AES Implementation in C++

Dhaval Patel, Prajeet Bhavsar, Mit Patel

***Abstract*— Ever since the evolution of cryptography, there has been a constant race to develop the perfect cipher. But with time, vulnerabilities are found in most of the cipher as the processing power of computers increase. When we say most, AES is among the few which is till date secure cipher in practice. Although there have been numerous papers on theoretical attacks on AES, none of them can be extended to practical implementation mainly because of lack of computational power. In this paper, we have discussed on how we implemented AES and along the journey what are the various crypto coding practices we learned and implemented in our project.**

***Index Terms*—Galois Field, Cryptanalysis, Side Channel Attacks, Confusion, Diffusion**

I. INTRODUCTION

AES is by far the most popular cipher in the crypto world. This is not because AES is very complex. In fact AES's core implementation could not be more simpler than that. IT involves two core principles of crypto: Confusion and Diffusion. Ever since we googled how was AES chosen, we were intrigued to know more details about the competition that NIST organized back in early 1997. As soon as the call went out to design a new cipher(because significant vulnerabilities were found in DES even including the complete key recovery), cryptographers all around the world started designing their algorithms for submission. NIST had in total 15 submissions in August of 1998 for round 1. After analyzing the algorithms and considering the comments received from Crypto community about the submissions, 5 out of 15 namely MARS, RC6, Rijndael, Serpent, and Twofish were selected for round 2. At the end of round 2 analysis, Rijndael was crowned victor and named AES in February 2001. The Advanced Encryption Standard (AES) was published as FIPS 197 on November 26, 2001[1]. Validation testing for conformance of AES implementations to FIPS 197 then began under the Cryptographic Algorithm Validation Program.

Our implementation for AES includes all the three variants of AES i.e. AES-128. AES-192 and AES-256. We have followed the FIPS 197 AES standard. We have used AES as a block cipher which encrypts individual blocks of input messages of length 128 bits. We first read a file and divide the input text into blocks of 128. Now according to the mode of operation used, the implementation changes. After the encryption is done, we obtain the output cipher in blocks of 128 bits which is then appended and written to file. Thus in this way input file is encrypted. The implementation details are mentioned in the next section. The implementation involves 2 modes of operations. We learned how the modes of operation are implemented and had actual practical implementation experience. We also noted an important fact that AES should not be used Electronic Codebook Mode(ECB). But we have used ECB as a building block to implement Cipher Block Chaining(CBC) since ECB is easy to implement. We learnt several coding practices which helped us implement AES according to crypto coding standards.

The most interesting things that we found during our project were 1) How the S Box value is calculated 2) How do we generate pseudo random numbers in C++ 3) How confusion and diffusion takes place in AES 4) How various kinds of attacks work on AES and how do we prevent them. Along with these, we learnt so many crypto coding standards which we have followed in our code too.

The paper is organized as follows: Section 2 discusses the implementation description of the algorithm, Section 3 describes various attacks on AES and precautions we have taken in our code to prevent such attacks, Section 4 showcases the coding practices we learnt and implemented in our code and finally Section 5 summarizes our work followed by references.

II. IMPLEMENTATION DESCRIPTION

There are three variants of AES depending on key length i.e 128, 192 or 256 bits. The basic processing unit in AES is bytes. Here, the internal operations during encryption and decryption are performed on 2 D array of bytes which is called the State. The term **Nb** is defined as the input/output block size divided by 32. So the State will consist of four rows of bytes and Nb columns. And the term **Nk** represents the number of 32-bit words in the cipher key K. Also, the number of rounds is represented by **Nr**. The Key-Block-Round combination according to the standards is mentioned in the following table [6]:

	Key Length (Nk words)	Block Size (Nb words)	Number of rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

In order to transform the plain text to cipher text, the following byte oriented operations are performed Nr number of times:

- Byte substitution using a table (S-box)
- Shift rows of the state array
- Mixing the column's data of state array
- Adding round key to the state array

Whereas during decryption (Inverse Cipher), the inverse of these transformations is performed.

A. Cipher

Here in each round, the four operations **SubBytes()**, **ShiftRows()**, **MixColumns()**, and **AddRoundKey()** are

performed. All the rounds are same except the last one, which doesn't include the MixColumns() operation. Initially, the plaintext is stored in the State array and an initial round key is added to it.

1) SubBytes():

This bytes substitution transformation is a non-linear substitution done for each byte individually using a substitution table (S box). **S box** is a 16 by 16 2-D array lookup table where the row indicates the leftmost 4 bits and columns indicate the rightmost 4 bits. This is how we update the State by substituting the bytes with corresponding bytes from the S box table.

2) ShiftRows():

In this transformation, the bytes in the last three rows of the State are cyclically shifted over by different offsets. The first row is not shifted, then the following second, third and fourth rows of the State are cyclically shifted by an offset of 1, 2 and 3 respectively.

3) MixColumns():

The MixColumns() transformation multiplies the State with a fixed matrix. The columns are considered as polynomials over GF(2^8). In this fixed matrix, the elements of the second, third and fourth rows are obtained by shifting the elements of the first row by an offset of 1, 2 and 3 respectively.

4) AddRoundKey():

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation.

B. Key Expansion

The key expansion routine generates a key schedule from cipher key K. It generates a total of Nb * (Nr + 1) words of four bytes in length. The resulting key schedule is an array of 4-byte words and each of these 4 byte words is denoted by $[w_i]$ where $0 \leq i < Nb * (Nr + 1)$. We have generated the key schedule in the following way: Let **SubWord()** be a function that applies S-box to a 4-byte input word byte by byte. The function **RotWord()** takes a word [a0,a1,a2,a3] as input and returns [a1,a2,a3,a0] i.e. it perform a cyclic shift of offset 1.The round constant word array, **Rcon[i]**, contains [x^{i-1},{00},{00},{00}], where x is denoted as {02} in the field GF(2^8). Now the cipher key is expanded such that the first Nk words of the expanded key are same as the Cipher Key. Every following word, w[i], is the XOR of the previous word, w[i-1], and the word that is Nk positions earlier, w[i-Nk]. Now for words at indices that are multiple of Nk, a particular transformation is applied to w[i-1] prior to the XOR, followed by an XOR with Rcon[i] [6]. In this transformation, the bytes in a word are shifted cyclically with an offset of 1, followed by the S-box lookup of the shifted 4-byte word. This can be described mathematically as:

$$[w_i] = [w_{i-Nk}] \oplus (\text{SubWord}(\text{RotWord}([w_{i-1}])) \oplus \text{Rcon}[i / Nk])$$

However, when key cipher key is 256-bits long, the key expansion routine is somewhat different. Here, if i-4 is a multiple of Nk, then SubWord() is applied to w[i-1] prior to the XOR.

