Université Libre de Bruxelles
Faculté des Sciences

2022-2023

INFO-F-405: Introduction to cryptography

2. Symmetric-key techniques

Keystream generators and stream ciphers

Exercise 1

What happens if the diversifier (or nonce) is not unique, i.e., if the stream cipher is incorrectly used and that two distinct plaintexts are encrypted with the same key and the same diversifier? What consequences does it have in the case of a known plaintext attack?

Answer of exercise 1

Like for a misused one-time pad, the difference between the two plaintexts is revealed.

In the case of a known plaintext attack, one of the two plaintexts can be known, and if the difference is known, the other one is revealed.

Exercise 2

Let us consider two linear feedback shift registers (LFSRs) of 4 bits. For each, simulate its execution and determine the cycles. What is the period of the longest possible cycle? Which LFSR has it?

1. $1 + D^2 + D^4$: an iteration is $(b_1, b_2, b_3, b_4) \leftarrow (b_2 + b_4, b_1, b_2, b_3)$.
2. $1 + D^3 + D^4$: an iteration is $(b_1, b_2, b_3, b_4) \leftarrow (b_3 + b_4, b_1, b_2, b_3)$.

Answer of exercise 2

For $1+D^2+D^4$, there are two cycles of period 6, one of period 3 and one of period 1.

- Period of 6: `1000` $\rightarrow$ `0100` $\rightarrow$ `1010` $\rightarrow$ `0101` $\rightarrow$ `0010` $\rightarrow$ `0001` and back.

- Period of 6: `1100` $\rightarrow$ `1110` $\rightarrow$ `1111` $\rightarrow$ `0111` $\rightarrow$ `0011` $\rightarrow$ `1001` and back.
- Period of 3: `0110` $\rightarrow$ `1011` $\rightarrow$ `1101` and back.
- And of course `0000` stays on itself.

The longest possible cycle has period $2^4 - 1 = 15$, and $1 + D^3 + D^4$ has it: `1000` $\rightarrow$ `0100` $\rightarrow$ `0010` $\rightarrow$ `1001` $\rightarrow$ `1100` $\rightarrow$ `0110` $\rightarrow$ `1011` $\rightarrow$ `0101` $\rightarrow$ `1010` $\rightarrow$ `1101` $\rightarrow$ `1110` $\rightarrow$ `1111` $\rightarrow$ `0111` $\rightarrow$ `0011` $\rightarrow$ `0001` and back, plus of course `0000` that stays on itself.

Exercise 3

The Mantin-Shamir attack on RC4 shows that the second byte of keystream has a probability of about $\frac{2}{257}$ of taking value 0 and a probability of about $\frac{1}{257}$ of taking each of the other 255 possible values. What is the probability of winning the IND-CPA game when exploiting this property? Give an upper bound on the security strength s of RC4.

Answer of exercise 3

Since the second byte of keystream is biased towards value 0, the second byte of ciphertext has slightly more chance of being equal to the second byte of the plaintext than to another value.

We therefore assume the following strategy for the adversary. The adversary chooses the plaintext m_0 (resp. m_1) such that the second byte has value a (resp. b), with $a \neq b$. When getting the second byte c of the encryption of m_0 or m_1, the adversary says that the plaintext is m_0 (resp. m_1) when $c = a$ (resp. $c = b$), and guesses randomly when $c \notin \{a, b\}$.

$$\begin{aligned}\Pr[\text{win}] = \Pr[\text{challenger encrypts } m_0](&\Pr[c = a \mid \text{challenger encrypts } m_0] \\ &+\frac{1}{2}\Pr[c \notin \{a,b\} \mid \text{challenger encrypts } m_0]) \\ +\Pr[\text{challenger encrypts } m_1](&\Pr[c = b \mid \text{challenger encrypts } m_1] \\ &+\frac{1}{2}\Pr[c \notin \{a,b\} \mid \text{challenger encrypts } m_1]).\end{aligned}$$

The challenger will choose m_0 or m_1 each with probability $\frac{1}{2}$. Therefore, by symmetry, we have

$$\begin{aligned}\Pr[\text{win}] &= \Pr[c = a \mid \text{challenger encrypts } m_0] + \frac{1}{2}\Pr[c \notin \{a,b\} \mid \text{challenger encrypts } m_0] \\ &\approx \frac{2}{257} + \frac{1}{2} \times \frac{254}{257} \\ &= \frac{258}{514} = \frac{1}{2} + \frac{1}{514}.\end{aligned}$$

The advantage is therefore $\epsilon = \frac{1}{514}$ and, considering the negligible amount of computations needed to mount this attack, we set $\log_2(t+d) = 0$ and deduce that RC4 has at most (a little over) 9 bits of IND-CPA security.

