Implementation des RSA Verschlüsselungsalgorithmus in Haskell

Manuel Jenny & Christian Glatthard, Projektteam 5

4. Semester (FS 2013)

1 Abstract

Dies ist ein Semesterprojekt im Modul Konzepte von Programmiersprachen an der Fachhochschule Nordwestschweiz. Ziel der Arbeit besteht darin Übung im Umgang mit funktionalen Programmiersprachen, insbesondere Haskell, zu bekommen.

2 Idee des Projektes

Ziel unseres Projektes ist es eine funktionierende Implementierung des Verschlüsselungssystem RSA umzusetzen. Diese wird nicht den Sicherheitsstandards für produktive RSA Verschlüsselungen entsprechen, sondern sie soll einen Überblick über die Funktionsweise des RSA Verschlüsselungsverfahrens bieten. Es werden daher die drei wichtigsten Funktionen bereitgestellt: das Generieren von Schlüsseln (inkl. Primzahlenprüfung), sowie Ver- und Entschlüsselung von Strings.

Soweit möglich versuchen wir dabei die im Modul Kryptografie kennengelernten Algorithmen zu verwenden.

3 Theoretischer Teil

RSA (benannt nach den Erfindern Ron Rivest, Adi Shamir und Leonard Adleman) ist ein asymmetrisches kryptografisches Verschlüsselungsverfahren, welches sowohl zur Verschlüsselung, als auch zur digitalen Signatur verwendet werden kann.

Es wird ein privater und ein öffentlicher Schlüssel generiert. Der Öffentliche wird zum Verschlüsseln und zum Prüfen von Signaturen verwendet und ist für jedermann zugänglich. Der private Schlüssel hingegen wird zum Entschlüsseln, sowie zum Signieren der Daten verwendet und muss geheim bleiben.

In der Praxis wird RSA meist nur noch verwendet, um die Schlüssel eines anderen Verschlüsselungsverfahren zu verschlüsseln und diesen so sicher übertragen zu können. Für unser Projekt verwenden wir RSA jedoch direkt um Strings zu verschlüsseln.

3.1 Vorgehen RSA

1. Wähle 2 Primzahlen p, q
2. n = p*q
3. Wähle natürliche Zahl e, teilerfremd zu $\phi(n)$, d.h. $gcd(e, \phi(n)) = 1$, für die gilt $1 < e < \phi(n)$
4. Bestimme natürliche Zahl d mit $d = e^{-1} \in \mathbb{Z}^*_{\phi(n)}$, d.h. $e * d \equiv 1 \; mod \; \phi(n) \Longleftrightarrow e * d * + k * \phi(n) = 1$

3.2 Verwendete mathematische Formeln und Algorithmen

Für die Umsetzung von RSA sind einige mathematische Formeln und Definitionen unentbehrlich, diese haben wir hier aufgelistet.

3.2.1 erweiterter euklidischer Algorithmus

$ggT(a, b) = ax + by$

3.2.2 modular multiplikatives Inverse

$a^{-1} \equiv x \ (mod \ m)$

3.2.3 Eulersche Phi-Funktion

Die eulersche Phi-Funktion gibt aus wie viele Zahlen $<$ n teilerfremd sind zu n. $\phi(n) = a \in N | 1 \leq a \leq ngcd(a, n) = 1$
$\phi(mn) = \phi(m) * \phi(n)$

3.2.4 Modulares Multiplizieren

Um effizient mit grossen Exponenten zu arbeiten, benutzen wir die binäre Exponentiation in Kombination mit Modulo. Dieser Algorithmus ist wie folgt definiert:
Seien $a, b, n \in \mathbb{Z}$ und $b, n > 1$. Berechne a^b mod n.

1.) binäre Darstellung von b: $b = \sum_{i=0}^{k} \alpha_i 2^i$ mit $\alpha \in \{0, 1\}$.

2.) Anwendung auf a:
$a^{\sum_{i=0}^{k} \alpha_i 2^i} = \prod_{i=0}^{k} a^{\alpha_i 2^i} = a^{\alpha_k 2^k} * a^{\alpha_{k-1} 2^{k-1}} * a^{\alpha_{k-2} 2^{k-2}} \dots a^{\alpha_1 2} * a^{\alpha_0} = (\dots((a^{a_k})^2 * a^{a_{k-1}})^2 \dots * a^{\alpha_1})^2 * a^{\alpha_0}$

3.) Das Verfahren besteht nun darin, den letzten Ausdruck von innen nach aussen auszuwerten und nach jeder Multiplikation das Resultat modulo n zu rechnen.

