kpsp: Schlussbericht hybride Kryptographie

Marianne Schoch, Pascal Schwarz

4. Juni 2013

1 Abstract

Dieser Teil wird im finalen Bericht enthalten sein.

2 Einleitung / Idee

Die Software erlaubt die Erzeugung von verschlüsselten und signierten Nachrichten sowie deren Entschlüsselung. Die Nachrichten werden in Dateien gespeichert resp. aus Dateien gelesen.

Die Nachricht selbst wird mit einem symmetrischen Verfahren unter Verwendung eines zufälligen Schlüssels verschlüsselt. Dieser Schlüssel wird dem Empfänger der Nachricht ebenfalls übermittelt. Dazu wird der public Key des asymmetrischen Verschlüsselungsverfahrens verwendet.

Um die Integrität der Nachricht sicherstellen zu können wird des Weiteren eine Signatur, wiederum mit Hilfe des asymmetrischen Verschlüsselungsverfahrens und einer zusätzlichen Hashfunktion, der Nachricht beigefügt.

Zusätzlich ermöglichen wir die Erzeugung von Schlüsselpaaren für das asymmetrische Verschlüsselungsverfahren.

3 Theorie

3.1 Nachrichtenformat

3.1.1 Überblick

Die zu übermittelnde Datei besteht aus drei Teilen, die in den nachfolgenden Abschnitten beschrieben werden. Die Teile werden dabei in der folgenden Art markiert:

```
1 ----BEGIN <Abschnittname> [Option]----
2 <Inhalt des Abschnittes, Base64 kodiert>
3 ----END <Abschnittname>----
```

3.1.2 Teil KEYCRYPTED

Enthält den zufälligen „Sitzungsschlüssel“, der für das symmetrische Verschlüsselungsverfahren verwendet wird. Als Option wird das verwendete asymmetrische Verschlüsselungsverfahren angegeben. Da bei RSA die Länge des Plaintexts bekannt sein muss, wird diese (in Bytes) ebenfalls angegeben. Die Abschnittsmarkierung kann dann beispielsweise folgendermassen aussehen:

```
1 ----BEGIN KEYCRYPTED RSA 8----
2 <Base64 kodiertes Resultat der RSA-Verschluesselung des
      Sitzungsschluessels>
3 ----END KEYCRYPTED----
```

Die Anwendung von RSA auf den Sitzungsschlüssel wird in Gruppen von 4 Bytes vorgenommen. Dabei werden die Bytes konkateniert und als Zahl intepretiert.

Für Keys, deren Länge nicht ohne Rest durch vier teilbar sind (dies ist beispielsweise bei DES der Fall) wird der Input um Nullen ergänzt. Damit diese Nullen beim Entschlüsseln ignoriert werden können, wird die Längenangabe benötigt.

3.1.3 Teil MSGCRYPTED

Für die Verschlüsselung der eigentlichen Nachricht kommt ein symmetrisches Verfahren zum Einsatz. Falls es sich dabei um eine Blockchiffre handelt, wird neben dem Namen des Algorithmus ebenfalls angegeben, in welchem Modus die Blöcke verkettet werden. Falls ein Initialisierungsvektor benötigt wird, wird dieser zufällig erzeugt und in diesem Teil der Nachricht, mit einem „,“ vom Ciphertext separiert, abgelegt.

Kommt AES mit CBC als Modus zum Einsatz, sieht der Abschnitt folgendermassen aus:

```
1 ----BEGIN MSGCRYPTED AES256 CBC----
2 <Base64 kodierter IV>,<Base64 kodiertes Resultat der Verschluesselung>
3 ----END MSGCRYPTED----
```

3.1.4 Teil SIGNATURE

Die Signatur wird erzeugt, indem die verschlüsselten Inhalte der Teile KEYCRYPTED und MSGCRYPTED konkateniert werden. Nach der Anwendung eines Hash-Verfahrens wird beispielsweise RSA für die Erstellung der Signatur verwendet. Die Optionen für diesen Teil der Datei enthalten das verwendete Hashverfahren (woraus die Länge des Hashes abgeleitet werden kann) als auch das für die Signatur verwendete Kryptosystem. Ein Beispiel mit SHA256 und RSA sähe demnach so aus:

```
1 ----BEGIN SIGNATURE SHA256 RSA----
2 <Base64 kodiertes Resultat der RSA-Signierung von
      SHA256(KEYCRYPTED|MSGCRYPTED)>
3 ----END SIGNATURE----
```

3.2 Schlüsselformat

3.2.1 RSA

RSA Schlüssel bestehen aus Exponent (e oder d) und dem Produkt der beiden Primzahlen (n). Diese werden in einer Datei mit dem folgenden Format gespeichert:

```
1 ----BEGIN RSA PUBLIC KEY----
2 <Base64 kodierte Binaerdarstellung von e>,<Base64 kodierte
      Binaerdarstellung von n>
3 ----END RSA PUBLIC KEY----
```

3.3 Verwendung

3.3.1 Dateien

plain Datei mit zu verschlüsselndem Inhalt oder Resultat der Entschlüsselung

crypt Datei mit Resultat der Verschlüsselung

rsapriv Datei mit eigenem privatem RSA-Schlüssel

rsapub Datei mit eigenem öffentlichem RSA-Schlüssel

rsapubrecv Datei mit öffentlichem RSA-Schlüssel des Empfängers

3.3.2 Schlüsselerzeugung

Erzeugung eines Schlüsselpaares:

```
1 ./hskeygenerator
```

3.3.3 Ver- und Entschlüsselung

Verschlüsselung:

```
1 ./hsencrypt <asymm. kryptosystem> <hashverfahren> <symm. kryptosystem>
      <modus> <empfaenger public key> <verschluesselte datei>
```

Entschlüsselung:

```
1 ./hsdecrypt <sender publickey> <verschluesselte datei>
```

4 Implementation

4.1 Hauptfiles

4.1.1 hskeygenerator.hs

```
1  import System.Environment
2  import Kpspcrypto.RSAKey
3  import System.Random
4
5  import qualified Data.ByteString.Char8 as B
6
7  main =
8          do      gen <- newStdGen
9                  putStrLn "Enter filename:"
10                 fileName <- getLine
11                 B.writeFile (fileName ++ "RsaPrivKey") $ getPrivK $ genK
                       gen
12                 B.writeFile (fileName ++ "RsaPubKey") $ getPubK $ genK
                       gen
13                       where
14                                         getPrivK (x, _) = x
15                                         getPubK (_, y) = y
```

4.1.2 hsencrypt.hs

```
1  -- needed for using string-literals with ByteString
2  -- see http://hackage.haskell.org/packages/archive/
       bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
3  {-# LANGUAGE OverloadedStrings #-}
4
5  -- runhaskell -XOverloadedStrings hsencrypt.hs params...
6
7  import System.Environment
8  import System.Random
9  import qualified Data.ByteString.Char8 as B
10 import Kpspcrypto.Msg
11 import qualified Kpspcrypto.MsgCrypted as M
12 import qualified Kpspcrypto.KeyCrypted as K
13 import qualified Kpspcrypto.Signature as S
14
15 main = do
16          args <- getArgs
17          handleArgs $ map B.pack args
18
19
20 handleArgs :: [B.ByteString] -> IO()
21 handleArgs args = do
22          if length args /= 7 then do
23                  printUsage
24          else do
25                  let asym = args !! 0
26                  let hash = args !! 1
```

```
27                  let sym = args !! 2
28                  let blockmode = args !! 3
29                  let ownprivkey = args !! 4
30                  let rcptpubkey = args !! 5
31                  let infile = args !! 6
32                  pubkey <- B.readFile $ B.unpack rcptpubkey
33                  privkey <- B.readFile $ B.unpack ownprivkey
34                  plainFileContent <- B.readFile $ B.unpack infile
35                  rgen <- getStdGen
36                  let (mMsgPart, symkey) = M.genMsgPart rgen sym blockmode
                        plainFileContent --generates the crypted Message
37                  let kMsgPart = K.genMsgPart asym pubkey symkey
                        --generates the crypted Key
38                  let plainS = [kMsgPart,mMsgPart]
39                  let sMsgPart = S.genMsgPart asym privkey hash $ plainS
                        --generates the signature
40                  let msgParts = map (B.pack . show)
                        [kMsgPart,mMsgPart,sMsgPart]
41                  B.writeFile (B.unpack infile ++ "Encrypted") $
                        B.intercalate "\n\n" msgParts
42
43  printUsage :: IO()
44  printUsage = do
45          putStrLn "you need to call this binary in this way:"
46          putStrLn "hsencrypt asymCipher hashAlg symCipher chainingMode
                privKey publicKey plainFile"
47          putStrLn "Example: hsencrypt RSA SHA256 AES256 CBC privkey
                pubkey plaintext.txt"
```

