Specification of AES Mini

Irad Nuriel

June 9, 2021

1 Cipher

The cipher takes a 64-bit key and 64-bit words and computes the ciphertext in 7 rounds. The words are divided into 8 bytes, called the state of the cipher.

1.1 Round function

As the name suggests, the round function of AES Mini is very similar to the round function of AES. As in AES it consists of a sequential application of 4 layers. Add Round Key(ARK), Sub Bytes(SB), Bit Permutation(BP) and Mix Columns(MC). To clarify how each layer works, we apply the first round of the plaintext : $0123\,4567\,89AB\,CDEF$

01	23
45	67
89	AB
CD	EF

with key: $0000\,0000\,FEDC\,BA98$

1.1.1 Add Round Key

In the ARK layer we doing a bitwise $\oplus$ of the round key with the cipher state. After adding the round key, the state of the cipher is:

01	23
45	67
77	77
77	77

1.1.2 Sub Bytes

In the SB layer, we apply the AES sbox(Which is derived from the multiplicative inverse over $GF(2^8)$) to every byte of the internal state.

After applying substitution layer, the state is:

$7C$	26
$6E$	85
$F5$	$F5$
$F5$	$F5$

1.1.3 Bit Permutation

In the BP layer, we pass each row through a bit permutation specific to that row(no bit moving between rows and the permutation for each row is different).

The first row, passes through:

$$\sigma_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\ 0 & 4 & 8 & C & 5 & 9 & D & 1 & A & E & 2 & 6 & F & 3 & 7 & B \end{pmatrix}$$

The second row, passes through:

$$\sigma_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\ 5 & 9 & D & 1 & A & E & 2 & 6 & F & 3 & 7 & B & 0 & 4 & 8 & C \end{pmatrix}$$

The third row, passes through:

$$\sigma_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\ A & E & 2 & 6 & F & 3 & 7 & B & 0 & 4 & 8 & C & 5 & 9 & D & 1 \end{pmatrix}$$

The fourth row, passes through:

$$\sigma_4 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\ F & 3 & 7 & B & 0 & 4 & 8 & C & 5 & 9 & D & 1 & A & E & 2 & 6 \end{pmatrix}$$

As you can see, basically all the rows passes through the permutation σ_1, and after that we apply the normal shift rows(for nibbles).

After the Bit Permutation layer, the state is:

$4B$	$E4$
$B3$	86
AF	AF
FA	FA

1.1.4 Mix Columns

In the MC layer, we mix the bytes in every column by multiplying each row with the MDS matrix:

$$MDS = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

After the Mix Columns layer, the state is:

$0D$	17
26	$E3$
$A8$	32
$2E$	$F1$

1.2 Key schedule

Given a master key K, the roundkey for the i-th round is given by:

$$k_i = \begin{cases} (k_{i-1} <<< 15) \oplus (k_{i-1} <<< 32) \oplus k_{i-1} \oplus 0x3 & i > 0 \\ K & i = 0 \end{cases}$$

1.3 Test vectors

Plaintext	Ciphertext	Key
0000000000000000	$5C56543E02F02358$	0000000000000000
0000000000000042	$5AB9E5B2C2DC4817$	0000000000000001
0123456789$ABCDEF$	$F0FE14D1C8C16C75$	00000000$FEDCBA98$

