From: Fre <frederik.vercauteren@gmail.com>
Sent: Thursday, January 11, 2018 3:24 PM
To: pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: OFFICIAL COMMENT: Giophantus

Dear designers of Giophantus, dear all,

Below we describe a very easy distinguishing attack on the Giophantus encryption scheme that breaks the claimed IND-CPA security (in less time than one decryption). The attack simply exploits the fact that for the chosen base ring, evaluation at 1 is a ring homomorphism to Z_q. The attack is well known in other settings such as NTRU or RLWE.

The good news is that the attack can be thwarted by switching to a different base ring R_q.

The scheme can be summarized as follows:

- Define the polynomial ring R_q = Z_q[t]/(t^n-1) where q = 2^31-1 and for the highest security level we have n = 2267.

- The public key defines a plane over R_q:

X(x,y) = f10*x + f01*y + f00 with f10, f01, f00 in R_q

- The private key is a "small" point in R_q^2 on this plane, i.e.

priv key = (ux(t), uy(t)) in R_q^2 and X(ux(t), uy(t)) = 0

and all coefficients are smaller than a parameter l, which is equal to 4 in practice.

- Encryption first takes a message M of length 2*n bits and creates an element in R_q

m(t) = m0 + m1*t + ... m_(n-1)*t^(n-1)

with the mi in [0..4[corresponding to 2 consecutive bits.

- Encryption is then defined as

Enc(m(t)) = m(t) + X(x,y)*r(x,y) + l*e(x,y)

where r(x,y) is a random polynomial of total degree 1, and e(x,y) a degree 2 polynomial with small coefficients (for the chosen parameters, infinity norm is < 4).

The attack then proceeds as follows:

- Evaluation at t = 1 defines a ring homomorphism from R_q to Z_q

- If we apply this to the public key, we obtain a plane over Z_q of the form

X1(x,y) = f10(1)*x + f01(1)*y + f00(1) mod q

Example: for the first case in the KAT file PQCencryptKAT_1134.rsp for the highest security level (256 bits) we obtain after evaluation in 1 the equation

X1(x,y) := 683072552*x + 288881024*y + 1624678229 mod (2^31-1).

- The small solution ux(t), uy(t) evaluates to a small solution in Z_q^2, which can be easily found by running through all possibilities for ux(1) (which is smaller than 4*n) and solving for uy(1) and verifying that uy(1) is also small.

Example: continuing the example above, we easily find the solution (ux(1), uy(1)) := (3355, 3383), which indeed has infinity norm smaller than 4*n = 9068.

- We can now evaluate the coefficients of the ciphertext c(x,y) at 1, and evaluate in the above small solution (ux(1), uy(1)) to recover m(1) mod l, in this case m(1) mod 4.

- Since it is possible to compute m(1) mod 4, it is possible to distinguish between the encryption of 2 chosen messages m1(t) and m2(t) such that m1(1) != m2(1) mod 4, which breaks the IND-CPA claim.

- The attack can be thwarted by using a better R_q such as for RLWE.

Best regards,

Wouter Castryck and Frederik Vercauteren

From: Jacob Alperin-Sheriff <jacobmas@gmail.com>
Sent: Thursday, January 11, 2018 5:14 PM
To: Fre
Cc: pqc-comments; pqc-forum
Subject: Re: [pqc-forum] OFFICIAL COMMENT: Giophantus

(Do you have a script? If not, we can [eventually] write one).

On Thu, Jan 11, 2018 at 3:24 PM, Fre <frederik.vercauteren@gmail.com> wrote:

> Dear designers of Giophantus, dear all,
>
> Below we describe a very easy distinguishing attack on the Giophantus encryption
> scheme that breaks the claimed IND-CPA security (in less time than one decryption).
> The attack simply exploits the fact that for the chosen base ring, evaluation at 1 is a
> ring homomorphism to Z_q. The attack is well known in other settings such as NTRU or RLWE.
>
> The good news is that the attack can be thwarted by switching to a different base ring R_q.
>
> The scheme can be summarized as follows:
>
> - Define the polynomial ring R_q = Z_q[t]/(t^n-1) where q = 2^31-1 and for the
> highest security level we have n = 2267.
>
> - The public key defines a plane over R_q:
>
> X(x,y) = f10*x + f01*y + f00 with f10, f01, f00 in R_q
>
> - The private key is a "small" point in R_q^2 on this plane, i.e.
>
> priv key = (ux(t), uy(t)) in R_q^2 and X(ux(t), uy(t)) = 0
>
> and all coefficients are smaller than a parameter l, which is equal to 4 in practice.
>
> - Encryption first takes a message M of length 2*n bits and creates an element in R_q
>
> m(t) = m0 + m1*t + ... m_(n-1)*t^(n-1)
>
> with the mi in [0..4[corresponding to 2 consecutive bits.
>
> - Encryption is then defined as
>
> Enc(m(t)) = m(t) + X(x,y)*r(x,y) + l*e(x,y)
>
> where r(x,y) is a random polynomial of total degree 1, and e(x,y) a degree 2 polynomial
> with small coefficients (for the chosen parameters, infinity norm is < 4).
>
> The attack then proceeds as follows:

From: koichiro.akiyama@toshiba.co.jp
Sent: Friday, January 12, 2018 6:48 AM
To: frederik.vercauteren@gmail.com; pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: RE: [pqc-forum] OFFICIAL COMMENT: Giophantus

Dear Frederik Vercauteren and Wouter Castryck,

Thank you for pointing out the attack against IND-CPA for Giophantus.

And we appreciate that you suggested an countermeasure switching R_q to that of RLWE employed as well. According to your comment, we are going to consider some secure and suitable R_q setting.

This is just note to thank you.

Koichiro Akiyama

From: Fre [mailto:frederik.vercauteren@gmail.com]
Sent: Friday, January 12, 2018 5:24 AM
To: pqc-comments@nist.gov
Cc: pqc-forum@list.nist.gov
Subject: [pqc-forum] OFFICIAL COMMENT: Giophantus

Dear designers of Giophantus, dear all,

Below we describe a very easy distinguishing attack on the Giophantus encryption scheme that breaks the claimed IND-CPA security (in less time than one decryption). The attack simply exploits the fact that for the chosen base ring, evaluation at 1 is a ring homomorphism to Z_q. The attack is well known in other settings such as NTRU or RLWE.

The good news is that the attack can be thwarted by switching to a different base ring R_q.

The scheme can be summarized as follows:

- Define the polynomial ring R_q = Z_q[t]/(t^n-1) where q = 2^31-1 and for the highest security level we have n = 2267.

- The public key defines a plane over R_q:

X(x,y) = f10*x + f01*y + f00 with f10, f01, f00 in R_q

- The private key is a "small" point in R_q^2 on this plane, i.e.

priv key = (ux(t), uy(t)) in R_q^2 and X(ux(t), uy(t)) = 0

From: Ward Beullens <ward@beullens.com>
Sent: Monday, June 11, 2018 8:43 AM
To: pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: OFFICIAL COMMENT: Giophantus
Attachments: Giophantus_weak_key.sage; Giophantus.sage; Giophantus.pdf

Dear All,

In January Wouter and Fré posted a distinguishing attack on Giophantus that exploits that evaluation at 1 is a ring homomorphism from R_q to ZZ_q.

At the conference in April, the designers claimed that the attack does not work because the evaluation of the noise terms at one is usually larger than q.

We did some experiments and we found that the evaluation of the noise terms at one is still small enough to perform a distinguishing attack with a significant advantage. For example, for a typical public key for security level 1, we can distinguish ciphertexts with an advantage of about 90%. Moreover, we found that for a sizeable fraction of public keys a distinguishing attack with an advantage close to 1 is possible.

We have attached a more detailed description of the attack and the Sage code for our experiments.