C. Inverse Cipher

To obtain plain text from cipher, the cipher transformations are inverted and then implemented in reverse order. The individual transformations are **InvShiftRows()**, **InvSubBytes()**, **InvMixColumns()** and **AddRoundKey()**.

1) InvShiftRows():

The InvShiftRows() transformation is the inverse of the ShiftRows(). The last three rows' bytes are cyclically shifted for a certain offset. The first row bytes of the State are not shifted. The second, third and fourth rows are shifted by an offset of 1, 2 and 3 respectively.

2) InvSubBytes():

In this transformation, the inverse S-box is applied to every byte of the State. This is obtained by applying the inverse of the affine transformation followed by taking the multiplicative inverse in GF(2^8). We have simply stored the inverse values of the S box values to obtain the inverse S box table.

3) InvMixColumns():

This transformation is the inverse of InvMixColumns(). The columns of the State are considered as polynomials over GF(2^8) and it is multiplied with a fixed matrix.

4) Inverse of AddRoundKey():

Since AddRoundKey() is simply an XOR operation, it is its own inverse.

D. Modes of Operation

For our AES block cipher, we have implemented two modes of operation namely Electronic Codebook Mode(ECB) and Cipher Block Chaining (CBC).

1) Electronic Codebook Mode(ECB):

As we have already mentioned, the ECB mode is not advisable to use with AES. The main problem with ECB mode is that encrypting the same block of plaintext will always yield the same block of cipher text. Thus an adversary can easily detect whether two ECB encrypted messages are identical or not. Thus we haven't used ECB as mode of operation but we have just used it as a base for CBC.

2) Cipher Block Chaining(CBC):

The most important part while implementing this mode of operation with AES was to generate random numbers. We felt this was quite fun because we already knew that the inbuilt C rand() function will not give us pseudo random

numbers mainly because they have short cycle and so the number s can be predictable after some time. The easy solution was to use distribution function for the random numbers. C++ standard library provides mechanism for fine grained control over pseudo random number generation[2]. The process consists of two parts: Firstly the engine which is responsible to provide random numbers and Second part is responsible for the distribution of random values through the density function. In our implementation, we have used Mersenne Twister algorithm as the core engine to generate random numbers and a uniform distribution.

Once we get the pseudo random number, we use it as IV in the first block and then we have implemented the normal CBC. The cipher obtained through CBC mac is semantically secure so the adversary has less probability of breaking it by just looking at the cipher. This concludes our observations on Modes of operations and now we discuss some features of AES that we found are very interesting.

Although the algorithm implementation is pretty straight forward once the details are clear, we found following features that intrigued us.

E. Galois Field

To understand the S boxes of AES, we need to understand the finite field arithmetic. The number of elements in finite field is always of the form p^n where p is a prime number and n is a positive integer. The prime p is called the characteristic of the field, and the positive integer n is called the dimension of the field over its prime field. Galois field is a finite field with p^n elements and is denoted by $GF(p^n)$.
Elements of $GF(p^n)$ may be represented as polynomials of degree strictly less than n over GF(p). Operations are then performed modulo R where R is an irreducible polynomial of degree n over GF(p), for instance using polynomial long division. The addition of two polynomials P and Q is done as usual; multiplication may be done as follows: compute $W = P \cdot Q$ as usual, then compute the remainder modulo R

AES uses the characteristic 2 finite field with 256 elements, that can also be represented as $GF(2^8)$ with the reduction polynomial as $x^8 + x^4 + x^3 + x + 1$

F. Multiplicative Inverse:

The multiplicative inverse is needed to compute the S box value for a given byte. The Extended Euclidean Algorithm is used in AES to compute the multiplicative inverse.

Now to calculate the S box value, we illustrate using an example. S[b] is first converted to polynomial representation. Then the multiplicative inverse of polynomial is calculated. Next the result is modulo with the reduction polynomial of AES. And finally affine transformation takes place to produce the final output. This output is the S Box value for b.

G. Confusion and Diffusion(Principles of Crypto)

To understand the true power of AES, we need to understand how the three basic ideas of crypto are incorporated in it.

1) Confusion:

The confusion in AES takes place in the S box where the relationship between each byte of message is obscured. Each and every byte of message is mapped to a completely different byte.

2) Diffusion:

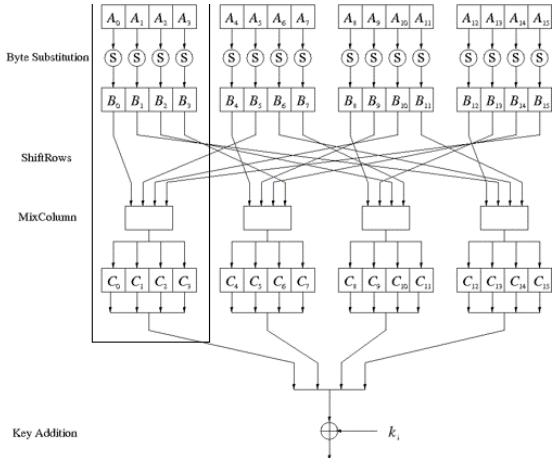
Diffusion in AES is implemented in two separate parts. The first part is when the shifting of rows occurs and secondly when the columns are mixed in the next stage after shifting rows.

3) Secrecy only in the Key:

Since AES's internal working is public, the secrecy remains only in the key chosen for encryption. At the end of each round, the round key is XORed with the output from last stage. In this way key secrecy is implemented in AES.

H. Change of one bit in input message and its propagation in the system

Suppose we flip one bit in one specific byte of message. And lets assume that on average, 4 bits are flipped in the output of S Box. Now the beauty of Diffusion in AES is that the 4 bit change in S box value will be propagated through all the 32 bits output value of Mix Column stage. This is very strong diffusion because with only one bit flip in input message results in 32 bit flips in output of Mix Columns stage. So within one round the quarter of the whole data pass is affected. So by the time all rounds of AES are completed, the change will have



AES round function for rounds 1,2,...,nr−1

integrated among numerous bytes. Above is a figure showing the same process.

