Security for IoT Assignment 1

Name: Vishnu Simha Dussa
Student ID: 22262621
Used programming language: C++
Used IDE: Eclipse for C++
Used online tool for text to hex conversions:
https://www.rapidtables.com/convert/number/ascii-to-hex.html

How to Run the code?

1) Extract the zip file and open the project with Eclipse for C++ IDE.
2) Build the project, go to main function file i.e., AssignmentPresent80/AssignmentAES and run as local C++ application.
3) All the inputs and keys hardcoded, so the code doesn't ask for any inputs and once the execution completes the output can be seen in console.
4) 3 standards inputs are kept commented for testing the code with different inputs. Please validate by uncommenting one after another.

Note: Instead of pasting the code and explaining the code at bottom, I have explained the AES-128 algorithm by adding code and explanation one after another.

I have also added more comments in each section of code. Most of those comments are omitted only in the report, to have report well structured.

Overview of AES-128 algorithm

The AES-128 algorithm is a **symmetric key encryption algorithm**, meaning that the ***same key*** is used for both ***encryption*** and ***decryption***.

Here are the steps involved in the AES-128 encryption algorithm:

Key Expansion: The main input 128-bit encryption key is initialised to expand into an array of 11 round keys, each of 128 bits long. The key expansion uses a key schedule algorithm that performs some steps with the original key and finally expands to the size of 176 bits.

The **initKeyForExpansion** function copies input key into **expandedKeys** array in its first 16 bytes and then it generates additional keys until it has created 176 bytes of keys (11 rounds * 16 bytes/round).

```
void Encryption::initKeyForExpansion(unsigned char inputKey[16]) {

unsigned char tempCoreStorage[4]; // Temporary storage for core

int rconstantCurrIteration = 1; // To know current Iteration with
round constant value

int totalBytesCreated = 16;

for (int i = 0; i < 16; i++) {

expandedKeysArray[i] = inputKey[i];

}

coreKeyExpansion(totalBytesCreated, tempCoreStorage,

rconstantCurrIteration);

}
```

Now while the total bytes created is less than 176 bytes, we perform some operations. First we store the main key in first 16 bytes and then we read the last 4 bytes of the previously generated key and stores them in an tmpCoreStroage array. If this is the first iteration of the loop, then tmpCoreStroage is last 4 bytes of the input key.

Then we call **KeyExpansionCore** function if the current byte count is a multiple of 16. Then KeyExpansionCore generates a new 4-byte tmpCoreStroage array by performing bitwise operations on the previous 4-byte array like left shift, substituting S box values in place of input 4 bits and finally it performs XOR operation.

```
void Encryption::coreKeyExpansion(int totalBytesCreated,
unsigned char tempCoreStorage[4], int rconstantCurrIteration) {

while (totalBytesCreated < 176) {

for (int i = 0; i < 4; i++) {

tempCoreStorage[i] = expandedKeysArray[i + totalBytesCreated -
4];}

if (totalBytesCreated % 16 == 0) {

bitWiseOperations(tempCoreStorage, rconstantCurrIteration++);

}

for (unsigned char j = 0; j < 4; j++) {

expandedKeysArray[totalBytesCreated] =

expandedKeysArray[totalBytesCreated - 16]

^ tempCoreStorage[j];

totalBytesCreated++;

}}}

void Encryption::bitWiseOperations(unsigned char *input, unsigned
char c) {

unsigned char ch = input[0];

input[0] = input[1];

input[1] = input[2];

input[2] = input[3];

input[3] = ch; // This part do the left shift by moving a bit to
left

input[0] = sBoxForSubstution[input[0]];

input[1] = sBoxForSubstution[input[1]];

input[2] = sBoxForSubstution[input[2]];

input[3] = sBoxForSubstution[input[3]]; // Substituting the S-box
values input the place of 4 bytes

input[0] ^= roundConstant[c]; // Performing the XOR operation with
the Round constant Constant}
```

At last by XORing each byte from the previous 16 bytes of the expanded key with a matching byte from the tmpCoreStroage array, it creates the last 16 bytes of the expanded key. The resultant bytes are added to the expandedKeys array, and a 16-byte increment is made to the byte count. The method continues generating keys in this way until it has generated a total of 176 bytes of expanded keys.

