

Implementation eines Verschlüsselungs-Algorithmus nach dem Beispiel von RSA in Haskell

Manuel Jenny & Christian Glatthard, Projektteam 5

4. Semester (FS 2013)

Inhaltsverzeichnis

1	Abstract	1
2	Idee des Projektes	1
3	Theoretischer Teil	1
3.1	Vorgehen RSA	1
3.2	Verwendete mathematische Formeln und Algorithmen	1
3.2.1	erweiterter euklidischer Algorithmus	1
3.2.2	modular multiplikatives Inverse	1
3.2.3	Eulersche Phi-Funktion	2
4	Haskell-Code	2
4.1	Header	2
4.2	Funktion generateKeyPair	2
4.2.1	Funktion enterPrimes	3
4.3	Funktion isPrime	3
4.3.1	Funktion inverseMod	4
4.4	Funktion encrypt	4
4.4.1	Funktion encryptString	5
4.4.2	Funktion powermod (Square & Multiply)	6
4.5	Funktion decrypt	6
4.5.1	Funktion decryptString	7

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 sämtliche Funktionen bereitgestellt wie das Generieren von Keys (inkl. Primzahlenerkennung) 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 öffentlich 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 Verschlüsselungsverfahrens 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 $d = e^{-1}$, d.h. $e * d \equiv 1 \mod \phi(n) \iff e * d + k * \phi(n) = 1$

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

$$\text{ggT}(a, b) = ax + by$$

3.2.2 modular multiplikatives Inverse

$$a^{-1} \equiv x \pmod{m}$$

3.2.3 Eulersche Phi-Funktion

Die eulersche Phi-Funktion gibt aus wieviele Zahlen $a \in \mathbb{N}$ teilerfremd sind zu n . $\phi(n) = a \in \mathbb{N} | 1 \leq a \leq n, \gcd(a, n) = 1$
 $\phi(mn) = \phi(m) * \phi(n)$

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 wird mithilfe des inversen Modulo berechnet. Die Schlüssel werden je in einem separaten File gespeichert (pub.key, priv.key), und können so theoretisch 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       -----
6       "
7       putStrLn "Key generation started (e = 65537): "
8       writeFile ("pub.key") ""
9       writeFile ("priv.key") ""
10      let e = 65537 :: Integer
11      putStrLn "NOTICE: If the primes don't match the requirements [(gcd e
12      phi) <> 1]"
13      putStrLn "you will have to enter different ones."
14      putStrLn "Enter exponent (leave blank for default [65537])"
15      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        if ((gcd e phi) == 1) then
15            return (p, q)
16        else
17            again

```

4.3 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) ==
7             0)) && (not ((mod x 7) == 0)) = isNotDivisor (getDivisorList x) x
8             | 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 whether there is a 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

```

4.3.1 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 zugrundeliegenden 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

```

4.4 Funktion encrypt

Um einen String zu verschlüsseln wird der öffentliche Schlüssel benötigt. Dieser wird entweder aus einem zuvor mit generateKeyPair generierten, oder einem von Hand erstellten File 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 jeweils der UTF8 Code 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 darstellt. 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, 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))

```

4.4.1 Funktion encryptString

Listing 7: encryptString

```

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 -- if n < m -> RSA not possible, returns number of chars, not actual size
33 getNextPossibleCharBlockSize :: (Integral b, Num a, Ord a) => a -> b
34 getNextPossibleCharBlockSize n = snd (getNextSmallerPowerOfN 256 n)
35
36 -- returns last power of b which is still smaller than x

```

```

37 getNextSmallerPowerOfN :: (Integral b, Num t, Ord t) => t -> t -> (t, b)
38 getNextSmallerPowerOfN b x = getNextSmallerPowerOfN2 b x 1
39
40 getNextSmallerPowerOfN2 :: (Integral b, Num t, Ord t) => t -> t -> b -> (t,
    b)
41 getNextSmallerPowerOfN2 b x e
42   | x > (b^e) = getNextSmallerPowerOfN2 b x (e+1)
43   | x == (b^e) = (b^e,e)
44   | otherwise = (b^(e-1),(e-1))

```

4.4.2 Funktion powermod (Square & Multiply)

Listing 8: powermod

```

1 -- modular exponentiation
2 powerMod :: Integer -> Integer -> Integer -> Integer
3 powerMod b e m = powerModExec b (toBin e) m 1
4
5 -- modular exponentiation execution
6 powerModExec :: Integer -> [Integer] -> Integer -> Integer -> Integer
7 powerModExec b e m c
8   | e == [] = c
9   | head e == 1 = powerModExec b (tail e) m ((c^2 'mod' m)*b 'mod' m)
10  | otherwise = powerModExec b (tail e) m (c^2 'mod' m)
11
12 -- convert Integer to an Integer list which represents original Integer in
    binary
13 toBin :: Integer -> [Integer]
14 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]

```

4.5 Funktion decrypt

Für die Verschlüsselung wird ein File mit einem gültigen Privatekey benötigt. Der Geheimtext muss in Form einer Integer-Liste angegeben werden, so wie er bei der Decrypt 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 9: encrypt

```

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))

```

4.5.1 Funktion decryptString

Listing 10: encrypt

```
1 -- main function to decrypt strings
2 decryptString :: Integer -> Integer -> [Integer] -> [Char]
3 decryptString d n cs
4   | (getNextPossibleCharBlockSize n) == 0 = [' ']
5   | otherwise = decryptBlocks d n cs
6
7 -- bs = blocklist
8 decryptBlocks :: Integer -> Integer -> [Integer] -> [Char]
9 decryptBlocks d n bs
10  | (length bs) == 1 = intBlockToCharBlock (fromIntegral (decryptExec d n (
    head bs)))
11  | otherwise = intBlockToCharBlock (fromIntegral (decryptExec d n (head bs)
    )) ++ (decryptBlocks d n (tail bs))
12
13 -- ib = int block, m = modulo, b = block size (chars)
14 intBlockToCharBlock :: Int -> [Char]
15 intBlockToCharBlock ib
16   | ib == 0 = []
17   | otherwise = [(chr (mod ib 256))] ++ intBlockToCharBlock (shiftR ib 8)
```

5 Testfälle

- powermod - encryptExcc - encryptString - decryptExec - decryptString