Implementation of Kasumi

Letizia D'Achille

Final individual report of the team project for the exam
Algebraic Cryptography: mod. 1 - Cryptography

December 30, 2022

Notation

$L \,\|\, R$	Concatenation of the two sequences (or strings) L and R
$L \oplus R$	Bitwise xor between the two sequences L and R
$L \cap R$	Bitwise and between the two sequences L and R
$L \cup R$	Bitwise or between the two sequences L and R
$\mathrm{Rot}(L, n)$	Left rotation of the sequence L by n bits

Overview

Kasumi is a block cipher with 128-bit key that takes as input a 64-bit plaintext, and returns as output a 64-bit ciphertext. In particular, Kasumi is an eight round Feistel cipher, so that the encryption function can be written as follows:

$$e_k = \overline{\varepsilon_{1,k}} \dots \overline{\varepsilon_{8,k}}$$
$$\overline{\varepsilon_{i,k}} = \begin{pmatrix} F_i(\_, k_i) & \mathbb{1}_{32} \\ \mathbb{1}_{32} & \mathbb{0}_{32} \end{pmatrix} \quad \forall\, i = 1, \dots, 8$$

where $k_1, \dots, k_8$ are the round keys derived by the input key using the key schedule. Equivalently, the algorithm proceeds as follows:

Kasumi Encryption

Input I - 64 bits, K - 128 bits
Output O - 64 bits

Compute $[k_1, \dots, k_8] = \mathrm{KeySchedule}(K)$;
Split I in two 32-bit sequences L_0 and R_0 such that $I = L_0 \,\|\, R_0$;
for $i \in [1, \dots, 8]$ **do**
    $L_i = R_{i-1} \oplus F_i(L_{i-1}, k_i)$;
    $R_i = L_{i-1}$;
endfor;
return $L_8 \,\|\, R_8$;

In particular, the function F_i is composed of three subfunctions named FI, FO and FL, that we will introduce below.

Decryption

Since Kasumi is a Feistel cipher, the decryption does not require to compute the inverse of the function F_i. Indeed we have:

$$e_k^{-1} = \overline{\varepsilon_{8,k}}^{-1} \dots \overline{\varepsilon_{1,k}}^{-1}$$

$$\overline{\varepsilon_{i,k}}^{-1} = \begin{pmatrix} \mathbb{0}_{32} & \mathbb{1}_{32} \\ \mathbb{1}_{32} & F_i(\_, k_i) \end{pmatrix} \quad \forall\, i = 1, \dots, 8$$

so that the algorithm proceeds as follows:

Kasumi Decryption

Input I - 64 bits, K - 128 bits
Output O - 64 bits

1: Compute $[k_1, \dots, k_8] = \text{KeySchedule}(K)$;
2: Split I in two 32-bit sequences L_0 and R_0 such that $I = L_0 \,\|\, R_0$;
3: **for** $i \in [1, \dots, 8]$ **do**
4: $\quad L_i = R_{i-1}$;
5: $\quad R_i = L_{i-1} \oplus F_i(R_{i-1}, k_{8-i+1})$;
6: **endfor**;
7: **return** $L_8 \,\|\, R_8$;

Key Schedule

The cipher takes as input a 128-bit key K. Each round of Kasumi uses as round key k_i a total of 128 bits derived from K. These bits are grouped in three different round keys, named KI_i, KO_i and KL_i, associated respectively to the functions FI, FO and FL. More specifically:

$$KI_i = [KI_{i,1}, KI_{i,2}, KI_{i,3}] \quad KO_i = [KO_{i,1}, KO_{i,2}, KO_{i,3}] \quad KL_i = [KL_{i,1}, KL_{i,2}]$$

where each subkey has length equal to 16 bits.
The key schedule computes $K' = K \oplus C$ where C is the following constant:[1]

$$C = \text{0x0123456789ABCDEFFEDCBA9876543210}$$

then splits K as $K = K_1 \,\|\, \dots \,\|\, K_8$ and K' as $K' = K'_1 \,\|\, \dots \,\|\, K'_8$ where K_i and K'_i have length equal to 16 bit for each i. The subkeys are then computed as follows:

$$KI_{i,1} = K'_{i+4} \qquad KI_{i,2} = K'_{i+3} \qquad KI_{i,3} = K'_{i+7}$$
$$KO_{i,1} = Rot(K_{i+1}, 5) \qquad KO_{i,2} = Rot(K_{i+5}, 8) \qquad KO_{i,3} = Rot(K_{i+6}, 13)$$
$$KL_{i,1} = Rot(K_i, 1) \qquad KL_{i,2} = K'_{i+2}$$

[1] The prefix 0x is used to indicate hexadecimal notation.

where the indexes i of K_i and K'_i are cyclic, therefore are computed modulo 8.
Overall, KI_i and KO_i have a total of 48 bits, while KL_i has a total of 32 bits.

