CSE539 Spring 2021 Project Report

Eric Bell, Jacob Rosengard, Raj Kane

Contents

1 Introduction

This report documents what we learned during our work for the CSE539S21 project. In this project, we implemented the AES block cipher with the goal of gaining practical knowledge of cryptography beyond the material taught in class. We do not claim to have implemented a fully cryptographically secure version of AES, as implementing secure cryptosystem implementations is a practice best left to experts. Instead, we learned the best practices for secure coding in general and secure coding for crypto in particular and made sure to apply these practices in the code when applicable. We document these best practices in several sections of this report as well as in the code itself.

The Advanced Encryption Standard (AES) is a symmetric block cipher that processes inputs of 16 bytes using keys of 16, 24, or 32 bytes [1]. Our implementation supports all three key lengths in addition to plaintexts of variable lengths which is accomplished by the use of modes of operation.

In Section 2 we describe the functional implementation. Rather than repeat facts about AES itself which are extensively documented elsewhere, we highlight important and interesting parts of the code as well as our learning process.

In Section 3, we describe what we learned about secure coding practices for crypto. We identify attacks on AES and how we addressed them in the implementation. We also identify attacks which we are aware of yet did not address and how we could have addressed them.

In Section 4, we describe what we learned about secure coding in general. We detail which SEI secure coding practices are followed in various portions of the code as well as those practices which we learned but did not use.

In Section 5, we provide a summary of the report.

Appendix A contains a printout of the documented code. Appendix B contains a list of the practices described in Section 3. Appendix C contains a list of the practices described in Section 4.

2 Implementation

2.1 Encryption

We implemented a function `encrypt` which closely matches the cipher specification provided in the AES standard [1] and calls the following functions: `keyExpansion`, `addRoundKey`, `subBytes`, `shiftRows`, and `mixColumns`. These functions adhere to the similarly named functions in the standard document by performing operations on the byte elements of the state array which is copied to from the input. The detailed descriptions of these functions can be read in the AES standard or can be gleaned by reading the code. Below, we note a few interesting portions of the cipher in our implementation.

In our implementation, the `subBytes` function, which replaces each element of the state array with its corresponding substitution box (s-box) value, dynamically calculates the s-box values by implementing the functions `galoisFieldInv` and `galoisFieldMult` to perform inversion and multiplication operations within a Galois field. We could have instead chosen to implement a lookup table. While it is both easier and more efficient to implement this function by hard-coding the s-box values into a lookup table, we chose to dynamically calculate the values because we prioritized cryptographically secure coding principles over efficiency. We revisit this choice in Section 3.

The `mixColumns` function implements multiplication in a Galois field by the polynomial $\{03\}, \{01\}, \{01\}, \{02\}$ modulo $x^4 + 1$. This function, too, takes advantage of the functions we implemented for performing mathematical operations within a Galois field. We describe the details of our implementations of Galois field operations in Section 2.3.

2.2 Decryption

We implemented a function `decrypt` which closely matches the inverse cipher specification provided in the AES standard [1] and calls the following functions: `keyExpansion`, `addRoundKey`, `invShiftRows`, `invSubBytes`, and `invMixColumns`. The `decrypt` function is the inverse of the `encrypt` function and the two functions largely follow the same form. A few differences between the two are that row shifting precedes byte substitution in the `decrypt` function and that the round function of `decrypt` loops backward over the number of rounds. Again, we omit a fully detailed description of our implementation as this can be learned from the AES standard or from the code. We note some interesting portions of our implementation of the inverse cipher.

`addRoundKey` is common to both the cipher and the inverse cipher, with the only difference in its implementations being that in the inverse cipher it is passed the last round key for the initial round and loops backwards over the number of rounds until it is passed the first round key.

The `invSubBytes` function replaces each element of the state array with its corresponding inverse s-box value. We call the s-box for the inverse cipher the inverse s-box, and it is similar to the s-box in that it is easier and more efficient to implement a hard-coded s-box that serves as a lookup table. However, our implementation dynamically calculates the inverse s-box values, a choice we discuss in Section 3.

`invMixColumns` implements multiplication in a Galois field by the polynomial $\{0b\}, \{0d\}, \{09\}, \{0e\}$ modulo $x^4 + 1$. The following section describes our implementation of Galois Field operations.

2.3 Galois fields

The AES algorithm makes extensive use of mathematical operations in the finite field $GF(2^8)$. AES specifically uses multiplication in the field for column mixing and for key expansion as well as the multiplicative inverse operation when computing the s-box. In our implementation, we defined functions for both multiplication and inversion: `galoisFieldMult` and `galoisFieldInv`, respectively.

Multiplication in the finite field with polynomials $a(x)$ and $b(x)$ is defined by first multiplying the two polynomials together followed by a modulo operation by the polynomial $x^8 + x^4 + x^3 + x + 1$. Because of the properties of the finite field, we can represent these polynomials in binary by setting a one in the bit position of each term in the polynomial. For example, the polynomial $x^8 + x^4 + x^3 + x + 1$ can be represented as $\{100011011\}$ in binary or as $\{11\text{b}\}$ in hexadecimal.

Our implementation utilizes the fact that the multiplication of two elements in the field can be represented as the sum of various powers of 2. For example, $\{01010111\} \bullet \{00010011\}$ can be calculated as

$$(\{01010111\} \bullet \{2^0\}) \oplus (\{01010111\} \bullet \{2^1\}) \oplus (\{01010111\} \bullet \{2^4\})$$

We programmed this by looping over the 8 bits in a byte and first checking if the least-significant bit in the $b(x)$ polynomial is one and if so xoring the $a(x)$ polynomial with the current product. Then, $a(x)$ is multiplied by two. This is accomplished through a left shift by one. Then, if the most-significant bit of $a(x)$ was one before the shift, $a(x)$ is xored with $\{1b\}$.

We implemented the inverse operation by taking advantage of the fact that for every element a in the field, $a^{255} = 1$. This shows that the inverse of an element a is equal to a^{254} since $a \cdot a^{254} = a^{255} = 1$. Our implementation then takes a value and uses the multiplication algorithm above to multiply the value by itself 254 times.

Overall, these operations are slow. Multiplication requires around 19 instructions per bit or about 152 instructions per byte. Multiplicative inverse uses 254 multiplications which sums up to at least 38,608 instructions. Each `subBytes` operation uses one inverse operation, and `subBytes` is called 16 times a round and with a minimum of 10 rounds this totals over 6 million instructions for our Galois field operations. This represents a major part of the running time of our AES implementation. We could have used lookup tables with the precomputed results for the s-box and other multiplication values but we discuss in Section 3 why sacrificing speed was important for security.

2.4 Modes of operation

From the Recommendation for Block Cipher Modes of Operations published by the NIST, we implemented all 5 modes specified in the document: ECB, CBC, CFB, OFB, and CTR [6]. Our implementation applies padding to the input regardless of mode, even if the specification does not require padding for modes such as OFB and CTR. This was mainly to give us the ability to determine the size of the plaintext after decrypting the ciphertext. We implemented the PKCS#7 padding scheme because it is easy to implement and because it is one of the most well-known and widely used padding schemes. For example, the documentation for OpenSSL refers to PKCS#5, the analogue of PKCS#7 for 8 byte blocks, as "standard block padding" [8]. We developed each mode in its own functions, with each mode having an encrypt function and corresponding decrypt function.

One detail the specification leaves to the implementer is how to initialize the CTR mode's counter. The specification only requires that the counter be unique for each message encrypted with the same key. The method we implemented was to have half of the counter, the upper 8 bytes, be a nonce and the lower 8 bytes be the counter which is incremented for every block.

The caller chooses the nonce unpredictably and uniquely across the same key.

A notable implementation detail with CFB mode is that we only implemented CFB with a 128 bit parameter. The specification allows for a variable number of bits of the plaintext to be xored in each block. For example, a parameter of 1 bit means that for each block, only one bit of the input is combined with the output of the cipher. We decided to only implement 128 bit mode, or use all 16 bytes of the cipher output, for the sake of simplicity.

2.5 Randomness

Our implementation provides an interface to get n bytes of random numbers from a generator. For the actual randomness, we decided to use the /dev/urandom device file present on most UNIX and UNIX-like operating systems. This was chosen over using the built-in `random_device` in the standard C++ library. The `random_device` is not required by the C++ standard to be implemented with a cryptographically secure generator or source of randomness [9]. The device /dev/urandom is listed in RFC 4086 as an example of a cryptographically secure random generator and as a "Complete Randomness Generator" [7]. While /dev/urandom and /dev/random do produce high quality randomness, this approach is not perfect. Currently, the code is platform dependent; the device file is only present on UNIX and UNIX-like operating systems. If the device file is not present, then there is no backup way to generate random numbers in our implementation.

2.6 Interface

All of the interaction a user has with our implementation is through the command line. To use our application, the user enters command-line arguments specifying whether they wish to perform encryption or decryption, the mode of operation, and the desired key length to use.

Upon the successful completion of encryption, all of the data that is necessary to retrieve the plaintext is outputted to the user. This includes the ciphertext, the randomly generated key, and the value for the initialization vector (IV) or initial counter block when required by the chosen mode of operation. Upon the successful completion of decryption, the recovered plaintext is displayed to the user.

2.7 Implementation validation

The NIST publishes AES validation requirements called the AESAVS [3]. They have test vectors for single block and multi-block messages for the ECB, CBC, CFB and OFB modes of operation. In order for us to test our implementation with those vectors they published, we created a Python script that loads the data from the response files and executes our implementation. The script then compares the output of the program to the expected output. Due to our implementation using padding on all messages, the script removes the last block of the ciphertext before comparison. This also means that we were are unable to test the decryption section of the vector files since they lack the PKCS#7 padding we use. To solve this, the ciphertext is sent back through the program in decryption mode. We then compare the result from the program with the starting plaintext.

Using this methodology, our implementation successfully encrypts and decrypts all of the test messages with the provided key and IV, if applicable, for the ECB, CBC, CFB in 128 bit mode, and OFB modes of operation. Due to the variability in CTR mode implementations, the AESAVS does not provide vectors for this mode. Instead, the testing laboratories are instructed to review the code for the CTR mode to ensure that it correctly implements the design of the mode [3]. We tested our implementation's CTR mode by encrypting a random message with a random key and random nonce, then sending the resulting ciphertext into our decyption function for CTR. This shows that our CTR mode is at least symmetrical in its encryption and decryption functions.

