21. Write a C program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?

#include <stdio.h>

#include <string.h>

#define BLOCK\_SIZE 8

void xor\_encrypt(unsigned char \*block, unsigned char \*key) {

for (int i = 0; i < BLOCK\_SIZE; i++) {

block[i] ^= key[i];

}

}

void xor\_blocks(unsigned char \*out, unsigned char \*a, unsigned char \*b) {

for (int i = 0; i < BLOCK\_SIZE; i++) {

out[i] = a[i] ^ b[i];

}

}

int pad(unsigned char \*input, int len) {

int pad\_len = BLOCK\_SIZE - (len % BLOCK\_SIZE);

input[len] = 0x80;

for (int i = 1; i < pad\_len; i++)

input[len + i] = 0x00;

return len + pad\_len;

}

void print\_hex(const char \*label, unsigned char \*data, int len) {

printf("%s: ", label);

for (int i = 0; i < len; i++) printf("%02x", data[i]);

printf("\n");

}

int main() {

unsigned char key[BLOCK\_SIZE] = "mysecret";

unsigned char iv[BLOCK\_SIZE] = "initvect";

unsigned char plaintext[64] = "This is a test of ECB, CBC, and CFB modes.";

unsigned char padded[80];

unsigned char ecb[80], cbc[80], cfb[80];

unsigned char block[BLOCK\_SIZE];

int len = strlen((char \*)plaintext);

memcpy(padded, plaintext, len);

int padded\_len = pad(padded, len);

printf("Plaintext (padded): %s\n", padded);

for (int i = 0; i < padded\_len; i += BLOCK\_SIZE) {

memcpy(block, &padded[i], BLOCK\_SIZE);

xor\_encrypt(block, key);

memcpy(&ecb[i], block, BLOCK\_SIZE);

}

unsigned char prev[BLOCK\_SIZE];

memcpy(prev, iv, BLOCK\_SIZE);

for (int i = 0; i < padded\_len; i += BLOCK\_SIZE) {

xor\_blocks(block, &padded[i], prev);

xor\_encrypt(block, key);

memcpy(&cbc[i], block, BLOCK\_SIZE);

memcpy(prev, block, BLOCK\_SIZE);

}

memcpy(prev, iv, BLOCK\_SIZE);

for (int i = 0; i < padded\_len; i += BLOCK\_SIZE) {

memcpy(block, prev, BLOCK\_SIZE);

xor\_encrypt(block, key);

xor\_blocks(&cfb[i], &padded[i], block);

memcpy(prev, &cfb[i], BLOCK\_SIZE);

}

print\_hex("ECB Mode Cipher", ecb, padded\_len);

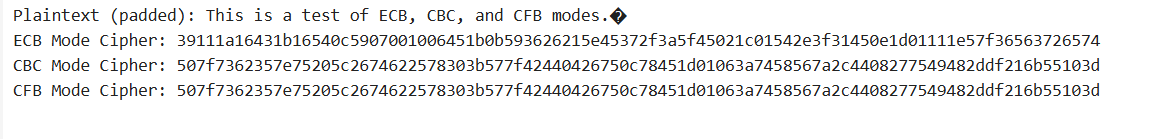
print\_hex("CBC Mode Cipher", cbc, padded\_len);

print\_hex("CFB Mode Cipher", cfb, padded\_len);

return 0;

}

OUTPUT:



22. Write a C program for Encrypt and decrypt in cipher block chaining mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.

#include <stdio.h>

#include <stdint.h>

uint8\_t IP[] = {1, 5, 2, 0, 3, 7, 4, 6};

uint8\_t IP\_INV[] = {3, 0, 2, 4, 6, 1, 7, 5};

uint8\_t EP[] = {3, 0, 1, 2, 1, 2, 3, 0};

uint8\_t P4[] = {1, 3, 2, 0};

uint8\_t P10[] = {2, 4, 1, 6, 3, 9, 0, 8, 7, 5};

uint8\_t P8[] = {5, 2, 6, 3, 7, 4, 9, 8};

uint8\_t S0[4][4] = {

{1, 0, 3, 2},

{3, 2, 1, 0},

{0, 2, 1, 3},

{3, 1, 3, 2}

};

uint8\_t S1[4][4] = {

{0, 1, 2, 3},

{2, 0, 1, 3},

{3, 0, 1, 0},

{2, 1, 0, 3}

};

uint8\_t permute(uint8\_t in, uint8\_t\* p, int n) {

uint8\_t out = 0;

for (int i = 0; i < n; i++) {

out <<= 1;

out |= (in >> (7 - p[i])) & 1;

}

return out;

}

uint8\_t leftShift5(uint8\_t in, int shifts) {

return ((in << shifts) | (in >> (5 - shifts))) & 0x1F;

}

uint16\_t keyGen(uint16\_t key, uint8\_t\* k1, uint8\_t\* k2) {

uint16\_t perm = 0;

for (int i = 0; i < 10; i++) {

perm <<= 1;

perm |= (key >> (9 - P10[i])) & 1;

