my plain text is symmetric encryption is very fast as compared to asymmetric encryption and my 256 bit key is aaaaaaaahhhhhhrrfdsertvfdsewqpnh and my iv is 1234567812345678 generate cipher text using aes algorithm with aescrypt in c

Here's an example C code that uses the Aescrypt library to encrypt a plaintext using AES with the provided key and initialization vector:

#include <stdio.h>

#include <string.h>

#include "aescrypt.h"

#define AES\_BLOCK\_SIZE 16

int main() {

char plaintext[] = "symmetric encryption is very fast compared to asymmetric encryption.";

char key[] = "aaaaaaaaahhhhhhrrfdsertvfdsewqpnh";

char iv[] = "1234567812345678";

uint8\_t key\_data[AES\_BLOCK\_SIZE \* 8];

uint8\_t iv\_data[AES\_BLOCK\_SIZE];

uint8\_t ciphertext[1024];

int ciphertext\_len;

// Convert key and IV to byte arrays

memcpy(key\_data, key, strlen(key));

memcpy(iv\_data, iv, AES\_BLOCK\_SIZE);

// Encrypt plaintext

aescrypt\_encrypt((uint8\_t\*)plaintext, ciphertext, strlen(plaintext), key\_data, sizeof(key\_data), iv\_data, sizeof(iv\_data));

ciphertext\_len = strlen(plaintext) + (AES\_BLOCK\_SIZE - strlen(plaintext) % AES\_BLOCK\_SIZE);

// Print ciphertext

printf("Ciphertext: ");

for (int i = 0; i < ciphertext\_len; i++) {

printf("%02x", ciphertext[i]);

}

printf("\n");

return 0;

}

Note that the aescrypt\_encrypt function in Aescrypt requires the plaintext to be a byte array, so we cast the plaintext string to a uint8\_t pointer.

Also note that the ciphertext length may be longer than the plaintext length due to padding, so we add (AES\_BLOCK\_SIZE - strlen(plaintext) % AES\_BLOCK\_SIZE) to the plaintext length to get the ciphertext length.

**AESCRYPT.H**

#ifndef AESCRYPT\_H

#define AESCRYPT\_H

#include <stdint.h>

#ifdef \_\_cplusplus

extern "C" {

#endif

#define AES\_BLOCK\_SIZE 16

void aescrypt\_generate\_key(const uint8\_t \*password, size\_t password\_len, uint8\_t \*key, size\_t key\_len);

void aescrypt\_encrypt(const uint8\_t \*plaintext, uint8\_t \*ciphertext, size\_t size, const uint8\_t \*key, size\_t key\_len);

void aescrypt\_decrypt(const uint8\_t \*ciphertext, uint8\_t \*plaintext, size\_t size, const uint8\_t \*key, size\_t key\_len);

#ifdef \_\_cplusplus

}

#endif

#endif /\* AESCRYPT\_H \*/

#include <stdio.h>

#include <string.h>

#include <digest/aes.h>

int main() {

// Initialize the key and IV

unsigned char key[] = "mykey12345678901";

unsigned char iv[] = "myiv12345678901";

// Initialize the plaintext string

char plaintext[] = "Hello world!";

// Initialize AES context and set key/IV

AES\_KEY aes\_ctx;

AES\_set\_encrypt\_key(key, 256, &aes\_ctx);

AES\_cbc\_encrypt(plaintext, ciphertext, strlen(plaintext), &aes\_ctx, iv, AES\_ENCRYPT);

// Print the resulting ciphertext

printf("Ciphertext: ");

for(int i = 0; i < strlen(plaintext); i++) {

printf("%02x", ciphertext[i]);

}

printf("\n");

return 0;

}

|  |
| --- |
| **AES.H**  #ifndef \_AES\_H |
|  | #define \_AES\_H |
|  |  |
|  | #ifndef uint8 |
|  | #define uint8 unsigned char |
|  | #endif |
|  |  |
|  | #ifndef uint32 |
|  | #define uint32 unsigned long int |
|  | #endif |
|  |  |
|  | typedef struct |
|  | { |
|  | uint32 erk[64]; /\* encryption round keys \*/ |
|  | uint32 drk[64]; /\* decryption round keys \*/ |
|  | int nr; /\* number of rounds \*/ |
|  | } |
|  | aes\_context; |
|  |  |
|  | int aes\_set\_key( aes\_context \*ctx, uint8 \*key, int nbits ); |
|  | void aes\_encrypt( aes\_context \*ctx, uint8 input[16], uint8 output[16] ); |
|  | void aes\_decrypt( aes\_context \*ctx, uint8 input[16], uint8 output[16] ); |
|  |  |
|  | #endif /\* aes.h \*/ |

**AES.C**

/\*

\* FIPS-197 compliant AES implementation

\*

\* Copyright (C) 2001-2004 Christophe Devine

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*/

#include "aes.h"

/\* uncomment the following line to run the test suite \*/

/\* #define TEST \*/

/\* uncomment the following line to use pre-computed tables \*/

/\* otherwise the tables will be generated at the first run \*/

#define FIXED\_TABLES

#ifndef FIXED\_TABLES

/\* forward S-box & tables \*/

uint32 FSb[256];

uint32 FT0[256];

uint32 FT1[256];

uint32 FT2[256];

uint32 FT3[256];

/\* reverse S-box & tables \*/

uint32 RSb[256];

uint32 RT0[256];

uint32 RT1[256];

uint32 RT2[256];

uint32 RT3[256];