4 Haskell-Code

4.1 Header

Wir schreiben sämtlichen Programmcode als Modul, so dass dieser auch in anderen Projekten genutzt werden kann. Dazu müssen sowohl Name des Moduls, als auch die nach aussen angebotenen Funktionen definiert werden (generateKeyPair, encrypt, decrypt). Per Import werden weitere Module geladen, die unser Programm benötigt.

Listing 1: Modul Header

```
1 -- # Define RSA module and its public functions
2 module RSA
3 (  generateKeyPair
4 ,  encrypt
5 ,  decrypt
6 ) where
7
8 -- # import required external modules
9 import System.Random
10 import Control.Monad.Fix
```

```
11 import Data.Int
12 import Data.Bits
13 import Data.Char
14 import System.IO
```

4.2 Funktion generateKeyPair

Die exportierten Funktionen sind interaktiv programmiert. Durch Nutzung von System.IO können die Primzahlen sowie der öffentliche Schlüssel eingegeben werden. Der private Schlüssel d wird mithilfe des inversen Modulo berechnet. Die Schlüssel werden je in einer separaten Datei gespeichert (pub.key, priv.key), und können so den entsprechenden Parteien zur Verfügung gestellt werden.

Listing 2: generateKeyPair

```
1  -- Interaction to generate key pair which are stored in pub.key/priv.key
2  generateKeyPair :: IO ()
3  generateKeyPair =
4      do putStrLn "
           ------------------------------------------------------------------------
           "
5          putStrLn "Key generation started (e = 65537): "
6          writeFile ("pub.key") ""
7          writeFile ("priv.key") ""
8          let e = 65537 :: Integer
9          putStrLn "NOTICE: If the primes don't match the requirements [(gcd e
               phi) <> 1]"
10         putStrLn "you will have to enter different ones."
11         putStrLn "Enter exponent (leave blank for default [65537])"
12         exp <- getLine
13         let e
14               | exp == "" = 65537 :: Integer
15               | otherwise = read exp :: Integer
16         primes <- enterPrimes e :: IO (Integer, Integer)
17         let p = fst primes
18             q = snd primes :: Integer
19             n = p*q
20             phi = (p-1)*(q-1)
21             d = inverseMod e phi :: Integer
22             resultPub = (e, n)
23             resultPriv = (d, n)
24         writeFile ("pub.key") (show resultPub)
25         writeFile ("priv.key") (show resultPriv)
26         putStrLn ("Key pair saved in pub.key and priv.key")
```

Listing 3: enterPrimes

```
1  -- Interaction to get fitting primes
2  -- Control.Monad.Fix idea from StackOverflow: http://stackoverflow.com/a
       /13301611
3  enterPrimes :: Integer -> IO (Integer, Integer)
4  enterPrimes e =
5      fix $ \again -> do
6          putStrLn "Enter first prime: "
7          prime <- getLine
8          let p = read prime
9          putStrLn "Enter second prime: "
```

```
10          prime <- getLine
11          let q = read prime :: Integer
12              phi = (p-1)*(q-1)
13          if ((gcd e phi) == 1 && (isPrime p) && (isPrime q)) then
14    -- ignore isPrime to use large possible primes generated with sage (http:
        //www.sagemath.org)
15          --if ((gcd e phi) == 1) then
16             return (p, q)
17          else
18             again
```

4.2.1 Funktion isPrime

Der folgende Code überprüft ob eine Zahl x eine Primzahl ist. Um die Performance zu steigern, wird vor der Listenabfrage überprüft, ob die Zahl x durch 2 (even), 3, 5 oder 7 teilbar ist. Ist dies der Fall, wird sofort False zurückgegeben. Ist dies nicht der Fall, wird über die Funktion getDivisorList eine Liste aller Zahlen von 2 bis $\sqrt{x}$ generiert.

Diese Liste wird nun mit der Funktion isNotDivisor auf Teiler von x überprüft. Sobald ein Teiler gefunden wird, liefert sie False zurück. Wird kein Teiler gefunden, ist die Zahl eine Primzahl und die Funktion gibt True zurück.