3 Crypto learning

3.1 Cache timing attack

A common pitfall for AES implementations is susceptibility to a cache timing attack, as described by Daniel Bernstein in 2005 [4]. A cache timing attack does not attack the algorithm itself but rather attacks faults in the implementation that allow data leakage to reveal patterns in looking up s-box values in memory. If the lookup tables are hard-coded, then looking up the s-box value for a given state array value caches those contents of the lookup table. It is faster to read a cached byte than a byte from the lookup table in memory, and these variable read times can leak information to an attacker who can reasonably expect the time for the algorithm to correlate with the time for the s-box lookups. In this way, an attacker can glean which bytes in the lookup table have been cached and thus which s-box values the algorithm has placed in the state array.

We handled this attack in our implementation by dynamically calculating s-box values and inverse s-box values rather than hard-coding the values into a lookup table, as noted in Section 2.1 and in Section 2.2. Tromer et al. have noted that an algorithm implemented with just logical and algebraic operations with no reliance on lookup tables is immune to cache timing attacks [11]. By following this approach, our implementation ensures that no s-box values are cached and thus leaked.

However, Tromer et al. have also noted that an implementation such as ours goes against the reasoning behind choosing Rijndael for the AES algorithm, since Rijndael's high performance was a highly desirable feature for the cipher. The low performance of our implementation may be undesirable in practical applications of AES, but our focus in this project was to prioritize cryptographic principles over efficiency. We describe the implementation of this dynamic calculation in Section 2.3, noting the obvious slowness of this approach.

3.2 Padding oracle attack

Another common pitfall for AES implementations that include modes of operation is susceptibility to a padding oracle attack, which was described by Serge Vaudenay in 2002 [12]. Vaudenay described an attack on the CBC mode of operation in particular, though the premise of the attack can be extended to all modes that use padding. In modes of operation that use padding, the plaintext is first padded to the appropriate length with the appropriate byte values based on the chosen padding scheme before being encrypted in blocks. Under such a scheme, the inverse cipher must strip decrypted blocks of padding. This process requires checking whether the padding format is valid, and if it isn't, an implementation can acknowledge this with an error message specifying a padding error. This leaking provides an attacker with a padding oracle which can be used to recover the plaintext.

Our implementation uses the PKCS#7 scheme to pad inputs. We used this padding scheme for all modes of operation, and explain our reasoning for doing so in Section 2.4. A naive implementation of this padding scheme would render all modes of operations susceptible to a padding oracle attack. However, we took care to not differentiate between a padding error and a generic decryption error. Our implementation checks the validity of the padding, and if it detects a padding error it acknowledges this with a ”Decryption Error” message that does not reveal that the error is due to the padding. Hence, an attacker on our implementation cannot exploit any information leaked about the padding of an input that would give them an advantage enabling a padding oracle attack.

3.3 Related key attack

The invertibility of key expansion in AES is desirable because it allows one to recover the full key given just the final round key, but this property can also be used to launch a related key attack [2]. In this attack, an attacker is given four keys k_1, k_2, k_3, k_4 related in a particular way with respect to the xor operation and also given plaintext/ciphertext pairs generated from each of the keys. In a 2009 paper, Biryukov and Khovratovich presented two ”boomerang attacks” in which attackers given this information can exploit the key expansion scheme in AES to recover the key in time $2^{99.5}$, which is much faster than an exhaustive search [5].

In our implementation, keys are always generated independently from /dev/urandom which is a cryptographically secure generator. It is thus highly improbable for a set of keys to be related in such a way that would make the related key attack possible. We discuss our implementation of /dev/urandom in Section 2.5.

3.4 Other attacks

In addition to cache timing attacks, padding oracle attacks, and related key attacks there exist a number of other side channel attacks which do not target the AES cipher itself but rather information leaked by its implementation. These additional attacks include power analysis attacks which target the amount of power it takes a piece of hardware to perform encryption and electromagnetic attacks which target the amount of electromagnetic radiation a device emits during encryption [2]. These attacks ought to be considerations for any practical implementations of AES, and such programmers should look into preventing conditional branching, reducing power consumption variability, and specific hardware concerns in order to address them. We did not address these attacks in our implementation because of scope, but we found it interesting to note the level to which interactions with the physical environment can be exploited to reveal cryptographic secrets.

4 Secure coding

This section elaborates on the specific coding practices we implemented to assure the reliability and security of the implementation. The two main areas we implemented were in dealing with errors and exceptions, and in using the container classes in the C++ standard library.

4.1 Exceptions

In our implementation, we implemented most of the rules specified in the SEI CERT secure coding practices for errors and exceptions. In all of the functions implementing the different modes of operation, which we expect to be the only functions a user would interface with, we ensured that those functions do not throw exceptions. We achieved this by handling all exceptions that could occur through a try-catch block that encapsulates all of the logic in the function. While this would be bad practice in most applications, in ours, we believe that a general catch block to catch all exceptions will leak the least amount of information to an adversary. Since we ensured that the function does not throw any exceptions, we then added in the function header `noexcept(true)` to indicate to the user and compiler that the function does not throw an exception. Since exceptions can still occur in the function, we also needed to guarantee exception safety, meaning that if an error does occur, then the program state is not modified. We achieved this by first making the input parameters such as the key, IV, and the plaintext constant in the function definition. Then we also reset the output vector in the event an exception occurs and is caught. This gives us strong exception safety defined by the SEI CERT secure coding practices [10].

4.2 Containers

Our implementation extensively uses the containers included in the C++ standard library, such as the `vector<>` class and `array<>` class. Because of this, we needed to ensure that any indices used to access the container are within the range $0 \leq i < length$. We implemented three different techniques to ensure that all accesses are valid. First, any variable or value used to access an index is of type `std::size_t`. This is to ensure that the index is both non-negative and not larger than the maximum size accessible. Next, we used if statements to check if the index is within the range. Finally, we used the `.at(i)` method of the containers instead of the array access operator `[]`. The `.at(i)` method has built-in range checking and will throw an exception if the index is out-of-bounds. Because the method throws an exception, we only used it inside of the functions implementing the modes of operations since they are exception safe.

5 Summary

Throughout this semester-long project, we have gained a deep understanding of how AES works in addition to an insight into the best practices that exist for secure coding and cryptographic applications. The time spent researching for this project was quite insightful, but being able to take what we learned and apply it to our very own implementation allowed us an even richer level of understanding. This experience has given us an appreciation for the various crypto schemes that exist today and the hard work that has gone into conceiving of and implementing them.

6 References

[1] *Advanced Encryption Standard (AES)*, Federal Information Processing Standard (FIPS) 197, National Institute of Standards and Technology (NIST), November 2001.

[2] Dan Boneh and Victor Shoup, "Block ciphers," in *A Graduate Course in Applied Cryptography*, Version 0.5, 2020.

[3] Lawrence E. Bassham III, "The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)," National Institute of Standards and Technology (NIST), November 2002.

[4] Daniel J. Bernstein, "Cache timing attacks on AES," The University of Illinois at Chicago, April 2005.

[5] Alex Biryukov and Dmitry Khovratovich, ”Related-key Cryptanalysis of the full AES-192 and AES-256,” in *Advances in Cryptology-ASIACRYPT 2009*, 2009.

[6] Morris Dworkin, ”Recommendation for Block Cipher Modes of Operation Methods and Techniques,” National Institute of Standards and Technology (NIST), Gaithersburg, MD, NIST Special Publication 800-38A, 2001.

[7] D. Eastlake 3rd, J. Schiller, and S. Crocker, ”Randomness Requirements for Security,” RFC 4086, ITEF, June 2005

[8] ”enc.” OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/docs/man1.1.1/man1/enc.html (accessed April 12, 2021).

[9] Programming languages — C++, ISO/IEC Standard 14882-2020

[10] *SEI CERT C++ Coding Standards*, May 2020. [Online]. Available: https://www.securecoding.cert.org.

[11] Eran Tromer, Dag Arne Osvik, and Adi Shamir, ”Efficient Cache Attacks on AES, and Countermeasures,” Massachussetts Institute of Technology, November 2005.

[12] Serge Vaudenay, ”Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS...,” in *EUROCRYPT 2002*, 2002.

