Chronophobia - Lessons learned

Why we could not solve chronophobia... Analysis for Coppersmith method more

What I Learned

- It's better to try multiple different strategies rather than focusing so much on the detailed analysis in search of an efficient solution strategy.
- Simple strategy is good, but tuning parameters not good. Applying past works is definitely better in terms of time efficiency. I could've used code/parameters from older CTF challenge's with a similar intended solution, but I second guessed my intuition too much.
- Always go with your gut... Trust your gut.

chronophobia (from idekCTF 2022)

```python
#!/usr/bin/env python3

from Crypto.Util.number import *
import random
import signal

class PoW():

    def __init__(self, kbits, L):

        self.kbits = kbits
```

```python
        self.L = L

        self.banner()
        self.gen()
        self.loop(1337)

    def banner(self):

        print("===================================")
        print("=== Welcome to idek PoW Service ===")
        print("===================================")
        print("")

    def menu(self):

        print("")
        print("[1] Broken Oracle")
        print("[2] Verify")
        print("[3] Exit")

        op = int(input(">>> "))
        return op

    def loop(self, n):

        for _ in range(n):

            op = self.menu()
            if op == 1:
                self.broken_oracle()
            elif op == 2:
                self.verify()
            elif op == 3:
                print("Bye!")
                break
```

```
    def gen(self):

        self.p = getPrime(self.kbits)
        self.q = getPrime(self.kbits)

        self.n = self.p * self.q
        self.phi = (self.p - 1) * (self.q - 1)

        t = random.randint(0, self.n-1)
        print(f"Here is your random token: {t}")
        print(f"The public modulus is: {self.n}")

        self.d = random.randint(128, 256)
        print(f"Do 2^{self.d} times exponentiation to get the valid ticket t^(2^(2^{self.d})) %
n!")

        self.r = pow(2, 1 << self.d, self.phi)
        self.ans = pow(t, self.r, self.n)

        return

    def broken_oracle(self):

        u = int(input("Tell me the token. "))
        ans = pow(u, self.r, self.n)
        inp = int(input("What is your calculation? "))
        if ans == inp:
            print("Your are correct!")
        else:
            print(f"Nope, the ans is {str(ans)[:self.L]}... ({len(str(ans)[self.L:])} remain
digits)")

        return

    def verify(self):
```

```python
        inp = int(input(f"Give me the ticket. "))

        if inp == self.ans:
            print("Good :>")
            with open("flag.txt", "rb") as f:
                print(f.read())
        else:
            print("Nope :<")

if __name__ == '__main__':

    signal.alarm(120)
    service = PoW(512, 200)
```

We are given token `t`, exponent `d`, and modulus `n=p*q`. The goal is to find `t^r % n` for `r=2^(2^d) % phi(n)` within 2 minutes. The assumption of Wesolowski's verifiable delay function (Efficient verifiable delay functions) is that this type of computation is slow if factorization of `n` is unknown. So the author may add an extra interface. We are given weird oracle "broken_oracle", which outputs most significant `L` digits for `u^r % n` given user input `u`.

Our Strategy on CTF

I saw the challenge after having nice progress by @soon_haari. His idea is:

1. obtain `u1=broken_token(t)`
2. obtain `u2=broken_token(t^2 % n)`
3. find rest of digits of `u` by LLL/BKZ

If we assume that `u=u1*(10^Ludown)+x`, `u^2 % n=u2*(10^Lu2down)+y`, then

$$(u1 \cdot (10^{\text{Ludown}}) + x)^2 - (u2 \cdot (10^{\text{Lu2down}}) + y) = 0 \pmod{n}$$

`x,y` are small ($\leq 10^{\text{Ludown}}, 10^{\text{Lu2down}}$), we may expect LLL could solve the challenge.
It is nice, but it only works `L=250`. In our setting, we have to solve it for `L=200`.

Then, I started tuning lattice, but it failed. And I tried to apply another idea: using `u^-1 % n` instead of `u^2 % n`. Even though it solved it for `L=210`, but it did not work for `L=200` ...