/\* round constants \*/

uint32 RCON[10];

/\* tables generation flag \*/

int do\_init = 1;

/\* tables generation routine \*/

#define ROTR8(x) ( ( ( x << 24 ) & 0xFFFFFFFF ) | \

( ( x & 0xFFFFFFFF ) >> 8 ) )

#define XTIME(x) ( ( x << 1 ) ^ ( ( x & 0x80 ) ? 0x1B : 0x00 ) )

#define MUL(x,y) ( ( x && y ) ? pow[(log[x] + log[y]) % 255] : 0 )

void aes\_gen\_tables( void )

{

int i;

uint8 x, y;

uint8 pow[256];

uint8 log[256];

/\* compute pow and log tables over GF(2^8) \*/

for( i = 0, x = 1; i < 256; i++, x ^= XTIME( x ) )

{

pow[i] = x;

log[x] = i;

}

/\* calculate the round constants \*/

for( i = 0, x = 1; i < 10; i++, x = XTIME( x ) )

{

RCON[i] = (uint32) x << 24;

}

/\* generate the forward and reverse S-boxes \*/

FSb[0x00] = 0x63;

RSb[0x63] = 0x00;

for( i = 1; i < 256; i++ )

{

x = pow[255 - log[i]];

y = x; y = ( y << 1 ) | ( y >> 7 );

x ^= y; y = ( y << 1 ) | ( y >> 7 );

x ^= y; y = ( y << 1 ) | ( y >> 7 );

x ^= y; y = ( y << 1 ) | ( y >> 7 );

x ^= y ^ 0x63;

FSb[i] = x;

RSb[x] = i;

}

/\* generate the forward and reverse tables \*/

for( i = 0; i < 256; i++ )

{

x = (unsigned char) FSb[i]; y = XTIME( x );

FT0[i] = (uint32) ( x ^ y ) ^

( (uint32) x << 8 ) ^

( (uint32) x << 16 ) ^

( (uint32) y << 24 );

FT0[i] &= 0xFFFFFFFF;

FT1[i] = ROTR8( FT0[i] );

FT2[i] = ROTR8( FT1[i] );

FT3[i] = ROTR8( FT2[i] );

y = (unsigned char) RSb[i];

RT0[i] = ( (uint32) MUL( 0x0B, y ) ) ^

( (uint32) MUL( 0x0D, y ) << 8 ) ^

( (uint32) MUL( 0x09, y ) << 16 ) ^

( (uint32) MUL( 0x0E, y ) << 24 );

RT0[i] &= 0xFFFFFFFF;

RT1[i] = ROTR8( RT0[i] );

RT2[i] = ROTR8( RT1[i] );

RT3[i] = ROTR8( RT2[i] );

}

}

#else

/\* forward S-box \*/

static const uint32 FSb[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

};