Note that the attack does not require to be in the *chosen plaintext* model, a *ciphertext-only* model is sufficient.

Block ciphers

Exercise 4

Complementating a bit string means changing the value of each bit or, equivalently, adding 1 to all bits (modulo 2 of course). Mathematically, if x is a bit string of length n, the complementation of x can be written as :

$$\overline{x} = x \oplus 1^n.$$

Let x and y be two bit strings of length n. Can you simplify the following expressions?

- $\overline{x} + \overline{y}$
- $\overline{x + y}$
- $\overline{(x \| y)}$
- $f(\overline{x})$,

where f is a bit transposition from n bits to n bits. Note that in this case we have $f(1^n) = 1^n$.

Answer of exercise 4

Using the definition :

- $\overline{x} \oplus \overline{y} = x \oplus 1^n \oplus y \oplus 1^n$
  $= x \oplus y \oplus (1^n \oplus 1^n)$
  $= x \oplus y$

- $\overline{x \oplus y} = x \oplus y \oplus 1^n$
  $= \overline{x} \oplus y \quad \text{or} \quad x \oplus \overline{y}$

- $\overline{(x\|y)} = x\|y \oplus 1^{2n}$ (as $x\|y$ is $2n$-bit long)
  $= x\|y \oplus 1^n\|1^n$
  $= (x \oplus 1^n)\|(y \oplus 1^n)$
  $= \overline{x}\|\overline{y}$

- $f(\overline{x}) = f(x \oplus 1^n)$
  $= f(x) \oplus f(1^n)$ (since f is linear)
  $= f(x) \oplus 1^n$ (since f is a bit transposition)
  $= \overline{f(x)}$

Exercise 5

Complementating a bit string means changing the value of each bit, or equivalently, adding 1 to all bits (modulo 2 of course). DES has a non-ideal property: If the input block and the key are complemented, then the output block is complemented as well. In other words, if $\text{DES}_k(x) = y$, then $\text{DES}_{\overline{k}}(\overline{x}) = \overline{y}$, with the overline to indicate complementation.

The goal of this exercise is to describe what happens inside the DES when input block and the key are complemented. (And by "what happens", it is meant "how do the processed values change when computing $\text{DES}_{\overline{k}}(\overline{x})$ instead of $\text{DES}_k(x) = y$".)

1. If the key is complemented, what happens to the subkeys?
2. If the input block is complemented, what happens to the left and right parts at the beginning of the Feistel network?
3. Consider the first round. The right part R enters the f function and goes through the expansion E. What happens to the output of E?
4. Still inside the f function, what happens when $E(R)$ is XORed with the subkey K?
5. What happens at to the left part L after being XORed with $f(R)$?
6. What happens in the remaining rounds and to the output?

7. Does this complementation property involve a particular property of the S-boxes?

Answer of exercise 5

1. If the key is complemented, all the subkeys are complemented. This is because the subkeys are obtained by a bit-transposition of the key, so this preserves the complementation.
2. When the input block is complemented, the left and right halves L and R are also complemented.
3. When R enters the f function, it is complemented, and it remains complemented after the expansion E.
4. When $E(R)$ and the subkey K are XORed together, both complementations cancel each other. To see this, let's look at the bit level, and say we compute $\overline{x} + \overline{y}$. Then, $\overline{x} + \overline{y} = (x + 1) + (y + 1) = x + y$ in $\mathbb{Z}_2$. So, the input of the S-boxes is *not* complemented anymore and therefore the output of the f function has the same value as before R and K were complemented.
5. The output of the f function is then XORed to the left half L, which is complented and remains so, since $\overline{x} + y = x + 1 + y = \overline{x + y}$.
6. The output of the round is therefore complemented, and the same reasoning can be extended to the 16 rounds of DES.
7. This property does not involve a particular property of the S-box, as the input of the S-box is unchanged after the complementation.