Function F_i

The function F_i takes as input a 32-bit sequence I and the round key k_i divided in the three distinct round keys KI_i, KO_i and KL_i. As previously stated, this function makes use of three subfunctions named FI, FO and FL. The first one is not directly used in the main function F_i, it is actually a subfunction of FO.

The form of the function depends on the parity of i, the number of the current round. In odd rounds we apply the subfunction FL, then we apply FO on the output. In even rounds the applications of these two subfunctions are swapped.

Function F_i

Input I - 32 bits, KI_i - 48 bits, KO_i - 48 bits, KL_i - 32 bits
Output O - 32 bits

1: **if** $i \bmod 2 \neq 0$ **then**
2: $\quad O = FO(FL(I, KL_i), KI_i, KO_i)$;
3: **else**;
4: $\quad O = FL(FO(I, KI_i, KO_i), KL_i)$;
5: **endif**;
6: **return** O;

Given the description of F_i, the structure of Kasumi can be reduced to the following scheme:

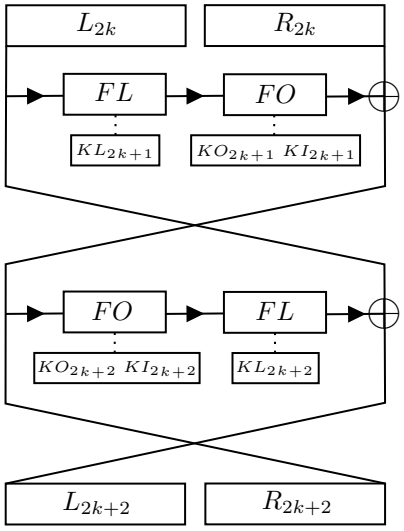

Fig. 1: Scheme of two consecutive rounds of Kasumi where the first one is an odd round

The subfunction FL uses the key $KL_i = KL_{i,1} \,\|\, KL_{i,2}$. The function splits the input I of 32 bits in two parts of 16 bits. The left part is combined with the first part of the round key using the bitwise *and*, then the result is rotated and added to the right part. The result, that is the right part of the output, is combined with the second part of the round key using the bitwise *or*, and again the result is rotated and added to the left part, obtaining the left part of the output.

Function FL

Input I - 32 bits, KL_i - 32 bits
Output O - 32 bits

1: Split I in two 16-bit sequences L and R such that $I = L \,\|\, R$;
2: $R' = R \oplus Rot(L \cap KL_{i,1}, 1)$;
3: $L' = L \oplus Rot(R' \cup KL_{i,2}, 1)$;
4: **return** $L' \,\|\, R'$;

The subfunction FO has a three round Feistel-like structure, so that at the beginning the 32-bit input is divided in two parts of length 16 bits. It takes as input the round keys KI_i and KO_i, where $KI_i = KI_{i,1} \,\|\, KI_{i,2} \,\|\, KI_{i,3}$ is passed as a parameter to the function FI, while $KO_i = KO_{i,1} \,\|\, KO_{i,2} \,\|\, KO_{i,3}$ is used directly in the body of FO. In particular, each round i uses the corresponding parts of the round keys, $KI_{i,j}$ and $KO_{i,j}$. Indeed, in each round we add $KO_{i,j}$ to the left part computed in the previous round. The result is one of the two parameters of FI, along with $KI_{i,j}$.

Having a Feistel-like structure, the function can be written as follows:

$$FO = \overline{\varepsilon_{1,KO_1,KI_1}} \cdot \overline{\varepsilon_{2,KO_2,KI_2}} \cdot \overline{\varepsilon_{3,KO_3,KI_3}}$$

$$\overline{\varepsilon_{i,KO_i,KI_i}} = \begin{pmatrix} \mathbb{0}_{16} & FI(\_ \oplus KO_{i,j}, KI_{i,j}) \\ \mathbb{1}_{16} & \mathbb{1}_{16} \end{pmatrix} \quad \forall\, i = 1, 2, 3$$

Looking at this representation, it is noticeable that this function is not invertible if FI is not invertible, contrary to the classic Feistel function.
Eventually, the algorithm proceeds as follows:

Function FO

Input I - 32 bits, KI_i - 48 bits, KO_i - 48 bits
Output O - 32 bits

1: Split I in two 16-bit sequences L_0 and R_0 such that $I = L_0 \,\|\, R_0$;
2: **for** $j \in [1, 2, 3]$ **do**
3: $\quad L_j = R_{j-1}$;
4: $\quad R_j = FI(L_{j-1} \oplus KO_{i,j}, KI_{i,j}) \oplus R_{j-1}$;
5: **endfor**;
6: **return** $L_3 \,\|\, R_3$;