/\* forward tables \*/

#define FT \

\

V(C6,63,63,A5), V(F8,7C,7C,84), V(EE,77,77,99), V(F6,7B,7B,8D), \

V(FF,F2,F2,0D), V(D6,6B,6B,BD), V(DE,6F,6F,B1), V(91,C5,C5,54), \

V(60,30,30,50), V(02,01,01,03), V(CE,67,67,A9), V(56,2B,2B,7D), \

V(E7,FE,FE,19), V(B5,D7,D7,62), V(4D,AB,AB,E6), V(EC,76,76,9A), \

V(8F,CA,CA,45), V(1F,82,82,9D), V(89,C9,C9,40), V(FA,7D,7D,87), \

V(EF,FA,FA,15), V(B2,59,59,EB), V(8E,47,47,C9), V(FB,F0,F0,0B), \

V(41,AD,AD,EC), V(B3,D4,D4,67), V(5F,A2,A2,FD), V(45,AF,AF,EA), \

V(23,9C,9C,BF), V(53,A4,A4,F7), V(E4,72,72,96), V(9B,C0,C0,5B), \

V(75,B7,B7,C2), V(E1,FD,FD,1C), V(3D,93,93,AE), V(4C,26,26,6A), \

V(6C,36,36,5A), V(7E,3F,3F,41), V(F5,F7,F7,02), V(83,CC,CC,4F), \

V(68,34,34,5C), V(51,A5,A5,F4), V(D1,E5,E5,34), V(F9,F1,F1,08), \

V(E2,71,71,93), V(AB,D8,D8,73), V(62,31,31,53), V(2A,15,15,3F), \

V(08,04,04,0C), V(95,C7,C7,52), V(46,23,23,65), V(9D,C3,C3,5E), \

V(30,18,18,28), V(37,96,96,A1), V(0A,05,05,0F), V(2F,9A,9A,B5), \

V(0E,07,07,09), V(24,12,12,36), V(1B,80,80,9B), V(DF,E2,E2,3D), \

V(CD,EB,EB,26), V(4E,27,27,69), V(7F,B2,B2,CD), V(EA,75,75,9F), \

V(12,09,09,1B), V(1D,83,83,9E), V(58,2C,2C,74), V(34,1A,1A,2E), \

V(36,1B,1B,2D), V(DC,6E,6E,B2), V(B4,5A,5A,EE), V(5B,A0,A0,FB), \

V(A4,52,52,F6), V(76,3B,3B,4D), V(B7,D6,D6,61), V(7D,B3,B3,CE), \

V(52,29,29,7B), V(DD,E3,E3,3E), V(5E,2F,2F,71), V(13,84,84,97), \

V(A6,53,53,F5), V(B9,D1,D1,68), V(00,00,00,00), V(C1,ED,ED,2C), \

V(40,20,20,60), V(E3,FC,FC,1F), V(79,B1,B1,C8), V(B6,5B,5B,ED), \

V(D4,6A,6A,BE), V(8D,CB,CB,46), V(67,BE,BE,D9), V(72,39,39,4B), \

V(94,4A,4A,DE), V(98,4C,4C,D4), V(B0,58,58,E8), V(85,CF,CF,4A), \

V(BB,D0,D0,6B), V(C5,EF,EF,2A), V(4F,AA,AA,E5), V(ED,FB,FB,16), \

V(86,43,43,C5), V(9A,4D,4D,D7), V(66,33,33,55), V(11,85,85,94), \

V(8A,45,45,CF), V(E9,F9,F9,10), V(04,02,02,06), V(FE,7F,7F,81), \

V(A0,50,50,F0), V(78,3C,3C,44), V(25,9F,9F,BA), V(4B,A8,A8,E3), \

V(A2,51,51,F3), V(5D,A3,A3,FE), V(80,40,40,C0), V(05,8F,8F,8A), \

V(3F,92,92,AD), V(21,9D,9D,BC), V(70,38,38,48), V(F1,F5,F5,04), \

V(63,BC,BC,DF), V(77,B6,B6,C1), V(AF,DA,DA,75), V(42,21,21,63), \

V(20,10,10,30), V(E5,FF,FF,1A), V(FD,F3,F3,0E), V(BF,D2,D2,6D), \

V(81,CD,CD,4C), V(18,0C,0C,14), V(26,13,13,35), V(C3,EC,EC,2F), \

V(BE,5F,5F,E1), V(35,97,97,A2), V(88,44,44,CC), V(2E,17,17,39), \

V(93,C4,C4,57), V(55,A7,A7,F2), V(FC,7E,7E,82), V(7A,3D,3D,47), \

V(C8,64,64,AC), V(BA,5D,5D,E7), V(32,19,19,2B), V(E6,73,73,95), \

V(C0,60,60,A0), V(19,81,81,98), V(9E,4F,4F,D1), V(A3,DC,DC,7F), \

V(44,22,22,66), V(54,2A,2A,7E), V(3B,90,90,AB), V(0B,88,88,83), \

V(8C,46,46,CA), V(C7,EE,EE,29), V(6B,B8,B8,D3), V(28,14,14,3C), \

V(A7,DE,DE,79), V(BC,5E,5E,E2), V(16,0B,0B,1D), V(AD,DB,DB,76), \

V(DB,E0,E0,3B), V(64,32,32,56), V(74,3A,3A,4E), V(14,0A,0A,1E), \

V(92,49,49,DB), V(0C,06,06,0A), V(48,24,24,6C), V(B8,5C,5C,E4), \

V(9F,C2,C2,5D), V(BD,D3,D3,6E), V(43,AC,AC,EF), V(C4,62,62,A6), \

V(39,91,91,A8), V(31,95,95,A4), V(D3,E4,E4,37), V(F2,79,79,8B), \

V(D5,E7,E7,32), V(8B,C8,C8,43), V(6E,37,37,59), V(DA,6D,6D,B7), \

V(01,8D,8D,8C), V(B1,D5,D5,64), V(9C,4E,4E,D2), V(49,A9,A9,E0), \

V(D8,6C,6C,B4), V(AC,56,56,FA), V(F3,F4,F4,07), V(CF,EA,EA,25), \

V(CA,65,65,AF), V(F4,7A,7A,8E), V(47,AE,AE,E9), V(10,08,08,18), \

V(6F,BA,BA,D5), V(F0,78,78,88), V(4A,25,25,6F), V(5C,2E,2E,72), \

V(38,1C,1C,24), V(57,A6,A6,F1), V(73,B4,B4,C7), V(97,C6,C6,51), \

V(CB,E8,E8,23), V(A1,DD,DD,7C), V(E8,74,74,9C), V(3E,1F,1F,21), \

V(96,4B,4B,DD), V(61,BD,BD,DC), V(0D,8B,8B,86), V(0F,8A,8A,85), \

V(E0,70,70,90), V(7C,3E,3E,42), V(71,B5,B5,C4), V(CC,66,66,AA), \

V(90,48,48,D8), V(06,03,03,05), V(F7,F6,F6,01), V(1C,0E,0E,12), \

V(C2,61,61,A3), V(6A,35,35,5F), V(AE,57,57,F9), V(69,B9,B9,D0), \

V(17,86,86,91), V(99,C1,C1,58), V(3A,1D,1D,27), V(27,9E,9E,B9), \

V(D9,E1,E1,38), V(EB,F8,F8,13), V(2B,98,98,B3), V(22,11,11,33), \

V(D2,69,69,BB), V(A9,D9,D9,70), V(07,8E,8E,89), V(33,94,94,A7), \

V(2D,9B,9B,B6), V(3C,1E,1E,22), V(15,87,87,92), V(C9,E9,E9,20), \

V(87,CE,CE,49), V(AA,55,55,FF), V(50,28,28,78), V(A5,DF,DF,7A), \

V(03,8C,8C,8F), V(59,A1,A1,F8), V(09,89,89,80), V(1A,0D,0D,17), \

V(65,BF,BF,DA), V(D7,E6,E6,31), V(84,42,42,C6), V(D0,68,68,B8), \

V(82,41,41,C3), V(29,99,99,B0), V(5A,2D,2D,77), V(1E,0F,0F,11), \

V(7B,B0,B0,CB), V(A8,54,54,FC), V(6D,BB,BB,D6), V(2C,16,16,3A)