4.1.3 hsdecrypt.hs

```
1  -- needed for using string-literals with ByteString
2  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
3  {-# LANGUAGE OverloadedStrings #-}
4
5  -- runhaskell -XOverloadedStrings hsencrypt.hs params...
6
7  import System.Environment
8  import Data.List
9
10 import Kpspcrypto.Msg
11 import qualified Kpspcrypto.KeyCrypted as K
12 import qualified Kpspcrypto.MsgCrypted as M
13 import qualified Kpspcrypto.Signature as S
14 import qualified Data.ByteString.Char8 as B
15
16 main = do
17          args <- getArgs
18          handleArgs args
19
20 handleArgs :: [String] -> IO()
21 handleArgs args = do
22          if length args /= 3 then do
23                  printUsage
```

```
24          else do
25                  [ourprivkey, senderpubkey, cryptcontent] <- mapM
                        B.readFile args
26                  let parts@[keypart,msgcpart,sigpart] = sort $
                        getMsgParts cryptcontent
27                  let sigOK = S.verifySig senderpubkey parts
28                  if sigOK then do
29                          let symkey = K.getSymKey ourprivkey keypart
30                          let plaintext = M.getPlain symkey msgcpart
31                          if "Encrypted" `isSuffixOf` (args !! 2) then do
32                                  let plainFile = dropEnd 9 $ args !! 2
33                                  B.writeFile plainFile plaintext
34                          else do
35                                  putStrLn "Output File?"
36                                  plainFile <- getLine
37                                  B.writeFile plainFile plaintext
38                  else do
39                          putStrLn "signature or key was wrong, exiting..."
40
41  printUsage :: IO()
42  printUsage = do
43          putStrLn "you need to call this binary in this way:"
44          putStrLn "hsdecrypt yourPrivKey sendersPublicKey cryptedfile.txt"
45
46  dropEnd :: Int -> [a] -> [a]
47  dropEnd n = reverse . drop n . reverse
```

4.2 Zwischenfiles

4.2.1 KeyCrypted.hs

```
1  module Kpspcrypto.KeyCrypted where
2
3  -- needed for using string-literals with ByteString
4  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5  {-# LANGUAGE OverloadedStrings #-}
6
7  import qualified Data.ByteString.Char8 as B
8  import qualified Data.Map as M
9  import Data.Maybe
10
11 import Kpspcrypto.Msg
12 import Kpspcrypto.Pad
13 import Kpspcrypto.Serial
14 import qualified Kpspcrypto.Base64 as B64
15 import qualified Kpspcrypto.RSA as RSA
16
17 -- creates a KEYCRYPTED-msgpart using the given asymmetric
18 -- cipher, the given key for the asymmetric cipher and
19 -- the given content in encrypted form
20 genMsgPart :: AsymCipher -> AsymKey -> B.ByteString -> MsgPart
21 genMsgPart "RSA" akey skey = MsgPart KEYCRYPTED ["RSA"] enckey
22         where
```

```
23                  enckeyed = map B64.encode [RSA.encrypt akey blocks |
                        blocks <- block 4 skey]
24                  enckey = B.intercalate "," enckeyed
25
26  getSymKey :: AsymKey -> MsgPart -> B.ByteString
27  getSymKey akey msg = (fromJust $ M.lookup cipher ciphers) akey msg
28          where
29                  cipher = head $ options msg
30
31  getSymKeyFromRSA :: AsymKey -> MsgPart -> B.ByteString
32  getSymKeyFromRSA akey msg = B.concat [RSA.decrypt akey $ B64.decode
        block | block <- B.split ',' $ content msg]
33
34  ciphers :: M.Map B.ByteString (AsymKey -> MsgPart -> B.ByteString)
35  ciphers = M.fromList [("RSA", getSymKeyFromRSA)]
36
37
38  {-----------------------
39  sample data and tests
40  -----------------------}
41  rsapubkey = "-----BEGIN RSA PUBLIC KEY-----\nBrk=,BAYh\n-----END RSA PUBLIC
        KEY-----" :: B.ByteString
42  rsaprivkey = "-----BEGIN RSA PRIVATE KEY-----\nBV0=,BAYh\n-----END RSA
        PRIVATE KEY-----" :: B.ByteString
43
44  -- reicht fuer 4 bytes :)
45  rsapriv2 = "-----BEGIN RSA PRIVATE KEY-----\nzFEWC0E=,AQro6bcX\n-----END
        RSA PRIVATE KEY-----" :: B.ByteString
46  rsapub2 = "-----BEGIN RSA PUBLIC KEY-----\nAQAB,AQro6bcX\n-----END RSA
        PUBLIC KEY-----" :: B.ByteString
47
48  ourdata = "ourdata" :: B.ByteString
49
50  simplegentest = genMsgPart "RSA" rsapriv2 ourdata
51  simplegetKeytest = getSymKey rsapub2 simplegentest
```

4.2.2 MsgCrypted.hs

```
1   module Kpspcrypto.MsgCrypted (genMsgPart, getPlain) where
2
3   -- needed for using string-literals with ByteString
4   -- see
        http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5   {-# LANGUAGE OverloadedStrings #-}
6
7   import qualified Data.ByteString.Char8 as B
8   import qualified Data.Map as M
9   import Data.Maybe
10  import System.Random
11  import Data.Char
12
13  import qualified Kpspcrypto.AES256 as AES
14  import qualified Kpspcrypto.Base64 as B64
15  import Kpspcrypto.Msg
16  import Kpspcrypto.BlockModes
```

```
17 import Kpspcrypto.Pad
18
19 type Key = B.ByteString
20 type SymCipher = B.ByteString
21 type ChainMode = B.ByteString
22
23 -- create a MSGCRYPTED-part using a random IV and a random Key, also
24 -- returns the used key for further usage (in the KEYCRYPTED part)
25 genMsgPart :: StdGen -> SymCipher -> ChainMode -> B.ByteString ->
       (MsgPart, Key)
26 genMsgPart rgen "AES256" "CBC" plain = (MsgPart MSGCRYPTED
       ["AES256","CBC"] (ivenc `B.append` "," `B.append` plainenc), key)
27           where
28                   [key,iv] = rndStrs [32,16] rgen
29                   plainenc = B64.encode $ (cbc (AES.encode key) iv) plain
30                   ivenc = B64.encode iv
31 genMsgPart rgen "AES256" "ECB" plain = (MsgPart MSGCRYPTED
       ["AES256","ECB"] plainenc, key)
32           where
33                   key = rndStr 32 rgen
34                   plainenc = B64.encode $ (ecb (AES.encode key) iv) plain
35                   --only length matters, must be the same as the blocksize
                         of the cipher
36                   iv = B.replicate 16 '\0'
37
38 -- decodes the content of a MSGCRYPTED-part using the supplied key
39 getPlain :: Key -> MsgPart -> B.ByteString
40 getPlain key msg = (fromJust $ M.lookup cipher decodingfunctions) key msg
41           where
42                   cipher = head $ options msg
43
44 -- decodes the content of a part encrypted using AES
45 getPlainFromAES :: Key -> MsgPart -> B.ByteString
46 getPlainFromAES key msg = (modef (AES.decode key) iv) cont
47           where
48                   mode = options msg !! 1
49                   -- find the function which "unapplies" the
                         block-chaining mode
50                   modef = fromJust $ M.lookup mode modes
51                   -- for ecb we have to supply a pseudo-iv, for cbc the iv
                         is
52                   -- part of the content of the msgpart
53                   [iv, cont]       | mode == "ECB" = [B.replicate 16 '\0',
                         B64.decode $ content msg]
54                                    | mode == "CBC" = map B64.decode
                                        $ B.split ',' $ content msg
55
56 -- maps the option-value in the msgpart-header to the function
57 -- responsible for decoding a part
58 decodingfunctions :: M.Map B.ByteString (Key -> MsgPart -> B.ByteString)
59 decodingfunctions = M.fromList [("AES256", getPlainFromAES)]
60
61 -- maps the option-value in the msgpart-header to the function
62 -- responsible for "unapplying" the block-chaining
```

```
63  modes :: M.Map B.ByteString ((Block -> Block) -> IV -> B.ByteString ->
        B.ByteString)
64  modes = M.fromList [("CBC",uncbc), ("ECB",unecb)]
65
66  {-----------------------------------
67  creating random keys, ivs etc...
68  -----------------------------------}
69  -- takes a list of lengths and returns random ByteStrings with
70  -- these lengths created using the supplied random generator
71  rndStrs :: [Int] -> StdGen -> [B.ByteString]
72  rndStrs lengths gen = split lengths allrndstrs
73          where
74                  split [] "" = []
75                  split (len:lens) tosplit = B.take len tosplit : (split
                        lens (B.drop len tosplit))
76                  allrndstrs = rndStr (sum lengths) gen
77
78  -- creates a random ByteString with given length using the
79  -- supplied random generator
80  rndStr :: Int -> StdGen -> B.ByteString
81  rndStr n gen = B.pack $ rndCL n gen
82
83  -- creates a random String with given length using the
84  -- supplied random generator
85  -- the reason for creating this method is that prepending
86  -- single Chars to [Char] is way faster (O(1)) the same operation
87  -- than on a ByteString (O(n))
88  rndCL :: Int -> StdGen -> String
89  rndCL 0 gen = ""
90  rndCL n gen = chr rc : rndCL (n-1) newgen
91          where
92                  (rc, newgen) = randomR (0,255) gen
93
94  {----
95  tests
96  ----}
97  runTests :: Bool
98  runTests = and [testAESECB,testAESCBC]
99
100 contents :: [B.ByteString]
101 contents =      ["the␣very␣secret␣and␣hopefully␣somewhat␣protected␣
        plaintext"
102                         ,"another,␣shorter␣text"
103                         ,""
104                         ,B.replicate 5000 't'
105                         ]
106
107 testAESECB :: Bool
108 testAESECB = and [plain m == getPlain (key m) (msg m) | m <- msgskeys]
109         where
110                 msgskeys = [(genMsgPart rnd "AES256" "ECB" cont,cont) |
                        rnd <- rnds, cont <- contents]
111                 msg = fst . fst
112                 key = snd . fst
113                 plain = snd
```

```
114
115 testAESCBC :: Bool
116 testAESCBC = and [plain m == getPlain (key m) (msg m) | m <- msgskeys]
117         where
118                 msgskeys = [(genMsgPart rnd "AES256" "CBC" cont,cont) |
                        rnd <- rnds, cont <- contents]
119                 msg = fst . fst
120                 key = snd . fst
121                 plain = snd
122
123 rnds :: [StdGen]
124 rnds = [mkStdGen i | i <- [13..63]]
```