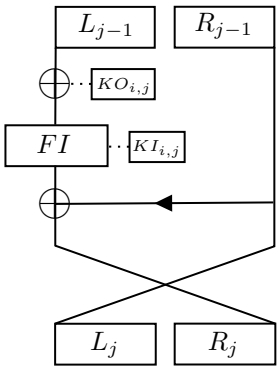

Fig. 2: Scheme of a single round of FO

The subfunction FI has an irregular Feistel-like structure made of 4 rounds. The main feature is that it splits the input in two parts of different length, respectively of nine and seven bits. The round key $KI_{i,j}$ is also split, so that the left part is made of seven bits, and the right one is made of nine bits. In order to combine sequences of these lengths, the function makes use of two auxiliary functions:

- TR Input: 9 bits Output: 7 bits
  Truncates the input by discarding the two most significant bits

- EX Input: 7 bits Output: 9 bits
  Extends the input by adding two zero bits to the most significant end

FI also uses two substitution boxes, named $S7$ and $S9$. The first one is a permutation of $\mathbb{F}^7$ and the second one is a permutation of $\mathbb{F}^9$, where $\mathbb{F}$ is the bit field $\mathbb{F}_2$. For the sake of completeness, we report the two functions as bitwise expressions.

$S7$ Input: $[x_6, x_5, x_4, x_3, x_2, x_1, x_0]$ Output: $[y_6, y_5, y_4, y_3, y_2, y_1, y_0]$

$$y_0 = x_1x_3 \oplus x_4 \oplus x_0x_1x_4 \oplus x_5 \oplus x_2x_5 \oplus x_3x_4x_5 \oplus x_6 \oplus x_0x_6 \oplus x_1x_6 \oplus x_3x_6 \oplus x_2x_4x_6 \oplus x_1x_5x_6 \oplus x_4x_5x_6$$
$$y_1 = x_0x_1 \oplus x_0x_4 \oplus x_2x_4 \oplus x_5 \oplus x_1x_2x_5 \oplus x_0x_3x_5 \oplus x_6 \oplus x_0x_2x_6 \oplus x_3x_6 \oplus x_4x_5x_6 \oplus 1$$
$$y_2 = x_0 \oplus x_0x_3 \oplus x_2x_3 \oplus x_1x_2x_4 \oplus x_0x_3x_4 \oplus x_1x_5 \oplus x_0x_2x_5 \oplus x_0x_6 \oplus x_0x_1x_6 \oplus x_2x_6 \oplus x_4x_6 \oplus 1$$
$$y_3 = x_1 \oplus x_0x_1x_2 \oplus x_1x_4 \oplus x_3x_4 \oplus x_0x_5 \oplus x_0x_1x_5 \oplus x_2x_3x_5 \oplus x_1x_4x_5 \oplus x_2x_6 \oplus x_1x_3x_6$$
$$y_4 = x_0x_2 \oplus x_3 \oplus x_1x_3 \oplus x_1x_4 \oplus x_0x_1x_4 \oplus x_2x_3x_4 \oplus x_0x_5 \oplus x_1x_3x_5 \oplus x_0x_4x_5 \oplus x_1x_6 \oplus x_3x_6 \oplus x_0x_3x_6 \oplus x_5x_6 \oplus 1$$
$$y_5 = x_2 \oplus x_0x_2 \oplus x_0x_3 \oplus x_1x_2x_3 \oplus x_0x_2x_4 \oplus x_0x_5 \oplus x_2x_5 \oplus x_4x_5 \oplus x_1x_6 \oplus x_1x_2x_6 \oplus x_0x_3x_6 \oplus x_3x_4x_6 \oplus x_2x_5x_6 \oplus 1$$
$$y_6 = x_1x_2 \oplus x_0x_1x_3 \oplus x_0x_4 \oplus x_1x_5 \oplus x_3x_5 \oplus x_6 \oplus x_0x_1x_6 \oplus x_2x_3x_6 \oplus x_1x_4x_6 \oplus x_0x_5x_6$$

$S9$ Input: $[x_8, x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0]$ Output: $[y_8, y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0]$