#define V(a,b,c,d) 0x##a##b##c##d

static const uint32 FT0[256] = { FT };

#undef V

#define V(a,b,c,d) 0x##d##a##b##c

static const uint32 FT1[256] = { FT };

#undef V

#define V(a,b,c,d) 0x##c##d##a##b

static const uint32 FT2[256] = { FT };

#undef V

#define V(a,b,c,d) 0x##b##c##d##a

static const uint32 FT3[256] = { FT };

#undef V

#undef FT

/\* reverse S-box \*/

static const uint32 RSb[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

};

/\* reverse tables \*/

#define RT \

\

V(51,F4,A7,50), V(7E,41,65,53), V(1A,17,A4,C3), V(3A,27,5E,96), \

V(3B,AB,6B,CB), V(1F,9D,45,F1), V(AC,FA,58,AB), V(4B,E3,03,93), \

V(20,30,FA,55), V(AD,76,6D,F6), V(88,CC,76,91), V(F5,02,4C,25), \

V(4F,E5,D7,FC), V(C5,2A,CB,D7), V(26,35,44,80), V(B5,62,A3,8F), \

V(DE,B1,5A,49), V(25,BA,1B,67), V(45,EA,0E,98), V(5D,FE,C0,E1), \

V(C3,2F,75,02), V(81,4C,F0,12), V(8D,46,97,A3), V(6B,D3,F9,C6), \

V(03,8F,5F,E7), V(15,92,9C,95), V(BF,6D,7A,EB), V(95,52,59,DA), \

V(D4,BE,83,2D), V(58,74,21,D3), V(49,E0,69,29), V(8E,C9,C8,44), \

V(75,C2,89,6A), V(F4,8E,79,78), V(99,58,3E,6B), V(27,B9,71,DD), \

V(BE,E1,4F,B6), V(F0,88,AD,17), V(C9,20,AC,66), V(7D,CE,3A,B4), \

V(63,DF,4A,18), V(E5,1A,31,82), V(97,51,33,60), V(62,53,7F,45), \

V(B1,64,77,E0), V(BB,6B,AE,84), V(FE,81,A0,1C), V(F9,08,2B,94), \

V(70,48,68,58), V(8F,45,FD,19), V(94,DE,6C,87), V(52,7B,F8,B7), \

V(AB,73,D3,23), V(72,4B,02,E2), V(E3,1F,8F,57), V(66,55,AB,2A), \

V(B2,EB,28,07), V(2F,B5,C2,03), V(86,C5,7B,9A), V(D3,37,08,A5), \

V(30,28,87,F2), V(23,BF,A5,B2), V(02,03,6A,BA), V(ED,16,82,5C), \

V(8A,CF,1C,2B), V(A7,79,B4,92), V(F3,07,F2,F0), V(4E,69,E2,A1), \

V(65,DA,F4,CD), V(06,05,BE,D5), V(D1,34,62,1F), V(C4,A6,FE,8A), \

V(34,2E,53,9D), V(A2,F3,55,A0), V(05,8A,E1,32), V(A4,F6,EB,75), \

V(0B,83,EC,39), V(40,60,EF,AA), V(5E,71,9F,06), V(BD,6E,10,51), \

V(3E,21,8A,F9), V(96,DD,06,3D), V(DD,3E,05,AE), V(4D,E6,BD,46), \

V(91,54,8D,B5), V(71,C4,5D,05), V(04,06,D4,6F), V(60,50,15,FF), \

V(19,98,FB,24), V(D6,BD,E9,97), V(89,40,43,CC), V(67,D9,9E,77), \

V(B0,E8,42,BD), V(07,89,8B,88), V(E7,19,5B,38), V(79,C8,EE,DB), \

V(A1,7C,0A,47), V(7C,42,0F,E9), V(F8,84,1E,C9), V(00,00,00,00), \

V(09,80,86,83), V(32,2B,ED,48), V(1E,11,70,AC), V(6C,5A,72,4E), \

V(FD,0E,FF,FB), V(0F,85,38,56), V(3D,AE,D5,1E), V(36,2D,39,27), \

V(0A,0F,D9,64), V(68,5C,A6,21), V(9B,5B,54,D1), V(24,36,2E,3A), \

V(0C,0A,67,B1), V(93,57,E7,0F), V(B4,EE,96,D2), V(1B,9B,91,9E), \

V(80,C0,C5,4F), V(61,DC,20,A2), V(5A,77,4B,69), V(1C,12,1A,16), \

V(E2,93,BA,0A), V(C0,A0,2A,E5), V(3C,22,E0,43), V(12,1B,17,1D), \

V(0E,09,0D,0B), V(F2,8B,C7,AD), V(2D,B6,A8,B9), V(14,1E,A9,C8), \

V(57,F1,19,85), V(AF,75,07,4C), V(EE,99,DD,BB), V(A3,7F,60,FD), \

V(F7,01,26,9F), V(5C,72,F5,BC), V(44,66,3B,C5), V(5B,FB,7E,34), \

V(8B,43,29,76), V(CB,23,C6,DC), V(B6,ED,FC,68), V(B8,E4,F1,63), \

V(D7,31,DC,CA), V(42,63,85,10), V(13,97,22,40), V(84,C6,11,20), \

V(85,4A,24,7D), V(D2,BB,3D,F8), V(AE,F9,32,11), V(C7,29,A1,6D), \

V(1D,9E,2F,4B), V(DC,B2,30,F3), V(0D,86,52,EC), V(77,C1,E3,D0), \