After that, I changed mind. I assume that these strategy does not work cause some high degree terms (`x^2,x*y` etc.) are included. So I determined to apply another method: **Coppersmith method**.

Coppersmith method is general framework for solving a polynomial equation over integer, $\mod N$ (not on finite field), and $\mod p$ for unknown modulus $p \mid N$. On Sagemath, small_roots method is implemented, but it only works for 1 variable polynomial. But we have alternative experimental extension by @defund. Then, I wrote just like the following code. (I clean up after ctf, but the essence is same.)

```
from sage.all import *

# defund/coppersmith
load("coppersmith.sage")

def solve(u1, Ludown, u2, L2udown, n):
    polyrng = PolynomialRing(Zmod(n), 2, "xy")
    x,y = polyrng.gen()
    f = (u1*(10**Ludown)+x)**2 - (u2*(10**L2udown)+y)

    print("computing small_root...")
    result = small_roots(f, [10**Ludown, 10**L2udown], m=2, d=2)
    if result == []:
        return None
    print(result)

    want_result_0 = int(int(result[0][0])%n)
    want_result_1 = int(int(result[0][1])%n)
```

```
    print((want_result_0, want_result_1))

    ans = u1*(10**Ludown)+want_result_0
    ans_2 = u2*(10**L2udown)+want_result_1
    assert (ans**2 - ans_2) % n == 0

    return ans
```

I run the code and it outputted some result within few seconds. But it did not pass answer checking for some reason. I manipulated small_roots parameters, but it did not change the status. And I added some small bruteforce for most significant digits for `x, y`, but did not... What can I do?

After CTF ended, I saw the writeup by @maple3142. I was both astonished and mildly upset, because **the method is almost same** except for using another alternative Coppersmith extension lattice-based-cryptanalysis by @joseph instead of defund's version. And his code includes the comment:

```
sys.path.append("./lattice-based-cryptanalysis")

# idk why defund/coppersmith doesn't work...
# need to remove `algorithm='msolve'` from solve_system_with_gb
from lbc_toolkit import small_roots
```

OK... I have to analyze why we were wrong...

Note: The intended solution is to use hidden number problem with some manipulation: (chronophobia). It is also good way for avoiding high degree terms.

Introduction to Coppersmith Method

Then, I review Coppersmith method. Recently, sophisticated overview has published: A Gentle Tutorial for Lattice-Based Cryptanalysis, J. Surin and S. Cohney, 2023. So I skip basics of lattice except citing the following theorem.

Theorem [LLL: Lenstra, Lenstra, Lovasz]

Let L be an integer lattice of $\dim L = \omega$. The LLL algorithm outputs a reduced basis spanned by $\{v_1, \ldots, v_\omega\}$ with

$$\|v_1\| \leq \|v_2\| \leq \ldots \leq \|v_i\| \leq 2^{\frac{\omega(\omega-i)}{4(\omega+1-i)}} \cdot \det L^{\frac{1}{\omega+1-i}} \ (i = 1, \ldots, \omega)$$

in polynomial time in ω and entries of the basis matrix for L.

Especially, LLL finds a short vector v_1 such that $\|v_1\| \leq 2^{\frac{\omega-1}{4}} \cdot \det L^{\frac{1}{\omega}}$, that is, v_1 is some multiples of $\det L^{\frac{1}{\dim L}}$. The multiples are called approximation factor. The multiples could be large, but in practice we may obtain much smaller vector (maybe, not shortest, though). So for analyzing lattice, firstly consider of $\det L^{\frac{1}{\dim L}}$.