Exercise 6

During each round of AES, the state (consisting of a 4×4 matrix of elements of GF(256), represented as bytes) undergoes the function MixColumns, which is meant to produce diffusion among the bytes of the state. For each column of the state, this function consists in a simple multiplication between the column and a fixed matrix $\mathbf{M}$. If $(s_0, s_1, s_2, s_3)^\top$ is the input of the function (i.e., a column of the state), we get the output $(b_0, b_1, b_2, b_3)^\top$:

$$\begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

What is the output of MixColumns for the column input $s = (\texttt{37}, \texttt{21}, \texttt{A5}, \texttt{C0})^\top$?

Answer of exercise 6

We can compute every byte b_i one by one. For b_0, the matrix multiplication yields

$$b_0 = (\texttt{02} \times \texttt{37}) + (\texttt{03} \times \texttt{21}) + (\texttt{01} \times \texttt{A5}) + (\texttt{01} \times \texttt{C0})$$

Recall that the hexadecimal notation is a compact way to express polynomials in GF(256). In this notation, an addition over GF(256) corresponds to a bitwise XOR over the bytes.

Before we do the multiplications, we have to translate the bytes into polynomials. The multiplication of the polynomials is done modulo $X^8 + X^4 + X^3 + X + 1$. If we wish to write the result in the hexadecimal notation, we have to translate back the resulting polynomial into the hexadecimal notation.

- $\texttt{02} \times \texttt{37} = 0000\,0010 \times 0011\,0111$
  $= X \times (X^5 + X^4 + X^2 + X + 1)$
  $= X^6 + X^5 + X^3 + X^2 + X$
  $= 0110\,1110 = \texttt{6E}$

Note that multiplying by $\texttt{02}$ does nothing but shifting the bits of $\texttt{37}$ one position towards the most signifiant bits (i.e., to the "left") because there is no X^7 term in $\texttt{37}$. (If there was a X^7 term, it would become X^8 after multiplication with X, hence an explicit modular reduction would be required to reduce the degree of the polynomial.)

- $\texttt{03} \times \texttt{21} = 0000\,0011 \times 0010\,0001$
  $= (X + 1) \times (X^5 + 1)$
  $= X^6 + X^5 + X + 1$
  $= 0110\,0011 = \texttt{63}$

Since $\texttt{03}$ equals $\texttt{01} + \texttt{02}$, we could have obtained the product by simply XORing the bitstring corresponding to $\texttt{21}$ with the same bitstring shifted by one position to the "left" (again, here because there is no X^8 or higher term that appears). We would indeed get

$$0010\,0001 \oplus 0100\,0010 = 0110\,0011$$

For the last two multiplications, since multiplying by $\texttt{01}$ is the identity, we immediately get

- $\texttt{01} \times \texttt{A5} = \texttt{A5} = 1010\,0101$
- $\texttt{01} \times \texttt{C0} = \texttt{C0} = 1100\,0000$.

Now that we have all four product, we can add them together (*i.e.* XOR them) to obtain b_0 :

$$\begin{aligned} b_0 &= 0110\,1110 \oplus 0110\,0011 \oplus 1010\,0101 \oplus 1100\,0000 \\ &= 0110\,1000 \\ &= \texttt{68} \end{aligned}$$

We now have the first coefficient of the output. All is left to do is perform the same operations for b_1, b_2 and b_3. For b_1, we would have to compute

$$b_1 = (\texttt{01} \times \texttt{37}) + (\texttt{02} \times \texttt{21}) + (\texttt{03} \times \texttt{A5}) + (\texttt{01} \times \texttt{C0})$$

After all the required computations, one would finally obtain

$$\begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix} \begin{pmatrix} \texttt{37} \\ \texttt{21} \\ \texttt{A5} \\ \texttt{C0} \end{pmatrix} = \begin{pmatrix} \texttt{68} \\ \texttt{41} \\ \texttt{1C} \\ \texttt{46} \end{pmatrix}$$

Exercise 7

Let $\mathbf{M}$ be the matrix of MixColumns. So a column of 4 bytes $(s_0, s_1, s_2, s_3)^\top$ is mapped to $(b_0, b_1, b_2, b_3)^\top = \mathbf{M}(s_0, s_1, s_2, s_3)^\top$ after MixColumns.

