Twenty Years of Attacks on the RSA Cryptosystem & Some Interesting RSA Problems

This project includes implementations of various attacks described in the famous paper Twenty Years of Attacks on the RSA Cryptosystem by Professor Dan Boneh, as well as a list of cryptography problems that I encounterd over the years in contests or were presented to me by teammates, and I found unique. The order of the attacks is not necessarily kept.

The primary language of choice is Python, and more specifically Sagemath. Note that Jupyter notebook with a SageMath 10.2 kernel were used (although some solvers may be written in 9.x it should still be compatible).

When first creating this project in late 2023, my goal was to get a better grasp of the RSA cryptosystem, as well as explore some of the cases that compromise security (even though I follow through with most proofs). Although fascinating, provable security, is out of the scope of this project, as I targeted to develop a practical understanding and get familiar with SageMath for cybersecurity Capture The Flag (CTF) competitions. That's why I have implemented a lot of fundamental algorithms myself based on their respective proofs, which are already implemented in the SageMath framework.

The highlight of this project is experimenting with lattice reduction, to an extent that is not fully shown here, through amazing resources such as Practical lattice reductions for CTF challenges and A Gentle Tutorial for Lattice-Based Cryptanalysis. I find it intriguing that LLL and other similar algorithms can traverse through an exponential search space ($\mathbb{Z}^n$) in polynomial running time, having to use it extensively for CTF challenges. It is important to mention that lattice problems seem to have the potential not only to encapsulate other cryptosystems but also to give rise to potentially post-quantum public-key schemes like Kyber.

Looking forward, I aspire to explore further both the theoretical side of cryptography (and more generally computationally intractable problems, and overview more open-source implementations of such algorithms.

Finally, I feel the need to apologize for not following a proper citation system, and instead leaving hyperlinks wherever I thought it was necessary.

Panagiotis Brezatis

Twenty Years of Attacks on the RSA Cryptosystem

1. Recovering p, q having d
2. Blinding

3. Hastad's attack
4. Common modulus
5. Franklin-Reiter related message attack
6. Wiener's attack
7. Coppersmith's Attack (LLL) on a partially known message

Recovering p, q having d

As stated in fact 1, for a public key $\langle N, e\rangle$ given the private key d, one can effictively recover the factorisation of N.

Notice that
$k = ed - 1$ and $k|\varphi(N)$, which is even. Therefore $g_1 = g^{k/2}$ is a square root of unity for $g \in \mathbb{Z}_{\mathrm{n}}^*$.

By applying the CRT it is evident that $g_1 \equiv \pm 1 \mod q, g_1 \equiv \pm 1 \mod p$ and thus 2 out of the possible 4 roots reveal the factorization of N.

According to the paper (proof of fact 1 - page 3) , for a random choice of g the probability that any element of the sequence $g^{k/2^t} \equiv -1 \mod p$ (or mod q) is 50%.

In [ ]:
```
p = random_prime(2^1024)
q = random_prime(2^1024)

n = p * q

e = 0x10001

phi = (p - 1)*(q - 1)

d = pow(e, -1, phi)
```

In [ ]:
```
k = e*d - 1

pp = 1
for g in range(2,2**16):

    k_t = k
    while k_t % 2 == 0:
        k_t //= 2
        rt = pow(g,k_t,n)

        pp = gcd(rt - 1, n)

        if pp > 1 and pp != n:
            print(pp)
            break
    if pp > 1 and pp != n:
        break

qq = n // pp
```

```python
print('[+] Recovered the factorisation of N')
print(f'{pp=} \n {qq=}')
```

Blinding

Let $\langle N, d\rangle$ be a private key. Let's suppose that one can sign arbitrary messages, except from some message, say $M \in Z_n^*$.
One can still sign $M' \equiv r^e M \mod N$, producing the following signature:
$S' \equiv (M')^d \equiv M^d r \mod N$.
It is obvious that we can recover M's signature by diving by r.

In [ ]:
```python
def bytes_to_long(b):
    return int(b.hex(), base=16)

def long_to_bytes(l):
    return bytes.fromhex(hex(l)[2:])
```

In [ ]:
```python
p = random_prime(2^1024)
q = random_prime(2^1024)

n = p * q


e = 0x10001
d = pow(e, -1, (p -1) * (q - 1))


M = bytes_to_long(b'Secret Message')
```

In [ ]:
```python
r = random_prime(2^100) #probabilistic guarantee that it's invertible

M_prime = (M * r^e) % n

S_prime = pow(M_prime, d, n)
S = pow(M, d, n)


assert (S_prime * pow(r, -1, n)) % n == S
```

Hastad's attack

We know that a message m has been encrypted using RSA keys of the form $\langle e, N_i\rangle$, k times.

Given that $k \geq e$, we can recover m^e (and consecutively m) by applying the Chinese Remainder Theorem (CRT) underlied by the following isomorphism:

$$\mathbb{Z}/N_1N_2\ldots N_k\mathbb{Z} \cong \mathbb{Z}/N_1\mathbb{Z}\times\ldots\times\mathbb{Z}/N_k\mathbb{Z}$$

Note that we can assume that all N are coprime, since in case they shared a factor, we could recover p_i and q_i.

https://en.wikipedia.org/wiki/Chinese_remainder_theorem#Using_the_existence_construction

In [ ]:
```
def bytes_to_long(bts):
    return int(bts.hex(), base=16)

def long_to_bytes(lng):
    return bytes.fromhex(hex(lng)[2:])
```

In [ ]:
```
e = 3

Ns = [ random_prime(2**1024) * random_prime(2**1024) for i in range(e)]



m = bytes_to_long(b"Well hidden message!!!! Lorem ipsum \
  dolor sit amet, consectetur adipiscing elit, \
  sed do eiusmod tempor incididunt ut labore ")

Cts = [pow(m, e, n) for n in Ns]
```