I. *Change in cipher when one bit is flipped the key*

Now for CBC mode, one flipped bit in key is not relevant because every time new random IV is chosen and so IV ⊕ m_i will change. So the cipher will change irrespective of change in the key.

Even if the mode of operation is ECB, cipher will always look completely random with one bit flipped in the key. Here's an example when we tried the same in our implementation.

m = Hello to the AES Encryption!

K_1 = 0x2c, 0x7e, 0x15, **0x16**, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c

K_2 = 0x2c, 0x7e, 0x15, **0x06**, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c

The hexadecimal representation of both ciphers is

C_1 = 72 b7 16 d1 be e3 ce b1 93 23 63 80 78 c9 74 fd 66 2b a2 82 4b 74 a1 d5 ed 7b ae 55 0e 97 1b b6

C2 = f9 03 00 ae 37 f9 24 16 ff 5e a0 c2 89 49 f4 b8 70 41 a5 ac 8f 7f 3f a2 91 66 d4 a4 5a d0 8b a9

The Unicode representation of both ciphers is

C_1 = r· Ñ¾ãÎ±'‘#c€xÉtýf+¢,Kt¡Õí{®U —¶

C_2 = C글路ᄃ廿숖駟룷穏겔彩ℂ暑T긂ლ3

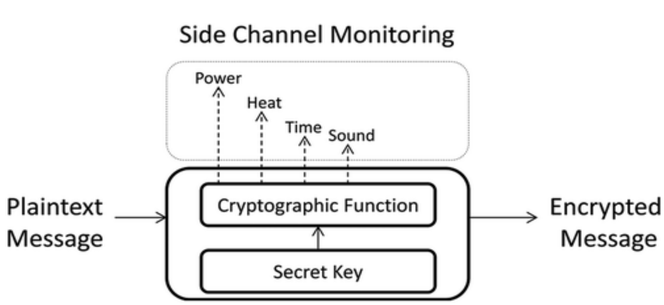
As seen from the ciphers, they seem to be completely random which is what we want.

III. CRYPTO LEARNING

As far as we know, there have been side channel attacks on implementation of AES, but there aren't any practical cryptanalytic attacks against the algorithm itself. First we discuss the side channel attacks.

A. *Side Channel Attacks.*

These types of attacks mainly exploit any vulnerabilities in the implementation of algorithm. Various factors like timing information, sound, power usage or electromagnetic leaks can leak some information which can be used by adversary to break the cipher.



Side channel monitoring

1) *Cache Timing Attacks*

The recently accessed data is automatically saved in a limited size cache. Now input dependent timing attack can exploit this property because reading from cache will take less time than reading from an non-cached block of memory. For ex. If S_0 [0] is cached and S_0 [200] is not cached, then reading S_0 [b] will take less time for b = 0 and more time for b = 200 leaking information about b.

To avoid this attack, the most intuitive way is to cache all the S boxes throughout the implementation of AES. However there are several points to consider:

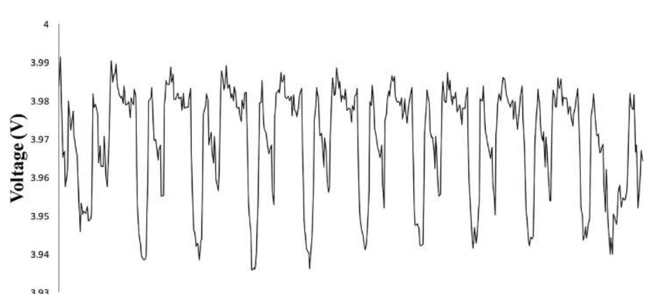
- The AES S-box lines may be removed from cache to make space for memory lines used by computations other than AES.
- Caching all S boxes is very expensive and some lines may be removed from cache to make room for AES computation itself.

In order to avoid such an attack, another possible solution is to calculate the S-Box value on the fly i.e. whenever it is needed, we compute the S-Box value.
We have used predefined S Boxes for our implementation as we didn't feel the need to calculate the value on the fly mainly because to carry out the above described cache timing attack, significant amount resources would be needed especially to store all the access times each time a S box value is needed. This was not possible in our use case.

2) *Power Analysis Attacks on AES-128*

These types of attacks are carried out by inspecting the power usage on a cryptographic system[3]. For these attacks to succeed, there has to be a correlation between the power consumed and cryptographic operations performed. There are 3 main categories of Power analysis namely Single Power Analysis (SPA), Differential Power Analysis (DPA) and Correlation Power Analysis (CPA).

SPA can be useful for gathering information about the type of algorithm. To provide an example of SPA, the figure below presents a power trace captured from an Arduino Uno as it runs a single AES-128 cryptographic operation. In AES-128, there are 10 rounds of operation which can be identified as the 10 nadirs in the figure presented. Thus, although one is unable to deduce the secret key using this technique, it does present the capability to identify the cryptographic algorithm running on device and enable more powerful attacks which specifically exploit any weakness of an algorithm to take place.



Simple power analysis on AES-128

An example of a more power attack is DPA. This attack makes use of statistical techniques to identify *differences* in power traces, thus revealing data leakage which may result in the correct secret key being guessed. CPA, on the other hand, aims to reveal data leakage by finding relationships between characteristics of power traces and a hypothesized power model. If correlation is found between these two variables, one has the capability to predict the correct secret key if enough power traces are gathered.

To avoid such attacks, much can be done from hardware side like optimizing the transistors and all. But as far as countermeasures at program level are concerned, insertion of dummy instructions is possible in any algorithm while changing the order of operations may be easier for some algorithms than for others. [4] This countermeasure is particularly efficient against higher-order DPA/DEMA as there the signals must be aligned in multiple places and any misalignment severely limits the effectiveness of attack. Also use instructions that are generally processed in constant time like AND, XOR, SHIFT, ROTATE, ADD which are guaranteed to complete in constant time. This is what we have implemented in our project too. Using the mentioned instructions, we can prevent this types of attacks to some level.

