Implementing the AES: Documentation

Benedikt Kluss, Hendrik Hagedorn

Seminar Thesis – February 5, 2021.
Research Group for Cyber Security

Principal Supervisor: Prof. Dr.-Ing. Thomas Hupperich
Associate Supervisor: Samaneh Rashidibajgan

Contents

1 Introduction

The document at hand contains the documentation of the deliverables created in the context of the project seminar "Implementing the Advanced Encryption Standard", held at the Research Group for Cyber Security. It aims at providing an overview of the project outcome, explaining its structure, as well as how and why it was designed that way. The focus is primarily put on those core aspects of the implementation, which are required to illustrate how the developed resources and modules interact and contribute to the overall outcome of a working program offering AES functionality. More detailed implementation details are spared for the sake of clarity and comprehensibility.

In order to serve this purpose well, the documentation is structured as follows. At first, a usage guide to the developed functionality is given in chapter 2, before chapter 3 outlines the general design decisions the implementation is based on. After setting this general framework, the developed modules are depicted. This begins with the description of the general modules in chapter 4, as these are shared by both the encryption and decryption implementations. Subsequently, the encryption and its respective modules are introduced in chapter 5, before the document is concluded by the depiction of the decryption modules in chapter 6.

All general functionality, introduced in chapters 2 to 4, has been developed by both Benedikt and Hendrik, except for the sBox introduced in section 4.3 which was created solely by Benedikt. The rest of the work has been split as requested by the supervisors - the encryption program has been implemented by Benedikt whilst Hendrik prepared the decryption part.

2 Usage Guide

The program has been designed to be well portable, making it usable on several platforms. It is built using CMake and has been successfully compiled and tested using both LLVM- and GCC-Compilers on machines running Microsoft Windows NT Kernels as well as Unix OSs. After compilation, the two independently executable en- and decryption algorithms are available on the top level of the project directory. Running them provides the respective functionality via the interfaces described below. Theoretically, the program offers cryptographic text processing of any size, as long as the text length is storable in a type unsigned long variable. In practice, texts of up to 1 GB in size were processed without any issues, larger sizes were not tested. The program could also easily be modified to work on all other kinds of data. Currently, reading binary files like pictures and processing them is already possible, but the limitation arises from the default outputs being specified as .txt files. If they would be changed to any other type, the algorithm could handle the respective files as well.

2.1 Encryption Interface

In order to run the encryption, a console user dialogue is available. Initially, the user is asked to specify whether he wants to enable the hexadecimal mode, which will expect the input text and key to be given in hexadecimal representation and will produce similar output. The user can type either 'y' or 'n' (capitalisation is irrelevant), any other input will result in an error shutting the program down. Afterwards, the required inputs can be specified, but only files located in the Files/-directory will be considered. The user is first asked to enter the name of the text file containing the message and subsequently to do the same concerning the key. To avoid overflows and thus ensure run-stability, all user inputs are limited to 32 characters. The message can be of any size, in case of not equalling a multiple of the AES message block size, padding will be applied. The key is only accepted if it exactly fits the required size, otherwise an error is printed and the program is shut down (the current version only supports encryption with 128 bit keys stored in either 16 or 32 characters, depending on the representation). Entering wrong file names for either the message- or the key-file immediately results in an exit of the program. After the program retrieved all required data, the encryption is performed. Throughout this process, the user is regularly updated about the progress. Upon completion, the result is stored in the default-file found at path *Files/encryptedMessage.txt* and the program is finished with a user dialog.

2.2 Decryption Interface

The decryption functionality is offered via a console user dialogue as well (the underlying process is illustrated in annex A.3). After informing the user about the selected AES mode, the message input path can be entered (the current version only supports encryption with 128 bit keys stored in either 16 or 32 characters, depending on the representation). Like with the encryption, file-names of text-files containing the required data are requested and only files in the Files/-directory are considered. In order to specify the file-name, the user can enter up to 32 characters. This limit is set to avoid any overflows and ensure maximum run-stability. Any invalid inputs or file-names will result in an error and termination of the program. If the user input was valid, the program asks whether the input is stored in hexadecimal representation. The user is asked to specify his answer as 'y' or 'n' (case insensitive), but only in case a 'n' is entered, the answer is perceived as non-positive. The reason for this comes with the exit questions in later steps, where incorrect inputs should not lead to termination of the program, but the user should rather continue by accident and shut down the program himself instead of causing an accidental shutdown. In case the message length is not equal to a multiple of the AES message block size, the last incomplete block gets cut off and the user is asked whether he wants to continue nevertheless. A continuation at this point will not occur very often, as the encryption output will always be of the correct size, but might be favoured in some cases and is therefore made possible. After successfully retrieving the message, the same steps are performed to get the key input from the user. The only deviation from the message retrieval process is that a too-short key will cause an immediate error, whilst too long keys might also be cut to an appropriate size. When all data is successfully retrieved, the decryption process begins. Throughout, this process the user receives regular updates on the progress. Before the output gets stored, the user can specify whether it should be stored in hexadecimal representation. Subsequently, the output is stored in the default-output-file located at *Files/decryptedMessage.txt*. Independently from the reason the program is finished, either orderly or due to a handled error, the user has to enter an arbitrary input to finally terminate the program. This is required, so that error or success messages can also be read on consoles which automatically close on termination.

3 Overall Design Decisions

3.1 C11 - Programming Language

We have chosen C11 as programming language, since it seems to be the perfect fit given the requirements of portability and efficiency. C is one of the older, yet well established, programming languages, which a number of high-level programming languages are based on. For example, Perl, PHP, Python, R, Matlab, Mathematica were all developed in C [Mun20]. One of the main advantages of C, is the low-level access of memory, arising from its original purpose of being used to program operating systems [Gee19]. This is perceived to be an important factor in cryptography, as the algorithm should be able to handle big files. Additionally, C is, compared to other languages, very fast in terms of execution. As cryptography is usually not the main part of a program (e.g. in a mobile messenger), encryption and decryption should take as few of the given resources as possible.

3.2 Internal Data Representation

We decided to represent the data as unsigned char internally, even though this is not necessarily the most efficient way [emb06]. Nevertheless, potential gains from other implementations are expected to be rather little and the simpler and better readable code achieved through this representation seems favourable. As the size of chars equals 1 Byte and the AES operations are performed byte-wise too, this representation is especially fitting. In order to test the eventual loss in efficiency, two analogous implementations of the key expansion were developed, which only differed in their way of representing the data. Whilst one used unsigned char representation, the other condensed each word (which equals 4 bytes in the AES context) into one int. It showed that the integer representation leads to more lines-of-code in assembly language, supposedly because the integers had to be transformed back to chars to be usable in the AES. An analysis of runtime was also performed, running each implementation 1,000,000 times on both a Unix OS and a Microsoft Windows NT kernel. The results were surprisingly different. Whilst the char implementation outperformed the integer version by a lot on the Unix-based system, on Windows the result was the other way around, even though not as significantly. We, therefore, decided to stick with the unsigned char representation, as efficiency couldn't be enhanced otherwise and the good fit to the AES specification was preferred.

4 General Modules

The following chapter describes the modules that both encryption and decryption use. The source code of those models is located in AES/src. As the project uses its own test library it will be also explained at the end of this chapter. This library is located in the lib directory.

4.1 charText.h

```
struct charText{
    unsigned long text_length;
    unsigned char * text;
};
```

Listing 4.1: struct charText

In C, arrays can only be passed by reference which comes with a significant limitation. A function that receives a pointer as a parameter is unable to efficiently determine the size of the corresponding array. In order to tackle this issue, the charText struct has been developed and frequently used throughout the project. It encapsulates only two variables, a text of unsigned char, which is the format AES messages are represented in, and its according length, represented by a 32 bit long value.

4.2 aes_config.h

The purpose of the config-module is to efficiently provide all required AES parameters to any module at hand. It does so by using the directive *#define*, setting the parameters as symbolic constants, whose alias can be included in the code and will be substituted to the respective value by the pre-processor. This non-variable, symbolic representation is for example used with the *message_byte_size* or the *number_of_rounds_128bit_key*, equaling values of 16 and 10. This implementation also enforces the general coding principle, to only define repeatedly used values in a single central place.

4.3 sBoxes

This section describes how the S-Box look-up table is built for the module, which will be later used by key expansion and the encryption algorithm. The two methods of getSBox() and freeSBox() won't be described here as they are self-explaining.