Reference crt implementations:

https://github.com/sympy/sympy/blob/master/sympy/polys/galoistools.py#L12

https://cp-algorithms.com/algebra/chinese-remainder-theorem.html

https://wiki.math.ntnu.no/_media/tma4155/2010h/euclid.pdf

Working mod a

In [ ]:
```
def xgcd(a, b):
    """
    Implementation of the Extended Euclidean Algorithm
    a, b -> integers
    """

    a1, b1 = a, b
    x0, x1 = 1, 0
    y0, y1 = 0, 1

    while b1 != 0:

        q = a1 // b1
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
        a1, b1 = b1, a1 - q * b1

    return (x0, y0, a1)





def crt(r, m):
    """
    Implementation of the Chinese Remainder Theorem
    r -> list of residues
    m -> list of modulos
    """
    assert len(m) == len(r)
```

```
    m1, r1 = m[0], r[0]

    for m2, r2 in zip(m[1:], r[1:]):
        #note that the moduli are assumed to be coprime
        a1, a2, _ = xgcd(m1, m2)


        """
        mod m1, everything except r1 cancels out since:
        a1*m1 + a2*m2 = 1
        SImilarly, mod m2 everything except r2 cancels out proving that
        this is a solution for (ri, r)
        """

        r1 = (r1 * a2 * m2 + r2 * a1 * m1) % (m1 * m2)
        m1 *= m2

    return (r1, m1)
```

Notice that $a_1 m_1 + a_2 m_2 = 1$

$\langle r_1, m_1 \rangle$ is indeed a recursively produced solution since:
$r_1 a_2 m_2 + r_2 a_1 m_1 \equiv r_1(1 - a_1 m_1) + r_2 a_1 m_1 \equiv r_1 \mod m_1$

Similarly, $r_1 a_2 m_2 + r_2 a_1 m_1 \equiv r_2 \mod m_2$

Having implemented CRT we can now recover m:

In [ ]:
```
m_e, _ = crt(Cts, Ns)

m = m_e.nth_root(3)

print(long_to_bytes(m))
```

Common Modulus

Suppose there is a message m and it is encrypted separately using keys $\langle e_1, N \rangle$ and $\langle e_2, N \rangle$ with $gcd(e_1, e_2) = 1$

Then we can apply the Extended Eucledean Algorithm (XGCD) to find the bezout coefficients for e_1 and e_2. Since e_1 and e_2 are coprime, we can get $a_1 e_1 + a_2 e_2 = 1$.

But notice that we have:
$c_1 = m^{e_1} \mod n$ and
$c_2 = m^{e_2} \mod n$

So we can produce
$m^{e_1 a_1} \mod n$ and
$m^{e_2 a_2} \mod n$

and thus,

$$m^{e_1 a_1 + e_2 a_2} \equiv m^1 \mod n$$

Since I have already implemented XGCD for the basic Hastad attack, I will utilize sage's built-in implementation for this proof-of-concept.

In [ ]:
```
from os import urandom

def bytes_to_long(bts):
    return int(bts.hex(), base=16)

def long_to_bytes(lng):
    return bytes.fromhex(hex(lng)[2:])
```

In [ ]:
```
p = random_prime(2**1024)
q = random_prime(2**1024)

n = p * q


e1 = random_prime(2**32)
e2 = random_prime(2**32)

assert gcd(e1, e2) == 1

m = bytes_to_long(b'Well hidden message!!!! ' + urandom(100))

c1 = pow(m, e1, n)
c2 = pow(m, e2, n)
```

Attack

In [ ]:
```
_, a1, a2 = xgcd(e1, e2)

k1 = pow(c1, a1, n)
k2 = pow(c2, a2, n)

pt = (k1 * k2) % n
print(long_to_bytes(pt))
```

Franklin-Reiter related message attack

Let $\langle e, N \rangle$ be the public key, and suppose $m_1 = f(m_2) \mod N$, for some known $f \in \mathbb{Z}_{\mathbb{N}}[x]$, where f is a linear polynomial ($f(x) = ax + b$). Given c_1, c_2, the algorithm can efficiently recover m_1, m_2 for any relatively small e.

Notice that m_2 is a root of both $f(x)^e - c_1 \mod N$ and $x^e - c_2 \mod N$. That said, we can apply polynomial G.C.D. in order to recover m_2.

The core idea is that for small exponents, the G.C.D is expected to be linear in most cases.

In [ ]:
```
def bytes_to_long(b):
    return int(b.hex(), base=16)

def long_to_bytes(l):
    return bytes.fromhex(hex(l)[2:])
```

In [ ]:
```
p = random_prime(2^1024)
q = random_prime(2^1024)

n = p * q

#
e = 3

a = randint(0,2^16)
b = randint(0,2^16)

m_2 =  bytes_to_long(b"Well hidden message!!!! Lorem ipsum \
   dolor sit amet, consectetur adipiscing elit, \
   sed do eiusmod tempor incididunt ut labore ")

# m_2 = bytes_to_long(b"Well hidden message!!!!")

m_1 = (a * m_2 + b) % n

c_2 = pow(m_2, e, n)
c_1 = pow(m_1, e, n)
```

The implementation below calculates the GCD in $\mathbb{Q}[x]$, thus works only when $x^e, f(x)^e$ are both less than N.

In [ ]:
```
from copy import copy


def polyDiv(x1, x2):
    assert x2 != 0
    q = 0
    r, d = x1, x2
    # print(r.poly, d.poly)
    while r.poly != 0 and d.poly != 0 and r.degr() >= d.degr():
#         print(r.poly, r.lead(), d.lead())
        t = r.lead() / d.lead()



        q += t * xs ^ (r.degr() - d.degr())
        r.poly -= t * d.poly * xs ^ (r.degr() - d.degr())
        r.poly = r.poly.simplify_full()

#     print('polyDiv ', q, r)
    return Poly(q), r


def polyGCD(x1, x2):
    if x2.poly == 0:
        return Poly(x1.poly / x1.lead())
```

```
    x1, x2 = x2, x1 % x2
#      print('polyGCD: ', x1, x2)


    return polyGCD(copy(x1), copy(x2))


class Poly:
    def __init__(self, poly):
        self.poly = poly

    def __repr__(self):
        return str(self.poly)

    def __eq__(self, other):
        if type(other) == type(self):
            return self.poly == other.poly
        else:
            return self.poly == other

    def __mod__(self, other):
        return polyDiv(self, other)[1]

    def degr(self):
        return self.poly.degree(xs)

    def lead(self):
        #print(self.poly.coefficient(xs, n=self.degr()), self.degr())
        return self.poly.coefficient(xs, n=self.degr())




xs = var('xs')
xx = Poly(xs ^ 3 + xs^2 + xs + 1)
xw = Poly(xs ^ 2 - 1)

res1 = polyGCD(copy(xx), copy(xw))

assert res1 == xs + 1
```

In [ ]:
```
m = var('xs')

P1 = (a*xs + b) ^ e - c_1
P2 = xs ^ e - c_2

P1 = Poly(P1)
P2 = Poly(P2)

print(P1, P2)
print(polyGCD(P1,P2))

msg = -polyGCD(P1, P2).poly.coefficient(xs, n=0)

print(msg)
```

