The python program AES.py implements the full AES algorithm. Given an encryption key and some plaintext, AES.py will write a file containing the correct encryption and decryption result. When command-line below is invoked, AES.py will perform AES encryption on the plaintext in message.txt using the key in key.txt and write the ciphertext to a file called encrypted.txt. '-e' symbolizes that AES.py will run the encryption process.

```
python AES.py -e message.txt key.txt encrypted.txt
```

Figure I. Example of Command-Line Syntax to Invoke AES Encryption using AES.py

An example of this is when a message.txt contains the following text:

Newly re-signed McLaren driver Lando Norris is confident that the team will be in the mix for race victories in 2024, but the Briton feels he may have to wait a little longer for a championship challenge. McLaren caught the eye last season by going from struggling to score points to regularly fighting for podiums, with highly effective upgrades being implemented following a technical reshuffle. Norris came close to scoring McLaren's first Grand Prix win since 2021 on several occasions, taking six P2 finishes, while team mate Oscar Piastri managed to triumph in the Qatar Sprint Race.

Figure II. Contents of the used message.txt

When the above command-line is invoked, AES.py will output the following to encrypted.txt:

3ba1ab4b7fe412ca26c7a25cff913d1b748da805c97c83554d9e9cf5b12243ff03a8c6b6dcbc520750a14df9b646fa480
d1e64cc2e9174a23dbed6aad77144350ff768093cf7571852a26ffa36fe47652a546acf9d4bc1ad395a92553b4b7e0a5
a7811d7b95d95cacc117e344ac093da247168cd4bbbda5bc2866fd044c8ca18ecd2b6a78bfe19520f22b7fa12862132
e32ee78c5e4200166c40f1a93f9b08c5f67b9bde38d34ed34bd03183a529a5a62d81b1cf084832fcb9139a51100a04c
7c631d3fbfa5bb9b8cbe970f02213ab07d3e179313142865fb8b022241552567964250cfa2aa97c59223d30a2a7da89
74d0f6c34f4f46ed6cab53e483f95d4ed157bb78ce078a88397c9d656830fadd080d729ac7428a6ca3c17ad67d0cf16d
35a8ecb35cd818a380309332c4cc29d00b6fe542b67724295b49804b2122b5b24e6f09e22451bb77c6876d51b7294
b405dcff0cdc83754538442fcc766bfe4fac839e932f757aebbe7f43c87d08249c6ef50d9adefa8eca175785ba0dbc31e
2e61ba32a75f596894ea736bcea8f351d3c4574539e7ad760c4a0c4b252e2dbc859c4b0a6b44fbf29b3fa7fddeace385
5c675130ef65d4fa7f8125d4575f329cc93d75d14fdcb1419678cae4d686d4b72f56ac4d7974e3b1f1bbb3776dda5db
94b7d2ef1f73f96f7b24378a1e299271006cd478bd84fe7a24c67794e663668c918bdb65097099351e1ebf6e7d11487
54f1051d33156e4fb7e96cce8f976f6a0ad71d12b10d1b43458c02002bf1fc14c9c63e9033dfdcbc9baae76efc8e12a85
0fdd21ead4e9b14fb359a27fc4943b0d76714

Figure III. Contents of the written output of to encrypted.txt using AES Encryption in AES.py

It is important to note that the output file of the AES encryption process of AES.py is saved as a single-line hex file. The overall AES encryption process is in accordance with the standard of Advanced Encryption Algorithm, where it first will generate a key schedule depending upon the provided key-length. The algorithm would then continuously read 128-bit block from the file to be encrypted and would pad enough bits to ensure that the length is 128-bit. Each of the 128-bit block will go through the following:

1. XOR-ed the input State Array with first 4 words of key schedule
2. Go through 14 rounds of Single-byte based substitution, Row-wise permutation, Column-wise mixing, XOR with the round key.

For further reference, below is the AES encryption algorithm implemented in AES.py:

```
def encrypt (self, plaintext :str, ciphertext :str) -> None:
    key_schedule = self.generateKeySchedule(self.key_str)
    self.get_STable()
    bv = BitVector(filename=plaintext)
    output_file = open(ciphertext, 'w')
    while bv.more_to_read:
        bitvec = bv.read_bits_from_file(128)
        bitvec.pad_from_right(128 - bitvec.length())

        statearray = [[0 for x in range(4)] for x in range(4)]
        for i in range(4):
            for j in range(4):
                statearray[i][j] = bitvec[32*j + 8*i:32*j + 8 * (i + 1)]

        key_array = [[0 for x in range(4)] for x in range(4)]
        for j in range(4):
            keyword = key_schedule[j]
            for i in range(4):
                key_array[i][j] = keyword[i * 8:i * 8 + 8]

        statearray = self.stateArrXor(statearray, key_array)

        for roundNum in range(14):

            key_array = [[0 for x in range(4)] for x in range(4)]
            for j in range(4):
                roundkw = key_schedule[j + 4 * (roundNum + 1)]
                for i in range(4):
                    key_array[i][j] = roundkw[i * 8:i * 8 + 8]

            statearray = self.subBytes(statearray)
            statearray = self.shiftRow(statearray)

            if roundNum != 13:
                statearray = self.mixCols(statearray)

            statearray = self.stateArrXor(statearray, key_array)

        for j in range(4):
            for i in range(4):
                bv_to_print = statearray[i][j]
                hexstr = bv_to_print.get_hex_string_from_bitvector()
                output_file.write(hexstr)
```