}

uint8\_t left = (perm >> 5) & 0x1F;

uint8\_t right = perm & 0x1F;

left = leftShift5(left, 1);

right = leftShift5(right, 1);

uint16\_t merged = (left << 5) | right;

\*k1 = 0;

for (int i = 0; i < 8; i++) {

\*k1 <<= 1;

\*k1 |= (merged >> (9 - P8[i])) & 1;

}

left = leftShift5(left, 2);

right = leftShift5(right, 2);

merged = (left << 5) | right;

\*k2 = 0;

for (int i = 0; i < 8; i++) {

\*k2 <<= 1;

\*k2 |= (merged >> (9 - P8[i])) & 1;

}

return 0;

}

uint8\_t sbox(uint8\_t in, uint8\_t box[4][4]) {

uint8\_t row = ((in & 0x8) >> 2) | (in & 0x1);

uint8\_t col = (in >> 1) & 0x3;

return box[row][col];

}

uint8\_t f(uint8\_t r, uint8\_t sk) {

uint8\_t ep = 0;

for (int i = 0; i < 8; i++) {

ep <<= 1;

ep |= (r >> (3 - EP[i])) & 1;

}

uint8\_t x = ep ^ sk;

uint8\_t left = (x >> 4) & 0xF;

uint8\_t right = x & 0xF;

uint8\_t out = (sbox(left, S0) << 2) | sbox(right, S1);

uint8\_t p4out = 0;

for (int i = 0; i < 4; i++) {

p4out <<= 1;

p4out |= (out >> (3 - P4[i])) & 1;

}

return p4out;

}

uint8\_t fk(uint8\_t in, uint8\_t k1, uint8\_t k2, int isDecrypt) {

uint8\_t ip = permute(in, IP, 8);

uint8\_t left = ip >> 4;

uint8\_t right = ip & 0xF;

uint8\_t t1 = f(right, isDecrypt ? k2 : k1);

left ^= t1;

uint8\_t swapped = (right << 4) | left;

right = swapped & 0xF;

left = swapped >> 4;

uint8\_t t2 = f(right, isDecrypt ? k1 : k2);

left ^= t2;

uint8\_t preout = (left << 4) | right;

uint8\_t out = permute(preout, IP\_INV, 8);

return out;

}

void encryptCBC(uint8\_t\* pt, uint8\_t\* ct, int n, uint8\_t k1, uint8\_t k2, uint8\_t iv) {

uint8\_t prev = iv;

for (int i = 0; i < n; i++) {

uint8\_t block = pt[i] ^ prev;

ct[i] = fk(block, k1, k2, 0);

prev = ct[i];

}

}

void decryptCBC(uint8\_t\* ct, uint8\_t\* pt, int n, uint8\_t k1, uint8\_t k2, uint8\_t iv) {

uint8\_t prev = iv;

for (int i = 0; i < n; i++) {

uint8\_t temp = fk(ct[i], k1, k2, 1);

pt[i] = temp ^ prev;

prev = ct[i];

}

}

int main() {

uint8\_t plaintext[2] = {0x01, 0x23};

uint8\_t ciphertext[2], decrypted[2];

uint8\_t iv = 0xAA;

uint8\_t k1, k2;

uint16\_t key = 0x1FD;

keyGen(key, &k1, &k2);

encryptCBC(plaintext, ciphertext, 2, k1, k2, iv);

printf("Encrypted: %02X %02X\n", ciphertext[0], ciphertext[1]);

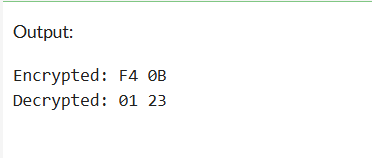
decryptCBC(ciphertext, decrypted, 2, k1, k2, iv);

printf("Decrypted: %02X %02X\n", decrypted[0], decrypted[1]);

return 0;

}

OUTPUT:



23. Write a C program for Encrypt and decrypt in counter mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES. Test data for S-DES using a counter starting at 0000 0000. A binary plaintext of 0000 0001 0000 0010 0000 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 0011 1000 0100 1111 0011 0010. Decryption should work correspondingly.

#include <stdio.h>

#include <stdint.h>

uint8\_t IP[] = {1, 5, 2, 0, 3, 7, 4, 6};

uint8\_t IP\_INV[] = {3, 0, 2, 4, 6, 1, 7, 5};

uint8\_t EP[] = {3, 0, 1, 2, 1, 2, 3, 0};

uint8\_t P4[] = {1, 3, 2, 0};

uint8\_t P10[] = {2, 4, 1, 6, 3, 9, 0, 8, 7, 5};

uint8\_t P8[] = {5, 2, 6, 3, 7, 4, 9, 8};

uint8\_t S0[4][4] = {

{1, 0, 3, 2},

{3, 2, 1, 0},

{0, 2, 1, 3},

{3, 1, 3, 2}

};

uint8\_t S1[4][4] = {

{0, 1, 2, 3},

{2, 0, 1, 3},

{3, 0, 1, 0},

{2, 1, 0, 3}

};

uint8\_t permute(uint8\_t in, uint8\_t\* p, int n) {

uint8\_t out = 0;

for (int i = 0; i < n; i++) {

out <<= 1;

out |= (in >> (7 - p[i])) & 1;