4.3.1 initializeAesSBox()

```
void initializeAesSBox() {
    sbox = malloc(SIZE_OF_SBOX);
    unsigned char galois_field_element = 1, galois_field_inverse = 1;

    for (int i = 0; i < (SIZE_OF_SBOX / 2); ++i) {
        sbox[galois_field_element] = performAffineTransformation(galois_field_inverse);
        sbox[galois_field_inverse] = performAffineTransformation(galois_field_element);

        galois_field_element = getNextElementOfGaloisFieldGenerator(galois_field_element);
        galois_field_inverse = getNextInverseElementOfGaloisFieldGenerator(
             galois_field_inverse);
    }

    /* 0 is a special case since it has no inverse */
    sbox[0] = 0x63;

    sbox_is_initialized = 1;
}
```

Listing 4.2: initializeAesSBox()

For calculation of the S-Box, firstly an array of 256 bytes is initialised. This equals a 16*16 byte matrix. Due to the used generator, which will be explained later, the first two elements of it need to be initialised with 1. As 1 is the neutral element of the Galois Field, it is its own inverse. The performAffineTransformation() function is always called on the current element and its inverse, thus in the case of 1, it is redundant. This repetition is not strutted, as it will only be called once in the total runtime and is therefore not considered to significantly impair the efficiency. However, this multiple function call helps to increase efficiency by applying the function not only on the inverse, but the current element as well, thus avoiding a complete iteration over all generator values. The existence of an inverse and the cumulative law of the Galois Field allow this operation. Consequently, we need half of the iterations to calculate the S-Box. The termination criterion of the for-loop has been set accordingly.

Afterwards, the next element and its inverse are calculated through the generator that is described below. As the first element is an exception to the explained procedure, it is set after the loop. For avoiding several initialisations of the S-Box a flag is set at the end.

4.3.2 performAffineTransformation()

```
static unsigned char performAffineTransformation(unsigned char multiplicative_inverse) {
    unsigned char affine_transformed_element = multiplicative_inverse;
    for (int shifts = 1; shifts <= 4 ; ++shifts) {
        affine_transformed_element ^=
          performCircularShiftLeft(multiplicative_inverse, shifts);
    }
    affine_transformed_element ^= 0x63;
    return affine_transformed_element;
}
```

Listing 4.3: performAffineTransformation()

This function performs the affine transformation. As it includes multiple additions of circular shifts, it is performed iterative for better readability. The method of the circular shift is not specified in the following as its duty should be clear.

4.3.3 Generators for Rijandel's Finite field

```
static unsigned char getNextElementOfGaloisFieldGenerator(unsigned char galois_field_element)
      {
    galois_field_element ^=
      (galois_field_element << 1) ^ (galois_field_element & 0x80 ? 0x1B : 0);
    return galois_field_element;
}

static unsigned char getNextInverseElementOfGaloisFieldGenerator(
    unsigned char galois_field_inverse) {
    galois_field_inverse ^= galois_field_inverse << 1;
    galois_field_inverse ^= galois_field_inverse << 2;
    galois_field_inverse ^= galois_field_inverse << 4;
    galois_field_inverse ^= galois_field_inverse & 0x80 ? 0x09 : 0;

    return galois_field_inverse;
}
```

Listing 4.4: getNextElementOfGaloisFieldGenerator()

The two depicted functions contain a commonly used generator for Rijndael's Galois field. They use the smallest odd element, which is number 3 or x + 1 in the writing of the Galois field, to calculate the next element and its inverse. Each last appended if-condition is to avoid escaping the finite field.

4.4 keyExpansion.h

The module described in the following provides the functionality to expand cryptographic keys of different sizes according to the specification in [oST01]. In order to do so, access to the modules sBoxes.h and aes_config.h is required.

4.4.1 getExpanded128bitKey()

```
unsigned char * getExpanded128bitKey(unsigned char *key) {
    return createExpandedKey(key, KEY_LENGTH_128BIT_BYTE_SIZE, NUMBER_OF_ROUNDS_128BIT_KEY);
}
```

Listing 4.5: getExpanded128bitKey()

The function printed above can be used to expand a 128 bit key. The key has to be entered via the functions only parameter and represented as a pointer to an array containing 16 bytes. The function obtains and returns the 176 byte expanded key by passing the key, along with the correct parameters, onto the general key expansion function. This contribution of the additional parameters is the reason why the given function is offered, as it enables users to expand their keys without requiring precise knowledge about the correct parameters.

4.4.2 createExpandedKey()

```
static unsigned char * createExpandedKey(
  const unsigned char * key, int key_length_in_bytes, int number_of_aes_rounds) {
    const unsigned char BYTES_IN_EXPANDED_KEY =
      key_length_in_bytes * (number_of_aes_rounds + 1);
    unsigned char * expanded_key = malloc(BYTES_IN_EXPANDED_KEY);
    initializeRCONValue();

    memcpy(expanded_key, key, key_length_in_bytes);
    for (int word = key_length_in_bytes; word < BYTES_IN_EXPANDED_KEY; word += BYTES_IN_WORD)
     {
        expanded_key = fillInNextWord(expanded_key, key_length_in_bytes, word);
    }

    return expanded_key;
}
```

Listing 4.6: createExpandedKey()

The function listed above can be used to expand keys of either 128 or 192 bit size. In order to do so, it requires the key as a pointer to an array holding the 16 or 24 bytes, followed by the information about the key length in bytes and the amount of AES rounds that the expanded key will be used in. Initially, the required memory space is allocated to the expanded key and the RCON Value, which is used in one of the underlying transformations and set in the global scope, is set back to its start value. Subsequently, the original key is copied into the *expanded_key*, as the original key is the first to be applied to the plain text. Then the rest of the expanded key is filled, one four-byte word at a time. The next word is generated by the *fillInNextWord()*-function, which essentially takes the previous word, applies some transformations on it every four or six rounds (depending on the key length), and XORs it with the same word in the previous key of the schedule. After all required words have been generated, the function returns a pointer to the array holding the newly expanded key.

4.5 Hexadecimal Conversion

4.5.1 readFromHex()

```c
struct charText * readFromHex(struct charText * hex_string) {
    unsigned long new_text_length = (hex_string->text_length + 1) / 2; // +1 for odd length
    unsigned char * result = calloc(new_text_length, 1);
    unsigned char current_char, hex_value, first_hex_digit, second_hex_digit;

    for(unsigned long i = 0; i < hex_string->text_length; i += 2) {
        current_char = hex_string->text[i];

        if (hex_string->text_length - i == 1 ) { // case last character in odd length
            hex_value = get4bitHexValueFromChar(current_char);
        } else {
            first_hex_digit = get4bitHexValueFromChar(current_char) * 16;
            second_hex_digit = get4bitHexValueFromChar(hex_string->text[i + 1]);
            hex_value = first_hex_digit ^ second_hex_digit;
        }

        result[(i / 2)] = hex_value;
    }
    setNewText(hex_string, result, new_text_length);

    return hex_string;
}
```

Listing 4.7: readFromHex()

The above code is reading hexadecimal digits from strings. As a parameter, it handles a struct of type charText. Firstly, it calculates the new length of the returned struct. Since common ASCII characters have a representation scale of 8 bits, two hexadecimal digits are needed. To handle odd length the calculation also adds one to the result size. Due to integer arithmetics, this has no effect in the case of even lengths. The actual transformation is handled inside of the loop. The first digits ASCII-value is retrieved and shifted four bits to the left before the second ASCII-value is appended. The case of an odd length needs to be handled as it shouldn't be shifted. Once all hexadecimal values have been converted, setNewText() is used to store the result back into the struct. The converted struct gets returned afterward.

4.6 Test Library

```c
int assertExpectedEqualsActual(unsigned char expected, unsigned char actual) {
    if (expected == actual) {
        return 0;
    } else {
        return 1;
    }
}
```

Listing 4.8: assertExpectedEqualsActual()

The above printed functions equals a common *assertEquals()* function. It indicates the order of the required parameters through its name and is made available as a library via CMake.

5 Encryption Modules

The following chapter primarily describes the core functions which are located in AES/encryption/src. This directory contains all methods which are only associated with the encryption implementation.

The dependency of the models for the encryption is shown in figure 5.1. This structure is chosen to provide a good orientation of the modules and to compile coherent methods. Furthermore, this structure provides decent modularity, where single components can be exchanged for further implementations or different usages. The following function documentation will start from the AES structure to its encryption layers (Red in Figure 5.1), then move on to the control structure (Blue in Figure 5.1) and at the end address the user in- and output (Yellow in Figure 5.1). The two purple modules are described above in the general module chapter 4. Nevertheless, to illustrate the whole encryption program, their affiliation is also included. The chapter on encryption will be concluded by an explanation of the developed test cases.

Furthermore, Figure 5.1 gives an overview of all core functions which will be described subsequently. For readability purposes, functions are mostly subdivided into smaller functions. Not all of these will be described in the following to keep the scope focussed, as they should be self-describing.