Encryption process:

From the main function we can the **encrypt function by passing input message and ciphertext pointer.**

```
int main() {

// Object creation of encryption

Encryption encryption;

unsigned char encryptedCipherText[16];

// Initializing the message and input key

unsigned char inputMessage[] = { 0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40,
0x9F, 0x96,

0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A }; // input as per
course pdf

unsigned char inputKey[] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE,
0xD2, 0xA6,

0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C }; // inputKey as per
course pdf

encryption.initKeyForExpansion(inputKey); // Initializing the
inputKey for the key expansion

printTextOrCipher("inputMessage : ",inputMessage);

//Encrypt the inputMessage

encryption.encrypt(inputMessage, encryptedCipherText);

cout << endl;

printTextOrCipher("encryptedCipherText : ",encryptedCipherText);

return 0;}
```

This **encrypt method** encrypts the input message in block encryption method. We send our inputMessage byte array to this function and it encrypts that block by block of each 16 bytes long.

```
void Encryption::encrypt(unsigned char *inputMessage,

unsigned char *encryptedCipherText, int noOfBlocks) {

for (int i = 0; i < noOfBlocks; i++) {

performBlockEncryption(inputMessage + (i * 16),

encryptedCipherText + (i * 16));}}
```

```
void Encryption::performBlockEncryption(unsigned char
*inputMessage,

unsigned char *encryptedMessage) {

int rounds = 9;

unsigned char currentState[16];

for (int i = 0; i < 16; i++) {

currentState[i] = inputMessage[i];

// storing starting 16 bytes input inputMessage}

firstRound(currentState, expandedKeysArray);

for (int i = 0; i < rounds; i++) { // main remaining rounds

// these rounds include mixing of columns

roundEncryption(currentState, expandedKeysArray + (16 * (i +
1)));}

lastRound(currentState, expandedKeysArray + 160);

// Copy encrypted state to buffer

for (int i = 0; i < 16; i++) {encryptedMessage[i] =
currentState[i];

}

}
```

This method calls the **performBlockEncryption** method to encrypt each block. The performBlockEncryption function takes a pointer to the input message block, and a pointer to the output encrypted block and the that variable I is used to iterate through the blocks.

performBlockEncryption performs main rounds in the ECB encryption process and it is also responsible for confusion and diffusion steps.

In that **firstRound** method, we perform XOR operation with the 128-bit plaintext block and first round key.

```
void Encryption::firstRound(unsigned char *currentState,

unsigned char *roundKey) {for (int i = 0; i < 16; i++) {

currentState[i] ^= roundKey[i];}}
```

```
void Encryption::roundEncryption(unsigned cha *currentState,
unsigned char *key) {

substituteBytes(currentState);

shiftingRows(currentState);

mixingColumns(currentState);

firstRound(currentState, key);

}
```

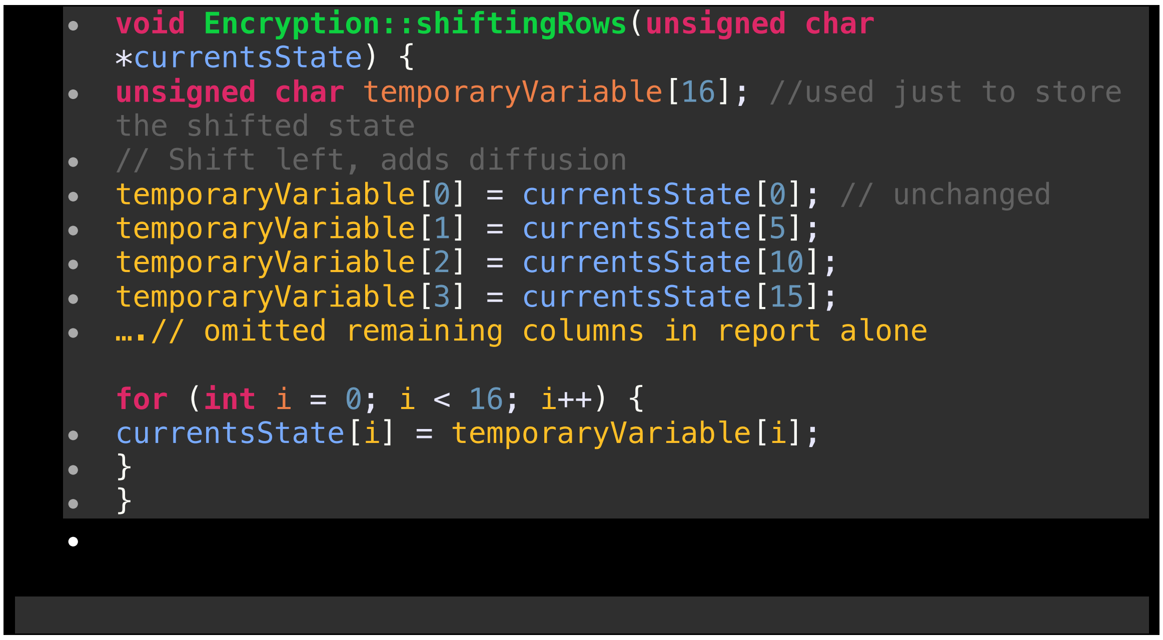
In the middle main rounds, for each round we call **roundEncryption** method and that performs four different operations on the input:

- **Substitution of Bytes:** The 128-bit block is replaced by a new block based on S-box substitution table. Each byte in the block is replaced with a new byte based on the value of the original byte using the S-box, a fixed table of 256 bytes.
- **Shifting Rows:** Different levels of leftward shifting is performed to the rows of the 128-bit block. The first row is unshifted/unchanged, the second row has a one-byte shift, the third row has a two-byte shift, and the fourth row has a three-byte shift. This row-wise shifting operation is a crucial part of the AES algorithm because it increases the security of the encrypted data and provides diffusion.
- **Mixing Columns:** The columns of the 128-bit block are transformed using a mathematical operation that combines each column byte with the other bytes in the column. In this transformation, each column of the 4x4 state matrix is multiplied with a fixed polynomial, resulting in a new column in this I have take multiplyBy2 and multiplyBy3 look up tables for mixing columns.

$$
\begin{aligned}
s'_{0,j} &= (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\
s'_{1,j} &= s_{0,j} \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j} \\
s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j}) \\
s'_{3,j} &= (3 \bullet s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \bullet s_{3,j})
\end{aligned}
$$

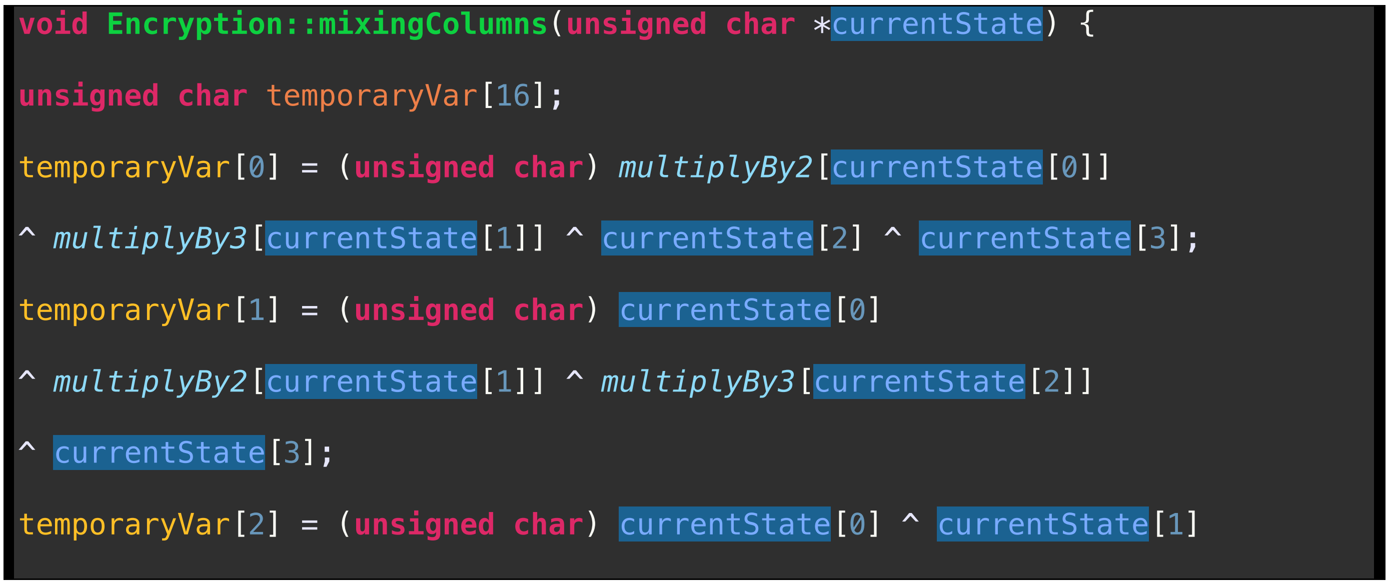
Image source: course notes

- **XOR with round key:** Now again 128-bit block is XORed with the current round key. Since the first round also performs the same activity we can put that logic into a separate function and call the same method.



```
• void Encryption::shiftingRows(unsigned char
  *currentsState) {
• unsigned char temporaryVariable[16]; //used just to store
  the shifted state
• // Shift left, adds diffusion
• temporaryVariable[0] = currentsState[0]; // unchanged
• temporaryVariable[1] = currentsState[5];
• temporaryVariable[2] = currentsState[10];
• temporaryVariable[3] = currentsState[15];
• ….// omitted remaining columns in report alone

  for (int i = 0; i < 16; i++) {
• currentsState[i] = temporaryVariable[i];
• }
• }
•
```



```
void Encryption::mixingColumns(unsigned char *currentState) {

unsigned char temporaryVar[16];

temporaryVar[0] = (unsigned char) multiplyBy2[currentState[0]]

^ multiplyBy3[currentState[1]] ^ currentState[2] ^ currentState[3];

temporaryVar[1] = (unsigned char) currentState[0]

^ multiplyBy2[currentState[1]] ^ multiplyBy3[currentState[2]]

^ currentState[3];

temporaryVar[2] = (unsigned char) currentState[0] ^ currentState[1]
^ multiplyBy2[currentState[2]] ^ multiplyBy3[currentState[3]];

temporaryVar[3] = (unsigned char) multiplyBy3[currentState[0]]

^ currentState[1] ^ currentState[2] ^ multiplyBy2[currentState[3]];