}

return out;

}

uint8\_t leftShift5(uint8\_t in, int shifts) {

return ((in << shifts) | (in >> (5 - shifts))) & 0x1F;

}

void keyGen(uint16\_t key, uint8\_t\* k1, uint8\_t\* k2) {

uint16\_t perm = 0;

for (int i = 0; i < 10; i++) {

perm <<= 1;

perm |= (key >> (9 - P10[i])) & 1;

}

uint8\_t left = (perm >> 5) & 0x1F;

uint8\_t right = perm & 0x1F;

left = leftShift5(left, 1);

right = leftShift5(right, 1);

uint16\_t merged = (left << 5) | right;

\*k1 = 0;

for (int i = 0; i < 8; i++) {

\*k1 <<= 1;

\*k1 |= (merged >> (9 - P8[i])) & 1;

}

left = leftShift5(left, 2);

right = leftShift5(right, 2);

merged = (left << 5) | right;

\*k2 = 0;

for (int i = 0; i < 8; i++) {

\*k2 <<= 1;

\*k2 |= (merged >> (9 - P8[i])) & 1;

}

}

uint8\_t sbox(uint8\_t in, uint8\_t box[4][4]) {

uint8\_t row = ((in & 0x8) >> 2) | (in & 0x1);

uint8\_t col = (in >> 1) & 0x3;

return box[row][col];

}

uint8\_t f(uint8\_t r, uint8\_t sk) {

uint8\_t ep = 0;

for (int i = 0; i < 8; i++) {

ep <<= 1;

ep |= (r >> (3 - EP[i])) & 1;

}

uint8\_t x = ep ^ sk;

uint8\_t left = (x >> 4) & 0xF;

uint8\_t right = x & 0xF;

uint8\_t out = (sbox(left, S0) << 2) | sbox(right, S1);

uint8\_t p4out = 0;

for (int i = 0; i < 4; i++) {

p4out <<= 1;

p4out |= (out >> (3 - P4[i])) & 1;

}

return p4out;

}

uint8\_t fk(uint8\_t in, uint8\_t k1, uint8\_t k2) {

uint8\_t ip = permute(in, IP, 8);

uint8\_t left = ip >> 4;

uint8\_t right = ip & 0xF;

uint8\_t t1 = f(right, k1);

left ^= t1;

uint8\_t swapped = (right << 4) | left;

right = swapped & 0xF;

left = swapped >> 4;

uint8\_t t2 = f(right, k2);

left ^= t2;

uint8\_t preout = (left << 4) | right;

uint8\_t out = permute(preout, IP\_INV, 8);

return out;

}

void ctrMode(uint8\_t\* input, uint8\_t\* output, int n, uint8\_t k1, uint8\_t k2, uint8\_t counterStart) {

for (int i = 0; i < n; i++) {

uint8\_t keystream = fk(counterStart + i, k1, k2);

output[i] = input[i] ^ keystream;

}

}

int main() {

uint8\_t plaintext[3] = {0x01, 0x02, 0x04};

uint8\_t ciphertext[3], decrypted[3];

uint8\_t k1, k2;

uint16\_t key = 0x1FD;

uint8\_t counter = 0x00;

keyGen(key, &k1, &k2);

ctrMode(plaintext, ciphertext, 3, k1, k2, counter);

printf("Encrypted: %02X %02X %02X\n", ciphertext[0], ciphertext[1], ciphertext[2]);

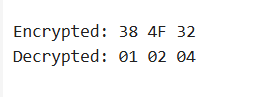
ctrMode(ciphertext, decrypted, 3, k1, k2, counter);

printf("Decrypted: %02X %02X %02X\n", decrypted[0], decrypted[1], decrypted[2]);

return 0;

}

OUTPUT:



24. Write a C program for RSA system, the public key of a given user is e = 31, n = 3599. What is the private key of this user? Hint: First use trial-and-error to determine p and q; then use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).

#include <stdio.h>

int gcdExtended(int a, int b, int\* x, int\* y) {

if (a == 0) {

\*x = 0;

\*y = 1;

return b;

}

int x1, y1;

int gcd = gcdExtended(b % a, a, &x1, &y1);

\*x = y1 - (b / a) \* x1;

\*y = x1;

return gcd;

}

int modInverse(int e, int phi) {

int x, y;

int g = gcdExtended(e, phi, &x, &y);

if (g != 1)

return -1;

else

return (x % phi + phi) % phi;

}

int main() {

int e = 31;

int n = 3599;

int p = 59, q = 61;

int phi = (p - 1) \* (q - 1);

int d = modInverse(e, phi);

printf("Public Key (e, n): (%d, %d)\n", e, n);

printf("p = %d, q = %d\n", p, q);

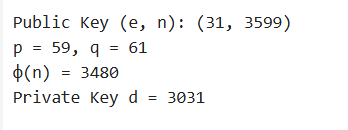
printf("φ(n) = %d\n", phi);

printf("Private Key d = %d\n", d);

return 0;

}

OUTPUT:



25. Write a C program for set of blocks encoded with the RSA algorithm and we don’t have the private key. Assume n = pq, e is the public key. Suppose also someone tells us they know one of the plaintext blocks has a common factor with n. Does this help us in any way?

