Twenty Years of Attacks on the RSA Cryptosystem

This is an attempt to implement the attacks described in the famous paper Twenty Years of Attacks on the RSA Cryptosystem. The primary language of choice is python, and more specifically SageMath.

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
```

```
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 [ ]:
```
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 = 0x10001
d = pow(e, -1, (p -1) * (q - 1))



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

In [ ]:
```
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:

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

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

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

In [ ]:
```python
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 [ ]:
```python
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

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 $\frac{e}{N}$

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('dd')
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))
```