4.2.3 Signature.hs

```
 1 module Kpspcrypto.Signature where
 2
 3 -- needed for using string-literals with ByteString
 4 -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
 5 {-# LANGUAGE OverloadedStrings #-}
 6
 7 import qualified Data.ByteString.Char8 as B
 8 import qualified Data.Map as M
 9 import Data.List
10 import Data.Maybe
11
12 import qualified Kpspcrypto.Base64 as B64
13 import qualified Kpspcrypto.RSA as RSA
14 import qualified Kpspcrypto.SHA256 as SHA
15 import Kpspcrypto.Pad
16 import Kpspcrypto.Msg
17
18 -- create a signature msgpart which contains the signed hash of the
       other msgparts
19 genMsgPart :: AsymCipher -> AsymKey -> HashType -> [MsgPart] -> MsgPart
20 genMsgPart "RSA" akey "SHA256" [kcpart,msgcpart] = MsgPart SIGNATURE
       ["RSA","SHA256"] signature
21         where
22                 hashed = SHA.hash $ B.concat [content kcpart, content
                       msgcpart]
23                 signed = map B64.encode [RSA.sign akey blocks | blocks
                       <- block 6 hashed]
24                 signature = B.intercalate "," signed
25
26
27 -- verifies the signature of the whole msg
28 verifySig :: AsymKey -> [MsgPart] -> Bool
29 verifySig akey parts = and $ zipWith (checksig akey) bsigs bs
30         where
31                 [kpart,mpart,spart] = sort parts
32                 [k,m,s] = map content [kpart,mpart,spart]
33                 msgh = hashf $ k `B.append` m
34                 bsigs = [B64.decode block | block <- B.split ',' s]
35                 bs = block 6 msgh
```

```
36                  sigtype = options spart !! 0
37                  hashtype = options spart !! 1
38                  checksig = fromJust $ M.lookup sigtype checksigs
39                  hashf = fromJust $ M.lookup hashtype hashfs
40
41  -- contains the hashfunctions, key is the value of the option in the
        MsgPart-Header
42  hashfs :: M.Map HashType (B.ByteString -> B.ByteString)
43  hashfs = M.fromList [("SHA256",SHA.hash)]
44
45  -- contains the "check signature" functions, key is the value of the
        option in the MsgPart-Header
46  checksigs :: M.Map AsymCipher (AsymKey -> B.ByteString -> B.ByteString
        -> Bool)
47  checksigs = M.fromList [("RSA",RSA.checksig)]
48
49  {----------------------
50  sample data and tests
51  ----------------------}
52  runTests :: Bool
53  runTests = and [verifySig pub $ (genMsgPart "RSA" priv "SHA256"
        otherparts) : otherparts | (pub,priv) <- keys]
54          where
55                  otherparts = [kcpart,msgcpart]
56                  kcpart = MsgPart KEYCRYPTED ["RSA"]
                        "ourkeyourkeyourkeyourkeyourkey"
57                  msgcpart = MsgPart MSGCRYPTED ["SHA256","CBC"]
                        "ourdataourdataourdataourdataourdata"
58
59  keys :: [(B.ByteString,B.ByteString)]
60  keys =  [("-----BEGIN RSA PUBLIC KEY-----\nAQAB,iUdRIBeyL3qX\n-----END RSA
        PUBLIC KEY-----"
61                  ,"-----BEGIN RSA PRIVATE
                        KEY-----\nH0vn/c/pBfHZ,iUdRIBeyL3qX\n-----END RSA
                        PRIVATE KEY-----")
62                  ,("-----BEGIN RSA PUBLIC
                        KEY-----\nAQAB,Y9G9TSdJNf0j\n-----END RSA PUBLIC
                        KEY-----"
63                  ,"-----BEGIN RSA PRIVATE
                        KEY-----\nH0jRB6FRS9Th,Y9G9TSdJNf0j\n-----END RSA
                        PRIVATE KEY-----")
64                  ,("-----BEGIN RSA PUBLIC
                        KEY-----\nAQAB,Lfh0qKrNlchv\n-----END RSA PUBLIC
                        KEY-----"
65                  ,"-----BEGIN RSA PRIVATE
                        KEY-----\nEVl5vcC88PQh,Lfh0qKrNlchv\n-----END RSA
                        PRIVATE KEY-----")
66                  ]
```

4.3 Symmetrische Verschlüsselung

4.3.1 AES256.hs

```
1  module Kpspcrypto.AES256 (encode, decode) where
```

```
2
3  -- needed for using string-literals with ByteString
4  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5  {-# LANGUAGE OverloadedStrings #-}
6
7  {--
8  this module uses Codec.Encryption.AES from the "crypto"-Package
9  to perform the actual encryption and decryption functions
10 the Word128-Interface of Codec.Encryption.AES is converted to
11 a simpler to use Interface using ByteStrings
12 --}
13
14 import qualified Data.ByteString.Char8 as B
15 import qualified Codec.Encryption.AES as CEAES
16 import Data.Word
17 import Data.LargeWord
18
19 import Kpspcrypto.Serial
20
21 type Block = B.ByteString
22 type Key = B.ByteString
23
24 {-------------------
25 encoding functions
26 -------------------}
27 -- crypt a single block using ECB and the supplied key
28 encode :: Key -> Block -> Block
29 encode key plain = w1282b $ CEAES.encrypt keyw plainw
30         where
31                 keyw = b2w128 key
32                 plainw = b2w128 plain
33
34 {-------------------
35 decoding functions
36 -------------------}
37 decode :: Key -> Block -> Block
38 decode key crypted = w1282b $ CEAES.decrypt keyw cryptedw
39         where
40                 keyw = b2w128 key
41                 cryptedw = b2w128 crypted
42
43
44 {-------
45 helpers
46 -------}
47 -- converts the last 16 Bytes of a BString to a Word128
48 b2w128 :: B.ByteString -> Word128
49 b2w128 = fromIntegral . asInt
50
51 -- converts a Word128 to a 16 Byte BString
52 w1282b :: Word128 -> B.ByteString
53 w1282b s = B.replicate (16-clen) '\0' `B.append` converted
54         where
55                 converted = asStr $ toInteger s
```

```
56                  clen = B.length converted
57
58 {----
59  tests
60 ----}
61 -- tests whether a given string (first in tuple) which gets
62 -- encrypted and then decrypted using the same or different
63 -- key(s) is (not) the same as the original string
64
65 runTests :: Bool
66 runTests = and [testAES test | test <- tests]
67
68 -- runs tests from "tests"
69 -- compares d(e(plain)) and plain using the supplied eq-function
70 -- see comment of "tests" for further information
71 testAES :: (Block,(Block->Block->Bool),(Block->Block),(Block->Block)) ->
       Bool
72 testAES (plain,eq,e,d) = plain `eq` (d $ e plain)
73
74 -- different keys
75 key1 = "justAKeyjustBKey" :: B.ByteString
76 key2 = "justBKeyjustAKey" :: B.ByteString
77
78 -- partially apply the encode and decode functions using a key
79 -- results in functions of the type (Block -> Block)
80 e1 = encode key1
81 e2 = encode key2
82 d1 = decode key1
83 d2 = decode key2
84
85 -- (plaintext, equality-function, encoding function, decoding function)
86 -- the equality-function should return true if d(e(plain)) matches
87 -- the expected result, if you decode using another key than the one
88 -- used for encode, you expect the result to be different from plain,
89 -- thus you need to supply (/=) as "equality"-function
90 tests :: [(Block,(Block->Block->Bool),(Block->Block),(Block->Block))]
91 tests = [(nulls,(==),e1,d1)
92                  ,(nulls,(/=),e1,d2)
93                  ,(nulls,(/=),e2,d1)
94                  ,(nulls,(==),e2,d2)
95                  ,(as,(==),e1,d1)
96                  ,(as,(/=),e1,d2)
97                  ,(as,(/=),e2,d1)
98                  ,(as,(==),e2,d2)
99                  ]
100          where
101                  nulls = B.replicate 16 '\0'
102                  as = B.replicate 16 'a'
```

