SYMMETRIC KEY CRYPTOGRAPHY: KASUMI

REPORT OF THE MAGMA IMPLEMENTATION

Paolo Piasenti

Bressanone, January 2019

ABSTRACT. In this report I'm going to describe and comment the MAGMA code which has been written by myself in order to refine the one which was drawn up by the group "Cantor" (composed by Sara Tizianel, Donato Pellegrino and me) and which implements the symmetric key encryption algorithm known as "Kasumi". The dissertation is divided into three parts: in the first, I briefly explain the encryption process and the main features of the cipher; in the second, after a preamble about the implementative choices, a detailed description of all the functions we wrote in our code is exhibited; the third is fully dedicated to tests and results in order to carry out an efficiency comparison between the old and the new code, and a reflection on the improvement of the code and on the rejected (though smart) ideas concludes the treatise.

Contents

1 Overview on the cipher **2**

2 Choices and implementation **2**
2.1 Implementative choices 2
2.2 The code 3
2.2.1 A curious choice: booleans 3
2.2.2 Auxiliary functions 4
2.2.3 Input and output management 4
2.2.4 Key schedule 5
2.2.5 The three protagonist functions 6
2.2.6 S-boxes 8
2.2.7 Kasumi: the final summary 10
2.2.8 Decryption 11

3 Comparisons, results, rejected ideas **11**
3.1 Experiments 11
3.2 Conclusions 14

1 Overview on the cipher

Kasumi is a block cipher that produces a 64-bit output from a 64-bit input under the control of a 128-bit key. In the proposed implementation, both the 64-bit input and output are in the form of a string of 16 hexadecimal characters, while the key is similarly represented as a string of 32 hexadecimal characters.

The core of Kasumi is an eight-round Feistel network, where in each round the round function uses a round key which consists of eight 16-bit subkeys derived from the original 128-bit key using a fixed key schedule, which then we are going to describe. The round function consists itself of the composition of three other Feistel functions (FL, FO, FI), combined in a certain way depending on the parity of the round. The whole process is also messed up with two substitution boxes (S7 and S9).

In my opinion, the complexity of Kasumi does not lie as much in its *conceptual* structure, but rather in its *concrete* structure, in the sense that Kasumi does not handle finite fields or polynomials (as, instead, is the case with "Twofish"), but conversely exhibits a very braided and jumping architecture.

2 Choices and implementation

This is the main chapter of the dissertation, in which I deeply eviscerate my MAGMA code.

2.1 Implementative choices

My optimization has been undertaken in order to respect the following four main ideas. A little experience suggests that some of these are always valid when programming in every language; the others (whose validity is restricted to MAGMA) are instead empirically backed by some MAGMA practice:

1. It is convenient to list everything that is constant and remains constant during the execution of the program, or everything that can be thought constant after previous calculations (for example, a function that operates on a finite set and always behaves in the same way on that input) once and for all among the global variables.

2. It is more efficient to allocate memory all at once instead of frequently extending and shortening variables (in particular sequences), or allocating memory in a dynamic way, (for example allocating a sequence component by component). Also methods and procedures that change the inner structure of variables, like "Reverse", "Rotate", "Append", "cat", "Eltseq", ect... should be avoided if not strictly necessary, in order to reduce execution times.

3. (A little bit inconsistent with the previous point, but actually not really, if one thinks about that for a second...) It is strongly recommended to

use MAGMA *library* functions instead of writing one's own functions. This is because MAGMA has got its own methods already optimized.

4. For a Feitsel network, it is clever to adopt a recursive notation, instead of occupying memory with useless variables, which are only used to compute the following ones. In this way, at every round we only have three variables a, b and c, where the first two are used to calculate the third one, and at the end of the round a is overwritten with b ($a := b$) and b is overwritten with c ($b := c$), so that only ever three memory blocks are employed.

These four principles are the framework in which I developed my new approach to the issue. In the next subsection I am going to explain how they have taken shape throughout the codelines.

2.2 The code

From this point on starts the core of the paper: for every function (or function package) I explain what it produces and how it works, how the guidelines of the previous subsection have been put in practice and what has been changed with respect to the group work code.

2.2.1 A curious choice: booleans

The first - and maybe the more drastic - modification I introduced is to substitute all the "1" with the boolean value "true" and all the "0" with the boolean value "false". Since both the input and the key are converted into binary sequences, so since the context is the binary world, the only thing that matters is whether there is current flowing in a certain cell or not. Getting closer to the machine mother tongue, one understands that a possible inexpensive solution is to represent every 1 as an "ON button" and, conversely, every 0 as an "OFF button". This approach is also useful for two other reasons:

- I don't really know how MAGMA stores the information contained in a sequence of ones and zeros (manages every element as a single bit? Or like an integer?). So, in this way, I am sure that the information stored is as minimal as possible.

- Dealing with sequences of booleans permits me to use the MAGMA library functions performing the standard logical operations, like the extremely important "Xor", but also "And"and "Or". By doing so, I just got a first achievement of the point number 3. of my basic rules.

Thanks to this idea, I have been able to delete all the functions we had written to perform the logical operations, which could be a source of slowdown because, indeed, they had been written by us.

2.2.2 Auxiliary functions

In this subsection I list all the small auxiliary functions I decided to write in my code in order to streamline and reduce the redundancy of the script. The functions are the same used in the group code (or even less); they are very simple and don't require many comments.

```
ZE := function(A)
  return [false, false] cat A;
end function;

TR := function(A)
  return A[3 .. #A];
end function;
```

The first function adds two false-bits at the beginning of a given sequence and the second cuts off the first two of a given sequence. These two functions are used in the FI function, where an *irregular* Feistel-like network takes place.

2.2.3 Input and output management

To deal with the string of 16 hexadecimal characters coming as input, our group's old idea was to create a *look-up table* for both all the sixteen possible hexadecimal characters and all the sixteen possible 4-bits states, and then to establish a bijection between these two structures. I didn't change anything in the two functions that follow but the boolean conversion and a simple notation).

```
BinTable := [ [false, false, false, false],
              [false, false, false, true ],
              [false, false, true , false],
              [false, false, true , true ],
                          ....
              [true , true , false, false],
              [true , true , false, true ],
              [true , true , true , false],
              [true , true , true , true ] ];

HexTable :=
    ["0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"];

HexToBin := function(S)
    return &cat [BinTable[Position(HexTable, S[i])] : i in [1 .. #S]];
end function;

BinToHex := function(S)
    return &cat [HexTable[Position(BinTable, S[i*4+1 .. i*4+4])] : i in
        [0 .. 15]];
end function;
```

How one can imagine, the function called "HexToBin" is the one which transforms the hexadecimal string into the binary/boolean one, while the other function, called "BinToHex" does the reverse operation. The two look-up table are declared as global variables, at the beginning of the script (according to principle no. 1.). The one-to-one correspondence is possible thanks to the MAGMA library function "Position", which returns the position of a certain element in a certain sequence, and thanks to the right order in which the two sequences have been filled in. Note that in the second function I wrote 15 because I already know that it is only used to convert the final output, and hence to convert a 64-bit sequence. Many experimental trials have been conducted to conclude that, among all the possible implementation of this step of the encryption that came to my mind, this one is the most efficient.

2.2.4 Key schedule

The key schedule has been completely redesigned: the main idea was that to avoid sequences filled with sequences of sequences of sequences... well, a literally chaos for the poor processor. I thought to output a sequence of length 8 from the key schedule function, where the i-th component had to contain the information about the keys used in the i-th round of encryption. Hence every component is a sequence that, in turn, contains 8 other 16-bit sequences.

The script of the key schedule function is the following:

```
KeySchedule := function(Key)
    Key := HexToBin(Key);
    K := Partition(Key, 16);
    Kp := Partition(Xor(Key, ConstKey), 16);
    KK := [];

    for i in [1 .. 8] do
      KKi := [];

      KKi[1] := Rotate(K[i], -1);
      KKi[2] := Kp[(i+1) mod 8 +1];

      KKi[3] := Rotate(K[ i  mod 8 +1], -5 );
      KKi[4] := Rotate(K[(i+4) mod 8 +1], -8 );
      KKi[5] := Rotate(K[(i+5) mod 8 +1], -13);

      KKi[6] := Kp[(3+i) mod 8 +1];
      KKi[7] := Kp[(2+i) mod 8 +1];
      KKi[8] := Kp[(6+i) mod 8 +1];

      KK[i] := KKi;
   end for;

   return KK;
end function;
```

The structure of the key generation has been clearly found on the Kasumi specifications and consequently implemented.

The key schedule begins with the conversion of the hexadecimal key into the boolean format (using the "HexToBin" function); then all the 16-bits subsequences of the new boolean key are required, and the best way to do this is to use the MAGMA library function "Partition" (rule no. 3.). Also a constant key is involves in some calculations: according to principle number 1., it has been previously converted and stored among the global variables at the beginning of the code. I thought not to distinguish between FO-keys, FL-keys and FI-keys because this classification is, in my opinion, unnecessary and time-consuming; instead I thought to pass as a parameters to all the functions FO, FI and FL the complete i-th round key, and then to select the right sub-key within the body of each function, remembering that the subkeys had been listed in the bigger sequence always in the same order (such as $KL1, KL2, KO1, KO2, KO3, KI1, KI2, KI3$). Unfortunately, many "Rotate" functions and many dynamic allocations have been involved, but I had no idea how to do this in a different and better way (rule no. 3.).

Someone could object about the way the key are stored, and whether there exists a more efficient solution for the issue. This will be a topic of the last chapter.

2.2.5 The three protagonist functions

Kasumi is based on the action of three central functions, which are called, as already mentioned, FL, FO and FI. An overwiev on each of them is right now shown.

- FL function: this function does nothing strange; the script I wrote is

```
1  FL := function(I, KKi)
2    L  := I[ 1 .. 16];
3    R  := I[17 .. 32];
4
5    R := Xor(R, Rotate(And(L, KKi[1]), -1));
6    L := Xor(L, Rotate(Or(Rp, KKi[2]), -1));
7
8    return Lp cat Rp;
9  end function;
```

  As I have already said, the parameter KKi is the complete i-th round key, and remembering the filling order I have that to access to $KL1$ and $KL2$ I only have to write $KKi[1]$ and $KKi[2]$ (note the use of the MAGMA library functions "And" and "Or").

- FI function: this is the function which uses an irregular Feistel pattern, and in which two S-boxes act. The code of the function is the following.

```
FI := function(I, KIi)
  A := KIi[1 ..  7];
  B := KIi[8 .. 16];

  L0 := I[ 1 .. 9];
  R0 := I[10 .. 16];

  L1 := R0;
  R1 := Xor(S9(L0), ZE(R0));

  L2 := Xor(R1, B);
  R2 := Xor(Xor(S7(L1), TR(R1)), A);

  L3 := R2;
  R3 := Xor(S9(L2), ZE(R2));

  L4 := Xor(S7(L3), TR(R3));
  R4 := R3;

  return L4 cat R4;
end function;
```

A recursive arrangement is not possible here. The passed parameter Ki is a 16-bit vector, which is then split into two different-sized subsequences. Here I decided not to pass the complete i-th round key because this function operates separately on each key $KI1, KI2$ and $KI3$ (and this happens when the next described function, such as FO, is called: in fact, as we are going to see, the complete i-th round key is there inputted). Here the reader can observe how the auxiliary functions are putted into practice. The functions whose names are $S7$ and $S9$ are the S-boxes and are described in the next subsection.

- FO function: this function is, among the three, maybe the more complex one. It has a Feistel structure, which has allowed me to write my code in a recursive way. Here the code:

```
FO := function(I, KKi)
    L  := I[ 1 .. 16];
    R1 := I[17 .. 32];

    for i in [3 .. 5] do
        R2 := Xor(FI(Xor(L, KKi[i]), KKi[i+3]), R1);
        L  := R1;
        R1 := R2;
    end for;

    return L cat R1;
end function;
```

Note how FI is used within this function. Here, unlike the case of FI but similarly to the case of FL, it makes sense to pass as parameter the complete i-th round key. Thanks to the Feistel structure, it is interesting to see how I have been able to implement the cycle in a recursive way, using a translation of the index interval (which permits me to catch the right subkeys in the right order) and the two updating assignments $L := R1$ and $R1 := R2$. This is a first example of application of the fourth point of my guidelines.

These three functions have been partly modified with respect to the old version, in order to make them work faster. Some notations have also undergone little modifications.

2.2.6 S-boxes

The way I treated the S-boxes is the most emblematic incarnation of the principle number 1. of my list. On the Kasumi specification it's described how formally the two S-boxes should work. They are two permutations, $S9$ on the set 0,...,511 and $S7$ on the set 0,...,127. Since they are *constant* permutation, in the sense that their output does not depend on other variables but the input (for example the parity of the round), we can precalculate all the images of all the possible values in the involved set and store the values in a sequence. The way we store these data has a precise sense: given the look up sequence S, the value $S[i]$ in the i-th position of S has to correspond to the image of $i-1$. The reason of the "minus one"is because the two sets collect the number starting from 0, but a MAGMA sequence doen not possess a 0-th position. Hence, to get the image of j under a substitution box, we have to take the value $S[j+1]$.

Differently from the group work, a serious improvement which came to my mind was that to store the values in the $S9$ and $S7$ sequences not as integers, but already in the binary form of boolean sequences. In this way, during the execution, a lot of time is spared, because the computer must non spend time to compute back the conversion to bit representation, because it has been already memorized (rube no. 1.). I'm not going post here my $S9$ and $S7$ look-up tables, because after the conversion they became hugely long. They can be found at the beginning of my code, among the global variables. However, assuming that one knows the two sequences $S9$ and $S7$, the code of the S-boxes is:

```
S9 := function(N)
    return S9Table[BinToInt_9(N)+1];
end function;

S7 := function(N)
    return S7Table[BinToInt_7(N)+1];
end function;
```

The two functions which appear up here are two other functions supporting my S-boxes implementation, and they are defined as follows:

```
BinToInt_7 := function(N)
  S := [0, 0, 0, 0, 0, 0, 0];

  for i in [1 .. 7] do
    if N[i] eq true then S[8-i] := 1;
    end if;
  end for;

  return Seqint(S, 2);
end function;

BinToInt_9 := function(N)
  S := [0, 0, 0, 0, 0, 0, 0, 0, 0];

  for i in [1 .. 9] do
    if N[i] eq true then S[10-i] := 1;
    end if;
  end for;

  return Seqint(S, 2);
end function;
```

The idea behind these two functions is strictly linked to the way we take advantage of the S-boxes tables. One has either a 9-boolean or a 7-boolean sequence and has to find the image of the corresponding integer under the permutation $S9$ or $S7$, respectively. To do that, one computes first of all the binary representation of the sequence (with 0s and 1s), and then uses the library function "Seqint" with specified base 2, which converts a binary string to the relative integer. These two functions have suffered several changes. Indeed, the reader might be demanding why two functions instead of a single one. The root of the problem is that MAGMA considers as less significant bit the leftmost one in a binary sequence. Conversely, in the Kasumi algorithm, everything has been built considering as less significant bit the rightmost one. At first glance, one could think to use the "Reverse" command, but in order to respect the second principle and to speed up the program, a smart intuition is to preallocate a sequence with all zeros of length the length of the input, and then to fill it in backwards (for example, if there is a "true" in the second position of the input, the penultimate position of the new sequence is declared to be a 1), so as to get the converted sequence in the correct flipped way. Here some problems begin: the only technique I know to allocate a sequence full of zeros of a certain length is to use the "ZeroSequence" command, whose inputs are the length (which could be obtained using "♯" on the sequence to be converted), but also the ring where the function has to take the "zero", i. e. the neutral element. I tried to construct the integer ring with "Integers", but I noticed that that this strategy was very very inefficient. That's why I thought to write two distinct functions, one for the 9-lengthy inputs and one for the 7-lengthy inputs, so I could write "by hand" two distinct sequences with all zeros, one for one kind of inputs and

one for the other kind.

2.2.7 Kasumi: the final summary

Now that all the fundamental subfunctions have been described, it's time to take a look at the final function, which contains the crucial algorithm of encryption. It has a Feistel network shape, which allows, again, to write the code in a recursive way, as one can note. An auxiliary function simplifies the notation, and its task it that to serve as a switch between the even and the odd rounds. In fast, as we have already underlined, the round function depends on the parity of the round. Here is the code of the fi auxiliary function:

```
fi := function(I, KKi, i)
    if IsEven(i) then
        return FL(FO(I, KKi), KKi);
    else
        return FO(FL(I, KKi), KKi);
    end if;
end function;
```

So the difference between an even and an odd round function lies in the different way the three basic functions FL, FO and FI are combined. Given that, the structure of the Kasumi function is not hard to understand:

```
Kasumi := function(P, K)
  KK := KeySchedule(K);
  P  := HexToBin(P);

  L0 := P[ 1 .. 32];
  R0 := P[33 .. 64];
  L1 := Xor(R0, fi(L0, KK[1], 1));

  for i in [2 .. 8] do
      f_i := fi(L1, KK[i], i);
      L2  := Xor(L0, f_i);

      L0 := L1;
      L1 := L2;
  end for;
  return BinToHex(L1 cat L0);
end function;
```

First of all, the input is converted and the key schedule function is called. The first launching round of encryption is done out of the for, and then the iterative machinery takes place. At the end, before outputting, the conversion back to the hexadecimals is pursued.

2.2.8 Decryption

The decryption process is very simple, because it is just doing the same operations which have been done for encrypting, but vice versa, starting from the end. If one has the key and knows the encryption procedure, it is a banality to derive the key schedule and to use the key in the reverse order,starting from the ciphertext. The code of the decryption function "KasumiDecryption" follows.

```
KasumiDecryption := function(C, K)
  KK := KeySchedule(K);
  C := HexToBin(C);

  L8 := C[ 1 .. 32];
  R8 := C[33 .. 64];
  R7 := Xor(L8, fi(R8, KK[8], 8));

  for i in [1 .. 7] do
    f_i := fi(R7, KK[8-i], 8-i);
    R6 := Xor(R8, f_i);

    R8 := R7;
    R7 := R6;
  end for;
  return BinToHex(R8 cat R7);
end function;
```

3 Comparisons, results, rejected ideas

3.1 Experiments

All the tests have been conducted on a Dell XPS 13 with processor Intel i5-6200U CPU @ 2.30GHz × 4 and the speed has always been measured through the MAGMA command "time". Every signle modification has been tested, and every time I registered an improvement in the performance of the program after a code modification, that modification was adopted.

In the following table, we can observe the comparison between the old and the new code, with respect to 100, 1000 and 10000 inputs.

No. of inputs	Old code	New code
100 inputs	0.390 s	0.140 s
1000 inputs	3.440 s	1.250 s
10000 inputs	33.580 s	12.340 s

This tests have been conducted after having just turned on the pc. Between two tests, the computer has been restarted. We can see how the new code is significantly faster than the old one: about 2/3 of the time have been spared.

Among all of the ideas I have had, but which have proved to be useless because not so efficient as I taught, I want highlight three ideas in particular:

- The first concerns the script I wrote for the binary conversion; which is the following:

```
function HexToBit(STR)
  P := [];
  ZERO := [0, 0, 0, 0];

  for i in [1 .. #STR] do
    X := STR[i];
    X := StringToInteger(X, 16);

    SEQ := Reverse(Intseq(X, 2));
    SEQ := Insert(ZERO, 5 - #SEQ, 4, SEQ);

    P := P cat SEQ;
  end for;

  return P;
end function;
```

  How the reader can clearly observe, it is a very complex functions and is fully in contradiction with the second principle of my list. However, it was the first function I wrote (before thinking to the booleans) and it worked well. I thought it could be faster than the look-up table, but I was wrong. Never hurts to try!

- The second pretty nice idea I had was to not convert the boolean sequences to integers in order to be user as indexes for the S-boxes look-up sequences, but to store in memory a sequence P written as a mathematical permutation of all the possible boolean sequences representing the numbers from 0 to 511 (or 127). In this way, to calculate the image under the S-box of a certain sequence S, one would have had to calculate the position in which S appeared in P and then to take the sequence located immediately after. In this way no conversion is needed, and I imagined the "Position"to be faster than how it actually is. I discarded this idea; here, though, I have conserved the P sequence for $S9$(not the converted one, because it does not fit here), which I got using some MAGMA tools.

```
P :=[151,151,336,336,2, 161, 292, 313, 178, 208, 173, 426, 283,
     467, 219, 430, 419, 297, 325, 88, 164, 393, 34, 489, 118, 113,
     300, 40, 44, 143, 124, 36, 206, 244, 234, 274, 78, 288, 72,
     240, 132, 68, 257, 103, 142, 442, 224, 280, 231, 495, 380,
     162, 46, 81, 407, 319, 190, 199, 305, 233, 388, 441, 258, 125,
     93, 421, 213, 368, 414, 92, 458, 348, 303, 355, 497, 66, 395,
     509, 464, 97, 158, 168, 456, 57, 264, 498, 60, 371, 96, 232,
     367, 84, 89, 74, 462, 33, 241, 225, 383, 201, 345, 435, 468,
     94, 350, 483, 122, 87, 428, 294, 431, 127, 27, 82, 249, 286,
     505, 237, 417, 422, 130, 482, 387, 144, 475, 102, 491, 449,
     353, 85, 186, 359, 269, 329, 484, 192, 465, 30, 356, 474, 263,
     146, 508, 436, 194, 252, 347, 357, 289, 26, 262, 453, 471,
     493, 323, 137, 403, 141, 115, 242, 203, 502, 202, 153, 169,
     267, 230, 155, 268, 256, 35, 37, 17, 253, 307, 71, 212, 386,
     343, 77, 507, 376, 413, 481, 477, 207, 282, 139, 20, 415, 404,
     396, 138, 339, 504, 8, 38, 0, 167, 24, 306, 389, 365, 104,
     191, 277, 205, 61, 290, 342, 304, 185, 209, 222, 166, 324,
     181, 401, 352, 47, 400, 311, 55, 405, 346, 21, 340, 492, 293,
     490, 180, 349, 332, 370, 394, 182, 434, 7, 338, 4, 391, 298,
     149, 170, 157, 321, 109, 227, 128, 487, 5, 334, 412, 443, 447,
     398, 335, 111, 135, 131, 41, 254, 140, 39, 333, 275, 410, 429,
     360, 296, 411, 228, 382, 302, 174, 309, 273, 116, 437,2, 1,
     239, 454, 470, 129, 446, 80, 501, 445, 187, 52, 54, 218, 195,
     287, 32, 175, 229, 408, 450, 105, 69, 480, 438, 117, 320,1, 3,
     379, 255, 374, 511, 461, 188, 56, 472, 64, 165, 327, 308, 448,
     284, 29, 159, 364, 31, 177, 506, 15, 397, 210, 418, 248, 503,
     432, 312, 110, 366, 433, 377, 361, 478, 114, 276, 10, 48, 95,
     163, 499, 455,184, 16, 183, 236, 459, 63, 76, 406, 123, 317,
     392, 251, 211, 67, 121, 243, 316, 469, 160, 363, 106, 193,
     416, 485, 42, 378, 409, 281, 45, 220, 49,3,6, 9, 226, 373,
     154, 73, 28, 216, 476, 189, 120, 11, 358, 100, 13, 385, 351,
     18, 147, 53, 235, 107, 425, 217, 291, 150, 479, 402, 328, 260,
     19, 331, 272, 50, 315, 12, 452, 299, 473, 179, 271, 285, 86,
     221, 79, 223, 330, 98, 134, 457, 200, 420, 23, 362, 270, 198,
     83, 265, 314, 322, 375, 148, 112, 344, 337, 318, 369, 486,
     381, 215, 101, 250, 424, 22, 51, 245, 14, 90, 440, 463, 451,
     390, 108, 152, 126, 496, 43, 58, 172, 488, 145, 384, 266, 444,
     247, 91, 196, 246, 301, 119, 278, 510, 59, 494, 136, 326, 75,
     176, 439, 295, 238, 62, 399, 133, 156, 279, 171, 460, 204, 65,
     197,6,25, 500, 341, 259, 427, 70, 423, 466, 310, 372, 99,
     354,25,214, 261,214];
```

Another problem of this method was that the permutation was not composed by one single cycle, so I had to construct the P sequence adding some extra numbers paying attention to what was happening. In fact, in a cycle, the last element is sent in the first, but when one pastes two cycle this information can be lost: for example, the image of 437 is 2, so in order to make my idea work I had to insert a 2 after 437 and before the other

cycle. But what happens when "Position(2)" is called? I read the specifications of that function and it is said that it takes the first occurrence of 2 which is found, so it was all okay. There were only few critical points, and I checked them all. The method worked, but it was very inefficient.

- The third idea is actually a collection of ideas concerning the key schedule step. I tried all possible groupings of the keys and sub keys, for example listing all the values in a single infinitely long (1024 bits) sequence, and then selecting the needed subkeys with the $K[x\,..\,y]$ notation, or for example grouping all the KO subkeys in a sequence, all the KL subkeys in another one and all the KI subkeys in one more. The bottom line is that among all the possibilities I have tried, the one I kept is the quickest one, as well as the tidiest and most understandable one

3.2 Conclusions

Concluding the dissertation, I want to underline how interesting, stimulating and challenging my whole work has been. I am very happy to have had the opportunity to "play" with MAGMA once again, and to have improved my capabilities with it.