#include <stdio.h>

int gcd(int a, int b) {

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

int gcdExtended(int a, int b, int\* x, int\* y) {

if (a == 0) {

\*x = 0;

\*y = 1;

return b;

}

int x1, y1;

int gcd = gcdExtended(b % a, a, &x1, &y1);

\*x = y1 - (b / a) \* x1;

\*y = x1;

return gcd;

}

int modInverse(int e, int phi) {

int x, y;

int g = gcdExtended(e, phi, &x, &y);

if (g != 1)

return -1;

else

return (x % phi + phi) % phi;

}

int main() {

int n = 3599;

int e = 31;

int known\_plaintext = 122;

int factor = gcd(known\_plaintext, n);

if (factor == 1 || factor == n) {

printf("No useful factor found.\n");

return 0;

}

int p = factor;

int q = n / p;

printf("Found factors: p = %d, q = %d\n", p, q);

int phi = (p - 1) \* (q - 1);

int d = modInverse(e, phi);

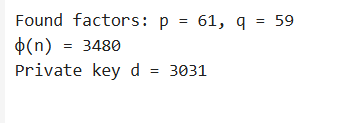
printf("φ(n) = %d\n", phi);

printf("Private key d = %d\n", d);

return 0;

}

OUTPUT:



26. Write a C program for RSA public-key encryption scheme, each user has a public key, e, and a private key, d. Suppose Bob leaks his private key. Rather than generating a new modulus, he decides to generate a new public and a new private key. Is this safe?

#include <stdio.h>

int gcdExtended(int a, int b, int\* x, int\* y) {

if (a == 0) {

\*x = 0;

\*y = 1;

return b;

}

int x1, y1;

int gcd = gcdExtended(b % a, a, &x1, &y1);

\*x = y1 - (b / a) \* x1;

\*y = x1;

return gcd;

}

int modInverse(int e, int phi) {

int x, y;

int g = gcdExtended(e, phi, &x, &y);

if (g != 1) return -1;

return (x % phi + phi) % phi;

}

int main() {

int p = 61, q = 53;

int n = p \* q;

int phi = (p - 1) \* (q - 1);

int e = 17;

int d = modInverse(e, phi);

printf("Original Keys:\n");

printf("Public Key: (e=%d, n=%d)\n", e, n);

printf("Private Key: (d=%d, n=%d)\n", d, n);

int new\_e = 31;

int new\_d = modInverse(new\_e, phi);

printf("\nNew Keys Using Same n:\n");

printf("New Public Key: (e=%d, n=%d)\n", new\_e, n);

printf("New Private Key: (d=%d, n=%d)\n", new\_d, n);

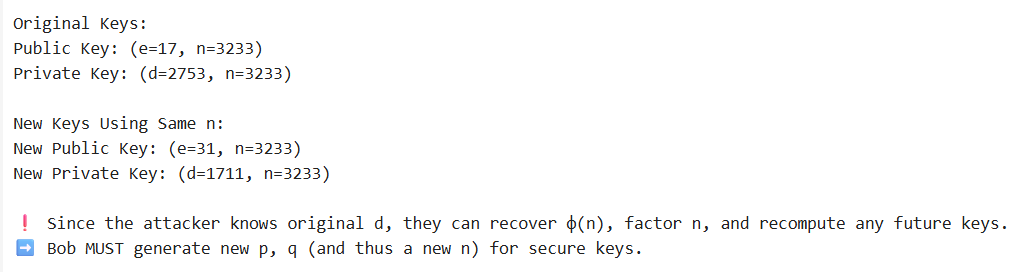
printf("\n❗ Since the attacker knows original d, they can recover φ(n), factor n, and recompute any future keys.\n");

printf("➡️ Bob MUST generate new p, q (and thus a new n) for secure keys.\n");

return 0;

}

OUTPUT:



27. Write a C program for Bob uses the RSA cryptosystem with a very large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends a message to Bob by representing each alphabetic character as an integer between 0 and 25 (A S 0, c, Z S 25) and then encrypting each number separately using RSA with large e and large n. Is this method secure? If not, describe the most efficient attack against this encryption method.

