ECE 404 Homework 4

Elias Talcott

February 18, 2020

Contents

1 Theory Problems

1.1 Problem 1

Determine the following in GF(11):

(a) $(3x^4 + 5x^2 + 10) - (8x^4 + 5x^2 + 2x + 1)$

(b) $(5x^2 + 2x + 7) \cdot (5x^3 + 3x^2 + 3x + 2)$

(c) $\dfrac{x^5 + 8x^4 + x^3 + 4x^2 + 8x}{6x^3 + 3x^2 + 2}$

Solution

(a)

$$\begin{aligned}(3x^4 + 5x^2 + 10) - (8x^4 + 5x^2 + 2x + 1) &= -5x^4 - 2x + 9 \\ &= (11-5)x^4 + (11-2)x + 9 \\ &= 6x^4 + 9x + 9\end{aligned}$$

$(3x^4 + 5x^2 + 10) - (8x^4 + 5x^2 + 2x + 1)$ is equivalent to $\mathbf{6x^4 + 9x + 9}$ in GF(11).

(b)

$$\begin{aligned}(5x^2 + 2x + 7) \cdot (5x^3 + 3x^2 + 3x + 2) &= (25x^5 + 10x^4 + 35x^3) + (15x^4 + 6x^3 + 21x^2) \\ &\quad + (15x^3 + 6x^2 + 21x) + (10x^2 + 4x + 14) \\ &= 25x^5 + 25x^4 + 56x^3 + 37x^2 + 25x + 14 \\ &= 3x^5 + 3x^4 + x^3 + 4x^2 + 3x + 3\end{aligned}$$

$(5x^2+2x+7)\cdot(5x^3+3x^2+3x+2)$ is equivalent to $\mathbf{3x^5 + 3x^4 + x^3 + 4x^2 + 3x + 3}$ in GF(11).

(c)

$$\begin{aligned}\frac{x^5 + 8x^4 + x^3 + 4x^2 + 8x}{6x^3 + 3x^2 + 2} &= 2x^2 + \frac{x^5 + 8x^4 + x^3 + 4x^2 + 8x - (x^5 + 6x^4 + 4x^2)}{6x^3 + 3x^2 + 2} \\ &= 2x^2 + \frac{2x^4 + x^3 + 8x}{6x^3 + 3x^2 + 2} \\ &= 2x^2 + 4x + \frac{2x^4 + x^3 + 8x - (2x^4 + x^3 + 8x)}{6x^3 + 3x^2 + 2} \\ &= 2x^2 + 4x\end{aligned}$$

$\dfrac{x^5 + 8x^4 + x^3 + 4x^2 + 8x}{6x^3 + 3x^2 + 2}$ is equivalent to $\mathbf{2x^2 + 4x}$ in GF(11).

1.2 Problem 2

For the finite field GF(2^3), calculate the following for the modulus polynomial $x^3 + x^2 + 1$.

(a) $(x^2 + x + 1) \cdot (x + 1)$

(b) $(x^2 + 1) - (x^2 + x + 1)$

(c) $\dfrac{x^2 + x + 1}{x^2 + 1}$

Solution

(a)

$$\begin{aligned}(x^2 + x + 1) \cdot (x + 1) &= (x^3 + x^2 + x) + (x^2 + x + 1) \\ &= x^3 + 2x^2 + 2x + 1 \\ &= x^3 + 1 \\ &= x^2\end{aligned}$$

$(x^2 + x + 1) \cdot (x + 1)$ is equivalent to $\mathbf{x^2}$ in GF(2^3).

(b)

$$(x^2 + 1) - (x^2 + x + 1) = x$$

$(x^2 + 1) - (x^2 + x + 1)$ is equivalent to $\mathbf{x}$ in GF(2^3).

(c)

$$\begin{aligned}\frac{x^2 + x + 1}{x^2 + 1} &= (x^2 + x + 1) \cdot x \\ &= (x^2 + x) \oplus (x^2 + 1) \\ &= x + 1\end{aligned}$$

$\dfrac{x^2 + x + 1}{x^2 + 1}$ is equivalent to $\mathbf{x + 1}$ in GF(2^3).

2 Programming Problem

Write a script in Python to implement the AES algorithm with a 256-bit key size.