We can edit this implementation so that it divides the polynomials in $\mathbb{Z}_{\mathbb{N}}[x]$

In [ ]:

```
###TODO
###add Zn solver from .sage file


def polyDivZn(x1, x2):
    assert x2 != 0
    q = 0
    r, d = x1, x2
    # print(r.poly, d.poly)
    while r.poly != 0 and d.poly != 0 and r.degr() >= d.degr():
        print(type(d.lead()))
        d_i = Integer(d.lead()).inverse_mod(n)
        print(d_i)
#         print(r.poly, r.lead(), d.lead())
        t = (Integer(r.lead()) * d_i) % n



        q += t * xs ^ (r.degr() - d.degr())
        r.poly -= t * d.poly * xs ^ (r.degr() - d.degr())
        r.poly = r.poly.simplify_full()

#     print('polyDiv ', q, r)
    return Poly(q), r

def polyGCDZn(x1, x2):
    if x2.poly == 0:
        return Poly(x1.poly * x1.inverse_mod(n))

    x1, x2 = x2, x1 % x2
    # print('polyGCD: ', x1, x2)


    return polyGCD(copy(x1), copy(x2))



class PolyZn:
    def __init__(self, poly):
        self.poly = poly

    def __repr__(self):
        return str(self.poly)

    def __eq__(self, other):
        if type(other) == type(self):
            return self.poly == other.poly
        else:
            return self.poly == other

    def __mod__(self, other):
        return polyDivZn(self, other)[1]

    def degr(self):
        return self.poly.degree(xs)

    def lead(self):
        #print(self.poly.coefficient(xs, n=self.degr()), self.degr())
```

```
        return self.poly.coefficient(xs, n=self.degr())


xs = var('xs')
xx = PolyZn(xs ^ 3 + xs^2 + xs + 1)
xw = PolyZn(xs ^ 2 - 1)

res1 = polyGCDZn(copy(xx), copy(xw))

assert res1 == xs + 1
```

Wiener's Attack

If d is smaller than $2^{n/4}$, then we can recover p,q.

In [ ]:
```
p = random_prime(2**1024)
q = random_prime(2**1024)

n = p * q

phi = (p - 1)*(q - 1)

bound = 2 ** (n.bit_length() // 4)



# generating d to be a prime, so that it is guaranteed that there's an inverse
# any coprime to phi can be used
# in any case, this doesn't affect numberical results

d = random_prime(int(1/3 * bound))

print(d)


e = pow(d, -1, phi)


print(f'{e=}')
print(f'{n=}')
```

Because $k < d < 1/3 * N^{1/4}$

$$|\frac{e}{N} - \frac{k}{d}| \leq \frac{1}{dN^{1/4}} < \frac{1}{2d^2}$$

Note, d is the private exponent, and k is derived from the relation $ed = 1 + k\varphi(N)$

As stated in the paper, all fractions of this form are obtained as convergents of the continued fraction expansion of $\dfrac{e}{N}$

https://math.stackexchange.com/a/2698953
https://en.wikipedia.org/wiki/Wiener%27s_attack#Example

In [ ]:
```
def continued_fraq(num, denom):
    decomp = []

    while num > 1:
        decomp.append(num // denom)

        num, denom = denom, num % denom

    return decomp


e1 = 17993 #test vars from wikipedia
n1 = 90581


decomp = continued_fraq(e, n)
print(decomp)
```

In [ ]:
```
from math import gcd

def calc_fraq(decomp):

    if len(decomp) == 1:
        return decomp[0]

    decomp = decomp[::-1]

    nom, denom = decomp[0], 1

    for idx in range(len(decomp) - 1):
        #reverse
        nom, denom = denom, nom

        #add nxt
        nom = nom + decomp[idx + 1] * denom


    return (nom, denom)




def calc_convergents(decomp):
    convergents = []


    #building all i-th fractions separately
    #runs in O(n^2), where n is log2(N), still negligible complexity.
    for i in range(len(decomp)):
        convergents.append(calc_fraq(decomp[:i + 1]))


    return convergents




# decomp = continued_fraq(e, n)

convergents = calc_convergents(decomp)
```

```
print(convergents)
```

Having the continued fractions expansion of $\frac{e}{N}$, we can recover p and q:

$$\varphi(N) = \frac{ed - 1}{k}$$

But since p, q primes, we can solve the following system

$$\begin{cases} \varphi(N) = (p-1)(q-1) = N - p - q + 1 \\ N = pq \end{cases}$$

In [ ]:
```
#we can use sage to solve this as a 2nd degree equation equation
#Develop a proof-of-concept that doesn't use sage, but rather Fact 1 from page
#Alternatively we can use the code from Recover_p_q
p = q = -1

for k, d in convergents[1:]:
    phi = (e*d - 1) // k
    R.<x> = PolynomialRing(ZZ)
    Eq = x^2 - (n - phi + 1)*x + n

    primes = Eq.roots()
    if not primes:
        continue
    print('[+]Found factorisation of n')
    p, q = [i[0] for i in primes]
    assert p * q == n

phi = (p - 1)*(q - 1)
d = pow(e, -1, phi)

print(f'{p = }\n{q = }\n{phi = }\n{d = }')
```

Coppersmith's Attack (LLL) on a partially known message

Suppose $m = m^{'} + x_0$, if x_0 is small we can recover it.
In particular, $|x_0| \leq \frac{N^{1/e}}{2}$ needs to hold.
For example, when $e = 3$, x_0 needs to be $\sim 1/3$ of $\log_2 N$ (the bits of N).
It is evident, that e needs to be relatively small for this attack to work.