1.4 Reference Implementation

```
#!/usr/Bin/env python3

sbox = [0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76, 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59,
        0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0, 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1,
        0x71, 0xD8, 0x31, 0x15, 0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75, 0x09, 0x83,
        0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84, 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B,
        0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF, 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C,
        0x9F, 0xA8, 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, 0xCD, 0x0C, 0x13, 0xEC,
        0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73, 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE,
        0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB, 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
        0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08, 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6,
        0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A, 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9,
        0x86, 0xC1, 0x1D, 0x9E, 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF, 0x8C, 0xA1,
        0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16]


def rotateLeft(word, n, wordSize=64):
    mask = 2**wordSize - 1
    return ((word << n) & mask) | ((word >> (wordSize - n) & mask))


def nextRoundKey(roundKey):
    return (rotateLeft(roundKey, 15) ^ rotateLeft(roundKey, 32) ^ roundKey ^ 0x3)


def getRows(word):
    row0 = (word >> 48) & 0xFFFF
    row1 = (word >> 32) & 0xFFFF
    row2 = (word >> 16) & 0xFFFF
    row3 = (word >> 0) & 0xFFFF
    return row0, row1, row2, row3


def sigma(word):
    """          Implementing the sigma permutation on the 16 bit word.  """
    newWord = 0
    newWord |= (word & 0x4000) >> 6 # 1
    newWord |= (word & 0x2000) >> 8 # 2
    newWord |= (word & 0x1000) >> 10# 3
    newWord |= (word & 0x0800) << 3 # 4
    newWord |= (word & 0x0400) << 1 # 5
    newWord |= (word & 0x0200) >> 5 # 6
    newWord |= (word & 0x0100) >> 7 # 7
    newWord |= (word & 0x0080) << 6 # 8
    newWord |= (word & 0x0040) << 4 # 9
    newWord |= (word & 0x0020) << 2 # A
    newWord |= (word & 0x0010) >> 4 # b
    newWord |= (word & 0x0008) << 9 # C
    newWord |= (word & 0x0004) << 7 # D
    newWord |= (word & 0x0002) << 5 # E
    newWord |= (word & 0x0001) << 3 # F
    return newWord



def bitPermutation(word):
    """          Shift rows implementation    """
    row0, row1, row2, row3 = getRows(word)
    # Applying bit initial permutation
    row0 = sigma(row0)
    row1 = sigma(row1)
    row2 = sigma(row2)
    row3 = sigma(row3)

    # apply the shiftrows transformation(to make sure that no active bits will stay only in one column)
    row0 = row0
    row1 = rotateLeft(row1, 4, 16)
    row2 = rotateLeft(row2, 8, 16)
    row3 = rotateLeft(row3, 12, 16)
    # reConstruCt the word
    newWord  = row0 << 48       # a |= b <==> a = a | b
    newWord |= row1 << 32
    newWord |= row2 << 16
    newWord |= row3 << 0
    return newWord



def galoisMult(a, b):
    """      MultipliCation in the Galois field GF(2^8).      """
    p = 0
    hibitSet = 0
    for i in range(8):
        if b & 1 == 1:
            p ^= a
        hibitSet = a & 0x80
        a <<= 1
        if hibitSet == 0x80:
            a ^= 0x1B
        b >>= 1
    return p % 256



def mixColumn(column):
    """          The AES mix column for a single Column  """
    newCol = [0, 0, 0, 0]
    newCol[0] = galoisMult(column[0],2) ^ galoisMult(column[3],1) ^ galoisMult(column[2],1) ^ galoisMult(column[1],3)
    newCol[1] = galoisMult(column[1],2) ^ galoisMult(column[0],1) ^ galoisMult(column[3],1) ^ galoisMult(column[2],3)
    newCol[2] = galoisMult(column[2],2) ^ galoisMult(column[1],1) ^ galoisMult(column[0],1) ^ galoisMult(column[3],3)
    newCol[3] = galoisMult(column[3],2) ^ galoisMult(column[2],1) ^ galoisMult(column[1],1) ^ galoisMult(column[0],3)
    return newCol

def mixColumns(word):
    """          Implementation of the mix columns operation on "AES Mini"(which is the AES mixColumns but with only two columns instead of
        four)     """
    row0, row1, row2, row3 = getRows(word)
    column0 = []
    column1 = []
    column0.append((row0&0xFF00)>>8)
    column0.append((row1&0xFF00)>>8)
    column0.append((row2&0xFF00)>>8)
    column0.append((row3&0xFF00)>>8)


    column1.append(row0&0x00FF)
    column1.append(row1&0x00FF)
    column1.append(row2&0x00FF)
    column1.append(row3&0x00FF)

    column0 = mixColumn(column0)
    column1 = mixColumn(column1)

    newWord = 0
    for i in range(4):
        newWord |= (((column0[i] << 8) | column1[i]) << ((4-i-1)*16))
    return newWord



def applySbox(word, sbox):
    """          apply the sbox to every byte     """
```

```
    wordNew = 0
    for i in range(8): # 8 bytes
        j = i * 8
        byte = (word >> j) & 0xFF # retrieve the ith byte
        # insert the permuted byte in the CorreCt position
        wordNew |= sbox[byte] << j
    return wordNew


def roundFunction(word, roundKey):
    """          "AES Mini" round function    """
    word = word ^ roundKey
    word = applySbox(word, sbox)
    word = bitPermutation(word)
    word = mixColumns(word)
    return word


def encrypt(word, masterKey, rounds=7):
    roundKey = masterKey
    for i in range(rounds):
        # apply the roundfunction to word
        word = roundFunction(word, roundKey)
        # go to the next round key
        roundKey = nextRoundKey(roundKey)
    return word


def create_test_vectors():
    state = 0x0123456789ABCDEF
    firstRoundkey = 0x00000000FEDCBA98
    print("%016X" % state)
    state = state ^ firstRoundkey
    print("%016X" % state)
    state = applySbox(state, sbox)
    print("%016X" % state)
    state = bitPermutation(state)
    print("%016X" % state)
    state = mixColumns(state)
    print("%016X" % state)
    print("%016X" % 0, "%016X" % 0, "%016X" % encrypt(0, 0))
    print("%016X" % 0x42, "%016X" % 0x1, "%016X" % encrypt(0x42, 0x1))
    print("%016X" % 0x0123456789ABCDEF, "%016X" % 0x00000000FEDCBA98, "%016X" % encrypt(0x0123456789ABCDEF, 0x00000000FEDCBA98))



if __name__ == "__main__":
    import sys
    import hashlib
    import random

    if len(sys.argv) == 1:
        create_test_vectors()
        print("Error occured")
        exit()
    key = int(sys.argv[1], 16)      # We seed the random generator with a hash of the key to get the same messages for the same key
    random.seed(hashlib.sha256(sys.argv[1].encode()).digest())
    for i in range(16):
        word = random.getrandbits(64)
        cipher = encrypt(word, key, rounds=7)
        print("%016X %016X" % (word, cipher))
```