B. *Cryptanalytic Attacks.*

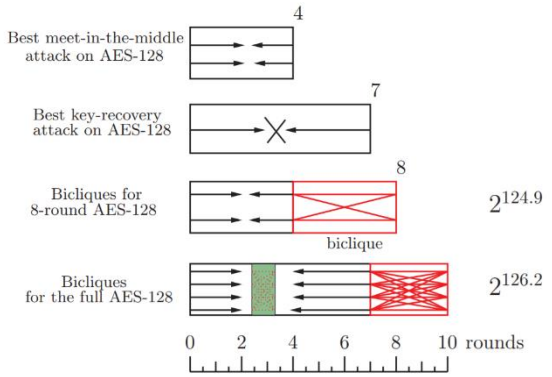
Now moving onto cryptanalysis, by far the only technique to fully break AES is called biclique attack. The important thing to mention here is that this doesn't mean that AES is no longer secure. The biclique attack is slightly better than the brute force attack. We will briefly discuss the biclique attack but first we discus meet in the middle attack.

1) *Meet in the middle Attacks*

The best known meet in the middle attack has been able to crack the cipher but only till 8 rounds[5]. The idea is that one entry of the cipher text after 4 rounds of AES encryption is expressed with only 15 or 13 bytes respectively based on the methodology used. In [5] they have extended this idea to 7 rounds of AES-128, AES-192 and 8 rounds in AES-256. The said attack on AES-256 is more beneficial than existing attacks in terms of time complexity. But none of the MITM attacks can be extended to all the rounds of AES. This is where bicliques attacks come into picture.

2) *Biclique Attack.*

This attack basically extends the best known MITM for AES to all its rounds. As discussed above, the best key recovery attack on AES-128 can only work for 7 rounds. Biclique attacks extend that to full 10 rounds in case of AES-128.[5]



Biclique on AES-128

The Biclique attack on other variants of AES yields the following results.

rounds	computations	data	memory	success prob.
AES-128 secret key recovery				
10 (full)	$2^{126.18}$	2^{88}	2^{8}	1
AES-192 secret key recovery				
12 (full)	$2^{189.74}$	2^{80}	2^{8}	1
AES-256 secret key recovery				
14 (full)	$2^{254.42}$	2^{40}	2^{8}	1

Biclique Attack on Variants of AES

Although the cryptanalysis attack have proven theoretically that they can recover the whole key for all variants of AES for full rounds, they are no where close to practical implementations. The best possible odds after brute force are shown above for fully fledged AES (all variants). As it is seen from the picture, these attacks only shave off a couple of bits from the actual key and thus are slightly good than brute force.

IV. SECURE CODING

After studying the SEI secure coding practices we came across several such practice that could be implemented in our project.

For all the loops in our implementation, we have taken care of the following secure coding practices.

- CTR50-CPP. Guarantee that container indices and iterators are within the valid range. In our implementation, we have ensured that the array and vector references are within their bounds and not exceeding them
- CTR52-CPP. Guarantee that library functions do not overflow. We have used the vector library function and made sure that the data being copied to it is large enough to hold the data.
- CTR53-CPP. Use valid iterator ranges. While iterating over elements such as an array or a vector, the iteration range must be valid.

For declaration and initialization of variables and arrays in our project following practices were followed.

- DCL51-CPP. Do not declare or define a reserved identifier
- DCL60-CPP. Obey the one-definition rule. This rule states that when the programs are divided into multiple translation units that are later linked together to form an executable, C++ restricts named object definitions, i.e. there should be a single definition for an object across all translation units.

We also followed certain secure coding practices that had direct relation to cryptography. Following are such practices.

- Use unsigned bytes to represent binary data
- MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers. The numbers generated by std::rand() have a relatively short cycle and hence the numbers can be predictable. For our AES implementation, a pseudorandom generator technique is discussed in section II.
- MSC51-CPP. Ensure your random number generator is properly seeded. If we call a pseudorandom generator in the same initial state without explicit seeding or with a constant seed value, then the resulting sequence of random numbers in different runs is identical.
- MSC52-CPP. Value-returning functions must return a value from all exit paths. A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior.

Also for reading the input, we have used files. And we have taken care of the following practice while handling files.

- FIO51-CPP Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination.
- ERR50-CPP. Do not abruptly terminate the program

V. SUMMARY

So far the project was quite fun and challenging to some extent. Our learning began when we were introduced to Galois Field and it continued till the secure coding practices. Although the AES algorithm was quite easy to understand at a higher level, later when the implementation began we uncovered the hidden intricacies. And we must admit that the final result was quite satisfactory.

As of now, AES is considered to be the most secure encryption algorithm. But the same was true when its predecessors (DES) were introduced. So who knows, maybe tomorrow or 10 years down the line, AES may not remain the most secure encryption cypher and once again we might need to rely on the brilliant minds of cryptographers to design a new secure cypher.

VI. REFERENCES

[1] Cryptographic Standards and Guidelines: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development

[2] SEI Cert Coding Standards: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

[3] Side channel attacks and their counter measures: https://keccak.team/files/NoteSideChannelAttacks.pdf

[4] Demirci, Hüseyin & Taskin, Ihsan & çoban, Mustafa & Baysal, Adnan. (2009). Improved Meet-in-the-Middle Attacks on AES. 144-156. 10.1007/978-3-642-10628-6_10.

[5] Biclique Cryptanalysis of the Full AES: https://pdfs.semanticscholar.org/ddda/c77fbb1a8c72ef90e1898bfda125c1ac0c0e.pdf

[6] AES Standard Documentation: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

