Paper-Reproduction

December 18, 2023

1 A new attack on some RSA variants

1.1 The New Attack Overview

This is the paper reproduction of A new attack on some RSA variants. We mainly discuss the following scenario :

Given $N = pq, e$ such that p, q share r bit least significant bits i.e. $|p - q| = 2^r u$ and $ed = k(p^2 - 1)(q^2 - 1) + 1$, where $e = N^\alpha$, $d = N^\delta$ and $2^r = N^\beta$. We want to factor N in polynomial time. That is :

$$ed = k(p^2 - 1)(q^2 - 1) + 1, \text{ with small d and p, q share some lsbs}$$

This problem can be solved in polynomial time when (there is a typo in paper):

$$\delta < \frac{7}{3} - \frac{4}{3}\beta - \frac{2}{3}\sqrt{(1 - 4\beta)(1 + 3\alpha - 4\beta)}$$

Some typical situations bound:

$$\begin{cases} d \leq N^{0.569} & e \approx N^2, \beta = 0 \\ d \leq N^{0.873} & e \approx N^2, \beta = 0.1 \end{cases}$$

[1]:
```
from math import ceil
from Crypto.Util.number import isPrime
from random import randrange, getrandbits

def gen_rsa_chall(delta, beta, nbits=1000):
    # alpha = 2, typically
    kbit = ceil(nbits * beta)
    ubit = (nbits - 2*kbit)//2
    while True:
        k = getrandbits(kbit)
        pp = getrandbits(ubit) << kbit
        qq = getrandbits(ubit) << kbit
        p = pp + k
        q = qq + k
        if isPrime(p) and isPrime(q):
```

```
                break
        if q > p:
            p, q = q, p

        n = p * q
        phi = (p ** 2 - 1) * (q ** 2 - 1)
        ub = int(n ** delta)
        lb = int(n ** (delta - 0.02))
        while True:
            d = randrange(lb, ub)
            if gcd(d, phi) == 1:
                break
        e = int(inverse_mod(d, phi))
        sk = (p, q, d)
        pk = (n, e)
        return pk, sk

# pk, sk = gen_rsa_chall(0.7, 0.1)

def delat_upper_bound(beta, alpha=2):
    # alpha = 2, typically
    return RR(7/3 - (4/3 * beta) - 2/3 * sqrt((1 - 4 * beta)*(1 + 3 * alpha - 4␣
 ↪* beta)))

print(f"[+] beta = 0.0, delta = {delat_upper_bound(0)}")
print(f"[+] beta = 0.1, delta = {delat_upper_bound(0.1)}")

nbits = 1000
beta =  0.1
delta = 0.7
r = ceil(nbits * beta)
print("[*] generating samples")
(N,e), (p,q,d) = gen_rsa_chall(delta, beta, nbits)
print("[+] generation done")
```

```
[+] beta = 0.0, delta = 0.569499125956940
[+] beta = 0.1, delta = 0.873350083857841
[*] generating samples
[+] generation done
```

1.2 More Leak Bits of p + q

Obviously, there is r-bit leak of both p,q given $N = pq$. Let $p = 2^r p_1 + u_0, p = 2^r q_1 + u_0$. We have $N \equiv u_0^2 \mod 2^r$ which can be used to recover u_0. However, we can get more leak bits of $p + q$.

Considering $p + q = 2^r p_1 + 2^r q_1 + 2u_0$ and $N = pq = 2^{2r} p_1 q_1 + (2^r p_1 + 2^r q_1)u_0 + u_0^2$, we have $N \equiv (2^r p_1 + 2^r q_1)u_0 + u_0^2 \mod 2^{2r}$ is known. $p + q \equiv 2^r p_1 + 2^r q_1 + 2u_0 \mod 2^{2r}$ denoted as v_0 is then recovered by computing $v_0 = 2u_0 + (N - u_0^2)u_0^{-1} \mod 2^{2r}$.

```
[2]: def derive_2r_bits_leak(N, r):
         Zr = Zmod(2^r)
         u0_list = Zr(N).nth_root(2, all = True)
         vs = []
         for u0 in u0_list:
             u0 = ZZ(u0)
             v0 = 2*u0 + (N - u0^2) * ZZ(inverse_mod(u0, 2^(2 * r)))
             vs.append(v0 % 2^(2*r))
         return vs

     v0_list = derive_2r_bits_leak(N, r)
     print(f"[+] find {len(v0_list) = } { r = }")
     print("[+] check : ", (p + q) % 2^(2 * r) in v0_list)
```

```
[+] find len(v0_list) = 4  r = 100
[+] check :  True
```

