührt Steffen Reith.

$$f :: \mathbb{N} \rightarrow \mathbb{N}$$

$$f(n) = n^2 + 1$$

Das Ergebnis ist für 42 die verschlüsselte Nachricht: 1756.

Diese Verschlüsselung ist wenig sicher. Jeder kann durch Kenntnis der Verschlüsselungsfunktion die inverse Funktion anwenden. Damit die Nachricht wieder entschlüsselt werden:

$$f^{-1}(n) = \sqrt{n - 1}$$

Die Funktion f ist also keine sichere Verschlüsselungsfunktion.

Wie sieht es mit folgender Funktion aus:

$$g(n) = n^{17} \bmod 247$$

Wenn wir mit dieser Funktion zum Beispiel die Nachricht 50 verschlüsseln, dann ist das Ergebnis $50^{17} \bmod 247 = 120$.

Hier ist die Umkehrfunktion, die die verschlüsselte Nachricht wieder entschlüsselt, nicht mehr so einfach zu finden.

Es ist ein asymmetrisches Verfahren und basiert auf einer Einwegfunktionen. Damit ist eine Funktion gemeint, die »leicht« (in Polynomialzeit) zu berechnen, deren Umkehrfunktion hingegen schwierig zu berechnen ist. Die Idee ist also, dass das Verschlüsseln einfach zu berechnen ist, das Entschlüsseln jedoch schwer.

Damit allein wäre einem Verschlüsselungsverfahren nicht geholfen, denn der Empfänger der Nachricht muss diese ja auch leicht entschlüsseln können. So braucht es zusätzlich für die Verschlüsselungsfunktion eine Möglichkeit, dass mit einer (geheimen) Zusatzinformation auch die Umkehrung leicht zu berechnen ist. Man spricht von einer Falltürfunktion. Genau dieses bietet die RSA-Verschlüsselung.

Diese Falltürfunktion basiert auf der Zerlegung einer Zahl in Primfaktoren. Eine Zahl aus den Primfaktoren durch Multiplikation zu errechnen ist einfach, hingegen ist für die Zerlegung großer Zahlen in Primfaktoren kein effizientes Verfahren bekannt.¹ Dass kein solches Verfahren existiert, ist nicht bewiesen. Es könnte sein, dass es ein einfaches Verfahren gibt, bisher aber alle Informatiker zu dumm waren, dieses zu finden.

1.1 Öffentlichen und privaten Schlüssels

Das RSA Verfahren benutzt als Schlüssel drei Zahlen:

- einer Zahl N , die als RSA-Modul bezeichnet wird.

¹Sofern man keinen Quantencomputer zur Verfügung hat.

- einer Zahl e (*encrypt*), die als Verschlüsselungsexponent bezeichnet wird.
- einer Zahl d (*decrypt*), die als Entschlüsselungsexponent bezeichnet wird.

Das Zahlenpaar (e, N) ist der öffentliche Schlüssel.

Das Zahlenpaar (d, N) ist der private Schlüssel.

Eine Nachricht m wird verschlüsselt durch die Funktion:

$$f(m) = m^e \bmod N.$$

Eine verschlüsselte Nachricht m wird entschlüsselt durch die Funktion:

$$f(m) = m^d \bmod N.$$

1.2 Algorithmus zur Schlüsselpaargenerierung

Wir zitieren hier direkt den Algorithmus zu Erzeugung eines Schlüsselpaares aus dem entsprechenden Wikipediaartikel[RSA19]. Eine gute Einführung finden Sie in [Buc04].