We can take $f(x) = (m^{'} + x)^e - c \mod N$ and find a polynomial that is guaranteed to have x_0 as a root over $\mathbb{Z}$. What is unique about Coppersmith is that we can traverse through an exponential search space in polynomial running time (complexity of LLL).

https://eprint.iacr.org/2023/032.pdf (5.1.1)

In [ ]:
```
def bytes_to_long(b):
    return int(b.hex(), base=16)

def long_to_bytes(l):
    return bytes.fromhex(hex(l)[2:])
```

In [ ]:
```
phi = 3
e = 3

#assure coprime to e
while phi % e == 0:
    p = random_prime(2**1024)
    q = random_prime(2**1024)

    n = p * q

    phi = (p - 1)*(q - 1)

e = 3

d = pow(e, -1, phi)

m = bytes_to_long(b"Well hidden message!!!! Lorem ipsum \
   dolor sit amet, consectetur adipiscing elit, \
   sed do eiusmod tempor incididunt ut labore ")

print(m.bit_length())

c = pow(m, e, n)
```

In [ ]:
```
R.<x> = PolynomialRing(Integers(n))

known = (m >> (m.bit_length() // 3)) * 2 ^ (m.bit_length() // 3)

f_x = (known + x) ^ 3 - c

a = f_x.coefficients()


X = round(n ^ (1/3))


B = matrix(ZZ, [
    [n,          0,         0,    0],
    [0,      n * X,         0,    0],
    [0,          0,   n * X^2,    0],
    [a[0], a[1]*X, a[2]*X^2, X^3]
])


# print(B.LLL())

coefs = B.rows()[0]
ff_x = sum([coefs[i]*x^i//(X**i) for i in range(len(coefs))])

print(ff_x.roots(multiplicities=False))
```

Some interesting RSA problems

1. ECCRSA (TU Delft CTF 2024)
2. krsa (Intigriti CTF 2024)
3. Redundancy (vsCTF 2023)
4. RSA se olous RSei (NTUAH4CK 3.0)
5. RSA-2024 (imaginaryCTF monthly - Round 42)
6. RSATogether (ECSC 2024)
7. small eqs (0xL4ugh 2024)
8. QRSA (Grey Cat The Flag 2023)

ECCRSA (TU Delft CTF 2024)

A custom cryptosystem is implemented. It attempts to combine RSA and Elliptic Curve Cryptography. We notice that the only difference from a standard RSA encryption is that we are given the sum of 2 points with x cordinates p and q.

In [ ]:
```
#source.sage

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
#from flag import FLAG
FLAG = b"TUDELFT{TEST_FLAGITO}"

###### NIST P256
p256 = 2^256-2^224+2^192+2^96-1
a256 = p256 - 3
b256 = 41058363725152142129326129780047268409114441015993725554835256314039467
## Curve order
n = 11579208921035624876269744694940757352999695522413576034242225906106851204
FF = GF(p256)
EC = EllipticCurve([FF(a256), FF(b256)])
EC.set_order(n)

while True:
    try:
        p = random_prime(p256)
        P = EC.lift_x(p)

        q = random_prime(p256)
        Q = EC.lift_x(q)

        S = P + Q
        break
    except:
         pass

N = int(p * q)
e = 65537

phi = (p - 1) * (q - 1)
```

```
d = int(pow(e, -1, phi))

key = RSA.construct((int(N), int(e), int(d)))

print(f"{N = }")
print(f"{e = }")
print(f"{S = }")

cipher = PKCS1_OAEP.new(key)
ciphertext = cipher.encrypt(FLAG)

print(f"{ciphertext = }")
```

$\ell = \frac{y_2 - y_1}{x_2 - x_1}$ is the slope *

And thus we have the following equations:

$$n = p \cdot q$$
$$S_x = \ell^2 - p - q$$
$$S_y = \ell \cdot (p - S_x) - y_p$$

At this point I was 90% convinced that the system was well-constrained, expanding all equations I was pleased to find that this was indeed the case.

We can use some very common linear algebra tricks, namely solving the system using a Groebner basis, and then reducing it by applying consecutive resultants to be left out with only 1 equation.

For reference, these are the 5 equations that we can deduce from the data:

$$p_1 = y_p^2 - (x_p^3 + a \cdot x_p + b)$$

$$p_2 = y_q^2 - (x_q^3 + a \cdot x_q + b)$$

$$pol_1 = (y_q - y_p)^2 - x_p \cdot (x_q - x_p)^2 - x_q \cdot (x_q - x_p)^2 - S_x \cdot (x_q - x_p)^2$$

$$pol_2 = (y_q - y_p) \cdot (x_p - S_x) - y_p \cdot (x_q - x_p) - S_y \cdot (x_q - x_p)$$

$$pol_3 = N - x_p \cdot x_q$$

In [ ]:

```
#solution.sage
# from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP


###### NIST P256
p256 = 2^256-2^224+2^192+2^96-1
a256 = p256 - 3
b256 = 410583637251521421293261297800472684091144410159937255548352563140394674
## Curve order
n = 115792089210356248762697446949407573529996955224135760342422259061068512044
FF = GF(p256)
EC = EllipticCurve([FF(a256), FF(b256)])
```

```
EC.set_order(n)

N = 25326015765171801519732728044724586621889113099028049276749731207316326684
e = 65537
S = EC(6424979814120561780916022306781452703504287991228053623068943048839085
ciphertext = b'\x12\xed\xb1r\xb0L]\xcff\x9b\xb1o\x88\xd3\xc9\xac~P{\x0e\x1e\x1

Sx, Sy = S.xy()
# n = p*q
# Sx = l^2 - p - q
# Sy = l*(p - Sx) - Y(p)

a = a256
b = b256

P.<xp, xq, yp, yq> = PolynomialRing(FF)
p1 = yp^2 - (xp^3 + a*xp + b)
p2 = yq^2 - (xq^3 + a*xq + b)

pol1 = (yq - yp)^2 - xp*(xq - xp)^2 - xq*(xq - xp)^2 - Sx*(xq - xp)^2
pol2 = (yq - yp)*(xp - Sx) - yp*(xq - xp) - Sy*(xq - xp)
pol3 = N - xp*xq

I = P * (p1, p2, pol1, pol2, pol3)
V = I.groebner_basis()
# print monomials of the polynomials in the groebner basis to inspect them manu
print(*[Vi.monomials() for Vi in V], sep= '\n')
print(len(V))


V1, V2, V3, V4 = V[:4]

def resultant(p1, p2, var):
    p1 = p1.change_ring(QQ)
    p2 = p2.change_ring(QQ)
    var = var.change_ring(QQ)
    r = p1.resultant(p2, var)
    return r.change_ring(FF)


# Get rid of variables
h12 = resultant(V1, V2, xp)
h34 = resultant(V3, V4, xp)
h1234 = resultant(h12, h34, yp)
print(h1234.variables())

# this polynomial only has one variable, so finding roots is trivial
unipol = resultant(h1234, p2, yq).univariate_polynomial()

poss_xq = unipol.roots(multiplicities= False)
print(poss_xq)
for r in poss_xq:
    if N % int(r) == 0:
        print("success")
        p = int(r)
        e = 65537
        q = N//p
        print(f"{p = }")
        print(f"{q = }")
```

```
        assert is_prime(p), is_prime(q)
        assert p*q == N
        phi = (p - 1) * (q - 1)

        d = int(pow(e, -1, phi))

        key = RSA.construct((int(N), int(e), int(d)))
        cipher = PKCS1_OAEP.new(key)
        ciphertext = b'\x12\xed\xb1r\xb0L]\xcff\x9b\xb1o\x88\xd3\xc9\xac~P{\x0e

        message = cipher.decrypt(ciphertext)
        print(message)
        exit()
```

krsa (Intigriti CTF 2024)

This is a textbook RSA-2048 implementation with no twists.

