Cybersecurity MSc: Applied Cryptography 2023-24

Practical Exercises: Cycles in OFB cipher mode

September 27, 2023 Due date: no date

Changelog

- v1.0 - Initial version.

Introduction

OFB is a cipher mode where we use a block cipher to implement the generator of a stream cipher. In this mode, the keystream results from successive sets of n-bit blocks produced by a block cipher when processing (encrypting or decrypting) an internal state S with a given key K, yielding an output O (which has a length equal or bigger then n).

These n bits are collected from the least significative bits (lsb) of O. These same n bits are then use to modify S as follows: S is shifted left n bits, thus discarding its n lsb, and the n bits of O are placed in the most significative bits (msb) of the resulting S.

The stream cipher operates by executing several iterations, where each produces n bits of keystream. Thus, in iteration i, we have S_{i-1}, we produce O_i from S_{i-1} with a block cipher and K, we pick the n left-most bits of O_i to our keystream, and we use them to produce

$$S_i = (S_i \ll n) + (O_i \gg (B - n))$$

where B represents the length (in bits) of the input and output block of the cipher function (e.g., 128 for AES), $\ll$ a bitwise left shift and $\gg$ a bitwise right shift.

The OFB generator requires the setup of an initial state S_0 that will be used on iteration 1. This initial state is commonly referred as Initial Vector, or Initialization Vector (IV). Usually, the IV varies from message to message when the same key K is used for encrypting all those messages. But it does not need to be secret, and many times it is not.

Usually, it is more practical to work in software with values of n that are a multiple of 8, in order to perform byte-aligned operations. In such cases, we pick up byte blocks and shift left and right bytes, instead of bits.

1 Implementation of a special OFB mode

Implement an n-bit OFB mode with an AES cipher using 128-bit keys. Allow it to work with the following values of n: 8, 16, 24 and 32. You do not need to implement a complete n-bit OFB cipher, just implement the generator (because we will not experiment the encryption or decryption with the OFB cipher).

In order to reduce the space of values that S_i values can take, modify your OFB generator to encrypt S'_i instead of S_i, where S'_i has all the 12 left-most bytes equal to zero, and the 4 right-most bytes equal to the exact same of S_i. The net effect of this transformation is that the input of the AES cipher will only get at most 2^{32} different values. Thus, the internal state of the OFB generator gets reduced to a manageable amount of different values (2^{32}), instead of having 2^{128} different possibilities.

2 Study the cycles on your special OFB mode

You get an m-long cycle the first time you find that $S_{i+m} = S_i$, for the minimum possible i. Therefore, you can study the statistics of m for a given key K.

It is possible, for the same key K, to have different cycles with different m lengths. Try to find, for each key K, how many cycles you may have and their lengths.

To avoid running along dejà vu sequences of S'_i values that end up in the same cycle, keep a log of S'_i values observed in previous experiments (for the same key), which should not be used any more. It takes an amount of 2^{29} bytes, thus 512 megabytes, to implement this log using one single bit to uniquely represent each value.