7 Appendix A

Below is a printout of the code from the src directory of our code, excluding the makefile.

```
/**
  @file interface.hpp: Methods for interacting with the user
*/
#ifndef SRC_INTERFACE_HPP
#define SRC_INTERFACE_HPP

#include <iostream>
```

```
#include <string>
#include <vector>
#include <algorithm>
#include <cstring>
#include "AESmath.hpp"

void printVector(std::vector<unsigned char>& vec);
void printEncryptionResults(std::vector<unsigned char>& output,
    std::vector<unsigned char>& key);
void printEncryptionResults(std::vector<unsigned char>& output,
    std::vector<unsigned char>& key, std::vector<unsigned char
   >& iv);
void printEncryptionResults(std::vector<unsigned char>& output,
    std::vector<unsigned char>& key, std::array<unsigned char,
    NUM_BYTES / 2>& nonce);
void printDecrpytionResults(std::vector<unsigned char>& output)
   ;
int getKeySizeInBytes(char* keySize);
void inputToVector(std::vector<unsigned char>& vec);

#endif
```

```
/**
  @file interface.cpp: Methods for interacting with the user
*/

#include "interface.hpp"


/**
   Print the contents of a vector as bytes
    @param vec: vector of unsigned char values to be printed in
        hex format
    @return none
 */
void printVector(std::vector<unsigned char>& vec) {
```

```
    for(std::size_t i = 0; i < vec.size(); i++) { // Secure
        coding: CTR50-CPP. Guarantee that container indices and
        iterators are within the valid range
        if((int) vec[i] < 16) {
            std::cout << '0';
        }
        std::cout << std::hex << (int) vec[i] << " ";
    }
    std::cout << std::endl;
}

/**
    Print the ciphertext, and key after an encryption
    Used for modes that do not require an IV (ECB)
    @param ouput: ciphertext received as output from encryption
    @param key: key used for encryption
    @return none
 */
void printEncryptionResults(std::vector<unsigned char>& output,
     std::vector<unsigned char>& key) {
    std::cout << "\nCIPHERTEXT: ";
    printVector(output);
    std::cout << "KEY: ";
    printVector(key);
}

/**
    Print the ciphertext, key, and IV after an encryption
    Used for modes that require an IV (CBC, CFB, OFB)
    @param ouput: ciphertext received as output from encryption
    @param key: key used for encryption
    @param iv: IV used for encryption in the chosen mode
    @return none
 */
void printEncryptionResults(std::vector<unsigned char>& output,
     std::vector<unsigned char>& key, std::vector<unsigned char
    >& iv) {
    std::cout << "\nCIPHERTEXT: ";
    printVector(output);
```

```
    std::cout << "KEY: ";
    printVector(key);
    std::cout << "IV: ";
    printVector(iv);
}


/**
    Print the ciphertext, key, and nonce for initial counter
       after an encryption
    Used for CTR
    @param ouput: ciphertext received as output from encryption
    @param key: key used for encryption
    @param nonce: IV used for encryption in the chosen mode
    @return none
 */
void printEncryptionResults(std::vector<unsigned char>& output,
     std::vector<unsigned char>& key, std::array<unsigned char,
     NUM_BYTES / 2>& nonce) {
    std::cout << "\nCIPHERTEXT: ";
    printVector(output);
    std::cout << "KEY: ";
    printVector(key);

    std::vector<unsigned char> vectorNonce;

    // Copy elements of array into vector for printing
    for(std::size_t i = 0; i < nonce.size(); i++) { // Secure
       coding: CTR50-CPP. Guarantee that container indices and
       iterators are within the valid range
        vectorNonce.push_back(nonce[i]);
    }

    std::cout << "NONCE: ";
    printVector(vectorNonce);
}
/**
    Print the recovered plaintext after a decrpytion
    @param ouput: plaintext recovered as output from encryption
```

```
    @return none
 */
void printDecrpytionResults(std::vector<unsigned char>&output)
    {
    std::cout << "\nDECRPYTED PLAINTEXT: ";
    printVector(output);
}

/**

    @param vec: vector of hex values to be printed
    @return none
 */
int getKeySizeInBytes(char* keySize) {
    int returnSize;

    if(std::strcmp(keySize, "128") == 0)
        returnSize = 16;
    else if(std::strcmp(keySize, "192") == 0)
        returnSize = 24;
    else if(std::strcmp(keySize, "256") == 0)
        returnSize = 32;
    else {
        returnSize = -1;
    }

    return returnSize;
}

/**
    Method for testing. Print the contents of vector as bytes
    @param vec: vector of hex values to be printed
    @return none
 */
void inputToVector(std::vector<unsigned char>& vec) {
    std::string line;
    unsigned char char_iterator;;

    std::getline(std::cin, line);
```

```
    // Remove spaces from input
    std::string::iterator end_pos = std::remove(line.begin(),
        line.end(), ' ');
    line.erase(end_pos, line.end());


    // Convert each byte to integer, then store as unsigned
        char in input vector
    for (std::size_t i = 0; i < line.size(); i += 2) { //
        Secure coding: CTR50-CPP. Guarantee that container
        indices and iterators are within the valid range
        unsigned char byteValue = (unsigned char) std::stoi(
            line.substr(i, 2), nullptr, 16);
        vec.push_back(byteValue); // Secure coding: OOP57-CPP.
            Prefer special member functions and overloaded
            operators to C Standard Library functions
    }
}



/**
  @file main.cpp
  Implementation of user interface with the program
*/

#include <iostream>
#include <string>
#include <algorithm>
#include <cstring>
#include "AESRand.hpp"
#include "AESmodes.hpp"
#include "encrypt.hpp"
#include "decrypt.hpp"
#include "interface.hpp"
```

```
// USAGE: ./main [enc, encrypt/ dec, decrypt] [ecb/cbc/cfb/ofb/
   ctr] [-r/-k] [128, 192, 256] (-iv/-nonce)

// [] - required parameters*
// () - optional parameters
// *[-r/-k] omitted for decryption

int main(int argc, char** argv) {
    AESRand rand;
    std::vector<unsigned char> input;
    std::vector<unsigned char> output;
    std::vector<unsigned char> key;
    std::vector<unsigned char> iv;

    const int IV_SIZE = 16;
    std::string line;
    unsigned char char_iterator;

    bool algorithmSuccess;

    if(argc >= 4) {
        char* aes_function = argv[1];
        char* mode = argv[2];
        char* keyType = argv[3];


        // Encryption
        if (std::strcmp(aes_function, "encrypt") == 0 || std::
           strcmp(aes_function, "enc") == 0) {
            char* keySize;
            int keyByteSize;
            // Extract key size from command line arguments if
               proper number of arguments provided
            if (argc > 4) {
                keySize = argv[4];
                // Convert key size to integer value
                keyByteSize = getKeySizeInBytes(keySize);
            }
            else {
```

```
    // Invalid number of arguments to extract key
       size
    keyByteSize = -1;
}

// If invalid key size is entered, output error
   message and terminate program
if (keyByteSize == -1) {
    std::cout << "Invalid parameter for key size.\n";

    return 2;
}

// Receive plaintext to encrypt
std::cout << "Enter plaintext: ";

inputToVector(input);

// Create random key if -r command line argument is
   provided
if (std::strcmp(keyType, "-r") == 0) {
    key = rand.generateBytes(keyByteSize);
}
// Receive key from user input if -k command line
   argument is provided
else if (std::strcmp(keyType, "-k") == 0) {
    std::cout << "Enter key: ";
    inputToVector(key);
    // Check that user entered correct number of
       bytes for designated key size
    if (key.size() != keyByteSize) {
        std::cout << "Invalid number of bytes entered
             for key.\n";
        return 2;
    }

}
// If no valid flag is provided, alert user and
   terminate execution
```

```
else {
    std::cout << "Invalid flag entered for key\n";
    return 2;
}

// Encrypt with mode entered by user

// Encryption with ECB
if (std::strcmp(mode, "ecb") == 0 || std::strcmp(
   mode, "ECB") == 0) {
    algorithmSuccess = encrypt_ecb(input, output,
       key);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;

    printEncryptionResults(output, key);
}
// Encryption with CBC
else if (std::strcmp(mode, "cbc") == 0 || std::
   strcmp(mode, "CBC") == 0) {
    // If -iv command line argument is provided,
       receive value of IV from user
    if (argc == 6 && std::strcmp(argv[5], "-iv") ==
       0) {
        std::cout << "Enter IV: ";
        inputToVector(iv);

        // Ensure IV size is correct
        if (iv.size() != NUM_BYTES) {
            std::cout << "Invalid number of bytes
               entered for IV\n";
            return 2;
        }

    }
    else {
        iv = rand.generateBytes(IV_SIZE);
```

```
    }
    algorithmSuccess = encrypt_cbc(input, output,
        key, iv);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;

    printEncryptionResults(output, key, iv);
}
// Encryption with CFB
else if (std::strcmp(mode, "cfb") == 0 || std::
    strcmp(mode, "CFB") == 0) {
    // If -iv command line argument is provided,
        receive value of IV from user
    if (argc == 6 && std::strcmp(argv[5], "-iv") ==
        0) {
        std::cout << "Enter IV: ";
        inputToVector(iv);

        // Ensure IV size is correct
        if (iv.size() != NUM_BYTES) {
            std::cout << "Invalid number of bytes
                entered for IV\n";
            return 2;
        }

    }
    else {
        iv = rand.generateBytes(IV_SIZE);
    }

    algorithmSuccess = encrypt_cfb(input, output,
        key, iv);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;
```

```
    printEncryptionResults(output, key, iv);
}
// Encryption with OFB
else if (std::strcmp(mode, "ofb") == 0 || std::
    strcmp(mode, "OFB") == 0) {
    // If -iv command line argument is provided,
        receive value of IV from user
    if (argc == 6 && std::strcmp(argv[5], "-iv") ==
        0) {
        std::cout << "Enter IV: ";
        inputToVector(iv);

        // Ensure IV size is correct
        if (iv.size() != NUM_BYTES) {
            std::cout << "Invalid number of bytes
                entered for IV\n";
            return 2;
        }

    }
    else {
        iv = rand.generateBytes(IV_SIZE);
    }

    algorithmSuccess = encrypt_ofb(input, output,
        key, iv);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;

    printEncryptionResults(output, key, iv);
}
// Encryption with CTR
else if (std::strcmp(mode, "ctr") == 0 || std::
    strcmp(mode, "CTR") == 0) {
    std::array<unsigned char, NUM_BYTES / 2> nonce;
    std::vector<unsigned char> vectorNonce;
```

```
        // Set value of nonce
        // If -nonce command line argument is provided,
           receive value of nonce from user
        if (argc == 6 && std::strcmp(argv[5], "-nonce")
           == 0) {
            std::cout << "Enter nonce: ";
            inputToVector(vectorNonce);

            if (vectorNonce.size() != NUM_BYTES / 2) {
                std::cout << "Invalid number of bytes
                   entered for nonce.\n";
                return 4;
            }

            std::copy(vectorNonce.begin(), vectorNonce.
               end(), nonce.begin());
        }
        // If not for debugging, generate nonce value
           for encryption with CTR
        else {
            vectorNonce = rand.generateBytes(NUM_BYTES /
               2);
            std::copy(vectorNonce.begin(), vectorNonce.
               end(), nonce.begin());
        }

        algorithmSuccess = encrypt_ctr(input, output,
           key, nonce);

        // Stop execution if encryption is unsuccessful
        if (!algorithmSuccess)
            return 3;

        printEncryptionResults(output, key, nonce);
    }

}

// Decrypt functionality