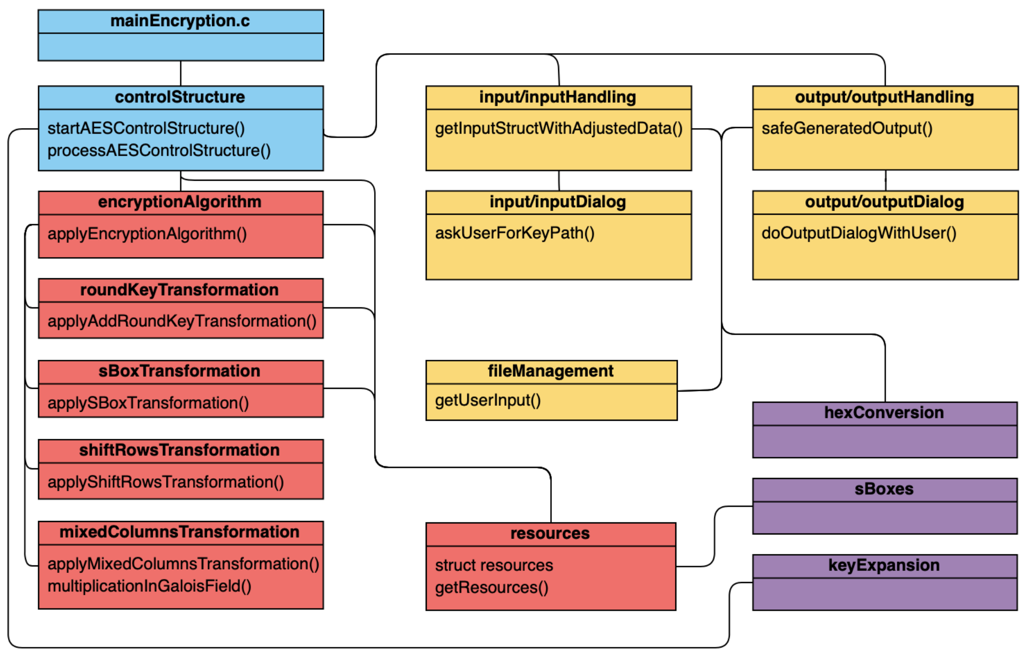
For an overall overview for the user interface figure A.1, in the appendix, is provided.

5.1 AES-Structure

This section introduces the most important part of the program as it handles the encryption itself. Therefore, the here described functions will be presented in a more detailed manner than others not included in this section.

5.1.1 struct resources

```
struct resources {
   unsigned char * sBox;
   unsigned char * message;
   unsigned char * expanded_key;
   unsigned int number_of_rounds;
```



git

Figure 5.1: Structure of directories and functions of the encryption

```
};
```

Listing 5.1: struct resources

The resources struct is one of the main elements of the encryption algorithm. It bundles all major resources of the AES which can be easily passed between functions. SBox, message, and expanded_key are all pointers pointing to arrays of the associated names. number_of_rounds depends on the chosen version of AES. This provides a reusability of the algorithm for all AES versions and will be set in the following function.

5.1.2 getResources()

```
struct resources * getResources(int aes_version) {
    struct resources * resources_pointer;
    resources_pointer = malloc(sizeof(struct resources));

    resources_pointer->message = calloc(MESSAGE_BYTE_SIZE, 1);
    resources_pointer->sBox = getSBox();
    resources_pointer->number_of_rounds = NUMBER_OF_ROUNDS_128BIT_KEY; //By default

    if (aes_version == 192) {
        resources_pointer->number_of_rounds = 12;
    }
    else if (aes_version == 256) {
        resources_pointer->number_of_rounds = 14;
    }

    return resources_pointer;
}
```

Listing 5.2: getResources()

getResources() takes the called AES version and sets the number of rounds accordingly. The default version is 128, other AES versions are only prepared but not fully implemented. Furthermore, the function handles the allocation of memory required to store the struct itself, the message, and sets up a pointer to the initialized sBox. It returns a pointer to the constructed resource bundle.

5.1.3 applyEncryptionAlgorithm()

```
struct resources * applyEncryptionAlgorithm(struct resources * aes_resources) {

    applyAddRoundKeyTransformation(aes_resources, 0);
    for (int round = 1; round <= aes_resources->number_of_rounds; ++round) {
        applySBoxTransformation(aes_resources);
        applyShiftRowsTransformation(aes_resources->message);
        if (round != aes_resources->number_of_rounds) {
            applyMixedColumnsTransformation(aes_resources->message);
        }
        applyAddRoundKeyTransformation(aes_resources, round);
    }

    return aes_resources;
}
```

Listing 5.3: applyEncryptionAlgorithm()

The above-listed function constitutes the core of the AES itself. It gets a struct containing all resources, which is explained above, to handle the encryption. Afterwards, it returns the same struct containing the encrypted message. These resources will already be completely initialized at this point. Due to this, it will handle a 128 bit message and an expanded key of at least 176 bytes, depending on the AES version.

As the documentation [oST01] specifies: The given expanded key will be applied first, followed by a repetitive application of byte substitution, a circular shift of rows, a mixing operation of the columns, and an addition of the corresponding round key. In the last round, the mixing column transformation will not be used. In the following, these functions will be explained in more detail.

5.1.4 applyAddRoundKeyTransformation()

```
void applyAddRoundKeyTransformation(struct resources * r, unsigned char round) {
    for (int byte_Position = 0; byte_Position < ROUND_KEY_BYTE_SIZE; ++byte_Position) {
            r->message[byte_Position] ^= r->expanded_key[byte_Position + (round *
                ROUND_KEY_BYTE_SIZE)];
        }
}
```

Listing 5.4: applyAddRoundKeyLayer()

applyAddRoundKeyTransformation() gets the resources to handle the message and the round key. The first index of the round key can be obtained by multiplying the current round with the ROUND_KEY_BYTE_SIZE. The application of the round key can then be performed by simply iterating over the message bytes and XORing them with the corresponding round key bytes. The result will be stored at the correct position of the given message pointer. As it operates directly on the allocated message there is no need for a return statement. Therefore this method is declared void.

5.1.5 applySBoxTransformation()

```
void applySBoxTransformation(struct resources * r) {
    for (int byte_Position = 0; byte_Position < MESSAGE_BYTE_SIZE; ++byte_Position) {
        r->message[byte_Position] = r->sBox[r->message[byte_Position]];
    }
}
```

Listing 5.5: applySBoxTransformation()

The applySBoxTransformation() also does an operation on every byte, represented by the iteration. It substitutes the message byte with values retrieved from the S-Box lookup-table. The required substitute is obtained by reading the S-Box array at the position equalling the value of the to-be-substituted byte. Afterwards, the message contains only substituted values. As well, this method operates directly on the message.

5.1.6 applyShiftRowTransformation()

```
void applyShiftRowsTransformation(unsigned char * message) {
    int dimension = BYTES_IN_WORD;
    unsigned char origin_positions[dimension], copied_row[dimension];
    unsigned char position_of_new_value;

    for (int row = 1; row < dimension; ++row) {
        //Copies Row
        for (int element_in_row = 0; element_in_row < dimension; ++element_in_row) {
            origin_positions[element_in_row] = row + element_in_row * dimension;
            copied_row[element_in_row] = message[origin_positions[element_in_row]];
        }
        //Shifts Row
        for (int element_in_row = 0; element_in_row < dimension; ++element_in_row) {
            position_of_new_value = (element_in_row + row) % dimension;
            message[origin_positions[element_in_row]] = copied_row[position_of_new_value];
        }
    }
}
```

Listing 5.6: applyShiftRowTransformation()

The applyShiftRowTransformation() function only takes a pointer to the message. It applies circular row shifts by treating the message array as a vertically build matrix and operates on its rows. Therefore, the first loop iterates row-wise, while both underlying loops operate on the columns. As the first row won't be shifted the outer loop starts with index 1.

The first column loop only copies the elements of the row. Therefore it calculates the position of each row element and stores it in origin_positions. The second column loop will store the value at the new shifted position. The new position gets calculated and stored in position_of _new_value to complete the shift afterwards. As both column loops have to finish sequentially, to start at the second row and through the vertical build matrix, two loops are required.