Listing 4: Überprüfen ob Zahl Primzahl ist

```
1  -- checks if x is prime
2  -- all even numbers equal directly to False
3  isPrime :: Integer -> Bool
4  isPrime 1 = False
5  isPrime 2 = True
6  isPrime x | even x == False && (not ((mod x 5) == 0)) && (not ((mod x 3) ==
       0)) && (not ((mod x 7) == 0)) = isNotDivisor (getDivisorList x) x
7            | otherwise = False
8
9  -- returns a list of possible divisors for x
10 getDivisorList :: Integer -> [Integer]
11 getDivisorList x = [y | y <- 2:filter (not.even) [2.. (round (sqrt (
       fromInteger x)))]]
12
13 -- checks wether there is an divisor or not
14 -- returns False as soon as one divisor is found
15 -- returns True if no divisor is found
16 isNotDivisor :: [Integer] -> Integer -> Bool
17 isNotDivisor [] _ = True
18 isNotDivisor (w:ws) x | (mod x w) == 0 = False
19                       | otherwise = isNotDivisor ws x
```

4.2.2 Funktion inverseMod

Das modulare multiplikative Inverse Modulo benötigt, um den geheimen Teil des privaten Schlüssels zu berechnen. Dies geschieht mit Hilfe des erweiterten euklidischen Algorithmus. Die zugrunde liegenden mathematischen Formeln haben wir von Wikipedia.

Listing 5: inverseMod

```
1  -- modular multiplicative inverse
2  inverseMod :: Integer -> Integer -> Integer
3  inverseMod e phi =
```

```
4   (x + phi) `mod` phi
5   where
6     (z, (x, y)) = ((gcd e phi),euclid e phi)
7
8 -- extended euclidean algorithm
9 euclid :: Integer -> Integer -> (Integer, Integer)
10 euclid 0 n = (0,1)
11 euclid e n
12   | n == 0 = (1,0)
13   | otherwise = (t, s-q*t)
14     where
15       (q, r) = quotRem e n
16       (s, t) = euclid n r
```

4.3 Funktion encrypt

Um einen String zu verschlüsseln wird der öffentliche Schlüssel benötigt. Dieser wird entweder aus einer zuvor mit generateKeyPair generierten oder einer von Hand erstellten Datei eingelesen. Die Funktion nimmt einen String auf, teilt ihn in möglichst grosse Blöcke (Blocklänge $<$ n) und verschlüsselt diese daraufhin Zeichen für Zeichen. Als Zahlenwert für die Zeichen wird die UTF8 Codierung verwendet. Damit bei der Entschlüsselung wieder dieselben Blöcke bearbeitet werden, wird der Cipher-Text (Geheimtext) in Form einer Integer-Liste ausgegeben, wobei jedes Element der Liste einen Nachrichten-Block repräsentiert. Dadurch ist natürlich die Sicherheit der Verschlüsselung bei kleinen Schlüsseln nicht gewährleistet, da gleiche Blöcke auch immer den gleichen Geheimtext produzieren. Nimmt man jedoch grosse Schlüssel, in der Praxis mindestens 2^{512}, so wird es nahezu unmöglich die Verschlüsselung zu knacken.

Listing 6: encrypt

```
1 -- interaction for encryption process
2 encrypt :: IO ()
3 encrypt =
4     do putStrLn "Please enter fileName which contains public key (e.g. pub.
        key): "
5        pubKeyFileName <- getLine
6        stringFileContents <- readFile (pubKeyFileName)
7        let en = read stringFileContents :: (Integer, Integer)
8            e = fst en
9            n = snd en
10       putStrLn "Please enter message to encrypt: "
11       message <- getLine
12       putStr "Encrypted text (as int blocks): "
13       putStrLn (show (encryptString e n message))
```

Listing 7: Hilfsfunktionen für encrypt

```
1 -- main function to encrypt strings
2 encryptString :: Integer -> Integer -> [Char] -> [Integer]
3 encryptString e n ms
4   | getNextPossibleCharBlockSize n == 0 = [-1]
5   | otherwise = encryptBlocks e n (getMessageBlocks ms (
      getNextPossibleCharBlockSize n))
6
7 -- bs = block list
8 encryptBlocks :: Integer -> Integer -> [Integer] -> [Integer]
9 encryptBlocks e n bs
```

```
10   | (length bs) == 1 = [encryptExec e n (head bs)]
11   | otherwise = [encryptExec e n (head bs)] ++ (encryptBlocks e n (tail bs))
12
13 -- build list of message blocks, b = block size
14 getMessageBlocks :: String -> Int -> [Integer]
15 getMessageBlocks m b
16   | (length m) <= b = [fromIntegral (charBlockToIntBlock m 0)]
17   | otherwise = [fromIntegral (charBlockToIntBlock (take b m) 0)] ++ (
       getMessageBlocks (drop b m) b)
18
19 -- cb = char block, e = exponent (start with 0)
20 charBlockToIntBlock :: [Char] -> Int -> Int
21 charBlockToIntBlock cb e
22   | (length cb) == 1 = (ord (head cb)) * (256^e)
23   | otherwise = ((ord (head cb)) * (256^e)) + charBlockToIntBlock (tail cb)
       (e+1)
24
25 -- get list of char blocks to encrypt / decrypt
26 -- info: chars are stored as utf8 (8bits)
27 getCharBlocks :: String -> Int -> [[Char]]
28 getCharBlocks m n
29   | (length m) <= (getNextPossibleCharBlockSize n) = [m]
30   | otherwise = [(take (getNextPossibleCharBlockSize n) m)] ++ (
       getCharBlocks (drop (getNextPossibleCharBlockSize n) m) n)
31
32 -- executes encryption
33 encryptExec :: Integer -> Integer -> Integer -> Integer
34 encryptExec e n m = powerMod m e n
```

