Université Libre de Bruxelles 2024-2025
Faculté des Sciences

INFO-F-405: Introduction to cryptography

Free implementation project on symmetric cryptography and hashing

This project starts from the description of the permutation ABCD specified in Exercise 13 of the symmetric-key techniques. It then builds on it by asking you to implement:

- the ABCD permutation itself;
- the sponge construction on top of ABCD;
- an authentication (MAC) scheme using the ABCD sponge;
- an encryption scheme using the ABCD sponge;
- a hash function.

1 The ABCD permutation

Write an implementation of the ABCD permutation. ABCD takes as input a 256-bit state that we represent as 4 rows of 64 bits each. We denote the individual bits in a rows using subscripts, e.g., row a is composed of the bits $a_0, a_1, \ldots, a_{63}$.

As a convention, to be able to write a row in a compact way, we represent it as the integer $\text{int}(a) = \sum_i a_i 2^i$, i.e., the bit with the lowest index is the least significant bit of the integer. We can then write the rows as integers in hexadecimal. For instance, if (a, b, c, d) is all zeroes, except $a_0 = b_5 = c_{63} = 1$, we can represent the state as:

```
a = 0x0000000000000001
b = 0x0000000000000020
c = 0x8000000000000000
d = 0x0000000000000000
```

You can check your implementation with the test vectors provided in the associated text file.

2 The ABCD sponge function

To build something useful with the ABCD permutation, let us implement a sponge function with it (see slides for the specifications of the sponge construction).

$$\text{output} = \text{ABCDsponge}(\text{input})$$

In order to make this as simple as possible, we consider that the sponge function has the following parameters:

- the rate is $r = 64$ bits,
- the capacity is $c = 192$ bits,
- the padding rule is pad10*1,
- the input is a string of bytes.

The first two parameters ensure that the outer part of the state is mapped to the a row of the ABCD, while the inner part is mapped to the b, c and d rows.

In the absorbing phase, the input string is divided into block of 64 bits (or 8 bytes), except the last one that can be shorter. As specificed in the sponge construction, we appy the ABCD permutation after bitwise adding (or XORing) the block into the outer part (i.e., a row). There is a bit of technicality to specify how to map the block of bytes onto the row m that is added to a, that is, $a \leftarrow a \oplus m$.

Tip: It is already a good first step to have an implementation that accepts only inputs that have a length multiple of 8 bytes and/or that provides the blocks directly as values m. If so, you can skip the following technical explanations and come back to them later.

In a nutshell, the first byte of the block is mapped to bits $0 \ldots 7$ of m, then the second byte to bits $8 \ldots 15$, etc.

- If the block is complete, i.e., it has 8 bytes, the row m receives all the bytes up to the 8th byte in bits $56 \ldots 63$. For instance, if we want to absorb `0x30`, `0x31`, ..., `0x37` ("01234567" in ASCII), we get $m =$ `0x3736353433323130`.
  - If this was the last block, we need to create one more block for the padding, where $m_i = 0$ except for $m_0 = 1$ and $m_{63} = 1$. In hexadecimal representation, this means $m =$ `0x8000000000000001`.

- Otherwise, if the block has $n < 8$ bytes, the row receives the n bytes up to bit $8n-1$. This is necessarily the last block. We need to add the padding, which means setting $m_{8n} = 1$ and $m_{63} = 1$. For instance, if $n = 3$ and we want to absorb `0x41`, `0x42`, `0x43` ("ABC" in ASCII) as last block, we get $m =$ `0x8000000001434241`.

Please see in the associated text file for more examples, as well as test vectors to check your implementation.

3 The ABCD message authentication code

As the next step, we build a MAC function on top of the ABCD sponge function. To compute the MAC, we simply use the ABCDsponge where the input is the concatenation of the secret key and of the message to authenticate:

$$\text{MAC} = \text{ABCDMAC}_K(\text{message}) = \lfloor \text{ABCDsponge}(K \| \text{message}) \rfloor_{128}$$

We here assume that the key is 128-bit long, and hence it fits in exactly two input blocks. Also, we assume that the MAC is 128-bit long, so we need exactly two output blocks.

As in the other steps, you can check your implementation with the test vectors provided in the associated text file.

4 The ABCD encryption scheme

For the encryption, we proceed with stream encryption using ABCDsponge as a keystream generator.

Conceptually, we proceed in two steps. First, we generate the keystream with ABCD-sponge taking as input the concatenation of the secret key and the diversifier, assuming it is an integer coded on 64 bits. We produce as many output bits as the plaintext we want to encrypt (or as the ciphertext we want to decrypt):

$$Z = \lfloor \text{ABCDsponge}(K \| \text{diversifier}) \rfloor_{|P|}.$$

Then, we use this keystream to encrypt the plaintext (or to decrypt the ciphertext).

$$C = P \oplus Z \quad \text{or} \quad P = C \oplus Z.$$