Figure IV. AES Encryption algorithm of AES.py

Furthermore, to invoke an AES decryption on such file called encrypted.txt using a key in key.txt with the output written to decrypted.txt, the following command-line is used, where '-d' symbolizes the usage of AES decryption:

```
python AES.py -d encrypted.txt key.txt decrypted.txt
```

Figure VI. Example of Command-Line Syntax to Invoke AES Decryption using AES.py

It might be important to note that the algorithm used for the AES encryption and AES encryption are very similar, in that it only differs in the last round for encryption does not involve the 'Mix Columns' step and the last round for decryption does not involve the 'Inverse Mix Columns' step. Below is the algorithm used for the AES decryption algorithm in AES.py:

```
def decrypt (self, ciphertext :str, decrypted :str) -> None:

    key_schedule = self.generateKeySchedule(self.key_str)

    self.get_STable()

    bv = BitVector(filename=ciphertext)
    output_file = open(decrypted, 'wb')

    while bv.more_to_read:
        encrypted_text = bv.read_bits_from_file(256)
        bitvec = BitVector(hexstring=encrypted_text.get_bitvector_in_ascii())

        bitvec.pad_from_right(128 - bitvec.length())

        stateArray = [[0 for x in range(4)] for x in range(4)]
        for i in range(4):
            for j in range(4):
                stateArray[i][j] = bitvec[32 * j + 8 * i:32 * j + 8 * (i + 1)]

        key_array = [[0 for x in range(4)] for x in range(4)]
        for j in range(4):
            keyword = key_schedule[56 + j]
            for i in range(4):
                key_array[i][j] = keyword[i * 8:i * 8 + 8]

        stateArray = self.stateArrXor(stateArray, key_array)

        for roundNum in range(14):
            key_array = [[0 for x in range(4)] for x in range(4)]
            for j in range(4):
                roundkm = key_schedule[j + 52 - 4 * roundNum]
                for i in range(4):
                    key_array[i][j] = roundkm[i * 8:i * 8 + 8]

            stateArray = self.invShiftedRow(stateArray)
            stateArray = self.invSubByte(stateArray)
            stateArray = self.stateArrXor(stateArray, key_array)
            if roundNum != 13:
                stateArray = self.invMixCol(stateArray)

        for j in range(4):
            for i in range(4):
                bv_to_print = stateArray[i][j]
                bv_to_print.write_to_file(output_file)
```

Figure VII. AES Decryption algorithm of AES.py

An example of the AES decryption algorithm can be performed using a file called encrypted.txt that contains the cipher text in Figure II (the output file of AES encryption), which, when invoked using the command-line in figure VI, AES.py writes the following into decrypted.txt:

Scuderia Ferrari is the racing division of luxury Italian auto manufacturer Ferrari and the racing team that competes in Formula One racing. The team is also known by the nickname "The Prancing Horse", in reference to their logo. It is the oldest surviving and most successful Formula One team, having competed in every world championship since the 1950 Formula One season. The team was founded by Enzo Ferrari, initially to race cars produced by Alfa Romeo. By 1947 Ferrari had begun building its own cars. Among its important achievements outside Formula One are winning the World Sportscar Championship, 24 Hours of Le Mans, 24 Hours of Spa, 24 Hours of Daytona, 12 Hours of Sebring, Bathurst 12 Hour, races for Grand tourer cars and racing on road courses of the Targa Florio, the Mille Miglia and the Carrera Panamericana. The team is also known for its passionate support base, known as the tifosi. The Italian Grand Prix at Monza is regarded as the team's home race.

Figure VIII. Contents of the written output to decrypted.txt using AES Decryption in AES.py

Note that the extra spaces at the end are caused by padding by zeroes (null byte characters) when the plaintext bit size is not divisible by the AES block size.