```

```
if (std::strcmp(argv[1], "decrypt") == 0 || std::strcmp
   (argv[1], "dec") == 0) {
    char* keySize = argv[3];

    int keyByteSize = getKeySizeInBytes(keySize);

    // If invalid key size is entered, output error
       message and terminate program
    if (keyByteSize == -1) {
        std::cout << "Invalid parameter for key size.\n";

        return 2;
    }


    // Receive ciphertext to decrypt
    std::cout << "Enter ciphertext: ";
    inputToVector(input);

    // Receive key
    std::cout << "Enter key: ";
    inputToVector(key);

    // Ensure key is right size
    if (key.size() != 16 && key.size() != 24 && key.size
        () != 32) {

        std::cout << "Invalid key size!!\n Please enter
            a valid key\n";
        return 2;
    }

    // Decrypt with mode entered by user

    // Decryption with ECB
    if (std::strcmp(mode, "ecb") == 0 || std::strcmp(
       mode, "ECB") == 0) {
        decrypt_ecb(input, output, key);
```

```
}
// Decryption with CBC
else if (std::strcmp(mode, "cbc") == 0 || std::
    strcmp(mode, "CBC") == 0) {
    // Receive IV
    std::cout << "Enter IV: ";

    inputToVector(iv);

    // Ensure IV size is correct
    if (iv.size() != NUM_BYTES) {
        std::cout << "Invalid number of bytes entered
             for IV\n";
        return 2;
    }

     algorithmSuccess = decrypt_cbc(input, output,
         key, iv);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;

}
// Decryption with CFB
else if (std::strcmp(mode, "cfb") == 0|| std::strcmp
    (mode, "CFB") == 0) {
    // Receive IV
    std::cout << "Enter IV: ";
    inputToVector(iv);


    // Ensure IV size is correct
    if (iv.size() != NUM_BYTES) {
        std::cout << "Invalid number of bytes entered
             for IV\n";
        return 2;
    }
```

```
    algorithmSuccess = decrypt_cfb(input, output,
       key, iv);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;
}
// Decryption with OFB
else if (std::strcmp(mode, "ofb") == 0 || std::
   strcmp(mode, "OFB") == 0) {
    // Receive IV
    std::cout << "Enter IV: ";
    inputToVector(iv);


    // Ensure IV size is correct
    if (iv.size() != NUM_BYTES) {
        std::cout << "Invalid number of bytes entered
             for IV\n";
        return 2;
    }

    algorithmSuccess = decrypt_ofb(input, output,
       key, iv);

    // Stop execution if encryption is unsuccessful
    if (!algorithmSuccess)
        return 3;
}
// Decryption with CTR
else if (std::strcmp(mode, "ctr") == 0 || std::
   strcmp(mode, "CTR") == 0) {
    std::array<unsigned char, NUM_BYTES / 2> nonce;
    std::vector<unsigned char> vectorNonce;

    // Receive nonce of initial counter
    std::cout << "Enter nonce: ";
    inputToVector(vectorNonce);
```

```
                // Ensure proper number of bytes entered for
                   upper half of initial counter
                if (vectorNonce.size() != NUM_BYTES / 2) {
                    std::cout << "Invalid number of bytes entered
                        for nonce.\n";
                    return 2;
                }


                std::copy(vectorNonce.begin(), vectorNonce.end(),
                     nonce.begin());

                algorithmSuccess = decrypt_ctr(input, output,
                   key, nonce);

                // Stop execution if encryption is unsuccessful
                if (!algorithmSuccess)
                    return 3;
            }

            printDecrpytionResults(output);
        }
    }

    return 0;
}
```

```
/**
  @file AESrand.hpp: randomness class
*/
#ifndef AES_AESRAND_HPP
#define AES_AESRAND_HPP

#include <vector>
#include <array>
#include <random>
#include <iostream>
```

```
#include <fstream>
#include "AESmath.hpp"



//AESRand class
class AESRand {
public:
    AESRand();

    ~AESRand();

    std::vector<unsigned char> generateBytes(unsigned int
        numBytes);

private:
    std::ifstream urandom;
};


#endif //AES_AESRAND_HPP



/**
  @file AESrand.cpp: randomness class
*/
#include <random>
#include <algorithm>
#include "AESRand.hpp"
#include "encrypt.hpp"

// Secure coding: MSC50-CPP. Do not use std::rand() for
   generating pseudorandom numbers

/**
  AESRand constructor
  @return none
*/
```

```
AESRand::AESRand() {
    //Open /dev/urandom with the input and binary parameter
    this->urandom = std::ifstream("/dev/urandom", std::ios_base
        ::in | std::ios_base::binary);
}

/**
  AESRand deconstructor
  @return none
*/
AESRand::~AESRand() {
    //Close the file
    this->urandom.close();
}

/**
  AESRand::generateBytes
  Generates some number of bytes using Unix/Linux /dev/urandom
  @param numBytes: The number of random bytes needed
  @return A vector with the random bytes
*/
std::vector<unsigned char> AESRand::generateBytes(unsigned int
   numBytes) {
    std::vector<unsigned char> ret(numBytes, 0);

    //Cast the unsigned char array to a char array since
        ifstream returns char
    this->urandom.read((char*)ret.data(), numBytes);

    return ret;
}
```

```
/**
  @file AESmath.hpp: Prototypes for math and functions common
     to encryption and decryption
*/
#ifndef AES_MATH_HPP
```

```
#define AES_MATH_HPP

#include <array>
#include <vector>

// State size
#define NUM_BYTES 16

unsigned char galoisFieldMult(unsigned char a, unsigned char b)
    ;
unsigned char galoisFieldInv(unsigned char a);
unsigned char getSboxValue(unsigned char index);
unsigned char invGetSboxValue(unsigned char index);
void keyExpansion(const std::vector<unsigned char>& key, std::
    vector<unsigned char>& expansion, unsigned char keysize); //
     Secure coding: DCL52-CPP. Never qualify a reference type
    with const or volatile
void addRoundKey(std::array<unsigned char, 16>& state, unsigned
     char* key);

#endif



/**
  @file AESmath.cpp: Math and common functions to encryption
     and decryption
*/
#include "AESmath.hpp"

// From Appendix A of the AES spec
// This is the first byte of the rcon word array which is x^(i
    -1) in GF(2^8)
std::array<unsigned char, 11> rcon1_i_bytes = { 0x01, 0x02, 0
    x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36 };


/**
  Multiplies a by b in GF(2^8)
```

```
  @param a: the first polynomial
  @param b: the second polynomial
  @return a * b in GF(2^8)
*/
unsigned char galoisFieldMult(unsigned char a, unsigned char b)
    {
        unsigned char product = 0;
        for (unsigned char i = 0; i < 8; i++) {
                if ((b & 1) == 1) {
                        product ^= a;
                }

                unsigned char aHighBit = a & 0x80;
                a = a << 1;
                if (aHighBit) {
                        a ^= 0x1b;
                }

                b = b >> 1;
        }

        return product;
}


/**
  Computes the mutiplicative inverse of the polynomial a in GF
     (2^8)
  @param a: the polynomial to find the inverse of
  @return the inverse of a
*/
unsigned char galoisFieldInv(unsigned char a) {
        unsigned char product = a;

        // The inverse in GF(2^8) is really x^(255-1)
        // This is 253 iterations because the product is already
            a or a^1

        for (int i = 0; i < 253; i++) {
```

```
            product = galoisFieldMult(product, a);
        }

        return product;
}


/**
  Computes the AES key expansion
  @param key: the input key array
  @param expansion: the array to put the key expansion into
                  The array needs 16 * (Nr + 1) bytes or 16 * (
                     keysize/4 + 7) bytes allocated
  @param keysize: the size of the input key in bytes
                              Note: the key should be 16, 24,
                                 or 32 bytes large
  @return none
*/
void keyExpansion(const std::vector<unsigned char>& key, std::
   vector<unsigned char>& expansion, unsigned char keysize) {
        int Nk = keysize / 4;
        int Nr = (Nk + 6);

        for (std::size_t i = 0 ; i < keysize; i++) { // Secure
           coding: CTR50-CPP. Guarantee that container indices
           and iterators are within the valid range
            expansion[i] = key[i];
        }

        unsigned char temp[4];

        // i < Nb * (Nr + 1)
        //The number of bytes in a word is 4
        for(std::size_t i = Nk; i < (4 * (Nr + 1)); i++) { //
           Secure coding: CTR50-CPP. Guarantee that container
           indices and iterators are within the valid range
            temp[0] = expansion[(4 * (i - 1))];
            temp[1] = expansion[(4 * (i - 1)) + 1];
            temp[2] = expansion[(4 * (i - 1)) + 2];
```

```
temp[3] = expansion[(4 * (i - 1)) + 3];

if (i % Nk == 0) {
        //ROTWORD on temp
        unsigned char rotTemp = temp[0];
        temp[0] = temp[1];
        temp[1] = temp[2];
        temp[2] = temp[3];
        temp[3] = rotTemp;

        //SUBWORD
        temp[0] = getSboxValue(temp[0]);
        temp[1] = getSboxValue(temp[1]);
        temp[2] = getSboxValue(temp[2]);
        temp[3] = getSboxValue(temp[3]);

        //Xor with Rcon[i/Nk]
        //Since the last 3 bytes of Rcon are
           always zero, then temp[0] is the only
           byte changing
        temp[0] ^= rcon1_i_bytes[(i / Nk) - 1];
}
else if (Nk > 6 && i % Nk == 4) {
        //SUBWORD
        temp[0] = getSboxValue(temp[0]);
        temp[1] = getSboxValue(temp[1]);
        temp[2] = getSboxValue(temp[2]);
        temp[3] = getSboxValue(temp[3]);
}

//w[i] = w[i-Nk] xor temp
expansion[4 * i] = expansion[4 * (i - Nk)] ^
   temp[0];
expansion[4 * i + 1] = expansion[4 * (i - Nk) +
   1] ^ temp[1];
expansion[4 * i + 2] = expansion[4 * (i - Nk) +
   2] ^ temp[2];
expansion[4 * i + 3] = expansion[4 * (i - Nk) +
   3] ^ temp[3];
```

```
	}
}


/**
  XORs each byte of the state array with the key.
  @param state: the state array to modify
  @param key: vector of hex values representing the key
  @return none
*/
void addRoundKey(std::array<unsigned char, 16>& state, unsigned
    char* key) {
	for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
	    coding: CTR50-CPP. Guarantee that container indices
	    and iterators are within the valid range
		state[i] = state[i] ^ key[i];
	}
}


/**
  Computes inverse sbox value.
  @param index: byte of state array whose value to compute
  @return inverse sbox value of index
*/
unsigned char getSboxValue(unsigned char index) {
	unsigned char inv = galoisFieldInv(index);
	unsigned char matRow = 0xF1; // 11110001
	unsigned char out = 0;

	//Per bit
	for (int i = 0; i < 8; i++) {
		//Find the bits that, when 'multiplied' by the
		    matrix row, are one
		unsigned char app = (unsigned char) ((int)inv &
		    (int)matRow);

		//Every bit of the application
		for (int j = 0; j < 8; j++) {
```

```
                        //Add the bits together (j) and put it
                           into the corresponding output bit (i)
                        out ^= (((app >> j) & 1) << i);
                }

                //Left rotate the matrix row
                matRow = (matRow << 1) | (matRow >> 7);
        }

        return out ^ 0x63;
}


/**
  Computes inverse sbox value.
  @param index: byte of state array whose value to compute
  @return inverse sbox value of index
*/
unsigned char invGetSboxValue(unsigned char index) {
  unsigned char matRow = 0xA4; // 10100100
  unsigned char out = 0;

  // Per bit
  for (int i = 0; i < 8; i++) {
    // Find the bits that, when 'multiplied' by the matrix row,
         are one
    unsigned char app = (unsigned char) ((int)index & (int)
       matRow);

    // Every bit of the application
    for (int j = 0; j < 8; j++) {
      // Set output bit to sum of bits
      out ^= (((app >> j) & 1) << i);
    }

    // Left rotate the matrix row
