**DAY-5 Lab programs**

**• 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?**

**C program:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void dh\_encrypt(int x, int a, int q, int \*result) {

\*result = (int) (pow(a, x)) % q;

}

int dh\_decrypt(int x, int a, int q, int \*result) {

\*result = (int) (pow(a, x)) % q;

return 0;

}

int main() {

int a = 3, q = 11;

int x\_alice = 2, x\_bob = 3;

int alice\_to\_bob = 0, bob\_to\_alice = 0;

dh\_encrypt(x\_alice, a, q, &alice\_to\_bob);

dh\_encrypt(x\_bob, a, q, &bob\_to\_alice);

printf("Alice sent: %d\n", alice\_to\_bob);

printf("Bob sent: %d\n", bob\_to\_alice);

return 0;

}

**Output:**

**• 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**

**C program:**

#include <stdio.h>

#include <string.h>

#include <stdint.h>

#define STATE\_SIZE 25

#define CAPACITY 576

#define RATE 1024

#define HASH\_SIZE 256

uint64\_t state[STATE\_SIZE];

const uint64\_t RC[24] = {

0x0000000000000001, 0x0000000000008082, 0x800000000000808A, 0x8000000080008000,

0x000000000000808B, 0x0000000080000001, 0x8000000080008081, 0x8000000000008009,

0x000000000000008A, 0x0000000000000088, 0x0000000080008009, 0x000000008000000A,

0x000000008000808B, 0x8000000000008082, 0x800000000000800A, 0x8000000080000001,

0x8000000080008080, 0x0000000080000001, 0x8000000000008008, 0x8000000080008082,

0x8000000080008000, 0x0000000080008082, 0x8000000000008083, 0x0000000000008082,

0x0000000080008000

};

void theta() {

uint64\_t C[5] = {0};

uint64\_t D[5] = {0};

for (int x = 0; x < 5; x++) {

C[x] = state[x] ^ state[x + 5] ^ state[x + 10] ^ state[x + 15] ^ state[x + 20];

}

for (int x = 0; x < 5; x++) {

D[x] = C[(x + 4) % 5] ^ ((C[(x + 1) % 5] << 1) | (C[(x + 1) % 5] >> 63));

}

for (int x = 0; x < 5; x++) {

for (int y = 0; y < 5; y++) {

state[x + 5 \* y] ^= D[x];

}

}

}

void absorb(const uint8\_t \*message, size\_t message\_length) {

int rate\_bytes = RATE / 8;

for (size\_t i = 0; i < message\_length; i++) {

int x = i % rate\_bytes;

int y = (i / rate\_bytes) % 5;

state[x + 5 \* y] ^= ((uint64\_t)message[i] << (8 \* (i % 8)));

}

if (message\_length % rate\_bytes == 0) {

}

}

void squeeze(uint8\_t \*hash\_output) {

}

int main() {

uint8\_t message[] = "Hello, SHA-3!";

size\_t message\_length = strlen((char\*)message);

memset(state, 0, sizeof(state));

absorb(message, message\_length);

uint8\_t hash\_output[HASH\_SIZE / 8];

squeeze(hash\_output);

printf("SHA-3 Hash: ");

for (int i = 0; i < HASH\_SIZE / 8; i++) {

printf("%02x", hash\_output[i]);

}

printf("\n");

return 0;

}

**Output:**

**• 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.**

**C program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void init\_sbox() {

}

void aes\_cbc\_encrypt(const unsigned char \*in, unsigned char \*out, const unsigned char \*key, const unsigned char \*iv, int len) {

}

void calc\_cbc\_mac(const unsigned char \*in, unsigned char \*mac, const unsigned char \*key, int len) {

}

int main() {

unsigned char X[] = { /\* ... (initial value of the one-block message X) \*/ };

unsigned char K[] = { /\* ... (initial value of the encryption key K) \*/ };

unsigned char T[16];

unsigned char T\_double[16];

unsigned char X\_double[16];

init\_sbox();

aes\_cbc\_encrypt(X, X\_double, K, X, 16);

calc\_cbc\_mac(X, T, K, 16);

calc\_cbc\_mac(X\_double, T\_double, K, 16);

if (memcmp(T, T\_double, 16) == 0) {

printf("The CBC MAC calculation is likely secure.\n");

} else {

printf("The CBC MAC calculation is likely not secure.\n");

}

return 0;

}

**Output:**

**4.** **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.**

**C program:**

#include <stdio.h>

#include <stdint.h>

// Constants for subkey generation

#define BLOCK\_SIZE\_64 8 // 64 bits (8 bytes)

#define BLOCK\_SIZE\_128 16 // 128 bits (16 bytes)

