Teeny-Tiny Multivariate Polynomial Public Key (MPPK)

Work in Progress

Author: Michel Barbeau

Version: December 15, 2021

```
clear;
```

Reference:

Randy Kuang, Michel Barbeau and Maria Perepechaenko, A New Quantum Safe Multivariate Public-keyCryptosystem Over Large Prime Galois Fields, October 2021.

Key Pair Construction

Base Polynomial $f(x_0, x_1, \ldots, x_m)$

```
% Galois Field chracteristic
p = 7;
% number of noise variables
m = 2;
% degree of base polynomial
n = 1;
% degree of multiplier polynomials
lambda = 1; % linear
% upper limits
ell = [ 1 1 ];
% randomly generate coefficients for f()
c = randi([0 p-1], n+1, ell(1)+1, ell(2)+1);
```

$c_{000}, c_{001}, c_{010}$ and c_{011}

```
disp([c(1,1,1) c(1,1,2) c(1,2,1) c(1,2,2)]);
```

```
     1     3     2     0
```

$c_{100}, c_{101}, c_{110}$ and c_{011}

```
disp([c(2,1,1) c(2,1,2) c(2,2,1) c(2,2,2)]);
```

```
     4     5     6     4
```

Mutliplier ploynomials $g(x_0)$ and $h(x_0)$

```
% randomly generate coefficients of g()
g = randi([0 p-1], 1,lambda+1);
```

g_0 and g_1

```
disp(g)
```

```
    4     5
```

```
% randomly generate coefficients of h()
h = randi([0 p-1], 1, lambda+1);
```

h_0 and h_1

```
disp(h)
```

```
    4     6
```

Product polynomials $\phi(x_0, x_1 \ldots, x_m)$ and $\psi(x_0, x_1 \ldots, x_m)$

```
% init \phi to zeros
phi = zeros(n+lambda+1, ell(1)+1, ell(2)+1);
for i=0:n+lambda
     % \phi(i)
     for j=(i-lambda):i
         if j>=0 && j<=n
            phi(i+1,:,:) = phi(i+1,:,:) + c(j+1,:,:).*g((i-j)+1);
         end
     end
end
phi = mod(phi,p);
```

$\phi_{000}, \phi_{001}, \phi_{010}$ and ϕ_{011}

```
disp([phi(1,1,1) phi(1,1,2) phi(1,2,1) phi(1,2,2)]);
```

```
    4     5     1     0
```

$\phi_{100}, \phi_{101}, \phi_{110}$ and ϕ_{111}

```
disp([phi(2,1,1) phi(2,1,2) phi(2,2,1) phi(2,2,2)]);
```

```
    0     0     6     2
```

$\phi_{200}, \phi_{201}, \phi_{210}$ and ϕ_{211}

```
disp([phi(3,1,1) phi(3,1,2) phi(3,2,1) phi(3,2,2)]);
```

```
    6     4     2     6
```

```
% init \psi to zeros
psi = zeros(n+lambda+1, ell(1)+1, ell(2)+1);
for i=0:n+lambda
     % \psi(i)
     for j=(i-lambda):i
         if j>=0 && j<=n
            psi(i+1,:,:) = psi(i+1,:,:) + c(j+1,:,:).*h((i-j)+1);
         end
     end
end
% \psi(n+lambda)
```

```
psi = mod(psi,p);
```

$\psi_{000}, \psi_{001}, \psi_{010}$ and ψ_{011}

```
disp([psi(1,1,1) psi(1,1,2) psi(1,2,1) psi(1,2,2)]);
```

```
     4     5     1     0
```

$\psi_{100}, \psi_{101}, \psi_{110}$ and ψ_{111}

```
disp([psi(2,1,1) psi(2,1,2) psi(2,2,1) psi(2,2,2)]);
```

```
     1     3     1     2
```

$\psi_{200}, \psi_{201}, \psi_{210}$ and ψ_{211}

```
disp([psi(3,1,1) psi(3,1,2) psi(3,2,1) psi(3,2,2)]);
```

```
     3     2     1     3
```

Private key $\vec{g}$, $\vec{h}$ and R

```
R = randi([0 p-1],1);
```

Public key $\vec{\Phi}$, $\vec{\Psi}$ and $N(x_1, \ldots, x_m)$

```
%Phi = phi(2:(n+lambda+1),:,:)
%Psi = psi(2:(n+lambda+1),:,:)
% N0 = R*c(1,:,:); Wrong!
N0 = R * [c(1,1,1) c(1,1,2) c(1,2,1) c(1,2,2)];
```