$$y_0 = x_0x_2 \oplus x_3 \oplus x_2x_5 \oplus x_5x_6 \oplus x_0x_7 \oplus x_1x_7 \oplus x_2x_7 \oplus x_4x_8 \oplus x_5x_8 \oplus x_7x_8 \oplus 1$$
$$y_1 = x_1 \oplus x_0x_1 \oplus x_2x_3 \oplus x_0x_4 \oplus x_1x_4 \oplus x_0x_5 \oplus x_3x_5 \oplus x_6 \oplus x_1x_7 \oplus x_2x_7 \oplus x_5x_8 \oplus 1$$
$$y_2 = x_1 \oplus x_0x_3 \oplus x_3x_4 \oplus x_0x_5 \oplus x_2x_6 \oplus x_3x_6 \oplus x_5x_6 \oplus x_4x_7 \oplus x_5x_7 \oplus x_6x_7 \oplus x_8 \oplus x_0x_8 \oplus 1$$
$$y_3 = x_0 \oplus x_1x_2 \oplus x_0x_3 \oplus x_2x_4 \oplus x_5 \oplus x_0x_6 \oplus x_1x_6 \oplus x_4x_7 \oplus x_0x_8 \oplus x_1x_8 \oplus x_7x_8$$
$$y_4 = x_0x_1 \oplus x_1x_3 \oplus x_4 \oplus x_0x_5 \oplus x_3x_6 \oplus x_0x_7 \oplus x_6x_7 \oplus x_1x_8 \oplus x_2x_8 \oplus x_3x_8$$
$$y_5 = x_2 \oplus x_1x_4 \oplus x_4x_5 \oplus x_0x_6 \oplus x_1x_6 \oplus x_3x_7 \oplus x_4x_7 \oplus x_6x_7 \oplus x_5x_8 \oplus x_6x_8 \oplus x_7x_8 \oplus 1$$
$$y_6 = x_0 \oplus x_2x_3 \oplus x_1x_5 \oplus x_2x_5 \oplus x_4x_5 \oplus x_3x_6 \oplus x_4x_6 \oplus x_5x_6 \oplus x_7 \oplus x_1x_8 \oplus x_3x_8 \oplus x_5x_8 \oplus x_7x_8$$
$$y_7 = x_0x_1 \oplus x_0x_2 \oplus x_1x_2 \oplus x_3 \oplus x_0x_3 \oplus x_2x_3 \oplus x_4x_5 \oplus x_2x_6 \oplus x_3x_6 \oplus x_2x_7 \oplus x_5x_7 \oplus x_8 \oplus 1$$
$$y_8 = x_0x_1 \oplus x_2 \oplus x_1x_2 \oplus x_3x_4 \oplus x_1x_5 \oplus x_2x_5 \oplus x_1x_6 \oplus x_4x_6 \oplus x_7 \oplus x_2x_8 \oplus x_3x_8$$

During each round a transformation is performed of the two parts of the input. In even rounds the left part has seven bits and the right one has nine bits, so that the S-box $S7$ is applied to the left part, and the result is combined with the right part truncated using the function TR. In odd rounds the lengths are swapped, so that the S-box $S9$ can be applied to the left part. The result is then combined with the right part extended using the function EX.

In the first three rounds the result of the previous calculation is the new right part, while the new left part is the old right part. In particular, in the second round, the two parts of the round key $KI_{i,j}$ are added to these values according to the length of the

sequences. In the last round these assignments are swapped.
Hence the algorithm proceeds as follows:

Function FI

Input I - 16 bits, $KI_{i,j}$ - 16 bits
Output O - 16 bits

1: Split I in two sequences, L_0 with 9 bits and R_0 with 7 bits, such that $I = L_0 \| R_0$;
2: Split $KI_{i,j}$ in two sequences, $KI_{i,j,1}$ with 7 bits and $KI_{i,j,2}$ with 9 bits, such that $KI_{i,j} = KI_{i,j,1} \| KI_{i,j,2}$;
3: $L_1 = R_0$; $R_1 = S9(L_0) \oplus EX(R_0)$;
4: $L_2 = R_1 \oplus KI_{i,j,2}$; $R_2 = S7(L_1) \oplus TR(R_1) \oplus KI_{i,j,1}$;
5: $L_3 = R_2$; $R_3 = S9(L_2) \oplus EX(R_2)$;
6: $L_4 = S7(L_3) \oplus TR(R_3)$; $R_4 = R_3$;
7: **return** $L_4 \| R_4$;

Implementation's details

All the computational experiments have been carried out using Magma V2.25-3 on the operating system Windows 11 22H2 with processor 3.20 GHz AMD Ryzen 7 5800H (8 cores, 16 threads) and 16GB RAM 3200 MHz SODIMM. Encryption and decryption times were recorded for a list of 10000 test vectors using the Magma function `time`.
Throughout the discussion we will name our code *code1* and then we'll consecutively number the codes obtained with the various modifications.