All the best,
Ward, Wouter and Fré

Giophantus.sage

```
n = 1201
#n = 1733;
#n = 2267;

q = 2^31-1;
l = 4;

K = GF(q);
PR.<t> = PolynomialRing(K);
R = PR.quotient(PR.ideal([t^n-1]));
PR2.<x,y> = PolynomialRing(R,2);

def SamplePoly():
    p = R.zero();
    for i in range(0,n):
        p += ZZ.random_element(0,q)*t^i;
    return p;

def SampleSmallPoly():
    sp = R.zero();
    for i in range(0,n):
        sp += ZZ.random_element(0,4)*t^i;
    return sp;

def keyGen():
    ux = SampleSmallPoly();
    uy = SampleSmallPoly();
    X10 = SamplePoly();
    X01 = SamplePoly();
    X00 = -X10*ux - X01*uy;
    X = X00 + X10*x + X01*y;
    return (ux,uy,X00,X10,X01);

def EvalAt1(X):
    return X.lift().substitute({t:1});

def FindSmallSolution(key):
    ux,uy,X00,X10,X01 = key;
    X00e = EvalAt1(X00);
    X10e = EvalAt1(X10);
    X01e = EvalAt1(X01);
    for a in range(0,4*n):
     uxe = a;
        uye = (-X00e -X10e*uxe)/X01e;
        if uye<4*n:
            return(uxe,uye);
     uye = a;
        uxe = (-X00e -X01e*uxe)/X10e;
        if uxe<4*n:
            return(uxe,uye);
```

```
def EncryptZero(key):
    ux,uy,X00,X10,X01 = key;
    Ctx = X00 + X10*x + X01*y;
    R = SamplePoly() + SamplePoly()*x + SamplePoly()*y;
    Ctx = Ctx*R;
    for i in range(0,3):
        for j in range(0,3-i):
            Ctx += 4*SampleSmallPoly()*x^i*y^j;
    return Ctx;

def Decrypt(Ctx, key):
    ux,uy,X00,X10,X01 = key;
    eval = Ctx.substitute({x:ux,y:uy});
    return eval;

def EvalCtx(Ctx,uxe,uye):
    coefs = Ctx.coefficients();
    monoms= Ctx.monomials();
    eval = K.zero();
    for i in range(0,len(coefs)):
        eval += EvalAt1(coefs[i])*monoms[i].substitute({x:uxe,y:uye});
    return eval;

def MostLikelyEval(uxe,uye):
    mean =
6*n*(1+int(uxe)+int(uye)+int(uxe)^2+int(uye)^2+int(uye)*int(uxe));
    variance =
20*n*(1+int(uxe)^2+int(uye)^2+int(uxe)^4+int(uye)^4+int(uye)^2*int(uxe
)^2);
    stdev = sqrt(variance);
    T = RealDistribution('gaussian',stdev);
    probabilities = [0,0,0,0];
    for i in range(0,2*mean,q):
        prob =
T.cum_distribution_function(i+q-mean)-T.cum_distribution_function(i-me
an);
        probabilities[int(i*q%4)] += prob;
    sp = sorted(probabilities);
    print("estimated distribution = "+str(probabilities));
    print("estimated advantage = "+str(sp[3]+sp[2]-sp[1]-sp[0]));
    return probabilities;

key = keyGen();
uxe , uye = FindSmallSolution(key);
var = int(uxe)^4+int(uye)^4+int(uxe)^2*int(uye)^2;

print("solution = "+str((uxe,uye)));
print("stddev/q = "+str(float(sqrt(20*n*var)/q)));

MostLikelyEval(uxe,uye)
```

```
List = []
for i in range(1,101):
    Ctx = EncryptZero(key);
    List.append(int(EvalCtx(Ctx,uxe,uye))%4);

    print("");
    for j in range(0,4):
      print(str(j)+": "+str(List.count(j)));
    print("total : "+str(i))
```

