In this report, we will see a toy implementation of the AES cipher in the Haskell programming language. This implementation differs from the normal AES in that it uses a smaller square grid of $3 \times 3$, instead of $4 \times 4$, a smaller field of $GF(2^4)$ instead of $GF(2^8)$, and has fewer rounds (5 instead of 10). The sources we have used are the textbook, the notes from class and the original AES proposal. We will thoroughly comment the code, describing design considerations.

First, we import the modules that are needed:

```
import List
import Array
import Bits
import Int
```

Since we use a different field than the original AES, we also need a different irreductible polynomial, to build it. The polynomial chosen is $x^4 + x + 1$. Note that the polynomial, and the elements of $GF(2^4)$ are represented by lists of booleans, where the least significant bit is the rightmost element of the list.

```
irreduct = [True, False, False, True, True]
```

Since we use lists of booleans to represent the elements of the field, we also need functions that convert 4 bit integers into these lists, and vice versa.

```
numToList n = reverse $ map (testBit (n::Int8)) [0..3]

listToNum :: [Bool] -> Int
listToNum l =
    foldr (\x y -> setBit y x) 0
      $ fst (unzip $ (filter (\x -> snd x == True) $ reverse $
      zip [0..] $ reverse l))
```

Addition of two elements in $GF(2^4)$ can be modelled as the exclusive or of the two bit strings representing the elements. Since this is the central operation of the cryptosystem, we declare a function that performs the exclusive or of two bit strings.

```
xorBits :: [Bool] -> [Bool] -> [Bool]
xorBits x y = reverse $ map (\x -> xor (fst x) (snd x)) $ zip (reverse x)
      (reverse y)
    where
    xor x y = (x || y) && ((not x) || (not y))
```

We also need a function that makes sure that a bit string is exactly 4 bits long.

```
make4 x
    | length x < 4 = (take (4 - length x) (repeat False)) ++ x
    | length x >= 4 = x
```

Now that the utility functions have been written, we can start implementing the different layers of the cryptosystem.
The first layer of AES is `byteSub`. This is a non-linear operation on a block of data, where each element in the block gets transformed to another byte, according to an S-box. This S-box is generated in the following way:
We start with a byte $x_3x_2x_1x_0$, where each $x_i$ is a binary bit. We then compute its inverse in $GF(2^4)$, to obtain another byte, $y_3y_2y_1y_0$. Note that 0000 does not have an inverse, so we must use 0000. We then do the following operation, to find the transformed value $z_3z_2z_1z_0$.

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix}$$

Since doing all this at run time would be quite expensive, we have pre-calculated the transformed value for every possible input. The results have been arranged in a table, for fast look-up: The first two bits of a number gives the row in the table, and the last two bits give the column.
The resulting table is:

$$sBox = \begin{bmatrix} 5 & 2 & 11 & 12 \\ 8 & 15 & 6 & 1 \\ 14 & 9 & 0 & 7 \\ 3 & 4 & 13 & 10 \end{bmatrix}$$

```
sBox :: Array (Int, Int) Int8
sBox = listArray ((0,0), (3,3)) [5, 2, 11, 12, 8, 15, 6, 1, 14, 9,
 0, 7, 3 ,4, 13, 10]
```

Since we also need to be able to decrypt, we have an inverse table:

$$invSBox = \begin{bmatrix} 10 & 7 & 1 & 12 \\ 13 & 0 & 6 & 11 \\ 4 & 9 & 15 & 2 \\ 3 & 14 & 8 & 5 \end{bmatrix}$$

```
invSBox :: Array (Int, Int) Int8
invSBox = listArray ((0,0), (3,3)) [10, 7, 1, 12, 13, 0, 6, 11, 4, 9, 15, 2,
   3, 14, 8, 5]
```

The `byteSub` function simply selects the transformed value from the input bit string, and table to use.

```
byteSub sbox l =
    map (byteSub' sbox) l
    where
    byteSub' sbox l = numToList (sbox ! (row l, col l))
    row l = listToNum $ take 2 l
    col l = listToNum $ drop 2 l
```

The next layer of the AES cryptosystem is `shiftRow`. In this layer, all that is done is shifting the rows of the matrix cyclically by 0, 1 and 2.

$$\begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} \\ c_{1,0} & c_{1,1} & c_{1,2} \\ c_{2,0} & c_{2,1} & c_{2,2} \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{pmatrix}$$