We report the code of the main function (encryption), that is the implementation of Algorithm 1:

```
function Kasumi(P, K)
    KL, KO, KI := KeySchedule(K);
    L := ChangeUniverse(Reverse(Intseq(StringToInteger(P[1..8], 16), 2, 32)),
    GF(2));
    R := ChangeUniverse(Reverse(Intseq(StringToInteger(P[9..16], 16), 2, 32)),
    GF(2));
    for i in [1, 3, 5, 7] do
        fL := FO(FL(L, KL[i]), KO[i], KI[i]);
        R := [R[j] + fL[j] : j in [1..32]];
        fL := FL(FO(R, KO[i + 1], KI[i + 1]), KL[i + 1]);
        L := [L[j] + fL[j] : j in [1..32]];
    end for;
    C := IntegerToString(Seqint(Reverse(ChangeUniverse(L cat R, Integers())), 2),
    16);
    while #C lt 16 do
        C := "0" cat C;
    end while;
    return C;
end function;
```

The input parameters P and K are strings of bytes in hexadecimal notation, respectively of 16 and 32 characters. From the beginning, we are computing all the operations in `GF(2)` seeing all the objects as sequences in this space. This is done since some bitwise operations (e.g. `Rotate`) in Magma cannot be computed on hexadecimal or integer

values, so we want to avoid unnecessary conversions in the middle of the code, that can be time-consuming. Therefore, we want to convert all the objects in `GF(2)` at this stage of the computation.
The function KeySchedule computes three sequences `KL`, `KO`, `KI` of 8 elements each, such that `KL[i]` $= KL_i$, `KO[i]` $= KO_i$, `KI[i]` $= KI_i$ for $i = 1, \ldots, 8$, where KL_i, KO_i, KI_i are those described before. This structure has been chosen to make the use of the keys in the different rounds straightforward. At this point of the computation, the building elements of the keys lie in `GF(2)`, so that every following operation will be computed in this space.

We use the functions `StringToInteger`, `Intseq` and `Reverse` in this order to implement the conversion of P in binary values. After the conversion we have to split the input in two parts. In particular, we chose to split the plaintext before the conversion since it is better from the point of view of the timings. For the same reason, we also coerce the elements in `GF(2)`. In this way we convert the values one time at the beginning of the algorithm instead of every time we use those variables thereafter. Table 1 reports the timings of our code compared to those of the code (*code2*) obtained when substituting lines 3-4 with the following ones:

```
3     P := Reverse(Intseq(StringToInteger(P, 16), 2, 64));
4     L := P[1..32];
5     R := P[33..64];
```

The eight total rounds to be executed in Kasumi are implemented in pairs, see lines 5-10. In this way we reduce the total number of assignments and we minimize the memory we use. Indeed, we avoid doing two assignments (one for the left part and one for the right one) per round defining the right part after two rounds as the left one after one round, and computing the left part after two rounds using the initial left part that is equal to the right part after one round. This also allows us to use always the same two variables `L` and `R`. All objects are in `GF(2)`, therefore the bitwise xor is implemented as a sum among these elements, see line 7.

In addition, we precompute the output of the functions FO and FL and we save it in the variable `fL`, see for example the lines 6-7. These lines could be replaced with the following line (*code3*):

```
6         R := [R[j] + FO(FL(L, KL[i]), KO[i], KI[i])[j] : j in [1..32]];
```

The latter approach is not efficient since the computation of FO and FL is repeated for each bit (32 times). Again Table 1 reports the compared timings.

At the end, the output parameter C is a string of 16 bytes in hexadecimal notation. The ciphertext is computed converting the binary values in integers with the function `ChangeUniverse`, so that we can apply `Seqint` after using `Reverse`. We can then use `IntegerToString` to convert the ciphertext in hexadecimal notation. Note that at the end we must add zeros to the beginning of the string in case the length was less than 16 characters, since the function `IntegerToString` does not allow setting the number of characters in output.

As a final remark, we make use of the function `ChangeUniverse` instead of coercing each element of a sequence, since the first choice makes the computation faster, see Table 1. The alternative code (*code4*), for example considering line 3, would be:

```
3     L := Reverse(Intseq(StringToInteger(P[1..8],16),2,32));
4     L := [GF(2)!L[i] : i in [1..32]];
```

Consider now the code of the decryption function, that is the implementation of Algorithm 2:

```
1  function KasumiDecryption(C, K)
2      KL, KO, KI := KeySchedule(K);
3      L := ChangeUniverse(Reverse(Intseq(StringToInteger(C[1..8], 16), 2, 32)),
       GF(2));
4      R := ChangeUniverse(Reverse(Intseq(StringToInteger(C[9..16], 16), 2, 32)),
       GF(2));
5      for i in [7, 5, 3, 1] do
6          fL := FL(FO(R, KO[i + 1], KI[i + 1]), KL[i + 1]);
7          L := [L[j] + fL[j] : j in [1..32]];
8          fL := FO(FL(L, KL[i]), KO[i], KI[i]);
9          R := [R[j] + fL[j] : j in [1..32]];
10     end for;
11     P := IntegerToString(Seqint(Reverse(ChangeUniverse(L cat R, Integers())), 2),
       16);
12     while #P lt 16 do
13         P := "0" cat P;
14     end while;
15     return P;
16 end function;
```

Again we execute the rounds in pairs, see lines 5-10. The assignments are the opposite of before, so we define the left part after two rounds as the right one after one round, and we compute the right part after two rounds using the initial right part that is equal to the left part after one round.
Apart from this, the code is analogous to the previous one, so we can make the same considerations. Note that the timings in Table 1 are computed for the various codes obtained by applying the distinct modifications to all the functions (not only to the one for which we have described the optimization).