Because it is a linear operation, we have that $(b_0, b_1, b_2, b_3)^\top = \mathbf{M}(s_0, 0, 0, 0)^\top + \mathbf{M}(0, s_1, 0, 0)^\top + \mathbf{M}(0, 0, s_2, 0)^\top + \mathbf{M}(0, 0, 0, s_3)^\top$.

1. Using the property above, propose a way to implement MixColumns using only look-up's in precomputed tables and XORs. How many tables do you need? What is the size of each table? What do they contain?

2. Can you extend the reasoning so as to implement MixColumns $\circ$ SubBytes in a similar way? *Hint:* SubBytes processes each byte individually.

3. Can you find a variant with just one table? *Hint:* The matrix $\mathbf{M}$ has a special form.

Answer of exercise 7

1. We build four tables of 256 entries each, and each entry contains a 32-bit value. In the first table, we store $\mathbf{M}(s_0, 0, 0, 0)^\top$ for each value of s_0. In the second

table, we store $\mathbf{M}(0, s_1, 0, 0)^\top$ for each value of s_1, and similarly for the third and fourth tables.

To compute $\mathbf{M}(s_0, s_1, s_2, s_3)^\top$, we look up entry $\#s_0$ in the first table, entry $\#s_1$ in the second table, entry $\#s_2$ in the third table and entry $\#s_3$ in the fourth table, then we XOR all the results together.

2. Let $\mathsf{S}(x)$ be SubBytes applied to an individual byte x. What we need to compute now is $(b_0, b_1, b_2, b_3)^\top = \mathbf{M}(\mathsf{S}(s_0), \mathsf{S}(s_1), \mathsf{S}(s_2), \mathsf{S}(s_3))^\top$. The idea is the same as above, but now in the first table, we store $\mathbf{M}(\mathsf{S}(s_0), 0, 0, 0)^\top$ for each value of s_0, and similarly for the other tables.

3. Notice that $\mathbf{M}$ is a circulant matrix. So the entries in the second (third, fourth, resp.) table are equal to those in the first table after a shift by one (two, three, resp.) position(s). Hence, we keep only the first table and replace the look-up's in the other tables by a look-up in the first table followed by a shift.

Exercise 8

The Electronic Codebook (ECB) mode is a flawed encryption mode on top of a block cipher. What fundamental property makes it an inherently insecure mode?

How could an adversary easily win the IND-CPA game with just a few queries, even if he is not allowed to make queries with identical plaintext values? Detail the queries made by the adversary and the choice of (m_0, m_1).

Answer of exercise 8

The ECB mode is deterministic and its definition does not include a diversifier. Hence, fundamentally, ECB is not semantically or IND-CPA secure for this very reason.

Even if the adversary is restricted in that it cannot submit identical plaintexts, ECB has the property that identical blocks always encrypt to identical blocks. We exploit this property to win the IND-CPA game. Here is one possible way: The attacker chooses two arbitrary distinct strings A and B whose length is equal to the block size of the underlying block cipher. It queries the encryption of $A\|B$ and receives the ciphertext $X\|Y$, where X and Y are blocks. Then, it submits $m_0 = A$ and $m_1 = B$ to the challenger. If $c = X$, it means m_0 was encrypted; otherwise $c = Y$ and m_1 was encrypted. The adversary always wins.

Exercise 9

AES with CTR mode Suppose that a 160 byte message m was encrypted with AES-128 in a counter mode in order to obtain the ciphertext c. Let's suppose that the 10th byte of c was corrupted during a transmission.

a. Would the receiver be able to detect which byte was corrupted?

b. If we decrypt c, how many bytes of m would be affected by the error?

c. Does this number depend on the position where the error occurred?

Answer of exercise 9

a. No, the CTR mode does not provide authentication, nor any form of integrity protection, even non-cryptographic. So the receiver does not have any way to detect any corruption in the ciphertext.

b. 1 byte of 1 block.

c. No.

Exercise 10

AES with CBC mode Suppose that a 160 bytes message m was encrypted with AES-128 in CBC mode in order to obtain the ciphertext c. The block size is 16 bytes. Let's suppose that the 10th byte of c was corrupted during a transmission.

a. Would the receiver be able to detect which byte was corrupted?

b. If we decrypt c, how many bytes of m would be affected by the error?

c. Does this number depend on the position where the error occurred?