APPENDIX A

Main.cpp

```
/*  RULE MSC52-CPP.
 *      Value-returning functions must return a value from all exit paths
 *  CTR50-CPP. Guarantee that container indices and iterators are within the valid range
 *  CTR52-CPP. Guarantee that library functions do not overflow
 *  CTR53-CPP. Use valid iterator ranges
 *  CTR55-CPP. Do not use an additive operator on an iterator if the result would
overflow
 *  MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers
 *  MSC51-CPP. Ensure your random number generator is properly seeded
 *  MSC52-CPP. Value-returning functions must return a value from all exit paths
 *  OOP58-CPP. Copy operations must not mutate the source object
 *  DCL51-CPP. Do not declare or define a reserved identifier
 *  DCL55-CPP. Avoid information leakage when passing a class object across a trust
boundary
 *  DCL60-CPP. Obey the one-definition rule
 *  CRYPTO CODING PRACTICE
 *      Use unsigned bytes to represent binary data
 */

#include "pch.h"
#include "aes.h"
#include "modes.h"
#include <iostream>
#include <vector>
#include <iomanip>
#include <fstream>
#include <iterator>
#include <random>

using namespace std;

vector<uint8_t> readFile(string fileName);
void writeVecToFile(string fileName, vector<uint8_t> data);

int main()
{
    int keyLength = 128;
    int bytes = keyLength / 8;
    modes mode;

    // Initialize the key
    uint8_t key_arr[32] = { 0x2c, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
                            0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c,
```

```
                            0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
                            0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
    std::vector<uint8_t> key;

    //  OOP58-CPP. Copy operations must not mutate the source object
    for (int i = 0; i < bytes; i++) {
        key.push_back(key_arr[i]);
    }

    // Encryption
    vector<uint8_t> data = readFile("Capture.png");   // Read the plain text from a file
    vector<uint8_t> cipherText = mode.cbcEncrypt(data, key);
    writeVecToFile("sec.png", cipherText);          // Storing the resultant cipher text

    // Decryption
    vector<uint8_t> cipher = readFile("sec.png");   // Read the cipher text from a file
    vector<uint8_t> plainText = mode.cbcDecrypt(cipher, key);
    writeVecToFile("real.png", plainText);  // Store the decryption result in a file

    return 0;
}
/*  RULE : FIO51-CPP
 *  Failing to properly close files may allow an attacker to exhaust system resources and
can increase
 *  the risk that data written into in-memory file buffers will not be flushed in the
event of abnormal
 *  program termination.
 */

// Write the given byte sequence in vector to a file
void writeVecToFile(string fileName, vector<uint8_t> data)
{
    ofstream fout;
    fout.open(fileName, ios::binary | ios::out);
    fout.write(reinterpret_cast<const char*>(&data[0]), data.size() * sizeof(uint8_t));
    fout.close();
}
// Read a file as a vector byte sequence
vector<uint8_t> readFile(string fileName)
{
    vector<uint8_t> bytes;
    ifstream file1(fileName, ios_base::in | ios_base::binary);
    uint8_t ch = file1.get();
    while (file1.good())
    {
        bytes.push_back(ch);
        ch = file1.get();
    }
```

```
    return bytes;
}
```

Modes.h

```
// Variables and Function declarations for modes.cpp
#include "aes.h"
#include <iostream>
#include <vector>
#include <iomanip>

class modes
{
public:
    modes();
    std::vector<uint8_t> ecbEncrypt(std::vector<uint8_t> message, std::vector<uint8_t>
key);
    std::vector<uint8_t> ecbDecrypt(std::vector<uint8_t> cipher, std::vector<uint8_t>
key);
    std::vector<uint8_t> cbcEncrypt(std::vector<uint8_t> message, std::vector<uint8_t>
key);
    std::vector<uint8_t> cbcDecrypt(std::vector<uint8_t> cipher, std::vector<uint8_t>
key);
    ~modes();
private:
    std::vector<std::vector<uint8_t>> pad(std::vector<uint8_t> message);
    std::vector<uint8_t> removePads(std::vector<std::vector<uint8_t>> blocks);
    std::vector<uint8_t> blockXOR(std::vector<uint8_t> a, std::vector<uint8_t> b);
    std::vector<uint8_t> generateRandomBlock();
    bool isValidKey(int length);
};
```