    matRow = (matRow << 1) | (matRow >> 7);
  }
```

```
  out ^= 0x5;

  return galoisFieldInv(out);
}
```

```
/**
  @file AESmodes.hpp: prototypes for modes of operation
     functions
*/
#ifndef AES_MODES_HPP
#define AES_MODES_HPP

#include "AESmath.hpp"
#include "encrypt.hpp"
#include "decrypt.hpp"
#include <vector>

bool remove_padding(std::vector<unsigned char> &input) noexcept
   (false);

bool encrypt_ecb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
                const std::vector<unsigned char> &key) noexcept
                   (true);

bool decrypt_ecb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
                const std::vector<unsigned char> &key) noexcept
                   (true);

bool encrypt_cbc(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
                const std::vector<unsigned char> &key, const
                   std::vector<unsigned char> &IV) noexcept(
                   true);
```

```
bool decrypt_cbc(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
              const std::vector<unsigned char> &key, const
                 std::vector<unsigned char> &IV) noexcept(
                 true);

bool encrypt_ctr(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
              const std::vector<unsigned char> &key,
              const std::array<unsigned char, NUM_BYTES / 2>
                 &nonce) noexcept(true);

bool decrypt_ctr(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
              const std::vector<unsigned char> &key,
              const std::array<unsigned char, NUM_BYTES / 2>
                 &nonce) noexcept(true);

bool encrypt_cfb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
              const std::vector<unsigned char> &key, const
                 std::vector<unsigned char> &IV) noexcept(
                 true);

bool decrypt_cfb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
              const std::vector<unsigned char> &key, const
                 std::vector<unsigned char> &IV) noexcept(
                 true);

bool encrypt_ofb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
              const std::vector<unsigned char> &key, const
                 std::vector<unsigned char> &IV) noexcept(
                 true);

bool decrypt_ofb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
```

```
                const std::vector<unsigned char> &key, const
                   std::vector<unsigned char> &IV) noexcept(
                   true);

void printEncryptOutput(std::vector<unsigned char> &output);

void printDecryptOutput(std::vector<unsigned char> &output);

#endif
```

```
/**
  @file AESmodes.cpp
  Implementation of modes of operation for AES-128
*/
#include "AESmodes.hpp"
#include <iostream>

bool remove_padding(std::vector<unsigned char> &input) noexcept
   (false) {
    const int lastByte = (int) input.back();
    //Only continue if the last byte is in the valid range
    if (lastByte <= NUM_BYTES && lastByte > 0) {
        //Verify that the padding is okay
        for (std::size_t i = 0; i < lastByte; i++) {
            if (input.at(input.size() - i - 1) != lastByte)
                //Do nothing and return
                return false;
        }
        input.erase(input.end() - lastByte, input.end());
        return true;
    }
    //If an improper padding value was given, also do nothing
    return false;
}

/**
  Cipher with ECB mode
```

```
  Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
            exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
        library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input and key are constant and output vector is
                cleared when catching an exception
  @param input: vector of hex values representing plaintext
  @param output: vector of hex values representing (padded)
     ciphertext
  @param key: vector of hex values representing key to use
  @return True on success
*/
bool encrypt_ecb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
                const std::vector<unsigned char> &key) noexcept
                    (true) {
    try {
        // Calculate padding length, then copy input array and
           padding into plaintext
        // Secure coding: CTR50-CPP. Guarantee that container
           indices and iterators are within the valid range
        const std::size_t inputSize = input.size();
        const std::size_t padLength = NUM_BYTES - (inputSize %
           NUM_BYTES);
        const std::size_t plaintextLength = inputSize +
           padLength;

        // Plaintext accommodates both the input and the
           necessary padding
        std::vector<unsigned char> plaintext;
        // Secure coding: OOP57-CPP. Prefer special member
           functions and overloaded operators to C Standard
           Library functions
        plaintext.reserve(plaintextLength);
        plaintext = input;
```

```
    for (std::size_t i = 0; i < padLength; i++) {
        // PKCS#7 padding (source: https://www.ibm.com/docs/
            en/zos/2.1.0?topic=rules-pkcs-padding-method)
        plaintext.push_back(padLength);
    }


    // Loop over number of blocks
    for (std::size_t i = 0; i < plaintextLength / NUM_BYTES;
         i++) {

        // Loop over block size and fill each block
        std::array<unsigned char, NUM_BYTES> block{0};
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Using .at instead of [] for internal bounds
                checking, see CTR50-CPP
            block.at(j) = plaintext.at(j + (i * NUM_BYTES));
        }

        // Encrypt each block
        std::array<unsigned char, NUM_BYTES> outputBlock{};

        encrypt(block, outputBlock, key);

        // Copy encrypted block to the output
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
                container indices and iterators are within
                the valid range
            output.push_back(outputBlock.at(j));
        }
    }

} catch (std::exception &e) {
    //Catch exception by lvalue or reference per ERR61-CPP
    std::cout << "Encryption Error" << std::endl;
    output.clear();
```

```
        return false;
    }
    return true;
}


/**
  Inverse cipher with ECB mode
  Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
            exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
        library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input and key are constant and output vector is
                cleared when catching an exception
  @param input: vector of hex values representing (padded)
     ciphertext
  @param output: vector of hex values representing plaintext (
     without padding)
  @param key: vector of hex values representing key to use
  @return True on success
*/
bool decrypt_ecb(const std::vector<unsigned char> &input, std::
    vector<unsigned char> &output,
                 const std::vector<unsigned char> &key) noexcept
                     (true) {
    try {
        const std::size_t inputSize = input.size();

        // Loop over number of blocks
        for (std::size_t i = 0; i < inputSize / NUM_BYTES; i++)
             {

            // Loop over block size and fill each block
            std::array<unsigned char, NUM_BYTES> block{0};
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
```

```
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            block.at(j) = input.at(j + (i * NUM_BYTES));
        }

        // Decrypt each block
        std::array<unsigned char, NUM_BYTES> outputPadded
           {0};
        decrypt(block, outputPadded, key);

        // Copy decrypted block to the output
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            output.push_back(outputPadded.at(j));
        }
    }

    if (!remove_padding(output)) {
        std::cout << "Decryption Error" << std::endl;
        //Erase the output to avoid any other information
           leaking
        output.clear();
        return false;
    }

} catch (std::exception &e) {
    //Catch exception by lvalue or reference per ERR61-CPP
    std::cout << "Decryption Error" << std::endl;
    output.clear();
    return false;
}

return true;

}
```

```
/**
  Cipher with CBC mode
  Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
           exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
       library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and IV are constant and output vector is
                 cleared when catching an exception
  @param input: vector of hex values representing plaintext
  @param output: vector of hex values representing (padded)
     ciphertext
  @param key: vector of hex values representing key to use
  @param IV: initialization vector to use
  @return True on success
*/
bool encrypt_cbc(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
               const std::vector<unsigned char> &key, const
                  std::vector<unsigned char> &IV) noexcept(
                  true) {
    try {
        // Calculate padding length, then copy input array and
           padding into plaintext
        const std::size_t inputSize = input.size();
        const std::size_t padLength = NUM_BYTES - (inputSize %
           NUM_BYTES);
        const std::size_t plaintextLength = inputSize +
           padLength;

        //std::cout << std::hex << inputSize << std::endl;

        // Plaintext accommodates both the input and the
           necessary padding
        std::vector<unsigned char> plaintext;
```

```
plaintext.reserve(plaintextLength);
plaintext = input;

for (std::size_t i = 0; i < padLength; i++) {
    // PKCS#7 padding (source: https://www.ibm.com/docs/
       en/zos/2.1.0?topic=rules-pkcs-padding-method)
    plaintext.push_back(padLength);
}

// Encrypt the first block
std::array<unsigned char, NUM_BYTES> block{0};
for (std::size_t j = 0; j < NUM_BYTES; j++) {
    // Secure coding: CTR50-CPP. Guarantee that
       container indices and iterators are within the
       valid range
    block.at(j) = plaintext.at(j) ^ IV.at(j);
}

std::array<unsigned char, NUM_BYTES> outputBlock{0};
encrypt(block, outputBlock, key);

for (std::size_t j = 0; j < NUM_BYTES; j++) {
    // Secure coding: CTR50-CPP. Guarantee that
       container indices and iterators are within the
       valid range
    output.push_back(outputBlock.at(j));
}

// Loop over the number of subsequent blocks
for (std::size_t i = 1; i < plaintextLength / NUM_BYTES;
     i++) {
    // Loop over block size and fill each block
    for (std::size_t j = 0; j < NUM_BYTES; j++) {
        // Secure coding: CTR50-CPP. Guarantee that
           container indices and iterators are within
           the valid range
        block.at(j) = plaintext.at(j + (i * NUM_BYTES))
           ^ output.at(j + ((i - 1) * NUM_BYTES));
    }
```

```
            // Encrypt each block
            encrypt(block, outputBlock, key);

            // Copy encrypted block to the output
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
                // Secure coding: CTR50-CPP. Guarantee that
                    container indices and iterators are within
                    the valid range
                output.push_back(outputBlock.at(j));
            }
        }

    } catch (std::exception &e) {
        //Catch exception by lvalue or reference per ERR61-CPP
        std::cout << "Encryption Error" << std::endl;
        output.clear();
        return false;
    }
    return true;
}


/**
  Inverse cipher with CBC mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
            exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
        library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and IV are constant and output vector is
                cleared when catching an exception