The code of the key schedule follows, whose algorithm is described in the corresponding section:

```
1  function KeySchedule(K)
2      C := [[GF(2)|0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1], ... ,
       [GF(2)|0,0,1,1,0,0,1,0,0,0,0,1,0,0,0,0]];
3      K := [ChangeUniverse(Reverse(Intseq(StringToInteger(K[i..(i + 3)],16), 2,
       16)), GF(2)) : i in [1..32 by 4]];
4      K1 := [[K[i][j] + C[i][j] : j in [1..16]] : i in [1..8]];
5      KL := [[[K[i][j mod 16 + 1] : j in [1..16]], K1[(i + 1) mod 8 + 1]] : i in
       [1..8]];
6      KO := [[Rotate(K[(i) mod 8 + 1], -5), Rotate(K[(i + 4) mod 8 + 1], -8),
       Rotate(K[(i + 5) mod 8 + 1], -13)] : i in [1..8]];
7      KI := [[K1[(i + 3) mod 8 + 1], K1[(i + 2) mod 8 + 1], K1[(i + 6) mod 8 + 1]]
       : i in [1..8]];
8      return KL, KO, KI;
9  end function;
```

It takes as input a string of 32 bytes in hexadecimal notation, that is converted in binary values in the same way we converted P in the main function. We need to split the key in eight parts, so we split it before the conversion as we did before to optimize the code, see line 3.
The constant that has to be combined with K is directly defined in `GF(2)` in order to avoid an unnecessary conversion when it is actually used. The alternative code (*code5*) where the definition in `GF(2)` is removed, has line 2 replaced with the following one:

```
2     C := [[0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1], ... ,
      [0,0,1,1,0,0,1,0,0,0,0,1,0,0,0,0]];
```

As in the description, the indexes of K and K' (corresponding to `K1`) are computed modulo 8 since they are cyclic. The left rotation of the sequences is implemented using the Magma function `Rotate`.

We report the code of the function FL, that is the implementation of Algorithm 4:

```
1 function FL(I, KL)
2     R := [I[j + 16] + I[j mod 16 + 1] * KL[1][j mod 16 + 1] : j in [1..16]];
3     L := [I[j] + Max([R[j mod 16 + 1], KL[2][j mod 16 + 1]]) : j in [1..16]];
4     return L cat R;
5 end function;
```

The input is a sequence of 32 elements in `GF(2)`. This has to be combined with the round key KL that is divided in two parts of 16 bits each.
We do not explicitly split the input in two parts as it is described in the algorithm, since the steps consist in bitwise operations. Therefore, we can access directly the original sequence.
In particular, the bitwise and is implemented as a multiplication, while the bitwise xor is implemented using the Magma function `Max` (returns the maximum of a sequence), since these pairs of functions are equivalent on the space `GF(2)`.

The left rotation is implemented computing the position in the sequence modulo 16, that is its length. Here we do not use the function `Rotate` since it would be computed one time for each bit (16 times), if it was used directly in the definition of the new sequences as follows:

```
2     R := [I[j + 16] + Rotate([I[i] * KL[1][i] : i in [1..16]], -1)[j] : j in
      [1..16]];
3     L := [I[j] + Rotate([Max([R[i],KL[2][i]]) : i in [1..16]], -1)[j] : j in
      [1..16]];
```

The code obtained with the above modification is named *code6* and its timings will be compared with those of the other codes in Table 1.
A possible alternative would be precomputing the rotated object. Nevertheless, while being equivalent from the point of view of clarity, this solution would require an additional assignment.

We now take into consideration the code of the function FO, that is the implementation of Algorithm 5:

```
1  function F0(I, K0, KI)
2      L := I[1..16];
3      R := I[17..32];
4      Fi := FI([L[j] + K0[1][j] : j in [1..16]], KI[1]);
5      L := [Fi[k] + R[k] : k in [1..16]];
6      Fi := FI([R[j] + K0[2][j] : j in [1..16]], KI[2]);
7      R := [Fi[k] + L[k] : k in [1..16]];
8      Fi := FI([L[j] + K0[3][j] : j in [1..16]], KI[3]);
9      L := [Fi[k] + R[k] : k in [1..16]];
10     return R cat L;
11 end function;
```