For further reference, below is any of the user-defined functions used in AES.py but not mentioned in the document:

```
def __init__(self, keyfile: str) -> None:
    self.AES_modulus = BitVector(bitstring='100011011')
    self.subBytesTable = []
    self.invSubByteTable = []

    key = open(keyfile)
    self.key_str = key.read()
```

Figure XIII. __main__ function of AES.py

```
def shiftRow(self, stateArray):
    shiftedRow = [[None for x in range(4)] for x in range(4)]

    for j in range(4):
        shiftedRow[0][j] = stateArray[0][j]
    for j in range(4):
        shiftedRow[1][j] = stateArray[1][(j + 1) % 4]
    for j in range(4):
        shiftedRow[2][j] = stateArray[2][(j + 2) % 4]
    for j in range(4):
        shiftedRow[3][j] = stateArray[3][(j + 3) % 4]
    return shiftedRow
```

Figure XIV. Function to Shift the Rows for AES Encryption

```
def generate_subbyte_table(self):
    subBytesTable = []
    cons = BitVector(bitstring='01100011')
    for i in range(0, 256):
        subbyte = BitVector(intVal = i, size=8).gf_MI(self.AES_modulus, 8) if i != 0 else
BitVector(intVal=0)
        a1, a2, a3, a4 = [subbyte.deep_copy() for x in range(4)]
        subbyte ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ cons
        subBytesTable.append(int(subbyte))
    return subBytesTable

def get_SubTable(self):
    sub1 = BitVector(bitstring='001100011')
    sub2 = BitVector(bitstring='00000101')
    for i in range(0, 256):
        subTable1 = BitVector(intVal = i, size=8).gf_MI(self.AES_modulus, 8) if i != 0 else
BitVector(intVal=0)
        a1,a2,a3,a4 = [subTable1.deep_copy() for x in range(4)]
        subTable1 ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ sub1
        self.subBytesTable.append(int(subTable1))
        subTable2 = BitVector(intVal = i, size=8)
        b1,b2,b3 = [subTable2.deep_copy() for x in range(3)]
        subTable2 = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ sub2
        check = subTable2.gf_MI(self.AES_modulus, 8)
        subTable2 = check if isinstance(check, BitVector) else 0
        self.invSubByteTable.append(int(subTable2))
```

Figure XV. Function to Generate the Substitution Bytes Tables for Encryption and Decryption in AES

```
def gkey(self, 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), self.AES_modulus, 8)
    return newword, round_constant
```

Figure XVI. Function to Apply the g function used to Generate the Keys

```
def mixCol(self, statearray):
    mixed = [[0 for x in range(4)] for x in range(4)]

    for j in range(4):
        bv1 = statearray[0][j].gf_multiply_modular(BitVector(hexstring='02'), self.AES_modulus, 8)
        bv2 = statearray[1][j].gf_multiply_modular(BitVector(hexstring='03'), self.AES_modulus, 8)
        mixed[0][j] = bv1 ^ bv2 ^ statearray[2][j] ^ statearray[3][j]
    for j in range(4):
        bv1 = statearray[1][j].gf_multiply_modular(BitVector(hexstring='02'), self.AES_modulus, 8)
        bv2 = statearray[2][j].gf_multiply_modular(BitVector(hexstring='03'), self.AES_modulus, 8)
        mixed[1][j] = bv1 ^ bv2 ^ statearray[0][j] ^ statearray[3][j]
    for j in range(4):
        bv1 = statearray[2][j].gf_multiply_modular(BitVector(hexstring='02'), self.AES_modulus, 8)
        bv2 = statearray[3][j].gf_multiply_modular(BitVector(hexstring='03'), self.AES_modulus, 8)
        mixed[2][j] = bv1 ^ bv2 ^ statearray[0][j] ^ statearray[1][j]
    for j in range(4):
        bv1 = statearray[3][j].gf_multiply_modular(BitVector(hexstring='02'), self.AES_modulus, 8)
        bv2 = statearray[0][j].gf_multiply_modular(BitVector(hexstring='03'), self.AES_modulus, 8)
        mixed[3][j] = bv1 ^ bv2 ^ statearray[1][j] ^ statearray[2][j]
    return mixed
```

Figure XVII. Function to Replace Each Byte of a Column by a Function of All the Bytes in the Same Column for AES Encryption

```
def subBytes(self, statearray):
    for i in range(4):
        for j in range(4):
            statearray[i][j] = BitVector(intVal = self.subBytesTable[int(statearray[i][j])], size=8)
    return statearray
```

Figure XVIII. Function for the Substitution step of AES encryption

```
def invSubByte(self, statearray):
    for i in range(4):
        for j in range(4):
            statearray[i][j] = BitVector(intVal = self.invSubByteTable[int(statearray[i][j])], size=8)
    return statearray
```