temporaryVar[4] = (unsigned char) multiplyBy2[currentState[4]]

^ multiplyBy3[currentState[5]] ^ currentState[6] ^ currentState[7];

….// omitted remaining columns in report alone

// For Mixing columns used multiplyBy2 and multiplyBy3 tables to replace
valuesfor (int i = 0; i < 16; i++) {

currentState[i] = temporaryVar[i];}}
```

4. **last Round:** In the last round, the **Substitution of Bytes**, **Shifting Rows**, and **XOR with round key** operations are performed again, but the **Mixing Columns** operation is removed.

```
void Encryption::lastRound(unsigned char *currentState, unsigned char
*key) {

substituteBytes(currentState);

shiftingRows(currentState);

firstRound(currentState, key);

}
```

5. **Output:** The resulting 128-bit **ciphertext** is the encrypted form of the original **plaintext**.

```
inputMessage : 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73
93 17 2a

encryptedCipherText : 3a d7 7b b4 d 7a 36 60 a8 9e ca
f3 24 66 ef 97
```

Testing AES algorithm:

The above code is tested against different standard inputs that is used to check the AES-128 algorithm and I have **got the desired cipher text outputs** as expected.

For Given Input 1:

```
unsigned char inputMessage [] = { 0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
                                  0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A };
unsigned char inputKey [] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6,
                                  0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };
```

The output in console is.

inputMessage: 6b c1 be e2 2e 40 9f 96 e9 3d 7e 11 73 93 17 2a
encryptedCipherText**: 3a d7 7b b4 d 7a 36 60 a8 9e ca f3 24 66 ef 97**

For Given Input 2:

```
unsigned char inputMessage[] = {
0xF6,0x9F,0x24,0x45,0xDF,0x4F,0x9B,0x17,0xAD,0x2B,0x41,0x7B,0xE6,0x6C,0x37,0x10};
unsigned char inputKey[] = {
0x2B,0x7E,0x15,0x16,0x28,0xAE,0xD2,0xA6,0xAB,0xF7,0x15,0x88,0x09,0xCF,0x4F,0x3C};
```

The output in console is.

```
inputMessage : f6 9f 24 45 df 4f 9b 17 ad 2b 41 7b e6 6c 37 10
encryptedCipherText : 7b c 78 5e 27 e8 ad 3f 82 23 20 71 4 72 5d d4
```

For Given Input 3:

```
unsigned char inputMessage[] = {
0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
unsigned char inputKey[] = {
0xAB,0xCD,0xEF,0x79,0x89,0x39,0x02,0x93,0xFE,0xDC,0xBA,0x90,0x59,0x80,0x03,0x31};
```

The output in console is.

```
inputMessage : ff ff ff ff ff ff ff ff 0 0 0 0 0 0 0 0
encryptedCipherText : 1f 77 22 18 55 f9 d9 4e f7 29 65 77 0 f 6d a9
```

Present-80 algorithm

The Present-80 algorithm is a lightweight block cipher with a block size of 64 bits and a key size of 80 bits. It was designed by Andrey Bogdanov in 2007 and has been widely used in the industry due to its simplicity, efficiency, and security.