1.3 Constructing Polynomial

Observe that $(p^2-1)(q^2-1) = (N+1)^2 - (p+q)^2$ and denote $p+q = 2^{2r}v + v_0$, we have

$$ed = k(p^2-1)(q^2-1) + 1$$

$$\implies k(p+q)^2 - k(N+1)^2 + 1 \equiv 0 \mod e$$

$$k(2^{4r}v^2 + 2^{2r+1}v_0 v) - k((N+1)^2 - v_0^2) - 1 \equiv 0 \mod e$$

Multiply 2^{-4r} make this equation monic :

$$kv^2 + \underbrace{\frac{v_0}{2^{2r-1}}}_{a_1} kv + \underbrace{\frac{-(N+1)^2 + v_0^2}{2^{4r}}}_{a_2} k + \underbrace{\frac{-1}{2^{4r}}}_{a_3} \equiv 0 \bmod e$$

We can constuct a polynomial with small roots $x = k, y = v$:

$$f(x,y) = xy^2 + a_1 xy + a_2 x + a_3 \mod e$$

```
[3]: def construct_polynomials(N, e, beta):
         nbits = ZZ(N).nbits()
         r = ceil(nbits * beta)
         v0_list = derive_2r_bits_leak(N, r)
         PR = PolynomialRing(Zmod(e), ["x", "y"],  order = "lex")
         x, y = PR.gens()
         polys = []
         for v0 in v0_list:
```

```
        a1 = v0 * ZZ(inverse_mod(2^(2 * r -1), e)) % e
        a2 = (v0^2 - (N+1)^2) * ZZ(inverse_mod(2^(4 * r), e)) % e
        a3 = ZZ(inverse_mod(-2^(4 * r), e))
        poly = x * y^2 + a1 * x * y + a2 * x + a3
        polys.append(poly)
    return v0_list, polys

v0_list, polys = construct_polynomials(N, e, beta)
```

1.4 Construct Coppersmith Polynomial Sequence

Although $f(x, y) = xy^2 + a_1xy + a_2x + a_3 \mod e$ has small roots k, v approximately bounded by N^δ and $N^{0.5-2\beta}$. The general coppersmith method as far as I known does not yield a solution. In this paper, the authors choose the following polynomial sequence to generate coppersmith's matrix :

$$\begin{aligned} G_{s,i,j}(x, y) =& x^{i-s}y^{j-2s}f(x, y)^s e^{m-s} \\ & \text{for } s = 0, \dots m, i = s, \dots, m, j = 2s, 2s + 1, \end{aligned}$$

and

$$\begin{aligned} H_{s,i,j}(x, y) =& x^{i-s}y^{j-2s}f(x, y)^s e^{m-s} \\ & \text{for } s = 0, \dots m, i = s, j = 2s + 2, \dots, 2s + t. \end{aligned}$$

I am not expert in lattice analysis therefore I am not going to discuss the reason why these polynomials are chosen. Construct matrix by all the coefficients of $H_{s,i,j}(Xx, Yy), G_{s,i,j}(Xx, Yy)$ where the exact bound $X = 2N^{\alpha+\delta-2}$ and $Y = 3N^{\frac{1}{2}-2\beta}$.

The following code uses the resultant method to recover the roots in integer ring. It seems that using groebner basis or variety method in sagemath does not work well.