  @param input: vector of hex values representing (padded)
     ciphertext
  @param output: vector of hex values representing plaintext (
     without padding)
```

```
  @param key: vector of hex values representing key to use
  @param IV: initialization vector to use
  @return True on success
*/
bool decrypt_cbc(const std::vector<unsigned char> &input, std::
    vector<unsigned char> &output,
                 const std::vector<unsigned char> &key, const
                     std::vector<unsigned char> &IV) noexcept(
                     true) {
    try {
        const std::size_t inputSize = input.size();

        // Decrypt the first block
        std::array<unsigned char, NUM_BYTES> block{0};
        for (std::size_t i = 0; i < NUM_BYTES; i++) {
            // Secure coding: CTR50-CPP. Guarantee that
                container indices and iterators are within the
                valid range
            block.at(i) = input.at(i);
        }

        std::array<unsigned char, NUM_BYTES> outputPadded{0};
        decrypt(block, outputPadded, key);

        for (std::size_t i = 0; i < NUM_BYTES; i++) {
            // Secure coding: CTR50-CPP. Guarantee that
                container indices and iterators are within the
                valid range
            outputPadded.at(i) ^= IV.at(i);
            output.push_back(outputPadded.at(i));
        }

        // Loop over the number of subsequent blocks
        for (std::size_t i = 1; i < inputSize / NUM_BYTES; i++)
             {

            // Loop over block size and fill each block
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
```

```
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            block.at(j) = input.at(j + (i * NUM_BYTES));
        }

        // Decrypt each block
        decrypt(block, outputPadded, key);

        // Copy decrypted block to the output
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            outputPadded.at(j) ^= input.at(j + ((i - 1) *
               NUM_BYTES));
            output.push_back(outputPadded.at(j));
        }
    }


    // Remove padding
    if (!remove_padding(output)) {
        std::cout << "Decryption Error" << std::endl;
        //Erase the output to avoid any other information
           leaking
        output.clear();
        return false;
    }

} catch (std::exception &e) {
    //Catch exception by lvalue or reference per ERR61-CPP
    std::cout << "Decryption Error" << std::endl;
    output.clear();
    return false;
}
return true;

}
```

```
/**
  increments the counter block by one
  @param counter: array of values containing a nonce and a
     counter section
  @param numCounterBytes: the number of bytes in the counter
     array that are actually part of the counter
  @return none
*/
void incrementCounter(std::array<unsigned char, NUM_BYTES> &
    counter, int numCounterBytes) noexcept(false) {
    for (int i = NUM_BYTES - 1; i >= numCounterBytes; i--) {
        //Increment the current byte
        // Secure coding: CTR50-CPP. Guarantee that container
           indices and iterators are within the valid range
        counter.at(i) = counter.at(i) + 1;
        //If the byte did not overflow to zero, then stop
        //Otherwise continue until an overflow does not happen
        if (counter.at(i) != 0)
            break;
    }
}

/**
  cipher with CTR mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
           exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
       library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and nonce are constant and output vector
                is cleared when catching an exception
  @param input: vector of hex values representing the plaintext
  @param output: vector of hex values representing ciphertext (
     with padding)
  @param key: vector of hex values representing key to use
```

```
  @param nonce: a NUM_BYTES/2 (8) byte random block for the
     counter
  @return True on success
*/
bool encrypt_ctr(const std::vector<unsigned char> &input, std::
    vector<unsigned char> &output,
                const std::vector<unsigned char> &key,
                const std::array<unsigned char, NUM_BYTES / 2>
                   &nonce) noexcept(true) {
    try {
        // Calculate padding length, then copy input array and
           padding into plaintext
        const std::size_t inputSize = input.size();
        const std::size_t padLength = NUM_BYTES - (inputSize %
           NUM_BYTES);
        const std::size_t plaintextLength = inputSize +
           padLength;

        // Plaintext accommodates both the input and the
           necessary padding
        std::vector<unsigned char> plaintext;
        plaintext.reserve(plaintextLength);
        plaintext = input;

        for (std::size_t i = 0; i < padLength; i++) {
            // PKCS#7 padding (source: https://www.ibm.com/docs/
               en/zos/2.1.0?topic=rules-pkcs-padding-method)
            plaintext.push_back(padLength);
        }

        std::array<unsigned char, NUM_BYTES> counter{0};
        counter.fill(0);
        std::copy(nonce.begin(), nonce.end(), counter.begin());

        std::array<unsigned char, NUM_BYTES> outputBlock{0};
        outputBlock.fill(0);

        for (std::size_t i = 0; i < plaintextLength / NUM_BYTES;
             i++) {
```

```
            //Encrypt the counter
            encrypt(counter, outputBlock, key);

            //XOR output with the plaintext and put into output
               block
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
                // Secure coding: CTR50-CPP. Guarantee that
                   container indices and iterators are within
                   the valid range
                output.push_back(plaintext.at(j + (i * NUM_BYTES
                   )) ^ outputBlock.at(j));
            }

            incrementCounter(counter, NUM_BYTES / 2);
        }

    } catch (std::exception &e) {
        //Catch exception by lvalue or reference per ERR61-CPP
        std::cout << "Encryption Error" << std::endl;
        output.clear();
        return false;
    }
    return true;

}

/**
  inverse cipher with CTR mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
           exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
       library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and nonce are constant and output vector
               is cleared when catching an exception
```

```
  @param input: vector of hex values representing the
     ciphertext (with padding)
  @param output: vector of hex values representing plaintext (
     without padding)
  @param key: vector of hex values representing key to use
  @param nonce: a NUM_BYTES/2 (8) byte random block for the
     counter
  @return True on success
*/
bool decrypt_ctr(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
               const std::vector<unsigned char> &key,
               const std::array<unsigned char, NUM_BYTES / 2>
                  &nonce) noexcept(true) {
    try {

        const std::size_t inputSize = input.size();

        std::array<unsigned char, NUM_BYTES> counter{0};
        // Secure coding: OOP57-CPP. Prefer special member
           functions and overloaded operators to C Standard
           Library functions
        counter.fill(0);
        std::copy(nonce.begin(), nonce.end(), counter.begin());

        std::array<unsigned char, NUM_BYTES> outputBlock{0};
        // Secure coding: OOP57-CPP. Prefer special member
           functions and overloaded operators to C Standard
           Library functions
        outputBlock.fill(0);

        for (std::size_t i = 0; i < inputSize / NUM_BYTES; i++)
             {
            //Encrypt the counter
            encrypt(counter, outputBlock, key);

            //XOR output with the plaintext and put into output
               block
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
```

```
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            output.push_back(input.at(j + (i * NUM_BYTES)) ^
                 outputBlock.at(j));
        }

        incrementCounter(counter, NUM_BYTES / 2);
    }

    // Remove padding
    if (!remove_padding(output)) {
        std::cout << "Decryption Error" << std::endl;
        //Erase the output to avoid any other information
           leaking
        output.clear();
        return false;
    }

  } catch (std::exception &e) {
    //Catch exception by lvalue or reference per ERR61-CPP
    std::cout << "Decryption Error" << std::endl;
    output.clear();
    return false;
  }
  return true;
}

/**
  Cipher with CFB128 mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
           exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
       library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
```

```
        Input, key, and IV are constant and output vector is
            cleared when catching an exception
  @param input: vector of hex values representing plaintext
  @param output: vector of hex values representing (padded)
     ciphertext
  @param key: vector of hex values representing key to use
  @param IV: initialization vector to use
  @return True on success
*/
bool encrypt_cfb(const std::vector<unsigned char> &input, std::
   vector<unsigned char> &output,
               const std::vector<unsigned char> &key, const
                  std::vector<unsigned char> &IV) noexcept(
                  true) {
    try {

        // Calculate padding length, then copy input array and
           padding into plaintext
        const std::size_t inputSize = input.size();
        const std::size_t padLength = NUM_BYTES - (inputSize %
           NUM_BYTES);
        const std::size_t plaintextLength = inputSize +
           padLength;

        // Plaintext accomodates both the input and the
           necessary padding
        std::vector<unsigned char> plaintext;
        plaintext.reserve(plaintextLength);
        plaintext = input;

        for (std::size_t i = 0; i < padLength; i++) {
            // PKCS#7 padding (source: https://www.ibm.com/docs/
               en/zos/2.1.0?topic=rules-pkcs-padding-method)
            plaintext.push_back(padLength);
        }

        // Encrypt the first block
        std::array<unsigned char, NUM_BYTES> block{0};
        std::array<unsigned char, NUM_BYTES> outputBlock{0};
```

```
    std::copy(IV.begin(), IV.end(), block.begin());

    encrypt(block, outputBlock, key);

    for (std::size_t j = 0; j < NUM_BYTES; j++) {
        // Secure coding: CTR50-CPP. Guarantee that
           container indices and iterators are within the
           valid range
        output.push_back(outputBlock.at(j) ^ plaintext.at(j)
           );
    }

    // Loop over the number of subsequent blocks
    for (std::size_t i = 1; i < plaintextLength / NUM_BYTES;
         i++) {
        // Loop over block size and fill each block
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            block.at(j) = output.at(j + ((i - 1) * NUM_BYTES
               ));
        }

        // Encrypt each block
        encrypt(block, outputBlock, key);

        // Copy encrypted block to the output
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
               container indices and iterators are within
               the valid range
            output.push_back(outputBlock.at(j) ^ plaintext.
               at(j + (i * NUM_BYTES)));
        }
    }

} catch (std::exception &e) {
```

```
        //Catch exception by lvalue or reference per ERR61-CPP
        std::cout << "Encryption Error" << std::endl;
        output.clear();
        return false;
    }
    return true;
}

/**
  Inverse cipher with CFB128 mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
            exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