4.3.2 SHA256.hs

```
1  module Kpspcrypto.SHA256 (hash) where
2
3  -- http://en.wikipedia.org/wiki/SHA-2
4
```

```
5  -- needed for using string-literals with ByteString
6  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
7  {-# LANGUAGE OverloadedStrings #-}
8
9  import qualified Data.ByteString.Char8 as B
10 import Data.Word
11 import Data.Bits
12 import Data.Char
13 import Text.Printf -- for tests only
14
15 import Kpspcrypto.Pad
16 import Kpspcrypto.Serial
17
18 -- apply SHA256 to given data, result will always be 32 bytes long
19 hash :: B.ByteString -> B.ByteString
20 hash msg = B.concat $ map w2b h
21         where
22                 h = foldl perchunk hs preprocessed
23                 preprocessed = chunks $ preprocess msg
24
25 -- adds padding (in the form 0x80[00]*) until there are 4 bytes left for
       the size
26 shapad :: B.ByteString -> B.ByteString
27 shapad input = fill $ input `B.snoc` chr 0x80
28         where
29                 -- 56bytes are 448bits
30                 fill unfilled
31                         | lenmod == 56 = unfilled
32                         | otherwise = unfilled `B.append` B.replicate
                               remaining '\0'
33                 lenmod = (B.length input) + 1 `mod` 64 -- +1 because we
                       already added a byte
34                 --120 because 62 `mod` 64 results in -2, which we cant
                       use for replicate
35                 remaining = (120 - lenmod) `mod` 64
36
37 -- adds padding and size, output length will always be a multiple of 64
       bytes
38 preprocess :: B.ByteString -> B.ByteString
39 preprocess input = shapad input `B.append` lenAsBStr
40         where
41                 len = 8 * B.length input --in bits
42                 lenAsBStr = B.pack
43                         ['\NUL','\NUL','\NUL','\NUL'
44                         ,chr $ shiftR (len .&. 0xFF000000) 24
45                         ,chr $ shiftR (len .&. 0x00FF0000) 16
46                         ,chr $ shiftR (len .&. 0x0000FF00) 8
47                         ,chr $ len .&. 0x000000FF
48                         ]
49
50 -- prepares a chunk, executes mainloop and adds the result to the hash
       so far
51 perchunk :: [Word32] -> B.ByteString -> [Word32]
52 perchunk curhash chunk = zipWith (+) curhash looped
```

```
53          where
54                  broken = map b2w $ block 4 chunk
55                  expanded = expandwords broken
56                  looped = mainloop 0 expanded curhash
57
58  -- executes 64 SHA2-Rounds on a chunk
59  mainloop :: Int -> [Word32] -> [Word32] -> [Word32]
60  mainloop 64 _ h = h
61  mainloop i w [a,b,c,d,e,f,g,h] = mainloop (i+1) w
        [temp2,a,b,c,newd,e,f,g]
62          where
63                  s1 = (e `rotateR` 6) `xor` (e `rotateR` 11) `xor` (e
                        `rotateR` 25)
64                  ch = (e .&. f) `xor` ((complement e) .&. g)
65                  temp = h + s1 + ch + ks!!i + w!!i
66                  newd = d + temp;
67                  s0 = (a `rotateR` 2) `xor` (a `rotateR` 13) `xor` (a
                        `rotateR` 22)
68                  maj = (a .&. (b `xor` c)) `xor` (b .&. c)
69                  temp2 = temp + s0 + maj
70
71  -- expands the 16 Word32s to 64 Word32s, according to SHA256 spec
72  expandwords :: [Word32] -> [Word32]
73  expandwords cw
74          | length cw == 64 = cw
75          | otherwise = expandwords $ cw ++ [newword cw]
76
77  -- creates the next Word for the expansion
78  newword :: [Word32] -> Word32
79  newword cw = cw!!(i-16) + s0 + cw!!(i-7) + s1
80          where
81                  i = length cw
82                  s0 = (cw!!(i-15) `rotateR` 7) `xor` (cw!!(i-15)
                        `rotateR` 18) `xor` (cw!!(i-15) `shiftR` 3)
83                  s1 = (cw!!(i-2) `rotateR` 17) `xor` (cw!!(i-2) `rotateR`
                        19) `xor` (cw!!(i-2) `shiftR` 10)
84
85  -- converts 4 Bytes to a Word32
86  b2w :: B.ByteString -> Word32
87  b2w = fromInteger . asInt
88
89  w2b :: Word32 -> B.ByteString
90  w2b = asStr . toInteger
91
92  -- break input into 512bit blocks
93  chunks :: B.ByteString -> [B.ByteString]
94  chunks = block 64
95
96  {---
97  Data
98  ---}
99  -- initial hash
100 hs :: [Word32]
101 hs =
```

```
102         [0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f,
               0x9b05688c, 0x1f83d9ab, 0x5be0cd19]
103
104 -- round constants, in every iteration of the innermost
105 -- loop (mainloop), one of these values is used
106 ks :: [Word32]
107 ks =
108         [0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
               0x59f111f1, 0x923f82a4, 0xab1c5ed5
109         ,0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74,
               0x80deb1fe, 0x9bdc06a7, 0xc19bf174
110         ,0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
               0x4a7484aa, 0x5cb0a9dc, 0x76f988da
111         ,0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
               0xd5a79147, 0x06ca6351, 0x14292967
112         ,0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354,
               0x766a0abb, 0x81c2c92e, 0x92722c85
113         ,0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
               0xd6990624, 0xf40e3585, 0x106aa070
114         ,0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3,
               0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3
115         ,0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa,
               0xa4506ceb, 0xbef9a3f7, 0xc67178f2
116         ]
117
118 {-------------------------------------
119 some tests
120 data from wikipedia and manual execution
121 of echo -ne "input" | sha256sum on linux
122 -------------------------------------}
123
124 -- printf "%08x" wandelt einen Int in die Hexdarstellung um
125 -- ueblicherweise werden hashes in Hex ausgegeben
126 hex :: B.ByteString -> B.ByteString
127 hex = B.pack . printf "%08x" . asInt
128
129 -- true if no tests failed
130 runtests :: Bool
131 runtests = and [testHash test | test <- tests]
132
133 testHash :: (B.ByteString,B.ByteString) -> Bool
134 testHash (input,expected) = (hex $ hash input) == expected
135
136 tests =
       [("","e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855")
137                 ,("\0","6e340b9cffb37a989ca544e6bb780a2c78901d3fb33738768511a30617afa0
138                 ,("a","ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48b
139                 ,("hallo","d3751d33f9cd5049c4af2b462735457e4d3baf130bcbb87f389e349fbae
140                 -- 55 bytes stay in one chunk
141                 ,(B.replicate 55
                        'a',"9f4390f8d30c2dd92ec9f095b65e2b9ae9b0a925a5258e241c9f1e910f734
142                 -- 56 bytes and more cause perchunk to be called at
                        least two times
143                 ,(B.replicate 56
                        'a',"b35439a4ac6f0948b6d6f9e3c6af0f5f590ce20f1bde7090ef7970686ec67
```

```
144                     ,(B.replicate 57
                            'b',"2dd0a6d14520f410e18bd2f443f0ff2e7389dfaf9242bb9257730fc190e808
145                     ,("Franz jagt im komplett verwahrlosten Taxi quer durch
                            Bayern",
146                             "d32b568cd1b96d459e7291ebf4b25d007f275c9f13149beeb782fac07166
147                     ,("Frank jagt im komplett verwahrlosten Taxi quer durch
                            Bayern",
148                             "78206a866dbb2bf017d8e34274aed01a8ce405b69d45db30bafa00f5eeed7
149                     ,(B.replicate 120 'a' `B.append` B.replicate 1000 'Z',
150                             "f44da844b446469e8a3c928e6f696b3994e404b1388282267e932744bbc74
151                     ]
```