1. Wähle zufällig und stochastisch unabhängig zwei Primzahlen $p \neq q$. Diese sollen die gleiche Größenordnung haben, aber nicht zu dicht beieinander liegen, sodass der folgende Rahmen ungefähr eingehalten wird: $0,1 < |\log_2 p - \log_2 q| < 30$ (In der Praxis erzeugt man dazu solange Zahlen der gewünschten Länge und führt mit diesen anschließend einen Primzahltest durch, bis man zwei Primzahlen gefunden hat.)
2. Berechne den RSA-Modul $N = p \cdot q$
3. Berechne die Eulersche φ -Funktion von N

$$\varphi(N) = (p - 1) \cdot (q - 1)$$
4. Wähle eine zu $\varphi(N)$ teilerfremde Zahl e , für die gilt $1 < e < \varphi(N)$.
5. Berechne den Entschlüsselungsexponenten d als multiplikativ Inverses von e bezüglich des Moduls $\varphi(N)$. Es soll also die folgende Kongruenz gelten:

$$e \cdot d \equiv 1 \pmod{\varphi(N)}$$

So weit der Algorithmus aus dem Wikipediaartikel.

Alles beginnt also mit zwei Primzahlen. Ihr Produkt sind der RSA-Modul. Der RSA-Modul wird als Teil des öffentlichen Schlüssels öffentlich gemacht. Wenn es jemanden gelingt, den Modul in seine zwei Primfaktoren zu zerlegen, dann kann man einfach mit obigen Algorithmus auch den privaten Schlüssel berechnen. Wir vertrauen also darauf, dass es nach wie vor schwer sein wird, große Zahlen in ihre Primfaktoren zu zerlegen.

2 Implementierung in Haskell

Wir entwickeln den Algorithmus in Haskell. Da wir im Endeffekt keine Zahlen sondern Texte verschlüsseln wollen, importieren wir auch das Modul `Data.Char`. Für den Umgang

mit Nullwerten greifen wir auf Funktionen des Moduls `Data.Maybe` zurück. Zur Generierung von zufälligen Zahlen verwenden wir das Modul `System.Random`. Zum Test, ob eine Zahl eine Primzahl ist, verwenden wir das Modul `Math.NumberTheory.Primes`.

Haskell: RSA

```
> module RSA where

> import Data.Char
> import Data.List
> import Data.Maybe

> import Math.NumberTheory.Primes
> import System.Random
```

Bevor wir das RSA-Verfahren implementieren, überlegen Sie, wie Sie die Verschlüsselung unseres illustrierenden Eingangsbeispiels knacken können.

Aufgabe 1

Wir haben als erstes Beispiel Zahlen verschlüsselt mit der Formel:

$$f(n) = n^{17} \bmod 247.$$

Wir kennen keinen privaten Schlüssel, um diese Verschlüsselung rückgängig zu machen. Schreiben Sie trotzdem eine Funktion, die so verschlüsselte Nachrichten entschlüsseln kann.

Haskell: RSA

```
> decryptExample msg = 0
```

2.1 Generierung von Schlüsselpaar für zwei Primzahlen

Nach dem Algorithmus lassen sich für einen Verschlüsselungsexponenten und zwei Primzahlen der öffentliche und der private Schlüssel direkt berechnen.

Zunächst führen wir ein Typsynonyme für Schlüssel ein:

Haskell: RSA

```
> type Key = (Integer,Integer)
> type KeyPair = (Key,Key)
```

Wir definieren eine Funktion, die einen öffentlichen und privaten Schlüssel für zwei Primzahlen und einem Verschlüsselungsexponenten berechnen. Allerdings muss dabei noch

sicher gestellt sein, dass der Verschlüsselungsexponenten und φ teilerfremd sind.

Bei falschen Parameterwerten kann also gar kein Schlüsselpaar generiert werden. Deshalb sehen wir eine Funktion vor, die vom Ergebnistyp (`Maybe KeyPair`) ist:

Haskell: RSA

```
> publicPrivateFor :: Integer -> Integer -> Integer -> Maybe KeyPair
> publicPrivateFor e p q = do
>   s <- multiplicativInverse e ((p-1)*(q-1))
>   return (let n = p*q in ((e,n),(s ,n)))
```

Die eigentliche Hauptarbeit ist die Berechnung der multiplikativen Inverse. Es ist nicht gesichert, dass dieses existiert. (Die Wahrsvcheinlichkeit, dass es existiert, liegt bei $\frac{6}{\pi^2} \approx 0.6$). Der Aufruf `multiplicativInverse e ((p-1)*(q-1))` ist deshalb vom Typ: (`Maybe Integer`).