Giophantus_weak_key.sage

```
n = 1201
#n = 1733;
#n = 2267;

q = 2^31-1;
l = 4;

K = GF(q);
PR.<t> = PolynomialRing(K);
R = PR.quotient(PR.ideal([t^n-1]));
PR2.<x,y> = PolynomialRing(R,2);

def SamplePoly():
    p = R.zero();
    for i in range(0,n):
        p += ZZ.random_element(0,q)*t^i;
    return p;

def SampleSmallPoly():
    sp = R.zero();
    for i in range(0,n):
        sp += ZZ.random_element(0,4)*t^i;
    return sp;

def keyGen():
    ux = SampleSmallPoly();
    uy = SampleSmallPoly();
    X10 = SamplePoly();
    X01 = SamplePoly();
    X00 = -X10*ux - X01*uy;
    X = X00 + X10*x + X01*y;
    return (ux,uy,X00,X10,X01);

def EvalAt1(X):
    return X.lift().substitute({t:1});

def Abs(a):
    return(min(int(a),int(-a)));

def INT(a):
    if int(a) < q/2 :
      return int(a)
    else :
      return -int(-a)

def FindSmallSolution(key):
    ux,uy,X00,X10,X01 = key;
    X00e = EvalAt1(X00);
    X10e = EvalAt1(X10);
    X01e = EvalAt1(X01);
```

```
    for a in range(0,4*n):
      uxe = a;
         uye = (-X00e -X10e*uxe)/X01e;
         if Abs(uye)<4*n :
             return(uxe,uye);
      uye = a;
         uxe = (-X00e -X01e*uxe)/X10e;
         if Abs(uxe)<4*n :
             return(uxe,uye);
      uxe = -a;
         uye = (-X00e -X10e*uxe)/X01e;
         if Abs(uye)<4*n :
             return(uxe,uye);
      uye = -a;
         uxe = (-X00e -X01e*uxe)/X10e;
         if Abs(uxe)<4*n :
             return(uxe,uye);

def EncryptZero(key):
    ux,uy,X00,X10,X01 = key;
    Ctx = X00 + X10*x + X01*y;
    R = SamplePoly() + SamplePoly()*x + SamplePoly()*y;
    Ctx = Ctx*R;
    for i in range(0,3):
        for j in range(0,3-i):
            Ctx += 4*SampleSmallPoly()*x^i*y^j;
    return Ctx;

def Decrypt(Ctx, key):
    ux,uy,X00,X10,X01 = key;
    eval = Ctx.substitute({x:ux,y:uy});
    return eval;

def EvalCtx(Ctx,uxe,uye):
    coefs = Ctx.coefficients();
    monoms= Ctx.monomials();
    eval = K.zero();
    for i in range(0,len(coefs)):
        eval += EvalAt1(coefs[i])*monoms[i].substitute({x:uxe,y:uye});
    return eval;

def MostLikelyEval(uxe,uye):
    mean =
6*n*(1+INT(uxe)+INT(uye)+INT(uxe)^2+INT(uye)^2+INT(uye)*INT(uxe));
    variance =
20*n*(1+Abs(uxe)^2+Abs(uye)^2+Abs(uxe)^4+Abs(uye)^4+Abs(uye)^2*Abs(uxe
)^2);
    stdev = sqrt(variance);
    T = RealDistribution('gaussian',stdev);
    probabilities = [0,0,0,0];
    for i in range(-100*q,100*q,q):
        prob =
```

```
T.cum_distribution_function(i+q-mean)-T.cum_distribution_function(i-me
an);
        probabilities[int(i*q%4)] += prob;
    sp = sorted(probabilities);
    print("distribution = "+str(probabilities))
    print("advantage = "+str(sp[3]+sp[2]-sp[1]-sp[0]));
    return probabilities;

print("looking for a weak key ...");

var = q*q;
counter = 0;
while sqrt(20*n*var) > 0.2*q:
    counter += 1;
    key = keyGen();
    uxe , uye = FindSmallSolution(key);
    var = Abs(uxe)^4+Abs(uye)^4+Abs(uxe)^2*Abs(uye)^2;
    print("try "+str(counter)+": stdev/q =
"+str(float(sqrt(20*n*var)/q)));

print("weak key found after "+str(counter)+" tries.");
MostLikelyEval(uxe,uye);

List = []
for i in range(1,101):
    Ctx = EncryptZero(key);
    List.append(int(EvalCtx(Ctx,uxe,uye))%4);

    print("");
    for j in range(0,4):
     print(str(j)+": "+str(List.count(j)));
    print("total : "+str(i))
```

IND-CPA attack on Giophantus

Ward Beullens Wouter Castryck Frederik Vercauteren

June 11, 2018

1 Giophantus

We briefly describe the Giophantus encryption system. To ease the presentation we fix the parameters $\deg X, \deg r$ and l to 1,1, and 4 respectively. This covers all the parameter sets that were submitted to the NIST PQC project.

Giophantus works over the ring $R_q = \mathbb{F}_q[t]/(t^n - 1)$, where the prime q and the integer n are parameters of the scheme. The Giophanuts NIST submission defines 3 parameter sets for three NIST security levels (see Table 1). Note that for security level V the last digit of the proposed prime q is a 5. However, the reference implementation submitted to NIST implements different parameters.

NIST Security Level	q	n	NIST Security Level	q	n
I	467424411	1201	I	$2^{31}-1$	1201
III	973190427	1733	III	$2^{31}-1$	1733
V	1665292875	2267	V	$2^{31}-1$	2267

Table 1: The parameter sets in the Giophantus NIST submission (left) and the parameter sets implemented by the reference implementation (right).