4.4 BlockModes.hs

```
1  module Kpspcrypto.BlockModes where
2
3  -- needed for using string-literals with ByteString
4  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5  {-# LANGUAGE OverloadedStrings #-}
6
7  import qualified Data.ByteString.Char8 as B
8  import Data.Bits
9
10 import Kpspcrypto.Pad
11 import Kpspcrypto.Serial
12
13 type Block = B.ByteString
14 type IV = Block
15
16 cbc :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
17 cbc cipher iv plain = B.concat $ docbc cipher iv $ pad blocklen plain
18         where
19                 blocklen = B.length iv
20
21 docbc :: (Block -> Block) -> IV -> [Block] -> [Block]
22 docbc _ _ [] = []
23 docbc cipher iv (x:xs) = cblock : docbc cipher cblock xs
24         where
25                 ivi = asInt iv
26                 xi = asInt x
27                 ivxorb = asStr $ ivi `xor` xi
28                 cblock = cipher ivxorb
29
30 uncbc :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
31 uncbc cipher iv crypt = unpadblocks $ douncbc cipher iv $ block blocklen
       crypt
32         where
33                 blocklen = B.length iv
34
35 douncbc :: (Block -> Block) -> IV -> [Block] -> [Block]
36 douncbc _ _ [] = []
37 douncbc cipher iv (x:xs) = plain : douncbc cipher x xs
38         where
39                 ivi = asInt iv
```

```
40                  xdec = asInt $ cipher x
41                  plain = asStr $ ivi `xor` xdec
42
43  ecb :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
44  ecb cipher iv plain = B.concat [cipher pblock | pblock <- pblocks]
45          where
46                  blocklen = B.length iv
47                  pblocks = pad blocklen plain
48
49  unecb :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
50  unecb cipher iv crypt = unpadblocks [cipher cblock | cblock <- cblocks]
51          where
52                  blocklen = B.length iv
53                  cblocks = block blocklen crypt
```

4.5 Asymmetrische Verschlüsselung

4.5.1 RSA.hs

```
1   module Kpspcrypto.RSA (encrypt, sign, decrypt, checksig) where
2
3   -- needed for using string-literals with ByteString
4   -- see
        http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5   {-# LANGUAGE OverloadedStrings #-}
6
7   import qualified Data.ByteString.Char8 as B
8   import qualified Kpspcrypto.Base64 as B64
9   import Kpspcrypto.Serial
10
11  type KeyFileContent = B.ByteString
12  -- first part is e or d, second is n
13  type Pubkey = (B.ByteString, B.ByteString)
14  type Privkey = Pubkey
15
16
17  {-----------------
18  public functions
19  -----------------}
20  encrypt :: KeyFileContent -> B.ByteString -> B.ByteString
21  encrypt key msgIn = asStr $ modexp msg e n
22          where
23                  e = toInt eIn
24                  n = toInt nIn
25                  msg = asInt msgIn
26                  (eIn,nIn) = fromFile key
27
28  sign :: KeyFileContent -> B.ByteString -> B.ByteString
29  sign = encrypt
30
31  decrypt :: KeyFileContent -> B.ByteString -> B.ByteString
32  decrypt = encrypt
33
34  checksig :: KeyFileContent -> B.ByteString -> B.ByteString -> Bool
```

```
35 checksig pubkey sig msg = encrypt pubkey sig == msg
36
37 {-----------------
38 helper functions
39 -----------------}
40 -- modexp b e n returns b^e mod n
41 -- slightly modified from http://pastebin.com/m142c0ca
42 modexp :: Integer -> Integer -> Integer -> Integer
43 modexp b 0 n = 1
44 modexp b e n
45         | even e = (modexp b (e `div` 2) n) ^ 2 `mod` n
46         | otherwise = (b * modexp b (e-1) n) `mod` n
47
48 {-------------------------------
49 less related utility functions
50 -------------------------------}
51 -- converts an Integer to Base64 encoded ByteString
52 toStr :: Integer -> B.ByteString
53 toStr = B64.encode . asStr
54
55 -- reads a Base64 encoded Integer from a ByteString
56 toInt :: B.ByteString -> Integer
57 toInt = asInt . B64.decode
58
59 -- extracts key from file
60 fromFile :: B.ByteString -> Pubkey
61 fromFile file = (e,n)
62         where
63                 contentline = B.lines file !! 1
64                 [e,n] = B.split ',' contentline
65
66 {-------------------------------------------------
67 sample data (from
68 http://de.wikipedia.org/wiki/RSA-Kryptosystem)
69 -------------------------------------------------}
70 rsapubkey = "-----BEGIN RSA PUBLIC KEY-----\nFw==,jw==\n-----END RSA PUBLIC
       KEY-----" :: B.ByteString
71 rsaprivkey = "-----BEGIN RSA PRIVATE KEY-----\nLw==,jw==\n-----END RSA
       PRIVATE KEY-----" :: B.ByteString
72 rsarecvpubkey = "-----BEGIN RSA PUBLIC KEY-----\nBrk=,BAYh\n-----END RSA
       PUBLIC KEY-----" :: B.ByteString
73 rsarecvprivkey = "-----BEGIN RSA PRIVATE KEY-----\nBV0=,BAYh\n-----END RSA
       PRIVATE KEY-----" :: B.ByteString
74
75 exmsg = toStr 7
76
77 expub = (toStr 23, toStr 143)
78 expriv = (toStr 47, toStr 143)
79
80 expub2 = (toStr 1721, toStr 263713)
81 expriv2 = (toStr 1373, toStr 263713)
82
83 smalltest = [checksig rsapubkey (sign rsaprivkey str) str | str <- map
       B.singleton ['\0'..'\140']]
```

```
84 smalltest2 = [checksig rsarecvpubkey (sign rsarecvprivkey str) str | str
     <- map B.singleton ['\0'..'\255']]
```

4.5.2 RSAKey.hs

```
 1 module Kpspcrypto.RSAKey (genK) where
 2
 3 import qualified Data.ByteString.Char8 as B
 4 import System.Random
 5 import Kpspcrypto.Serial
 6 import qualified Kpspcrypto.Base64 as B64
 7
 8
 9 type P = Integer
10 type Q = Integer
11 type Privkey = B.ByteString
12 type Pubkey = B.ByteString
13
14 genK :: StdGen -> (Privkey, Pubkey)
15 genK rgen = (genPrivK rgen, genPubK rgen)
16
17 genPrivK :: StdGen -> Privkey
18 genPrivK rgen = begin `B.append` toStr (getD $ genKeys rgen) `B.append`
     "," `B.append` toStr (getN $ genKeys rgen) `B.append` end
19                 where
20                         begin = "-----BEGIN RSA PRIVATE KEY-----\n"
21                         end = "\n-----END RSA PRIVATE KEY-----"
22                         getN :: (Integer, Integer) -> Integer
23                         getN (_, n) = n          --oder getN = snd
24                         getD :: (Integer, Integer) -> Integer
25                         getD (d, _) = d          --oder getD = fst
26
27 genPubK :: StdGen -> Pubkey
28 genPubK rgen = begin `B.append` toStr 65537 `B.append` "," `B.append`
     toStr (getN $ genKeys rgen) `B.append` end
29                 where
30                         begin = "-----BEGIN RSA PUBLIC KEY-----\n"
31                         end = "\n-----END RSA PUBLIC KEY-----"
32                         getN :: (Integer, Integer) -> Integer
33                         getN = snd
34
35
36 genKeys :: StdGen -> (Integer, Integer)
37 genKeys rgen = (d, n)
38                 where
39                         p = head $ genPrime getP
40                         q = head $ genPrime getQ
41                         (getP, newGen) = (randomR (2^34, 2^35-1) rgen)
                              -- different range for p and q to ensure p!=q
42                         (getQ, newGen') = (randomR (2^35, 2^37) newGen)
43                         d = genD p q
44                         n = p*q
45
46
47 --calculate d (decoding) e * d = 1 mod phi(N)
```

```
48 genD :: P -> Q -> Integer
49 genD p q = if scnd (extendedEuclid 65537 phiN) > 0 then scnd
       (extendedEuclid 65537 phiN) else phiN + scnd (extendedEuclid 65537
       phiN)
50           where
51                   phiN = (p-1)*(q-1)
52                   scnd (_, x, _) = x
53
54 --helper functions
55 extendedEuclid :: Integer -> Integer -> (Integer, Integer, Integer)
56 extendedEuclid a b
57                                   | b == 0 = (a, 1, 0)
58                                   | otherwise = (d,t,s - (div a b)*t)
59                                         where
60                                                 (d,s,t) = extendedEuclid
                                                      b (a `mod` b)
61
62 --first version with pattern matching -> chose guards to do sth
       different for once ;)
63 --extendedEuclid a 0 = b == 0 = (a, 1, 0)
64 --extendedEuclid a b = (d,t,s - (div a b)*t)
65 --                                    where
66 --                                            (d,s,t) = extendedEuclid
       b (a `mod` b)
67
68 --create a list containing primes to get random p and q
69 genPrime :: Integer -> [Integer]
70 genPrime n = if even n then genPrime (n+1) else [x | x <- [n,n+2..],
       isPrime x]
71
72
73
74 isPrime :: Integer -> Bool
75 isPrime x = null [y | y <- takeWhile (\y -> y*y <= x) [2..], x `mod` y
       == 0]
76
77
78 -- converts an Integer to Base64 encoded ByteString
79 toStr :: Integer -> B.ByteString
80 toStr = B64.encode . asStr
```