5.1.7 applyMixedColumnsTransformation()

```
void applyMixedColumnsTransformationr(unsigned char * message) {
    unsigned char stored_word[4];
    unsigned char number1, number2;
    const unsigned char multiply_array[] = {
            0x02, 0x03, 0x01, 0x01,
            0x01, 0x02, 0x03, 0x01,
            0x01, 0x01, 0x02, 0x03,
            0x03, 0x01, 0x01, 0x02
    };

    for (int element_of_message = 0; element_of_message < MESSAGE_BYTE_SIZE;
      ++element_of_message) {
        if (element_of_message % MESSAGE_SIZE_IN_WORDS == 0) {
            memcpy(stored_word, message + element_of_message, BYTES_IN_WORD);
        }
        message[element_of_message] = 0;

        for (int row = 0; row < BYTES_IN_WORD; ++row) {
            number1 = stored_word[row];
            number2 = multiply_array[row + (element_of_message % BYTES_IN_WORD) *
                 BYTES_IN_WORD];
            message[element_of_message] ^= multiplicationInGaloisField(number1, number2);
        }
    }
}
```

Listing 5.7: applyMixedColumnsTransformation()

The above function gets only the message and applies the whole operation directly to it. Initially, it stores the multiply_array as a constant for better performance. Subsequently, the MixColumns Transformation is applied word by word. Therefore, whenever the element_of_message index reaches the beginning of a new word, the respective word is copied to stored_word. This is represented by the if clause.

The following actual MixColumns operations consist out of multiplication in the Galois field between two numbers and can be performed using the function described below. The multiplication is applied to the current element in the word and the associated value of the multiply_array. The result will be XORed with the current element in the message and is stored in the same position.

5.1.8 multiplicationInGaloisField()

```c
static unsigned char multiplicationInGaloisField(unsigned char number1,
  unsigned char number2) {
    unsigned char result;

    if (number2 == 1) {
        result = number1;
    }
    else {
        result = (number1 << (number2 / 2)) ^ (number1 & 0x80 ? 0x1B : 0);
        if (number2 % 2 == 1) {
            result ^= number1; //To perform an odd multiplication
        }
    }
    return result;
}
```

Listing 5.8: multiplicationInGaloisField()

The above function handles the needed multiplication in the Rijndaels Galois field. That means the multiplication of any number in the Galois field with either 1, 2, or 3. Higher numbers might lead to a wrong result but are not needed in this context. Therefore, there are three cases:

Second number is 1 As 1 is the neutral element in the Galois field, the result is the same as number1.

Second number is even This results in a shift of number1 by half of number2. Which is on a binary level the same as the multiplication between those two numbers. As Galois fields are finite, the case of ending beyond the maximum value has to be escaped by the modulo operation. The use of 0x1B instead of 0x11B is chosen because unsigned chars can only handle 8 bits and therefore would drop the first bit if it exceeds through the multiplication.

Second number is odd After applying an even operation, a single XOR operation of number1 is applied to perform the uneven multiplication.

5.2 Control Structure

The "ControlStructure" includes all superior functions for splitting, padding, calling of the AES-Encryption algorithm itself, and calling all user interactions. This is the core unit of the program which manages all subsystems.

5.2.1 startAESControlStructure()

```
int startAESControlStructure(int aes_version) {
   setDecisionForHex();

   struct resources * aes_resources = getResources(aes_version);
   struct charText * text_to_encrypt = getPlainTextFromInput();
   struct charText * key_input = getKeyAccordingToVersionFromInput(aes_version);

   text_to_encrypt = processAESControlStructure(aes_resources, text_to_encrypt, key_input);

   safeGeneratedOutput(text_to_encrypt);
   freeAllAllocatedMemory(aes_resources, text_to_encrypt, key_input);

   return 0;
}
```

Listing 5.9: startAESControlStructure()

startAESControlStructure() provides all the needed data to run a correct encryption. To enable a hexadecimal representation of the in- and output, the user gets asked for a decision to enable or disable the corresponding representation. Afterwards, the needed structs will be created. The key and text to encrypt will be imported by a user-specified path (which will not be completely described in the following). The next step is to process the AES-algorithm on single 128 bit blocks, for which the below-described function is used. In the end, this function saves the encrypted text and frees all allocated resources. It returns a 0 to the main function to signal everything ran correctly.

5.2.2 processAESControlStructure()

```
struct charText * processAESControlStructure(struct resources * aes_resources,
  struct charText * text_to_encrypt, struct charText * key_input) {
   aes_resources->expanded_key=getExpanded128bitKey(key_input->text);

   for (int text_block = 0; text_block < text_to_encrypt->text_length; text_block +=
        MESSAGE_BYTE_SIZE) {
      copyNextTextBlockToMessage(aes_resources, text_to_encrypt, text_block);
      applyEncryptionAlgorithm(aes_resources);

      copyEncryptedMessageBlockToText(text_to_encrypt, aes_resources, text_block);
      printProgress(text_to_encrypt->text_length, text_block);
   }

   return text_to_encrypt;
}
```

Listing 5.10: processAESControlStructure()

The function printed above takes the earlier initialized struct, initializes the expanded key, and performs the encryption process on those. To do so, this function copies blocks of 128 bit to the message stored in resources. After the encryption of the current block is finished, the encrypted message will be stored back to the text. This iteration is repeated until the whole text is encrypted. For performance enhancement, multithreading could be applied at this part for later implementations.

5.3 Input and Output Handling

The following section will not explain all of the modules core functions, as its main purpose is user contact and data provision. It is designed to handle a simple user dialog with the smallest amount of interaction. Therefore, small scope of error handling has to be provided and the usage gets simplified. To even increase the simplicity for users, decent colourings are included in the dialog.

As initialising the key and the plain text are rather similar it resigned to describe resembling methods.

5.3.1 getPlainTextFromInput()

```
struct charText * getPlainTextFromInput() {
    askUserForPlainTextPath(getDefaultPath());
    char * file_path = getInputPath();
    struct charText * plain_text = getCharTextStructWithInput(file_path);

    plain_text = expandPlainTextToAesDivisor(plain_text, MESSAGE_BYTE_SIZE);
    plain_text = adjustCharTextToHexDecision(plain_text);

    return plain_text;
}
```

Listing 5.11: getPlainTextFromInputCode()

This function calls the user dialog, to get the path of the plain text that should get encoded. It is possible to import files of any extension given by the user (A limitation is explained in 5.3.3). After reading the file, that is provided by the user, the struct with the given text is created. To handle any file length and to perform the AES on 128 bit, the plain text needs to get expanded to a divisor of the corresponding amount of bits. This padding is processed by allocating memory to zero through the calloc() function provided by the <stdlib.h> library. Therefore, the decryption can be independent and no additional functionality, to remove the padding, is needed. It is considered to be secure, as the AES is resistant against known-plaintext attacks [DR02]. For performing on hexadecimal the given input has to get transformed and the length accordingly, which is executed by adjustCharTextToHexDecision(). Finally, the struct has the right size and its adjusted length, which gets returned.

5.3.2 askUserForKeyPath()

```
void askUserForKeyPath(char * default_path, int length_of_key) {
   printf("Please enter the filename containing the %d Byte long key you want to use to
        encrypt from the %s directory:\n", length_of_key, default_path);
}
```

Listing 5.12: askUserForKeyPath()

askUserForKeyPath() is only exemplary for the input dialogue functions, which typically only gets a few parameters to include in the print statement. This should give a better understanding to the user what he needs to provide and where.

5.3.3 safeGeneratedOutput()

```
void safeGeneratedOutput(struct charText * encrypted_text) {
   char * output_path = getOutputPath();

   if (getHexMode()) {
      encrypted_text = convertToHexString(encrypted_text);
   }
   saveEncryptedMessage(encrypted_text, output_path);
   doOutputDialogWithUser(output_path);
   getUserInput();
}

}
```

Listing 5.13: safeGeneratedOutput()

safeGeneratedOutput() stores the encrypted text and transforms it back to a hexadecimal representation regarding the decision the user made at the beginning. The saved file will be stored in the default folder /Files and be called encryptedMessage.txt. Thereby, files of different file extension get converted to an encrypted .txt file. Hence, it gets the encrypted text and does the following output dialogue. To exit on command, the latter step is performed.

5.3.4 doOutputDialogWithUser()

```
void doOutputDialogWithUser(char * output_path) {
    printf("\rProgress: 100 %%.\n");
    fflush(stdout);
    printf("The message is successfully encrypted and will be stored in %s. \n", output_path);
}
```

Listing 5.14: doOutputDialogWithUser()

The above function firstly flushes the last print, as the before executed progress dialogue might, for internal reasons, not finish with an output of exactly 100%. Therefore this function overwrites the entry by a 100% message, tells the user about the success, and where the encrypted text will be stored. As it only prints on the console, there is no need for a return type.

5.3.5 getUserInput()

```
char * getUserInput() {
    char user_input[MAX_USER_INPUT_LENGTH];
    char * return_user_input = malloc(MAX_USER_INPUT_LENGTH);

    fgets(user_input, MAX_USER_INPUT_LENGTH, stdin);
    sscanf(user_input, "%s", return_user_input);
    fflush(stdin);

    return return_user_input;
}
```

Listing 5.15: getUserInput()

getUserInput() is one of the core file-management functions, as it is used for the decision of the hexadecimal mode, the message path, and the key path.