        library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and IV are constant and output vector is
                cleared when catching an exception
  @param input: vector of hex values representing (padded)
      ciphertext
  @param output: vector of hex values representing plaintext (
      without padding)
  @param key: vector of hex values representing key to use
  @param IV: initialization vector to use
  @return True on success
*/
bool decrypt_cfb(const std::vector<unsigned char> &input, std::
    vector<unsigned char> &output,
                 const std::vector<unsigned char> &key, const
                     std::vector<unsigned char> &IV) noexcept(
                     true) {
    try {
        const std::size_t inputSize = input.size();

        // Encrypt the first block
        std::array<unsigned char, NUM_BYTES> block{0};
        std::array<unsigned char, NUM_BYTES> outputBlock{0};
```

```
}
```

```
// Secure coding: OOP57-CPP. Prefer special member
   functions and overloaded operators to C Standard
   Library functions
std::copy(IV.begin(), IV.end(), block.begin());

encrypt(block, outputBlock, key);

for (std::size_t j = 0; j < NUM_BYTES; j++) {
    // Secure coding: CTR50-CPP. Guarantee that
       container indices and iterators are within the
       valid range
    output.push_back(outputBlock.at(j) ^ input.at(j));
}

// Loop over the number of subsequent blocks
for (std::size_t i = 1; i < inputSize / NUM_BYTES; i++)
     {
    // Loop over block size and fill each block
    for (std::size_t j = 0; j < NUM_BYTES; j++) {
        // Secure coding: CTR50-CPP. Guarantee that
           container indices and iterators are within
           the valid range
        block.at(j) = input.at(j + ((i - 1) * NUM_BYTES)
           );
    }

    // Encrypt each block
    encrypt(block, outputBlock, key);

    // Copy encrypted block to the output
    for (std::size_t j = 0; j < NUM_BYTES; j++) {
        // Secure coding: CTR50-CPP. Guarantee that
           container indices and iterators are within
           the valid range
        output.push_back(outputBlock.at(j) ^ input.at(j +
            (i * NUM_BYTES)));
    }
}
```

```
        // Remove padding
        if (!remove_padding(output)) {
            std::cout << "Decryption Error" << std::endl;
            //Erase the output to avoid any other information
                leaking
            output.clear();
            return false;
        }

    } catch (std::exception &e) {
        //Catch exception by lvalue or reference per ERR61-CPP
        std::cout << "Decryption Error" << std::endl;
        output.clear();
        return false;
    }
    return true;
}

/**
  Cipher with OFB mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
            exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
        library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and IV are constant and output vector is
                cleared when catching an exception
  @param input: vector of hex values representing plaintext
  @param output: vector of hex values representing (padded)
     ciphertext
  @param key: vector of hex values representing key to use
  @param IV: initialization vector to use
  @return True on success
*/
bool encrypt_ofb(const std::vector<unsigned char> &input, std::::
   vector<unsigned char> &output,
```

```
            const std::vector<unsigned char> &key, const
               std::vector<unsigned char> &IV) noexcept(
               true) {
try {
    // Calculate padding length, then copy input array and
       padding into plaintext
    const std::size_t inputSize = input.size();
    const std::size_t padLength = NUM_BYTES - (inputSize %
       NUM_BYTES);
    const std::size_t plaintextLength = inputSize +
       padLength;

    // Plaintext accommodates both the input and the
       necessary padding
    std::vector<unsigned char> plaintext;
    plaintext.reserve(plaintextLength);
    plaintext = input;

    for (std::size_t i = 0; i < padLength; i++) {
        // PKCS#7 padding (source: https://www.ibm.com/docs/
           en/zos/2.1.0?topic=rules-pkcs-padding-method)
        plaintext.push_back(padLength);
    }

    // Encrypt the first block
    std::array<unsigned char, NUM_BYTES> block{0};
    std::array<unsigned char, NUM_BYTES> outputBlock{0};

    // Secure coding: OOP57-CPP. Prefer special member
       functions and overloaded operators to C Standard
       Library functions
    std::copy(IV.begin(), IV.end(), block.begin());

    encrypt(block, outputBlock, key);

    for (std::size_t j = 0; j < NUM_BYTES; j++) {
        // Secure coding: CTR50-CPP. Guarantee that
           container indices and iterators are within the
           valid range
```

```
            output.push_back(outputBlock.at(j) ^ plaintext.at(j)
                );
        }

        // Loop over the number of subsequent blocks
        for (std::size_t i = 1; i < plaintextLength / NUM_BYTES;
             i++) {
            // Loop over block size and fill each block
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
                // Secure coding: CTR50-CPP. Guarantee that
                    container indices and iterators are within
                    the valid range
                block.at(j) = outputBlock.at(j);
            }

            // Encrypt each block
            encrypt(block, outputBlock, key);

            // Copy encrypted block to the output
            for (std::size_t j = 0; j < NUM_BYTES; j++) {
                // Secure coding: CTR50-CPP. Guarantee that
                    container indices and iterators are within
                    the valid range
                output.push_back(outputBlock.at(j) ^ plaintext.
                    at(j + (i * NUM_BYTES)));
            }
        }

    } catch (std::exception &e) {
        //Catch exception by lvalue or reference per ERR61-CPP
        std::cout << "Encryption Error" << std::endl;
        output.clear();

        return false;
    }
    return true;
}

/**
```

```
  Inverse cipher with OFB mode
    Guaranteed no exceptions by:
    handling all exceptions per ERR51-CPP
        Related: Honoring exception specifications, all
            exceptions will be caught per ERR55-CPP
    not throwing exceptions across execution boundaries (
        library to application) per ERR59-CPP
    Guaranteeing Strong exception safety per ERR56-CPP
        Program state will not be modified
            Input, key, and IV are constant and output vector is
                cleared when catching an exception
  @param input: vector of hex values representing (padded)
      ciphertext
  @param output: vector of hex values representing plaintext (
      without padding)
  @param key: vector of hex values representing key to use
  @param IV: initialization vector to use
  @return True on success
*/
bool decrypt_ofb(const std::vector<unsigned char> &input, std::
    vector<unsigned char> &output,
                 const std::vector<unsigned char> &key, const
                     std::vector<unsigned char> &IV) noexcept(
                     true) {
    try {

        const std::size_t inputSize = input.size();

        // Encrypt the first block
        std::array<unsigned char, NUM_BYTES> block{0};
        std::array<unsigned char, NUM_BYTES> outputBlock{0};

        // Secure coding: OOP57-CPP. Prefer special member
            functions and overloaded operators to C Standard
            Library functions
        std::copy(IV.begin(), IV.end(), block.begin());

        encrypt(block, outputBlock, key);
```

```
    for (std::size_t j = 0; j < NUM_BYTES; j++) {
        // Secure coding: CTR50-CPP. Guarantee that
            container indices and iterators are within the
            valid range
        output.push_back(outputBlock.at(j) ^ input.at(j));
    }

    // Loop over the number of subsequent blocks
    for (std::size_t i = 1; i < inputSize / NUM_BYTES; i++)
         {
        // Loop over block size and fill each block
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
                container indices and iterators are within
                the valid range
            block.at(j) = outputBlock.at(j);
        }

        // Encrypt each block
        encrypt(block, outputBlock, key);

        // Copy encrypted block to the output
        for (std::size_t j = 0; j < NUM_BYTES; j++) {
            // Secure coding: CTR50-CPP. Guarantee that
                container indices and iterators are within
                the valid range
            output.push_back(outputBlock.at(j) ^ input.at(j +
                 (i * NUM_BYTES)));
        }
    }
    // Remove padding
    if (!remove_padding(output)) {
        std::cout << "Decryption Error" << std::endl;
        //Erase the output to avoid any other information
            leaking
        output.clear();
        return false;
    }
} catch (std::exception &e) {
```

```
        //Catch exception by lvalue or reference per ERR61-CPP
        std::cout << "Decryption Error" << std::endl;
        output.clear();
        return false;
    }

    return true;
}
```

```
/**
  @file encrypt.hpp: prototypes for encryption
*/
#ifndef ENCRYPT_HPP
#define ENCRYPT_HPP

#include "AESmath.hpp"
#include <array>

void encrypt(std::array<unsigned char, 16>& input, std::array<
   unsigned char, 16>& output, const std::vector<unsigned char
   >& key); // Secure coding: DCL52-CPP. Never qualify a
   reference type with const or volatile
void subBytes(std::array<unsigned char, NUM_BYTES>& state);
void shiftRows(std::array<unsigned char, NUM_BYTES>& state);
void mixColumns(std::array<unsigned char, NUM_BYTES>& state);

#endif
```

```
/**
  @file encrypt.cpp: Cipher implementation
*/
#include "encrypt.hpp"
#include <iostream>
```

```
/**
	Substitutes bytes in the state for bytes from a
	   substitution box
	@param state: the state array to modify
	@return none
*/
void subBytes(std::array<unsigned char, NUM_BYTES>& state) {
	for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
	     coding: CTR50-CPP. Guarantee that container indices
	     and iterators are within the valid range
		state[i] = getSboxValue(state[i]);
	}
}

/**
	Left shifts the bytes in each row of the state based on
	    that rows index, i.e. row 0 gets no shift, row 1
	   once to left...
	@param state: the state array to modify
	@return none
*/
void shiftRows(std::array<unsigned char, NUM_BYTES>& state) {
	std::array<unsigned char, NUM_BYTES> shiftedState;

	// Row 1 - Bytes remain unchanged
	shiftedState[0] = state[0];
	shiftedState[4] = state[4];
	shiftedState[8] = state[8];
	shiftedState[12] = state[12];

	// Row 2 - Bytes are shifted over one position to the
	   left
	shiftedState[1] = state[5];
	shiftedState[5] = state[9];
	shiftedState[9] = state[13];
	shiftedState[13] = state[1];

	// Row 2 - Bytes are shifted over two positions to the
	   left
```

```
        shiftedState[2] = state[10];
        shiftedState[6] = state[14];
        shiftedState[10] = state[2];
        shiftedState[14] = state[6];

        // Row 4 - Bytes are shifted over three positions to
           the left
        shiftedState[3] = state[15];
        shiftedState[7] = state[3];
        shiftedState[11] = state[7];
        shiftedState[15] = state[11];

        //Replace the state array with the shifted bytes
        for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
            coding: CTR50-CPP. Guarantee that container indices
            and iterators are within the valid range
               state[i] = shiftedState[i]; // Secure coding:
                  OOP57-CPP. Prefer special member functions
                  and overloaded operators to C Standard