2 Why I designed the cipher that way?

2.1 Structure

I decided to go with an SPN and not with a feistel network, because I think that SPN sructures are more interesting, and most of the course we focused on them, so it may be more clear.

The idea to have the block devided into bytes instead of nibbles came to me when I thought on a way to build a good sbox.

2.2 Sub bytes

I decided to go with the AES sbox, as the AES cipher is proven to be a secure cipher, and thus its sbox is also secured, and the sbox is known to have a good non-linearity properties.

2.3 Bit permutation

I decided to go with a bit permutation and not on the normal shift rows, because in this way, I can make an active bit in the state move easily between columns, and not just stay in his column most of the time.

I decided that each row will have its own permutation, so that no active column will stay all together(as we disscused in the last lesson)

I decided that no bit will move between rows, because in this case, the permutation need to be good for the mix columns, and in that way I ensure that the permutation will not cancel the mix columns.

2.4 Mix columns

I decided to go with the AES Mix columns matrix, as I had the cipher deviced into bytes already, and because this matrix is a max distance seperator, so any little bit change in the column will cause the whole column to be different at the end of the mix columns operation.

2.5 Key shecule

I decided to go with the $TC01$ key schedule algorithm, as I couldn't really think of a different key sheduler, and I think that the key shedule is not really important for that cipher.

2.6 Naming

I decided to go with the name AES Mini, as I think that this name suite this cipher well, because this cipher use a lot of elements from AES.

2.7 Optimized implementation speed

The optimized implementation of the cupher could get to up to $5.395 \cdot 10^6 \frac{Encryptions}{second}$

Which is about $370 \frac{clockCycles}{Encryption}$ which I think is pretty fast