Modes.cpp

```
#include "pch.h"
#include "modes.h"
#include "aes.h"
#include <random>

modes::modes()
{
}

/*
 *  MSC52-CPP.
```

```
 *      Value-returning functions must return a value from all exit paths
 *  CTR50-CPP.
 *      Guarantee that container indices and iterators are within the valid range.
 *      All the vectors are accessed after finding their range.
 *  CTR52-CPP.
 *      Guarantee that library functions do not overflow
 *  CTR53-CPP.
 *      Use valid iterator ranges
 */

// Encryption using ECB Mode
std::vector<uint8_t> modes::ecbEncrypt(std::vector<uint8_t> message, std::vector<uint8_t>
key)
{
    int len = key.size() * 8;
    if (!isValidKey(len)) {
        std::cout << "ERROR" << std::endl;
        return message;
    }
    AES aes(key);
    std::vector<std::vector<uint8_t>> blocks = pad(message); // Padding the message
    std::vector<uint8_t> cipher;
    int totalBlocks = blocks.size();
    for (int i = 0; i < totalBlocks; i++) {
        std::vector<uint8_t> block = blocks[i];
        std::vector<uint8_t> out = aes.encrypt(block);
        for (int j = 0; j < 16; j++) {
            cipher.push_back(out[j]);
        }
    }
    return cipher;
}

/*
 *  MSC52-CPP.
 *      Value-returning functions must return a value from all exit paths
 */

// Decryption using ECB Mode
std::vector<uint8_t> modes::ecbDecrypt(std::vector<uint8_t> cipher, std::vector<uint8_t>
key)
{
    int len = key.size() * 8;
    if (!isValidKey(len)) {
        std::cout << "ERROR" << std::endl;
        return cipher;
    }
    AES aes(key);
```

```
    int totalBlocks = cipher.size() / 16;
    std::vector<std::vector<uint8_t>> blocks;
    for (int i = 0; i < totalBlocks; i++) {
        std::vector<uint8_t> block(16);
        for (int j = 0; j < 16; j++) {
            block[j] = cipher[(i * 16) + j];
        }
        blocks.push_back(block);
    }
    std::vector<std::vector<uint8_t>> plainTextBlocks;
    for (int i = 0; i < totalBlocks; i++) {
        uint8_t in[16];
        std::vector<uint8_t> block = blocks[i];
        for (int j = 0; j < 16; j++) {
            in[j] = block[j];
        }
        std::vector<uint8_t> out = aes.decrypt(block);
        std::vector<uint8_t> plainBlock(16);
        for (int j = 0; j < 16; j++) {
            plainBlock[j] = out[j];
        }
        plainTextBlocks.push_back(plainBlock);
    }
    std::vector<uint8_t> plainText = removePads(plainTextBlocks); // Removing the padding
    return plainText;
}

/*
 *  MSC52-CPP.
 *      Value-returning functions must return a value from all exit paths
 */

// Encryption using CBC Mode
std::vector<uint8_t> modes::cbcEncrypt(std::vector<uint8_t> message, std::vector<uint8_t>
key)
{
    int len = key.size() * 8;
    if (!isValidKey(len)) {
        std::cout << "ERROR" << std::endl;
        return message;
    }
    AES aes(key);
    std::vector<std::vector<uint8_t>> blocks = pad(message);
    std::vector<std::vector<uint8_t>> cipherBlocks;
    std::vector<uint8_t> IV = generateRandomBlock();
    cipherBlocks.push_back(IV);
    int totalBlocks = blocks.size();
    for (int i = 0; i < totalBlocks; i++) {
```

```
        std::vector<uint8_t> block = blocks[i];
        std::vector<uint8_t> out = aes.encrypt(blockXOR(block, cipherBlocks[i]));
        cipherBlocks.push_back(out);
    }
    std::vector<uint8_t> cipher;
    for (int i = 0; i < totalBlocks+1; i++) {
        std::vector<uint8_t> block = cipherBlocks[i];
        for (int j = 0; j < 16; j++) {
            cipher.push_back(block[j]);
        }
    }
    return cipher;
}

/*
 *  MSC52-CPP.
 *      Value-returning functions must return a value from all exit paths
 */

// Decryption using CBC Mode
std::vector<uint8_t> modes::cbcDecrypt(std::vector<uint8_t> cipher, std::vector<uint8_t>
key)
{
    int len = key.size() * 8;
    if (!isValidKey(len)) {
        std::cout << "ERROR" << std::endl;
        return cipher;
    }
    AES aes(key);
    int totalBlocks = cipher.size() / 16;
    std::vector<std::vector<uint8_t>> blocks;
    for (int i = 0; i < totalBlocks; i++) {
        std::vector<uint8_t> block(16);
        for (int j = 0; j < 16; j++) {
            block[j] = cipher[(i * 16) + j];
        }
        blocks.push_back(block);
    }
    std::vector<uint8_t> IV = blocks[0];
    std::vector<std::vector<uint8_t>> plainTextBlocks;
    for (int i = 1; i < totalBlocks; i++) {
        std::vector<uint8_t> block = blocks[i];
        std::vector<uint8_t> out = blockXOR(aes.decrypt(block), blocks[i-1]);
        plainTextBlocks.push_back(out);
    }
    std::vector<uint8_t> plainText = removePads(plainTextBlocks);
    return plainText;
}
```

```
modes::~modes()
{
}
// Function for padding the message to make it a multiple of block length
std::vector<std::vector<uint8_t>> modes::pad(std::vector<uint8_t> message)
{
    int len = message.size();
    int fullBlocks = len / 16;
    std::vector<std::vector<uint8_t>> blocks;
    for (int i = 0; i < fullBlocks; i++) {
        std::vector<uint8_t> block(16);
        for (int j = 0; j < 16; j++) {
            block[j] = message[(i * 16) + j];
        }
        blocks.push_back(block);
    }
    int remainingBytes = 16 - (len - (fullBlocks * 16));
    std::vector<uint8_t> block;
    for (int i = (fullBlocks * 16); i < len; i++) {
        block.push_back(message[i]);
    }
    remainingBytes = remainingBytes == 0 ? 16 : remainingBytes;
    for (int i = 0; i < remainingBytes; i++) {
        block.push_back(remainingBytes);
    }
    blocks.push_back(block);
    return blocks;
}
// Remove padding
std::vector<uint8_t> modes::removePads(std::vector<std::vector<uint8_t>> blocks)
{
    int noOfBlocks = blocks.size();
    std::vector<uint8_t> message;
    for (int i = 0; i < noOfBlocks - 1; i++) {
        std::vector<uint8_t> block = blocks[i];
        for (int j = 0; j < 16; j++) {
            message.push_back(block[j]);
        }
    }
    std::vector<uint8_t> block = blocks[noOfBlocks - 1];
    int len = 16 - block[15];
    for (int i = 0; i < len; i++) {
        message.push_back(block[i]);
    }
    return message;
}
```

```
std::vector<uint8_t> modes::blockXOR(std::vector<uint8_t> a, std::vector<uint8_t> b)
{
    std::vector<uint8_t> ans(16);
    for (int i = 0; i < 16; i++) {
        ans[i] = a[i] ^ b[i];
    }
    return ans;
}

/*
 *  RULE MSC50-CPP
 *      Do not use std::rand() for generating pseudorandom numbers
 *  RULE MSC51-CPP
 *      Ensure your random number generator is properly seeded
 */

// Generating the pseudo random IV
std::vector<uint8_t> modes::generateRandomBlock()
{
    std::vector<uint8_t> randomBlock(16);
    std::uniform_int_distribution<int> distribution(0, 255);
    std::random_device rd;
    std::mt19937 engine(rd());
    for (int i = 0; i < 16; i++) {
        randomBlock[i] = distribution(engine);
    }
    return randomBlock;
}

bool modes::isValidKey(int length)
{
    if (length == 128 || length == 192 || length == 256)    return true;
    return false;
}
```

AES.h

```
#include <stdint.h>
#include <vector>

class AES
{
public:
    AES(std::vector<uint8_t> key);
    std::vector<uint8_t> encrypt(std::vector<uint8_t> in);
    std::vector<uint8_t> decrypt(std::vector<uint8_t> in);
    ~AES();
```

```
private:
    int Nk;
    int Nb;
    int Nr;
    std::vector<std::vector<uint8_t>> rcon;
    std::vector<std::vector<uint8_t>> expandedKey;
    // S Box tables
    const uint8_t S_BOX[16][16] = {
        {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}
    };
    const uint8_t INV_S_BOX[16][16] = {
        {0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb},
      {0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb},
      {0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e},
      {0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25},
      {0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92},
      {0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84},
      {0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06},
      {0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b},
      {0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73},
      {0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e},
      {0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b},
      {0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4},
      {0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f},
      {0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef},
      {0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61},
      {0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d}
    };

    std::vector<std::vector<uint8_t>> keyExpansion(std::vector<uint8_t> key);
    std::vector<uint8_t> rotWord(std::vector<uint8_t> a);
    std::vector<uint8_t> RCON(int i);
    std::vector<uint8_t> wordXOR(std::vector<uint8_t> a, std::vector<uint8_t> b);
```

```
    std::vector<uint8_t> subWord(std::vector<uint8_t> a);
    void addRoundKey(uint8_t state[][4], int round, std::vector<std::vector<uint8_t>>
key);
    void subBytes(uint8_t state[][4]);
    void shiftRows(uint8_t state[][4]);
    void mixColumns(uint8_t state[][4]);
    void invSubBytes(uint8_t state[][4]);
    void invShiftRows(uint8_t state[][4]);
    void invMixColumns(uint8_t state[][4]);
    uint8_t multiply(uint8_t a, uint8_t b);
    std::vector<std::vector<uint8_t>> create2Darray(int row, int col);
    std::vector<uint8_t> createWord(uint8_t a, uint8_t b, uint8_t c, uint8_t d);

};
```