4.6 Base64.hs

```
1  module Kpspcrypto.Base64 (encode, decode) where
2
3  -- http://www.haskell.org/haskellwiki/DealingWithBinaryData
4  -- http://en.wikipedia.org/wiki/Base64
5
6  -- needed for using string-literals with ByteString
7  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
8  {-# LANGUAGE OverloadedStrings #-}
9
10 import qualified Data.ByteString.Char8 as B
11 import qualified Data.Vector as V
```

```
12 import qualified Data.Map as M
13 import Data.Bits
14 import Data.Maybe
15 import Data.Char
16
17 {----------------
18 public functions
19 ----------------}
20 -- encode a given ByteString using Base64
21 -- the output length will be a multiple of 4
22 encode :: B.ByteString -> B.ByteString
23 encode input = encodeR $ addpad input
24
25 -- decode Base64 encoded content of a ByteString
26 -- input length must be a multiple of 4
27 decode :: B.ByteString -> B.ByteString
28 decode encoded = B.take resultlen decWithTrash
29         where
30                 (unpadded, padlen) = unpad encoded
31                 decWithTrash = decodeR unpadded
32                 resultlen = B.length decWithTrash - padlen
33
34 {----------------
35 encoding helpers
36 ----------------}
37 -- recursively substitute 3 bytes with the 4 bytes
38 -- that result from Base64-encoding
39 -- the padding length is required for marking the
40 -- amount of added padding in the final output
41 encodeR :: (B.ByteString, Int) -> B.ByteString
42 encodeR ("",_) = ""
43 encodeR (x,padlen)
44         | B.length x /= 3 || padlen == 0 = subs next `B.append` encodeR
               (rest, padlen)
45         -- otherwise: last 3 bytes and we have padding
46         | otherwise =
47                 if padlen == 1 then B.init (subs x) `B.append` "="
48                 else B.take 2 (subs x) `B.append` "=="
49         where
50                 (next,rest) = B.splitAt 3 x
51                 -- convert to 6-bit-values, find the corresponding char
                       and
52                 -- convert the [Char] to a ByteString
53                 subs input = B.pack $ map (table V.!) (toB64BitGroups
                       input)
54
55 -- splits a ByteString (with length 3) into four 6-bit values
56 toB64BitGroups :: B.ByteString -> [Int]
57 toB64BitGroups x = [
58         shiftR (ord (B.index x 0)) 2,
59         shiftL (ord (B.index x 0) .&. 3) 4 .|. shiftR (ord (B.index x
               1)) 4,
60         shiftL (ord (B.index x 1) .&. 15) 2 .|. shiftR (ord (B.index x
               2)) 6,
61         ord (B.index x 2) .&. 63]
```

```
62
63 -- expands an input to a multiple of 3 Bytes for processing
64 -- second element of tuple is the length of the added padding
65 addpad :: B.ByteString -> (B.ByteString, Int)
66 addpad x = (x `B.append` B.replicate padlen '\0', padlen)
67         where
68                 len = B.length x
69                 padlen = (3 - len `mod` 3) `mod` 3
70
71 -- base64 index table
72 -- contains the allowed characters in the Base64 output
73 table :: V.Vector Char
74 table = V.fromListN 64 (['A'..'Z'] ++ ['a'..'z'] ++ ['0'..'9'] ++
       ['+','/'])
75
76 {----------------
77 decoding helpers
78 ----------------}
79 -- combines 4 6-bit values into a ByteString with length 3
80 fromB64BitGroups :: [Int] -> B.ByteString
81 fromB64BitGroups x = B.pack $ map chr ords
82         where
83                 ords = [
84                         shiftL (x !! 0) 2 .|. shiftR (x !! 1) 4,
85                         shiftL ((x !! 1) .&. 15) 4 .|. shiftR (x !! 2) 2,
86                         shiftL ((x !! 2) .&. 3) 6 .|. (x !! 3)]
87
88 -- removes the padding from a Base64-encoded input
89 -- second element in the returned tuple is the amount
90 -- of bytes to be discarded after decoding
91 unpad :: B.ByteString -> (B.ByteString, Int)
92 unpad x = (
93         B.takeWhile (/= '=') x `B.append` B.replicate padlen '0',
94         padlen )
95         where
96                 padlen = B.length (B.dropWhile (/= '=') x)
97
98 -- recursively substitute 4 "Base64-Bytes" with 3 Bytes from
99 -- the plaintext which was encoded
100 decodeR :: B.ByteString -> B.ByteString
101 decodeR "" = ""
102 decodeR x = subs next `B.append` decodeR rest
103         where
104                 (next, rest) = B.splitAt 4 x
105                 -- convert ByteString to [Char], restore the original 4
106                 -- 6-bit-values and convert them to the original 3 Bytes
107                 subs input = fromB64BitGroups [fromJust (M.lookup c
                        tableR) | c <- B.unpack input]
108
109 -- contains the 6bit-values for the allowed chars in Base64 encoded data
110 tableR :: M.Map Char Int
111 tableR = M.fromList [(table V.! v ,v) | v <- [0..63]]
112
113 {----
114 tests
```

```
115 -----}
116 runTests :: Bool
117 runTests = and [str == (decode . encode $ str) | str <- teststrings]
118
119 -- returns some strings of various lengths
120 teststrings = map B.pack [replicate i (chr $ i+j+k) ++ replicate j (chr
        $ 7*i+9*j) ++ replicate k (chr $ 11*i+3*k+2*j) |
121                   i <- [0..10], j <- [0..10], k <- [0..10]]
```

4.7 Hilfsfiles

4.7.1 Pad.hs

```
1  module Kpspcrypto.Pad (pad, unpad, unpadblocks, block) where
2
3  -- needed for using string-literals with ByteString
4  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5  {-# LANGUAGE OverloadedStrings #-}
6
7  import qualified Data.ByteString.Char8 as B
8  import Data.Char
9
10 -- separate input into blocks and (always!) add padding
11 -- if input length mod blocklength == 0 we add a block that
12 -- only contains padding
13 pad :: Int -> B.ByteString -> [B.ByteString]
14 pad n input
15         -- not the last block, recurse on following blocks
16         | B.length input >= n = next : pad n rest
17         -- last block: add padding
18         | otherwise = [input `B.append` (B.replicate padlen padchar)]
19         where
20                 (next,rest) = B.splitAt n input
21                 padlen = n - B.length input
22                 padchar = chr padlen
23
24 -- split input into block of given length
25 block :: Int -> B.ByteString -> [B.ByteString]
26 block n "" = []
27 block n x = next : (block n rest)
28         where
29                 (next,rest) = B.splitAt n x
30
31 -- undo "pad n", uses "unpadblocks"
32 unpad :: Int -> B.ByteString -> B.ByteString
33 unpad n input = unpadblocks $ block n input
34
35 -- remove the padding in the last block
36 unpadblocks :: [B.ByteString] -> B.ByteString
37 -- last block: at least one byte is padding, because
38 -- pad always adds a padding
39 unpadblocks [x] = B.take padlen x
40         where
```

```
41                  n = B.length x
42                  padlen = n - (ord $ B.last x)
43 -- not the last block: recursion on following blocks
44 unpadblocks (x:xs) = x `B.append` unpadblocks xs
45
46 {-----
47 tests
48 -----}
49 runTests :: Bool
50 runTests = and [(unpad len $ B.concat $ pad len s) == s | s <-
       testInputs, len <- [1..30]]
51
52 testInputs :: [B.ByteString]
53 testInputs = [B.replicate i 'a' | i <- [1..200]]
```