2.1 Python Code

```
#!/usr/bin/env python3

# Homework Number: 4
# Name: Elias Talcott
# ECN Login: etalcott
# Due Date: February 18, 2020

###
## Encryption call: python3 AES.py -e message.txt key.txt encrypted.txt
## Decryption call: python3 AES.py -d encrypted.txt key.txt decrypted.
    ↪ txt
###

import sys
from BitVector import *

BLOCKSIZE = 128

###
## Key schedule generation
###
AES_modulus = BitVector(bitstring='100011011')

def gee(keyword, round_constant, byte_sub_table):
    rotated_word = keyword.deep_copy()
    rotated_word << 8
    newword = BitVector(size = 0)
    for i in range(4):
        newword += BitVector(intVal = byte_sub_table[rotated_word[8*i
            ↪ :8*i+8].intValue()], size = 8)
    newword[:8] ^= round_constant
    round_constant = round_constant.gf_multiply_modular(BitVector(
        ↪ intVal = 0x02), AES_modulus, 8)
    return newword, round_constant

def gen_key_schedule_256(key_bv):
    byte_sub_table = subBytesTable
    # We need 60 keywords (each keyword consists of 32 bits) in the
        ↪ key schedule for
```

```
    # 256 bit AES. The 256-bit AES uses the first four keywords to xor
        ↪ the input
    # block with. Subsequently, each of the 14 rounds uses 4 keywords
        ↪ from the key
    # schedule. We will store all 60 keywords in the following list:
    key_words = [None for i in range(60)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(8):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(8,60):
        if i%8 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant,
                ↪ byte_sub_table)
            key_words[i] = key_words[i-8] ^ kwd
        elif (i - (i//8)*8) < 4:
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        elif (i - (i//8)*8) == 4:
            key_words[i] = BitVector(size = 0)
            for j in range(4):
                key_words[i] += BitVector(intVal = byte_sub_table[
                    ↪ key_words[i-1][8*j:8*j+8].intValue()], size = 8)
            key_words[i] ^= key_words[i-8]
        elif ((i - (i//8)*8) > 4) and ((i - (i//8)*8) < 8):
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        else:
            sys.exit("error in key scheduling algo for i = %d" % i)
    num_rounds = 14
    round_keys = [None for i in range(num_rounds+1)]
    for i in range(num_rounds+1):
        round_keys[i] = key_words[i*4] + key_words[i*4+1] + key_words[i
            ↪ *4+2] + key_words[i*4+3]
    return round_keys


###
## Create state array from 128-bit bitvector
###
def createStateArray(bv):
    return [[bv[0:8],   bv[32:40], bv[64:72], bv[96:104]],
            [bv[8:16],  bv[40:48], bv[72:80], bv[104:112]],
            [bv[16:24], bv[48:56], bv[80:88], bv[112:120]],
            [bv[24:32], bv[56:64], bv[88:96], bv[120:128]]]

def deconstructStateArray(bv_state):
    return bv_state[0][0] + bv_state[1][0] + bv_state[2][0] + bv_state
        ↪ [3][0] + bv_state[0][1] + bv_state[1][1] + bv_state[2][1] +
        ↪ bv_state[3][1] + bv_state[0][2] + bv_state[1][2] + bv_state
        ↪ [2][2] + bv_state[3][2] + bv_state[0][3] + bv_state[1][3] +
```

```
    ↪ bv_state[2][3] + bv_state[3][3]

###
## Substitute bytes and inverse substitute bytes table generation
###
subBytesTable = [99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43,
  ↪  254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240, 173,
  ↪ 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38, 54, 63,
  ↪ 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4, 199, 35, 195,
  ↪ 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44,
  ↪ 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132, 83,
  ↪ 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88,
  ↪ 207, 208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80,
  ↪ 60, 159, 168, 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218,
  ↪  33, 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196,
  ↪ 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42, 144,
  ↪ 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6,
  ↪ 36, 92, 194, 211, 172, 98, 145, 149, 228, 121, 231, 200, 55, 109,
  ↪  141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186,
  ↪ 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139,
  ↪ 138, 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134,
  ↪ 193, 29, 158, 225, 248, 152, 17, 105, 217, 142, 148, 155, 30,
  ↪ 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104,