                  Library functions
        }
}

/**
        mixColumns, multiplies the columns of the state by the
           polynomial {02}, {03} shifted around the rows
        @param state: the state array to modify
        @return none
*/
void mixColumns(std::array<unsigned char, NUM_BYTES>& state) {
        std::array<unsigned char, NUM_BYTES> tmp;

        for (std::size_t i = 0; i < 4; i++) { // Secure coding:
            CTR50-CPP. Guarantee that container indices and
           iterators are within the valid range
               tmp[4 * i] = galoisFieldMult(0x02, state[i * 4])
                   ^ galoisFieldMult(0x03, state[i * 4 + 1]) ^
                  state[i * 4 + 2] ^ state[i * 4 + 3];
```

```
            tmp[4 * i + 1] = state[i * 4] ^ galoisFieldMult
                (0x02, state[i * 4 + 1]) ^ galoisFieldMult(0
                x03, state[i * 4 + 2]) ^ state[i * 4 + 3];
            tmp[4 * i + 2] = state[i * 4] ^ state[i * 4 + 1]
                 ^ galoisFieldMult(0x02, state[i * 4 + 2]) ^
                galoisFieldMult(0x03, state[i * 4 + 3]);
            tmp[4 * i + 3] = galoisFieldMult(0x03, state[i *
                 4]) ^ state[i * 4 + 1] ^ state[i * 4 + 2] ^
                galoisFieldMult(0x02, state[i * 4 + 3]);
        }

        //Replace the state array with the mixed bytes
        for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
             coding: CTR50-CPP. Guarantee that container indices
             and iterators are within the valid range
            state[i] = tmp[i]; // Secure coding: OOP57-CPP.
                Prefer special member functions and
                overloaded operators to C Standard Library
                functions
        }
}


/**
  Cipher, which implements shiftRows, sSubBytesand mixColumns
  @param input: array of hex values representing the input
     bytes
  @param output: array of hex values that is copied to from
     final state
  @param key: vector of hex values representing key to use
  @return none
*/
void encrypt(std::array<unsigned char, 16>& input, std::array<
   unsigned char, 16>& output, const std::vector<unsigned char
   >& key) {
  // Create the state array from input
  std::array<unsigned char, NUM_BYTES> state;
        for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
             coding: CTR50-CPP. Guarantee that container indices
```

```
            and iterators are within the valid range
               state[i] = input[i]; // Secure coding: OOP57-CPP.
                   Prefer special member functions and
                  overloaded operators to C Standard Library
                  functions
      }

// Expand key
const std::size_t keysize = key.size();
const std::size_t numRounds = keysize/4 + 6;
std::vector<unsigned char> expandedKey;
expandedKey.reserve(16 * (numRounds + 1));
      keyExpansion(key, expandedKey, keysize);

      // Intial Round
      addRoundKey(state, &(expandedKey[0]));

      for (std::size_t i = 0; i < numRounds-1; i++) { //
         Secure coding: CTR50-CPP. Guarantee that container
         indices and iterators are within the valid range
             subBytes(state);
             shiftRows(state);
             mixColumns(state);
             //The key index is supposed to be 4*roundNum but
                 since the key is bytes, it is 4*4*roundNum
             addRoundKey(state, &(expandedKey[16*(i+1)]));
      }

      // Final Round - No MixedColumns
      subBytes(state);
      shiftRows(state);
      addRoundKey(state, &(expandedKey[16*numRounds]));

      for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
          coding: CTR50-CPP. Guarantee that container indices
          and iterators are within the valid range
             output[i] = state[i]; // Secure coding: OOP57-
                CPP. Prefer special member functions and
                overloaded operators to C Standard Library
```

```
                    functions
        }
}
```

```
/**
  @file decrypt.hpp: Prototypes for decryption
*/
#ifndef DECRYPT_HPP
#define DECRYPT_HPP

#include "AESmath.hpp"
#include <array>


void decrypt(std::array<unsigned char, 16> input, std::array<
   unsigned char, 16>& output, const std::vector<unsigned char
   >& key); // Secure coding: DCL52-CPP. Never qualify a
   reference type with const or volatile
void invSubBytes(std::array<unsigned char, NUM_BYTES>& state);
void invShiftRows(std::array<unsigned char, NUM_BYTES>& state);
void invMixColumns(std::array<unsigned char, NUM_BYTES>& state)
   ;

#endif
```

```
/**
  @file decrypt.cpp: Inverse cipher implementation
*/
#include "decrypt.hpp"
#include <iostream>

/**
  Inverse of shiftRows(). Shifts bytes in last three rows of
     state over different offsets.
  @param state: state array to modify
```

```
  @return none
*/
void invShiftRows(std::array<unsigned char, NUM_BYTES>& state)
   {
  std::array<unsigned char, NUM_BYTES> shiftedState;

  // Row 1 - Bytes remain unchanged
  shiftedState[0] = state[0];
  shiftedState[4] = state[4];
  shiftedState[8] = state[8];
  shiftedState[12] = state[12];

  // Row 2 - Bytes are shifted over three positions to the left
  shiftedState[1] = state[13];
  shiftedState[5] = state[1];
  shiftedState[9] = state[5];
  shiftedState[13] = state[9];

  // Row 3 - Bytes are shifted over two positions to the left
  shiftedState[2] = state[10];
  shiftedState[6] = state[14];
  shiftedState[10] = state[2];
  shiftedState[14] = state[6];

  // Row 4 - Bytes are shifted over one position to the left
  shiftedState[3] = state[7];
  shiftedState[7] = state[11];
  shiftedState[11] = state[15];
  shiftedState[15] = state[3];

  for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
      coding: CTR50-CPP. Guarantee that container indices and
      iterators are within the valid range
    state[i] = shiftedState[i]; // Secure coding: OOP57-CPP.
        Prefer special member functions and overloaded operators
         to C Standard Library functions
  }
}
```

```
/**
  Inverse of subBytes(). Replaces each byte of state with
     computed inverse sbox value.
  @param state: state array to modify
  @return none
*/
void invSubBytes(std::array<unsigned char, NUM_BYTES>& state) {
  for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
     coding: CTR50-CPP. Guarantee that container indices and
     iterators are within the valid range
    state[i] = invGetSboxValue(state[i]);
  }
}


/**
  Inverse of mixColumns(). Multiplies columns of state by
     polynomial {0b},{0d},{09},{0e} mod x^4 +1 over GF(2^8).
  @param state: state array to modify
  @return none
*/
void invMixColumns(std::array<unsigned char, NUM_BYTES>& state)
    {
  std::array<unsigned char, NUM_BYTES> tmp;

        for (std::size_t i = 0; i < 4; i++) { // Secure coding:
             CTR50-CPP. Guarantee that container indices and
           iterators are within the valid range
                tmp[4 * i] = galoisFieldMult(0x0e, state[i * 4])
                     ^ galoisFieldMult(0x0b, state[i * 4 + 1]) ^
                    galoisFieldMult(0x0d, state[i * 4 + 2]) ^
                    galoisFieldMult(0x09, state[i * 4 + 3]);
                tmp[4 * i + 1] = galoisFieldMult(0x09, state[i *
                     4]) ^ galoisFieldMult(0x0e, state[i * 4 +
                    1]) ^ galoisFieldMult(0x0b, state[i * 4 + 2])
                     ^ galoisFieldMult(0x0d, state[i * 4 + 3]);
                tmp[4 * i + 2] = galoisFieldMult(0x0d, state[i *
                     4]) ^ galoisFieldMult(0x09, state[i * 4 +
```

```
                1]) ^ galoisFieldMult(0x0e, state[i * 4 + 2])
                 ^ galoisFieldMult(0x0b, state[i * 4 + 3]);
            tmp[4 * i + 3] = galoisFieldMult(0x0b, state[i *
                4]) ^ galoisFieldMult(0x0d, state[i * 4 +
                1]) ^ galoisFieldMult(0x09, state[i * 4 + 2])
                 ^ galoisFieldMult(0x0e, state[i * 4 + 3]);
        }

        for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
            coding: CTR50-CPP. Guarantee that container indices
            and iterators are within the valid range
            state[i] = tmp[i]; // Secure coding: OOP57-CPP.
                Prefer special member functions and
                overloaded operators to C Standard Library
                functions
        }
}


/**
  Inverse cipher, which implements invShiftRows, invSubBytes,
     invMixColumns
  @param input: array of hex values representing output of
     cipher
  @param output: array of hex values that is copied to from
     final state
  @param key: key to use
  @param keysize: size of the key
  @return none
*/
void decrypt(std::array<unsigned char, 16> input, std::array<
    unsigned char, 16>& output, const std::vector<unsigned char
    >& key) {
  // Create the state array from input
  std::array<unsigned char, NUM_BYTES> state;
  for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
     coding: CTR50-CPP. Guarantee that container indices and
     iterators are within the valid range
```

```
    state[i] = input[i]; // Secure coding: OOP57-CPP. Prefer
       special member functions and overloaded operators to C
       Standard Library functions
  }

  // Expand key
  const std::size_t keysize = key.size();
  const std::size_t numRounds = keysize/4 + 6;
  std::vector<unsigned char> expandedKey;
  expandedKey.reserve(16 * (numRounds + 1));
        keyExpansion(key, expandedKey, keysize);

  // Initial round
  addRoundKey(state, &(expandedKey[numRounds*NUM_BYTES]));

  // Rounds
  for (std::size_t round = numRounds-1; round > 0; round--){ //
      Secure coding: CTR50-CPP. Guarantee that container
     indices and iterators are within the valid range
    invShiftRows(state);
    invSubBytes(state);
    addRoundKey(state, &(expandedKey[round*NUM_BYTES]));
    invMixColumns(state);
  }

  // Final round
  invShiftRows(state);
  invSubBytes(state);
  addRoundKey(state, &(expandedKey[0]));

  // Set output to state
  for (std::size_t i = 0; i < NUM_BYTES; i++) { // Secure
     coding: CTR50-CPP. Guarantee that container indices and
     iterators are within the valid range
    output[i] = state[i]; // Secure coding: OOP57-CPP. Prefer
       special member functions and overloaded operators to C
       Standard Library functions
  }
}
```

8 Appendix B

The following is a list of cypto specific coding practices that we have learned.

- Avoiding memory accesses to mitigate cache timing attacks.
- Using a random number generator that uses external entropy to generate cryptographically secure random numbers.
- Avoiding random number generators with states that can be calculated from a small number of outputs. Specifically C++ `std::mt19937` Mersenne Twister generator.
- Avoiding acknowledgement of padding errors to mitigate padding oracle attacks.

9 Appendix C

The following is a list of secure coding practices we learned and used followed by a list of secure coding practices we learned and did not use. Each list item is a guideline from the SEI CERT C++ Coding Standards [10].

Coding practices learned and used:

- DCL52-CPP. Never qualify a reference type with const or volatile
- CTR50-CPP. Guarantee that container indices and iterators are within the valid range
- ERR51-CPP. Handle all exceptions
- ERR55-CPP. Honor exception specifications
- ERR59-CPP. Do not throw an exception across execution boundaries
- ERR56-CPP. Guarantee exception safety
- OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions
- MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers

Coding practices learned and not used:

- ERR57-CPP. Do not leak resources when handling exceptions
- ERR53-CPP. Do not reference base classes or class data members in a constructor or destructor function-try-block handler