Trabalho Prático 3

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

-- João Carlos Pereira Rodrigues, PG46534

Rainbow

> Rainbow belongs to the family of the multivariate public key cryptosystems, one of the main families of post-quantum cryptosystems. Rainbow was designed in 2004 by Jintai Ding and Dieter Schmidt and it is based on the Oil-Vinegar signature scheme invented by Jacques Patarin. In July 22. 2020, Rainbow was select as one of the three NIST Post-quantum signature finalists. The theoretical security of Rainbow is based on the fact that solving a set of random multivariate quadratic system is an NP-hard problem. The mathematical theory behind is the theory of multivariate polynomials -- algebraic geometry. Rainbow offers very small signatures of only a few hundred bits (only 528 bits=66 bytes for the NIST level I security), which are much shorter than those of other (post-quantum) signature schemes. Furthermore, since Rainbow uses only simple operations over small finite fields, signature generation and verification are extremely efficient.
>
> https://www.pqcrainbow.org/

Implementação: https://github.com/Crypto-TII/partial-key-exposure-attacks

In [ ]:
```
from copy import copy
from itertools import product
#from sage.functions.other import floor
#from sage.functions.log import log
#from sage.misc.misc_c import prod
#from sage.misc.functional import round
#from sage.modules.free_module_element import vector
#from sage.rings.finite_rings.finite_field_constructor import FiniteField
#from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
#from sage.structure.sequence import Sequence
from sage.all import *
from tii.asymmetric_ciphers.mpkc.utils import random_affine_map
from tii.asymmetric_ciphers.mpkc.complexities.determined_system import hybrid_approach_quadratic_system
from tii.asymmetric_ciphers.mpkc.utils import random_affine_map
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_39423/3283060390.py in <cell line: 12>()
     10 #from sage.structure.sequence import Sequence
     11 from sage.all import *
---> 12 from tii.asymmetric_ciphers.mpkc.utils import random_affine_map
     13 from tii.asymmetric_ciphers.mpkc.complexities.determined_system import hybrid_approach_quadratic_system
     14 from tii.asymmetric_ciphers.mpkc.utils import random_affine_map

ModuleNotFoundError: No module named 'tii'
```

In [ ]:
```
class Rainbow:
    def __init__(self, q, n, v):
        """
        Construct an instance of Rainbow.

        INPUT:

        - ``q`` -- order of the finite field
        - ``n`` -- no. of variables
        - ``v`` -- a list of integers denoting the size of each vinegar variables

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=16, n=10, v=[3, 6, 9])
            sage: R
            Rainbow signature over GF(16) with 10 variables and 7 polynomials
        """
        if any(number >= n for number in v):
            raise ValueError("all integers in v must be strictly less than n")

        self._base_field = FiniteField(q)
        self._v = list(v) + [n]
        self._m = n - self._v[0]
        self._u = len(self._v) - 1

        self._S = None
        self._T = None
        self._F = None
        self._ring = None
        self._P = None

    @property
    def base_field(self):
        """
        Return the base field

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=10, v=[3, 6, 9])
            sage: R.base_field
            Finite Field of size 31
        """
        return self._base_field

    @property
    def nvariables(self):
        """
        Return the number of variables

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=10, v=[3, 6, 9])
            sage: R.nvariables
            10
        """
        return self._v[-1]

    @property
    def npolynomials(self):
        """
```

```
        Return the number of polynomials

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=10, v=[3, 6, 9])
            sage: R.npolynomials
            7
        """
        return self._m

    @property
    def nlayers(self):
        """
        Return the number of layers

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=10, v=[3, 6, 9])
            sage: R.nlayers
            3
        """
        return self._u

    def inner_affine_map(self):
        """
        Return the inner affine map

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: M, v = R.inner_affine_map()
            sage: M  # random
            [17 21 16  3 19]
            [27 25 15 11  0]
            [19  3 29  3 30]
            [ 9 13 13  7 25]
            [13 14 12 29 23]
            sage: v  # random
            (7, 26, 8, 30, 0)
        """
        if self._S is not None:
            return self._S

        self._S = self._random_affine_map_(self.nvariables)
        return self._S

    def outer_affine_map(self):
        """
        Return the outer affine map

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: M, v = R.outer_affine_map()
            sage: M  # random
            [10 28 11]
            [ 5 22  5]
            [24  4 18]
            sage: v  # random
```

```
            (10, 3, 12)
        """
        if self._T is not None:
            return self._T

        self._T = self._random_affine_map_(self.npolynomials)
        return self._T

    def ring(self):
        """
        Return a polynomial ring for the polynomial system

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: R.ring()
            Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Finite Field of size 31
        """
        if self._ring is not None:
            return self._ring

        base_field = self.base_field
        n = self.nvariables

        self._ring = PolynomialRing(base_field, n, 'x', order="degrevlex")
        return self._ring

    def vars(self):
        """
        Return a tuple of variables in the polynomial ring

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: R.vars()
            (x0, x1, x2, x3, x4)
        """
        return self.ring().gens()

    def vinegar_vars_at_layer(self, l):
        """
        Return a list of vinegar variables at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: R.vinegar_vars_at_layer(0)
            [x0, x1]
            sage: R.vinegar_vars_at_layer(1)
            [x0, x1, x2, x3]

        TESTS::

            sage: R.vinegar_vars_at_layer(2)
            Traceback (most recent call last):
            ...