We chose not to implement the three rounds of the Feistel-like structure using a `for` structure to reduce assignments and minimize the used memory. Actually, we avoid updating the right and the left part every round. We only update the variable that is not necessary for the following computation, since every round effectively updates only one of the two parts. The lines 4-9 can indeed be replaced by the equivalent code:

```
4      for i in [1,2,3] do
5          Fi := FI([L[j] + K0[i][j] : j in [1..16]], KI[i]);
6          L := R;
7          R := [Fi[k] + R[k] : k in [1..16]];
8      end for;
```

Again, we precompute the output of FI saving it in the variable `Fi` in order to avoid repeating the same computation for 16 times, see for example the lines 4-5. These lines could be replaced with the following line (included in *code3*):

```
4      L := [FI([L[j] + K0[1][j] : j in [1..16]], KI[1])[k] + R[k] : k in [1..16]];
```

Finally it is presented the code of the function FI, that is the implementation of Algorithm 6:

```
1  function S7(N)
2      return [54, ... , 3][N];
3  end function;
4
5  function S9(N)
6      return [167, ... , 461][N];
7  end function;
8
9  function FI(I, KI);
10     L := I[1..9];
11     R := I[10..16];
12     A := ChangeUniverse(Reverse(Intseq(S9(Seqint(Reverse(ChangeUniverse(L,
       Integers())), 2) + 1), 2, 9)), GF(2));
13     L := [A[i] + ([0, 0] cat R)[i] : i in [1..9]];
14     A := ChangeUniverse(Reverse(Intseq(S7(Seqint(Reverse(ChangeUniverse(R,
       Integers())), 2) + 1), 2, 7)), GF(2));
15     R := [A[i] + L[i + 2] + KI[i] : i in [1..7]];
16     A := ChangeUniverse(Reverse(Intseq(S9(Seqint(Reverse(ChangeUniverse([L[i] +
       KI[i + 7] : i in [1..9]], Integers())), 2) + 1), 2, 9)), GF(2));
17     L := [A[i] + ([0, 0] cat R)[i] : i in [1..9]];
18     A := ChangeUniverse(Reverse(Intseq(S7(Seqint(Reverse(ChangeUniverse(R,
       Integers())), 2) + 1), 2, 7)), GF(2));
19     R := [A[i] + L[i + 2] : i in [1..7]];
20     return R cat L;
21 end function;
```

The input parameters I and KI are sequences of 16 elements of `GF(2)`. Here we have an irregular Feistel-like structure, hence we are applying the same optimization devices that we presented for the previous functions. Indeed, in the code we assign the partial computations to the already defined variables correspondent to the left and the right part of the input. In particular, since the partial computation is always split in two parts of seven and nine bits respectively, we always use the variable L to store the part with nine bits, and R to store the other one. So, at the beginning, we actually store the left part of the input in L since it has nine bits, and the right part of the input in the variable R. At the end the left part of the output will be stored in R and the right part in R, so that we must return $R \| L$.
With the same purpose as before, we also precompute the output of the substitution boxes (and we store it in the variable A) to avoid doing the same computation more than one time when defining the subsequent sequence.

Since we are computing all bitwise operations using sequences, the function EX is easily implemented by concatenating a sequence of two zeros with the desired sequence of seven bits. On the other hand, the function TR is implemented by considering the sequence starting from its third position, i.e. ignoring the first two elements. See lines 12 and 14 for an example.

The substitution boxes, see lines 1-7, are implemented as look-up tables instead of bitwise expressions since this choice resulted in a faster computation. Considering the code (*code7*) where those lines are replaced by the following ones:

```
1  function S7(x)
2      return [x[6]*x[5] + ... + x[1], ... , x[6]*x[4] + ... + x[3]*x[2]*x[1]];
3  end function;
4
5  function S9(x)
6      return [x[9]*x[8] + ... + x[6]*x[1], ... , x[9]*x[7] + ... + 1];
7  end function;
```

and, as an example, line 12 is replaced by the following line:

```
12     A := S9(L);
```

we have a faster execution, see the timings reported in Table 1. On the other hand, the alternative implementation is clearer, since the implementation of the function as a look-up table requires the conversion of the sequence into an integer before the application of the function. It is necessary to use this transformation since the s-box returns the element that has position equal to the input value, hence the latter value must be an integer. At the end we require to apply the inverse conversion in order to obtain a sequence again. For an in-depth description of the conversion refer to the previous functions.
The conversion from a sequence to an integer returns a value greater or equal than 0, hence we increment it by 1 since Magma sequences are indexed from 1. Note that we do not have to shift when we convert the output integer in a sequence since the values returned by the s-boxes start from 0.

We chose to implement the substitution boxes as external functions since this solution is faster than declaring the equivalent sequence inside the function FI. See Table 1 for a comparison of the timings of the original code with the one (*code8*) obtained by writing the function FI as follows:

```
function FI(I,KI);
    S7 := [54, ... , 3];
    S9 := [167, ... , 461];
    L := I[1..9];
    R := I[10..16];
    ...
end function;
```

where we removed the outer definition of $S7$ and $S9$, declaring them as sequences at the beginning of FI. Note that in this case $S7$ and $S9$ would be declared every time we execute FI, hence we expect a better performance.