This challenge simply requires to decrypt a ciphertext corresponding to a random 32-bit plaintext encrypted with a textbook RSA-2048 instance. While normally this would be bruteforcable, a tight timeout is enforced that prohibits exhaustive enumeration of all 32-bit messages that could possible produce the given ciphertext. In order to bypass this constraint, the solution is to employ a Meet-in-the-Middle approach, which decreases the amount of bruteforce needed from 2^32 to ~2^17 bits, a singificant optimation that makes the attack run in < 1a. The catch is that in order to carry out the MitM attack, the message needs to be able to be expressed as the product of two 16-bit numbers . While this is not guaranteed to be always the case, it occurs with highenough probability that simply resetting the server connection and making a new attempt is guaranteed to succeed within a few tries.

In [ ]:

```
#server.py
from Crypto.Util.number import *
import signal

def timeout_handler(signum, frame):
    print("Secret key expired")
    exit()

signal.signal(signal.SIGALRM, timeout_handler)
signal.alarm(300)

FLAG = "INTIGRITI{fake_flag}"
SIZE = 32

class Alice:
    def __init__(self):
        self.p = getPrime(1024)
        self.q = getPrime(1024)
        self.n = self.p*self.q
        self.e = 0x10001

    def public_key(self):
        return self.n,self.e

    def decrypt_key(self, ck):
```

```
        phi = (self.p-1)*(self.q-1)
        d = inverse(e, phi)
        self.k = pow(ck, d, n)

class Bob:
    def __init__(self):
        self.k = getRandomNBitInteger(SIZE)

    def key_exchange(self, n, e):
        return pow(self.k, e, n)

alice = Alice()
bob = Bob()

n,e = alice.public_key()
print("Public key from Alice :")
print(f"{n=}")
print(f"{e=}")

ck = bob.key_exchange(n, e)
print("Bob sends encrypted secret key to Alice :")
print(f"{ck=}")

alice.decrypt_key(ck)
assert(alice.k == bob.k)

try:
    k = int(input("Secret key ? "))
except:
    exit()

if k == bob.k:
    print(FLAG)
else:
    print("That's not the secret key")
```

In [ ]:

```
#sol.py
from pwn import *
from gmpy2 import mpz

def attempt():
    #conn = remote('krsa.ctf.intigriti.io', 1346)
    conn = process(["python", "server.py"])
    conn.recvline()
    n = mpz(int(conn.recvline().decode().split("=")[-1]))
    e = mpz(int(conn.recvline().decode().split("=")[-1]))
    conn.recvline()
    c = mpz(int(conn.recvline().decode().split("=")[-1]))
    conn.recvuntil(b"? ")

    forward = {}
    backward = {}

    for k in range(2**15, 2**16):
        f = pow(k, e, n)
        forward[f] = k
        b = c * pow(f, -1, n) % n
        backward[b] = k
```

```
    intersect = list(set(forward.keys()).intersection(set(backward.keys())))
    if intersect == []:
        conn.close()
        return
    print(intersect)
    k = intersect[0]
    m = forward[k]*backward[k]
    print(m, m.bit_length())
    conn.sendline(str(m).encode())
    print(conn.recvline())
    exit()


while True:
    attempt()
```

Redundancy (vsCTF 2023)

This challenge encrypts the same message with the same modulo, but with different exponents $e1, e2$. Our first key observation is that both exponents are extremely small, and they have 5 as a common factor.

We are also given quit a lot of bytes on the MSB of the message, something that always hints at lattice approaches.

Therefore, we can conclude that this is a twist on two standard RSA attacks. Each one individually is not enough to break the system, but chaining them in the right way makes decryption possible. Specifically:

- A message is encrypted twice with a common modulus but **without** the two public exponents being coprime.
- A known prefix is added to the message before encryption.

In [ ]:

```
#chall.py
from flag import flag

from Crypto.Util.number import getPrime as gP

e1, e2 = 5*2, 5*3
assert len(flag) < 16
flag = "Wow good job the flag is (omg hype hype): vsctf{"+flag+"}"
p = gP(1024)
q = gP(1024)
n = p * q

m = int.from_bytes(flag.encode(), 'big')
c1 = pow(m, e1, n)
c2 = pow(m, e2, n)
print(f"n = {n}")
print(f"c1 = {c1}")
print(f"c2 = {c2}")
```

Since the public exponents are not coprime, the standard "common modulus" attack cannot be used to directly recover the message. However, it can be employed to caclulate the encryption of the initial message as if it were encrypted with the gcd of the two actual exponents.

Thus, it is trivial to calculate $m^5 \mod n$.

In combination with the given prefix, it enables a Coppersmith short-pad attack to be carried out, by significantly decreasing the degree of the polynomial.

Since the flag is so small, it is possible to just iterate through all lengths until small_roots finds a solution.

In [ ]:

```
#solve.py
from sage.all import *
from Crypto.Util.number import bytes_to_long, long_to_bytes

n = 170177484387050664859802656105049739416895071582140489079348640539518248890
c1 = 9003062544361468960014218470636404669173735044866342965869660382166263123
c2 = 2546072448640808612556238065690407010381885201320761372614998667179031247
e1, e2 = 5*2, 5*3

e3, u, v = xgcd(e1, e2)
print(f"New exp: {e3}")
c3 = pow(c1, u, n) * pow(c2, v, n) % n

# we know these about the message:
# assert len(flag) < 16
# flag = "Wow good job the flag is (omg hype hype): vsctf{"+flag+"}"

prefix = bytes_to_long(b"Wow good job the flag is (omg hype hype): vsctf{")
suffix = ord("}")
PP = PolynomialRing(Zmod(n), "x")
x = PP.gen()
for flaglen in range(1, 16):
    pol = (prefix*256**(flaglen + 1) + x*256 + suffix)**e3 - c3
    pol = pol.monic()
    sroot = pol.small_roots(X= 256**flaglen)
    if sroot == []:
        continue
    flag = long_to_bytes(int(sroot[0])).decode()
    print(f"Found flag for flag length {flaglen}")
    print(f"vsctf{{{flag}}}")
```

RSA se olous RSei (NTUAH4CK 3.0)

This RSA challenge combines an unconventinal method of encoding messages as integers, as well as using a small public exponent (e = 3).