Key generation

- One samples two random 'small' elements $u_x, u_y \in R_q$ (meaning that the coefficients lie in $\{0, 1, 2, 3\}$).
- One chooses a random linear polynomial $X = X_{00} + X_{10} \cdot x + X_{01} \cdot y$ in $R_q[x, y]$ of which (u_x, u_y) is a solution.

The public key is $X(x, y)$, the secret key is (u_x, u_y).

Encryption

- One chooses a random linear polynomial $r(x, y) = r_{00} + r_{10} \cdot x + r_{01} \cdot y$ in $R_q[x, y]$ (i.e. with big coefficients).

- One chooses a random quadratic polynomial $e(x,y) = e_{00} + e_{10} \cdot x + e_{20} \cdot x^2 + e_{01} \cdot y + e_{11} \cdot xy + e_{02} \cdot y^2$ with small coefficients.

  The encryption of a small element $m \in R_q$ is now $E_m = m + r(x,y) \cdot X(x,y) + 4 \cdot e(x,y)$

Decryption

To decrypt, one plugs in the values of (u_x, u_y) in the cyphertext to get

$$E_m(u_x, u_y) = m + r(u_x, u_y) \cdot 0 + 4 \cdot e(x,y)\,.$$

This is equal to the message m mod 4, because $4 \cdot e(x,y)$ (a polynomial with small coefficients evaluated in small values) is small enough.

2 IND-CPA attack

Our attack uses the fact that evaluation at $t = 1$ is a ring-homomorphism from $R_q = \mathbb{F}_q[t]/(t^n - 1)$ to $\mathbb{F}_q$. Moreover, this homomorphism sends small elements of R_q to relatively small elements of $\mathbb{F}_q$ (they lie in the range $[0, 3n]$).

The first step of the attack is to evaluate the public key $X(x,y)$ in $t = 1$ to obtain a linear polynomial

$$X'(x,y) = X'_{00} + X'_{10} \cdot x + X'_{01} \cdot y \in \mathbb{F}_q[x,y].$$

Then we look for (u'_x, u'_y), the evaluation of (u_x, u_y) at $t = 1$. This is done easily by going through all values for u'_x (there are only $3n + 1$ possibilities), solving for u'_y and checking if $u'_y \in [0, 3n]$.

Given a cyphertext $E_m(x,y)$, we can evaluate its coefficients at $t = 1$ to get $E'_m \in \mathbb{F}_q[x,y]$ and plug (u'_x, u'_y) into this. This results in

$$E'_m(u'_x, u'_y) = m' + r'(u'_x, u'_y) \cdot X'(u'_x, u'_y) + 4e' \equiv m' + 4 \cdot e'(u'_x, u'_y) \pmod q\,.$$

Now we can try to get rid of the $4 \cdot e'(u'_x, u'_y)$ term by reducing modulo 4 to recover m' mod 4, which suffices to break IND-CPA security. This is not guaranteed to work because $4 \cdot e'(u'_x, u'_y)$ is generally larger than q such that first reducing $m' + 4 \cdot e'(u'_x, u'_y)$ mod q and then mod 4 does not necessarily result in m' mod 4. But still, $4 \cdot e'(u'_x, u'_y)$ is small enough such that $E'_m(u'_x, u'_y)$ leaks some information about m'. Moreover, for a sizeable fraction of public keys, there exists an additional solution to $X'(x,y) = 0$ (which is not necessarily the evaluation of a small solution in R_q) which is so small that we can distinguish cipher text with probability close to 1.

Attack on generic public keys

To show that information is leaked about m, we estimate the distribution of $E'_m(u'_x, u'_y)$ (over the integers). For fixed values of m and (u'_x, u'_y), the distribution of $E'_m(u'_x, u'_y)$ can be approximated by a Gaussian distibution. Indeed, we have

$$E'_m(u'_x, u'_y) = m' + 4 \cdot \left(e'_{00} + e'_{10}u'_x + e'_{20}u'^2_x + e'_{01}u'_y + e'_{11}u'_x u'_y + e'_{02}u'^2_y\right)\,,$$