V(2B,B3,16,6C), V(A9,70,B9,99), V(11,94,48,FA), V(47,E9,64,22), \

V(A8,FC,8C,C4), V(A0,F0,3F,1A), V(56,7D,2C,D8), V(22,33,90,EF), \

V(87,49,4E,C7), V(D9,38,D1,C1), V(8C,CA,A2,FE), V(98,D4,0B,36), \

V(A6,F5,81,CF), V(A5,7A,DE,28), V(DA,B7,8E,26), V(3F,AD,BF,A4), \

V(2C,3A,9D,E4), V(50,78,92,0D), V(6A,5F,CC,9B), V(54,7E,46,62), \

V(F6,8D,13,C2), V(90,D8,B8,E8), V(2E,39,F7,5E), V(82,C3,AF,F5), \

V(9F,5D,80,BE), V(69,D0,93,7C), V(6F,D5,2D,A9), V(CF,25,12,B3), \

V(C8,AC,99,3B), V(10,18,7D,A7), V(E8,9C,63,6E), V(DB,3B,BB,7B), \

V(CD,26,78,09), V(6E,59,18,F4), V(EC,9A,B7,01), V(83,4F,9A,A8), \

V(E6,95,6E,65), V(AA,FF,E6,7E), V(21,BC,CF,08), V(EF,15,E8,E6), \

V(BA,E7,9B,D9), V(4A,6F,36,CE), V(EA,9F,09,D4), V(29,B0,7C,D6), \

V(31,A4,B2,AF), V(2A,3F,23,31), V(C6,A5,94,30), V(35,A2,66,C0), \

V(74,4E,BC,37), V(FC,82,CA,A6), V(E0,90,D0,B0), V(33,A7,D8,15), \

V(F1,04,98,4A), V(41,EC,DA,F7), V(7F,CD,50,0E), V(17,91,F6,2F), \

V(76,4D,D6,8D), V(43,EF,B0,4D), V(CC,AA,4D,54), V(E4,96,04,DF), \

V(9E,D1,B5,E3), V(4C,6A,88,1B), V(C1,2C,1F,B8), V(46,65,51,7F), \

V(9D,5E,EA,04), V(01,8C,35,5D), V(FA,87,74,73), V(FB,0B,41,2E), \

V(B3,67,1D,5A), V(92,DB,D2,52), V(E9,10,56,33), V(6D,D6,47,13), \

V(9A,D7,61,8C), V(37,A1,0C,7A), V(59,F8,14,8E), V(EB,13,3C,89), \

V(CE,A9,27,EE), V(B7,61,C9,35), V(E1,1C,E5,ED), V(7A,47,B1,3C), \

V(9C,D2,DF,59), V(55,F2,73,3F), V(18,14,CE,79), V(73,C7,37,BF), \

V(53,F7,CD,EA), V(5F,FD,AA,5B), V(DF,3D,6F,14), V(78,44,DB,86), \

V(CA,AF,F3,81), V(B9,68,C4,3E), V(38,24,34,2C), V(C2,A3,40,5F), \

V(16,1D,C3,72), V(BC,E2,25,0C), V(28,3C,49,8B), V(FF,0D,95,41), \

V(39,A8,01,71), V(08,0C,B3,DE), V(D8,B4,E4,9C), V(64,56,C1,90), \

V(7B,CB,84,61), V(D5,32,B6,70), V(48,6C,5C,74), V(D0,B8,57,42)

#define V(a,b,c,d) 0x##a##b##c##d

static const uint32 RT0[256] = { RT };

#undef V

#define V(a,b,c,d) 0x##d##a##b##c

static const uint32 RT1[256] = { RT };

#undef V

#define V(a,b,c,d) 0x##c##d##a##b

static const uint32 RT2[256] = { RT };

#undef V

#define V(a,b,c,d) 0x##b##c##d##a

static const uint32 RT3[256] = { RT };

#undef V

#undef RT

/\* round constants \*/

static const uint32 RCON[10] =

{

0x01000000, 0x02000000, 0x04000000, 0x08000000,

0x10000000, 0x20000000, 0x40000000, 0x80000000,

0x1B000000, 0x36000000

};

int do\_init = 0;

void aes\_gen\_tables( void ) /\* #nocov start \*/

{

} /\* #nocov end \*/

#endif

/\* platform-independant 32-bit integer manipulation macros \*/

#define GET\_UINT32(n,b,i) \

{ \

(n) = ( (uint32) (b)[(i) ] << 24 ) \

| ( (uint32) (b)[(i) + 1] << 16 ) \

| ( (uint32) (b)[(i) + 2] << 8 ) \

| ( (uint32) (b)[(i) + 3] ); \

}

#define PUT\_UINT32(n,b,i) \

{ \

(b)[(i) ] = (uint8) ( (n) >> 24 ); \

(b)[(i) + 1] = (uint8) ( (n) >> 16 ); \

(b)[(i) + 2] = (uint8) ( (n) >> 8 ); \

(b)[(i) + 3] = (uint8) ( (n) ); \

}

/\* decryption key schedule tables \*/

int KT\_init = 1;

uint32 KT0[256];

uint32 KT1[256];

uint32 KT2[256];

uint32 KT3[256];

/\* AES key scheduling routine \*/

int aes\_set\_key( aes\_context \*ctx, uint8 \*key, int nbits )