More specifically:

$$m = \prod_{i=1}^{\mathrm{LEN}} \left(s_i^{f_{\mathrm{LEN}-i}} \mod n \right),$$

where f is the flag byte array and LEN is the flag length.

In [ ]:
```python
# source.py

from Crypto.Util.number import getPrime
from math import prod
from sympy import sieve
from secret import FLAG

FLAGLEN = 18
assert len(FLAG) == FLAGLEN

p = getPrime(2048)
q = getPrime(2048)
n = p*q
e = 3
m = prod(pow(sieve[i], FLAG[FLAGLEN - i], n) for i in range(1, FLAGLEN + 1))
c = pow(m, e, n)
print(f"{n = }")
print(f"{c = }")

'''
n = 3213495155903142066539758954326431610241987253645020979012156315646031772
c = 2613327202261379765303581377851987570898720777379472354946711996838317344
'''
```

To attack it, the RSA homomorphic properties can be leveraged, in combination with known properties of the plaintext format. This enables us to "chip" away bytes both by the flag format (`NH4CK{...}`), and the fact that $f_i > 32$

Multiplying with the inverse of s_i^{32} and the inverse of the flag format, leaves us with minimal bruteforce to be done.

In [ ]:
```python
# solution.py

from gmpy2 import iroot
from sympy import factorint, sieve

n = 3213495155903142066539758954326431610241987253645020979012156315646031772
c = 2613327202261379765303581377851987570898720777379472354946711996838317344

e = 3
FLAGLEN = 18
```

```
known_primes = sieve[1:FLAGLEN + 1][::-1]

known = b"NH4CK{"
smallest_ascii = b"!"
smallest_ascii_val = smallest_ascii[0]
flag_dict = {}
smallest_flag = known + smallest_ascii*(FLAGLEN - 1 - len(known)) + b"}"

for p, char in zip(known_primes, smallest_flag):
    c = c * pow(p, -e * char, n) % n
    flag_dict[p] = char



bfsize = 2
next2primes = known_primes[len(known):len(known) + bfsize]

for num1 in range(128 - smallest_ascii_val):
    c1 = c * pow(next2primes[0], -e * num1, n) % n
    for num2 in range(128 - smallest_ascii_val):
        c2 = c1 * pow(next2primes[1], -e * num2, n) % n
        root, check = iroot(c2, e)
        if check:
            flag_dict[next2primes[0]] += num1
            flag_dict[next2primes[1]] += num2
            print("success", c2.bit_length(), num1, num2)
            facs = factorint(root)
            flag = ''
            for sp in sieve[1:FLAGLEN + 1]:
                num = facs.get(sp, 0)
                flag += chr(flag_dict[sp] + num)
            print(flag[::-1])
            exit()
```

RSA-2024 (imaginaryCTF monthly - Round 42)

While this initially seems like a simple challenge, it is deceptively complicated, since we aren't given the value of e. To solve it, it is required to approach RSA in an unconventional means. While we are used to thinking about the order of the group used in standard RSA instances using Euler's phi, the key to solving the challenge is to instead utilize Carmichael's lambda function.

In [ ]:

```
from Crypto.Util.number import *
FLAG = b'ictf{REDACTED}'

print("Let's build an RSA-2024 public key together! I provide the exponent, you
e = getRandomNBitInteger(2024)
N = int(input("N = "))
assert e.bit_length() == N.bit_length() == 2024, "We failed to collaborate on a

m = bytes_to_long(FLAG)
c = pow(m, e, N)
print(f"{c = }")
```

Since this value divides phi, and the server doesn't check in any way that the modulus N we provide is the product of two primes, we can instead construct a "malicious" value of N, with as small Carmichael's lambda as possible. Finally, since the value of lambda is so small, then we can enumerate all possible values of the secret exponent, and attemp to decrypt the message until we get a value of the desired format (i.e. printable english).

In [ ]:
```
#sol.py
from pwn import *
from Crypto.Util.number import *
from sympy import sieve
from sage.all import carmichael_lambda, factor, is_prime, is_prime_power, euler
from itertools import chain, combinations, product
from math import prod
from tqdm import trange
from gmpy2 import mpz
from random import randint, choices



def powerset(iterable):
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(1, len(s)+1))


# precalculate N of appropriate bit length and as small Carmichael's lambda as
target_bitlength = 2024
small_primes = list(sieve[1:7]) + [17]
uses = [6, 4, 2, 2, 2, 2, 2]
assert len(small_primes) == len(uses)
prime_uses = {s: u for s, u in zip(small_primes, uses)}

print(small_primes)
diff_primes = set()
cnt = 0
all_subsets = powerset(small_primes)
for subset in all_subsets:
    for up in product(*[list(range(i)) for i in [prime_uses[p] for p in subset]
        potp = prod([subset[i]**up[i] for i in range(len(subset))]) + 1
        if is_prime(potp):
            if potp not in diff_primes:
                diff_primes.add(potp)

diff_primes = list(diff_primes)
diff_primes = sorted(diff_primes, key = lambda num: num.bit_length())
s = 124
mul = 1
for np in diff_primes[::-1]:
    mul *= np
    if mul.bit_length() > target_bitlength:
        mul //= np
        break

small_primes = diff_primes[:40]
while True:
    tN = mul * prod(choices(small_primes, k= randint(1, 10)))
    if tN.bit_length() == target_bitlength:
        break
```

```
cl = carmichael_lambda(tN)
print(factor(cl))
print(f"{cl = }")
print(f"{cl.bit_length()}")

while True:
    conn = process(["python", "server.py"])
    conn.sendlineafter(b"= ", str(tN).encode())
    c = int(conn.recvline().decode().strip().split()[-1])
    conn.close()

    c, n = mpz(c), mpz(tN)
    c0 = c
    for d in trange(cl):
        c = c*c0 % n
        flag = long_to_bytes(c)
        if flag.startswith(b"ictf{"):
            print(flag)
            exit()
```

RSATogether (ECSC 2024 jeopardy)

The setting of this challenge involves an RSA secret that is "Shamir-Secret-Shared" among many participants. A slight mistake in the implementation of the secret sharing allows the attacker to use a polynomial of degree smaller than the amount of shares they're given. This in turn enables them to recover the secret using some clever linear algebra and undo the RSA encryption.

