БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет ИНО

Специальность ИиТП

Практическая работа № 2

по дисциплине «Методы защиты информации»

Выполнил студент: Дегтярев А.А.

группа 393551

Зачетная книжка № 902021-26

Минск 2018

**ИПР №2**

#### Часть 2.1 Блочные шифры. AES

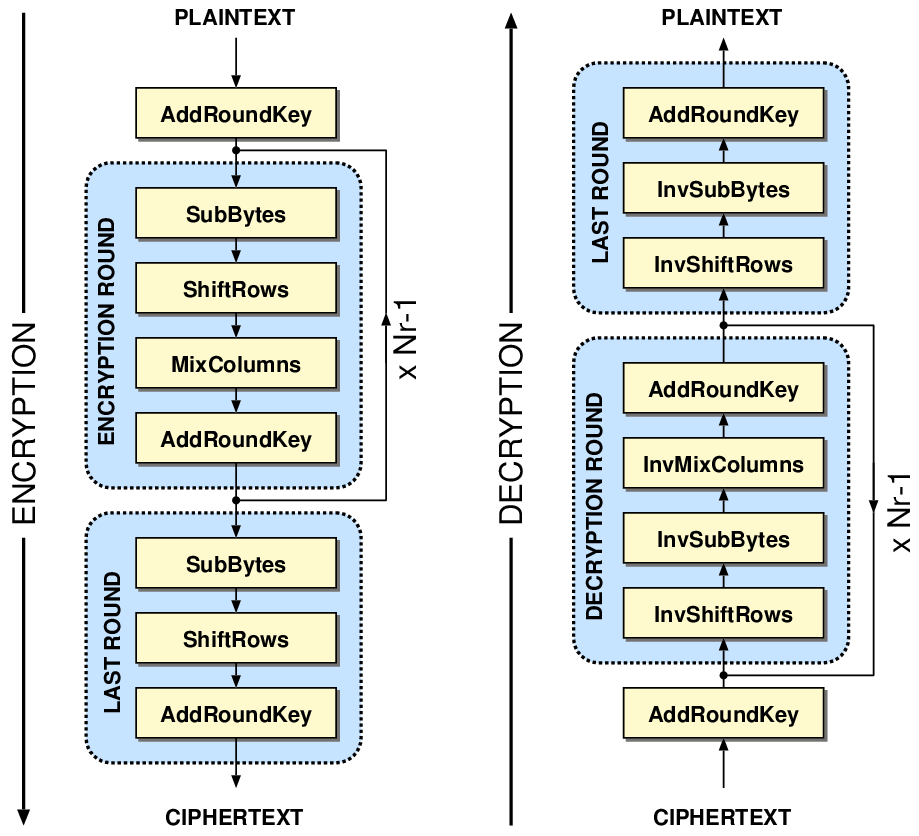
**1. Введение**

**Цель:** реализовать на языке программирования C++ алгоритм блочного шифрования AES.

**Результат:** Программа шифрования, осуществляющая криптографическое преобразование введенного текста с помощью алгоритма блочного шифрования AES.

**Общая постановка задачи:**  Создать программу, читающую данные из файла, шифрующие и дешифрующие их с помощью алгоритма блочного шифрования AES.

**2. Схема алгоритма**



**3. Скриншоты ввода данных и результатов выполнения программы**

Порядок использования программы:



Необходимо указать файл для чтения с помощью флага **–i**, выходной файл указывать необязательно, программа автоматически создаст его, но если это необходимо, можно указать после флага **–o**. По умолчанию программа работает в режиме шифрования, для дешифрования необходима указать флаг **–d**

**Использовать флаг AES(aes) для указания алгоритма**

Алгоритм использует 128 битный ключ. Для его использования в коде представлено простое API:

**void** aes\_generate\_key(uint8\_t \*roundKey, **const** uint8\_t \*key);  
**void** aes\_process\_block(uint8\_t\* block, uint8\_t\* roundKey,**bool** encrypt);

AES является блочным шифром, с ключом. Для генерации ключа представлен отдельный метод, с аргументами (*начальный ключ, указатель на адрес памяти для полного ключа*)

В качестве аргументов метода обработки блока передается (*указатель на блок данных*, *ключ*, *флаг шифрование/расшифровка*)

Содержимое тестового файла:  


