CAA 24-25

Exercise Sheet on Symmetric Encryption #2
Solutions

1 CTR

1. If the nonce is 118-bit long, we have $128 - 118 = 10$ bits for the counter. This makes 2^{10} blocks which makes $128 \cdot 2^{10} = 131072$ bits.

2. The nonce being very small, we are limited in the number of messages we an encrypt. With a deterministic nonce, we can encrypt 2^8 messages. With a random nonce, this value drops significantly due to the birthday paradox. We can encrypt at most 2^4 messages if we limit our probability of collision to 40%, which is very high. If we encrypt only 4 messages, we have a collision probability of 3%.

3. The block size is 64 bits. The largest transaction is 2^{20} blocks long. Hence, we need to reserve 20 bits for the counter and the rest can be used for the nonce. We have, thus, 44 bits of nonce. Since they send 2^{30} transaction be year, we cannot use a random nonce: the probability of collision is too high (close to 1). We have to use a deterministic nonce.

2 IoT Scenarios

1. The constraint on the size means we cannot have any tag (hence, no AES-GCM nor Chacha20-Poly1305). This is not an issue regarding security since we are considering only passive adversaries. The size constraint means also that we cannot send any IV along with the message. Hence, the IV has to be fixed to a constant (e.g. 0). Let's analyse the confidentiality of the constructions when the IV is fixed. For ECB, we leak equalities in the plaintext blocs. For AES-CTR, we have a known plaintext decryption attack and a leak in the XOR of the plaintexts. For CBC, we leak if plaintexts share the same prefix (i.e., if their first block is equal and if the whole message is equal). Thus, the best solution is AES-CBC with a fixed IV.

2. Since we protect against active adversaries, we have to remove all the non-authenticated modes of operation, i.e., ECB, CBC and CTR. The other two algorithms are in fact similar. We will first have to choose between a deterministic nonce or a random nonce. We will send in total 8760 messages and we can send only 64 bits per hour. Out of these 64 bits, 8 are needed for the ciphertext. The rest has to be used for the nonce and/or for the tag. Since, we have already a 16-bit memory that we can use, this could be a way of storing the current state of a deterministic nonce. This is enough to send $2^{16} > 8760$

messages. We will use the remaining 64-8 bits as a tag (we simply truncate the tag to 56 bits). This reduces the security (forging a correct message has $1/2^{56}$ chances of success) but is the best we can do given the constraints.

3 Authenticated Encryption with Libsodium

1. Libsodium is a perfect library for doing everything in a secure manner. It implements only secure parameters and secure algorithms. It also makes sure to protect against side-channel attacks, manages well the memory, . . . However, its functionalities are a bit limited. If one wants to implement something that is a bit outside of the traditional cryptographic algorithms, they won't find it in Libsodium. Libsodium also pushes a bit its choice of algorithm towards algorithms that were designed or promoted by its authors (Eduard curves, Argon2, Blake2, . . .).

2. By default, libsodium uses XSalsa20 and Poly1305. One can also use AES256-GCM if its processor has special AES instructions, ChaCha20-Poly1305 and XChaCha20-Poly1305.

3. In the combined mode, the tag and the ciphertext are stored together, while in the detached mode, both items will be put in separate places.

4. For this simple implementation, we can use the default authenticated encryption tool:

```c
#include <sodium.h>
#include <string.h>

int main(int argc, char** argv)
{
        if(argc != 2){
                fprintf(stderr, "You have to provide an argument: the message to encrypt\n");
                return -1;
        }
        if (sodium_init() == -1) {
                return 1;
        }
        int message_len = strlen(argv[1])+1; //+1 to keep \0
        int ciphertext_len = message_len + crypto_secretbox_MACBYTES;

        unsigned char key[crypto_secretbox_KEYBYTES];
        unsigned char nonce[crypto_secretbox_NONCEBYTES];
        unsigned char ciphertext[ciphertext_len];

        crypto_secretbox_keygen(key);
        randombytes_buf(nonce, sizeof nonce);
        //encryption
        crypto_secretbox_easy(ciphertext, argv[1],  message_len, nonce, key);

        //decryption
        unsigned char decrypted[message_len];
        if (crypto_secretbox_open_easy(decrypted, ciphertext, ciphertext_len, nonce, key) != 0) {
                fprintf(stderr, "authentication error\n");
        }
        else {
                printf("Decrypted: %s\n",decrypted);
        }

        //flip a bit in ciphertext
        ciphertext[0] ^= 1;
        if (crypto_secretbox_open_easy(decrypted, ciphertext, ciphertext_len, nonce, key) != 0) {
                fprintf(stderr, "authentication error when flipped\n");
        }
        else {
                printf("Decrypted when flipped: %s\n",decrypted);
        }

}
```

5. For this one, we need to specify the algorithms:

```
#include <sodium.h>
#include <string.h>

int main(int argc, char** argv)
{
        if(argc != 3){
                fprintf(stderr, "You have to provide two argument: the message to encrypt and authenticated data\n");
                return -1;
        }
        if (sodium_init() == -1) {
                return 1;
        }
        if(crypto_aead_aes256gcm_is_available() == 0){
                abort(); //No hardware AES accelaration on this CPU
        }
        int message_len = strlen(argv[1])+1; //+1 to keep \0
        int authenticated_len = strlen(argv[2])+1;

        unsigned char key[crypto_aead_aes256gcm_KEYBYTES];
        unsigned char nonce[crypto_aead_aes256gcm_NPUBBYTES];
        unsigned char ciphertext[message_len + crypto_aead_aes256gcm_ABYTES];
        unsigned long long ciphertext_len;

        crypto_aead_aes256gcm_keygen(key);
        randombytes_buf(nonce, sizeof nonce);
        //encryption
        crypto_aead_aes256gcm_encrypt(ciphertext, &ciphertext_len, argv[1], message_len, argv[2], authenticated_len, NULL, nonce, key);

        //decryption
        unsigned char decrypted[message_len];
        unsigned long long decrypted_len;
        //We "transmit" authenticated data
        unsigned char authenticated[authenticated_len];
        strcpy(authenticated, argv[2]);

        if (ciphertext_len < crypto_aead_aes256gcm_ABYTES ||
```

```c
                        crypto_aead_aes256gcm_decrypt(decrypted, &decrypted_len,
                                NULL,
                                ciphertext, ciphertext_len,
                                authenticated, authenticated_len,
                                nonce, key) != 0) {
                fprintf(stderr, "Error while decrypting message\n");
        }
        else {
                printf("Decrypted = %s\n", decrypted);
        }

        //flip a bit in authenticated data
        authenticated[0]^=1;
        if (ciphertext_len < crypto_aead_aes256gcm_ABYTES ||
                        crypto_aead_aes256gcm_decrypt(decrypted, &decrypted_len,
                                NULL,
                                ciphertext, ciphertext_len,
                                authenticated, authenticated_len,
                                nonce, key) != 0) {
                fprintf(stderr, "Error while decrypting modified message\n");
        }
        else {
                printf("Decrypted modified message = %s\n", decrypted);
        }

}
```