The inverse operation, `invShiftRow` is the same, except shifting right.

```
rotateLeft n l = let split = splitAt n l in
    (snd split) ++ (fst split)

rotateRight n l = rotateLeft (length l - n) l


shiftRow b = shiftRow' rotateLeft b
invShiftRow b = shiftRow' rotateRight b

shiftRow' f b =
    let split = splitAt 3 b in
shiftRow'' f 0 (fst split) (snd split)
    where
    shiftRow'' f n l [] = f n l
    shiftRow'' f n l rest = let split' = splitAt 3 rest in
 (f n l) ++  (shiftRow'' f (n + 1) (fst split')
      (snd split'))
```

The next layer of AES is `mixColumns`. In this layer, we multiply the block of data in $GF(2^4)$ by another matrix:

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & 2 \\ 2 & 1 & 1 \end{pmatrix} \begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} \\ c_{1,0} & c_{1,1} & c_{1,2} \\ c_{2,0} & c_{2,1} & c_{2,2} \end{pmatrix} = \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} \\ d_{1,0} & d_{1,1} & d_{1,2} \\ d_{2,0} & d_{2,1} & d_{2,2} \end{pmatrix}$$

This matrix was chosen because:

- It has to be invertible, since we need to be able to decrypt.
- If one input byte changes, 3 bytes change in in the output, since if we expand the operation, we see that

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} \\ d_{1,0} & d_{1,1} & d_{1,2} \\ d_{2,0} & d_{2,1} & d_{2,2} \end{pmatrix} = \begin{pmatrix} c_{0,0} + 2c_{1,0} + c_{2,0} & c_{0,1} + 2c_{1,1} + c_{2,1} & c_{0,2} + 2c_{1,2} + c_{2,2} \\ c_{0,0} + c_{1,0} + 2c_{2,0} & c_{0,1} + c_{1,1} + 2c_{2,1} & c_{0,2} + c_{1,2} + 2c_{2,2} \\ 2c_{0,0} + c_{1,0} + c_{2,0} & 2c_{0,1} + c_{1,1} + c_{2,1} & 2c_{0,2} + c_{1,2} + c_{2,2} \end{pmatrix}$$

```
mixColumns c =
    elems $ matMult mixMat cmat
    where
    mixMat = listArray ((1,1),(3,3)) (map numToList [1,2,1,1,1,2,2,1,1])
    cmat = listArray ((1,1), (3,3)) $ map make4 c

invMixColumns c =
    elems $ matMult invMixMat cmat
    where
    invMixMat = listArray ((1,1), (3,3)) (map numToList [7,7,9,9,7,7,7,9,7])
    cmat = listArray ((1,1), (3,3)) $ map make4 c
```

The code that performs multiplication in $GF(2^4)$ follows.

```
gMult x y =
    let s = dropWhile (not . ((==) True)) y in -- Remove leading 0s
case s of
[] -> [False, False, False, False]
[True] -> x
otherwise -> xorBits (make4 $ mult (length s - 1) x)
     (make4 $ gMult x $ tail s)
    where
```

To multiply a number by $x^n$, we simply shift it to the left $n$ times. If the result is greater or equal to $x^4$, we obtain the $(\bmod\ x^4+x+1)$, by `XOR`ing it with the irreductible polynomial.

```
    mult 0 x = x
    mult n x =
        let shifted = dropWhile (not . ((==) True)) $ shiftLeft 1 x in
            if (length shifted > 4) && (head shifted == True)
               then make4 $ mult (n-1) $ xorBits shifted irreduct
               else make4 $ mult (n-1) shifted
        where
        shiftLeft n l = l ++ (take n $ repeat False)
```

This is to perform matrix multiplication.

```
matMult x y = array ((1,1), (3,3))
              [((i, j), xorSum [gMult (x ! (i, k)) (y ! (k, j))
                                  | k <- range (lj,uj)])
               | i <- range (li,ui), j <- range (lj', uj')]
    where
    ((li,lj),(ui,uj)) = bounds x
    ((li',lj'),(ui',uj')) = bounds y
    xorSum = foldr xorBits $ take 4 $ repeat False
```

The last layer of AES is `roundKey`. In this round we simply `XOR` the block by the round key, which we'll discuss later.

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} \\ d_{1,0} & d_{1,1} & d_{1,2} \\ d_{2,0} & d_{2,1} & d_{2,2} \end{pmatrix} \bigoplus \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} \\ k_{1,0} & k_{1,1} & k_{1,2} \\ k_{2,0} & k_{2,1} & k_{2,2} \end{pmatrix} = \begin{pmatrix} e_{0,0} & e_{0,1} & e_{0,2} \\ e_{1,0} & e_{1,1} & e_{1,2} \\ e_{2,0} & e_{2,1} & e_{2,2} \end{pmatrix}$$

```
roundKey :: Int -> [[Bool]] -> [[Bool]] -> [[Bool]]
roundKey i k d = roundKey' d $ keyMat i k
    where
    keyMat i k = concat $ transpose $ [keySchedule (3*i) k] ++
 [keySchedule (3*i + 1) k] ++ [keySchedule (3*i + 2) k]
    roundKey' d k = map (\x -> xorBits (fst x) (snd x)) $ zip d k
```