Answer of exercise 10

a. No, the CBC mode does not provide authentication, nor any form of integrity protection, even non-cryptographic. So the receiver does not have any way to detect any corruption in the ciphertext.

b. Two blocks would be affected by this modification: the 16 bytes of one block and only one byte of the next block, so 17 bytes in total.

c. Yes. If a byte of the last block is corrupted, then only the last block is affected.

Exercise 11

Consider the following encryption mode called DXCTR for "Diversifier-Xored CounTeR". It is a block cipher-based mode very similar to the original CTR, but the diversifier is XORed into the output of the block cipher instead of being part of its input. More precisely, DXCTR works as in Algorithm 1 and is illustrated in Figure 1. However, DXCTR is flawed.

1. What is/are intuitively the security problem(s) of DXCTR?
2. Describe how an adversary can win the IND-CPA game against DXCTR with only practical data and time complexities.
3. Why the original CTR mode does not have this problem?

Algorithm 1 The DXCTR encryption mode on top of block cipher E with block size n bits and key size m bits.

Input: secret key $K \in \mathbb{Z}_2^m$, plaintext $p \in \mathbb{Z}_2^*$ and diversifier $d \in \mathbb{N}$
Output: ciphertext $c \in \mathbb{Z}_2^{|p|}$

Cut p into blocks of n bits $(p_0, p_1, p_2, \ldots, p_l)$, except for the last one that can be shorter
Generate the keystream, for $i = 0$ to l

$$k_i = E_K(\text{block}(i)) \oplus \text{block}(d)$$

(here block(x) encodes the integer x into a string in $\mathbb{Z}_2^n$)
Truncate the last keystream block k_l to the size of p_l
Compute the ciphertext $c_i = k_i \oplus p_i$, for $i = 0$ to l

---

Answer of exercise 11

1. The problem here is that the diversifier does not diversify well enough the keystream. Since the diversifier is public and is only XORed, anyone can compute $c_i \oplus \text{block}(d) = E_K(\text{block}(i)) \oplus p_i$, which does not depend on the diversifier anymore. When the plaintext is known, this yields the sequence $E_K(\text{block}(i))$ that can then be used to decrypt other messages. When the plaintext is unknown, one can recover the difference between two different messages, say p

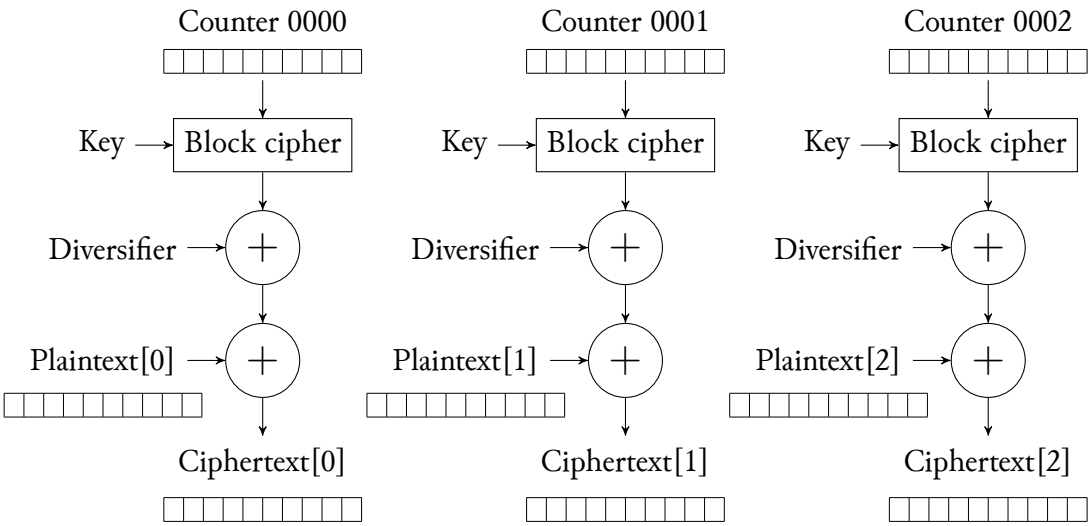

Figure 1: The DXCTR encryption mode.

and p', encrypted under different diversifiers d and d', respectively, by computing

$$c_i \oplus \text{block}(d) \oplus c'_i \oplus \text{block}(d') = p_i \oplus p'_i \ .$$

2. There are several ways to win the IND-CPA game, but the simplest one is probably to first determine a short sequence of $E_K(\text{block}(i))$ as explained above, then use it to decrypt the ciphertext returned by the challenger. For simplicity, we assume that the diversifier is a counter that is incremented at each encryption.