It explains briefly how the input is read. fgets() reads the input provided by the user through the console. The input length is limited to 32 characters by MAX_USER_INPUT _LENGTH which prevents a buffer overflow. Afterwards, sscanf() reads the string and stores it in the variable user_input. Different implementations resulted in non-deterministic errors depending on the used compiler. After reading and storing the input, the actual terminal input will be flushed to avoid later problems. Finally, the function returns the acquired input of the user.

5.4 Tests

The test module of the encryption uses the self-made small library from section 4.6. The test cases are hard coded and based on the examples given in the documentation of the AES [oST01]. It has been decided to not implement all test cases in all modules, as the program as a whole provides escaping functionality when confronted with wrong inputs. For example, the AES-Encryption-Algorithm can't get a message input that is not a divisor of 128 bit, because the input handling module expands every input which is smaller than that. Thus, it has been considered that the documentation example is enough to prove the correctness of the program. Additionally, through the rounds of AES128, every module will be tested at least ten times with different input when the whole algorithm is tested (even though the additional insights are limited as all are part of the same equivalence class).

The control structures test cases handle to check whether the program runs correctly on empty texts or texts of odd length. It is decided not to include test cases on different file inputs, as they would be very extensive to integrate while guaranteeing those cases to be highly portable. Nevertheless, the test cases were run manually. The test cases included reading empty files, hexadecimal files, huge blind texts, huge hexadecimal texts, and surely all the documentation examples.

The following will be one definitive test to give a better understanding.

5.4.1 ControlStructureEncryptionTest

```
[...]
int testLargerInputAs128() {
    int result = 0;
    int message_size = 32;

    unsigned char expected[] = {0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30, 0xd8, 0xcd, 0
        xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a, 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30, 0
        xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a};
    unsigned char temp_message[] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99
        , 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
         0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff };
    unsigned char temp_key[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0
        x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

    struct resources * aes_resources = getResources(128);
    struct charText * message_test = initializeStructWithArray(temp_message, message_size);
    struct charText * key_test = initializeStructWithArray(temp_key, 16);

    message_test = processAESControlStructure(aes_resources, message_test, key_test);

    for (int i = 0; i < message_size; ++i) {
        result += assertExpectedEqualsActual(expected[i], message_test->text[i]);
    }

   freeAllAllocatedMemory(aes_resources, message_test, key_test);

    return result;
}

int main () {
    int result = 0;

    result += testLargerInputAs128();
    result+= testEmptyInput();
    result+= testOddInput();

    return result;
}
```

Listing 5.16: ControlStructureEncryptionTest()

This exemplary test above shows the main structure of the written tests. The first entry to the test is the main method, which calls all the test cases and accumulates the results.

testLargerInputAs128() tests if the splitting of the text runs correctly in the processAESControlStructure() function. Therefore, it creates three arrays:

expected: Handles the result which the algorithm should produce.

temp_message: Contains the message that should be split and encrypted.

temp_key: Is the key that is used for encryption.

Next, the structs need to get initialised. Therefore, the above-explained getResources() is called and the input arrays are stored in charText struct. After the call of processAESControlStructure(), the result has to be compared with the expected result. Therefore, the method of the test library is called. As the method returns 1 in case the test fails, the number of failed tests can be counted by simply adding the test outcome to the result variable. Once all actual and expected values are compared, the result can be returned.

6 Decryption Modules

In the following chapter, the structure and functionality of the AES decrpytion implementation are described. The general underlying algorithm has been modelled previously in the project definition phase and is illustrated in annex A.2. All depicted modules are exclusive to the decryption part of the project and can be found in the AES/decryption/src directory. In order to provide a clear overview and a good understanding of how the decryption program has been designed, this chapter will only focus on the core concepts and selected functions of each module.

The interdependencies between all decryption exclusive modules are illustrated in figure 6.1. As the colouring indicates, the six independent modules can be split into three different categories. The major two groups are composed by those modules coloured in red, which entail the actual implementation of the decryption algorithm, and those coloured in blue, which contain functionality required for the application of the decryption, like for example implementation-specific routines, shared functionality, and user interfaces. The third category of modules encapsulates general functionality, that is required by, but unspecific to the project. This category is depicted in yellow and its only module offers methods for file management.

In order to give an overview of how the modules are linked and work together to enable AES decryption for the user, each module will be introduced in the following. First, the algorithm-specific modules will be introduced in sections 6.1 and 6.2 before the overhead functionality is described in sections 6.3 to 6.5. The chapter is concluded by the description of the file management module in section 6.6.

Within the decryption modules, an additional naming convention has been used. This was necessary, since all code within this module enforces the recommendations given in [MC19] to use return statements as often as possible. This is to make the code easier to understand, by making more obvious what the function does and returns. Since in languages like C, this is not without issues as it may lead to dangling pointers or allocated memory not being accessible any longer after a function call, it was introduced that all functions named "apply" return a pointer to the same address as has been put in. This means that calling a function like *array = applyTransformation(array);* is safe in terms of the above mentioned issues.

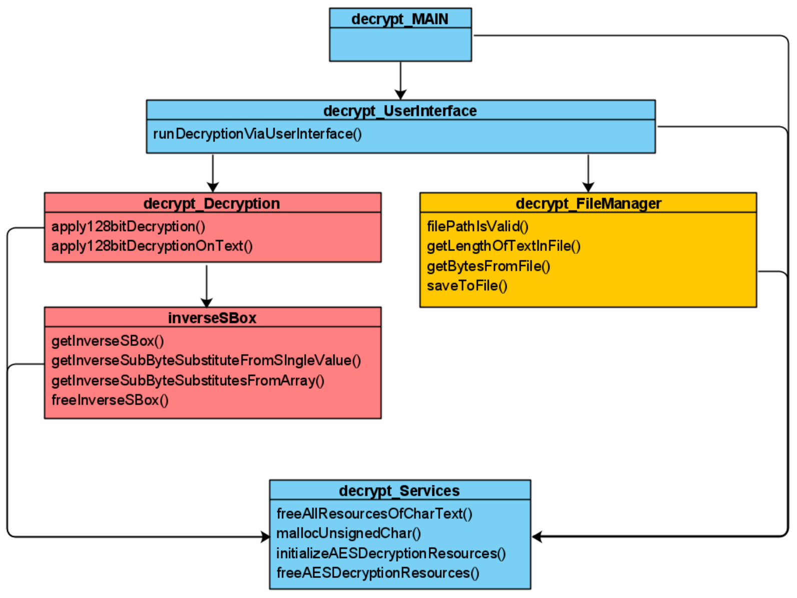


Figure 6.1: Interdependencies of Decryption Modules

6.1 decrypt_Decryption.h

The module decrypt_Decryption.h contains the core AES decryption algorithm and provides the respective decryption functionality. In order to allow this, the module in turn depends on decrypt_Services.h (see section 6.5) and inverseSbox.h (section 6.2).

6.1.1 apply128bitDecryptionOnText()

```c
struct charText * apply128bitDecryptionOnText(
  struct charText * message_struct, unsigned char * key) {
    unsigned char * expanded_key = getExpanded128bitKey(key);
    unsigned char * temp_block = mallocUnsignedChar(MESSAGE_BYTE_SIZE);

    for (unsigned long current_message_block = 0;
      current_message_block < message_struct->text_length;
      current_message_block+=MESSAGE_BYTE_SIZE) {
        copyTextBlockFromMessage(message_struct, temp_block, current_message_block);
        temp_block = apply128bitDecryption(temp_block, expanded_key);

        copyDecryptedBlockToMessage(temp_block, message_struct, current_message_block);
        printProgress(message_struct->text_length, current_message_block);
    }

    free(temp_block);
    free(expanded_key);
    return message_struct;
}
```

Listing 6.1: apply128bitDecryptionOnText()

The above-displayed function offers AES decryption for written texts. The first required parameter *message_struct*, is used to pass the ciphered text in the form of a *charText* struct. The ciphered text can theoretically be of any size, as long as the length is storable in a variable of type long and a multiple of the *message_byte_size*, which is equal to 16. The decryption key gets passed via the second parameter, which expects a pointer to an array of size 16, storing the key byte by byte.

After the entered key has been expanded, the function decrypts the text block by block, extracting a copy of each block and passing it, together with the expanded key, on to the appropriate function described below. After decryption, all blocks are stored back into the original *message_struct*, overwriting the original ciphered text. In line with the naming conventions, the function returns a pointer to the original input *message_struct* upon completion.