The timings for the various codes are summed up in the following table:[2]

attempts	code1	code2	code3	code4	code5	code6	code7	code8
1 - enc	12.438	13.563	462.500	14.453	14.078	28.438	23.547	12.656
1 - dec	13.438	14.547	483.188	15.828	16.063	28.984	23.922	13.594
2 - enc	12.047	13.734	465.722	14.406	12.672	27.781	23.453	12.234
2 - dec	13.188	14.672	480.238	14.953	14.531	28.969	24.234	13.500
3 - enc	12.125	13.563	463.121	14.203	12.578	26.797	23.594	12.750
3 - dec	13.375	14.781	480.993	14.984	14.109	28.922	24.281	13.516

Table 1: Timings of encryption and decryption with the modifications described in the current section over three attempts

Personal contribution and final considerations

The project described in this report has been implemented by the group consisting of Sara Montanari and myself.
The work was initially divided so that I wrote a draft of the code of the main function *Kasumi* and the function *KeySchedule*, and she wrote the functions *FO*, *FI*, *FN*. We wrote the decryption function at the end of the work together, on the basis of the function *Kasumi*, since they have the same structure.
From the beginning I inserted the precomputation of the objects in the code (the modification associated to *code3*), since it was necessary to avoid time-consuming computations in the main function. When we analyzed the code together, we then extended the modification to the other functions, obtaining a clear improvement.
Moreover, the draft already contained the idea of dividing the eight rounds in pairs and

[2]The modification correspondent to *code3* has not been incorporated in the main functions *Kasumi* and *KasumiDecryption*, otherwise the computation would have lasted more than one day.

was nearly optimized from the point of view of the assignments. Those were later refined together.
The function `ChangeUniverse` was used from the beginning by both of us, and only in the following we checked together that it was actually better than converting the elements one by one. However, in the project we presented, there were some lines in which the modification was not applied. This problem has been fixed by me after the submission of the project.
The implementation of the substitution boxes was discussed together, and Sara then tested that the implementation as look-up tables was actually better. At the beginning the substitution boxes were defined inside the function FI, then following a suggestion of mine we tried to move them outside of the function and together we tested that the latter was the best solution.

After the submission of the project, some personal improvements have been made. I already mentioned the use of `ChangeUniverse` in some lines where it had been omitted, for example in the function *KeySchedule*. Then I decided to define the constant C in the *KeySchedule* directly in `GF(2)` (modification associated to *code5*) to avoid unnecessary conversions.
Another modification I made was to split the objects written in hexadecimal notation before converting them in integers, and the in `GF(2)` (see *code2*). This affected both the main functions and the function *KeySchedule*.
In Table 2 are reported the timings of the submitted code and the modified one, that allow one to see how much these final improvements are essential from the point of view of the optimization.

Some further modifications can be tried, for example using other Magma structures. Since I could not find a proper documentation on how some objects are implemented, and the language Magma is rarely mentioned in online forums, there was no reason to think that other structures would have behaved better than the ones I used.
Moreover, multi-threaded algorithms cannot be implemented in Magma. Otherwise, it is possible to observe that parallelization on multiple core systems could have been useful since some of the above computations can be carried out in parallel. This concept is correspondent to the fact that many consecutive assignments are independent from the point of view of the variables they use. For example, one can see lines 4 and 6 of function FO, or lines 12 and 14 (and 16 and 18) of function FI.

As a final remark, Magma function `time` is not adequate to assess the speed of a program. This is because it executes the statement that follows and returns the real time taken when the statement is completed. This implementation results in counting the time taken by other processes that are not related to the execution of the program.
Magma function `Cputime` is not useful either, since it returns the overall CPU time used, considering again the time taken by other processes.
Table 3 reports the timings recorded when executing five consecutive times the same code (that is the encryption function of the final program). One can see that the maximum error is $\approx 3\%$, that is far from being negligible.

For this reason it is difficult to evaluate if a certain modification is actually more efficient than another one. The experiments above have been carried out over multiple attempts, in such a way that we could tell with high probability that certain choices were better from the point of view of optimization.

attempts	n_code	o_code
1 - enc	11.938	13.266
1 - dec	12.781	14.531
2 - enc	11.938	13.219
2 - dec	12.797	14.487
3 - enc	11.953	13.322
3 - dec	12.688	14.518

Table 2: Timings of encryption and decryption of the new code and the original one over three attempts

attempts	timings
1	12.469
2	23.047
3	24.938
4	18.922
5	8.967

Table 3: Timings of encryption of final code over five consecutive attempts