Die in `publicPrivateFor` verwendete `do`-Notation wird verwendet, um den Nullwert `Nothing` des Typs `Maybe` zu propagieren. Hat die Funktion `multiplicativInverse` den Wert `Nothing` als Ergebnis, dann automatisch auch die Funktion `publicPrivateFor`.

Typischer Weise wird ein besonderer Verschlüsselungsexponent gewählt.

Haskell: RSA

```
> publicPrivate = publicPrivateFor e
> e = (2^17+1)
```

Einzig die multiplikative Inverse von e bezüglich des Moduls $\varphi(N)$ ist noch zu bestimmen. Hierzu fehlt noch eine Umsetzung des Erweiterte Euklidischen Algorithmus.

2.2 Erweiterte Euklidischer Algorithmus

Der hinreichend bekannte Euklid Algorithmus berechnet den größten gemeinsamen Teiler zweier Zahlen. Der erweiterte Euklidische Algorithmus berechnet nicht nur den größten gemeinsamen Teiler sondern noch zwei ganze Zahlen s und t , so dass folgende Gleichung gilt:

$$\text{ggT}(a, b) = s \cdot a + t \cdot b$$

Wenn die beiden Zahlen a und b teilerfremd sind, dann ist der größte gemeinsame Teiler 1. Dann gilt also $1 = s \cdot a + t \cdot b$.

Es gilt: $(s \cdot a + t \cdot b) \bmod a = t \cdot b \bmod a = 1$.

Damit ist t das multiplikative Inverse, das nach dem RSA Algorithmus zu bestimmen ist.

Der erweiterte Euklidische Algorithmus ist rekursiv wie folgt formuliert:

$$eEuclid(a, b) = \begin{cases} (a, 1, 0) & \text{wenn } b = 0 \\ (d', t', s' - t' * (a \text{ div } b)) & \text{mit } (d', s', t') = eEuclid(b, a \bmod b) \end{cases}$$

Aufgabe 2

Implementieren Sie den erweiterten Euklidischen Algorithmus.

Haskell: RSA

```
> gcdExt :: Integer -> Integer -> (Integer, Integer, Integer)
> gcdExt a b = (1, 1, 1)
```

Damit lässt sich jetzt die zur Erzeugung der RSA Schlüssel multiplikative Inverse er-
rechnen.

Haskell: RSA

```
> multiplicativInverse a b = mkPos $ gcdExt a b
> where
>   mkPos (1, x, _)
>     | x < 0 = Just (x + b)
>     | otherwise = Just x
>   mkPos (_, _, _) = Nothing
```

Für erste Tests seien exemplarisch zwei etwas umfangreichere Primzahlen gegeben:

Haskell: RSA

```
> p = 0x19fbd41d69aa3d86009a967db3379c63cd501f24f7
> q = 0x1b6f141f98eeb619bc0360220160a5f75ea07cdf1d
```

2.3 Verschlüsseln und Entschlüsseln mit Schlüsselpaar

Zur Verschlüsselung und Entschlüsselung einer Nachricht wird diese mit dem Schlüssel-
exponenten potenziert und dann modulo zum RSA-Modul gerechnet. Die naiver Imple-
mentierung in Haskell mündet in folgender Definition:

Haskell: RSA

```
> pow x y z = x^y `mod` z
```

Diese Funktion kann zum Entschlüsseln und Verschlüsseln direkt verwendet werden.

Haskell: RSA

```
> encrypt (public ,n) msg = pow msg public n  
> decrypt = encrypt
```

Wie man sieht, sind die beiden Funktionen identisch.

Aufgabe 3

Die einfache Implementierung von `pow`, die wir gegeben haben, ist für große Zahlen viel zu ineffizient. Es ist erst eine sehr hohe Potenz zu rechnen und dieses Ergebnis anschließend durch eine Modularechnung wieder zu einer kleineren Zahl umzuwandeln.

Dieses lässt sich unter Berücksichtigung folgender Gleichungen effizienter implementieren.