6.1.2 apply128bitDecryption()

```
unsigned char * apply128bitDecryption(
unsigned char * ciphered_message, unsigned char * expanded_key) {
    return applyDecryption(ciphered_message, expanded_key, NUMBER_OF_ROUNDS_128BIT_KEY);
}
```

Listing 6.2: apply128bitDecryption()

The function printed above can be used to perform 128 bit AES decryption on a single text block. The text block is entered via the parameter *ciphered_message*, which expects a pointer to an array of length 16. The other required input expects a pointer to an array of length 176, sequentially storing the 44 required key-words of size four each. To this information, the function adds the number of rounds a decryption with a 128 bit Key takes, and passes them all together on to the generic decryption function described below. This contribution of the additional parameter is the main reason the function is provided, as it can be seen as an interface enabling AES decryption to users without requiring them to choose the correct parameters themselves. As the function name implies, a pointer to the input *ciphered_message*, which now stores the decrypted message, is returned.

6.1.3 applyDecryption()

```
static unsigned char * applyDecryption(
  unsigned char * message, unsigned char * expanded_key, const int NUMBER_OF_ROUNDS) {
    int current_round = NUMBER_OF_ROUNDS;
    message = applyAddRoundKeyTransformation(message, expanded_key, current_round);
    for (--current_round; current_round >= 0 ; --current_round) {
        message = applyInverseSubByteTransformation(message);
        message = applyInverseShiftRowsTransformation(message);
        message = applyAddRoundKeyTransformation(message, expanded_key, current_round);
        if (current_round > 0 ) {
            message = applyInverseMixColumnsTransformation(message);
        }
    }

    freeInverseMixColumnsMultiplicationMatrix();
    return message;
}
```

Listing 6.3: applyDecryption()

The static function applyDecryption is used to decrypt a single block of ciphered text, using either 128, 192 or 256 bit keys. In order to do so, it requires the 16 byte array of encrypted text to be put in as pointer via the parameter *message*. The respective cryptographic key has to be provided in his expanded form, where the actual length of the input depends on the chosen key-size (see section 4.4). Another parameter that depends on the length of the chosen key, is the number of decryption cycles the algorithm carries out[oST01]. This information can be entered through the parameter *number_ of _rounds*.

Based on these three inputs the algorithm starts to decrypt the message by applying four different transformations repetitively in the defined order and as often as the *number_ of _rounds* parameter specified. The only deviations are that the *applyRoundKeyTransformation()* is performed once more than the others and that the *applyInverseMixColumnsTransformation()* is not executed in the last cycle. Upon completion, the multiplication matrix used by the applyInverseMixColumnsTransformation is freed, as it is initialised on the first function call and then reused for reasons of efficiency (see section 6.1.7). Finally, the pointer to the, now deciphered, message is returned.

6.1.4 applyAddRoundKeyTransformation()

```
static unsigned char * applyAddRoundKeyTransformation(unsigned char * message,
const unsigned char * expanded_key, const int CURRENT_ROUND) {
    unsigned char * round_key = mallocUnsignedChar(ROUND_KEY_BYTE_SIZE);
    int position_of_round_key = CURRENT_ROUND * ROUND_KEY_BYTE_SIZE;

    memcpy(round_key, expanded_key + position_of_round_key, ROUND_KEY_BYTE_SIZE);
    message = applyRoundKeyOnMessage(message, round_key);

    free(round_key);
    return message;
}
```

Listing 6.4: applyAddRoundKeyTransformation()

The first of the decryption transformations apply the expanded cryptographic key to the message the same way as previously performed by the encryption (see section 5.1.4), except in reverse order. The function accepts a pointer to the ciphered message and to the expanded key, as well as an additional information about the current round of the overhead decryption algorithm. Based on this parameter *current_round*, the function is able to copy the currently required 16 byte round-key out of the *expanded_key*. Subsequently, the round key is applied to the message by calling *applyRoundKeyOnMessage()*, which basically bit-wise XORs all bytes of the round key with the corresponding bytes of the message. Once finished, the temporary variable *round_key* is freed and a pointer to the now transformed message is returned.

6.1.5 applyInverseSubByteTransformation()

```
static unsigned char * applyInverseSubByteTransformation(unsigned char * message) {
   return getInverseSubByteSubstitutesFromArray(message, MESSAGE_BYTE_SIZE);
}
```

Listing 6.5: applyInverseSubByteTransformation()

The above printed function applies the inverse SubBytes-Transformation on the message. This refers to mapping each byte to a substitute within Rijndaels finite field. In order to perform this, a lookup table, the so-called inverse sbox, can be used. The inverse s-box is provided by the inverseSBox.h module which also offers the here called function. Given the array length, which equals *message_byte_size* in this case, the called function applies the substitution on the message. In line with the naming conventions, a pointer to the transformed input array is returned.

6.1.6 applyInverseShiftRowsTransformation()

```
static unsigned char * applyInverseShiftRowsTransformation(unsigned char * message) {
   unsigned char * result = mallocUnsignedChar(MESSAGE_BYTE_SIZE);
   int old_position_of_byte, new_position_of_byte;

   for (int row = 0; row < 4; ++row) {
      for (int byte_in_row = 0; byte_in_row < 4; ++byte_in_row) {
         old_position_of_byte = (byte_in_row * 4) + row;
         new_position_of_byte = (old_position_of_byte + (4 * row)) % 16;
         result[new_position_of_byte] = message[old_position_of_byte];
      }
   }

   free(message);
   message = result;
   return message;
}
```

Listing 6.6: applyInverseShiftRowsTransformation()

The inverse ShiftRows-Transformation assumes the 16 bytes of the ciphered message as a four by four matrix in which it performs circular right shifts on each row. The number of positions shifted equals the index of the row, from 0 to 3. In order to iterate through this matrix the displayed two for-loops are required, one iterating the rows and one the four bytes in each row. First the position of the current byte in the sequential message array is calculated and stored in *old_ position_ of_ byte*. Based on this information, the *new_ position_ of_ byte* can be calculated and the corresponding byte be copied to its correct position in the result array. As the function returns the same pointer that was put in, the original message is freed and reassigned the transformed contents stored in *result*.

6.1.7 applyInverseMixColumnsTransformation()

```
static unsigned char * applyInverseMixColumnsTransformation(unsigned char * message) {
   unsigned char * result = mallocUnsignedChar(MESSAGE_BYTE_SIZE);

   for (int result_column = 0; result_column < 4; ++result_column) {
      for (int result_row = 0; result_row < 4; ++result_row) {
         result[result_row + result_column * 4] =
            getResultElementOfMixColumnsMatrixMultiplication(
              message, result_column, result_row);
      }
   }

   free(message);
   message = result;
   return message;
}
```

Listing 6.7: applyInverseMixColumnsTransformation()

Like the inverse ShiftRows-Transformation, the inverse MixColumns-Transformation assumes the message as a four by four matrix as well, the major difference lies in the column- instead of the row-wise transformation operation, it applies. The applied transformation treats each column as a four term polynomial and multiplies each of these within Rijndaels finite field, using a fixed second polynomial. This operation can also be written as matrix multiplication, which is exploited in the underlying logic of the function *getResultElementOfMixColumnsMatrixMultiplication()*. This function can be used to obtain the matrix multiplication result for any single element in the message, only requiring the specification of its position by entering its column and row. After this has been performed on every value, the result is stored back in to the original message pointer which is returned subsequently. In order to deliver the results efficiently, the required multiplication matrix is initialised in a static global variable on the first function call only and reused in the subsequent ones. Once all MixColumns-Transformations have been applied, it can be freed by calling the provided function.

6.1.8 Testing decrypt_Decryption.h

In order to facilitate constant and easy testing throughout the development and code-enhancement process aiming at high portability to different systems and settings, a number of tests were written for this module. Their source code is located in AES/decryption/tests and compiles into independent CTest executables. All tests rely on the TestLibrary.c (Section 4.6), providing the functionality for actual and expected value comparison.

All in all, five independently executable tests were developed, four for the decryption transformations described in sections 6.1.4 to 6.1.7 and one to test the two functions described in sections 6.1.1 and 6.1.2, which offer complete decryption algorithm functionality. All testing relies on hard-coded test messages and expected results and does only cover cases of correct inputs. The testing of incorrect inputs has been performed by users via the interface described below.

6.2 inverseSBox.h

The module described in the following contains functionality to initialise and apply the inverse sbox, a substitution table which maps every byte to a substitute in Rijndaels finite field. In order to work properly it requires access to the decryption module decrpyt_Services.h, which is described in section 6.5.