  ↪  65, 153, 45, 15, 176, 84, 187, 22]

invSubBytesTable = [82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163,
  ↪ 158, 129, 243, 215, 251, 124, 227, 57, 130, 155, 47, 255, 135,
  ↪ 52, 142, 67, 68, 196, 222, 233, 203, 84, 123, 148, 50, 166, 194,
  ↪ 35, 61, 238, 76, 149, 11, 66, 250, 195, 78, 8, 46, 161, 102, 40,
  ↪ 217, 36, 178, 118, 91, 162, 73, 109, 139, 209, 37, 114, 248, 246,
  ↪  100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146,
  ↪ 108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141,
  ↪ 157, 132, 144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5,
  ↪ 184, 179, 69, 6, 208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189,
  ↪  3, 1, 19, 138, 107, 58, 145, 17, 65, 79, 103, 220, 234, 151,
  ↪ 242, 207, 206, 240, 180, 230, 115, 150, 172, 116, 34, 231, 173,
  ↪ 53, 133, 226, 249, 55, 232, 28, 117, 223, 110, 71, 241, 26, 113,
  ↪ 29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27, 252, 86,
  ↪ 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90, 244,
  ↪  31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128,
  ↪ 236, 95, 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159,
  ↪ 147, 201, 156, 239, 160, 224, 59, 77, 174, 42, 245, 176, 200,
  ↪ 235, 187, 60, 131, 83, 153, 97, 23, 43, 4, 126, 186, 119, 214,
  ↪ 38, 225, 105, 20, 99, 85, 33, 12, 125]

###
## Shift rows and inverse shift rows algorithms
###
```

```python
def shiftRows(bv_state):
    # Shift each row left by its index
    return [[bv_state[0][0], bv_state[0][1], bv_state[0][2], bv_state[0][3]],
            [bv_state[1][1], bv_state[1][2], bv_state[1][3], bv_state[1][0]],
            [bv_state[2][2], bv_state[2][3], bv_state[2][0], bv_state[2][1]],
            [bv_state[3][3], bv_state[3][0], bv_state[3][1], bv_state[3][2]]]

def invShiftRows(bv_state):
    # Shift each row right by its index
    return [[bv_state[0][0], bv_state[0][1], bv_state[0][2], bv_state[0][3]],
            [bv_state[1][3], bv_state[1][0], bv_state[1][1], bv_state[1][2]],
            [bv_state[2][2], bv_state[2][3], bv_state[2][0], bv_state[2][1]],
            [bv_state[3][1], bv_state[3][2], bv_state[3][3], bv_state[3][0]]]

###
## Mix columns and inverse mix columns tables for matrix multiplication
###
mixColumnsTable = [[BitVector(hexstring = "02"), BitVector(hexstring = "03"), BitVector(hexstring = "01"), BitVector(hexstring = "01")],
                   [BitVector(hexstring = "01"), BitVector(hexstring = "02"), BitVector(hexstring = "03"), BitVector(hexstring = "01")],
                   [BitVector(hexstring = "01"), BitVector(hexstring = "01"), BitVector(hexstring = "02"), BitVector(hexstring = "03")],
                   [BitVector(hexstring = "03"), BitVector(hexstring = "01"), BitVector(hexstring = "01"), BitVector(hexstring = "02")]]

invMixColumnsTable = [[BitVector(hexstring = "0E"), BitVector(hexstring = "0B"), BitVector(hexstring = "0D"), BitVector(hexstring = "09")],
                      [BitVector(hexstring = "09"), BitVector(hexstring = "0E"), BitVector(hexstring = "0B"), BitVector(hexstring = "0D")],
                      [BitVector(hexstring = "0D"), BitVector(hexstring = "09"), BitVector(hexstring = "0E"), BitVector(hexstring = "0B")],
                      [BitVector(hexstring = "0B"), BitVector(hexstring = "0D"), BitVector(hexstring = "09"),
```

```
        ↪ BitVector(hexstring = "0E")]]

# Multiply two 4x4 matrices of BitVectors
def fourByFourMultiply(a, b):
    c = [[BitVector(size = 8) for x in range(4)] for x in range(4)]
    for i in range(4):
        for j in range(4):
            for k in range(4):
                c[i][j] ^= a[i][k].gf_multiply_modular(b[k][j],
                    ↪ AES_modulus, 8)
    return c

###
## Encryption algorithm
###
def encrypt(infile, keyfile, outfile):
    # Create bitvectors for plaintext, key, and ciphertext
    with open(infile, "r") as fpin:
        plaintext_bv = BitVector(textstring = fpin.read())
    ciphertext_bv = BitVector(size = 0)
    with open(keyfile, "r") as fpkey:
        key_text = fpkey.read()
    if len(key_text) != 32:
        sys.exit("Key generation needs 32 characters exactly!")
    key_bv = BitVector(textstring = key_text)

    # Generate key schedule
    round_keys = gen_key_schedule_256(key_bv)

    # Encrypt plaintext
    plaintext_bv.pad_from_right(BLOCKSIZE - (len(plaintext_bv) %
        ↪ BLOCKSIZE))