4.4 Funktion decrypt

Für die Entschlüsselung wird eine Datei mit einem gültigen geheimen Schlüssel benötigt. Der Geheimtext muss in Form einer Integer-Liste angegeben werden, so wie er bei der encrypt-Methode generiert wird. Für die Entschlüsselung werden die Blöcke zuerst separat entschlüsselt und danach die einzelnen Zeichen wieder in UTF8 Zeichen umgewandelt und zu einem String zusammengesetzt.

Listing 8: decrypt

```
1  -- interaction for decryption process
2  decrypt :: IO ()
3  decrypt =
4      do putStrLn "Please enter fileName which contains private key (e.g. priv
           .key): "
5         privKeyFileName <- getLine
6         stringFileContents <- readFile (privKeyFileName)
7         let dn = read stringFileContents :: (Integer, Integer)
8             d = fst dn
9             n = snd dn
10        putStrLn "Please enter message to decrypt (as int blocks): "
11        cipher <- getLine
12        let c = read cipher :: [Integer]
13        putStr "Decrypted text: "
14        putStrLn (show (decryptString d n c))
```

Listing 9: Hilfsfunktionen für decrypt

```
-- main function to decrypt strings
decryptString :: Integer -> Integer -> [Integer] -> [Char]
decryptString d n cs
  | (getNextPossibleCharBlockSize n) == 0 = [' ']
  | otherwise = decryptBlocks d n cs

-- bs = blocklist
decryptBlocks :: Integer -> Integer -> [Integer] -> [Char]
decryptBlocks d n bs
  | (length bs) == 1 = intBlockToCharBlock (fromIntegral (decryptExec d n (
      head bs)))
  | otherwise = intBlockToCharBlock (fromIntegral (decryptExec d n (head bs)
      )) ++ (decryptBlocks d n (tail bs))

-- ib = int block, m = modulo, b = block size (chars)
intBlockToCharBlock :: Int-> [Char]
intBlockToCharBlock ib
  | ib == 0 = []
  | otherwise = [(chr (mod ib 256))] ++ intBlockToCharBlock (shiftR ib 8)

-- executes decryption
decryptExec :: Integer -> Integer -> Integer -> Integer
decryptExec d n c = powerMod c d n
```

5 Hilfsfunktionen

Hier sind die Hilfsfunktionen aufgeführt, welche von den verschiedenen Funktionen verwendet werden.

5.1 Funktion powerMod

Es wird eine Hilfsfunktion, powerModExec definiert, welche die eigentliche Berechnung übernimmt. Es wird die Liste von 0 und 1 abgearbeitet und das Resultat c zurückgegeben, sobald die Liste leer ist. Die Liste repräsentiert den Exponenten in binärer Form. Ist das erste Element der Liste eine 0 wird c (vorheriges Resultat) quadriert, modulo n genommen und als neues c rekursiv übergeben. Ist das erste Element der Liste eine 1 wird zusätzlich danach zusätzlich mit der Basis b multipliziert und das Resultat wieder modulo n genommen. Den so erhaltenen Rest wird dann als neues c übergeben.

Listing 10: powermod

```
-- modular exponentiation
powerMod :: Integer -> Integer -> Integer -> Integer
powerMod b e m = powerModExec b (toBin e) m 1

 -- modular exponentiation execution
powerModExec :: Integer -> [Integer] -> Integer -> Integer -> Integer
powerModExec b e m c
    | e == [] = c
    | head e == 1 = powerModExec b (tail e) m ((c^2 `mod ` m)*b `mod` m)
    | otherwise = powerModExec b (tail e) m (c^2 `mod` m)

-- convert Integer to an Integer list which represents original Integer in
    binary
toBin :: Integer -> [Integer]
toBin 0 = [0]
```

```
15 toBin 1 = [1]
16 toBin n
17     | n `mod` 2 == 0 = toBin (n `div` 2) ++ [0]
18     | otherwise = toBin (n `div` 2) ++ [1]
```