#include <stdio.h>

#include <math.h>

long long modExp(long long base, long long exp, long long mod) {

long long result = 1;

base = base % mod;

while (exp > 0) {

if (exp % 2 == 1) result = (result \* base) % mod;

exp = exp >> 1;

base = (base \* base) % mod;

}

return result;

}

int main() {

long long e = 17;

long long n = 3233;

int message = 2;

long long ciphertext = modExp(message, e, n);

printf("Encrypted 'C' (2): %lld\n", ciphertext);

printf("\nAttacker trying brute-force:\n");

for (int m = 0; m < 26; m++) {

long long test = modExp(m, e, n);

printf("Trying m = %2d → %4lld", m, test);

if (test == ciphertext)

printf(" ← Match! m = %d ('%c')\n", m, 'A' + m);

else

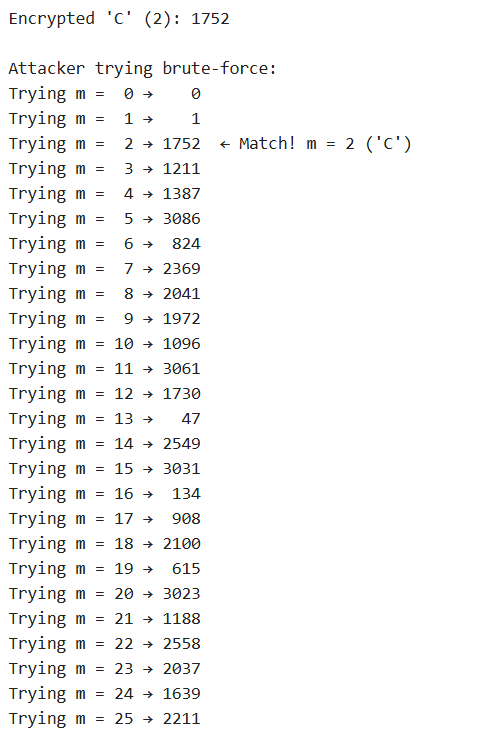
printf("\n");

}

return 0;

}

OUTPUT:



28. Write a C program for Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant ax mod q for some public number a. What would happen if the participants sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the secret numbers?

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int modExp(int base, int exp, int mod) {

int result = 1;

base = base % mod;

while (exp > 0) {

if (exp % 2 == 1)

result = (result \* base) % mod;

exp = exp >> 1;

base = (base \* base) % mod;

}

return result;

}

int main() {

int q = 23;

int a = 5;

int x = 6;

int y = 15;

int A = modExp(a, x, q);

int B = modExp(a, y, q);

int K\_alice = modExp(B, x, q);

int K\_bob = modExp(A, y, q);

printf("Public values: a = %d, q = %d\n", a, q);

printf("Alice sends: %d\n", A);

printf("Bob sends: %d\n", B);

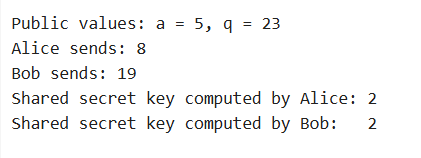
printf("Shared secret key computed by Alice: %d\n", K\_alice);

printf("Shared secret key computed by Bob: %d\n", K\_bob);

return 0;

}

OUTPUT:



29. Write a C program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <stdint.h>

#define TOTAL\_LANES 25

#define RATE\_LANES 16

#define CAPACITY\_LANES 9

int all\_capacity\_filled(uint64\_t\* state) {

for (int i = RATE\_LANES; i < TOTAL\_LANES; i++) {

if (state[i] == 0) return 0;

}

return 1;

}

int main() {

uint64\_t state[TOTAL\_LANES] = {0};

srand(time(NULL));

for (int i = 0; i < RATE\_LANES; i++) {

do {

state[i] = ((uint64\_t)rand() << 32) | rand();

} while (state[i] == 0);

}

for (int i = RATE\_LANES; i < TOTAL\_LANES; i++) {

state[i] = 0;

}

int rounds = 0;

while (!all\_capacity\_filled(state)) {

uint64\_t block[RATE\_LANES];

for (int i = 0; i < RATE\_LANES; i++) {

do {

block[i] = ((uint64\_t)rand() << 32) | rand();

} while (block[i] == 0);

}

for (int i = 0; i < RATE\_LANES; i++) {

state[i] ^= block[i];

}

int cap\_index = RATE\_LANES + rand() % CAPACITY\_LANES;

state[cap\_index] |= (1ULL << (rand() % 64));

rounds++;

}

printf("All capacity lanes filled with nonzero bits after %d rounds.\n", rounds);

return 0;

}

OUTPUT:



30. Write a C program for CBC MAC of a oneblock message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.

#include <stdio.h>

#include <stdint.h>

#include <string.h>

void block\_cipher(uint8\_t \*block, uint8\_t \*key, uint8\_t \*out) {

for (int i = 0; i < 16; i++) {

out[i] = block[i] ^ key[i];

}

}

void cbc\_mac(uint8\_t \*key, uint8\_t \*message, int blocks, uint8\_t \*mac\_out) {

uint8\_t prev[16] = {0};

uint8\_t temp[16];

for (int b = 0; b < blocks; b++) {

for (int i = 0; i < 16; i++)

temp[i] = message[b \* 16 + i] ^ prev[i];

block\_cipher(temp, key, prev);

}

memcpy(mac\_out, prev, 16);