[4]:
```
from subprocess import check_output
from re import findall

def derive_2r_bits_leak(N, r):
    Zr = Zmod(2^r)
    u0_list = Zr(N).nth_root(2, all = True)
    vs = []
    for u0 in u0_list:
        u0 = ZZ(u0)
        v0 = 2*u0 + (N - u0^2) * ZZ(inverse_mod(u0, 2^(2 * r)))
        vs.append(v0 % 2^(2*r))
    return vs

def flatter(M):
    # compile https://github.com/keeganryan/flatter and put it in $PATH
    z = "[[" + "]\n[".join(" ".join(map(str, row)) for row in M) + "]]"
```

```
    ret = check_output(["flatter"], input=z.encode())
    return matrix(M.nrows(), M.ncols(), map(int, findall(b"-?\\d+", ret)))

def construct_polynomials(N, e, beta):
    nbits = ZZ(N).nbits()
    r = ceil(nbits * beta)
    v0_list = derive_2r_bits_leak(N, r)
    PR = PolynomialRing(Zmod(e), ["x", "y"],  order = "lex")
    x, y = PR.gens()
    polys = []
    for v0 in v0_list:
        a1 = v0 * ZZ(inverse_mod(2^(2 * r -1), e)) % e
        a2 = (v0^2 - (N+1)^2) * ZZ(inverse_mod(2^(4 * r), e)) % e
        a3 = ZZ(inverse_mod(-2^(4 * r), e))
        poly = x * y^2 + a1 * x * y + a2 * x + a3
        polys.append(poly)
    return polys

def G_sij(f, s, i, j, m, e):
    x, y = f.parent().gens()
    return x^(i - s) * y^(j - 2*s) * f^s * e^(m - s)

def gen_copper_polys(f, m, t, e):
    gpolys = []
    x, y = f.parent().gens()
    for s in range(0, m+1):
        for i in range(s, m+1):
            for j in range(2*s, 2*s+2):
                gpolys.append(G_sij(f(x, y), s, i, j, m, e))
    hpolys = []
    for s in range(0, m+1):
        for i in range(s, s + 1):
            for j in range(2*s + 2, 2*s + t + 1):
                hpolys.append(G_sij(f(x, y), s, i, j, m, e))
    return gpolys, hpolys

def bivarivate_small_roots(f, X, Y, m, t, poly_num=3):
    R = f.base_ring()
    e = R.cardinality()
    f = f.change_ring(ZZ)

    g_polys, hpolys = gen_copper_polys(f, m, t, e)

    G = Sequence(g_polys + hpolys, f.parent())
    B, monomials = G.coefficient_matrix()
    monomials = vector(monomials)
    factors = [monomial(X, Y) for monomial in monomials]
```

```
    for i, factor in enumerate(factors):
        B.rescale_col(i, factor)

    print(f"[+] start LLL with {B.dimensions() = }")
    B = flatter(B.dense_matrix())
    print("[+] LLL done")

    B = B.change_ring(QQ)
    for i, factor in enumerate(factors):
        B.rescale_col(i, 1/factor)
    polys = B * monomials
    selected_polys = polys[:poly_num]

    x, y = polys[0].parent().gens()
    roots = []
    for poly1 in selected_polys:
        for poly2 in selected_polys:
            if poly1 == poly2:
                continue
            poly_res = poly1.resultant(poly2, x)
            if poly_res.is_constant():
                continue
            poly_univar_y = poly_res.univariate_polynomial()
            y_roots = poly_univar_y.roots(ring=ZZ, multiplicities=False)
            if len(y_roots)!= 0:
                for y_root in y_roots:
                    if abs(y_root) >= Y:
                        print(f"[+] unbounded root find {y_root = }, please␣
↪check")
                        continue
                    poly_univar_x = poly1(x, y_root).univariate_polynomial()
                    x_roots = poly_univar_x.roots(ring=ZZ, multiplicities=False)
                    for x_root in x_roots:
                        if abs(x_root) >= X:
                            print(f"[+] unbounded root find {x_root = }, please␣
↪check")
                            continue
                        roots.append((x_root, y_root))
                # we will not check other polynomials
                return roots
    return roots

def check_paper_samples():
    N = 611402847577596838649117628567007514815745193613363898749361
    e =␣
↪2566205365878363253892897423089936609824669728195688966566612491050818871042664142920003896