5.2 Funktionen für Blockgrössenbestimmung

Da n nicht unendlich gross ist, muss der zu verschlüsselnde Text m (falls $m > n$) in Blöcke aufgeteilt werden. Um die grösstmögliche Blockgrösse zu bestimmen, werden folgende Funktionen verwendet:

Listing 11: Blockgrössenbestimmung

```
1  -- if n < m -> RSA not possible, returns number of chars, not actual size
2  getNextPossibleCharBlockSize :: (Integral b, Num a, Ord a) => a -> b
3  getNextPossibleCharBlockSize n = snd (getNextSmallerPowerOfN 256 n)
4
5  -- returns last power of n which is still smaller than x
6  getNextSmallerPowerOfN :: (Integral b, Num t, Ord t) => t -> t -> (t, b)
7  getNextSmallerPowerOfN n x = getNextSmallerPowerOfNExec n x 1
8
9  getNextSmallerPowerOfNExec :: (Integral b, Num t, Ord t) => t -> t -> b -> (
       t, b)
10 getNextSmallerPowerOfNExec n x e
11   | x > (n^e) = getNextSmallerPowerOfNExec n x (e+1)
12   | x == (n^e) = (n^e,e)
13   | otherwise = (n^(e-1),(e-1))
```

6 Limitierungen

Da dieses Projekt dazu dient Erfahrungen in der Programmierung mit Haskell zu sammeln, ist die RSA-Implementierung sehr einfach gehalten und sollte **NICHT** als Produktivsystem verwendet werden. Zudem müssen folgende Punkte bei der Interaktion mit dem Modul berücksichtigt werden:

1. Die Primzahlen sollte nicht zu gross gewählt werden, da es bei zu grossen Zahlen ($p * q = n$) zu einem Overflow kommen kann ($n < max(Int32)$).
2. Die Rückführung der Integer-Liste in Cipher-Text wurde nicht berücksichtigt, da es hier zu Problemen mit der Zeichenkodierung kommen kann.

7 Testfälle

7.1 powerMod

powerMod ist kritisch für das Verschlüsseln mit grossen Zahlen. Um sicher zu gehen dass unsere Implementation korrekt ist, haben wir eine Testfunktion dafür geschrieben. Es ist zu beachten, dass dieser Test bei grossen Exponenten länger dauert!

Listing 12: testPowerMod

```
1 testPowerMod :: Integer -> Integer -> Integer -> Bool
2 testPowerMod b e m = powerMod b e m == mod (b^e) m
```

7.2 encryptExec, decryptExec

Um zu testen, dass die verschlüsselte Nachricht entschlüsselt wieder dieselbe ist, existiert auch hier eine Testfunktion:

Listing 13: testEncryptExec/testDecryptExec

```
1 testEncryptExec :: Integer -> Integer -> Integer -> Integer -> Bool
2 testEncryptDecryptExec e n m c = (encryptExec e n m) == c
3
4 testDecryptExec :: Integer -> Integer -> Integer -> Integer -> Bool
5 testDecryptExec d n m c = (decryptExec d n c) == m
```

Beispiel: $e = 65537$, $n = 1098922499$, $m = 123123$, $c = 942137302$
testEncryptExec: *testEncryptExec* 65537 1098922499 123123 942137302
Liefert: $True$
testDecryptExec: *testDecryptExec* 55783673 1098922499 942137302 123123
Liefert: $True$

7.3 encryptString, decryptString

Hier testen wir, ob die Zeicheneingabe richtig umgewandelt wird und ob aus der Integer-Liste wieder derselbe String zurück kommt.

Listing 14: testEncryptDecryptString

```
1 testEncryptDecryptString :: String -> Integer -> Integer -> Integer -> Bool
2 testEncryptDecryptString m e d n = m == (decryptString d n (encryptString d
      n m))
```

Beispiel: $e = 241$, $n = 341$, $m =$"Test?äöü!_-+ç", $d = 61$
testEncryptDecryptString m e d n
Liefert: $True$

8 Ehrlichkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der in den Kommentaren genannten Quellen angefertigt haben. Der Code unseres RSA Moduls ist so gut wie ausschliesslich im Pair-Programming entstanden und es kann nicht genau ermittelt werden, wer welche Codezeile beigesteuert hat. Wir versichern zudem, diese Arbeit nicht bereits anderweitig als Leistungsnachweis verwendet zu haben.

Windisch, 4. Juni 2013