}

void print\_block(const char \*label, uint8\_t \*b) {

printf("%s: ", label);

for (int i = 0; i < 16; i++) printf("%02X ", b[i]);

printf("\n");

}

int main() {

uint8\_t key[16] = {0x0F};

uint8\_t X[16] = {

0x10, 0x11, 0x12, 0x13,

0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1A, 0x1B,

0x1C, 0x1D, 0x1E, 0x1F

};

uint8\_t T[16];

cbc\_mac(key, X, 1, T);

print\_block("CBC-MAC of X", T);

uint8\_t second\_block[16];

for (int i = 0; i < 16; i++)

second\_block[i] = X[i] ^ T[i];

uint8\_t two\_block\_msg[32];

memcpy(two\_block\_msg, X, 16);

memcpy(two\_block\_msg + 16, second\_block, 16);

uint8\_t T2[16];

cbc\_mac(key, two\_block\_msg, 2, T2);

print\_block("CBC-MAC of X || (X⊕T)", T2);

if (memcmp(T, T2, 16) == 0)

printf("\n⚠️ Vulnerability demonstrated: MAC(X) == MAC(X || (X⊕T))\n");

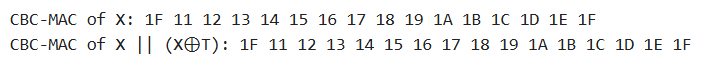
else

printf("\n✅ Secure (but shouldn't be in CBC-MAC without length fix)\n");

return 0;

}

OUTPUT:



31. Write a C program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. a. What constants are needed for block sizes of 64 and 128 bits? b. How the left shift and XOR accomplishes the desired result.

#include <stdio.h>

#include <stdint.h>

#include <string.h>

#define BLOCK\_SIZE 128

#define BYTE\_LEN (BLOCK\_SIZE / 8)

const uint8\_t Rb\_128 = 0x87;

void dummy\_encrypt\_zeros(const uint8\_t\* key, uint8\_t\* out) {

for (int i = 0; i < BYTE\_LEN; i++) {

out[i] = i + 1;

}

}

void left\_shift\_1bit(uint8\_t\* input, uint8\_t\* output) {

uint8\_t carry = 0;

for (int i = BYTE\_LEN - 1; i >= 0; i--) {

output[i] = (input[i] << 1) | carry;

carry = (input[i] & 0x80) ? 1 : 0;

}

}

void xor\_rb(uint8\_t\* block) {

block[BYTE\_LEN - 1] ^= Rb\_128;

}

void print\_block(const char\* label, uint8\_t\* block) {

printf("%s: ", label);

for (int i = 0; i < BYTE\_LEN; i++) {

printf("%02X ", block[i]);

}

printf("\n");

}

int main() {

uint8\_t key[BYTE\_LEN] = {0};

uint8\_t L[BYTE\_LEN];

uint8\_t K1[BYTE\_LEN];

uint8\_t K2[BYTE\_LEN];

dummy\_encrypt\_zeros(key, L);

print\_block("L", L);

left\_shift\_1bit(L, K1);

if (L[0] & 0x80) xor\_rb(K1);

print\_block("K1", K1);

left\_shift\_1bit(K1, K2);

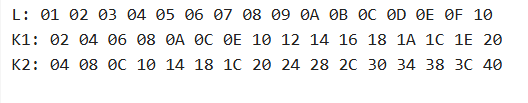
if (K1[0] & 0x80) xor\_rb(K2);

print\_block("K2", K2);

return 0;

}

OUTPUT:



32. Write a C program for DSA, because the value of k is generated for each signature, even if the same message is signed twice on different occasions, the signatures will differ. This is not true of RSA signatures. Write a C program for implication of this difference?

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

unsigned int hash\_message(const char\* msg) {

unsigned int hash = 0;

for (int i = 0; msg[i]; i++)

hash = (hash \* 31 + msg[i]) % 10007;

return hash;

}

unsigned int rsa\_sign(const char\* msg, unsigned int d, unsigned int n) {

unsigned int h = hash\_message(msg);

unsigned int sig = 1;

for (int i = 0; i < d; i++)

sig = (sig \* h) % n;

return sig;

}

unsigned int dsa\_sign(const char\* msg, unsigned int private\_key, unsigned int q) {

unsigned int h = hash\_message(msg);

unsigned int k = rand() % (q - 1) + 1;

unsigned int sig = (h \* k + private\_key) % q;

return sig;