```

```
            ValueError: l must be in the range 0 <= l < 2
        """
        if not 0 <= l < self.nlayers:
            raise ValueError(f"l must be in the range 0 <= l < {self.nlayers}")

        vl = self.nvinegar_vars_at_layer(l)
        R = self.ring()
        return list(map(R, ["x%d" % (i) for i in range(vl)]))

    def oil_vars_at_layer(self, l):
        """
        Return a list of oil variables at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: R.oil_vars_at_layer(0)
            [x2, x3]
            sage: R.oil_vars_at_layer(1)
            [x4]

        TESTS::

            sage: R.oil_vars_at_layer(2)
            Traceback (most recent call last):
            ...
            ValueError: l must be in the range 0 <= l < 2
        """
        if not 0 <= l < self.nlayers:
            raise ValueError(f"l must be in the range 0 <= l < {self.nlayers}")

        R = self.ring()
        return list(map(R, ["x%d" % (i) for i in range(self._v[l], self._v[l + 1])]))

    def nvinegar_vars_at_layer(self, l):
        """
        Return the number of vinegar variables at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: R.nvinegar_vars_at_layer(0)
            2
            sage: R.nvinegar_vars_at_layer(1)
            4

        TESTS::

            sage: R.nvinegar_vars_at_layer(2)
            Traceback (most recent call last):
            ...
            ValueError: l must be in the range 0 <= l < 2
        """
```

```
        if not 0 <= l < self.nlayers:
            raise ValueError(f"l must be in the range 0 <= l < {self.nlayers}")
        return self._v[l]

    def noil_vars_at_layer(self, l):
        """
        Return the number of oil variables at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: R.noil_vars_at_layer(0)
            2
            sage: R.noil_vars_at_layer(1)
            1

        TESTS::

            sage: R.noil_vars_at_layer(2)
            Traceback (most recent call last):
            ...
            ValueError: l must be in the range 0 <= l < 2
        """
        if not 0 <= l < self.nlayers:
            raise ValueError(f"l must be in the range 0 <= l < {self.nlayers}")
        return self._v[l + 1] - self._v[l]

    def polynomials_at_layer(self, l):
        """
        Return a list of polynomials at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: R.polynomials_at_layer(0)  # random
            [x1^2 - x0*x2 - x1*x2 - x1*x3 - x0 + x3 + 1, -x0*x1 - x1^2 + x0*x2 + x1*x2 + x0*x3 + x1 - x2 - x3]
            sage: R.polynomials_at_layer(1)  # random
            [-x0*x1 - x1^2 - x0*x2 - x0*x3 - x1*x3 + x2*x3 - x1*x4 + x2*x4 + x3*x4 - x0 + x1 + x2 + x3 + 1]
        """
        F = self.central_map()
        begin = sum([self.noil_vars_at_layer(i) for i in range(l)])
        end = begin + self.noil_vars_at_layer(l)
        return F[begin:end]

    def npolynomials_at_layer(self, l):
        """
        Return the number of polynomials at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::
```

```
            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: R.npolynomials_at_layer(0)
            2
            sage: R.npolynomials_at_layer(1)
            1

        TESTS::

            sage: R.npolynomials_at_layer(2)
            Traceback (most recent call last):
            ...