In [ ]:
```
#rsatogether.py

#!/usr/bin/env sage

from Crypto.Util.number import getPrime, bytes_to_long
import random
import os

random = random.SystemRandom()

flag = os.getenv("FLAG", "ECSC{testflag}")

def gen_key(n_bits):
    p = getPrime(n_bits//2)
    q = getPrime(n_bits//2)
    n = p*q
    phi = (p-1)*(q-1)
    e = 65537
    d = pow(e, -1, phi)

    return phi, d, n, e

def eval_poly(poly, x, n):
    return sum(pow(x, i, n) * poly[i] for i in range(len(poly))) % n

def create_shares(phi, poly):
    n_shares = int(input("With how many friends you want to share the private k
```

```
    if n_shares < 1:
        print("Don't be mean, sharing is caring!")
        exit()
    elif n_shares > 101:
        print("Come on, you don't have that many friends...")
        exit()

    n_shares += 1 # you also get one part of the key, don't worry
    poly = poly[:n_shares]
    ys = [eval_poly(poly, i, phi) for i in range(1, n_shares+1)]
    M = matrix(ZZ, [[x**i for i in range(n_shares)] for x in range(1, n_shares+
    coeffs = M.solve_left(vector(ZZ, [1] + [0]*(n_shares - 1)))

    shares = [(c*y) % phi for c,y in zip(coeffs, ys)]
    yours = shares.pop(n_shares - 2)
    print(f"Here is your part: {yours}")

    return shares


def comput_partial_decryption(c, shares, n):
    return [pow(c, s, n) for s in shares]


n_bits = 2048
phi, d, n, e = gen_key(n_bits)
print(f"{n = }")
print(f"{e = }")

poly = [d] + [random.getrandbits(n_bits) for _ in range(99)]
shares = create_shares(phi, poly)

while True:
    choice = int(input("""
Select:
1) Decrypt something
2) Reshare
3) That's enough
> """))
    if choice == 1:
        c = int(input("Ciphertext: "))
        partial_dec = comput_partial_decryption(c, shares, n)
        print("Here are the partial decryptions of your friends!")
        for pt in partial_dec:
            print(pt)
    elif choice == 2:
        shares = create_shares(phi, poly)
    elif choice == 3:
        break

pad_flag = os.urandom((n_bits - 8)//8 - len(flag)) + flag.encode()
print(f"Bye bye, take this with you!\n{pow(bytes_to_long(pad_flag), e , n)}")
```

In [ ]:

```
#solve.py
from sage.all import *
from pwn import *
from Crypto.Util.number import getPrime, bytes_to_long, long_to_bytes
```

```
from tqdm import tqdm
from gmpy2 import mpz, gcd

def get_num(conn):
    return int(conn.recvline().decode().strip().split()[-1])

def get_flag(conn):
    conn.sendlineafter(b"> ", b"3")
    conn.recvline()
    enc = get_num(conn)
    return enc

def decrypt_from_d(conn, d, enc= None):
    if enc == None:
        enc = get_flag(conn)
    pt = long_to_bytes(pow(enc, d, n))
    if b"ECSC" in pt:
        print(pt)
    return enc, pt

def reshare(conn, num):
    conn.sendlineafter(b"> ", b"2")
    conn.sendlineafter(b"? ", str(num).encode())
    share = get_num(conn)
    return share

def gcd_list(llist):
    if len(llist) == 2:
        return gcd(llist[0], llist[1])
    return gcd_list([gcd(llist[0], llist[1])] + llist[2:])

conn = process(["sage", "rsatogether.sage"])
#conn = remote("rsatogether.challs.jeopardy.ecsc2024.it", 47001)
n = get_num(conn)
e = get_num(conn)

conn.sendlineafter(b"? ", b"2")
get_num(conn)

FF = QQ
size = 100
M1 = matrix(FF, size, size)
M2 = matrix(FF, size, size)
v1 = vector(FF, size)
v2 = vector(FF, size)

for i in tqdm(range(size+1)):
    nshares = i + 2
    M = matrix(ZZ, [[x**i for i in range(nshares)] for x in range(1, nshares+1)
    coeffs = M.solve_left(vector(ZZ, [1] + [0]*(nshares - 1)))
    coeffs = [int(ii) for ii in coeffs]
    mycoeff = coeffs[-2]

    polyy = [1]*size
    polyy = polyy[:nshares]
    polyy += [0]*(size - len(polyy))

    my_x = nshares - 1
    share = reshare(conn, my_x)
```

```
    if i < size-1:
        v1[i] = share
        v2[i] = share
    elif i == size-1:
        assert v1[-1] == 0
        v1[-1] = share
    elif i == size:
        assert v2[-1] == 0
        v2[-1] = share
    coeff = nshares * (-1)**nshares
    assert coeff == mycoeff

    if i < size-1:
        for j in range(size):
            M1[i, j] = polyy[j]*coeff*(my_x**j)
            M2[i, j] = polyy[j]*coeff*(my_x**j)
    elif i == size-1:
        for j in range(size):
            assert all(ii == 1 for ii in polyy)

            M1[-1, j] = polyy[j]*coeff*(my_x**j)
    elif i == size:
        for j in range(size):
            M2[-1, j] = polyy[j]*coeff*(my_x**j)

enc = get_flag(conn)

print("[+] Solving ... this will take some time ...")
R1 = (M1.augment(v1)).rref().column(-1)
R2 = (M2.augment(v2)).rref().column(-1)
RD = R1 - R2
common_denom = prod([rd.denominator() for rd in RD])
RDD = common_denom*RD
maybe_phi = gcd_list([int(ii) for ii in RDD])
my_d = pow(e, -1, maybe_phi)
decrypt_from_d(conn, my_d, enc)
```

small_eqs (0xL4ugh CTF 2024)

This challenge uses an unorthodox method for generating 2 out of the 3 primes used to compose the public modulus of this multiprime RSA instance. By abusing the relation between the 2 primes we cam bruteforcing the unknown of small size and find a multiple of a divisor of the order of the quotient ring F_x^2/(some random polynomial). We can then raise a random element to that value and get a multiple of one of the primes. From there, caclulating the other 2 primes and decrypting the message is trivial.