Пример использования:



В результате был создан новый файл test\_encrypted:  
 

Для тестирования использовался начальный ключ  
 **2b7e151628aed2a6abf7158809cf4f3c**

Тестовые данные и результат совпали

Попробуем расшифровать сообщение, добавим флаг –d и укажем файл test\_encrypted

  
В результате получим исходный файл:



**4. Исходные файлы:**

AES.h

#ifndef MZI\_AES\_H  
#define MZI\_AES\_H  
#include <cstdint>  
**typedef** uint8\_t state\_t[4][4];  
  
**void** aes\_generate\_key(uint8\_t \*roundKey, **const** uint8\_t \*key);  
**void** aes\_process\_block(uint8\_t\* block, uint8\_t\* roundKey,**bool** encrypt);  
  
**void** aes\_add\_round\_key(uint8\_t round, state\_t\* state, uint8\_t\* roundKey);  
**void** aes\_subbytes(state\_t\* state, **bool** inv);  
**void** aes\_shift\_rows(state\_t\* state, **bool** inv);  
**void** aes\_mix\_columns(state\_t \* state,**bool** inv);  
**void** aes\_encrypt(state\_t\* state, uint8\_t\* roundKey);  
**void** aes\_decrypt(state\_t\* state,uint8\_t\* roundKey);  
  
  
**static const** uint8\_t aes\_default\_key[16] = {  
 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c  
};  
  
**static const** uint8\_t aes\_sbox[256] = {  
 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,  
 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,  
 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,  
 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,  
 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,  
 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,  
 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,  
 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,  
 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,  
 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,  
 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,  
 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,  
 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,  
 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,  
 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,  
 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16  
};  
  
**static const** uint8\_t aes\_inv\_sbox[256] = {  
 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,  
 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,  
 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,  
 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,  
 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,  
 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,  
 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,  
 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,  
 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,  
 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,  
 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,  
 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,  
 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,  
 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,  
 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,  
 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d  
};  
  
**static const** uint8\_t aes\_rcon[11] = {  
 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36  
};  
  
**static const int** aes\_ncol = 4; //col number  
**static const int** aes\_nkeys = 4; //32 words in key  
**static const int** aes\_nrounds = 10;  
  
#endif //MZI\_AES\_H

AES.cpp

//  
// Created by xdegtyarev on 1/11/18.  
//  
  
#include "aes.h"  
**void** aes\_generate\_key(uint8\_t \*roundKey, **const** uint8\_t \*key) {  
 **unsigned** i, j, k;  
 uint8\_t tempa[4];  
 // The first round key is the key itself.  
 **for** (i = 0; i < aes\_nkeys; ++i)  
 {  
 roundKey[(i \* 4) + 0] = key[(i \* 4) + 0];  
 roundKey[(i \* 4) + 1] = key[(i \* 4) + 1];  
 roundKey[(i \* 4) + 2] = key[(i \* 4) + 2];  
 roundKey[(i \* 4) + 3] = key[(i \* 4) + 3];  
 }  
  
 **for** (i = aes\_nkeys; i < aes\_ncol \* (aes\_nrounds + 1); ++i)  
 {  
 {  
 k = (i - 1) \* 4;  
 tempa[0]=roundKey[k + 0];  
 tempa[1]=roundKey[k + 1];  
 tempa[2]=roundKey[k + 2];  
 tempa[3]=roundKey[k + 3];  
 }  
 **if** (i % aes\_nkeys == 0)  
 {  
 // 4byte cyclic shift - Function RotWord()  
 {  
 k = tempa[0];  
 tempa[0] = tempa[1];  
 tempa[1] = tempa[2];  
 tempa[2] = tempa[3];  
 tempa[3] = k;  
 }  
 // Function Subword() SBOX per each 4bytes  
 {  
 tempa[0] = aes\_sbox[tempa[0]];  
 tempa[1] = aes\_sbox[tempa[1]];  
 tempa[2] = aes\_sbox[tempa[2]];  
 tempa[3] = aes\_sbox[tempa[3]];  
 }  
 tempa[0] = tempa[0] ^ aes\_rcon[i/aes\_nkeys];  
 }  
 j = i \* 4; k=(i - aes\_nkeys) \* 4;  
 roundKey[j + 0] = roundKey[k + 0] ^ tempa[0];  
 roundKey[j + 1] = roundKey[k + 1] ^ tempa[1];  
 roundKey[j + 2] = roundKey[k + 2] ^ tempa[2];  
 roundKey[j + 3] = roundKey[k + 3] ^ tempa[3];  
 }  
}  
  