Figure XVIII. Function for the Substitution step of AES decryption

```
def generateKeySchedule(self, key: str) -> list:
    schedule = [None for i in range(60)]
    round_constant = BitVector(intVal = 0x01, size=8)

    key_bv = BitVector(textstring=key)

    byte_sub_table = self.generate_subbyte_table()

    for i in range(8):
        schedule[i] = key_bv[i * 32: i * 32 + 32]
    for i in range(8, 60):
        if i % 8 == 0:
            kwd, round_constant = self.gkey(schedule[i - 1], round_constant, byte_sub_table)
            schedule[i] = schedule[i - 8] ^ kwd
        elif(i - (i // 8) * 8) < 4:
            schedule[i] = schedule[i - 8] ^ schedule[i - 1]
        elif (i - (i // 8) * 8) == 4:
            schedule[i] = BitVector(size=0)
            for j in range(4):
                schedule[i] += BitVector(intVal=byte_sub_table[schedule[i - 1][8 * j:8 * j + 8].intValue()], size=8)
            schedule[i] ^= schedule[i - 8]
        elif ((i - (i // 8) * 8) > 4) and ((i - (i // 8) * 8) < 8):
            schedule[i] = schedule[i - 8] ^ schedule[i - 1]
        else:
            sys.exit(f"error in key scheduling algorithm for i = {i}")
    return schedule
```

Figure XIX. Function to Generate the AES Key Schedule

```
def stateArrXor(self, sa1, sa2):
    for i in range(4):
        for j in range(4):
            sa1[i][j] = sa1[i][j] ^ sa2[i][j]
    return sa1
```

Figure XX. Function to XOR the Two State Array

```
def invShiftedRow(self, statearray):
    shifted = [[None for x in range(4)] for x in range(4)]

    for j in range(4):
        shifted[0][j] = statearray[0][j]
    for j in range(4):
        shifted[1][j] = statearray[1][(j - 1) % 4]
    for j in range(4):
        shifted[2][j] = statearray[2][(j - 2) % 4]
    for j in range(4):
        shifted[3][j] = statearray[3][(j - 3) % 4]
    return shifted
```

Figure XXI. Function to Inverse Shifted Rows

```
def invMixCol(self, statearray):
    mixed = [[0 for x in range(4)] for x in range(4)]

    for j in range(4):
        bv1 = statearray[0][j].gf_multiply_modular(BitVector(hexstring='0e'), self.AES_modulus, 8)
        bv2 = statearray[1][j].gf_multiply_modular(BitVector(hexstring='0b'), self.AES_modulus, 8)
        bv3 = statearray[2][j].gf_multiply_modular(BitVector(hexstring='0d'), self.AES_modulus, 8)
        bv4 = statearray[3][j].gf_multiply_modular(BitVector(hexstring='09'), self.AES_modulus, 8)
        mixed[0][j] = bv1 ^ bv2 ^ bv3 ^ bv4
    for j in range(4):
        bv1 = statearray[0][j].gf_multiply_modular(BitVector(hexstring='09'), self.AES_modulus, 8)
        bv2 = statearray[1][j].gf_multiply_modular(BitVector(hexstring='0e'), self.AES_modulus, 8)
        bv3 = statearray[2][j].gf_multiply_modular(BitVector(hexstring='0b'), self.AES_modulus, 8)
        bv4 = statearray[3][j].gf_multiply_modular(BitVector(hexstring='0d'), self.AES_modulus, 8)
        mixed[1][j] = bv1 ^ bv2 ^ bv3 ^ bv4
    for j in range(4):
        bv1 = statearray[0][j].gf_multiply_modular(BitVector(hexstring='0d'), self.AES_modulus, 8)
        bv2 = statearray[1][j].gf_multiply_modular(BitVector(hexstring='09'), self.AES_modulus, 8)
        bv3 = statearray[2][j].gf_multiply_modular(BitVector(hexstring='0e'), self.AES_modulus, 8)
        bv4 = statearray[3][j].gf_multiply_modular(BitVector(hexstring='0b'), self.AES_modulus, 8)
        mixed[2][j] = bv1 ^ bv2 ^ bv3 ^ bv4
    for j in range(4):
        bv1 = statearray[0][j].gf_multiply_modular(BitVector(hexstring='0b'), self.AES_modulus, 8)
        bv2 = statearray[1][j].gf_multiply_modular(BitVector(hexstring='0d'), self.AES_modulus, 8)
        bv3 = statearray[2][j].gf_multiply_modular(BitVector(hexstring='09'), self.AES_modulus, 8)
        bv4 = statearray[3][j].gf_multiply_modular(BitVector(hexstring='0e'), self.AES_modulus, 8)
        mixed[3][j] = bv1 ^ bv2 ^ bv3 ^ bv4
    return mixed
```

Figure XXII. Function to Replace each byte of a column by a function of all the bytes in the same column for AES Decryption