Below is the explanation and implementation of Present-80 algorithm:

1. **Key Generation:** This is the first step in encryption. In this a key schedule is initialised with 80-bit size key. This key schedule is used to generate another 32 round sub keys each 32 bits long.

   At first the 80 bit key is converted a **keyLeftHigh** and **keyRightLow**. We take high 64 bits of the input key and assign to **keyLeftHigh** and in the same way we assign low 16 bits to **keyRightLow.** Now we set **keyLeftHigh** as the first sub key.

   Next, we run a for loop to generate total of 32 sub keys. In the loop, the left half of the previous sub key is rotated by 61 bits and right half is rotated by 45 bits. Now we take these two to form new sub key left half by performing the OR operation with the left half and the rotated right half and also shifting the previous left half by 19 bits to the right side. And final output is stored to keyLeftHigh.

   The right half new subkey is made with previous left half by shifting it 3 bits to the right and masking out the upper 48 bits to get 16-bit right half of the new subkey and that is stored in keyRightLow. Now, new sub key right half is XORed with value that is dependent on iteration count and left half is stored as new sub key, in this way the loop works and continues.

   Finally all the generated sub keys are stored in subKeys array.

```
uint64_t* generateSubkeys(char *inputKey) {

//Generating 32 64-bit sub keys with the input key

uint64_t keyLeftHigh = hexaDecimalStringToLong(inputKey);

uint16_t keyRightLow = getKeyRightLow(inputKey);

uint64_t *subKeys = new uint64_t[32]; // 32 (64-bit) sub keys will gets stored
here

cout << "Key/keyLeftHigh/keyRightLow " << inputKey << " "

<< longToHexaDecimalString(keyLeftHigh) << " "

<< longToHexaDecimalString(keyRightLow) << endl;
subKeys[0] = keyLeftHigh;

for (int i = 1; i < 32; i++) {

uint64_t temporaryVar1 = keyLeftHigh;//64 bit var

uint64_t temporaryVar2 = keyRightLow;//64 bit var

uint8_t temporaryVar; //80 bit

keyLeftHigh = (keyLeftHigh << 61) | (temporaryVar2 << 45) | (temporaryVar1 >>
19);

keyRightLow = ((temporaryVar1 >> 3) & 0xFFFF);

temporaryVar = substitutionWithSBox(keyLeftHigh >> 60);

keyLeftHigh = keyLeftHigh & 0x0FFFFFFFFFFFFFFF;

keyLeftHigh = keyLeftHigh | (((uint64_t) temporaryVar) << 60);

keyRightLow = keyRightLow ^ ((i & 0x01) << 15);

keyLeftHigh = keyLeftHigh ^ (i >> 1);

subKeys[i] = keyLeftHigh;// substuting key left at i place in sub keys

}

return subKeys;

}
```

```
uint64_t doPermutationOnInput(uint64_t input) {

uint64_t permutateIntVariable = 0;

for (int i = 0; i < 64; i++) {

int distance = 63 - i; //getting dist for each bit

permutateIntVariable = permutateIntVariable | ((input >> distance & 0x1) << 63 -
P[i]); // bitwise operations

}return permutateIntVariable;

}

uint16_t getKeyRightLow(char *inputKey) {

uint16_t keyRightLow = 0;
for (int i = 16; i < 20; i++)

keyRightLow = (keyRightLow << 4)

| (((inputKey[i] >= '0' && inputKey[i] <= '9') ?

(inputKey[i] - '0') : (inputKey[i] - 'a' + 10)) & 0xF);

return keyRightLow;

}
```

Encryption Process:

This process is explained in few steps. This encrypt function performs 31 rounds of encryption that includes XOR operation, substitution and permutation.

In each round,

i) **XOR with Round Key**: first the input key block is XORed with the first round key for initial diffusion. Then we convert 64-bit integer state into an byte array. Then we convert 64-bit integer state into an byte array.
ii) **Substitution**: In this process, each 4-bit nibble of the input block is substituted using an S-box value. There are eight different S-boxes, and each S-box is a permutation of the 16 possible 4-bit inputs.
iii) **Permutation**: we pass state bytes into permutation function to perform permutation. Permutation gives us diffusion, so that the S-box substitution output is spread across the entire block.