6.2.1 getInverseSBox()

```
unsigned char * getInverseSBox() {
    unsigned char * inverse_sbox = mallocUnsignedChar(INVERSE_SBOX_SIZE);
    for (int initial_value=0; initial_value < INVERSE_SBOX_SIZE; ++initial_value) {
        inverse_sbox[initial_value] = initial_value;
    }

    inverse_sbox = applyInverseAffineTransformation(inverse_sbox);
    inverse_sbox = applyMappingToMultiplicativeInverse(inverse_sbox);

    /* 0x00 has no inverse, set to 0x52 according to specification */
    inverse_sbox[0] = 0x52;

    return inverse_sbox;
}
```

Listing 6.8: getInverseSBox()

The function listed above creates and fills an array containing the inverse sbox. First, the required 256 bytes of memory, equalling the number of values a byte can assume, are allocated and the newly created array initially filled with all numbers from 0 to 255. Subsequently, two transformations are applied to these initial values to obtain the required substitutes. At first, the values undergo the inverse of the affine transformation used to build the non-inversed sbox (see section 4.3.2). The transformed values are then mapped to their multiplicative inverses in Rijndaels finite field. In order to achieve this, the called function uses a lookup table containing inverses for all values, which was previously filled using a generator iterating through all possible elements in the respective Galois field. As zero is the only value not having an inverse, the substitute is set to a pre-specified value before the pointer to the fully initialised inverse sbox is returned.

6.2.2 getInverseSubByteSubstitutesFromArray()

```c
unsigned char * getInverseSubByteSubstitutesFromArray(
  unsigned char * input, int input_array_length) {
    if (!inverse_sbox_is_initialised) initializeInverseSBox();
    for (int i = 0; i < input_array_length; ++i) {
        input[i] = inv_sbox[input[i]];
    }
    return input;
}
```

Listing 6.9: getInverseSubByteSubstitutesFromArray()

The above-printed function can be used to obtain inverse S-Box substitutes for an array of values. In order to obtain the substitutes without initialising the complete S-Box on every function call, the function relies on copy of the inverse S-Box available via the *inv_sbox* pointer, set in the global scope. It first checks if the S-Box has been initialised before and, if that is not the case, calls the available static function that does so. The local copy can be freed by calling the respective provided function. Afterwards, all values of the array which was put in, along with its length via the corresponding parameters, are substituted. The substitutes can be obtained by reading from the S-Box at the position which equals the to-be-substituted byte. In the end, the function returns the pointer to the transformed array back.

6.3 decrypt_MAIN.c

The main module of the decryption holds the *main()*-function of the decryption executable. This function essentially serves two purposes. The first is to call the routines documented in section 6.5, which initialise and, ultimately, free all resources that are required for the AES decryption and independent of any module used to carry it out. The second is to call the wished type of interface or control structure to perform the decryption. Currently the only available option to do so is using the module decrypt_UserInterface.h, which is documented in section 6.4.

6.4 decrypt_UserInterface.h

```c
void runDecryptionViaUserInterface() {
    printf("128 bit AES Decryption enabled.\n");
    struct charText * message = getConformedMessageFromUser();
    struct charText * key = getConformed128bitKeyFromUser();

    printf("\nBeginning AES Decryption...\n");
    message = apply128bitDecryptionOnText(message, key->text);

    printDecryptionComplete();
    saveDecryptionResultToDefaultFile(message);

    freeAllResourcesOfCharText(message);
    freeAllResourcesOfCharText(key);
    startExitDialogue();
}
```

Listing 6.10: runDecryptionViaUserInterface()

The module decrypt_UserInterface.h can be used to deploy the AES decryption offered by the decrypt_Decryption.h module (see section 6.1) via a user dialogue interface. As a major control structure, it relies on all of the other decryption modules, except the inverseSBox.h. The above printed function is offered to commence the dialogue based decryption. Additional to its following depiction, an activity diagram, illustrating the fundamental process of the module, is provided in annex A.3.

After informing the user about the selected decryption mode (so far only 128 bit decryption is available) the program, through the respective method, retrieves the message from a file specified by the user and conforms it to the required format. This includes, given a valid input by the user, potentially converting the message from hexadecimal representation to a conventional string and, in case its length isn't equal to a multiple of the required message block size and the user approves, cutting it to the correct length. After the message is obtained, the key is acquired from the user as well. The process of importing and conforming is mostly equal to the message retrieval described before, except too small keys are not accepted, their input results in an error and termination of the program. If both message and key are successfully received, the message can be decrypted using the function described above (see section 6.1.1) and subsequently be stored back into the default output file. Depending on another user decision, the decrypted key will either be stored as ASCII-characters or in hexadecimal representation. After freeing the message and key using the respective functions offered by the decrypt_Services module, the dialogue can be closed and the program is finished.

6.5 decrypt_Services.h

```
void initializeAESDecryptionResources() {
    printf("Initialization of AES-Decryption complete.\n");
}

void freeAESDecryptionResources() {
    freeSBox();
    freeInverseSBox();

    printf("\nFreeing of AES-Decryption Resources finished.\n");
}
```

Listing 6.11: Routines concerning AES Decrytpion Resources

The *decrypt_Services.h* module offers service functions, which are specific to the context of the decryption implementation. It entails a function that can be used to free a struct of type charText and all its resources, as well as a function for allocation of memory to pointers of type unsigned char. The latter checks if the memory allocation was successful, which is especially useful when large sets of data like the ciphered text need to be stored, and terminates the program in case a failure occurs.

Additional to the mentioned functions, the module also offers the above printed routines for the initialization and orderly termination of the AES decryption. These are called by the main method as the first and last line of the executable and are meant to deal with resources that need to be initialised or freed, no matter which module is used as interface for the actual deployment of the decryption (at the moment only the dialogue-based module described in section 6.4 is available for this). Currently, only the internal copies of the S-Box and its inverse require freeing after running the decryption and nothing is initialised on start-up. The decision to keep the empty initialisation routine nevertheless, is based on the idea to keep the structure up in the *main()*-Method, to facilitate an easy understanding of where to add functionality that might arise, when additional modules are added in the future.

6.6 decrypt_FileManager.h

The *decrypt_FileManager.h*-module offers the basic file-functionality required by the userinterface module described above (see section 6.4). The two functions *getBytes-FromFile()* and *saveToFile()* constitute the major functionality offered here. Whilst one can be used to read text from a file, given a valid file path and the length of the text-to-read, the other can be used to store given text to a specified file path. In order to enable successful usage of these, two additional helper functions are provided. The first, called *filepathIsValid()*, can be used to check if a entered filepath exists whilst the second, called *getLengthOfTextInFile()*, returns the length of text contained in a file at a valid filepath.