    for i in range(len(plaintext_bv) // BLOCKSIZE):
        # XOR block with first round key and convert to state array
        bv = plaintext_bv[i * BLOCKSIZE:(i + 1) * BLOCKSIZE]
        bv ^= round_keys[0]
        bv_state = createStateArray(bv)
        # Do 14 rounds of processing
        for j in range(14):
            # Substitute bytes
            bv_state = [[BitVector(size = 8, intVal = subBytesTable[int
                ↪ (val)]) for val in row] for row in bv_state]
            # Shift rows
            bv_state = shiftRows(bv_state)
            # Mix columns except for last round
            if j != 13:
                bv_state = fourByFourMultiply(mixColumnsTable, bv_state
                    ↪ )
```

```
            # Add round key
            bv = deconstructStateArray(bv_state)
            bv ^= round_keys[j + 1]
            bv_state = createStateArray(bv)
        # Add encrypted block to ciphertext
        ciphertext_bv += deconstructStateArray(bv_state)

    # Save ciphertext to output file
    with open(outfile, "w") as fpout:
        fpout.write(ciphertext_bv.get_bitvector_in_hex())



###
## Decryption algorithm
###
def decrypt(infile, keyfile, outfile):
    # Create bitvectors for the plaintext, ciphertext, and key
    with open(infile, "r") as fpin:
        ciphertext_bv = BitVector(hexstring = fpin.read())
    plaintext_bv = BitVector(size = 0)
    with open(keyfile, "r") as fpkey:
        key_text = fpkey.read()
    if len(key_text) != 32:
        sys.exit("Key generation needs 32 characters exactly!")
    key_bv = BitVector(textstring = key_text)

    # Generate key schedule
    round_keys = gen_key_schedule_256(key_bv)

    # Decrypt ciphertext
    for i in range(len(ciphertext_bv) // BLOCKSIZE):
        # XOR block with last round key and convert to state array
        bv = ciphertext_bv[i * BLOCKSIZE:(i + 1) * BLOCKSIZE]
        bv ^= round_keys[-1]
        bv_state = createStateArray(bv)
        # Do 14 rounds of processing
        for j in range(14):
            # Inverse shift rows
            bv_state = invShiftRows(bv_state)
            # Inverse substitute bytes
            bv_state = [[BitVector(size=8, intVal = invSubBytesTable[
                ↪ int(val)]) for val in row] for row in bv_state]
            # Add round key
            bv = deconstructStateArray(bv_state)
            bv ^= round_keys[13 - j]
            bv_state = createStateArray(bv)
            # Inverse mix columns except for last round
            if j != 13:
```

```
                bv_state = fourByFourMultiply(invMixColumnsTable,
                    ↪ bv_state)
        # Add decrypted block to plaintext
        plaintext_bv += bv

    # Save plaintext to output file
    with open(outfile, "w") as fpout:
        fpout.write(plaintext_bv.get_text_from_bitvector())


###
## Check arguments and choose encrypt or decrypt option
###
if len(sys.argv) != 5:
    sys.exit("Wrong arguments!")

if sys.argv[1] == "-e":
    encrypt(sys.argv[2], sys.argv[3], sys.argv[4])
elif sys.argv[1] == "-d":
    decrypt(sys.argv[2], sys.argv[3], sys.argv[4])
else:
    sys.exit("Wrong arguments!")
```

2.2 Code Explanation

My program implements AES encryption and decryption in two separate functions. This is because the order of steps in encryption is different from the order of steps in decryption.

For encryption, I first XOR each 128-bit block with the first round key, then do 14 rounds of processing. Each round includes substituting bytes, shifting rows, mixing columns, and finally adding the round key.

For decryption, I first XOR each block with the last round key, then do 14 rounds of processing. The order of each round is inverse shifting rows, inverse substituting bytes, adding the round key, and finally inverse mixing columns.

A number of other functions were used to keep the encrypt and decrypt functions short and readable. For example, creating and deconstructing state arrays, shifting rows, and multiplying matrices were all done in their own separate functions.

When the program is run, it first checks if there is the proper number of arguments and then sends the arguments to the appropriate function based on the "-e" or "-d" input. The files are opened and read/written inside of the encryption and decryption functions. All of the functionality of AES is held within this file, and only the BitVector and sys modules are used.