4.7.2 Msg.hs

```
1  module Kpspcrypto.Msg where
2
3  -- needed for using string-literals with ByteString
4  -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5  {-# LANGUAGE OverloadedStrings #-}
6
7  import qualified Data.ByteString.Char8 as B
8  import Text.Regex.Posix
9
10 type AsymCipher = B.ByteString -- z.B. "RSA"
11 type AsymKey = B.ByteString -- kompletter inhalt des pub- oder
       privatekeyfiles
12 type HashType = B.ByteString -- z.B. "SHA256"
13
14 sampleMsgStr =  "-----BEGIN KEYCRYPTED RSA 8----\nbla//+ba\n-----END
       KEYCRYPTED----\n\n" `B.append`
15                                  "-----BEGIN MSGCRYPTED AES256
                                        CBC----\nbla+bl/ubb\n-----END
                                        MSGCRYPTED----\n\n" `B.append`
16                                  "-----BEGIN SIGNATURE SHA256 RSA
                                        8----\nbl/+ubb+/+lubb\n-----END
                                        SIGNATURE----\n\n"
17
18 -- possible types of messageparts
19 data MsgType = KEYCRYPTED | MSGCRYPTED | SIGNATURE deriving (Show, Read,
       Eq, Ord)
20
21 -- this can hold any type of messagepart
22 data MsgPart = MsgPart  {       msgtype :: MsgType
23                                                 ,       options ::
                                                        [B.ByteString]
24                                                 ,       content ::
                                                        B.ByteString
25                                                 } deriving (Read, Eq)
26
27 makeMsg :: (MsgType, [B.ByteString]) -> B.ByteString -> MsgPart
28 makeMsg (msgtype, options) content = MsgPart msgtype options content
```

```
29
30 -- make the msg print in the way we want and expect it in a file
31 instance Show MsgPart where
32         show msg = "----BEGIN " ++ show (msgtype msg) ++ " " ++
33                 B.unpack (B.intercalate " " (options msg)) ++
34                 "----\n" ++ B.unpack (content msg) ++ "\n" ++
35                 "----END " ++ show (msgtype msg) ++ "----"
36 -- alternative for intercalate: foldl (\acc option -> acc `B.append` " "
       `B.append` option) "" (options msg)
37
38 -- make the msg sortable by type
39 instance Ord MsgPart where
40         compare a b = compare (msgtype a) (msgtype b)
41
42 -- interprets the first line of a msgpart
43 readHdr :: B.ByteString -> (MsgType, [B.ByteString])
44 readHdr hdr = (msgtype, msgoptions)
45         where
46                 -- drop 10 drops "----BEGIN "
47                 contents = B.words . B.takeWhile (/= '-') . B.drop 10
48                 msgtype = read . B.unpack . head $ contents hdr
49                 msgoptions = tail $ contents hdr
50
51 -- interpret a ByteString as MsgPart
52 -- the input has to be in the right form
53 readMsg :: B.ByteString -> MsgPart
54 readMsg input = makeMsg (readHdr headerLine) content
55         where
56                 headerLine = head $ B.lines input
57                 -- outermost 'init' is for removing the trailing \n
                       added by unlines
58                 content = B.init . B.unlines . init . tail $ B.lines
                       input
59
60 -- gets messageparts from a file
61 --
       http://stackoverflow.com/questions/7636447/raise-no-instance-for-regexcontext-rege
62 getMsgParts :: B.ByteString -> [MsgPart]
63 getMsgParts input = map readMsg regmatches
64         where
65                 regmatches = getAllTextMatches (input =~ regex ::
                       AllTextMatches [] B.ByteString)
66                 regex = "----BEGIN [A-Z0-9
                       ]+----\n[a-zA-Z0-9+/=,]+\n----END [A-Z0-9]+----" ::
                       B.ByteString
```

4.7.3 Serial.hs

```
1 module Kpspcrypto.Serial where
2
3 -- needed for using string-literals with ByteString
4 -- see
       http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-Byte
5 {-# LANGUAGE OverloadedStrings #-}
6
```

```
7  import qualified Data.ByteString.Char8 as B
8  import Data.Char
9
10 -- interpret the bytes of a ByteString as
11 -- Integer, result will always be positive
12 asInt :: B.ByteString -> Integer
13 asInt str = f . reverse $ B.unpack str
14         where
15                 f "" = 0
16                 f (x:xs) = toInteger (ord x) + 256 * f xs
17
18 -- convert a positive Integer to a ByteString
19 asStr :: Integer -> B.ByteString
20 -- edge case here because we dont want 0 to become "", but
21 -- we need a ""-edge-case in the helper function
22 asStr 0 = B.singleton '\0'
23 asStr i = B.pack . reverse $ f i
24         where
25                 f 0 = []
26                 f i = chr (fromInteger (i `mod` 256)) : f (i `div` 256)
```

5 Testfälle

5.1 Key Generator

Der Keygenerator soll zufällige RSA-Keys erzeugen. Wir können ihn nach der Kompilation mittles *ghc -XOverloadedStrings hskeygenerator.hs* aufrufen, um Keys für unseren Sender und den Empfänger zu generieren:

```
1  [iso@iso-t530arch tmp]$ ./hskeygenerator
2  Enter filename:
3  sender
4  [iso@iso-t530arch tmp]$ ./hskeygenerator
5  Enter filename:
6  receiver
```

Die Inhalte der erzeugten Schlüsselfiles sehen dann wie folgt aus:

```
1  [iso@iso-t530arch tmp]$ cat senderRsaPubKey
2  -----BEGIN RSA PUBLIC KEY-----
3  AQAB,uLEhNs/uRmG9
4  -----END RSA PUBLIC KEY-----
5
6  iso@iso-t530arch tmp]$ cat senderRsaPrivKey
7  -----BEGIN RSA PRIVATE KEY-----
8  Ia9kwGVnj+Bh,uLEhNs/uRmG9
9  -----END RSA PRIVATE KEY-----
10
11 [iso@iso-t530arch tmp]$ cat receiverRsaPubKey
12 -----BEGIN RSA PUBLIC KEY-----
13 AQAB,OSD0b/N9Btp5
14 -----END RSA PUBLIC KEY-----
15
16 [iso@iso-t530arch tmp]$ cat receiverRsaPrivKey
```

```
17 -----BEGIN RSA PRIVATE KEY-----
18 EbzFqCfx729x,OSD0b/N9Btp5
19 -----END RSA PRIVATE KEY-----
```

Wie erwünscht enthalten die jeweils zueinander gehörigen Private- und Public-Keys den selben Wert für N („uLEhNs/uRmG9“ bei sender, „OSD0b/N9Btp5“ bei receiver). Bei beiden Schlüsselpaaren wird für e der Wert 65537 verwendet, d hingegen unterscheidet sich zwischen den Schlüsselpaaren.