   (a) The challenger generates a secret key K, obviously unknown to the adversary.

   (b) The adversary queries $\text{DXCTR}_K(p, d)$ with $d = 1$ for an arbitrarily chosen plaintext p, and receives the corresponding ciphertext c. It then computes $c_i \oplus \text{block}(d) \oplus p_i = E_K(\text{block}(i))$.

   (c) The adversary arbitrarily chooses two different plaintexts p_0 and p_1 of the same length (as required by the game) that he sends to the challenger.

   (d) The challenger randomly chooses $b \in \{0, 1\}$, encrypts p_b with $d = 2$ and returns $c = \text{DXCTR}_K(p_b, d)$ the adversary.

   (e) The adversary deciphers c using the values of $E_K(\text{block}(i))$ computed at step 2 and determines whether p_0 or p_1 was chosen by the challenger. The adversary always wins despite the small number (1) of chosen-plaintext queries and the very limited amount of computations needed.

3. In CTR, the diversifier is used as input to the block cipher. The diversifier then has an unpredictable influence (for someone not knowing the key) on the keystream.

Exercise 12

Consider SHAKE128, a standard sponge function with permutation f, capacity $c = 256$ bits and rate $r = 1344$ bits. Let us assume that it is used in an application that uses it for the authentication of small messages.

In more details, the application computes a 128-bit MAC on a message M under the secret key K as SHAKE128$(K\|M)$. The secret key is 128-bit long and M is sufficiently short so that $K\|M$ fits in a single bock of r bits, even after padding[1].

1. Write symbolically, i.e., as a mathematical expression, the MAC as a function of the key and of the message for the restricted use case of this application.
2. Knowing that f and its inverse f^{-1} can both be evaluated efficiently, can the secret key be recovered from the value of the MAC? If not, please justify briefly. If so, please describe how.
3. Same question if we extend the MAC length to 1600 bits.
4. Same question if we extend the MAC length to 1600 bits and set $c = 0$, $r = 1600$.

Answer of exercise 12

1. After padding, the first (and only) block of input is $K\|M\|1\|0^{r-|M|-|K|-1}$. The number of zeroes is such that the block is r-bit long. After adding the block to the outer part, the state is $K\|M\|1\|0^{r-|M|-|K|-1+c}$, i.e., the block followed by c additional zero bits at the end. After applying the permutation f, the state is now $f(K\|M\|1\|0^{r-|M|-|K|-1+c})$.

   So the MAC is $f(K\|M\|1\|0^{r-|M|-|K|-1+c})$ truncated to the first 128 bits.

2. No, the key cannot be recovered.

   Even if we can compute f^{-1} efficiently, the MAC value reveals only the first 128 bits of $f(K\|M\|1\|0^{r-|M|-|K|-1+c})$. Exhaustively searching through the $1600 - 128 = 1472$ remaining bits would take much more time than doing an exhaustive key search.

[1] For simplicity, we assume that the input $K\|M$ is simply padded as $K\|M\|10^*$, i.e., with a single bit 1 followed by the minimum number of bits 0 such that the padded message is r-bit long.

3. No, the key cannot be recovered either.

   Remember that the output is squeezed out in blocks of $r = 1344$ bits. If we extend the MAC length to 1600 bits, the output will be given in two iterations, first 1344 bits, then the remaining 256 bits after an iteration of f. The first MAC block reveals the first 1344 bits of $f(K\|M\|1\|0^{r-|M|-|K|-1+c})$, but again exhaustively searching through the $1600 - 1344 = 256$ remaining bits would take much more time than doing an exhaustive key search.

4. Yes, in this case, the key can be recovered.

   Now the MAC is given in one iteration of $r = 1600$ bits, revealing the entire state, i.e., the entire value of $f(K\|M\|1\|0^{r-|M|-|K|-1+c})$. Computing f^{-1} on it and extracting the first 128 bits yields K.

   This highlights that a (keyed) sponge function with $c = 0$ has no security.

Design of symmetric primitives

Exercise 13

Design of symmetric crypto primitives Below are the specifications of a cryptographic permutation under development, called ABCD. We first give the specifications of the permutation, then ask you some questions about it.