{

int i;

uint32 \*RK, \*SK;

if( do\_init )

{

aes\_gen\_tables(); /\* #nocov start \*/

do\_init = 0; /\* #nocov end \*/

}

switch( nbits )

{

case 128: ctx->nr = 10; break;

case 192: ctx->nr = 12; break;

case 256: ctx->nr = 14; break;

default : return( 1 ); /\* #nocov \*/

}

RK = ctx->erk;

for( i = 0; i < (nbits >> 5); i++ )

{

GET\_UINT32( RK[i], key, i \* 4 );

}

/\* setup encryption round keys \*/

switch( nbits )

{

case 128:

for( i = 0; i < 10; i++, RK += 4 )

{

RK[4] = RK[0] ^ RCON[i] ^

( FSb[ (uint8) ( RK[3] >> 16 ) ] << 24 ) ^

( FSb[ (uint8) ( RK[3] >> 8 ) ] << 16 ) ^

( FSb[ (uint8) ( RK[3] ) ] << 8 ) ^

( FSb[ (uint8) ( RK[3] >> 24 ) ] );

RK[5] = RK[1] ^ RK[4];

RK[6] = RK[2] ^ RK[5];

RK[7] = RK[3] ^ RK[6];

}

break;

case 192:

for( i = 0; i < 8; i++, RK += 6 )

{

RK[6] = RK[0] ^ RCON[i] ^

( FSb[ (uint8) ( RK[5] >> 16 ) ] << 24 ) ^

( FSb[ (uint8) ( RK[5] >> 8 ) ] << 16 ) ^

( FSb[ (uint8) ( RK[5] ) ] << 8 ) ^

( FSb[ (uint8) ( RK[5] >> 24 ) ] );

RK[7] = RK[1] ^ RK[6];

RK[8] = RK[2] ^ RK[7];

RK[9] = RK[3] ^ RK[8];

RK[10] = RK[4] ^ RK[9];

RK[11] = RK[5] ^ RK[10];

}

break;

case 256:

for( i = 0; i < 7; i++, RK += 8 )

{

RK[8] = RK[0] ^ RCON[i] ^

( FSb[ (uint8) ( RK[7] >> 16 ) ] << 24 ) ^

( FSb[ (uint8) ( RK[7] >> 8 ) ] << 16 ) ^

( FSb[ (uint8) ( RK[7] ) ] << 8 ) ^

( FSb[ (uint8) ( RK[7] >> 24 ) ] );

RK[9] = RK[1] ^ RK[8];

RK[10] = RK[2] ^ RK[9];

RK[11] = RK[3] ^ RK[10];

RK[12] = RK[4] ^

( FSb[ (uint8) ( RK[11] >> 24 ) ] << 24 ) ^

( FSb[ (uint8) ( RK[11] >> 16 ) ] << 16 ) ^

( FSb[ (uint8) ( RK[11] >> 8 ) ] << 8 ) ^

( FSb[ (uint8) ( RK[11] ) ] );

RK[13] = RK[5] ^ RK[12];

RK[14] = RK[6] ^ RK[13];

RK[15] = RK[7] ^ RK[14];

}

break;

}

/\* setup decryption round keys \*/

if( KT\_init )

{

for( i = 0; i < 256; i++ )

{

KT0[i] = RT0[ FSb[i] ];

KT1[i] = RT1[ FSb[i] ];

KT2[i] = RT2[ FSb[i] ];

KT3[i] = RT3[ FSb[i] ];

}

KT\_init = 0;

}

SK = ctx->drk;

\*SK++ = \*RK++;

\*SK++ = \*RK++;

\*SK++ = \*RK++;

\*SK++ = \*RK++;

for( i = 1; i < ctx->nr; i++ )

{

RK -= 8;

\*SK++ = KT0[ (uint8) ( \*RK >> 24 ) ] ^

KT1[ (uint8) ( \*RK >> 16 ) ] ^

KT2[ (uint8) ( \*RK >> 8 ) ] ^

KT3[ (uint8) ( \*RK ) ]; RK++;

\*SK++ = KT0[ (uint8) ( \*RK >> 24 ) ] ^

KT1[ (uint8) ( \*RK >> 16 ) ] ^

KT2[ (uint8) ( \*RK >> 8 ) ] ^

KT3[ (uint8) ( \*RK ) ]; RK++;

\*SK++ = KT0[ (uint8) ( \*RK >> 24 ) ] ^

KT1[ (uint8) ( \*RK >> 16 ) ] ^

KT2[ (uint8) ( \*RK >> 8 ) ] ^

KT3[ (uint8) ( \*RK ) ]; RK++;

\*SK++ = KT0[ (uint8) ( \*RK >> 24 ) ] ^

KT1[ (uint8) ( \*RK >> 16 ) ] ^

KT2[ (uint8) ( \*RK >> 8 ) ] ^

KT3[ (uint8) ( \*RK ) ]; RK++;

}

RK -= 8;

\*SK++ = \*RK++;

\*SK++ = \*RK++;

\*SK++ = \*RK++;

\*SK++ = \*RK++;

return( 0 );

}

/\* AES 128-bit block encryption routine \*/

void aes\_encrypt( aes\_context \*ctx, uint8 input[16], uint8 output[16] )

{

uint32 \*RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;

RK = ctx->erk;

GET\_UINT32( X0, input, 0 ); X0 ^= RK[0];

GET\_UINT32( X1, input, 4 ); X1 ^= RK[1];