    beta = 0.1
```

```
    delta = 0.7
    r = 20

    alpha = RR(log(e, N))
    X = int(2 * N^(alpha + delta - 2))
    Y = int(3 * N^(0.5 - 2*beta))
    m = 4
    t = 4

    k = 173874778620245362592189712830325998289 07
    v = 1433911212640302358

    polys = construct_polynomials(N, e, beta)
    for poly in polys:
        roots = bivarivate_small_roots(poly, X, Y, m, t)
        if len(roots) != 0:
            print(f"[+] recovered roots {roots = }")

# check_paper_samples()

alpha = RR(log(e, N))
X = int(2 * N^(alpha + delta - 2))
Y = int(3 * N^(0.5 - 2*beta))
m = 4
t = 4

# k = (e*d - 1)//((p^2 -1)*(q^2-1))
# v = ((p+q) - (p+q)%(2^(2*r)))>>(2*r)

for poly, v0 in zip(polys, v0_list):
    roots = bivarivate_small_roots(poly, X, Y, m, t)
    if len(roots) != 0:
        print(f"[+] recovered roots {roots = }")
        k, v = roots[0]
        if v < 0:
            p_plus_q = (- v - 1) * 2^(2*r) + (2^(2*r) - v0)
        else:
            p_plus_q = v * 2^(2*r) + v0

        p_minus_q = ZZ(sqrt(p_plus_q**2 - 4 * N))
        rp = (p_plus_q + p_minus_q) // 2
        rq = (p_plus_q - p_minus_q) // 2
        assert rp * rq == N
        print(f"[+] successfully factored {N} = {p} * {q}")
```

```
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
```

```
[+] recovered roots roots = [(33278628587387063474554135681348416821311709343373
84949350630502726849904688594352633671918552793819554489605330589632411900514012
40152709547954109855056057004127568922299650283108403792837118465338648141404927
, -12524401779589530941171630988189118050267233329090426039574820422548620686934
00327746048732)]
[+] successfully factored 529440863547104437629608664225708286295074615957610119
56904026935999307927531218209196392767423763445154732904904240554424433452053429
01880513884180776336661696395021575324880554637455043661593451685039990765794235
18573959301232565726587422713354419063373480828215805723074504964970117860843593
911041 = 17014174194535802376464553910364164771779023051748199546816837500629869
76689836352184102670210332422216012557830857050394426544901028441495255812881023
* 311176350667161586491339885744901581816670102113896817085112847809805456607046
643188453779966910367663741415343876535814215069250153644835218289277567
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
[+] recovered roots roots = [(33278628587387063474554135681348416821311709343373
84949350630502726849904688594352633671918552793819554489605330589632411900514012
40152709547954109855056057004127568922299650283108403792837118465338648141404927
, 125244017795895309411716309881891180502672333290904260395748204225486206869340
0327746048731)]
[+] successfully factored 529440863547104437629608664225708286295074615957610119
56904026935999307927531218209196392767423763445154732904904240554424433452053429
01880513884180776336661696395021575324880554637455043661593451685039990765794235
18573959301232565726587422713354419063373480828215805723074504964970117860843593
911041 = 17014174194535802376464553910364164771779023051748199546816837500629869
76689836352184102670210332422216012557830857050394426544901028441495255812881023
* 311176350667161586491339885744901581816670102113896817085112847809805456607046
643188453779966910367663741415343876535814215069250153644835218289277567
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
```

[5]:

```
check_paper_samples()
```

```
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
[+] recovered roots roots = [(17387477862024536259218971283032599828907,
1433911212640302358)]
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
[+] recovered roots roots = [(17387477862024536259218971283032599828907,
-1433911212640302359)]
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
[+] start LLL with B.dimensions() = (45, 45)
[+] LLL done
```