kpsp Zwischenbericht

Manuel Jenny & Christian Glatthard

4. Semester (FS 2013)

Inhaltsverzeichnis

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 bereitgestellen 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.

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

Bisher verwenden wir für unseren RSA Algorithmus die folgenden mathematischen Definitionen.

3.2.1 erweiterter euklidischer Algorithmus

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

3.2.2 modulare multiplikative Inverse

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

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. 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.Bits
```

4.2 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 2: 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
```

4.3 Funktion inverseMod

Das multiplikative inverse Modulo wird benötigt um den einen Teil des privaten Schlüssels zu berechnen. Dies geschieht mit Hilfe des erweiterten euklidischen Algorithmus. Die zugrundeliegenden mathematischen Formeln haben wir von Wikipedia.

Listing 3: Teil d des Privatekeys berechnen

```
1  -- modular multiplicative inverse
2  inverseMod :: Integer -> Integer -> Integer
3  inverseMod e n =
4    (x + n) `mod` n
5    where
6      (z, (x, y)) = ((gcd e n),euclid e n)
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
```

5 Testfälle

tbd