In [ ]:
```
# chall.py
from Crypto.Util.number import getPrime, isPrime, bytes_to_long


p=getPrime(512)
while True:
    w=getPrime(20)
    x=2*w*p-1
    if isPrime(x):
```

```
        break

q=getPrime(512*2)
n = p * q * x
e = 65537
m = bytes_to_long(b'redacted')
c = pow(m, e, n)
print(f"{n = }")
print(f"{e = }")
print(f"{c = }")
print(w)

'''
n = 1818667284960960333134418258456864294107889310480230121724102862446960702l7
e = 65537
c = 161799929355762072415753553777874133500465628665513459757970683869056617845
'''
```

In [ ]:

```
from Crypto.Util.number import long_to_bytes
from sage.all import gcd, PolynomialRing, Zmod
from gmpy2 import next_prime
from random import randint
from tqdm import tqdm
from multiprocessing import Pool
import os

n = 1818667284960960333134418258456864294107889310480230121724102862446960702l7
e = 65537
c = 161799929355762072415753553777874133500465628665513459757970683869056617845


def process_prime(prime):
    t = a**(2*prime*n)
    for i, ele in enumerate(list(t)):
        res = gcd(int(ele), n)
        if res != 1 and res != n:
            print(res, "success", i, ele, prime)
            p2 = res
            assert n % p2 == 0
            p1 = (p2 + 1)//2//prime
            assert n % p1 == 0
            p3 = n//p1//p2
            phi = (p1 - 1)*(p2 - 1)*(p3 - 1)
            d = pow(e, -1, phi)
            flag = long_to_bytes(int(pow(c, d, n)))
            print(flag)
            return True
    return False

if __name__ == "__main__":

    Zn = Zmod(n)
    PR = PolynomialRing(Zn, 'x')
    x = PR.gen()

    primes20 = [next_prime(2**19)]
    while True:
        primes20.append(next_prime(primes20[-1]))
```

```
        if primes20[-1] > 2**20:
            primes20 = primes20[:-1]
            print(f"{len(primes20) = }")
            break

    while True:
        d1 = randint(0, n)
        d2 = randint(0, n)
        QR = PR.quotient_ring(x**2 + d1*x + d2)
        a = QR.random_element()
        with Pool(os.cpu_count() - 2) as pool:
            results = list(tqdm(pool.imap(process_prime, primes20), total=len(p
        if sum(results) != 0:
            break
    print('done')
```

QRSA (Grey Cat The Flag 2023)

This challenge implements RSA in $\mathbb{Q}[\sqrt{41}]$. After some searching we can find this paper, https://www.diva-portal.org/smash/get/diva2:1170568/FULLTEXT01.pdf which explains that if the norm of N can be factorized, then we can recover phi, and effectively recover the message (indeed, our N is very smooth). The catch is that phi is not calculated like Z because the structure of the group differs. But, we can follow 5.2 to calculate phi.

Note that $\mathbb{Q}[\sqrt{41}]$ is a UFD.

So,

$$\mathbb{Q}[d]/x = \mathbb{Q}[d]/y_1^{a_1} \cdot \mathbb{Q}[d]/y_2^{a_2} \cdots \mathbb{Q}[d]/y_n^{a_n}$$

If $y \in \mathbb{Z}$, then $\mathrm{Norm}(y) = p^2$ with p being prime.

$$\mathrm{ord}(\mathbb{Q}[d]/y^a) = (p^2 - 1) \cdot p^{2(a-1)}$$

If $y \notin \mathbb{Z}$, then $\mathrm{Norm}(y)$ is prime.

$$\mathrm{ord}(\mathbb{Q}[d]/y^a) = (p - 1) \cdot p^{a-1}$$

Since the norm is multiplicative:

$$\mathrm{Norm}(x) = \mathrm{Norm}(y_1)^{a_1} \cdot \mathrm{Norm}(y_2)^{a_2} \cdots \mathrm{Norm}(y_n)^{a_n}$$

In [ ]:

```
# main.py

from Crypto.Util.number import bytes_to_long
from secret import qa, qb, pa, pb

FLAG = b'fake_flag'

class Q:
    d = 41
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

```python
    def __add__(self, other):
        return Q(self.a + other.a, self.b + other.b)

    def __sub__(self, other):
        return Q(self.a - other.a, self.b - other.b)

    def __mul__(self, other):
        a = self.a * other.a + Q.d * self.b * other.b
        b = self.b * other.a + self.a * other.b
        return Q(a, b)

    def __mod__(self, other):
        # Implementation Hidden
        # ...
        return self

    def __str__(self) -> str:
        return f'({self.a}, {self.b})'

def power(a, b, m):
    res = Q(1, 0)
    while (b > 0):
        if (b & 1): res = (res * a) % m
        a = (a * a) % m
        b //= 2
    return res

p, q = Q(pa, pb), Q(qa, qb)
N = p * q
m = Q(bytes_to_long(FLAG[:len(FLAG)//2]), bytes_to_long(FLAG[len(FLAG)//2:]))
e = 0x10001
c = power(m, e, N)

print(f"N_a = {N.a}")
print(f"N_b = {N.b}")
print(f"C_a = {c.a}")
print(f"C_b = {c.b}")
print(f"e = {e}")
print(f"D = {Q.d}")

'''
N_a = 2613240571441392195964088630982261349682821645613497396226742971850092862
N_b = 40663129138106306270836864062443319517762988712832499215653621542242708852
C_a = 25487111945839052428384829000782948591998824843752299647155504697907674160
C_b = 40094115814829986666511543614608455529715264691422343398829396189384820677
e = 65537
D = 41
'''
```

In [ ]:

```python
# solver.sage

N_a = 2613240571441392195964088630982261349682821645613497396226742971850092862
N_b = 40663129138106306270836864062443319517762988712832499215653621542242708852
C_a = 25487111945839052428384829000782948591998824843752299647155504697907674160
C_b = 40094115814829986666511543614608455529715264691422343398829396189384820677
e = 65537
D = 41
```

```
Q = QuadraticField(D, name='d')
d = Q.gen()
N = N_a + d*N_b
C = C_a + d*C_b


# fs = factor(N.norm())
# print(list(fs))

fs = [(2, 100), (3, 30), (5, 14), (7, 8), (11, 2), (19, 2), (23, 3), (29, 2), (

phi = 1
for (i, pwr) in fs:
    i = int(i)
    phi *= i**(pwr - 1) * (i**2 - 1)

Q_modN.<dmod> = Q.quo(N)

C_mod = C_a + dmod * C_b



rsa_d = pow(e, -1, phi)
res = (C_mod ^ rsa_d).lift()
coeffs = list(res)

flag = ''
for i in coeffs:
    flag += (int(i)).to_bytes(30, 'big').decode()

print(flag)
```