- $n^{2*p} \bmod m = (n^p \bmod m)^2 \bmod m$
- $n^{2*p+1} \bmod m = (n^p \bmod m)^2 * n \bmod m$

Reimplementieren Sie eine effiziente Version von `pow` unter Verwendung der obigen Gleichungen.

Aufgabe 4

Testen Sie mit den beiden großen Beispielen einmal, ob die Verschlüsselung und Entschlüsselung großer Zahlen gelingt.

2.4 Generierung großer Primzahlen

Das `A` und `O` der RSA Verschlüsselung sind große Primzahlen. Der naive Primzahlengenerator mit dem Sieb des Eratosthenes ist bei weiten nicht effizient genug, um sehr große Primzahlen zu generieren:

Haskell: RSA

```
> primes = sieb [2..]  
> where  
> sieb (x:xs) = x:sieb [y|y<-xs,y `mod` x /= 0]
```

Das gängige Verfahren, zur Generierung großer Primzahlen ist, zufällig sehr große Zahlen zu generieren und mit einem effizienten Schnelltest zu checken, ob die generierte Zahl

eine Primzahl sein kann.

Das Haskell-Modul `Math.NumberTheory.Primes` aus dem Paket *arithmoi* [Fis11] enthält einen effizienten Check, zum Testen, ob eine Zahl eine Primzahl ist. Zusammen mit dem Standard-Modul `System.Random` lassen sich große Primzahlen generieren.

Zum Generieren einer Primzahl kann mit `randomRIO` so lange eine neue Zahl mit einer vorgegebenen Bitlänge generiert werden, bis der Primzahlentest positiv ist.

Haskell: RSA

```
> genPrime :: Int -> IO Integer
> genPrime bits = do
>   x <- randomRIO (2^(bits - 1), 2^bits - 1)
>   if isPrime x then return x else genPrime bits
```

Für das RSA-Verfahren werden zwei unterschiedliche Primzahlen benötigt. Die folgende Funktion generiert zunächst die erste Primzahl und dann so lange eine zweite, bis diese ungleich der ersten Zahl ist.

Haskell: RSA

```
> genPrimes :: Int -> IO (Integer, Integer)
> genPrimes bits = do
>   p1 <- genPrime bits
>   genPrim2 p1
>   where
>     genPrim2 p1 = do
>       p2 <- genPrime bits
>       if p1 /= p2 then return (p1, p2) else genPrim2 p1
```

Um einen Schlüssel zu generieren, braucht man noch den Verschlüsselungsexponenten, der zu den beiden Primzahlen passt. Wir lassen uns zwei Zahlen mit 1024 Bit Länge generieren und schauen, ob diese zum Standardexponenten e passen.

Haskell: RSA

```
> genKey = do
>   (p1,p2) <- genPrimes 1024
>   let ks = publicPrivate p1 p2
>   if isNothing ks then genKey else return (fromJust ks)
```

Jetzt lassen sich große Schlüsselpaare generieren. Die folgende Funktion ist eine kleine Testfunktion, die eine Zahl verschlüsselt und wieder entschlüsselt anzeigt.

Haskell: RSA

```
> testRSA1 = do
>   (pub,priv) <- genKey
>   let enc = encrypt pub
>
↪   123456789009876543211111222233334444555566667777888899990000
>   print (decrypt priv enc)
```

2.5 Texte Verschlüsseln

Das RSA-Verfahren verschlüsselt einzelne Zahlen. Diese Zahlen können maximal so groß wie der RSA-Modul werden.

Meistens möchte man Texte aus Unicode Zeichen verschlüsseln. Diese Texte sind also zur Verwendung des RSA-Verfahrens als Zahlen darzustellen. Ein einzelnes Zeichen braucht bereits eine 16 Bit Zahl.