Encryption

Random secret s

```
s = randi([0 p-1],1);
```

Noise variables $v_1, \ldots, v_m$

```
v = randi([0 p-1],1,m);
```

Evaluate $\Phi = \Phi(s, v_1, \ldots, v_m)$, $\Psi = \Psi(s, v_1, \ldots, v_m)$ and $N = N(v1, ..., vm)$

```
Phi = 0;
Psi = 0;
for i=1:n+lambda
    for j1=0:ell(1)
        for j2=0:ell(2)
            Phi = Phi + phi(i+1,j1+1,j2+1)*v(1)^j1*v(2)^j2*s^i;
            Psi = Psi + psi(i+1,j1+1,j2+1)*v(1)^j1*v(2)^j2*s^i;
        end
    end
```

```
end
Phi = mod(Phi,p);
Psi = mod(Psi,p);
% noise
N = 0;
for j1=0:ell(1)
    for j2=0:ell(2)
        N = N + N0(1,2*j1+j2+1)*v(1)^j1*v(2)^j2;
    end
end
N = mod(N,p);
%disp(Phi)
%disp(Psi)
%disp(N)
```

Decryption

Evaluate $\phi(s, v_1, \ldots, v_m)$ and $\psi(s, v_1, \ldots, v_m)$

```
if ~R
    fprintf('*** Variable R is null\n')
    return;
end
phiEval = mod(g(1) * modinv(R,p) * N + Phi, p);
psiEval = mod(h(1) * modinv(R,p) * N + Psi, p);
if ~psiEval
    fprintf('*** Variable psiEval is null\n')
    return;
end
k = mod(phiEval * modinv(psiEval,p), p);

if ~mod(g(2) - k*h(2),p)
    fprintf('*** Denominator g(2) - k*h(2) is null\n')
    return;
end
news = (k*h(1) - g(1)) * modinv(g(2) - k*h(2), p);
fprintf('Secret is %d, Decrypted value is %d\n', s, mod(news, p));
```

```
Secret is 1, Decrypted value is 1
```

Calculation of Modular Multiplicatve Inverse

Let x be an element of $GF(p)$, its mutiplicative inverse $xinv$ modulo p is such that mod(x*xinv,p) == 1.

```
function xinv = modinv(x,p)
% % modinv: mutiplicative modular inverse of X, mod p
% usage: y = modinv(x,p)
%
% arguments: (input)
%  x - integer(s) to compute the modular inverse in the field of integers
%      for some modular base b.
%
%      x may be scalar, vector or array
```

```
%  p - integer modulus. SCALAR only.
%      When p is not a prime number, then some numbers will not have a
%      multiplicative inverse.
%
% arguments: (output)
%  xinv - an array of the same size and shape as X, such that
%      mod(x.*xinv,p) == 1
%
%      In those cases where x does not have a multiplicative inverse in the
%      field of integers modulo p, then xinv will be returned as a NaN.
%
% Examples:
% % In the field of integers modulo 12, only 1,5,7, and 11 have a
% % multiplicative inverse. As it turns out, they are all their own inverses.
%
%  xinv = modinv(0:11,12)
%  xinv =
%   NaN   1   NaN   NaN   NaN    5   NaN    7   NaN   NaN   NaN   11
%
% % In the field generated by modular base 7 (which is prime) only 0 will
% % lack a modular multiplicative inverse.
%
%  xinv = modinv(0:6,7)
%  xinv =
%   NaN     1     4     5     2     3     6
%
% % Works for large (symbolic) integers.
%
%  p = sym('1243435434354523534525 3')
%  p =
%  12434354343545235345253
%
%  modinv(2,p)
%  ans =
%  6217177171772617672627
%
% See also: gcd, sqrtmodp
%
% Author: John D'Errico
% Creation date: 1/2/2020
    if numel(p) ~= 1
        error('p must be a scalar')
    end
    % pre-allocate xinv as NaN in case some elements of x have no inverse
    xinv = NaN(size(x));
    % if p is symbolic, then Xinv should also be symbolic.
    if isa(p,'sym')
        xinv = sym(xinv);
    end
    % all the hard work will be done by gcd.
    [G,C] = gcd(x,p);
    % if G is not equal to 1, then no solution exists.
    k = G == 1;
    xinv(k) = mod(C(k),p);
```

```
end
```