AES.cpp

```
#include "pch.h"
#include "AES.h"
#include <iostream>
#include <iomanip>
#include <stdint.h>
#include <vector>


/*
 *  OOP53-CPP. Write constructor member initializers in the canonical order
 */
// Initializing the AES parameters
AES::AES(std::vector<uint8_t> key)
{
    int len = key.size() * 8;
    switch (len)
    {
    case 128:
        Nk = 4;
        Nb = 4;
        Nr = 10;
        break;
    case 192:
        Nk = 6;
        Nb = 4;
        Nr = 12;
        break;
    case 256:
```

```
        Nk = 8;
        Nb = 4;
        Nr = 14;
        break;
    default:
        std::cout << "error";
        break;
    }
// Initializing Rcon for key expansion
    rcon.push_back(createWord(0x00, 0x00, 0x00, 0x00));
    rcon.push_back(createWord(0x01, 0x00, 0x00, 0x00));
    uint8_t x = 0x01;
    for (int i = 2; i < Nb*(Nr + 1); i++) {
        x = multiply(x, 0x02);
        rcon.push_back(createWord(x, 0x00, 0x00, 0x00));
    }
    expandedKey = keyExpansion(key); // Key Expansion

}
/*
 *  RULE OOP58-CPP.
 *      Copy operations must not mutate the source object.
 *  CTR52-CPP.
 *      Guarantee that library functions do not overflow.
 */
// Encryption
std::vector<uint8_t> AES::encrypt(std::vector<uint8_t> in)
{
    std::vector<uint8_t> out(16);
    uint8_t state[4][4];
    for (int r = 0; r < 4; r++) {
        for (int c = 0; c < 4; c++) {
            state[r][c] = in[r + 4 * c];
        }
    }
    addRoundKey(state, 0, expandedKey);
    for (int round = 1; round < Nr; round++) {
        subBytes(state);
        shiftRows(state);
        mixColumns(state);
        addRoundKey(state, round, expandedKey);
    }
    subBytes(state);
    shiftRows(state);
    addRoundKey(state, Nr, expandedKey);

    for (int r = 0; r < 4; r++) {
        for (int c = 0; c < 4; c++) {
```

```
            out[r + 4 * c] = state[r][c];
        }
    }
    return out;
}
// Decryption
std::vector<uint8_t> AES::decrypt(std::vector<uint8_t> in)
{
    std::vector<uint8_t> out(16);
    uint8_t state[4][4];
    for (int r = 0; r < 4; r++) {
        for (int c = 0; c < 4; c++) {
            state[r][c] = in[r + 4 * c];
        }
    }

    addRoundKey(state, Nr, expandedKey);
    for (int round = (Nr - 1); round >= 1; round--) {
        invShiftRows(state);
        invSubBytes(state);
        addRoundKey(state, round, expandedKey);
        invMixColumns(state);
    }
    invShiftRows(state);
    invSubBytes(state);
    addRoundKey(state, 0, expandedKey);

    for (int r = 0; r < 4; r++) {
        for (int c = 0; c < 4; c++) {
            out[r + 4 * c] = state[r][c];
        }
    }
    return out;
}

AES::~AES()
{
}

/*
 *  CTR52-CPP.
 *      Guarantee that library functions do not overflow.
 */
// Function for key expansion
std::vector<std::vector<uint8_t>> AES::keyExpansion(std::vector<uint8_t> key)
{
    std::vector<uint8_t> temp(4);
    std::vector<std::vector<uint8_t>> w = create2Darray(Nb * (Nr + 1), 4);
```

```
    for (int i = 0; i < Nk; i++) {
        w[i] = createWord(key[4 * i + 0], key[4 * i + 1], key[4 * i + 2], key[4 * i +
3]);
    }
    for (int i = Nk; i < Nb*(Nr + 1); i++) {
        temp = w[i - 1];
        if (i % Nk == 0) {
            temp = wordXOR(subWord(rotWord(temp)), rcon[i / Nk]);
        }
        else if ((Nk > 6) && (i % Nk == 4)) {
            temp = subWord(temp);
        }
        w[i] = wordXOR(w[i - Nk], temp);
    }

    return w;
}
// Rotation function
std::vector<uint8_t> AES::rotWord(std::vector<uint8_t> a)
{
    std::vector<uint8_t> word(4);
    for (int i = 0; i < 4; i++) {
        word[i] = a[(i + 1) % 4];
    }
    return word;
}
// AddRoundKey function
void AES::addRoundKey(uint8_t state[][4], int round, std::vector<std::vector<uint8_t>>
key)
{
    int l = round * Nb;
    for (int c = 0; c < 4; c++) {
        std::vector<uint8_t> word = key[l + c];
        for (int r = 0; r < 4; r++) {
            state[r][c] = state[r][c] ^ word[r];
        }
    }
}
// Substitution function
void AES::subBytes(uint8_t state[][4])
{
    for (int r = 0; r < 4; r++) {
        for (int c = 0; c < 4; c++) {
            int col = static_cast<int>((state[r][c] & 0x0f));
            int row = static_cast<int>((state[r][c] & 0xf0) >> 4);
            state[r][c] = S_BOX[row][col];
        }
    }
```

```
}
// Shift function
void AES::shiftRows(uint8_t state[][4])
{
    for (int r = 1; r < 4; r++) {
        uint8_t shift[4];
        for (int c = 0; c < 4; c++) {
            shift[c] = state[r][(c + r) % Nb];
        }
        for (int c = 0; c < 4; c++) {
            state[r][c] = shift[c];
        }
    }
}
// Mix Columns function
void AES::mixColumns(uint8_t state[][4])
{
    for (int c = 0; c < Nb; c++) {
        uint8_t s[4];
        s[0] = (multiply(state[0][c], 0x02)) ^ (multiply(state[1][c], 0x03)) ^
(state[2][c]) ^ (state[3][c]);
        s[1] = (state[0][c]) ^ (multiply(state[1][c], 0x02)) ^ (multiply(state[2][c],
0x03)) ^ (state[3][c]);
        s[2] = (state[0][c]) ^ (state[1][c]) ^ (multiply(state[2][c], 0x02)) ^
(multiply(state[3][c], 0x03));
        s[3] = (multiply(state[0][c], 0x03)) ^ (state[1][c]) ^ (state[2][c]) ^
(multiply(state[3][c], 0x02));
        for (int r = 0; r < 4; r++) {
            state[r][c] = s[r];
        }
    }
}
// Inverse Substitution function
void AES::invSubBytes(uint8_t state[][4])
{
    for (int r = 0; r < 4; r++) {
        for (int c = 0; c < 4; c++) {
            int col = static_cast<int>((state[r][c] & 0x0f));
            int row = static_cast<int>((state[r][c] & 0xf0) >> 4);
            state[r][c] = INV_S_BOX[row][col];
        }
    }
}
// Inverse Shift function
void AES::invShiftRows(uint8_t state[][4])
{
    for (int r = 1; r < 4; r++) {
        uint8_t shift[4];
```

```
        for (int c = 0; c < 4; c++) {
            shift[c] = state[r][(c + (Nb-r)) % Nb];
        }
        for (int c = 0; c < 4; c++) {
            state[r][c] = shift[c];
        }
    }
}
// Inverse Mix Columns function
void AES::invMixColumns(uint8_t state[][4])
{
    for (int c = 0; c < Nb; c++) {
        uint8_t s[4];
        s[0] = (multiply(state[0][c], 0x0e)) ^ (multiply(state[1][c], 0x0b)) ^
(multiply(state[2][c], 0x0d)) ^ (multiply(state[3][c], 0x09));
        s[1] = (multiply(state[0][c], 0x09)) ^ (multiply(state[1][c], 0x0e)) ^
(multiply(state[2][c], 0x0b)) ^ (multiply(state[3][c], 0x0d));
        s[2] = (multiply(state[0][c], 0x0d)) ^ (multiply(state[1][c], 0x09)) ^
(multiply(state[2][c], 0x0e)) ^ (multiply(state[3][c], 0x0b));
        s[3] = (multiply(state[0][c], 0x0b)) ^ (multiply(state[1][c], 0x0d)) ^
(multiply(state[2][c], 0x09)) ^ (multiply(state[3][c], 0x0e));
        for (int r = 0; r < 4; r++) {
            state[r][c] = s[r];
        }
    }
}
// GF Multiplication function
uint8_t AES::multiply(uint8_t a, uint8_t b)
{
    uint8_t ans = 0x00;
    uint8_t extra = 0x00;
    while (b) {
        if (b & 0x01)   ans = ans ^ a;
        else extra = extra ^ a;
        b = b >> 1;
        if (a & 0x80) {
            a = (a << 1) ^ 0x1b;
        }
        else {
            a = (a << 1) ^ 0x00;
        }
    }
    return ans;
}
// Helper function
std::vector<std::vector<uint8_t>> AES::create2Darray(int row, int col)
{
    std::vector<std::vector<uint8_t>> array(row);
```

```
    for (int r = 0; r < row; r++) {
        std::vector<uint8_t> a(col);
        for (int c = 0; c < col; c++) {
            a[c] = 0x00;
        }
        array[r] = a;
    }
    return array;
}
// Helper function
std::vector<uint8_t> AES::createWord(uint8_t a, uint8_t b, uint8_t c, uint8_t d)
{
    std::vector<uint8_t> word(4);
    word[0] = a;
    word[1] = b;
    word[2] = c;
    word[3] = d;
    return word;
}
// Helper function
std::vector<uint8_t> AES::wordXOR(std::vector<uint8_t> a, std::vector<uint8_t> b)
{
    std::vector<uint8_t> ans(4);
    for (int i = 0; i < 4; i++) {
        ans[i] = a[i] ^ b[i];
    }
    return ans;
}
// Helper function
std::vector<uint8_t> AES::subWord(std::vector<uint8_t> a)
{
    std::vector<uint8_t> word(4);
    for (int i = 0; i < 4; i++) {
        int col = static_cast<int>((a[i] & 0x0f));
        int row = static_cast<int>((a[i] & 0xf0) >> 4);
        word[i] = S_BOX[row][col];
    }
    return word;
}
```