            ValueError: l must be in the range 0 <= l < 2
        """
        return self.noil_vars_at_layer(l)

    def random_ov_polynomial_at_layer(self, l):
        """
        Return a random Oil-Vinegar polynomial at the `l`-th layer

        INPUT:

        - ``l`` -- a non-negative integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
            sage: f = R.random_ov_polynomial_at_layer(0)
            sage: f  # random
            7*x0*x1 - 3*x0*x2 + 7*x1*x2 + 9*x0*x3 + 8*x1*x3 - 7*x0 - 11*x1 + 4*x2 + 8*x3 - 6

        TESTS::

            sage: O = R.oil_vars_at_layer(0)
            sage: from itertools import combinations
            sage: all([monomial not in combinations(O, 2) for monomial in f.monomials()])
            True
        """
        if not 0 <= l < self.nlayers:
            raise ValueError(f"l must be in the range 0 <= l < {self.nlayers}")

        R = self.ring()
        base_field = self.base_field
        f = R.zero()

        Vl = self.vinegar_vars_at_layer(l)
        Ol = self.oil_vars_at_layer(l)

        for xi, xj in product(Vl, Vl):
            f += base_field.random_element() * xi * xj

        for xi, xj in product(Vl, Ol):
            f += base_field.random_element() * xi * xj

        for x in Vl + Ol:
            f += base_field.random_element() * x

        f += base_field.random_element()

        return f
```

```
    def central_map(self):
        """
        Return a list of polynomials representing the central map

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: R.central_map()  # random
            [-x0*x1 - x1*x2 + x0*x3 - x1*x3 + x0 - x2 - 1,
             x0*x1 - x0*x2 - x0*x3 + x0 - x2 + x3 + 1,
             -x0*x1 + x0*x2 - x0*x3 - x0 - x1 + x3 - x4]
        """
        if self._F is not None:
            return self._F

        F = []
        for l in range(self.nlayers):
            ml = self.npolynomials_at_layer(l)
            F += [ self.random_ov_polynomial_at_layer(l) for _ in range(ml) ]

        self._F = Sequence(F)
        return self._F

    def eval_central_map(self, v):
        """
        Return the output of evaluation `v` of the central map

        INPUT:

        - ``v`` -- a list of `self.base_field` elements

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=7, v=[2, 5])
            sage: v = R.random_vector(7)
            sage: R.eval_central_map(v)  # random
            (30, 21, 14, 13, 3)
        """
        if len(v) != self.nvariables:
            raise ValueError(f"the length of v must be equal to {self.nvariables}")

        F = self.central_map()
        y = F.subs({x: val for x, val in zip(self.vars(), v)})

        return vector(self.base_field, y)

    def preimage_central_map(self, y):
        """
        Return a preimage of vector `y`

        INPUT:

        - ``y`` -- a list of `self.base_field` elements

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: GF_3 = R.base_field
            sage: y = vector(GF_3, [2, 1, 0])
            sage: v = R.preimage_central_map(y)
```

```
            sage: v  # random
            (0, 1, 2, 1, 1)

        TESTS::

            sage: y == R.eval_central_map(v)
            True
            sage: all(R.eval_central_map(R.preimage_central_map(y)) == y for y in VectorSpace(R.base_field, R.npolynomials))
            True
            sage: R = Rainbow(q=3, n=5, v=[2, 3])
            sage: all(R.eval_central_map(R.preimage_central_map(y)) == y for y in VectorSpace(R.base_field, R.npolynomials))
            True
            sage: R= Rainbow(q=3, n=28, v=[3, 5, 10, 17, 22])
            sage: y = R.random_vector(R.npolynomials)
            sage: x = R.preimage_central_map(y)
            sage: R.eval_central_map(x) == y
            True
        """
        if len(y) != self.npolynomials:
            raise ValueError(f"the length of y must be equal to {self.npolynomials}")

        n0 = self.nvinegar_vars_at_layer(0)
        x = None
        while x is None:
            try:
                x = self._preimage_central_map_(y, list(self.random_vector(n0)))
            except ValueError:
                continue

        return vector(self.base_field, x)

    def _preimage_central_map_(self, y, x):
        """
        Return a preimage of vector `y` with some initial values `x` in the preimage

        INPUT:

        - ``y`` -- a list of `self.base_field` elements
        - ``x`` -- a list of `self.base_field` elements
        """
        if len(y) != self.npolynomials:
            raise ValueError(f"the length of y must be equal to {self.npolynomials}")

        n0 = self.nvinegar_vars_at_layer(0)
        if len(x) != n0:
            raise ValueError(f"the length of x must be equal to {n0}")

        o = [self.npolynomials_at_layer(l) for l in range(self.nlayers)]
        GF_ = self.base_field

        for l in range(self.nlayers):
            begin = sum(o[:l])
            end = begin + o[l]
            coeff_vector = y[begin:end]

            Fl = self.polynomials_at_layer(l)
            F = Sequence([f - c for f, c in zip(Fl, coeff_vector)])

            v = self.vinegar_vars_at_layer(l)