The next C program finds all the cycles for a random key and a given n.

```c
#include <stdint.h>
#include <openssl/crypto.h>
#include <openssl/evp.h>
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <memory.h>
#include <assert.h>

#define KEY_LEN 16 // bytes

typedef struct {
    uint8_t S[AES_BLOCK_SIZE];
    uint32_t n;
    EVP_CIPHER_CTX * ctx;
} OFB_generator_t;

void
init_ofb_generator( OFB_generator_t * g, uint8_t * key, uint32_t iv )
{
    EVP_CipherInit( g->ctx, EVP_aes_128_ecb(), key, 0, 1 );

    // Set S = iv >> 96

    memset( g->S, 0, sizeof(g->S) - 4 );
    *((uint32_t *) (g->S + 12)) = iv;
}

// Since with only use 2^32 values for S,
// the IV can be an unsigned 32-bt integer

void
new_ofb_generator( OFB_generator_t * g, uint8_t * key, uint32_t iv, uint32_t n )
{
    assert( key != 0 && (n == 8 || n == 16 || n == 24 || n == 32));

    g->ctx = EVP_CIPHER_CTX_new();

    // Store n as the number of bytes
    g->n = n / 8;
```

```
    init_ofb_generator( g, key, iv );
}

void
free_ofb_generator( OFB_generator_t * g )
{
    EVP_CIPHER_CTX_free( g->ctx );
}

uint32_t
iterate_ofb_generator( OFB_generator_t * g )
{
    uint8_t O[AES_BLOCK_SIZE];
    uint32_t o_len = sizeof(O);

    // Set to 0 the 12 left-most bytes of S

    memset( g->S, 0, sizeof(g->S) - 4 );

    // Encrypt S, producing O

    EVP_CipherUpdate( g->ctx, O, &o_len, g->S, sizeof(g->S) );

    assert( o_len == sizeof(O) );

    // Shift S left n bytes

    for (int j = 0; j < AES_BLOCK_SIZE - g->n; j++) {
        g->S[j] = g->S[j + g->n];
    }

    // Place the left-most n bytes of O into the right-most bytes of S

    for (int j = 0; j < g->n; j++) {
        g->S[AES_BLOCK_SIZE - g->n + j] = O[j];
    }

    return *((uint32_t *) (g->S + 12));
}

uint8_t used_ivs[1 << 29];
uint8_t sequence_generated[1 << 29];
uint64_t unused_ivs;
uint8_t cache[255];

uint32_t
first_zero( uint8_t byte )
{
    for (int i = 0; i < 8; i++) {
        if (((1 << i) & byte) == 0) return i;
    }
}

void
setup()
{
    for (int i = 0; i < 255; i++) { // 255 is not necessary because it has no zeros
        cache[i] = first_zero( i );
    }
}

uint32_t last_iv_idx = 0;

void
reset_ivs()
{
    memset( used_ivs, 0, sizeof(used_ivs) );
    unused_ivs = (uint64_t) 1 << 32;
    last_iv_idx = 0;
}

uint32_t
get_unused_iv()
{
    uint32_t ret;
```

```
    for (;; last_iv_idx++) {
        if (used_ivs[last_iv_idx] != 0xFF) {
            ret = last_iv_idx * 8 + cache[used_ivs[last_iv_idx]];
            used_ivs[last_iv_idx] |= 1 << (ret % 8);
            unused_ivs--;
            break;
        }
    }

    return ret;
}

void
find_cycles( uint8_t * key, uint32_t n )
{
    uint32_t iv = 0;
    OFB_generator_t g;
    uint64_t last_percentage = 101;

    new_ofb_generator( &g, key, iv, n );

    for (; unused_ivs != 0;) {
        memset( sequence_generated, 0, sizeof(sequence_generated) );
        iv = get_unused_iv();
        init_ofb_generator( &g, key, iv );
        sequence_generated[iv / 8] |= 1 << (iv % 8);

        for (;;) {
            uint32_t new;

            new = iterate_ofb_generator( &g );

            if (used_ivs[new / 8] & (1 << (new % 8))) { // Already used
                if (sequence_generated[new / 8] & (1 << (new % 8))) { // In current sequence

                    // Calculate cycle length starting on the colliding value
                    init_ofb_generator( &g, key, new );

                    for (int j = 1;; j++) {
                        if (new == iterate_ofb_generator( &g ) ) {
                            printf( "\tcycle, len = %u\n", j );
                            break;
                        }
                    }
                }

                break;
            }

            unused_ivs--;
            used_ivs[new / 8] |= (1 << (new % 8));
            sequence_generated[new / 8] |= (1 << (new % 8));
        }
    }
}

int
main( int argc, char ** argv )
{
    uint8_t key[KEY_LEN];
    uint32_t n;

    if (argc != 2) {
         fprintf( stderr, "usage: %s n\n", argv[0] );
         return 1;
    }

    if (sscanf( argv[1], "%u", &n ) == 0) {
         fprintf( stderr, "n must be 8, 16, 24 or 32\n" );
         return 1;
    }

    if (n != 8 && n != 16 && n != 24 && n != 32) {
         fprintf( stderr, "n = %u is not valid; it must be 8, 16, 24 or 32\n", n );
         return 1;
    }
```

```
    setup();

    for (;;) {
        RAND_bytes( key, sizeof(key) );
        reset_ivs();

        printf ( "New key\n" );

        find_cycles( key, n );
    }
}
```