}

int main() {

srand(time(NULL));

const char\* message = "SecureMessage";

unsigned int rsa\_n = 3233;

unsigned int rsa\_d = 17;

unsigned int dsa\_private = 23;

unsigned int dsa\_q = 101;

printf("Message: \"%s\"\n\n", message);

unsigned int rsa\_sig1 = rsa\_sign(message, rsa\_d, rsa\_n);

unsigned int rsa\_sig2 = rsa\_sign(message, rsa\_d, rsa\_n);

printf("RSA Signatures:\n");

printf("Signature 1: %u\n", rsa\_sig1);

printf("Signature 2: %u\n", rsa\_sig2);

printf("✅ RSA produces same signature for same message.\n\n");

unsigned int dsa\_sig1 = dsa\_sign(message, dsa\_private, dsa\_q);

unsigned int dsa\_sig2 = dsa\_sign(message, dsa\_private, dsa\_q);

printf("DSA Signatures:\n");

printf("Signature 1: %u\n", dsa\_sig1);

printf("Signature 2: %u\n", dsa\_sig2);

if (dsa\_sig1 != dsa\_sig2)

printf("✅ DSA produces different signatures due to random nonce k.\n");

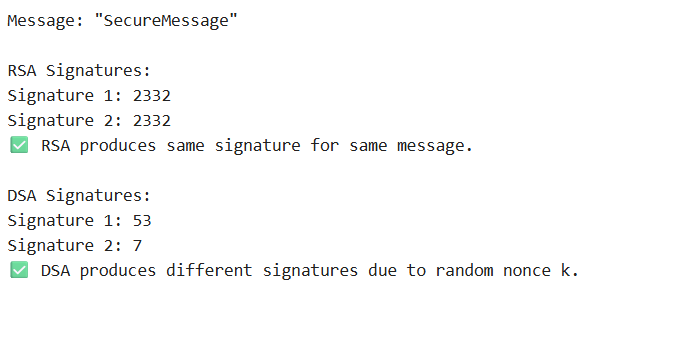
else

printf("⚠️ Unexpected result — signatures matched (rare).\n");

return 0;

}

OUTPUT:



33. Write a C program for Data encryption standard (DES) has been found vulnerable to very powerful attacks and therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of 64 bits each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. Implement in C programming.

#include <stdio.h>

#include <stdint.h>

#include <string.h>

int IP[] = {

58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7

};

int FP[] = {

40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25

};

uint64\_t key = 0x133457799BBCDFF1;

uint32\_t feistel(uint32\_t half, uint64\_t subkey) {

return (half ^ (subkey & 0xFFFFFFFF));

}

uint64\_t permute(uint64\_t input, const int\* table, int n) {

uint64\_t output = 0;

for (int i = 0; i < n; i++) {

output <<= 1;

output |= (input >> (64 - table[i])) & 0x01;

}

return output;

}

uint64\_t des\_process(uint64\_t block, int encrypt) {

uint64\_t permuted = permute(block, IP, 64);

uint32\_t L = (permuted >> 32) & 0xFFFFFFFF;

uint32\_t R = permuted & 0xFFFFFFFF;

for (int round = 0; round < 16; round++) {

uint64\_t subkey = key >> (encrypt ? round : 15 - round);

uint32\_t temp = R;

R = L ^ feistel(R, subkey);

L = temp;

}

uint64\_t preoutput = ((uint64\_t)R << 32) | L;

return permute(preoutput, FP, 64);

}

uint64\_t str\_to\_uint64(const char\* str) {

uint64\_t val = 0;

for (int i = 0; i < 8; i++) {

val = (val << 8) | (unsigned char)str[i];

}

return val;

}

void uint64\_to\_str(uint64\_t val, char\* str) {

for (int i = 7; i >= 0; i--) {

str[i] = val & 0xFF;

val >>= 8;

}

str[8] = '\0';

}

int main() {

char plaintext[9] = "DESDEMO!";

char decrypted[9];

uint64\_t pt = str\_to\_uint64(plaintext);

printf("Plaintext: %s\n", plaintext);

uint64\_t ct = des\_process(pt, 1);

printf("Encrypted (hex): %016llX\n", ct);

uint64\_t dt = des\_process(ct, 0);

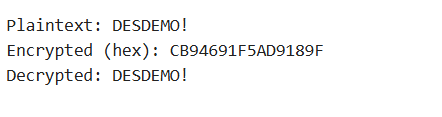
uint64\_to\_str(dt, decrypted);

printf("Decrypted: %s\n", decrypted);

return 0;

}

OUTPUT:



34. Write a C program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed?

#include <stdio.h>

#include <stdint.h>

#include <string.h>

#define BLOCK\_SIZE 8

void xor\_cipher(uint8\_t \*input, uint8\_t \*key, uint8\_t \*output) {

for (int i = 0; i < BLOCK\_SIZE; i++) {

output[i] = input[i] ^ key[i];

}

}

int add\_padding(uint8\_t \*data, int len) {

int pad\_len = BLOCK\_SIZE - (len % BLOCK\_SIZE);

data[len] = 0x80;

for (int i = 1; i < pad\_len; i++)

data[len + i] = 0x00;

return len + pad\_len;

}

void ecb\_encrypt(uint8\_t \*data, int len, uint8\_t \*key, uint8\_t \*output) {

for (int i = 0; i < len; i += BLOCK\_SIZE)

xor\_cipher(data + i, key, output + i);