In diesem Fall wurden die folgenden Schlüssel erzeugt. Die Werte können mittels folgendem Befehl (nach dem Laden von hsencrypt.hs) zurückgewonnen werden:

```
1 Kpspcrypto.Serial.asInt $ Kpspcrypto.Base64.decode "Ia9kwGVnj+Bh"
```

- Sender e: 65537, d: 621380992428485369953, n: 3406964452648185913789
- Receiver e: 65537, d: 327197112392121020273, n: 1053839058196540873337

5.2 Ver- und Entschlüsselung von Nachrichten

5.2.1 Vorbereitung

Mit den folgenden Befehlen werden (unter Linux) zufällige Testnachrichten erstellt und deren Hashwerte (für den späteren Vergleich) ermittelt:

```
1  [iso@iso-t530arch tmp]$ dd bs=1k count=1 if=/dev/urandom of=1k.msg
2  1+0 records in
3  1+0 records out
4  1024 bytes (1.0 kB) copied, 0.000435418 s, 2.4 MB/s
5  [iso@iso-t530arch tmp]$ dd bs=1k count=100 if=/dev/urandom of=100k.msg
6  100+0 records in
7  100+0 records out
8  102400 bytes (102 kB) copied, 0.0180468 s, 5.7 MB/s
9  [iso@iso-t530arch tmp]$ sha256sum 1k.msg 100k.msg
10 fd6df86538db0013a9f943b2d8a03d52a5d6a40cbe3243408167dc15e29a855d  1k.msg
11 1b13213582917f5b36751ab66bf22ae19233e259de319181acbeb64d712a3559
       100k.msg
```

5.2.2 Verschlüsselung ECB

Für diese Tests werden die zuvor erzeugten Schlüssel verwendet. Wir verschlüsseln die beiden Nachrichten mit den öffentlichen Schlüssel des Empfängers und signieren den Inhalt der Nachricht mit Hilfe des privaten Schlüssels des Senders:

```
1 [iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 ECB
      senderRsaPrivKey receiverRsaPubKey 1k.msg
2 [iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 ECB
      senderRsaPrivKey receiverRsaPubKey 100k.msg
```

Den Inhalt des 1k-Files schauen wir uns an (Zeilenumbrüche innerhalb der Msg-Teile wurden manuell hinzugefügt):

```
[iso@iso-t530arch tmp]$ cat 1k.msgEncrypted
-----BEGIN KEYCRYPTED RSA-----
KB25vJ46fR06,ISPN+nHCYyt3,AdvzIdRlMOpz,LFpPe8rS7WCM,NOgghBG7w74E,
Li21sI2vAGBH,GfDWpqTKuuP+,KgFI/+tsr+NX
-----END KEYCRYPTED-----

-----BEGIN MSGCRYPTED AES256 ECB-----
NC5ja4U25ZMXVRQAFlYn0aIJcr26nx+gEYlARbw52QHDoQnywSEZRTFXuQ5K4kSvx1AiW
L+s719SBx0OGi/o+JOKp+cutp4ArUSHK7aWdBVrJpTi6a0Bj0fyKr5xsAVGZAfC3XCx14
li16+05/p+fOwN8LzixmqQ2R0T49n+iI7n/9Uw1wu1UbYPfjgBIeXT8HGHc+GevYAkrXv
QHXVdbFaBZXQBQWWcfr+A0rbfOkxG2bDh5FwR7WF+7PFDK7h2peXSFJ/nFu4SSLBVbEED
PFbbhG/fS+IcQ4y5Mu5/ICfc2WeZg8r83cRhu2nDJouOzQXM8qxvLrpY0IZ5xhbB2b0mT
vg01KaD9mp4UrtxDvsqLs9kwuGgMKruqKolM3C49zx3uhBSb0T4uF/2hgowPuhrN0Xppd
ytMuMvstvJGImPEuj+CAFJ6GbFbBQj5xwlvsgx3tsYiCzTe6A62m0yuATioFDAuGB7A6a
tdgDLiNyfV3oNFGuBukIe1UAZyz5xDWyfCbR0bG8Ok+38oXqMzRrdyv/zhtwZaugnhnLa
H/Eu8H3AmqoMz6hVp6xDtX72HcYu/FXOaRtFZsH6PWEPJdu02uCGQ7G/1dUM2frG6SPSj
lfRxPpUmQTkoDOtk51t/nv2BQUqcwYuDHWHzL4wIw/+wAY1xe5LU/WiPd9/3Galv4saan
wDsrpFqvbgMdPkOW6wNHBu4Vl8KG4TPRpymwYKvwhg2hIU01RT+zJ+ezVsgrl74cNj5DP
3NrlNdiKhRVmd7oAJPHde2g6ZhdB1YrcAhSsiVOq118UiKpbn/LfobGUKOTa51/wollRs
riC4uU4ULyXix0C+WHLHdGd/xwaGmoBSBf0h+I/fUZ97xw6fgdN0oyfek75tdQpiPI18X
NHVwYd2qAdH/BTuM+ODuqjgPcuunUzJXfGJpQVDBPPQh6akzIyyHfQMBJN0N4o1jfUKL6
CFWaHmXRQpCnVkFwsKP5NlOYfpjsY7N34OmrqAOZP/wIBYs+HjcF4YxirE98iOcII5Rsf
O5i/wtQiyXgc/kFkY93uluwrZQT0OJKWzctH0isSIRPqGk3ySliB7Ceh7spqMIVPuPpn4
W63yFtf9XrzDtW8gMv/roX/0HJmeoW1ysjPjJ+teT6lZ50REmh4/LFGjkRQx2n1x+wflC
wwTusYA/1I/lzZRDvHhpljHGX1x8H9fYJekBBKYmPu6ufS0uEFS2UwaHFHalwIXuE2kVj
3BfiOp5rqgZZimVfoFKnFjo+V0d4SXuuqAVv+IFnPorG4nXThHf2TN5h6cn40xrlOfrM3
iRLB/CMCuvyvaV0luRLSKmYbrjDQr9ShFhLf+ER6Mp0eVxp8GfxnKdkCCpiHjrTM4eVLM
IoIMH1s=
-----END MSGCRYPTED-----

-----BEGIN SIGNATURE RSA SHA256-----
i7XYWSqqv8b7,P3ERQK9xnLYD,kMwA8TdQpUcb,OmazloPVsEod,YllxmJXn8mmf,
FmK2J2p4xgnI
-----END SIGNATURE-----
```

Im Header des MSGCRYPTED-Teils ist wie erwartet die ECB-Option gesetzt.

5.2.3 Entschlüsselung ECB

Die beiden Nachrichten können mit Hilfe des privaten Schlüssels des Empfängers und des öffentlichen Schlüssels des Senders entschlüsselt werden (dabei werden die in unserem Fall noch vorhandenen Original-Plaintext-Dateien überschrieben). Die SHA256-Summen der (neuen) Plaintext-Dateien entsprechenen denjenigen vor der Ver- und Entschlüsselung.

```
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey
    1k.msgEncrypted
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey
    100k.msgEncrypted
[iso@iso-t530arch tmp]$ sha256sum 1k.msg 100k.msg
fd6df86538db0013a9f943b2d8a03d52a5d6a40cbe3243408167dc15e29a855d  1k.msg
1b13213582917f5b36751ab66bf22ae19233e259de319181acbeb64d712a3559
    100k.msg
```

5.2.4 Verschlüsselung CBC

Die Nachrichten werden mit den selben Keys diesmal im CBC-Modus verschlüsselt und der Header des MSGCRYPTED-Teils überprüft:

```
1  [iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 CBC
       senderRsaPrivKey receiverRsaPubKey 1k.msg
2  [iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 CBC
       senderRsaPrivKey receiverRsaPubKey 100k.msg
3  [iso@iso-t530arch tmp]$ head -5 1k.msgEncrypted
4  -----BEGIN KEYCRYPTED RSA-----
5  DiUzRSdJA72E,EdXLBpnsii33,HL4nUeyNrdUn,BL/Saw/ZQls8,IDrIpuKmMjkJ,
6  Hrx7GSmFoNQV,NQOf4xGv0XMJ,KtodRCoiyeZp
7  -----END KEYCRYPTED-----
8
9  -----BEGIN MSGCRYPTED AES256 CBC-----
10 ....
```

5.2.5 Entschlüsselung CBC

Wiederum Entschlüsseln wir die Nachrichten mit den geeigneten Keys und kontrollieren die Hashwerte:

```
1  [iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey
       1k.msgEncrypted
2  [iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey
       100k.msgEncrypted
3  [iso@iso-t530arch tmp]$ sha256sum 1k.msg 100k.msg
4  fd6df86538db0013a9f943b2d8a03d52a5d6a40cbe3243408167dc15e29a855d  1k.msg
5  b99126267061a7ca4a63a0a59ec6e4b331d63e1dd852e2f4ba4fb72b98048a5d
       100k.msg
```

Die Hashwerte stimmen überein, woraus geschlossen werden kann, dass die Ver- und Entschlüsselung der Nachrichten keinen Informationsverlust zur Folge hat.

5.2.6 Versuch der Entschlüsselung von modifizierten Crypt-Files

In diesem Test wird in der CBC-verschlüsselten Datei 1k.msgEncrypted eine Anpassung innerhalb einer der drei Msg-Teilen vorgenommen und versucht, die Nachricht zu entschlüsseln:

```
1  [iso@iso-t530arch tmp]$ nano 1k.msgEncrypted
2  [iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey
       1k.msgEncrypted
3  signature or key was wrong, exiting...
```

Wir erhalten die erwartete Fehlermeldung und die Datei wurde nicht entschlüsselt.