ABCD is a 256-bit permutation. The authors lacked inspirations and gave it its name because the 256-bit input block and running state are represented as 4 words (or rows) of 64 bits each, denoted a, b, c and d. We denote the individual bits in a rows using subscripts, e.g., row a is composed of the bits $a_0, a_1, \ldots, a_{63}$. The four bits (a_i, b_i, c_i, d_i) at the same position in each row is called a column. In other words, the state can be viewed as a grid of 4 rows by 64 columns.

The evaluation of the permutation consists in 13 rounds. The evaluation of permutation goes as follows:

for each round $r = 0$ to 12 **do**
  ShakeColumns
  PreShiftRows
  StirColumns
  PostShiftRows
  AddRoundConstant(r)

We now describe the five step mappings AddRoundConstant, ShakeColumns, PreShiftRows, StirColumns and PostShiftRows, all specified at the bit level, or more formally, in the Galois field $\mathrm{GF}(2) = \{0, 1\}$. In this field, $x + y$ (resp. xy) denotes the modulo-2 addition (resp. multiplication) of $x, y \in \mathrm{GF}(2)$. We use the operator $\oplus$ to express the component-wise addition of vectors of elements in $\mathrm{GF}(2)$, such as rows, columns or states.

Definition of AddRoundConstant(r)

for each column $i = 0$ to 63 **do**
  if $i \leq r$ **then**
    $a_i \leftarrow a_i + 1$

Definition of ShakeColumns

for each column $i = 0$ to 63 **do**
  $p \leftarrow a_i + b_i + c_i + d_i$
  $a_i \leftarrow a_i + p$
  $b_i \leftarrow b_i + p$
  $c_i \leftarrow c_i + p$
  $d_i \leftarrow d_i + p$

Definition of PreShiftRows

$a \leftarrow a$
$b \leftarrow \mathrm{rot}^1(b)$
$c \leftarrow \mathrm{rot}^3(c)$
$d \leftarrow \mathrm{rot}^9(d)$

For a row x, $\mathrm{rot}(x)$ denotes the operation that consists in cyclically shifting all the bits by one position, i.e.,

$$(x_0, x_1, x_2, \ldots, x_{63}) \rightarrow (x_1, x_2, \ldots, x_{63}, x_0).$$

Definition of StirColumns

for each column $i = 0$ to 63 **do**
  $a_i \leftarrow a_i + b_i c_i$
  $b_i \leftarrow b_i + c_i d_i$
  $c_i \leftarrow c_i + d_i a_i$
  $d_i \leftarrow d_i + a_i b_i$

Definition of PostShiftRows

$a \leftarrow a$

$b \leftarrow \text{rot}^1(b)$
$c \leftarrow \text{rot}^5(c)$
$d \leftarrow \text{rot}^{25}(d)$

A: Classify each step mapping as linear, affine or non-linear. For affine or linear mappings, please provide a short justification (one line). For non-linear mappings, please provide an example of input pair that violates the definition of affinity.

B: Point out the step mapping(s) that provide diffusion, with a short justification.

C: For the step mapping ShakeColumns, please explain what happens at the output when:

- one flips 1 bit at the input;
- one flips 2 bits of the same column at the input;
- one flips 2 bits of different columns at the input.

D: Write the specifications of the inverse of ABCD.

E: For each the five step mappings AddRoundConstant, ShakeColumns, PreShiftRows, StirColumns and PostShiftRows, would you characterize it as *bend*, *mix*, *notch* or *shuffle*?

F: Without AddRoundConstant, ABCD would satisfy an (undesired) symmetry property. Which one is it? By symmetry property, we are looking for an invertible operation S on 256-bit strings that commutes with ABCD, i.e., S is such that $\forall x \in \{0,1\}^{256}$, we have $\text{ABCD}(S(x)) = S(\text{ABCD}(x))$.

Answer of exercise 13

A:

- AddRoundConstant is affine. It is made of additions with a constant.
- ShakeColumns is linear. Each bit is the sum of input bits, and there are no additions with a constant (i.e., $\text{ShakeColumns}(0) = 0$).