APPENDIX B

List of coding practices that we have learned and used

- MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers
- MSC51-CPP. Ensure your random number generator is properly seeded
- MSC52-CPP. Value-returning functions must return a value from all exit paths
- Use unsigned bytes to represent binary data
- Avoid table look-ups indexed by secret data
- Avoid secret dependent loop bounds
- Avoid branching controlled by secret data
- Compare secret strings in constant time

APPENDIX C

List of coding practices that we have learned and used

- DCL51-CPP. Do not declare or define a reserved identifier
- DCL60-CPP. Obey the one-definition rule
- CTR50-CPP. Guarantee that container indices and iterators are within the valid range
- CTR52-CPP. Guarantee that library functions do not overflow
- CTR53-CPP. Use valid iterator ranges
- CTR55-CPP. Do not use an additive operator on an iterator if the result would overflow
- FIO51-CPP Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination.

List of coding practices that we learned but didn't use.

- EXP53-CPP. Do not read uninitialized memory
- EXP54-CPP. Do not access an object outside of its lifetime
- EXP57-CPP. Do not cast or delete pointers to incomplete classes
- DCL55-CPP. Avoid information leakage when passing a class object across a trust boundary
- INT50-CPP. Do not cast to an out-of-range enumeration value
- STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string
- MEM51-CPP. Properly deallocate dynamically allocated resources
- MEM52-CPP. Detect and handle memory allocation errors
- ERR57-CPP. Do not leak resources when handling exceptions