            n = self.nvinegar_vars_at_layer(l)

            Fs = F.subs({ v[i] : x[i] for i in range(n) })
            A = Fs.coefficient_matrix(sparse=False)[0]
```

```
            if self.ring().one() in Fs.monomials():
                M = A.delete_columns([A.ncols() - 1])
                v = -A.column(A.ncols() - 1)
            else:
                M = A
                v = vector(GF_, [0]*A.nrows())

            s = M.solve_right(v)

            if len(s) < self.noil_vars_at_layer(l):  # there exists a free variable
                temp = [None]*self.noil_vars_at_layer(l)
                oil_vars = self.oil_vars_at_layer(l)

                free_vars = [oil_var not in Fs.variables() for oil_var in oil_vars]
                free_var_indices = [oil_vars.index(free_var) for free_var in free_vars]
                for i in free_var_indices:
                    temp[i] = GF_.random_element()

                other_vars = Fs.variables()
                other_var_indices = [oil_vars.index(other_var) for other_var in other_vars]
                for i, val in zip(other_var_indices, s):
                    temp[i] = val

                s = copy(temp)

            x += list(s)

        return vector(GF_, x)

    def inner_affine_polynomials(self):
        """
        Return a list of polynomials representing the inner affine map

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: R.inner_affine_polynomials()  # random
            [-x0 - x3 - 1,
             -x1 + x3 + x4 - 1,
             -x0 + x1 - x2 - x3 + x4 + 1,
             x0 - x2 - x4,
             x0 - x3 + x4 - 1]
        """
        R = self.ring()
        M, v = self.inner_affine_map()
        x = vector(R, R.gens())

        return (M*x + v).list()

    def random_vector(self, n):
        """
        Return a random vector of length `n` over `self.base_field`

        INPUT:

        - ``n`` -- a positive integer

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
```

```
            sage: R.random_vector(3)  # random
            (1, 2, 0)
            sage: R.random_vector(5)  # random
            (1, 2, 1, 1, 0)
        """
        if n < 1:
            raise ValueError("n must be >= 1")

        GF_ = self.base_field
        return vector(GF_, [GF_.random_element() for _ in range(n)])

    def random_msg(self):
        """
        Return a random vector representing a message

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: R.random_msg()  # random
            (2, 0, 1)
        """
        return self.random_vector(self.npolynomials)

    def random_signature(self):
        """
        Return a random vector reprensenting a signature

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: R.random_signature()  # random
            (0, 1, 2, 2, 0)
        """
        return self.random_vector(self.nvariables)

    def sign(self, msg):
        """
        Return a signature for the given message

        INPUT:

        - ``msg`` -- a list of `self.base_field` elements

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: msg = R.random_msg()
            sage: msg  # random
            (0, 1, 0)
            sage: signature = R.sign(msg)
            sage: signature  # random
            (1, 0, 0, 0, 1)
            sage: R.is_valid_signature(signature, msg)
            True
        """
        if len(msg) != self.npolynomials:
            raise ValueError(f"msg must be of length {self.npolynomials}")

        v = vector(self.base_field, msg)
        Si, si = self.inverse_outer_affine_map()
```

```
        Fi = self.preimage_central_map
        Ti, ti = self.inverse_inner_affine_map()

        return Ti*Fi(Si * v + si) + ti

    def public_key(self):
        """
        Return a list of polynomials for the public key

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: P = R.public_key()
            sage: P  # random
            [x1*x2 - x2^2 - x0*x3 + x1*x3 + x2*x3 + x3^2 + x0*x4 + x1*x4 - x4^2 - x1 + x2 + x3 - x4,
             -x0^2 - x0*x1 + x1*x2 + x1*x3 - x2*x3 - x0*x4 + x1*x4 + x2*x4 + x0 + x2 - 1,
             -x0*x1 + x0*x2 - x1*x2 + x2^2 + x0*x3 + x1*x3 - x2*x3 + x3^2 - x0*x4 + x1*x4 - x2*x4 - x4^2 + x0 + x1 - x3 - x4]

        TESTS::

            sage: x = R.vars()
            sage: signature = R.random_signature()
            sage: msg = P.subs( {x[i] : signature[i] for i in range(R.nvariables)} )
            sage: T, t = R.inner_affine_map()
            sage: v = T*signature + t
            sage: F = R.central_map()
            sage: w = F.subs( {x[i] : v[i] for i in range(R.nvariables)} )
            sage: S, s = R.outer_affine_map()
            sage: msg == (S * vector(w) + s).list()
            True
        """
        if self._P is not None:
            return self._P

        T = self.inner_affine_polynomials()
        F = self.central_map()

        FT = []
        for f in F:
            p = 0

            for coefficient, monomial in f:
                variables = monomial.variables()
                exp_vector = monomial.exponents()[0]

                indices = list(map(lambda v : int(str(v)[1:]), variables))