After 31 rounds, we perform XOR operation between the state and the last round subkey. Then we get final ciphertext in the hexadecimal form, that we can again convert into string.

```
char* encrypt(char *plaintext, char *inputKey) {

uint64_t *substitutionkeys = generateSubkeys(inputKey); //this fun gets us array
of sub keys

uint64_t state = hexaDecimalStringToLong(plaintext); //converting plain text
hexString into 64-bit uint64_t

for (int i = 0; i < 31; i++) {

//performing XOR operation with the state & round sub key

state = state ^ substitutionkeys[i];

//converting state from uint64_t into bytes/nibbles
byte *stateBytes = longToBytes(state);

for (int j = 0; j < 8; j++) {

//substituting state nibble bytes with S box bytes

stateBytes[j].bytesNibbleA = substitutionWithSBox(stateBytes[j].bytesNibbleA);

stateBytes[j].bytesNibbleB = substitutionWithSBox(stateBytes[j].bytesNibbleB);

}//converting bytes into uint64_t format and passing to permutation function

state = doPermutationOnInput(bytesToLong(stateBytes));//clearing the allocated memory of stateBytes pointer

delete[] stateBytes;

}

//Finally we perform XOR operation with the state & round

state = state ^ substitutionkeys[31];

//clearing the allocated memory of sub keys pointer

delete[] substitutionkeys;

return longToHexaDecimalString(state);

}
```

```
int main() {

// initialising variables

char *ciphertext;

char *plaintext = new char[17];

char *inputKey = new char[21];

strcpy(plaintext, "ffffffffffffffff"); // Assigning values

strcpy(inputKey, "ffffffffffffffffffff");

// strcpy(plaintext, "ffffffffffffffff");

// strcpy(inputKey, "00000000000000000000");

// strcpy(plaintext, "0000000000000000");
// strcpy(inputKey, "ffffffffffffffffffff");

ciphertext = encrypt(plaintext, inputKey); // calling encrypt by passing
inputs

cout << "cipher text: " << ciphertext << endl;

delete[] inputKey;

delete[] plaintext;

delete[] ciphertext;

return 0;

}
```

To sum up, the Present-80 method, which applies substitution, permutation, and XOR operations on the input block, is straightforward and effective. The round keys are then independently generated using the key generation process, and diffusion is provided by permutation tables to distribute the result of the S-box replacement over the whole block. This technique is commonly employed in devices with limited resources, such as smart cards and RFID tags.

Testing Present Algorithm:

Given input 1:

strcpy(**plaintext**, "ffffffffffffffff");
strcpy(**inputKey**, "ffffffffffffffffffff");

Output 1

cipher text: 3333dcd3213210d2

Given input 2

strcpy(plaintext, "0000000000000000");
strcpy(inputKey, "ffffffffffffffffffff");

Output 2

cipher text: e72c46c0f5945049

Given input 3

strcpy(plaintext, "ffffffffffffffff");
strcpy(inputKey, "00000000000000000000");

Output 3

cipher text: a112ffc72f68417b

Note: All inputs are kept commented in the code, please verify by uncommenting one by one.

Comparison of AES-128 and Present-80 Algorithms

AES and Present algorithms are used for encryption and decryption of data. But they are different in terms of input size, block size, key size and the internal logic.

	AES-128	Present-80

Input/output size	Input = 128 bits
	cipher = 128 bits
		Input = 64 bits
		
		Cipher = 64bits

Block size	block size = 128 bits Block size for AES is greater than Present since it has 128 bits. So here AES algorithm can encrypt more data than present algorithm which is more efficient and useful for most of the software applications that needs high throughput. *Even I once used AES encryption in Android development*	block size = 64 bits Since the block size is small for present algorithm, it can be only used in small applications where input data is also small.
Key size	key size= 128 /192/ 256 bits Since AES has more key sizes, it is highly secure against brute force attacks.	key size = 80/128 bits present algorithm is secure against differential and linear attacks
No of rounds	AES has different rounds based on the input bit size. AES has 10/12/14 rounds for 128 /192/ 256 bits key size	Present algorithm undergoes 31 rounds
Algorithm design	AES uses SPN substitution permutation network design, which performs substitution and permutation on input data	Present is light weight design SPN which does repeat basic round functionality on the input data.
Execution time	Execution time of AES algorithm will be little slower when compared to present as it has more business logic behind it.	Execution time of present algorithm is faster than AES.

usage	It is used for high end software applications and for hardware. example: Android and iOS application	This is mostly used on small hardware applications.
Security level	0.70	0.84
Applications	AES used in mobile applications, cloud storage, databases, mostly used for file encryptions which I personally did.	Used in RFID technology, IoT, mobiles, tablets, smart home products and any lightweight applications.