GET\_UINT32( X2, input, 8 ); X2 ^= RK[2];

GET\_UINT32( X3, input, 12 ); X3 ^= RK[3];

#define AES\_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3) \

{ \

RK += 4; \

\

X0 = RK[0] ^ FT0[ (uint8) ( Y0 >> 24 ) ] ^ \

FT1[ (uint8) ( Y1 >> 16 ) ] ^ \

FT2[ (uint8) ( Y2 >> 8 ) ] ^ \

FT3[ (uint8) ( Y3 ) ]; \

\

X1 = RK[1] ^ FT0[ (uint8) ( Y1 >> 24 ) ] ^ \

FT1[ (uint8) ( Y2 >> 16 ) ] ^ \

FT2[ (uint8) ( Y3 >> 8 ) ] ^ \

FT3[ (uint8) ( Y0 ) ]; \

\

X2 = RK[2] ^ FT0[ (uint8) ( Y2 >> 24 ) ] ^ \

FT1[ (uint8) ( Y3 >> 16 ) ] ^ \

FT2[ (uint8) ( Y0 >> 8 ) ] ^ \

FT3[ (uint8) ( Y1 ) ]; \

\

X3 = RK[3] ^ FT0[ (uint8) ( Y3 >> 24 ) ] ^ \

FT1[ (uint8) ( Y0 >> 16 ) ] ^ \

FT2[ (uint8) ( Y1 >> 8 ) ] ^ \

FT3[ (uint8) ( Y2 ) ]; \

}

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 1 \*/

AES\_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 2 \*/

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 3 \*/

AES\_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 4 \*/

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 5 \*/

AES\_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 6 \*/

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 7 \*/

AES\_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 8 \*/

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 9 \*/

if( ctx->nr > 10 )

{

AES\_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 10 \*/

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 11 \*/

}

if( ctx->nr > 12 )

{

AES\_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 12 \*/

AES\_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 13 \*/

}

/\* last round \*/

RK += 4;

X0 = RK[0] ^ ( FSb[ (uint8) ( Y0 >> 24 ) ] << 24 ) ^

( FSb[ (uint8) ( Y1 >> 16 ) ] << 16 ) ^

( FSb[ (uint8) ( Y2 >> 8 ) ] << 8 ) ^

( FSb[ (uint8) ( Y3 ) ] );

X1 = RK[1] ^ ( FSb[ (uint8) ( Y1 >> 24 ) ] << 24 ) ^

( FSb[ (uint8) ( Y2 >> 16 ) ] << 16 ) ^

( FSb[ (uint8) ( Y3 >> 8 ) ] << 8 ) ^

( FSb[ (uint8) ( Y0 ) ] );

X2 = RK[2] ^ ( FSb[ (uint8) ( Y2 >> 24 ) ] << 24 ) ^

( FSb[ (uint8) ( Y3 >> 16 ) ] << 16 ) ^

( FSb[ (uint8) ( Y0 >> 8 ) ] << 8 ) ^

( FSb[ (uint8) ( Y1 ) ] );

X3 = RK[3] ^ ( FSb[ (uint8) ( Y3 >> 24 ) ] << 24 ) ^

( FSb[ (uint8) ( Y0 >> 16 ) ] << 16 ) ^

( FSb[ (uint8) ( Y1 >> 8 ) ] << 8 ) ^

( FSb[ (uint8) ( Y2 ) ] );

PUT\_UINT32( X0, output, 0 );

PUT\_UINT32( X1, output, 4 );

PUT\_UINT32( X2, output, 8 );

PUT\_UINT32( X3, output, 12 );

}

/\* AES 128-bit block decryption routine \*/

void aes\_decrypt( aes\_context \*ctx, uint8 input[16], uint8 output[16] )

{

uint32 \*RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;

RK = ctx->drk;

GET\_UINT32( X0, input, 0 ); X0 ^= RK[0];

GET\_UINT32( X1, input, 4 ); X1 ^= RK[1];

GET\_UINT32( X2, input, 8 ); X2 ^= RK[2];

GET\_UINT32( X3, input, 12 ); X3 ^= RK[3];

#define AES\_RROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3) \

{ \

RK += 4; \

\

X0 = RK[0] ^ RT0[ (uint8) ( Y0 >> 24 ) ] ^ \

RT1[ (uint8) ( Y3 >> 16 ) ] ^ \

RT2[ (uint8) ( Y2 >> 8 ) ] ^ \

RT3[ (uint8) ( Y1 ) ]; \

\

X1 = RK[1] ^ RT0[ (uint8) ( Y1 >> 24 ) ] ^ \

RT1[ (uint8) ( Y0 >> 16 ) ] ^ \

RT2[ (uint8) ( Y3 >> 8 ) ] ^ \

RT3[ (uint8) ( Y2 ) ]; \

\

X2 = RK[2] ^ RT0[ (uint8) ( Y2 >> 24 ) ] ^ \

RT1[ (uint8) ( Y1 >> 16 ) ] ^ \

RT2[ (uint8) ( Y0 >> 8 ) ] ^ \

RT3[ (uint8) ( Y3 ) ]; \

\

X3 = RK[3] ^ RT0[ (uint8) ( Y3 >> 24 ) ] ^ \

RT1[ (uint8) ( Y2 >> 16 ) ] ^ \

RT2[ (uint8) ( Y1 >> 8 ) ] ^ \

RT3[ (uint8) ( Y0 ) ]; \

}

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 1 \*/

AES\_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 2 \*/

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 3 \*/

AES\_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 4 \*/

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 5 \*/

AES\_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 6 \*/

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 7 \*/

AES\_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 8 \*/

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 9 \*/

if( ctx->nr > 10 )