**void** aes\_add\_round\_key(uint8\_t round, state\_t\* state, uint8\_t\* roundKey){  
 uint8\_t i,j;  
 **for** (i = 0; i < 4; ++i)  
 {  
 **for** (j = 0; j < 4; ++j)  
 {  
 \*state[i][j] ^= roundKey[(round \* aes\_ncol \* 4) + (i \* aes\_ncol) + j];  
 }  
 }  
}  
  
**void** aes\_subbytes(state\_t\* state, **bool** inv){  
 uint8\_t i, j;  
 **for** (i = 0; i < 4; ++i)  
 {  
 **for** (j = 0; j < 4; ++j)  
 {  
 **if**(inv) {  
 (\*state)[j][i] = aes\_inv\_sbox[(\*state)[j][i]];  
 }**else**{  
 (\*state)[j][i] = aes\_sbox[(\*state)[j][i]];  
 }  
 }  
 }  
}  
  
**void** aes\_shift\_rows(state\_t\* state, **bool** inv) {  
 uint8\_t temp;  
 **if**(inv){  
 //r1 right 1  
 temp = (\*state)[3][1];  
 (\*state)[3][1] = (\*state)[2][1];  
 (\*state)[2][1] = (\*state)[1][1];  
 (\*state)[1][1] = (\*state)[0][1];  
 (\*state)[0][1] = temp;  
 //r2 right 2  
 temp = (\*state)[0][2];  
 (\*state)[0][2] = (\*state)[2][2];  
 (\*state)[2][2] = temp;  
 temp = (\*state)[1][2];  
 (\*state)[1][2] = (\*state)[3][2];  
 (\*state)[3][2] = temp;  
 //r3 right 3  
 temp = (\*state)[0][3];  
 (\*state)[0][3] = (\*state)[1][3];  
 (\*state)[1][3] = (\*state)[2][3];  
 (\*state)[2][3] = (\*state)[3][3];  
 (\*state)[3][3] = temp;  
 }**else** {  
 //row1 left 1  
 temp = (\*state)[0][1];  
 (\*state)[0][1] = (\*state)[1][1];  
 (\*state)[1][1] = (\*state)[2][1];  
 (\*state)[2][1] = (\*state)[3][1];  
 (\*state)[3][1] = temp;  
 //row2 left 2  
 temp = (\*state)[0][2];  
 (\*state)[0][2] = (\*state)[2][2];  
 (\*state)[2][2] = temp;  
 temp = (\*state)[1][2];  
 (\*state)[1][2] = (\*state)[3][2];  
 (\*state)[3][2] = temp;  
 //row3 left 3  
 temp = (\*state)[0][3];  
 (\*state)[0][3] = (\*state)[3][3];  
 (\*state)[3][3] = (\*state)[2][3];  
 (\*state)[2][3] = (\*state)[1][3];  
 (\*state)[1][3] = temp;  
 }  
}  
  
uint8\_t xtime(uint8\_t x)  
{  
 **return** ((x<<1) ^ (((x>>7) & 1) \* 0x1b));  
}  
  
uint8\_t aes\_multiply(uint8\_t x, uint8\_t y)  
{  
 **return** (((y & 1) \* x) ^  
 ((y>>1 & 1) \* xtime(x)) ^  
 ((y>>2 & 1) \* xtime(xtime(x))) ^  
 ((y>>3 & 1) \* xtime(xtime(xtime(x)))) ^  
 ((y>>4 & 1) \* xtime(xtime(xtime(xtime(x))))));  
}  
  
**void** aes\_mix\_columns(state\_t \* state,**bool** inv)  
{  
 uint8\_t i;  
 uint8\_t a, b, c, d;  
 **for** (i = 0; i < 4; ++i)  
 {  
 **if**(inv) {  
 a = (\*state)[i][0];  
 b = (\*state)[i][1];  
 c = (\*state)[i][2];  
 d = (\*state)[i][3];  
  