The original key consists of $4 * 3 * 3 = 36$ bits. The key is arranged in a $3 \times 3$ matrix. This matrix is then expanded by adding 15 additional columns. We label each column $W(i)$. Then, if $i$ is not a multiple of 3,

$$W(i) = W(i-3) \oplus W(i-1)$$

If $i$ is a multiple of 3,

$$W(i) = W(i-3) \oplus T(W(i-1))$$

where $T(x)$ is the transformation of $x$ obtained as follows:
Let the elements of $x$ be $a, b, c$. We then rotate them, to obtain $b, c, a$. Then, we replace each of these bytes with the corresponding element in the S-box from `byteSub`, to get 3 bytes $e, f, g$. Then, $T(x)$ is the column vector

$$(e \oplus 0010^{(i-3)/3}, f, g)$$

The round key for the $i$th round is then the columns

$$W(3i), W(3i+1), W(3i+2)$$

```
keySchedule :: Int -> [[Bool]] -> [[Bool]]
keySchedule i l =
    if i < 3
       then take 3 $ drop (3*i) l
       else
       if mod i 3 == 0
    then map (\x -> xorBits (fst x) (snd x)) $
 zip (keySchedule (i-3) l) (trans i (keySchedule (i-1) l))
    else map (\x -> xorBits (fst x) (snd x)) $
 zip (keySchedule (i-3) l) (keySchedule (i-1) l)
    where
    trans i l' = [xorBits (head (sbox l')) (roundConst i)] ++  (tail (sbox l'))
    sbox l'' = byteSub sBox $ rotateLeft 1 l''
    roundConst 0 = [False, False, False, True]
    roundConst i = foldr (\x y -> x y) [True] $ take (1 + div (i - 3)  3) $
   repeat (gMult [False, False, True, False])
```

Now that we have seen what each layer of AES consists of, we can see how they are arranged to form a round.
Each round is made of `byteSub`, `shiftRow`, `mixColumn`, `roundKey`, applied in this order.
The last round does not have `mixColumn`.

```
aesRound i k a = roundKey i k $ mixColumns $ shiftRow $ byteSub sBox a
lastRound i k a = roundKey i k $ shiftRow $ byteSub sBox a
```

Therefore, to encrypt a block of data, we apply `roundKey`, with the 0th round key. Then we do 4 rounds, and finally, we do the last round. The resulting block is encrypted with the supplied key.

```
encrypt k a = map listToNum $ lastRound 5 k' $ aesRound 4 k' $ aesRound 3 k' $
      aesRound 2 k' $ aesRound 1 k' $ roundKey 0 k' a'
    where
    k' = map numToList k
    a' = map numToList a
```

To decrypt one round, we simply apply `invMixColumns`, `roundKey` with the $i$th key, `invShiftRow`, `invByteSub` The last round of the decryption does not involve `invMixColumns`

```
invAesRound i k a = invMixColumns $ roundKey i k $ invShiftRow $
    byteSub invSBox a
lastInvRound i k a = roundKey i k $ invShiftRow $ byteSub invSBox a
```

Therefore, to decrypt a block of encrypted data, we apply `roundKey` with the 5th round key, then do 4 rounds, and finally, we do the last round. The resulting block has been decrypted.

```
decrypt k a = map listToNum $ lastInvRound 0 k' $ invAesRound 1 k'
      $ invAesRound 2 k' $ invAesRound 3 k' $ invAesRound 4 k'
      $ roundKey 5 k' a'
    where
    k' = map numToList k
    a' = map numToList a
```

An example encryption would be to encrypt

$$P = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

with the key

$$K = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

The result should be

$$E = \begin{bmatrix} 14 & 5 & 13 \\ 8 & 14 & 14 \\ 6 & 9 & 10 \end{bmatrix}$$

When decrypting, the result should be the starting value.

The following snippet is the program, encrypting $A$ with the key $K$, and then decrypting the result, obtaining $A$ again.

```
encrypt [9,8,7,6,5,4,3,2,1] [1,2,3,4,5,6,7,8,9]
>>> [14,5,13,8,14,14,6,9,10]

decrypt [9,8,7,6,5,4,3,2,1] [14,5,13,8,14,14,6,9,10]
>>> [1,2,3,4,5,6,7,8,9]
```

We have seen the implementation in the Haskell programming language and design decision of this AES variant, that has a different field, square grid, and number of round than the original AES.