#define CONSTANT\_64 0x1B

#define CONSTANT\_128 0x87

// Left-shift operation

void leftShift(uint8\_t \*data, int size) {

int i;

uint8\_t carry = 0;

for (i = size - 1; i >= 0; i--) {

uint8\_t next\_carry = (data[i] & 0x80) >> 7;

data[i] = (data[i] << 1) | carry;

carry = next\_carry;

}

}

// Derive the subkeys for CMAC

void generateCMACSubkeys(uint8\_t \*key, int block\_size, uint8\_t \*K1, uint8\_t \*K2) {

uint8\_t L[block\_size];

uint8\_t constant = (block\_size == BLOCK\_SIZE\_64) ? CONSTANT\_64 : CONSTANT\_128;

// Calculate L

for (int i = 0; i < block\_size; i++) {

L[i] = key[i];

}

// Calculate K1

leftShift(L, block\_size);

if (key[0] & 0x80) {

L[block\_size - 1] ^= constant;

}

for (int i = 0; i < block\_size; i++) {

K1[i] = L[i];

}

// Calculate K2

leftShift(K1, block\_size);

if (K1[0] & 0x80) {

K1[block\_size - 1] ^= constant;

}

for (int i = 0; i < block\_size; i++) {

K2[i] = K1[i];

}

}

int main() {

uint8\_t key64[BLOCK\_SIZE\_64] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6}; // 64-bit key

uint8\_t key128[BLOCK\_SIZE\_128] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,

0xab, 0xf7, 0x97, 0xd2, 0x98, 0x03, 0x18, 0x09}; // 128-bit key

uint8\_t K1\_64[BLOCK\_SIZE\_64];

uint8\_t K2\_64[BLOCK\_SIZE\_64];

uint8\_t K1\_128[BLOCK\_SIZE\_128];

uint8\_t K2\_128[BLOCK\_SIZE\_128];

generateCMACSubkeys(key64, BLOCK\_SIZE\_64, K1\_64, K2\_64);

generateCMACSubkeys(key128, BLOCK\_SIZE\_128, K1\_128, K2\_128);

printf("Subkeys for 64-bit key:\n");

printf("K1: ");

for (int i = 0; i < BLOCK\_SIZE\_64; i++) {

printf("%02x ", K1\_64[i]);

}

printf("\n");

printf("K2: ");

for (int i = 0; i < BLOCK\_SIZE\_64; i++) {

printf("%02x ", K2\_64[i]);

}

printf("\n");

printf("\nSubkeys for 128-bit key:\n");

printf("K1: ");

for (int i = 0; i < BLOCK\_SIZE\_128; i++) {

printf("%02x ", K1\_128[i]);

}

printf("\n");

printf("K2: ");

for (int i = 0; i < BLOCK\_SIZE\_128; i++) {

printf("%02x ", K2\_128[i]);

}

printf("\n");

return 0;

}

**Output:**

**5.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?**

**C program:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int generate\_key() {

return rand();

}

int generate\_signature(int message, int key) {

return message ^ key;

}

int main() {

srand(time(NULL));

char \*message1 = "Hello, World!";

char \*message2 = "Hello, World!";

char \*initial\_message1 = message1;

char \*initial\_message2 = message2;

int key1 = generate\_key();

int key2 = generate\_key();

int signature1 = generate\_signature(atoi(message1), key1);

int signature2 = generate\_signature(atoi(message2), key2);

if (signature1 != signature2) {

printf("The digital signatures are different, even though the messages are the same.\n");

} else {

printf("The digital signatures are the same, which should not happen.\n");

}

return 0;

}

**Output:**

**6.** **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.**

**C program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define DES\_KEY\_LENGTH 8

#define DES\_BLOCK\_LENGTH 8

typedef struct {

unsigned char key[DES\_KEY\_LENGTH];

} DESKey;

typedef struct {

unsigned char data[DES\_BLOCK\_LENGTH];

} DESDataBlock;

DESKey createDESKey(const char \*key) {

DESKey desKey;

strncpy(desKey.key, key, DES\_KEY\_LENGTH);

return desKey;

}

DESDataBlock encryptDESDataBlock(const DESDataBlock \*dataBlock, const DESKey \*key) {

DESDataBlock encryptedDataBlock;

// DES encryption logic goes here

return encryptedDataBlock;

}

int main() {

DESKey key = createDESKey("example key");

DESDataBlock dataBlock = {{'H', 'e', 'l', 'l', 'o', ',', ' '}};

DESDataBlock encryptedDataBlock = encryptDESDataBlock(&dataBlock, &key);

printf("Encrypted data block: ");

for (int i = 0; i < DES\_BLOCK\_LENGTH; i++) {

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

}

printf("\n");

return 0;

}

**Output:**