                if len(indices) == 1 and exp_vector[indices[0]] == 2:
                    indices.append(indices[0])

                p += coefficient * prod([T[index] for index in indices])

            FT.append(p)

        S, s = self.outer_affine_map()

        self._P = Sequence(S * vector(FT) + s)
        return self._P

    def inverse_inner_affine_map(self):
        """
        Return the inverse of the inner affine map
```

```
        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: Ti, ti = R.inverse_inner_affine_map()
            sage: Ti  # random
            [0 0 1 1 0]
            [2 2 2 2 2]
            [1 1 1 0 1]
            [2 1 2 2 2]
            [1 2 0 2 0]
            sage: ti  # random
            (2, 2, 1, 2, 1)

        TESTS::

            sage: T, t = R.inner_affine_map()
            sage: v = VectorSpace(R.base_field, R.nvariables).random_element()
            sage: Ti*(T*v + t) + ti == v
            True
        """
        T, t = self.inner_affine_map()
        return T.inverse(), -T.inverse()*t

    def inverse_outer_affine_map(self):
        """
        Return the inverse of the outer affine map

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=3, n=5, v=[2, 4])
            sage: Si, si = R.inverse_outer_affine_map()
            sage: Si  # random
            [1 2 2]
            [1 1 2]
            [2 2 2]
            sage: si  # random
            (0, 0, 2)

        TESTS::

            sage: S, s = R.outer_affine_map()
            sage: v = VectorSpace(R.base_field, R.npolynomials).random_element()
            sage: Si*(S*v + s) + si == v
            True
        """
        S, s = self.outer_affine_map()
        return S.inverse(), -S.inverse()*s

    def is_valid_signature(self, signature, msg):
        """
        Return whether the signature is valid for the message

        INPUT:

        - ``signature`` -- a list of `self.base_field` elements
        - ``msg`` -- a list of `self.base_field` elements

        EXAMPLES::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
```

```
            sage: R = Rainbow(3, n=5, v=[2, 4])
            sage: msg = [0, 1, 2]
            sage: signature = [0, 1, 2, 1, 2]
            sage: R.is_valid_signature(signature, msg)  # random
            False

        TESTS::

            sage: R.is_valid_signature([0, 1, 2, 1], msg)
            Traceback (most recent call last):
            ...
            ValueError: signature length must be equal to 5
            sage: R.is_valid_signature(signature, [0, 1, 2, 1])
            Traceback (most recent call last):
            ...
            ValueError: message length must be equal to 3
        """
        if len(signature) != self.nvariables:
            raise ValueError(f"signature length must be equal to {self.nvariables}")

        if len(msg) != self.npolynomials:
            raise ValueError(f"message length must be equal to {self.npolynomials}")

        s = list(signature)
        m = list(msg)

        P = self.public_key()
        x = self.vars()

        w = P.subs({x[i] : s[i] for i in range(self.nvariables)})

        return m == w

    def complexity_classical_high_rank(self, use_gate_count=True):
        """
        Return the complexity of high rank attack against this instance of Rainbow using classical computer

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.complexity_classical_high_rank()
            150
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.complexity_classical_high_rank()
            410
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_classical_high_rank()
            539
        """
        return self.complexity_high_rank(use_quantum=False, use_gate_count=use_gate_count)

    def complexity_quantum_high_rank(self, use_gate_count=True):
        """
        Return the complexity of high rank attack against this instance of Rainbow using quantum computer

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.complexity_quantum_high_rank()
            86
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
```

```
            sage: R_III.complexity_quantum_high_rank()
            218
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_quantum_high_rank()
            283
        """
        return self.complexity_high_rank(use_quantum=True, use_gate_count=use_gate_count)

    def complexity_high_rank(self, use_quantum, use_gate_count):
        """
        Return the complexity of high rank attack

        INPUT:

        - ``use_quantum`` -- whether to return the complexity in quantum settings (True/False)
        - ``use_gate_count`` -- whether to return the complexity in the number of gate counts (True/False)
        """
        q = self.base_field.order()
        n = self.nvariables

        o = self.noil_vars_at_layer(self.nlayers - 1)
        if use_quantum:
            o /= 2

        nmultiplications = q**o * n**3 / 6
        if use_gate_count:
            complexity = self._ngates_(nmultiplications)
        else:
            complexity = nmultiplications

        return floor(log(complexity, base=2))