Haskell: RSA

```
> unicodeSize = 216
```

Aufgabe 5

Schreiben Sie Funktionen, mit denen Sie einen String auf eine Integer-Zahl abbilden können und eine Inverse Funktion. Es soll also gelten:

```
id = integerToString.stringToInteger
```

Haskell: RSA

```
> stringToInteger :: String -> Integer
> stringToInteger xs = 1

> integerToString :: Integer -> String
> integerToString n = ""
```

Damit lassen sich Strings verschlüsseln, deren Abbild als Integer-Zahl kleiner als der RSA-Modul sind.

Haskell: RSA

```
> verschlüssel key = (encrypt key).stringToInteger
> entschlüssel key = integerToString.(decrypt key)
```

Testweise lassen sich für einen generierten Schlüsselpaar mal ein Text ver- und entschlüsseln.

Haskell: RSA

```
> test = do
> (pub,priv) <- genKey
> let enc = verschlüssel pub "hallo welt jetzt mal ein richtig langer
↪ text das wird doch wohl gut gehen"
> print (entschlüssel priv enc)
```

Aufgabe 6

Schreiben Sie eine Funktion, die einen String in eine Liste von Teilstrings der Länge n zerlegt. Es soll für $n > 0$ gelten:

`id = concat.splitString n`

Haskell: RSA

```
> splitString n str = []
```

Damit lassen sich jetzt Textdateien ver- und entschlüsseln:

Zum Verschlüsseln wird eine Datei in Teilstrings gesplittet und diese jeweils verschlüsselt und die Liste der verschlüsselten Teil-Strings gespeichert.

Haskell: RSA

```
> encryptFile key inFile outFile = do
> content <- readFile inFile
> writeFile outFile $ show $map (verschlüssel key) $ splitString 100
↪ content
```

Beim Entschlüsseln wird die verschlüsselte Datei als Liste von Integerzahlen gelesen, diese alle einzeln entschlüsselt und die Teil-Strings dann konkateniert.

Haskell: RSA

```
> decryptFile key inFile outFile = do
> content <- readFile inFile
> writeFile outFile $ concat $map (entschlüssel key) (read content)
```

In der Praxis werden nicht ganze Textdateien mit einem RSA-Schlüssel verschlüsselt, sondern ein Schlüssel generiert und mit RSA verschlüsselt. Dieser Schlüssel wird dann verwendet, um die eigentliche Textnachricht mit einem symmetrischen Verfahren (z.B. Advanced Encryption Standard (AES)) zu verschlüsseln.

Aufgabe 7

Sie sollten jetzt ein Programm haben, mit dem Sie RSA-Schlüssel generieren und ganze Dateien verschlüsseln und wieder entschlüsseln können. Testen Sie die Funktionalität aus und verschlüsseln und entschlüsseln große Dateien.

3 Lernzuwachs

Folgende Erkenntnisse sollte sich nach Studium dieses Kurses gesammelt haben.

- Mit Integer braucht man keine Angst vor großen Zahlen zu haben.
- Besonders rekursiv definierte Algorithmen lassen sich in Haskell oft direkt umsetzen.
- Im Zusammenhang mit Zufallszahlen und Dateien wurde schon mit der IO-Monade gearbeitet.
- Der Typ `Maybe` zum arbeiten mit Nullwerten.

Aufgabe 8

Das RSA Verfahren gibt es in Haskell im Modul `Codec.Crypto.RSA` des Pakets *RSA-1.0.6.2* implementiert [Wic18]. Machen Sie sich mit dem Modul vertraut. Schauen Sie sich die Implementierung an und vergleichen Sie diese mit der Implementierung unserer Aufgabe.

Literatur

- [Buc04] J. Buchmann. *Einführung in die Kryptographie*. Springer-Lehrbuch. Springer, 2004.
- [Fis11] Fischer, Daniel . *arithmoi: Efficient basic number-theoretic functions.*, 2011. [Online; accessed 25-March-2019].
- [RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.
- [RSA19] RSA-Kryptosystem. RSA-Kryptosystem — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 25-March-2019].
- [Wic18] Wick, Adam. *Rsa-1.0.6.2: Implementation of rsa, using the padding schemes of pkcs#1 v2.1.*, 2018. [Online; accessed 25-March-2019].