- PreShiftRows and PostShiftRows are linear. These are bit transpositions, hence linear.
- StirColumns is non-linear. The use of multiplication gives a hint. Since StirColumns consists in the application of 64 identical mappings on each column, we can give a counterexample on a single column. Using Table 1, we see that

$$\begin{aligned}\mathsf{StirColumn}(0001) \oplus \mathsf{StirColumn}(0010) &= 0011 \\ &\neq 0111 = \mathsf{StirColumn}(0001 \oplus 0010).\end{aligned}$$

B: Only ShakeColumns and StirColumns provide diffusion. Clearly, each input bit influences more than one output bit.

PreShiftRows and PostShiftRows are bit transpositions, hence each input bit only influences exactly one output bit. In AddRoundConstant too, each input bit only influences exactly one output bit.

C:

- When one flips 1 bit at the input, in that column, the three other bits of the same column flip at the output. Let us assume that a_i is flipped at the input. Then, the value of p flips, and b_i, c_i and d_i flip at the output. The output a_i does not flip, as $a_i \leftarrow (a_i + 1) + (p + 1) = a_i + p$. (The reasoning is similar when b_i, c_i or d_i is flipped at the input.)
- When one flips 2 bits of the same column at the input, in that column, only these two bits flip at the output. When two bits are flipped in the same column, the value of p is unchanged since $1 + 1 = 0$. Hence, only the same two bits flip at the output.
- When one flips 2 bits of different columns at the input, the three other bits of these two columns flip at the output. The reasoning is the same as that of the first bullet, but on two columns in parallel, since ShakeColumns works on columns independently.

D: Each operation is inverted, starting from the last one. One can notice that AddRoundConstant(r) is clearly self-inverse. ShakeColumns is self-inverse too, as detailed below. Hence, the inverse of ABCD works like this:

for each round $r = 12$ down to 0 **do**

AddRoundConstant(r)
InvPostShiftRows
InvStirColumns
InvPreShiftRows
ShakeColumns

To see that ShakeColumns is self-inverse, notice that p, the parity of a column, is unchanged by the application of ShakeColumns. Hence, to undo the effect of ShakeColumns, it is sufficient to add again the parity of each column.

The inverses of PostShiftRows and PreShiftRows are obtained by applying the cyclic shift in the other direction.

Notice that StirColumns works as a Feistel network. To undo it, it suffices to run it backwards:

Definition of InvStirColumns
for each column $i = 0$ to 63 **do**
$d_i \leftarrow d_i + a_i b_i$
$c_i \leftarrow c_i + d_i a_i$
$b_i \leftarrow b_i + c_i d_i$
$a_i \leftarrow a_i + b_i c_i$

E:

- *bend*: StirColumns provides non-linearity over a column;
- *mix*: ShakeColumns provides good diffusion over a column;
- *notch*: AddRoundConstant breaks the symmetry of translation along the rows (see below);
- *shuffle*: PreShiftRows and PostShiftRows are bit transpositions that ensure that the four bits of a column are moved to different columns before the next StirColumns or ShakeColumns. Hence they ensure the global diffusion and the global spread of non-linearity as more rounds are applied.

F: Without AddRoundConstant, all the operations are invariant under a cyclic translation of all the rows by the same amount. Hence, without AddRoundConstant, the permutation ABCD would have this symmetry property, i.e., shifting the whole input along the rows would just shift the whole output.

More formally, an invertible operation S that would commute with ABCD would be:

$$S(a, b, c, d) = (\mathrm{rot}(a), \mathrm{rot}(b), \mathrm{rot}(c), \mathrm{rot}(d)).$$

With AddRoundConstant, if one shifts the whole input of ABCD along the rows, we can expect that the output will change in a complicated way.

$a_i b_i c_i d_i$	$a'_i b'_i c'_i d'_i = \mathsf{StirColumn}(a_i b_i c_i d_i)$
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 0	0 0 1 0
0 0 1 1	0 1 1 1
0 1 0 0	0 1 0 0
0 1 0 1	0 1 0 1
0 1 1 0	1 1 1 1
0 1 1 1	1 0 0 1
1 0 0 0	1 0 0 0
1 0 0 1	1 0 1 1
1 0 1 0	1 0 1 0
1 0 1 1	1 1 0 0
1 1 0 0	1 1 0 1
1 1 0 1	1 1 1 0
1 1 1 0	0 1 1 0
1 1 1 1	0 0 1 1

Table 1: Truth table of StirColumns applied to a single column.