    def complexity_classical_uov(self, use_gate_count=True):
        """
        Return the complexity of UOV attack against this instance of Rainbow using classical computer

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.complexity_classical_uov()
            165
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.complexity_classical_uov()
            437
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_classical_uov()
            567

        NOTE:

            The result for type I Rainbow is different from the one in the NIST Round 3 Submission
        """
        return self.complexity_uov(use_quantum=False, use_gate_count=use_gate_count)

    def complexity_quantum_uov(self, use_gate_count=True):
        """
        Return the complexity of UOV attack against this instance of Rainbow using quantum computer

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
```

```
            sage: R_I.complexity_quantum_uov()
            95
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.complexity_quantum_uov()
            233
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_quantum_uov()
            299

        NOTE:

            The result for type I Rainbow is different from the one in the NIST Round 3 Submission
        """
        return self.complexity_uov(use_quantum=True, use_gate_count=use_gate_count)

    def complexity_uov(self, use_quantum, use_gate_count):
        """
        Return the complexity of UOV attack

        INPUT:

        - ``use_quantum`` -- whether to return the complexity in quantum settings (True/False)
        - ``use_gate_count`` -- whether to return the complexity in the number of gate counts (True/False)
        """
        q = self.base_field.order()
        n = self.nvariables
        o = self.noil_vars_at_layer(self.nlayers - 1)
        e = n - 2*o - 1

        if use_quantum:
            e /= 2

        nmultiplications = q**e * o**4
        if use_gate_count:
            complexity = self._ngates_(nmultiplications)
        else:
            complexity = nmultiplications

        return floor(log(complexity, base=2))

    def complexity_classical_direct_attack(self, use_gate_count=True):
        """
        Return the complexity of direct attack against this instance of Rainbow using classical computer

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.complexity_classical_direct_attack()  # official result with XL-Wiedemann: 164
            152
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.complexity_classical_direct_attack()  # official result with XL-Wiedemann: 234
            221
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_classical_direct_attack()  # official result with XL-Wiedemann: 285
            272
        """
        return self.complexity_direct_attack(use_quantum=False, use_gate_count=use_gate_count)
```

```
    def complexity_quantum_direct_attack(self, use_gate_count=True):
        """
        Return the complexity of direct attack against this instance of Rainbow using classical computer

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.complexity_quantum_direct_attack()  # official result with XL-Wiedemann: 122
            111
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.complexity_quantum_direct_attack()  # official result with XL-Wiedemann: 200
            187
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_quantum_direct_attack()  # official result with XL-Wiedemann: 243
            230
        """
        return self.complexity_direct_attack(use_quantum=True, use_gate_count=use_gate_count)

    def complexity_direct_attack(self, use_quantum, use_gate_count):
        """
        Return the complexity of direct attack

        INPUT:

        - ``use_quantum`` -- whether to return the complexity in quantum settings (True/False)
        - ``use_gate_count`` -- whether to return the complexity in the number of gate counts (True/False)
        """
        q = self.base_field.order()
        m = self.npolynomials

        nmul = hybrid_approach_quadratic_system(q, m, use_quantum=use_quantum)
        if use_gate_count:
            complexity = self._ngates_(nmul)
        else:
            complexity = nmul

        return floor(log(complexity, base=2))

    def complexity_classical_minrank(self, use_gate_count=True):
        """
        Return the complexity of minrank attack against this instance of Rainbow

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)

        NOTE:

            This function is currently implemented for 2-layer Rainbow based on the complexity described in (23) and
            (24) of https://arxiv.org/pdf/2002.08322.pdf

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.complexity_classical_minrank(use_gate_count=False)  # official result: 162
            160
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
```

```
            sage: R_III.complexity_classical_minrank(use_gate_count=False)  # official result: 228
            227
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_classical_minrank(use_gate_count=False)  # official result: 296