 (\*state)[i][0] = aes\_multiply(a, 0x0e) ^ aes\_multiply(b, 0x0b) ^ aes\_multiply(c, 0x0d) ^ aes\_multiply(d, 0x09);  
 (\*state)[i][1] = aes\_multiply(a, 0x09) ^ aes\_multiply(b, 0x0e) ^ aes\_multiply(c, 0x0b) ^ aes\_multiply(d, 0x0d);  
 (\*state)[i][2] = aes\_multiply(a, 0x0d) ^ aes\_multiply(b, 0x09) ^ aes\_multiply(c, 0x0e) ^ aes\_multiply(d, 0x0b);  
 (\*state)[i][3] = aes\_multiply(a, 0x0b) ^ aes\_multiply(b, 0x0d) ^ aes\_multiply(c, 0x09) ^ aes\_multiply(d, 0x0e);  
 }**else** {  
 a = (\*state)[i][0];  
 b = (\*state)[i][0] ^ (\*state)[i][1] ^ (\*state)[i][2] ^ (\*state)[i][3];  
 c = (\*state)[i][0] ^ (\*state)[i][1];  
 c = xtime(c);  
 (\*state)[i][0] ^= c ^ b;  
 c = (\*state)[i][1] ^ (\*state)[i][2];  
 c = xtime(c);  
 (\*state)[i][1] ^= c ^ b;  
 c = (\*state)[i][2] ^ (\*state)[i][3];  
 c = xtime(c);  
 (\*state)[i][2] ^= c ^ b;  
 c = (\*state)[i][3] ^ a;  
 c = xtime(c);  
 (\*state)[i][3] ^= c ^ b;  
 }  
 }  
}  
  
**void** aes\_encrypt(state\_t\* state, uint8\_t\* roundKey)  
{  
 uint8\_t round = 0;  
 aes\_add\_round\_key(0, state, roundKey);  
  
 **for** (round = 1; round < aes\_nrounds; ++round)  
 {  
 aes\_subbytes(state,**false**);  
 aes\_shift\_rows(state, **false**);  
 aes\_mix\_columns(state,**false**);  
 aes\_add\_round\_key(round, state, roundKey);  
 }  
 aes\_subbytes(state,**false**);  
 aes\_shift\_rows(state,**false**);  
 aes\_add\_round\_key(aes\_nrounds, state, roundKey);  
}  
  
**void** aes\_decrypt(state\_t\* state,uint8\_t\* roundKey)  
{  
 uint8\_t round = 0;  
 aes\_add\_round\_key(aes\_nrounds, state, roundKey);  
  
 **for** (round = (aes\_nrounds - 1); round > 0; --round)  
 {  
 aes\_shift\_rows(state,**true**);  
 aes\_subbytes(state,**true**);  
 aes\_add\_round\_key(round, state, roundKey);  
 aes\_mix\_columns(state,**true**);  
 }  
 aes\_shift\_rows(state, **true**);  
 aes\_subbytes(state,**true**);  
 aes\_add\_round\_key(0, state, roundKey);  
}  
  
**void** aes\_process\_block(uint8\_t\* block, uint8\_t\* roundKey,**bool** encrypt){  
 state\_t\* state = (state\_t\*) block;  
 **if**(encrypt){  
 aes\_encrypt(state,roundKey);  
 }**else**{  
 aes\_decrypt(state,roundKey);  
 };  
}

Фрагмент Main.cpp с вызовом метода алгоритма:

}**else if**(config.type == *AES*){  
 printf("%s %s\n", "AES", config.encrypt ? "encrypt" : "decrypt");  
 uint8\_t \*roundKey = **new** uint8\_t[176];  
 aes\_generate\_key(roundKey,aes\_default\_key);  
 uint8\_t\* buf8 = (uint8\_t \*)malloc(**sizeof**(uint8\_t)\*16);  
 **while** (fread(buf8, **sizeof**(uint8\_t),16,inputFile) == 16){  
 aes\_process\_block(buf8,roundKey,config.encrypt);  
 fwrite(buf8, **sizeof**(buf8[0]),16,outputFile);  
 }  
 free(buf8);  
}

**5. Вывод:**

Симметричный алгоритм блочного шифрования AES(Advance Encryption Standart) является переемником алгоритма DES, хорошо проанализирован и сейчас широко используется, имеет поддержку в современных процессорах.