where each e'_{ij}, being the evaluation of a random small element, is the sum of n independent uniform variables on $\{0,1,2,3\}$. Relying on the central limit theorem, we can approximate the distribution of each e'_{ij} by a Gaussian distribution with mean $\mu_{e'} = n \cdot \frac{0+1+2+3}{4}$ and variance $\sigma^2_{e'} = n \cdot \frac{(-1.5)^2+(-0.5)^2+0.5^2+1.5^2}{4}$. Adding these, we get that $E'_m(u'_x, u'_y)$ approximately follows a Gaussian distribution with mean $\mu_{E'_m} = m' + \mu_{e'}(1 + u'_x + u'_y + u'^2_x + u'_x u'_y + u'^2_y)$ and variance $\sigma^2_{E'_m} = \sigma^2_{e'}(1 + u'^2_x + u'^2_y + u'^4_x + u'^2_x u'^2_y + u'^4_y)$.

Given this estimate for the distribution of $E'_m(u'_x, u'_y)$, we can estimate $(E'_m(u'_x, u'_y) \bmod q) \bmod 4$. We denote by $a \bmod b$ the unique integer in $\{0, \cdots, b-1\}$ which is congruent to a modulo b, i.e. $a \bmod b = a - \lfloor \frac{a}{b} \rfloor b$. We now have

$$(E'_m(u'_x, u'_y) \bmod q) \bmod 4 = (m' - q \cdot \lfloor \frac{E'_m(u'_x, u'_y)}{q} \rfloor) \mod 4\,.$$

Therefore, if the standard deviation of $E'_m(u'_x, u'_y)$ is small compared to q, the value of $(E'_m(u'_x, u'_y) \bmod q) \bmod 4$ will be strongly biased towards $(m' - q \cdot \lfloor \frac{\mu_{E'_m}}{q} \rfloor) \mod 4$.

Table 2 shows some results of our experiments. For security level I, the standard deviation of $\sigma_{E'_m}/q$ is smaller than q, and this results in a high distinguishing advantage (i.e. the probability of the 2 most likely values of $(E'_m(u'_x, u'_y) \bmod q) \bmod 4$ minus the probability of the 2 least likely values.). For higher security levels the standard deviation becomes larger, and the distinguishing advantage decreases, but it still remains significant.

NIST Security Level	q	n	$\sigma_{E'_m}/q$	Distinguishing advantage
I	$2^{31}-1$	1201	$0.4 - 0.45$	$0.85 - 0.95$
III	$2^{31}-1$	1733	$0.95 - 1.05$	$0.25 - 0.35$
V	$2^{31}-1$	2267	$1.95 - 2.05$	$0.007 - 0.012$

Table 2: The standard deviation of $E'_m(u'_x, u'_y)$ relative to q and the distinguishing probability for typical public keys for the 3 security levels.

Attack on a subset of weak keys

For most public keys the smallest solution of $X'(x, y) = 0$ will be (u'_x, u'_y), the evaluation of (u_x, u_y) at $t = 1$. Since u_x and u_y are random small elements, the values of u'_x and u'_y are usually close to $1.5n$. However, for some public keys, there exist smaller solutions to $X'(x, y) = 0$, which are not necessarily the evaluation of a small solution in R^2_q. Using such a solutions (a, b) instead of (u'_x, u'_y) can result in a much smaller variance of $E'_m(a, b)$ and much larger distinguishing advantage.

Heuristically, we can expect that there exists a solution $X'(a, b) = 0$ with $-L < a\,,\, b < L$ with a probability of about $4L^2/q$, since there are roughly $4L^2$ possibilities for (a, b), and for each possibility there is a $1/q$ chance that $X'(a, b) = 0$. Suppose we hope for a solution (a, b) such that the standard deviation of $E'_m(a, b)$ is less than $q/5$. If such a solution

exists we can use it to distinguish ciphertexts with an advantage very close to 1. Since $\sigma_{E'_m} = \sigma_{e'}(1 + a^2 + b^2 + a^4 + a^2b^2 + b^4)^{1/2}$ it suffices to have

$$a, b < \sqrt{\frac{q}{5\sqrt{3}\sigma_{e'}}}$$

Using the $4L^2/q$ estimate, the probability that such a solution exists is roughly

$$\frac{4}{5\sqrt{3}\sigma_{e'}} \approx \frac{0.1}{\sqrt{n}}\,.$$

Therefore, we can estimate that for a fraction of $0.1/\sqrt{n}$ of the public keys, we can break IND-CPA security with an advantage close to 1. We have confirmed this with computer experiments; small solutions to $X'(x, y) = 0$ can be found after trying a small number (a few hundred) of public keys, and these solutions can be used to break the IND-CPA security with an advantage close to 1.