            295
        """
        if self.nlayers != 2:
            raise ValueError("minrank complexity is only implemented for 2-layer Rainbow")

        from sage.functions.other import binomial

        o1 = self.noil_vars_at_layer(0)
        o2 = self.noil_vars_at_layer(1)
        v1 = self.nvinegar_vars_at_layer(0)

        K = o2 + 1
        r = v1 + o1
        m = self.nvariables

        def is_condition_satisfied(b, n):
            return binomial(n, r) * binomial(K + b - 1, b) - 1 <= \
                   sum(
                       (-1)**(i + 1) * binomial(n, r + i) * binomial(m + i - 1, i) * binomial(K + b - i - 1, b - i)
                       for i in range(1, b + 1)
                   )

        min_nmul = 2 ** 512
        for b in range(1, r + 2):
            for n in range(r + b, self.nvariables):
                if not is_condition_satisfied(b, n):
                    continue
                nmul = K * (r + 1) * (binomial(n, r) * binomial(K + b - 1, b))**2
                min_nmul = min(nmul, min_nmul)

        if use_gate_count:
            complexity = self._ngates_(min_nmul)
        else:
            complexity = min_nmul

        return floor(log(complexity, base=2))

    def complexity_quantum_minrank(self, use_gate_count=True):
        """
        Return the complexity of minrank attack against this instance of Rainbow in quantum settings

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)
        """
        return self.complexity_classical_minrank(use_gate_count=use_gate_count)

    def complexity_classical_rbs(self, use_gate_count=True):
        """
        Return the complexity of RBS (Rainbow Band Separation) attack against this instance of Rainbow

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
```

```
            sage: R_I.complexity_classical_rbs()  # long time ; official result: 147
            146
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.complexity_classical_rbs()  # long time
            217
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.complexity_classical_rbs()  # long time
            281
        """
        if self.nlayers != 2:
            raise ValueError("minrank complexity is only implemented for 2-layer Rainbow")

        v1, o1 = self.nvinegar_vars_at_layer(0), self.noil_vars_at_layer(0)
        o2 = self.noil_vars_at_layer(1)

        nx = v1 + o1
        ny = o2

        mx = o1 + o2
        mxy = self.nvariables - 1

        from sage.rings.power_series_ring import PowerSeriesRing
        from sage.rings.rational_field import QQ

        max_prec = 50
        R = PowerSeriesRing(QQ, names=['t', 's'], default_prec=max_prec)
        t, s = R.gens()
        series_0 = 1 / ((1 - t)**(nx + 1) * (1 - s)**(ny + 1))
        series_1 = ((1 - t**2)**mx * (1 - s*t)**mxy) / ((1 - t)**(nx + 1) * (1 - s)**(ny + 1))
        coefficients_0 = series_0.coefficients()

        min_nmul = 2 ** 512
        for deg in range(1, max_prec):
            f = series_1[deg]
            neg_monomials_f = [monomial for (monomial, coeff) in f.coefficients().items() if coeff < 0]
            for monomial in neg_monomials_f:
                c = coefficients_0[monomial]
                nmul = 3 * c ** 2 * (nx + 1) * (ny + 1)
                min_nmul = min(nmul, min_nmul)

            if len(neg_monomials_f) > 0:
                break

        if use_gate_count:
            complexity = self._ngates_(min_nmul)
        else:
            complexity = min_nmul

        return floor(log(complexity, base=2))

    def complexity_quantum_rbs(self, use_gate_count=True):
        """
        Return the quantum complexity of RBS (Rainbow Band Separation) attack against this instance of Rainbow

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)
        """
        return self.complexity_classical_rbs(use_gate_count=use_gate_count)

    def security_level_classical(self, use_gate_count=True):
        """
        Return the security level of Rainbow in classical setting
```

```
        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.security_level_classical()  # long time
            146
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.security_level_classical()  # long time
            217
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.security_level_classical()  # long time
            272
        """
        sec_level = min(self.complexity_classical_direct_attack(use_gate_count=use_gate_count),
                        self.complexity_classical_minrank(use_gate_count=use_gate_count),
                        self.complexity_classical_high_rank(use_gate_count=use_gate_count),
                        self.complexity_classical_uov(use_gate_count=use_gate_count),
                        self.complexity_classical_rbs(use_gate_count=use_gate_count))

        return sec_level

    def security_level_quantum(self, use_gate_count=True):
        """
        Return the security level of Rainbow in quantum setting

        INPUT:

        - ``use_gate_count`` -- return the results in terms of the number of gate (default: True)