}

void cbc\_encrypt(uint8\_t \*data, int len, uint8\_t \*key, uint8\_t \*iv, uint8\_t \*output) {

uint8\_t temp[BLOCK\_SIZE];

for (int i = 0; i < len; i += BLOCK\_SIZE) {

for (int j = 0; j < BLOCK\_SIZE; j++)

temp[j] = data[i + j] ^ iv[j];

xor\_cipher(temp, key, output + i);

memcpy(iv, output + i, BLOCK\_SIZE);

}

}

void cfb\_encrypt(uint8\_t \*data, int len, uint8\_t \*key, uint8\_t \*iv, uint8\_t \*output) {

uint8\_t cipher\_out[BLOCK\_SIZE];

for (int i = 0; i < len; i += BLOCK\_SIZE) {

xor\_cipher(iv, key, cipher\_out);

for (int j = 0; j < BLOCK\_SIZE; j++)

output[i + j] = data[i + j] ^ cipher\_out[j];

memcpy(iv, output + i, BLOCK\_SIZE);

}

}

void print\_hex(const char \*label, uint8\_t \*data, int len) {

printf("%s: ", label);

for (int i = 0; i < len; i++)

printf("%02X ", data[i]);

printf("\n");

}

int main() {

uint8\_t key[BLOCK\_SIZE] = {0x1A, 0x2B, 0x3C, 0x4D, 0x5E, 0x6F, 0x70, 0x81};

uint8\_t iv1[BLOCK\_SIZE] = {0x00};

uint8\_t iv2[BLOCK\_SIZE] = {0x00};

uint8\_t iv3[BLOCK\_SIZE] = {0x00};

char message[] = "HELLO BLOCK WORLD!";

uint8\_t data[64];

memset(data, 0, sizeof(data));

memcpy(data, message, strlen(message));

int padded\_len = add\_padding(data, strlen(message));

uint8\_t ecb\_out[64], cbc\_out[64], cfb\_out[64];

memset(ecb\_out, 0, 64);

memset(cbc\_out, 0, 64);

memset(cfb\_out, 0, 64);

ecb\_encrypt(data, padded\_len, key, ecb\_out);

cbc\_encrypt(data, padded\_len, key, iv1, cbc\_out);

cfb\_encrypt(data, padded\_len, key, iv2, cfb\_out);

print\_hex("Plaintext", data, padded\_len);

print\_hex("ECB Cipher", ecb\_out, padded\_len);

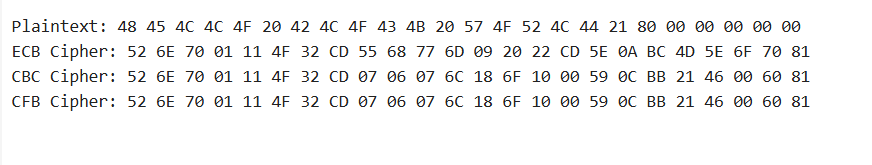
print\_hex("CBC Cipher", cbc\_out, padded\_len);

print\_hex("CFB Cipher", cfb\_out, padded\_len);

return 0;

}

OUTPUT:



35. Write a C program for one-time pad version of the Vigenère cipher. In this scheme, the key is a stream of random numbers between 0 and 26. For example, if the key is 3 19 5 . . . , then the first letter of plaintext is encrypted with a shift of 3 letters, the second with a shift of 19 letters, the third with a shift of 5 letters, and so on.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <ctype.h>

#include <string.h>

#define MAX\_LEN 1024

char encrypt\_char(char c, int key) {

if (isupper(c)) {

return 'A' + (c - 'A' + key) % 26;

} else if (islower(c)) {

return 'a' + (c - 'a' + key) % 26;

}

return c;

}

char decrypt\_char(char c, int key) {

if (isupper(c)) {

return 'A' + (c - 'A' - key + 26) % 26;

} else if (islower(c)) {

return 'a' + (c - 'a' - key + 26) % 26;

}

return c;

}

int main() {

char plaintext[MAX\_LEN];

char ciphertext[MAX\_LEN];

char decrypted[MAX\_LEN];

int key[MAX\_LEN];

printf("Enter plaintext (letters only): ");

fgets(plaintext, sizeof(plaintext), stdin);

plaintext[strcspn(plaintext, "\n")] = '\0';

srand(time(NULL));

int len = strlen(plaintext);

for (int i = 0; i < len; i++) {

if (isalpha(plaintext[i])) {

key[i] = rand() % 27;

ciphertext[i] = encrypt\_char(plaintext[i], key[i]);

} else {

key[i] = 0;

ciphertext[i] = plaintext[i];

}

}

ciphertext[len] = '\0';

for (int i = 0; i < len; i++) {

decrypted[i] = decrypt\_char(ciphertext[i], key[i]);

}

decrypted[len] = '\0';

printf("\nGenerated key: ");

for (int i = 0; i < len; i++) {

if (isalpha(plaintext[i]))

printf("%d ", key[i]);

else

printf("\_ ");

}

printf("\nCiphertext: %s", ciphertext);

printf("\nDecrypted : %s\n", decrypted);

return 0;

}

OUTPUT:

