University of Trento

Faculty of Mathematical, Physical and Natural Sciences
Department of Mathematic

Construction and Evaluation of Block Ciphers

Stefano Martin

Supervisor: Prof. Massimiliano Sala

Referee: Prof. Andrea Caranti

Academic Year 2010/2011

Part IV

Programmes

15 Programming languages

In this part of the thesis we will give you a summary about the programmes that we have created to analyze the block ciphers. All these programmes are tested in the operative system Linux. We used three programming languages to build each programmes.

MAGMA

The programme, which we used more, is MAGMA. It has a its programming language.
It is a powerful computer algebra system designed to solve problems in algebra, number, theory, geometry and combinatorics. MAGMA is not a free software and it is available for purchase from the web-site of MAGMA: `http://magma.maths.usyd.edu.au`.

A good documentation is available on the same web-site here:

`http://magma.maths.usyd.edu.au/magma/handbook`

Singular

We used this programme to compute the Gröbner basis and to solve some equations. It has a its programming language.
It is a computer algebra system for polynomial computations with special emphasis on the needs of commutative algebra, algebraic geometry and singularity theory. Singular is a free software and you can download it from the web-site of Singular: `http://www.singular.uni-kl.de`.

A good documentation is available on the same web-site here:

`http://www.singular.uni-kl.de/Manual/latest/index.htm`

C++

We used this language to use the NIST tests and to build a faster version of BunnyTN. C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is very fast and for this reason we prefer sometimes to use it instead MAGMA.

16 Programmes

In this section we describe the programme that we will give available. We will use this notation.

The title of the subsection will be the name of the folder where the programme is contained.

PROGRAMME: name of the programme
DESCRIPTION: summary of the programme
MATH: mathematical concept and section where we have described the programme
LANGUAGE: programming languages that we used for the programme
INPUT: the input of the programme
GLOBAL ASSUMPTIONS: eventually global assumptions that this programme uses
FUNCTIONS: eventually other functions that this programme uses
OUTPUT: the output of the programme

16.1 Isomorphism

PROGRAMME: Vector2Element
DESCRIPTION: This programme transforms a n-dimensional vector in $\mathbb{F}_p$ (p prime value) in an element of the field $\mathbb{F}_{p^n}$.
MATH: Isomorphism ξ in the Section 5.1
LANGUAGE: MAGMA
INPUT: A vector v
OUTPUT: An element $c = \xi(c)$

```
1 Vector2Element := function(v)
    local length, F, char, k;
3
    length := NumberOfColumns(v);
```

```
5   F := Parent(v[1]);
    char := #F;
7   F<e> := GF(char,length);
    k := F!0;
9
    for i in [0..length-1] do
11    k := k + v[length-i]*e^(i);
    end for;
13
    return k;
15 end function;
```

EXAMPLE

```
1 V := VectorSpace(GF(2),6);
  v := Random(V);
3 Vector2Element(v);
```

PROGRAMME: Element2Vector
DESCRIPTION: This programme transforms an element of the field $\mathbb{F}_{p^n}$ in a n-dimensional vector over $\mathbb{F}_p$.
MATH: Isomorphism ξ^{-1} in the Section 5.1
LANGUAGE: MAGMA
INPUT: An element c
OUTPUT: A vector $v = \xi^{-1}(c)$

```
1 Element2Vector := function(c)
    local F, e, degree, char, VDeg, R, prim, v, ln, p;
3
    F := Parent(c);
5   e := PrimitiveElement(F);
    degree := Degree(F);
7   char := Characteristic(F);
    VDeg := VectorSpace(GF(char), degree);
9   R<x> := PolynomialRing(GF(char));
    prim := PrimitivePolynomial(GF(char), degree);
11  v := [];

13
```

```
  if c eq 0 then
    v := Zero(VDeg);
  else
    if c eq 1 then
      ln := 0;
    else
      ln := Log(e, c);
    end if;
    p := x^ln mod(prim);
    for i in [0..degree-1] do
      v:=Append(v,Coefficient(p,degree-1-i));
    end for;
  end if;

  return VDeg!v;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
c := Random(E);
Element2Vector(c);
```

16.2 Delta-differential Uniformity

PROGRAMME: DeltaDiffUnif
DESCRIPTION: This programme says us the δ of the δ-differential uniformity of an invertible function f.
MATH: δ-differential uniformity in the Section 6.2
LANGUAGE: MAGMA
INPUT: An invertible map f
OUTPUT: The value δ of the δ-differential uniformity of f.

```
DeltaDiffUnif := function(f)
  local V, max, numV, m;

  V := Domain(f);
  max:=0;
  numV := #V;

  for a in V do
    for b in V do
      m:=0;
      for x in V do
        if ((f(x) + f(x+a)) eq b) then
          m:=m+1;
        end if;
      end for;
      if (m gt max) and (m ne numV) then
        max := m;
      end if;
    end for;
  end for;

  return max;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
DeltaDiffUnif(f);
```

PROGRAMME: DDT
DESCRIPTION: This programme gives us the differential distribution table (DDT) of an invertible function f.
MATH: DDT in Section 6.2 Definition 15
LANGUAGE: MAGMA
INPUT: An invertible map f
OUTPUT: The matrix DDT of f.

```
DDT := function(f)
  local V, max, numV, DDT, i, j, m;

  V := Domain(f);
  numV := #V;
  DDT := Matrix(IntegerRing(), numV, numV, []);

  i := 0;
  for a in V do
    i := i + 1;
    j := 0;
    for b in V do
      j := j + 1;
      m:=0;
      for x in V do
        if ((f(x) + f(x+a)) eq b) then
          m:=m+1;
        end if;
      end for;
      DDT[i][j] := m;
    end for;
  end for;

  return DDT;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
```

```
3 DDT(f);
```

PROGRAMME: WDeltaDiffUnif
DESCRIPTION: This programme says us the δ of the weakly δ-differential uniformity of an invertible function f.
MATH: Weakly δ-differential uniformity in the Section 11.1
LANGUAGE: MAGMA
INPUT: An invertible map f
OUTPUT: The value δ of the weakly δ-differential uniformity of f.

```
1  WDeltaDiffUnif := function(f)
     local V, min, numV, A, y, division, val;
3
     V := Domain(f);
5    dim := Degree(V);
     char := Root(#V,dim);
7    min := 0;
     numV := char^(dim - 1);
9
     for u in V do
11     A := {};
       for x in V do
13       y := f(x + u) + f(x);
         A := Include(A, y);
15     end for;
       m := #A;
17     if (m gt min) then
         min := m;
19     end if;
     end for;
21
     for delta in [1..numV] do
23     m := numV/(delta*char);
       if m lt min then
25       return delta*2;
       end if;
27   end for;

29   return "Error!";
```

```
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
WDeltaDiffUnif(f);
```

16.3 Classical Non-Linearity

PROGRAMME: SumBF
DESCRIPTION: We receive a order set of functions, a vector and a vector of the domain of the functions. We will have the output the sum of all this functions.
MATH: Creation of a function from a order set of functions. In Section 6.3 equation 6.1.
LANGUAGE: MAGMA
INPUT: An order of Boolean function $H := [h_1, \ldots, h_n]$ where $h_i : \mathbb{F}^r \to \mathbb{F}\ \forall i \in \{1, \ldots, n\}$, a n-dimensional vector w and a r-dimensional vector v
OUTPUT: $\sum_{i=1}^{n} w_i h_i(v)$.

```
SumBF := function(H,w,v)
  local V, val, sum;

  V := Domain(H[1]);
  val := NumberOfColumns(w);
  sum := 0;

  for i in [1..val] do
    sum := sum + w[i]*H[i](v);
  end for;

  return sum;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
E2 := VectorSpace(E,2);
H := [map<E -> E|x :-> x^6>, map<E -> E|x :-> x + e>];
v := e^2;
w := E2![e,e^2];
SumBF(H,w,v);
```

PROGRAMME: BooleanFunction
DESCRIPTION: We receive a invertible multiboolean function f. We will have the output all the combinations of the Boolean functions which compose f.

MATH: Creation of the order set of the functions created by the combinations of the Boolean functions, which compose a multi-Boolean function. In Section 6.3 equation 6.1.
LANGUAGE: MAGMA
INPUT: $f : \mathbb{F}^n \to \mathbb{F}^n$, $f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n))$ where $f_i : \mathbb{F}^n \to \mathbb{F}\ \forall i \in \{1, \ldots, n\}$
FUNCTIONS: Element2Vector, SumBF, Vector2Element
OUTPUT: An order set $F = \{F_1, \ldots, F_{2^n-1}\}$ where $F_i : \mathbb{F}^n \to \mathbb{F}\ F_i(v) = \sum_{j=1}^{n} w_{i_j} h_j(v)$ $\forall i \in \{\ldots, n\}$ and w_i is a Boolean vector of $\mathbb{F}^{n*}$.

```
BooleanFunction := function(f)
  local F, Y, numF, dim, g, H;

  F := Domain(f);
  numF := #F;
  dim := Degree(F);
  Vdim := VectorSpace(GF(2), dim);
  Y := [x:x in Vdim | x ne 0];
  g := map<Vdim -> Vdim | v :-> Element2Vector(f(Vector2Element(v)))>;

  H := [];
  F := [];
  for i in [1..dim] do
    H[i] := map<Vdim -> GF(2) | v :-> (g(v))[i]>;
  end for;
  for i in [1..numF-1] do
    F[i] := map<Vdim -> GF(2) | v :-> SumBF(H,Y[i],v)>;
  end for;

  return F;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
BooleanFunction(f);
```

PROGRAMME: Distance
DESCRIPTION: We receive two invertible Boolean function f, g. We compute the Hamming's distance between f and g.
MATH: Computation of distance between two Boolean functions. In Section 6.3 Definition 16.
LANGUAGE: MAGMA
INPUT: Two functions $f, g : \mathbb{F}^n \to \mathbb{F}$.
OUTPUT: $d(f, g)$

```
Distance := function(f,g)
  local dist, V, X;

  dist := 0;
  V := Domain(f);
  X := [x : x in V];

  for x in X do
    if f(x) ne g(x) then
      dist := dist + 1;
    end if;
  end for;

  return dist;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> GF(2)| x :-> x[1]>;
g := map<V -> GF(2)| x :-> x[2] + 1>;
Distance(f,g);
```

PROGRAMME: DistanceFromAGLBoolean
DESCRIPTION: We receive an invertible Boolean function f. We compute the Hamming's distance between f and the group of the affine functions.
MATH: Computation of distance between a function f and the group of the affine functions. In Section 6.3 Definition 16.

LANGUAGE: MAGMA
INPUT: A function $f : \mathbb{F}^n \to \mathbb{F}$
FUNCTIONS: Distance
OUTPUT: $\mathrm{d}(f, \mathrm{AGL}(\mathbb{F}^n, \mathbb{F}))$.

```
DistanceFromAGLBoolean := function(f)
  local V, Y, max, g, d;

  V := Domain(f);
  Y := {x : x in V | x ne 0};
  max := #V;

  for w in Y do
    for a in GF(2) do
      g := map<V -> GF(2) | v :-> (InnerProduct(v,w) + a)>;
      d := Distance(g, f);
      if d lt max then
        max := d;
      end if;
    end for;
  end for;

  return max;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> GF(2)| x :-> x[1]>;
DistanceFromAGLBoolean(f);
```

PROGRAMME: DistanceFromAGLNBoolean
DESCRIPTION: We receive a set F of invertible Boolean functions. We compute the Hamming's distance between F and the group of the affine functions.
MATH: Computation of distance between a set F of Boolean functions and the group of the affine functions. In Section 6.3 Definition 16.
LANGUAGE: MAGMA
INPUT: A set of Boolean functions $F = \{\mathbb{F}_1, \ldots, \mathbb{F}_n\}$ where $F_i = \mathbb{F}^r \to \mathbb{F}$, $\forall i \in$

$\{1, \ldots, n\}$
FUNCTIONS: DistanceFromAGLBoolean
OUTPUT: $\max_i(\mathrm{d}(F_i, \mathrm{AGL}(\mathbb{F}^n, \mathbb{F})))$.

```
DistanceFromAGLNBoolean := function(F)
  local V, max, d;

  V := Domain(F[1]);
  max := #V;
  for i in [1..#F] do
    d := DistanceFromAGLBoolean(F[i]);
    if d le max then
      max := d;
    end if;
  end for;

  return max;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
F := [map<V -> GF(2)| x :-> x[1]>, map<V -> GF(2)| x :-> x[2] + 1>];
DistanceFromAGLNBoolean(F);
```

PROGRAMME: ClassicalNonLinearity
DESCRIPTION: We receive an invertible multiboolean function f. We compute its classical non-linearity.
MATH: Computation of classical non-linearity of the multiboolean function f. In Section 6.3 equation 6.1.
LANGUAGE: MAGMA
INPUT: An invertible multiboolean function $f : \mathbb{F}_n \to \mathbb{F}_n$
FUNCTIONS: BooleanFunction, DistanceFromAGLNBoolean
OUTPUT: N(f)

```
ClassicalNonLinearity := function(f)
  local F;
  F := BooleanFunction(f);
```

```
  return DistanceFromAGLNBoolean(F);
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
ClassicalNonLinearity(f);
```

In the file "ClassicalLinearity" there are all these functions in the correct order.

16.4 MDS

PROGRAMME: IsMDSMatrix
DESCRIPTION: This programme says us if a matrix is MDS or not.
MATH: MDS matrix. In Section 7.1.
LANGUAGE: MAGMA
INPUT: A matrix M
OUTPUT: "The Matrix is not square" is the matrix is not square, "The matrix is not MDS" if the matrix is not MDS, "The matrix is MDS" if the matrix is MDS

```
IsMDSMatrix := function(M)
  local n,V,X,I,J;

  if Nrows(M) ne Ncols(M) then
    return "The␣Matrix␣is␣not␣square";
  end if;

  n := Ncols(M);
  V := VectorSpace(GF(2), n);
  X := {x : x in V | x ne 0};

  for v in X do
    for w in X do
      if Weight(v) eq Weight(w) then
        I := [];
        for i in [1..n] do
          if v[i] eq 1 then
            I := Append(I,i);
          end if;
        end for;
        J := [];
        for j in [1..n] do
          if w[j] eq 1 then
            J := Append(J,j);
          end if;
        end for;
        if Minor(M, I, J) eq 0 then
          printf"Righe␣=␣%o\n",I;
          printf"Colonne␣=␣%o\n",J;
          return "The␣matrix␣is␣not␣MDS";
```

```
31          end if;
        end if;
33      end for;
    end for;
35
    return "The Matrix is MDS";
37  end function;
```

EXAMPLE

```
1  E<e> := GF(2,6);
   M := Random(GL(6,E));
3  IsMDSMatrix(M);
```

16.5 Operations

PROGRAMME: Divide
DESCRIPTION: This programme divides a vector v in n vectors smaller than v.
MATH: Function div, in Section 6.
LANGUAGE: MAGMA
INPUT: A r-dimensional vector $v = (v_1, \ldots, v_r)$, the value n
OUTPUT: If n does not divide r then: "n does not divide the length of v". If n does not divide r then we receive a $n \times \frac{r}{n}$ matrix b where the line $b_i = (v_{n\cdot(i-1)+1}, \ldots, v_{n\cdot(i-1)+n})$ $\forall i \in \{1, \ldots, \frac{r}{n}\}$

```
Divide := function(v,n)
  local m, F, b;

  if (NumberOfColumns(v) mod n) ne 0 then
    return "n does not divide the length of v";
  end if;

  m := NumberOfColumns(v) div n;
  F := Parent(v[1]);
  b := Matrix(F, n, m, []);
  for i in [1..n] do
    for j in [1..m] do
      b[i][j] := v[(i-1)*m + j];
    end for;
  end for;

  return b;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
v := Random(V);
Divide(v,4);
```

PROGRAMME: Join
DESCRIPTION: This programme joins the lines of a matrix b in a unique vector v.

MATH: Function div^{-1}, in Section 6.
LANGUAGE: MAGMA
INPUT: A $n \times r$ matrix b
OUTPUT: A nr-dimensional vector v where $v_i = b_{\lfloor \frac{i-1}{r} \rfloor + 1, (i \bmod r)+1}$

```
Join := function(b)
  local nrows, ncolumns, F, v, V;

  nrows := NumberOfRows(b);
  ncolumns := NumberOfColumns(b);
  F := Parent(b[1][1]);
  v := Matrix(F, 1, nrows * ncolumns, []);
  V := VectorSpace(F, nrows * ncolumns);

  for i in [1..nrows] do
    for j in [1..ncolumns] do
      v[1][(i-1)*ncolumns + j] := b[i][j];
    end for;
  end for;

  return V!v;
end function;
```

EXAMPLE

```
b := Matrix(GF(2), 3, 4, [1,0,0,1,
                          0,1,0,1,
                          0,1,0,0]);
Join(b);
```

PROGRAMME: SB
DESCRIPTION: This programme defines the s-box of BunnyTN using the non-linear functions over the field $\mathbb{E}$.
MATH: Function γ, in Section 6.
LANGUAGE: MAGMA
INPUT: A 1×4 matrix v over $\mathbb{E}$
GLOBAL ASSUMPTIONS: The field $\mathbb{E}$
OUTPUT: $\gamma(v)$

```
SB := function(v)
  local d, f1, f2, f3, f4;

  f1 := map<E -> E | x :-> x^62>;
  f2 := map<E -> E | x :-> x^5>;
  f3 := map<E -> E | x :-> x^17>;
  f4 := map<E -> E | x :-> x^62 + e^2>;
  d := Matrix(E, 1, 4, [f1(v[1]), f2(v[2]),
f3(v[3]), f4(v[4])]);

  return d;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
E4 := VectorSpace(E,4);
v := Random(E4);
SB(v);
```

PROGRAMME: SBinv
DESCRIPTION: This programme defines the inverse of the s-boxes of BunnyTN.
MATH: Function γ^{-1}, in Section 6.
LANGUAGE: MAGMA
INPUT: A 1×4 matrix v over $\mathbb{E}$
GLOBAL ASSUMPTIONS: The field $\mathbb{E}$
OUTPUT: $\gamma^{-1}(v)$

```
SBinv := function(v)
  local d, f1, f2, f3, f4;

  f1 := map<E -> E | x :-> x^62>;
  f2 := map<E -> E | x :-> x^38>;
  f3 := map<E -> E | x :-> x^26>;
  f4 := map<E -> E | x :-> (x + e^2)^62>;
  d := Matrix(E, 1, 4, [f1(v[1]), f2(v[2]),
f3(v[3]), f4(v[4])]);
```

```
10
  return d;
12 end function;
```

EXAMPLE

```
E<e> := GF(2,6);
2 E4 := VectorSpace(E,4);
v := Random(E4);
4 SBinv(v);
```

PROGRAMME: MatrixLambdaRSC
DESCRIPTION: This programme create the MDS matrix using the Reed Solomon Code.
MATH: MDS matrix, in Section 7 in Definition 17.
LANGUAGE: MAGMA
INPUT: The integer q, r, n. They are respectively the number of elements of the field, the size of our matrix λ, the first column we choose from the generating Matrix of the Reed Solomon code. You have to choose $q > 2r$ and $k < n < q - r - 1$, q shall to be a power of a prime
OUTPUT: A $r \times r$ MDS matrix over $\mathbb{F}_q$

```
MatrixLambdaRSC := function(q, k, n)
2   local RSC, A, G, lambda;

4   if 2*k ge q then
      printf"Instead␣%o,␣you␣must␣put␣a␣value␣bigger␣than
6 %o\n", q, 2*k;
      return"Errore!";
8   end if;

10  if n lt k+1 then
      printf"Instead␣%o,␣you␣must␣put␣a␣value␣bigger␣than
12 %o\n", n, k+1;
      return"Errore!";
14  end if;

16  RSC := ReedSolomonCode(q - 1, q - k);
```

```
  A := GeneratorMatrix(RSC);
18  G<e> := Parent(A[1][1]);
  lambda := Matrix(G, k, k, []);
20
  for i in [1..k] do
22    for j in [n..n+k-1] do
        lambda[i][j-n+1]:= A[i][j];
24    end for;
  end for;
26
  return lambda;
28 end function;
```

EXAMPLE

```
MatrixLambdaRSC(64,4,12);
```

16.6 Key-schedule

PROGRAMME: RotByte
DESCRIPTION: This programme makes the rotation of t bits to links of a vector v over $\mathbb{F}$.
MATH: Rotation of a vector, in Section 8.2.2 the function RB if $t = 1$.
LANGUAGE: MAGMA
INPUT: A vector $v = (v_1, \ldots, v_n)$, an integer t
OUTPUT: A vector $v = (v_{t+1}, \ldots, v_{(n-1+t) \bmod n+1})$

```
RotByte := function(v, t)
  local length, F, Vlength, D;

  length := NumberOfColumns(v);
  F := Parent(v[1]);
  Vlength := VectorSpace(F, length);

  D := Matrix(F, 1, length, []);
  for i in [1..length] do
    D[1][i] := v[(i - 1 + t) mod length + 1];
  end for;
  D := Vlength!D;

  return D;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
v := Random(V);
RotByte(v,1);
```

PROGRAMME: KeyScheduleStep1
DESCRIPTION: The first step of the key-schedule of BunnyTN.
MATH: The first step of the key-schedule of BunnyTN, in Section 8.2.1.
LANGUAGE: MAGMA
INPUT: A vector k (the master key)
FUNCTIONS: Element2Vector, Vector2Element

GLOBAL ASSUMPTIONS: The 6-dimensional vector space over $\mathbb{F}$ and the field $\mathbb{E}$
OUTPUT: $W_{-8}, \ldots, W_{-1}$

```
KeyScheduleStep1 := function(k)
  local W0, f1, f2, f3, f4;

  W0 := Matrix(GF(2), 8, 6,[]);
  f1 := map<V6 -> V6 | v :->
Element2Vector((Vector2Element(v))^62)>;
  f2 := map<V6 -> V6 | v :->
Element2Vector((Vector2Element(v))^5)>;
  f3 := map<V6 -> V6 | v :->
Element2Vector((Vector2Element(v))^17)>;
  f4 := map<V6 -> V6 | v :->
Element2Vector((Vector2Element(v))^62 + e^2)>;

  W0[1] := V6![k[1], k[2], k[3], k[4], k[5], k[6]];
  W0[2] := V6![k[7], k[8], k[9], k[10],k[11],k[12]];
  W0[3] := V6![k[13],k[14],k[15],k[16],k[17],k[18]];
  W0[4] := V6![k[19],k[20],k[21],k[22],k[23],k[24]];

  W0[5] := V6!f1(W0[1]) + W0[2];
  W0[6] := V6!f2(W0[2]) + W0[3];
  W0[7] := V6!f3(W0[3]) + W0[4];
  W0[8] := V6!f4(W0[4]) + W0[1];

  return W0;
end function;
```

EXAMPLE

```
V24 := VectorSpace(GF(2),24);
V6 := VectorSpace(GF(2),6);
v := Random(V24);
KeyScheduleStep1(v);
```

PROGRAMME: KeyScheduleStep2
DESCRIPTION: The second step of the key-schedule of BunnyTN.

MATH: The second step of the key-schedule of BunnyTN, in Section 8.2.2.
LANGUAGE: MAGMA
INPUT: The vectors $W_{-8}, \ldots, W_{-1}$ and the number of rounds N of BunnyTN
FUNCTIONS: Element2Vector, RotByte, Vector2Element
GLOBAL ASSUMPTIONS: The 6-dimensional vector space over $\mathbb{F}$ and the field $\mathbb{E}$
OUTPUT: The vectors W_i, $i = \{-8, \ldots, 4N + 8\}$

```
KeyScheduleStep2 := function(W0, NumRound)
  local num, W, f2, f3;
  NumRound := NumRound + 1;

  while (NumRound mod 5) ne 0 do
    NumRound := NumRound + 1;
  end while;

  num := NumRound * 4 + 8;
  W := Matrix(GF(2), num, 6,[]);

  f2 := map<V6 -> V6 | v :->
Element2Vector((Vector2Element(v))^5)>;
  f3 := map<V6 -> V6 | v :->
Element2Vector((Vector2Element(v))^17)>;

  for m in [1..8] do
    W[m] := W0[m];
  end for;

  for m in [9..num] do
    if (m mod 8) eq 1 then
      W[m] := W[m-8] + V6!(f2((RotByte(W[m-1],1)))) +
V6!([1,0,1,0,1,0]);
    end if;
    if (m mod 8) eq 5 then
      W[m] := W[m-8] + V6!(f3(W[m-1]));
    end if;
    if (m mod 4) ne 1 then
      W[m] := W[m-8] + W[m-1];
    end if;
  end for;
```

```
34    return W;
  end function;
```

EXAMPLE

```
1 V24 := VectorSpace(GF(2),24);
  V6 := VectorSpace(GF(2),6);
3 v := Random(V24);
  KeyScheduleStep2(KeyScheduleStep1(v),15);
```

PROGRAMME: KeyScheduleStep3
DESCRIPTION: The third step of the key-schedule of BunnyTN.
MATH: The third step of the key-schedule of BunnyTN, in Section 8.2.2.
LANGUAGE: MAGMA
INPUT: The vectors W_i, $i = \{-8, \ldots, 4N + 8\}$ and the integer *round*, which says which round is it
FUNCTIONS: Join
GLOBAL ASSUMPTIONS: The 24-dimensional vector space over $\mathbb{F}$
OUTPUT: The round key k_{round}

```
  KeyScheduleStep3 := function(W, nround)
2   local a, temp;

4   a := ((nround-1) div 5)*20 + 9 + (nround-1) mod 5;
    temp := Matrix(GF(2), 4, 6, [W[a], W[a  + 5], W[a + 10],
6 W[a + 15]]);

8   return V24!Join(temp);
  end function;
```

EXAMPLE

```
1 V24 := VectorSpace(GF(2),24);
  V6 := VectorSpace(GF(2),6);
3 v := Random(V24);
  W := KeyScheduleStep2(KeyScheduleStep1(v),15);
5 KeyScheduleStep3(W,1);
```

In the file “KeySchedule” there are all these functions in the correct order.

16.7 Diffusion

PROGRAMME: DiffTest1
DESCRIPTION: We give the number of round that we want to compute and the dimension of the message space. This give us as output the bits "accesi" (turned on) and the bits "spenti" (turned off).
MATH: It is the first algorithm explained in the Section 10
LANGUAGE: MAGMA
INPUT: The integer that say how many round we want to test and the dimension of the message space.
FUNCTIONS: Roundtest (it is our block cipher, we can avoid the key schedule and the sum with the key because we are interested at the output difference between vectors.
OUTPUT: The bits "accesi" (turned on) and the bits "spenti" (turned off).

```
DiffTest1:=function(round, value)
  local Accesi,Spenti,Vval,nullo,ei,accesi,r,v,z,Out_v,Out_z,
BitChanged,Stanchi,CoppieSpente;
  Accesi:=[];
  Spenti:=[];
  Vval := VectorSpace(GF(2),value);
  nullo:=Zero(Vval);
  for i in [1..value] do
    ei:=nullo;
    ei[i]:=1;
    accesi:={};
    r:=0;
    repeat
      r+:=1;
      v:=Random(Vval);
      z:=v+ei;
      Out_v:=Roundtest(v,round)[1];
      Out_z:=Roundtest(z,round)[1];
      BitChanged:={b: b in [1..value]|Out_v[b] ne Out_z[b]};
      accesi:=accesi join BitChanged;
    until (#accesi eq value) or r eq 200;
    Accesi[i]:=accesi;
    Spenti[i]:={1..value} diff accesi;
  end for;
  Stanchi:={};
```

```
  CoppieSpente:={};
  for i in [1..#Spenti] do
    if #Spenti[i] ne 0 then
      Stanchi:=Stanchi join {i};
      for j in Spenti[i] do
        CoppieSpente:=CoppieSpente join {[i,j]};
      end for;
    end if;
  end for;

  return([*Accesi,Spenti,Stanchi,CoppieSpente*]);
end function;
```

EXAMPLE
We have to define the typical Round of our block cipher without key-schedule in "RoundTest"; each RoundTest functions shall have a cicle for which repeats itself $round$ times.

```
DiffTest1(3,24);
```

PROGRAMME: diff1
DESCRIPTION: We give the h number of round that we want to compute and the dimension of the message space. It is said us if in h round we obtained the diffusion or not.
MATH: It is the first algorithm explained in the Section 10
LANGUAGE: MAGMA
INPUT: The integer that say how many round we want to test and the dimension of the message space.
FUNCTIONS: DiffTest1
OUTPUT: It is said us if in h round we obtained the diffusion or not.

```
diff1 := function(round, value)
  local OutTest1, Accesi1, Spenti1, a, Stanchi1, Coppiespente1;

  OutTest1:=DiffTest1(round, value);
  Accesi1:=OutTest1[1];
  Spenti1:=OutTest1[2];
  a := 0;
```

```
  for i in [1..value] do
    a := a + #Spenti1[i];
  end for;
  Stanchi1:=OutTest1[3];
  CoppieSpente1:=OutTest1[4];
  if a ne 0 then
    return "Test␣I␣negativo";
  else
    return "Test␣I␣positivo";
  end if;

end function;
```

EXAMPLE

```
diff1(3,24);
```

PROGRAMME: Bit2Index
DESCRIPTION: It is the operation which select the word to "turn on".
MATH: It is the operation which "turns on" the word in the second algorithm of the Section 10.
LANGUAGE: MAGMA
INPUT: The bit with 1 and the size of each word.
OUTPUT: The word which contains the 1 bit.

```
Bit2Index:=function(_n, bloc)
  local m, _ind ;
  m:=_n mod bloc;
  if m eq 0 then
    _ind:=_n-bloc+1;
  else
    _ind:=_n-m+1;
  end if;

  return (_ind);
end function;
```

EXAMPLE

```
Bit2Index(9,6);
```

PROGRAMME: DiffTest2
DESCRIPTION: We give the h number of round that we want to compute, the dimension of the message space and the dimension of each word. This give us as output the bits "accesi" (turned on) and the bits "spenti" (turned off).
MATH: It is the second algorithm explained in the Section 10
LANGUAGE: MAGMA
INPUT: The h number of round that we want to compute, the dimension of the message space and the dimension of each word.
FUNCTIONS: Bit2Index, MixingLayer (the mixing layer of our block cipher).
OUTPUT: The bits "accesi" (turned on) and the bits "spenti" (turned off).

```
DiffTest2:=function(round, value, bloc)
  local Vval, nullo, Accesi, Spenti, Stanchi, CoppieSpente,
Output1, AccesiML, IndAccesiML, Y1, ind, accesi, spenti;

  Vval := VectorSpace(GF(2), value);
  nullo:= Zero(Vval);
  Accesi:=[];
  Spenti:=[];
  Stanchi:={};
  CoppieSpente:={};
  for i in [1..value] do
    Output1:=nullo;
    Output1[i] := 1;
    AccesiML:=[a: a in [1..NumberOfColumns(Output1)]| Output1[a]
ne 0];
    IndAccesiML:={a:a in [1..value],b in AccesiML| a eq
Bit2Index(i, bloc)};
    for r in [1..round] do
      Y1:=nullo;
      for b in IndAccesiML do
        ind:=b;
        for t in [ind..ind+bloc-1] do
          Y1[t]:=1;
        end for;
      end for;
```

```
      Output1:=MixingLayer(Y1);
27    AccesiML:=[a: a in [1..NumberOfColumns(Output1)]| Output1[a]
ne 0];
29    IndAccesiML:={a:a in [1..value],b in AccesiML| a eq
Bit2Index(b, bloc)};
31  end for;
    accesi:={j: j in AccesiML};
33  if #accesi lt value then
      spenti:={1..value} diff accesi;
35    Spenti[i]:=spenti;
      Stanchi:=Stanchi join {i};
37    for s in spenti do
        CoppieSpente:=CoppieSpente join {[i,s]};
39    end for;
    end if;
41  Accesi[i]:=accesi;
  end for;
43
  return([*Accesi,Spenti,Stanchi,CoppieSpente*]);
45 end function;
```

EXAMPLE
We have to define the typical MixingLayer of our block cipher in the field $\mathbb{F}$.

```
1 DiffTest2(3,24,6);
```

PROGRAMME: diff2
DESCRIPTION: We give the h number of round that we want to compute, the dimension of the message space and the dimension of each word. It is said us if in h round we obtained the diffusion or not.
MATH: It is the second algorithm explained in the Section 10
LANGUAGE: MAGMA
INPUT: The h number of round that we want to compute, the dimension of the message space and the dimension of each word.
FUNCTIONS: DiffTest2
OUTPUT: It is said us if in h round we obtained the diffusion or not.

```
1 diff2 := function(round, value, bloc)
  local OutTest2, Accesi2, Spenti2, Stanchi2, Coppiespente2;
```

```
3
  if (value mod bloc) ne 0 then
5   return "ERROR!";
  end if;
7 OutTest2:=DiffTest2(round, value, bloc);
  Accesi2:=OutTest2[1];
9 Spenti2:=OutTest2[2];
  Stanchi2:=OutTest2[3];
11 CoppieSpente2:=OutTest2[4];
  if #Spenti2 ne 0 then
13   return "Test␣II␣negativo";
  else
15   return "Test␣II␣positivo";
  end if;
17
end function;
```

EXAMPLE

```
diff2(3,24,6);
```

In the files “BunnyTN - diff1” and “BunnyTN - diff2” there are all these functions applied to BunnyTN.

16.8 BunnyTN-Encode

PROGRAMME: TypicalRound
DESCRIPTION: This is the typical round of the encode of BunnyTN.
MATH: It is the typical round of the encode of BunnyTN, in Section 5.2.1.
LANGUAGE: MAGMA
INPUT: The 24-dimensional vector m_{i-1}, which is the output of the previous round; the matrix of the mixing layer lambda; the round-key k_i
FUNCTIONS: Divide, Element2Vector, Join, SB, Vector2Element
GLOBAL ASSUMPTIONS: The 6-dimensional and the 24-dimensional vector space over $\mathbb{F}$, the field $\mathbb{E}$ and the 4-dimensional vector space over $\mathbb{E}$
OUTPUT: The coded message of the round, $\rho^i_{k_i}(m_{i-1})$.

```
TypicalRound := function(a, lambda, k)

  a := Divide(a,4);
  a := E4!Matrix(E, 1, 4, [Vector2Element(a[1]),
Vector2Element(a[2]), Vector2Element(a[3]),
Vector2Element(a[4])]);
  a := SB(a);
  a := a * lambda;
  a := Matrix(GF(2), 4, 6,[Element2Vector(a[1][1]),
Element2Vector(a[1][2]), Element2Vector(a[1][3]),
Element2Vector(a[1][4])]);
  a := V24!Join(a);
  a := a  + k;

  return a;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
E := GF(2,6);
E4 := VectorSpace(E,4);
lambda := MatrixLambdaRSC(64,4,12);
v := Random(V);
k := Random(V);
TypicalRound(v, lambda, k);
```

DESCRIPTION: This is the encoding part of BunnyTN.
MATH: It is the encoding part of BunnyTN, in Section 2.3.
LANGUAGE: MAGMA
INPUT: The chosen message 24-dimensional vector m, the matrix W obtained from the second step of the key-schedule.
FUNCTIONS: KeyScheduleStep3, MatrixLambdaRSC, TipicalRound
GLOBAL ASSUMPTIONS: The 6-dimensional and the 24-dimensional vector space over $\mathbb{F}$, the field $\mathbb{E}$ and the 4-dimensional vector space over $\mathbb{E}$
OUTPUT: The coded message $c = \mathrm{BTN}_k(m)$

```
Encode := function(a, W)
  local lambda, k;

  lambda := MatrixLambdaRSC(64, 4, 12);
  k := KeyScheduleStep3(W, 1);
  a := a + k;
  for nround in [1..15] do
    k := KeyScheduleStep3(W, nround + 1);
    a := TypicalRound(a, lambda, k);
  end for;

  return a;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
E := GF(2,6);
E4 := VectorSpace(E,4);
v := Random(V);
k := Random(V);
W := KeyScheduleStep1(k);
W := KeyScheduleStep2(W, 15);
Encode(v, W);
```

PROGRAMME: BunnyTN
DESCRIPTION: This is the block cipher of BunnyTN, it has got both the encoding

part and the key-schedule.
MATH: It is our toy cipher, in Section 5.2.
LANGUAGE: MAGMA
INPUT: The chosen message 24-dimensional vector m and the master key 24-dimensional vector k
FUNCTIONS: KeyScheduleStep1, KeyScheduleStep2, Encode
GLOBAL ASSUMPTIONS: The 6-dimensional and the 24-dimensional vector space over $\mathbb{F}$, the field $\mathbb{E}$ and the 4-dimensional vector space over $\mathbb{E}$
OUTPUT: The coded message $c = \mathrm{BTN}_k(m)$

```
BunnyTN := function(m, k)
  local W;

  W := KeyScheduleStep1(k);
  W := KeyScheduleStep2(W, 15);

  return Encode(m, W);
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
E := GF(2,6);
E4 := VectorSpace(E,4);
v := Random(V);
k := Random(V);
BunnyTN(v, k);
```

In the file "BunnyTNEncode" there are all these functions in the correct order.

16.9 BunnyTN-Decode

PROGRAMME: RoundInv
DESCRIPTION: This is the typical round of the decode of BunnyTN.
MATH: It is the typical round of the decode of BunnyTN, in Section 5.2.1.
LANGUAGE: MAGMA
INPUT: The 24-dimensional vector m_{i-1}, which is the output of the previous round; the inverse matrix of the mixing layer unlambda; the round-key k_i
FUNCTIONS: Divide, Element2Vector, Join, SBInv, Vector2Element
GLOBAL ASSUMPTIONS: The 6-dimensional and the 24-dimensional vector space over $\mathbb{F}$, the field $\mathbb{E}$ and the 4-dimensional vector space over $\mathbb{E}$
OUTPUT: The coded message of the round, $(\rho^i_{k_i})^{-1}(m_{i-1})$.

```
RoundInv := function(a, unlambda, k)

  a := a + k;
  a := Divide(a,4);
  a := Matrix(E, 1, 4, [Vector2Element(a[1]),
Vector2Element(a[2]), Vector2Element(a[3]),
Vector2Element(a[4])]);
  a := E4!(a * unlambda);
  a := SBinv(a);
  a := Matrix(GF(2), 4, 6, [Element2Vector(a[1][1]),
Element2Vector(a[1][2]), Element2Vector(a[1][3]),
Element2Vector(a[1][4])]);
  a := V24!Join(a);

  return a;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
E := GF(2,6);
E4 := VectorSpace(E,4);
lambda := MatrixLambdaRSC(64,4,12);
unlambda := lambda^(-1);
v := Random(V);
k := Random(V);
```

```
RoundInv(v, lambda, k);
```

PROGRAMME: Decode
DESCRIPTION: This is the decoding of BunnyTN.
MATH: It is the decoding part of BunnyTN, in Section 2.3.
LANGUAGE: MAGMA
INPUT: The coded message 24-dimensional vector c, the matrix W obtained from the second step of the keyschedule.
FUNCTIONS: KeyScheduleStep3, MatrixLambdaRSC, Round
GLOBAL ASSUMPTIONS: The 6-dimensional and the 24-dimensional vector space over $\mathbb{F}$, the field $\mathbb{E}$ and the 4-dimensional vector space over $\mathbb{E}$
OUTPUT: The decoded message $m = \mathrm{BTN}_k^{-1}(c)$

```
Decode := function(a, W)
  local unlambda, k;

  unlambda := MatrixLambdaRSC(64, 4, 12);
  unlambda := unlambda^(-1);
  a := V24!a;
  for nround in [1..15] do
    k := KeyScheduleStep3(W, 17-nround);
    a := RoundInv(a, unlambda, k);
  end for;
  k := KeyScheduleStep3(W, 1);
  a := a + k;

  return V24!a;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
E := GF(2,6);
E4 := VectorSpace(E,4);
v := Random(V);
k := Random(V);W := KeyScheduleStep1(k);
W := KeyScheduleStep2(W, 15);
Decode(v, W);
```

PROGRAMME: BunnyTNInv
DESCRIPTION: This is the decoding of the block cipher of BunnyTN, it has both the decoding part and the keyschedule.
MATH: It is our Toy Cipher, in Section 5.2.
LANGUAGE: MAGMA
INPUT: The coded message 24-dimensional vector m and the master key 24-dimensional vector k
FUNCTIONS: KeyScheduleStep1, KeyScheduleStep2, Decode
GLOBAL ASSUMPTIONS: The 6-dimensional and the 24-dimensional vector space over $\mathbb{F}$, the field $\mathbb{E}$ and the 4-dimensional vector space over $\mathbb{E}$
OUTPUT: The decoded message $m = \mathrm{BTN}_k^{-1}(c)$

```
BunnyTNInv := function(m, k)
  local W;

  W := KeyScheduleStep1(k);
  W := KeyScheduleStep2(W, 15);

  return Decode(m, W);
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
E := GF(2,6);
E4 := VectorSpace(E,4);
v := Random(V);
k := Random(V);
BunnyTNInv(v, k);
```

In the file "BunnyTNDecode" there are all these functions in the correct order.

16.10 BruteForce

PROGRAMME: BruteForce
DESCRIPTION: It is brute force attack with BunnyTN.
MATH: This attack is known plaintext: it tests all the keys until it finds the chosen one. It is explained in Section 4.1.
INPUT: The message m and the decode message $c = \mathrm{BTN}(m, k)$
FUNCTIONS: BunnyTN
OUTPUT: The key k such that $c = \mathrm{BTN}(m, k)$

```
BruteForce := function(m, c)
  local E, E4, V6, V24;

  E<e> := GF(2,6);
  E4 := VectorSpace(E,4);
  V6 := VectorSpace(GF(2), 6);
  V24 := VectorSpace(GF(2), 24);

  for k in V24 do
    if BunnyTN(m,k) eq c then
      return k;
    end if;
  end for;

  return "Error!";
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
m := Random(V);
c := Random(V);
BruteForce(m,c);
```

16.11 For C

PROGRAMME: Bit2Integer
DESCRIPTION: This programme is to pass from a vector over $\mathbb{F}$ to an integer.
MATH: The function BIN in the Section 14.
LANGUAGE: MAGMA
INPUT: A vector $v = c_2$ where for 2 we indicate the basis of the numeral system, k is used to choice if we want to read the vector from left or from right.
OUTPUT: An integer $k = c_{10}$ where for 10 we indicate the basis of the numeral system.

```
Bit2Integer := function(v,k)
  local length, b;

  length := NumberOfColumns(v);
  b := 0;
  if k eq 0 then
    for i in [1..length] do
      b := b + IntegerRing()!(v[length+1-i])*2^(i-1);
    end for;
  else
    if k eq 1 then
        for i in [1..length] do
        b := b + IntegerRing()!(v[i])*2^(i-1);
      end for;
    else
      return"k must be equal at 0 or 1";
    end if;
  end if;

  return b;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),24);
v := Random(V);
Bit2Integer(v);
```

PROGRAMME: Integer2Bit
DESCRIPTION: This programme is to pass from an integer to a vector over $\mathbb{F}$.
MATH: The function BIN^{-1} in the Section 14.
LANGUAGE: MAGMA
INPUT: A vector $v = c_{10}$ where for 10 we indicate the basis of the numeral system
OUTPUT: An integer $k = c_2$ where for 2 we indicate the basis of the numeral system

```
Integer2Bit := function(b, length)
  local V, v;

  V := VectorSpace(GF(2), length);
  v := Matrix(GF(2), 1, length, []);

  for i in [1..length] do
    v[1][length+1-i] := IntegerRing()!b mod 2;
    b := (b - IntegerRing()!v[1][length+1-i]) div 2;
    b;
  end for;

  return V!v;
end function;
```

EXAMPLE

```
Integer2Bit(30,6);
```

PROGRAMME: CreateTableSBox
DESCRIPTION: This programme writes the non linear functions of the s-box step as tables.
MATH: We writes a non linear function of the s-box step as a table. In the Section 14.1.
LANGUAGE: MAGMA
INPUT: A function $f : \mathbb{F}^n \to \mathbb{F}^n$
FUNCTIONS: Element2Vector, Vector2Element
OUTPUT: A $2^n \times n$ matrix such that $M[(\xi^{-1}(x))_{10} + 1] = f(x)_{10}\ \forall x \in V$

```
CreateTableSBox := function(f)
  local V, num, dim, M, k, vv;

  V := Domain(f);
  num := #V;
  dim := Degree(V);
  VV := VectorSpace(GF(2), dim);
  M := Matrix(GF(2), num, dim, []);
  for v in VV do
    k := 0;
    for i in [1..dim] do
      k := k + IntegerRing()!(v[i]) * 2^(i-1);
    end for;
    vv := f(Vector2Element(v));
    vv := VV!(Element2Vector(vv));
    M[k+1] := vv;
  end for;

  return M;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
CreateTableSBox(f);
```

PROGRAMME: CreateLambdaC
DESCRIPTION: This programme converts a matrix over $\mathbb{F}_{p^q}$ in a matrix over $\mathbb{F}$.
MATH: We convert a matrix over $\mathbb{F}_{p^q}$ in a matrix over $\mathbb{F}_p$. In the Section 14.2.
LANGUAGE: MAGMA
INPUT: A $n \times m$ matrix M over $\mathbb{F}_{p^q}$
FUNCTIONS: Element2Vector
OUTPUT: A $nq \times mq$ matrix N over $\mathbb{F}$ such that $\xi(N\xi^{-1}(x)) = Mx \ \forall x \in \mathbb{F}_{p^q}^n$

```
CreateLambdaC := function(lambda)
  local F, e, degree, nrowslambda, ncolumns, nrows, V,
M, W, W2;
```

```
  F := Parent(lambda[1][1]);
  e := PrimitiveElement(F);
  degree := Degree(F);
  nrowslambda := NumberOfRows(lambda);
  ncolumns := degree * NumberOfColumns(lambda);
  nrows := degree * nrowslambda;
  V := VectorSpace(GF(2), degree * nrowslambda);
  M := Matrix(GF(2), nrows, ncolumns, []);

  for j in [1..nrowslambda] do
    for i in [1..degree] do
      W := Matrix(F, 1, nrowslambda, []);
      W[1][j] := e^(i-1);
      W := W * lambda;
      W2 := Matrix(GF(2), nrowslambda, degree, []);
      for k in [1..nrowslambda] do
        W2[k] := Element2Vector(W[1][k]);
      end for;
      W := V!(Join(W2));
      M[i + degree*(j-1)] := W;
    end for;
  end for;

  return M;
end function;
```

EXAMPLE

```
lambda := MatrixLambdaRSC(64,4,12);
CreateLambdaC(lambda);
```

16.12 Degree and Density

PROGRAMME: PolynomialCreation
DESCRIPTION: If we give a function f, this programme gives us his polynomial.
MATH: We have a Boolean function $f : \mathbb{F}^n \to \mathbb{F}$. The programme write f as the polynomial p. In the Section 11.2.
LANGUAGE: MAGMA
INPUT: A Boolean function $f : \mathbb{F}^n \to \mathbb{F}$
FUNCTIONS: Bit2Integer
OUTPUT: The polynomial p of f

```
PolynomialCreation := function(f)
  local V, dim, R, tt, X, A, b, Vdim, i, j, temp, c,
poli, poli2;

  V := Domain(f);
  dim := Dimension(V);
  R := PolynomialRing(GF(2), dim);
  tt := "x" cat IntegerToString(1);

  X := [tt];
  for val in [2..dim] do
    tt := "x" cat IntegerToString(val);
    X := Append(X, tt);
  end for;
  AssignNames(~R, X);

  A := Matrix(GF(2), #V, #V, []);
  b := Matrix(GF(2), 1, #V, []);
  Vdim := VectorSpace(GF(2), #V);

  for v in V do
    i := Bit2Integer(v,1);
    if i eq 0 then
      i := #V;
    end if;
    b[1][i] := f(v);
    for w in V do
      temp := 1;
      j := Bit2Integer(w,0);
```

```
30        if j eq 0 then
            j := #V;
32        end if;
          for k in [1..dim] do
34          temp := temp *
(IntegerRing()!v[k])^(IntegerRing()!w[k]);
36        end for;
          A[i][j] := temp;
38      end for;
    end for;
40
    b := Vdim!b;
42  c := b*Transpose(A^(-1));

44  poli := 0;
    for v in V do
46    i := Bit2Integer(v,0);
      if i eq 0 then
48      i := #V;
      end if;
50    poli2 := c[i];
      for j in [1..dim] do
52      poli2 := poli2 * R.j^(IntegerRing()!v[j]);
      end for;
54    poli := poli + poli2;
    end for;
56
    return poli;
58 end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
2 f := map<V -> GF(2)| x :-> x[1]>;
PolynomialCreation(f);
```

PROGRAMME: degree
DESCRIPTION: Let be a function f, this programme gives us the degree of f.
MATH: Let be a function f, this programme gives us the degree of f. In the Section

11.2.
LANGUAGE: MAGMA
INPUT: An invertible multi-Boolean function $f : \mathbb{F}^n \to \mathbb{F}^n$
FUNCTIONS: BooleanFunction, PolynomialCreation
OUTPUT: The degree of f.

```
degree := function(f)
  local F, V, dim, min, a;

  F := BooleanFunction(f);
  V := Domain(f);
  dim := Degree(V);
  min := dim + 1;
  for i in [1..#V-1] do
    a := Degree(PolynomialCreation(F[i]));
    if a lt min then
      min := a;
    end if;
  end for;

  return a;
end function;
```

EXAMPLE

```
E := GF(2,6);
f := map<E -> E| x :-> x^5>;
degree(f);
```

PROGRAMME: density
DESCRIPTION: Let be a function f, this programme gives us the density of f.
MATH: Let be a function f, this programme gives us the density of f. In the Section 11.2.
LANGUAGE: MAGMA
INPUT: An invertible multi-Boolean function $f : \mathbb{F}^n \to \mathbb{F}^n$
FUNCTIONS: BooleanFunction, PolynomialCreation
OUTPUT: The density of f.

```
density := function(f)
  local F, V, W, min, a;

  F := BooleanFunction(f);
  V := Domain(f);
  W := Codomain(f);
  min := #W + 1;
  for i in [1..#V-1] do
    a := #Coefficients(PolynomialCreation(F[i]));
    if a lt min then
      min := a;
    end if;
  end for;

  return min;
end function;
```

EXAMPLE

```
E := GF(2,6);
f := map<E -> E| x :-> x^5>;
density(f);
```

In the file "DegreeAndDensity" there are all these functions in the correct order.

16.13 Anti-invariance

PROGRAMME: CreateB
DESCRIPTION: It creates a basis of V such that the property of i-anti-invariance is possible (if it is possible).
MATH: It creates a basis of V as it is explained in Section 11.3.
LANGUAGE: MAGMA
INPUT: An integer i to choose i anti-invariance, a function f, the set of vectors B did not use and some value of basis in A
OUTPUT: A basis of V or an empty set

```
CreateB := function(i, f, B, A)
  local V, dim, F, D, G, W, G2, AG1, AG2;
  V := Domain(f);
  dim := Degree(V);

  for v in B do
    if i ne 1 then
      F := A join {v};
      D := $$(i-1, f, {x : x in V | (x in sub<V|F>) eq false}, F);
      if D ne {} then
        return D;
      end if;
    end if;
                 if i eq 1 then
      D := A join {v};
      G := sub<V|D>;
      W := {f(x): x in G};
      G2 := sub<V | W>;
      AG1 := {x : x in G};
      AG2 := {x : x in G2};
      if AG1 eq AG2 then
        D;
        return D;
      end if;
    end if;
  end for;

  return {};
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> V | x :-> x + V![1,0,0,1,0,1]>;
A := {Zero(V)};
B := {x : x in V | x ne 0};
CreateB(2,f,B,A);
```

PROGRAMME: Antiinvariance
DESCRIPTION: It says us if the function f is i anti-invariant or not.
MATH: It analyzes the i anti-invariance of f as explained in Section 11.3
LANGUAGE: MAGMA
INPUT: An integer i and a function f
FUNCTIONS: CreateB
OUTPUT: "OK!" if f is i anti-invariant, "NO!" although.

```
Antiinvariance := function(i, f)
  local V, A, B, A2, B2, G, G2;

  V := Domain(f);
  A := {Zero(V)};
  B := {x : x in V | x ne 0};
  A2 := CreateB(i,f,B,A);
  B2 := {f(x): x in A2};
  G := sub<V|A2>;
  G2 := sub<V|B2>;

  if G eq G2 then
    return "OK!";
  end if;
  return "NO!";
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> V | x :-> x + V![1,0,0,1,0,1]>;
```

```
Antiinvariance(2,f);
```

PROGRAMME: CreateB2
DESCRIPTION: It creates a basis of V such that the property of i strongly-anti-invariance is possible (if it is possible).
MATH: It creates a basis of V as it is explained in Section 11.3.
LANGUAGE: MAGMA
INPUT: An integer i to choose i strongly-anti-invariance, a function f, the set of vectors B did not use and some value of basis in A.

```
CreateB2 := function(i, f, B, A)
  local V, dim, F, D, G, W, G2, AG1, AG2;
  V := Domain(f);
  dim := Degree(V);

  for v in B do
    if i ne 1 then
      F := A join {v};
      D := $$(i-1, f, {x : x in V | (x in sub<V|F>) eq false}, F);
      if D ne {} then
        return D;
      end if;
    end if;
    if i eq 1 then
      D := A join {v};
      G := sub<V | D>;
      W := {f(x): x in G};
      G2 := sub<V | W>;
      DimG := Dimension(G);
      DimG2 := Dimension(G2);
      if DimG eq DimG2 then
        D;
        return D;
      end if;
    end if;
  end for;

  return {};
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> V | x :-> x + V![1,0,0,1,0,1]>;
A := {Zero(V)};
B := {x : x in V | x ne 0};
CreateB2(2,f,B,A);
```

PROGRAMME: Strongantiinvariance
DESCRIPTION: It says us if the function f is i-strong-anti-invariant or not.
MATH: It analyzes the i-strong-anti-invariance of f as explained in Section 11.3
LANGUAGE: MAGMA
INPUT: An integer i and a function f
FUNCTIONS: CreateB2
OUTPUT: "OK!" if f is i-strong-anti-invariant, "NO!" although.

```
Strongantiinvariance := function(i, f)
  local V, A, B, A2, B2, G, G2;

  V := Domain(f);
  A := {Zero(V)};
  B := {x : x in V | x ne 0};
  A2 := CreateB2(i,f,B,A);
  B2 := {f(x): x in A2};
  G := sub<V|A2>;
  G2 := sub<V|B2>;

  if Dimension(G) eq Dimension(G2) then
    return "OK!";
  end if;
  return "NO!";
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> V | x :-> x + V![1,0,0,1,0,1]>;
```

```
Strongantiinvariance(2,f);
```

16.14 Effective Non Linearity

PROGRAMME: SumDDT
DESCRIPTION: It sums the square of the entry of a square matrix.
MATH: This operation in important to approximate the effective non linearity, as explained in the Section 11.4 in the Theorem 28.
LANGUAGE: MAGMA
INPUT: A $n \times n$ square matrix DDT
OUTPUT: $\sum_{i,j} \mathrm{DDT}_{i,j}^2$

```
sumDDT := function(DDT)
  local sum, n;

  sum := 0;
  n := NumberOfColumns(DDT);
  for i in [1..n] do
    for j in [1..n] do
      sum := sum + DDT[i][j]^2;
    end for;
  end for;

  return sum;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
f := map<E -> E| x :-> x^5>;
A := DDT(f);
sumDDT(A);
```

PROGRAMME: EffectiveNonLinearity
DESCRIPTION: It gives us an approximation of the effective non linearity of a round of our block cipher.
MATH: This operation give us an approximation the effective non linearity, as explained in the Section 11.4 in the Theorem 28.
LANGUAGE: MAGMA
INPUT: An order set with the non linear functions of s-box of the block cipher φ.

FUNCTIONS: DDT, sumDDT
OUTPUT: The approximation of the Effective Non Linearity of φ

```
EffectiveNonLinearity := function(SBOX)
  local n, DDTS, dim, EF;

  n := #SBOX;
  DDTS := [];
  dim := 0;

  for i in [1..n] do
    DDTS[i] := DDT(SBOX[i]);
    dim := dim + Degree(Domain(SBOX[i]));
  end for;
  EF := 1;
  for i in [1..n] do
    EF := EF * (sumDDT(DDTS[i]));
  end for;
  EF := EF - 2^(2*dim);
  EF := -1 + 2^(-2*dim) * EF;

  return EF;
end function;
```

EXAMPLE

```
E<e> := GF(2,6);
A := [map<E -> E| x :-> x^5>, map<E -> E| x :-> x^17>];
EffectiveNonLinearity(A);
```

16.15 Proper

PROGRAMME: ProperDisp
DESCRIPTION: We see if a block between a matrix ML goes into itself or not.
MATH: It is the analysis of the wall of the property "proper" in Section 12.1.
LANGUAGE: MAGMA
INPUT: A subset I of the set $\{1, \ldots, nrows\}$ where $nrows$ is the number of the rows of matrix ML, the matrix over a field $\mathbb{K}$ ML and the order set with the elements of the field $\mathbb{K}$
OUTPUT: A couple which says us if the ML is proper and the dispersion in relation at I.

```
ProperDisp := function(I,ML,LR)
  local F, nrows, V, null, n, stop, II, ind_list, x_list, r,
X1, Y1, OnBlocks;

  F := Parent(ML[1][1]);
  nrows := NumberOfRows(ML);
  V := VectorSpace(F, nrows);
  null := Zero(V);
  n := #I;
  stop := 0;
  II := [i:i in I];
  ind_list := [0:i in [1..n]];
  for j in [1..n] do
    ind_list[j] := II[j];
  end for;
  r := 0;

  repeat
    X1 := null;
    for j in [1..n] do
      r := r + 1;
      X1[ind_list[j]] := LR[r];
    end for;
    Y1 := X1*ML;
    OnBlocks := {b:b in [1..nrows]|Y1[b] ne 0};
  until (#OnBlocks eq nrows) or (r eq (#LR div n)*n);
  if OnBlocks eq I then
    printf "STOP: the %o-th %o goes over itself: %o.\n",
```

```
n,I,OnBlocks;
    stop := 1;
  end if;

  return([*OnBlocks,stop*]);
end function;
```

EXAMPLE

```
E := GF(2,6);
A := Random(GL(4,E));
I := {1,3};
B := [Random(E): i in [1..#E]];
ProperDisp(I,A,B);
```

PROGRAMME: Proper
DESCRIPTION: This programme says us if the matrix ML is proper or not.
MATH: This programme says us if the matrix ML is proper or not. From Section 12.1.
LANGUAGE: MAGMA
INPUT: A square matrix ML.
FUNCTIONS: ProperDisp
OUTPUT: "The map is not proper" if the map is not proper, "The map is proper" if the map is proper, "ERROR" although.

```
Proper := function(ML)
  local F, nrows, LR, nibble, I_set, I_L, c, A;

  F := Parent(ML[1][1]);
  nrows := NumberOfRows(ML);
  LR := [Random(F) : x in F];
  nibble := {1..nrows};
  I_set :={i:i in Subsets(nibble,1)};
  I_L:=[i:i in I_set];
  c:=0;

  for J in I_L do
    A := J;
```

```
    repeat
15      OUT := ProperDisp(A, ML, LR);
        if OUT[2] eq 1 then
17        return "The map is not proper.";
        else
19        A := A join OUT[1];
        end if;
21    until #A eq nrows;
      c := c+1;
23  end for;

25  if c eq #I_L then
      return "The map is proper.";
27  else
      return "ERROR";
29  end if;
end function;
```

EXAMPLE

```
E := GF(2,6);
2 A := Random(GL(4,E));
Proper(A);
```

PROGRAMME: StronglyProperDisp
DESCRIPTION: We see if a block between a matrix ML goes into other wall or not.
MATH: It is the analysis of the walls for the property "proper" in Section 12.1.
LANGUAGE: MAGMA
INPUT: A subset I of the set $\{1, \ldots, nrows\}$ where $nrows$ is the number of the rows of matrix ML, the matrix over a field $\mathbb{K}$ ML and the order set with the elements of the field $\mathbb{K}$
OUTPUT: A couple that says us if the ML is strongly proper and the dispersion in relation at I.

```
1 StronglyProperDisp := function(I,ML,LR)
    local F, nrows, V, null, n, stop, II, ind_list, r,
3 X1, Y1, OnBlocks;
```

```
  F := Parent(ML[1][1]);
  nrows := NumberOfRows(ML);
  V := VectorSpace(F, nrows);
  null := Zero(V);
  n := #I;
  stop := 0;
  II := [i:i in I];
  ind_list := [0:i in [1..n]];
  for j in [1..n] do
    ind_list[j] := II[j];
  end for;
  r := 0;

  repeat
    X1 := null;
    for j in [1..n] do
      r := r+1;
      X1[ind_list[j]]:=LR[r];
    end for;
    Y1 := X1*ML;
    OnBlocks := {b:b in [1..nrows]|Y1[b] ne 0};
  until (#OnBlocks eq nrows) or (r eq (#LR div n)*n);
  if #OnBlocks eq n then
    printf "STOP: the %o-th %o goes over %o-th.\n",n,I,#OnBlocks;
    stop := 1;
  end if;

  return([*OnBlocks,stop*]);
end function;
```

EXAMPLE

```
E := GF(2,6);
A := Random(GL(4,E));
I := {1,3};
B := [Random(E): i in [1..#E]];
StronglyProperDisp(I,A,B);
```

PROGRAMME: StronglyProper

DESCRIPTION: This programme says us if the matrix ML is proper or not.
MATH: This programme says us if the matrix ML is proper or not. From Section 12.1.
LANGUAGE: MAGMA
INPUT: A square matrix ML.
FUNCTIONS: StronglyProperDisp
OUTPUT: "The map is not strongly proper" if the map is not strongly proper, "The map is strongly proper" if the map is strongly proper, "ERROR" although.

```
1  StronglyProper := function(ML)
     local MLinv, F, nrows, LR, nibble, I_set, I_L, c, t, OUT;
3
     MLinv := ML^(-1);
5    F := Parent(ML[1][1]);
     nrows := NumberOfRows(ML);
7    LR := [x : x in F];
     nibble := {1..nrows};
9    I_set :={i:i in Subsets(nibble,1)};
     I_L:=[i:i in I_set];
11   c:=0;

13   for J in I_L do
       t:=0;
15     OUT:=[*J,0*];
       repeat
17       t := t+1;
         if (t mod 2) eq 1 then
19         OUT:=StronglyProperDisp (OUT[1],ML,LR);
         else
21         OUT:=StronglyProperDisp (OUT[1],MLinv,LR);
         end if;
23       if OUT[2] eq 1 then
           return "The map is not strongly proper.";
25       end if;
       until #OUT[1] eq nrows;
27     c := c+1;
     end for;
29
     if c eq #I_L then
31     return "The map is strongly proper.";
```

```
  else
33    return "ERROR";
  end if;
35 end function;
```

EXAMPLE

```
1 E := GF(2,6);
  A := Random(GL(4,E));
3 StronglyProper(A);
```

16.16 Maximal Linear Potential

PROGRAMME: LPmax
DESCRIPTION: It calculate the $\mathrm{LP}_{\max}$.
MATH: This algorithm computes the $\mathrm{LP}_{\max}$ of a function f with the use of Hadamard Transformation (Theorem 53) in the Section 13.2.
LANGUAGE: MAGMA
INPUT: a function $f : \mathbb{F}^n \to \mathbb{F}^n$
OUTPUT: $\mathrm{LP}^f_{\max}$

```
LPmax := function(f)
  local V, Y, dim, sum, val, max;

  V := Domain(f);
  Y := {x : x in V| x ne 0};
  dim := Degree(V);
  max := 0;
  for u in Y do
    for v in V do
      sum := 0;
      for x in V do
        val := InnerProduct(u,f(x)) + InnerProduct(v,x);
        sum := sum + (-1)^IntegerRing()!val;
      end for;
      sum := sum/(2^dim);
      sum := sum^2;
      if sum gt max then
        max := sum;
      end if;
    end for;
  end for;

  return max;
end function;
```

EXAMPLE

```
V := VectorSpace(GF(2),6);
f := map<V -> V | x :-> x + V![1,0,1,0,0,1]>;
```

```
LPmax(f);
```

16.17 Affine Equivalence

PROGRAMME: Buchberger-Moeller; indeed there are other functions: "nfunctional", "HnFunc" and "HBMfunc", but they are all useless alone and fundamental for the algorithm of Buchberger-Möller. For this reason we consider it as a single programme.
DESCRIPTION: It is a programme to build the vanishing ideal from the functions of the s-boxes.
MATH: We build the vanishing ideal as it is described into the Section 11.5.2.
LANGUAGE: Singular
INPUT: A polynomial ring $\mathbb{K}[x, y]$ where $\mathbb{K}$ is a field. A function f. You have to stay alert: it is difficult to describe a function so we have built a polynomial ring with a new variable t, i.e. $\mathbb{K}[x, y, t]$. So f is a polynomial (for example t^5). When we want a precise value, we will make a substitution.
OUTPUT: The vanishing ideal I such that $V(I) = \{(x, f(x)) : x \in \mathbb{K}\}$.

16.18 NIST tests - sts.2.1.1

PROGRAMME: assess (it is created by makefile into sts.2.1.1)
DESCRIPTION: This programme examines a string with a lot of tests and it says if this string is casual or not.
MATH: This programme contains all the tests of the NIST; they are described shortle into Section 9.1.
LANGUAGE: C++
INPUT: The value n which says us how many bytes we want to examine, the name of the file which contains the string that we want examine, the choice of the tests that we want to use and the number of times that we want to repeat the tests.
OUTPUT: In the folder "experiments - AlgorithmTesting" there are the results of the tests. Each folder has his own results (for example the folder "Runs" have got the results of the "Runs test"). A summary of all the tests are in the file "finalAnalysisReport.txt".

16.19 BunnyTN - EncodeC

PROGRAMME: Encode2 compiled with Encode2.cpp, Operazioni2.cpp and Operazioni.hh
DESCRIPTION: This is BunnyTN created in C++ language for a Windows pc.
MATH: It is the encoding part of BunnyTN but written with C++. Section 5.2.

LANGUAGE: C++
INPUT: The key and the message that we want to encode, in the file "datibunny.txt".
OUTPUT: The coded message in file "outputbunny.txt".

16.20 BunnyTN - DecodeC

PROGRAMME: Decode2 compiled with Decode2.cpp, OperazioniD2.cpp, OperazioniD2.hh
DESCRIPTION: This is BunnyTN created in C++ language for a Windows pc.
MATH: It is the decoding part of BunnyTN but written with C++. Section 5.2.
LANGUAGE: C++
INPUT: The key and the coded message that we want to decode, in the file "datibunnydecode.txt".
OUTPUT: The decoded message in file "outputbunny.txt".

16.21 BunnyTN2.0.1

PROGRAMME: In the folder "Sources" there is the file "BunnyTest.cpp". This programme was created by Emanuele Bellini.
DESCRIPTION: This programme examines the encoding and the decoding of BunnyTN and AES.
MATH: It is the programme BunnyTN but written with C++. Section 5.2.
LANGUAGE: C++
INPUT: The key and the message that we want to encode or to decode. They have to put into the "main" of the function.
OUTPUT: The coded or decoded message.