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R_I = Rainbow(q=16, n=100, v=[36, 68])
            sage: R_I.security_level_quantum()  # long time
            86
            sage: R_III = Rainbow(q=256, n=148, v=[68, 100])
            sage: R_III.security_level_quantum()  # long time
            187
            sage: R_V = Rainbow(q=256, n=196, v=[96, 132])
            sage: R_V.security_level_quantum()  # long time
            230
        """
        sec_level = min(self.complexity_quantum_direct_attack(use_gate_count=use_gate_count),
                        self.complexity_quantum_minrank(use_gate_count=use_gate_count),
                        self.complexity_quantum_high_rank(use_gate_count=use_gate_count),
                        self.complexity_quantum_uov(use_gate_count=use_gate_count),
                        self.complexity_quantum_rbs(use_gate_count=use_gate_count))

        return sec_level

    def _random_affine_map_(self, n):
        """
        Return a random invertible affine map

        TESTS::

            sage: from tii.asymmetric_ciphers.mpkc.rainbow import Rainbow
            sage: R = Rainbow(q=31, n=5, v=[2, 4])
```

```
            sage: M2, v2 = R._random_affine_map_(2)
            sage: M2.is_invertible()
            True
            sage: len(v2) == 2
            True
            sage: M2.dimensions()
            (2, 2)
        """
        return random_affine_map(self.base_field, n)

    def _ngates_(self, nmul):
        """
        Return the number of gates for a given number of field multiplication

        INPUT:

        - ``nmul`` -- number of multiplications
        """
        q = self.base_field.order()
        return nmul * (2 * log(q, 2)**2 + log(q, 2))

    def __repr__(self):
        q = self.base_field.order()
        n = self.nvariables
        m = self.npolynomials

        return f"Rainbow signature over GF({q}) with {n} variables and {m} polynomials"


def generate_instances(sec_level_classical, sec_level_quantum, min_q=2, max_q=256, min_nvars=1, max_nvars=196):
    """
    Generate instances of Rainbow satisfying the security level

    INPUT:

    - ``sec_level_classical`` -- the classical security level
    - ``sec_level_quantum`` -- the quantum security level
    - ``min_q`` -- minimum order of the finite field (default: 2)
    - ``max_q`` -- maximum order of the finite field (default: 256)
    - ``min_nvars`` -- minimum no. of variables (default: 1)
    - ``max_nvars`` -- maximum no. of variables (default: 196)

    EXAMPLES::

        sage: from tii.asymmetric_ciphers.mpkc.rainbow import generate_instances
        sage: instances = generate_instances(sec_level_classical=25, sec_level_quantum=20, min_nvars=9, max_nvars=10, min_q=251)  # long time
        sage: len(instances) # long time
        36
        sage: instances[0]  # long time
        Rainbow signature over GF(251) with 9 variables and 8 polynomials
    """
    from sage.arith.misc import is_prime_power

    rainbow_instances = []
    for q in range(min_q, max_q + 1):
        if not is_prime_power(q):
            continue

        for nvars in range(min_nvars, max_nvars + 1):
            for nvinegar_vars0 in range(1, nvars):
                for nvinegar_vars1 in range(nvinegar_vars0 + 1, nvars):
                    R = Rainbow(q=q, n=nvars, v=[nvinegar_vars0, nvinegar_vars1])
                    if R.security_level_classical() >= sec_level_classical and \
```

```
                            R.security_level_quantum() >= sec_level_quantum:
                        rainbow_instances.append(R)

    return rainbow_instances
```

In [ ]:
```
R = Rainbow(q=3, n=5, v=[2, 4])
msg = R.random_msg()

msg  # random
signature = R.sign(msg)
signature  # random
R.is_valid_signature(signature, msg)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_39423/1482578122.py in <cell line: 5>()
      3
      4 msg  # random
----> 5 signature = R.sign(msg)
      6 signature  # random
      7 R.is_valid_signature(signature, msg)

/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_39423/3435711892.py in sign(self, msg)
    628
    629         v = vector(self.base_field, msg)
--> 630         Si, si = self.inverse_outer_affine_map()
    631         Fi = self.preimage_central_map
    632         Ti, ti = self.inverse_inner_affine_map()

/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_39423/3435711892.py in inverse_outer_affine_map(self)
    740             True
    741         """
--> 742         S, s = self.outer_affine_map()
    743         return S.inverse(), -S.inverse()*s
    744

/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_39423/3435711892.py in outer_affine_map(self)
    130             return self._T
    131
--> 132         self._T = self._random_affine_map_(self.npolynomials)
    133         return self._T
    134

/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_39423/3435711892.py in _random_affine_map_(self, n)
   1204             (2, 2)
   1205         """
-> 1206         return random_affine_map(self.base_field, n)
   1207
   1208     def _ngates_(self, nmul):

NameError: name 'random_affine_map' is not defined
```