{

AES\_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 10 \*/

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 11 \*/

}

if( ctx->nr > 12 )

{

AES\_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); /\* round 12 \*/

AES\_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); /\* round 13 \*/

}

/\* last round \*/

RK += 4;

X0 = RK[0] ^ ( RSb[ (uint8) ( Y0 >> 24 ) ] << 24 ) ^

( RSb[ (uint8) ( Y3 >> 16 ) ] << 16 ) ^

( RSb[ (uint8) ( Y2 >> 8 ) ] << 8 ) ^

( RSb[ (uint8) ( Y1 ) ] );

X1 = RK[1] ^ ( RSb[ (uint8) ( Y1 >> 24 ) ] << 24 ) ^

( RSb[ (uint8) ( Y0 >> 16 ) ] << 16 ) ^

( RSb[ (uint8) ( Y3 >> 8 ) ] << 8 ) ^

( RSb[ (uint8) ( Y2 ) ] );

X2 = RK[2] ^ ( RSb[ (uint8) ( Y2 >> 24 ) ] << 24 ) ^

( RSb[ (uint8) ( Y1 >> 16 ) ] << 16 ) ^

( RSb[ (uint8) ( Y0 >> 8 ) ] << 8 ) ^

( RSb[ (uint8) ( Y3 ) ] );

X3 = RK[3] ^ ( RSb[ (uint8) ( Y3 >> 24 ) ] << 24 ) ^

( RSb[ (uint8) ( Y2 >> 16 ) ] << 16 ) ^

( RSb[ (uint8) ( Y1 >> 8 ) ] << 8 ) ^

( RSb[ (uint8) ( Y0 ) ] );

PUT\_UINT32( X0, output, 0 );

PUT\_UINT32( X1, output, 4 );

PUT\_UINT32( X2, output, 8 );

PUT\_UINT32( X3, output, 12 );

}

#ifdef TEST

#include <string.h>

#include <stdio.h>

/\*

\* Rijndael Monte Carlo Test: ECB mode

\* source: NIST - rijndael-vals.zip

\*/

static unsigned char AES\_enc\_test[3][16] =

{

{ 0xA0, 0x43, 0x77, 0xAB, 0xE2, 0x59, 0xB0, 0xD0,

0xB5, 0xBA, 0x2D, 0x40, 0xA5, 0x01, 0x97, 0x1B },

{ 0x4E, 0x46, 0xF8, 0xC5, 0x09, 0x2B, 0x29, 0xE2,

0x9A, 0x97, 0x1A, 0x0C, 0xD1, 0xF6, 0x10, 0xFB },

{ 0x1F, 0x67, 0x63, 0xDF, 0x80, 0x7A, 0x7E, 0x70,

0x96, 0x0D, 0x4C, 0xD3, 0x11, 0x8E, 0x60, 0x1A }

};

static unsigned char AES\_dec\_test[3][16] =

{

{ 0xF5, 0xBF, 0x8B, 0x37, 0x13, 0x6F, 0x2E, 0x1F,

0x6B, 0xEC, 0x6F, 0x57, 0x20, 0x21, 0xE3, 0xBA },

{ 0xF1, 0xA8, 0x1B, 0x68, 0xF6, 0xE5, 0xA6, 0x27,

0x1A, 0x8C, 0xB2, 0x4E, 0x7D, 0x94, 0x91, 0xEF },

{ 0x4D, 0xE0, 0xC6, 0xDF, 0x7C, 0xB1, 0x69, 0x72,

0x84, 0x60, 0x4D, 0x60, 0x27, 0x1B, 0xC5, 0x9A }

};

int main( void )

{

int m, n, i, j;

aes\_context ctx;

unsigned char buf[16];

unsigned char key[32];

for( m = 0; m < 2; m++ )

{

printf( "\n Rijndael Monte Carlo Test (ECB mode) - " );

if( m == 0 ) printf( "encryption\n\n" );

if( m == 1 ) printf( "decryption\n\n" );

for( n = 0; n < 3; n++ )

{

printf( " Test %d, key size = %3d bits: ",

n + 1, 128 + n \* 64 );

fflush( stdout );

memset( buf, 0, 16 );

memset( key, 0, 16 + n \* 8 );

for( i = 0; i < 400; i++ )

{

aes\_set\_key( &ctx, key, 128 + n \* 64 );

for( j = 0; j < 9999; j++ )

{

if( m == 0 ) aes\_encrypt( &ctx, buf, buf );

if( m == 1 ) aes\_decrypt( &ctx, buf, buf );

}

if( n > 0 )

{

for( j = 0; j < (n << 3); j++ )

{

key[j] ^= buf[j + 16 - (n << 3)];

}

}

if( m == 0 ) aes\_encrypt( &ctx, buf, buf );

if( m == 1 ) aes\_decrypt( &ctx, buf, buf );

for( j = 0; j < 16; j++ )

{

key[j + (n << 3)] ^= buf[j];

}

}

if( ( m == 0 && memcmp( buf, AES\_enc\_test[n], 16 ) != 0 ) ||

( m == 1 && memcmp( buf, AES\_dec\_test[n], 16 ) != 0 ) )

{

printf( "failed!\n" );

return( 1 );

}

printf( "passed.\n" );

}

}

printf( "\n" );

return( 0 );

}

#endif