From: Phong Nguyen <Phong.Nguyen@ens.fr>
Sent: Wednesday, June 13, 2018 11:20 AM
To: pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: Giophanthus and *LWR-based submissions

Dear all,

Many lattice submissions rely on the hardness of BDD:
given a target t, a lattice L and a radius R, find all vectors v in L such that ||v-t|| <= R.
Typically, t is of the form t = u+e where u \in L and e is a noise with known distribution D, and the choice of R is based on D.

However, for some submissions, the distribution D is non-centered in the sense that the expectation E(e) is non-zero, which was apparently not always taken into account, leading to security overestimates.
If one wants to optimize the lattice attack to solve BDD, one may want to modify t and D in order to decrease E(||e||^2) and R.

Consider for instance Giophantus [1], whose parameters are chosen based on an analysis of a key-recovery attack described in [1].
For the parameters proposed to NIST, Giophantus works in the ring R=F_q[X]/(X^n-1) where q and n are prime numbers.
The secret key is (u_x,u_y) \in R^2 such that the coefficients of u_x and u_y are small, namely uniformly distributed over {0,1,2,3}.
The public key is (a,b,c) \in R^3 such that a*u_x + b*u_y + c = 0 in R.
The key-recovery attack of [1] transforms this equation into a BDD instance with noise e=(u_x,u_y) viewed as an integer vector.
The security analysis of [1] relies on E(||e||^2) = 7n, coming from E(z^2) = 7/2 when z is uniformly distributed over {0,1,2,3}.
However, if one subtracts to the BDD target t the vector (3/2,3/2,…,3/2), then E(||e||^2) becomes smaller, namely 5n/2.
So the attack of [1] is not optimal: the analysis should replace 7n by 5n/2, which decreases the claimed security in Table 5 of [1], by decreasing the estimated blocksize required to solve the BDD instance.

There was a similar phenomenon for low-density lattice attacks on the subset sum problem (see the survey [2]).

Non-centered noise also occurs for LWR, because of rounding ties.
The « naive » interpretation of LWR as a BDD-instance uses a noise distribution D which each coefficient is uniformly distributed over {-(u-1),…,u} for some integer u > 0.

In the Lizard and Saber submissions and [3], the hardness of an *LWR instance is not estimated through BDD: instead, it is assumed to be equivalent to the hardness of the LWE instance whose noise parameters are chosen to make the variance of each noise coordinate matching that of LWR.

However, Lizard, Saber and [3] seem to assume that the variance of each coefficient is u^2/3, as if the distribution was uniform over the interval (-u,u].
But the variance over {-(u-1),…,u} is slightly smaller, namely u^2/3-1/12.
So in Lizard and Saber analyses, terms of the form q^2/(12p^2) (where p and q are LWR parameters) should be replaced by (q^2/p^2-1)/12.

We make another observation about Giophantus.
For the NIST parameters, q and n are such that most elements in R are invertible (see [4] for precise statements). In the public key equation, a and b are chosen uniformly at random, so are very likely to be invertible.
This implies that by suitable multiplication, we can make a or b equal to 1, without affecting the security of the scheme.
This reduces the public key size by 33%, and simplifies the scheme, making it « closer » to NTRU.
In NTRU, the public key h and the secret key (f,g) are such that h*f = g in R.
Here, for instance, we can get u_x + h*u_y + d = 0 where (h,d) is the new public key.

Best regards,
Phong

[1] K. Akiyama and Y. Goto and S. Okumura and T. Takagi and K. Nuida and G. Hanaoka and H. Shimizu and Y. Ikematsu. A Public-key Encryption Scheme Based on Non-linear Indeterminate Equations (Giophantus).
https://eprint.iacr.org/2017/1241

[2] P. Q. Nguyen and J. Stern. The two faces of lattices in cryptology, Cryptography and Lattices – Proc. CALC '01, LNCS, vol. 2146, Springer-Verlag, 2001

[3] https://estimate-all-the-lwe-ntru-schemes.github.io/docs/

[4] J. Silverman. Invertibility in Truncated Polynomial Rings. NTRU Cryptosystems Technical Report #009. 1998.