A Annex of Figures

A.1 General AES Encryption Algorithm

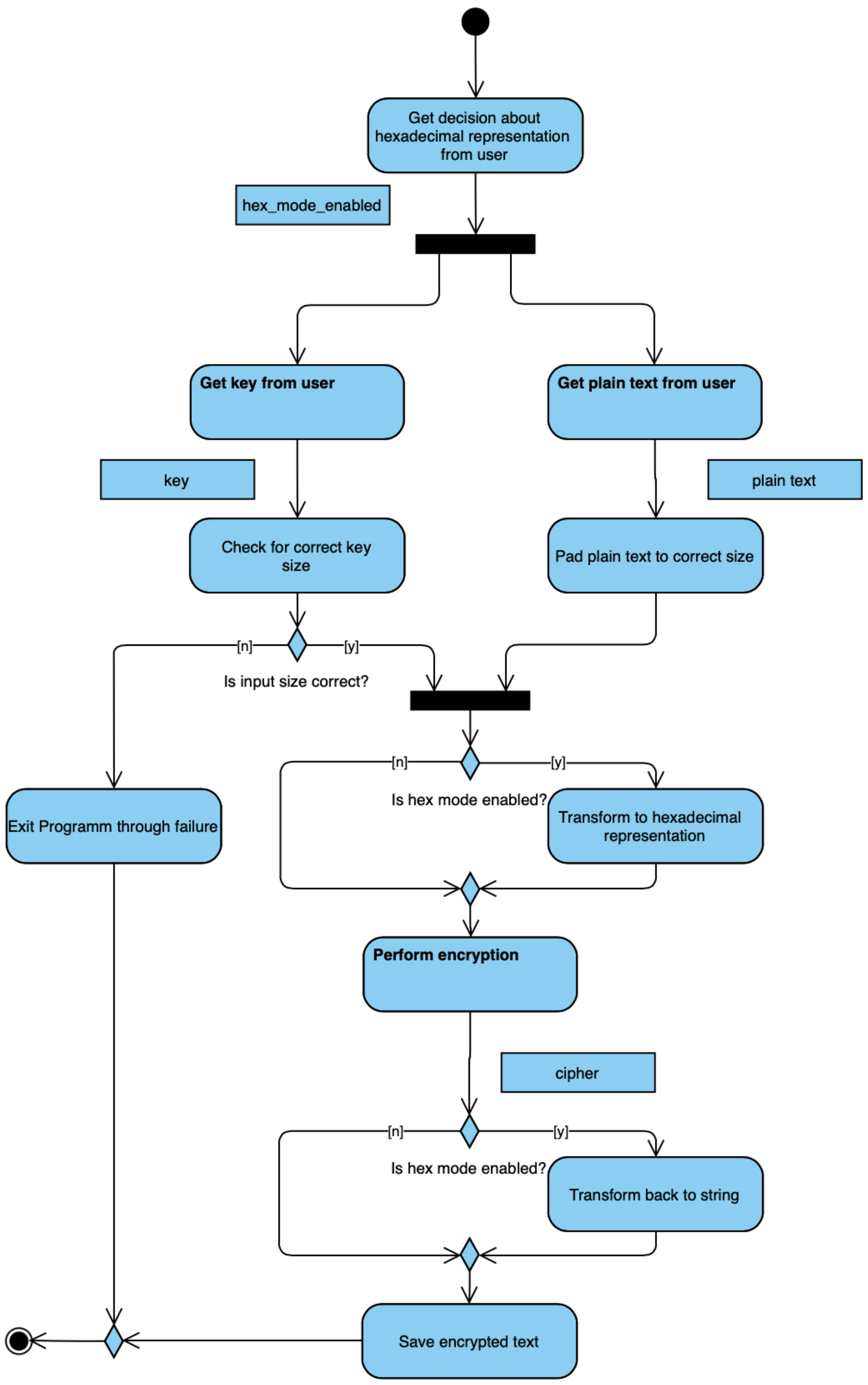


Figure A.1: AES Encryption Algorithm

A.2 General AES Decryption Algorithm

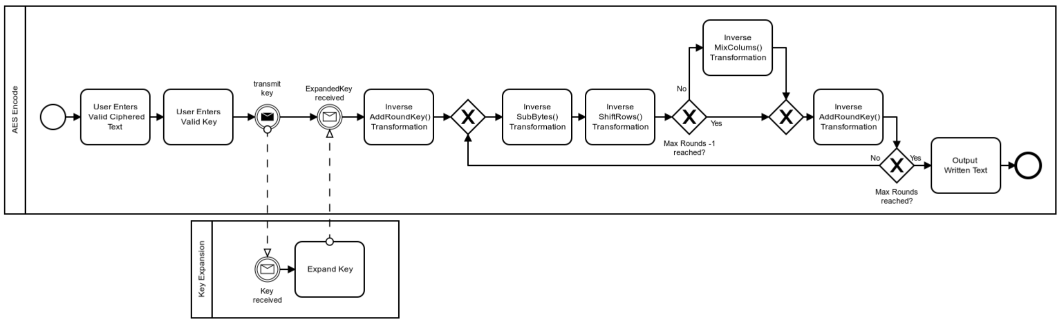

AES Encode
User Enters Valid Ciphered Text
User Enters Valid Key
transmit key
ExpandedKey received
Inverse AddRoundKey() Transformation
Inverse SubBytes() Transformation
Inverse ShiftRows() Transformation
No
Yes
Max Rounds -1 reached?
Inverse MixColums() Transformation
Inverse AddRoundKey() Transformation
No
Yes
Max Rounds reached?
Output Written Text
Key Expansion
Key received
Expand Key

Figure A.2: AES Decryption Algorithm

A.3 Decryption UI Activity Diagram

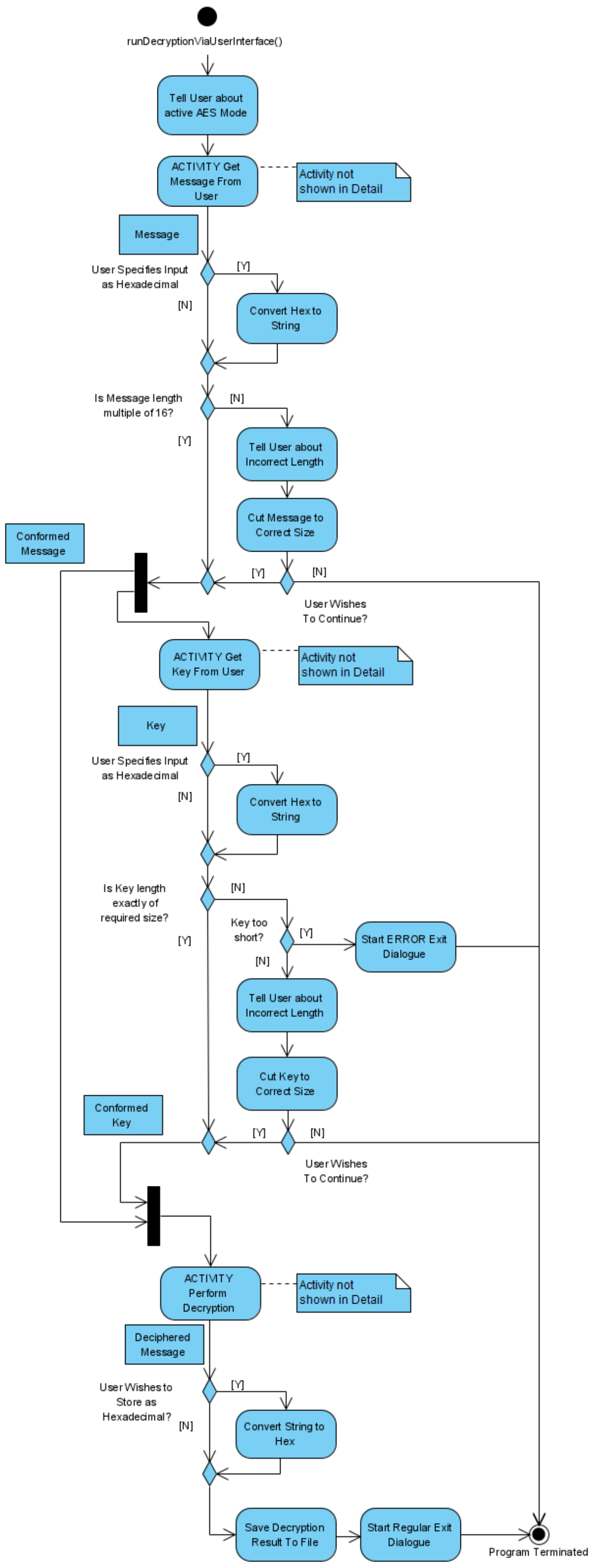

runDecryptionViaUserInterface()
Tell User about active AES Mode
ACTIVITY Get Message From User
Activity not shown in Detail
Message
User Specifies Input as Hexadecimal
[Y]
[N]
Convert Hex to String
Is Message length multiple of 16?
[N]
[Y]
Tell User about Incorrect Length
Cut Message to Correct Size
Conformed Message
[Y]
[N]
User Wishes To Continue?
ACTIVITY Get Key From User
Activity not shown in Detail
Key
User Specifies Input as Hexadecimal
[Y]
[N]
Convert Hex to String
Is Key length exactly of required size?
[N]
[Y]
Key too short?
[Y]
[N]
Start ERROR Exit Dialogue
Tell User about Incorrect Length
Cut Key to Correct Size
Conformed Key
[Y]
[N]
User Wishes To Continue?
ACTIVITY Perform Decryption
Activity not shown in Detail
Deciphered Message
User Wishes to Store as Hexadecimal?
[Y]
[N]
Convert String to Hex
Save Decryption Result To File
Start Regular Exit Dialogue
Program Terminated

Figure A.3: runDecryptionViaUserInterface() Activity Diagram

List of Figures

List of Listings

Bibliography

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography).* Springer, 1 edition, 2002.

[emb06] embedded. Guidelines for writing efficient c/c++ code, 2006. `https://www.embedded.com/guidelines-for-writing-efficient-c-c-code/`, as of February 5, 2021.

[Gee19] Geeks For Geeks. Why learning c programming is a must?, 2019. `https://www.geeksforgeeks.org/why-learning-c-programming-is-a-must/`, as of February 5, 2021.

[MC19] Robert C. Martin and James O. Coplien. *Clean code: a handbook of agile software craftsmanship.* Prentice Hall, Upper Saddle River, NJ [etc.], 2019.

[Mun20] Daniel Munoz. After all these years, the world is still powered by c programming, 2020. `https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming`, as of February 5, 2021.

[oST01] Information Technology Laboratory (National Institute of Standards and Technology). Announcing the advanced encryption standard (aes). In *Federal information processing standards publication ; 197.*, 2001.

Declaration

I hereby declare that, to the best of my knowledge and belief, this Seminarthesis titled "Implementing the AES: Documentation" is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 5.2.2021

Date

[illegible]

Signature

Hamm, 05.02.2021

[illegible]

Consent Form

for the use of plagiarism detection software to check my thesis

Full Name: Benedikt Kluss, Hendrik Hagedorn
Student Number: 464709, 417924
Course of Study: Information Systems
Title of Thesis: Implementing the AES: Documentation

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 5.2.2021	[illegible]
Date	Signature

Hamm, 05.02.2021 [illegible]