Then, I focus Coppersmith method.
For introduction, we assume we want to solve the following equation. The modulus `N` is the product of some two 512-bit primes.

```
x^2 +
1596058470571678521138415442954622180023833193841383628246558842756751148302767004698706810428218010382683228651646908385821063994954285795515864223053218134321393365750798455962869048375466526653345993796536631700075252303184643664965293694411905687695249804270166236173641934842157432185973838101780307015 05*x +
```

```
665334599379653663170007525230318464366496529369441190568769524980427016623617364193484215743218
633044486831389275043
```

If we could solve this type of equation in general, we could factor `N` efficiently and could break RSA! (It would be impossible.) But, luckily, we can deduce the following equation by just subtracting `N*x+N`:

$$x^2 - 35660676653358573538x - 1006707748483862406125449872514995205080 = 0$$

If the solution `x0` is small, we can assume that the modulus solution `x0` can be find by just solving over integer. (The modulus equation can be reduced to infinitely integer equations $= 0, = \pm N, \pm 2N, \ldots$, but $= 0$ is only case if `x0` is small enough.) Solving modulus equation is hard, but solving integer equation is easier. In fact, Sagemath solve it in seconds.

```
sage: P=PolynomialRing(ZZ, 'x')
sage: x=P.gens()[0]
sage: f= x^2 -35660676653358573538*x-1006707748483862406125449872514995205080
sage: f.roots()
[(54225787401085700998, 1), (-18565110747727127460, 1)]
```

This is the essence of Coppersmith method: reducing modulus equation to *small* integer equation.

Let's state Howgrave-Graham theorem. First, let $h(x_1, x_2, \ldots, x_n) = \sum_{(i_1, i_2, \ldots, i_n)} h_{i_1, i_2, \ldots, i_n} {x_1}^{i_1} \cdot {x_2}^{i_2} \cdots {x_n}^{i_n} \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$. And $X_1, X_2, \ldots, X_n \in \mathbb{Z}_{>0}$. Then, we define

$$\|h(x_1 X_1, \ldots, x_n X_n)\|_2 := \sqrt{\sum_{(i_1, i_2, \ldots, i_n)} \left(h_{i_1, i_2, \ldots, i_n} {X_1}^{i_1} \cdot {X_2}^{i_2} \cdots {X_n}^{i_n}\right)^2}$$

Then, we can prove the following:

Theorem: Howgrave-Graham

Let N is a positive integer, $h(x_1, x_2, \ldots, x_n) = \sum_{(i_1, i_2, \ldots, i_n)} h_{i_1, i_2, \ldots, i_n} {x_1}^{i_1} \cdot {x_2}^{i_2} \cdots {x_n}^{i_n} \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$, and the number of monomials $\omega = \#\{(i_1, i_2, \ldots, i_n) \mid h_{i_1, i_2, \ldots, i_n} \neq 0\}$. If

1. $h(r_1, \ldots, r_n) = 0 \pmod{N}$ for some $|r_1| < X_1, \ldots, |r_n| < X_n$
2. $\|h(x_1 X_1, \ldots, x_n X_n)\|_2 < \frac{N}{\sqrt{\omega}}$

are satisfied, then $h(r_1, \ldots, r_n) = 0$ holds over the integers.

Proof

$$|h(r_1, r_2, \ldots, r_n)| = \left| \sum_{(i_1, \ldots, i_n)} h_{i_1, \ldots, i_n} {r_1}^{i_1} \cdots {r_n}^{i_n} \right| \leq \sum_{(i_1, \ldots, i_n)} |h_{i_1, \ldots, i_n} {X_1}^{i_1} \cdots {X_n}^{i_n}| \leq \sqrt{\omega} ||h(x_1 X_1, \ldots, x_n X_n)||_2 < N$$

The last inequality follows from Cauchy-Schwaltz inequality. ■

Note

On the proof of above, we uses Cauchy-Schwaltz for obtaining the condition about $\|\cdot\|_2$ (L2-norm). But, obviously, it is sufficient to check the condition $\|h(x_1 X_1, \ldots, x_n X_n)\|_1 := \sum_{(i_1, i_2, \ldots, i_n)} |h_{i_1, i_2, \ldots, i_n} {X_1}^{i_1} \cdot {X_2}^{i_2} \cdots {X_n}^{i_n}| < N$. We will use the L1-norm condition for checking obtaining polynomials are good or not. ■

Then, if we want to find a solution $(r_1, \ldots, r_n) \in \mathbb{Z}^n$ for $f(x_1, \ldots, x_n) = 0 \pmod{N}$ given $|r_1| < X_1, \ldots, |r_n| < X_n$, we do the following:

1. Collect polynomials $g_i(x_1, \ldots, x_n)$ which satisfies $g_i(r_1, \ldots, r_n) = 0 \pmod{N^t}$ for fixed $t \geq 1$
2. Find polynomials $h_1, \ldots, h_n$ which satisfies Howgrave-Graham condition for modulus N^t. h_j are found by LLL for the lattice generated by the coefficients for $g_i(x_1 X_1, \ldots, x_n X_n)$. ($h_j$ is linear combination of g_i)

3. Find $(r_1, \dots, r_n)$ by solving h_j over the integer

On first introductory example, $g_1 = f, g_2 = N, g_3 = Nx$ (all satisfies $= 0 \pmod N$), $h = g_1 - g_2 - g_3$. But in general, we might not find small polynomial by using only f and Nx^i (so we consider not only $t = 1$ but $t \geq 1$). Thus, Coppersmith introduced shift polynomial:

Shift Polynomial (Coppersmith)

For $f(x_1, \dots, x_n) \in \mathbb{Z}[x_1, \dots, x_n]$ and the modulus N,

$$g_{i_f, i_N, j_1, \dots, j_n} := f^{i_f} \cdot N^{i_N} \cdot {x_1}^{j_1} \dots {x_n}^{j_n}$$

for $i_f, i_N, j_1, \dots j_n \geq 0$ and $i_f + i_N \geq t$ are called as shift polynomials for f.
If $(r_1, \dots, r_n) \in \mathbb{Z}^n$ is a solution for $f \pmod N$, then $g_{i, j_1, \dots, j_n}(r_1, \dots, r_n) = 0 \pmod{N^t}$. ■

Though powering of f increases involving monomials, it generates many g_i, so we may expect we can find good h_j. The drawback of this is that computation complexity of LLL is high if too many g_i are involved.

So we should choose good shift polynomials and tweak some modification for each tasks. We will analyze each case.

Univariate case

The paper Finding Small Solutions to Small Degree Polynomials, D. Coppersmith, 2001 states the following:

Theorem: Coppersmith

Let N is a (large) positive integer, which has a divisor $b \geq N^\beta, 0 < \beta \leq 1$. Let $f(x)$ be a univariate polynomial of degree δ, where the leading coefficient f is invertible over $\mod N$. And let $0 < X$ for an expected bound for a root of $f(x)$. Then, we can find a solution r of the equation

$$f(r) = 0 \pmod{b} \ (|r| < X)$$

, if around $X < 1/2N^{\beta^2/\delta}$.

Proof

The leading coefficient of $f(x)$ is invertible, we can assume $f(x)$ is monic by multiplying inverse of leading coefficient of $f(X)$ over $\mod N$.
Write $f(x) = x^\delta + f_{\delta-1}x^{\delta-1} + \cdots + f_0$. Let t, u are some non-negative integers (tuned later).
Let consider the following lattice L (row vectors):

$$\begin{pmatrix} X^{t\delta+u-1} & * & \dots & \dots & \dots & \dots & * \\ 0 & X^{t\delta+u-2} & * & \dots & \dots & \dots & * \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & X^{t\delta} & * & \dots & * \\ 0 & 0 & 0 & 0 & NX^{t*\delta-1} & \dots & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & N^t \end{pmatrix}$$

Each row vector corresponds to:

- $x^i f(x)^t \ (i = u-1, \dots, 0)$
- $x^j f(x)^{t-i} N^i \ (i = 1, \dots, t, \ j = \delta - 1, \dots, 0)$

These polynomials satisfy $= 0 \pmod{b^t}$ with substituting $x = r$. You can see $\dim L = t\delta + u$ and $\det L = N^{\delta t(t+1)/2} \cdot X^{(t\delta+u-1)\cdot(t\delta+u)/2}$. We maximize X on u as $\det L^{1/\dim L} < N^{t\beta}$, then $t\delta + u \simeq (t+1)/\beta$ (actually, exact optimized value is a bit smaller) and $\max X \simeq N^{(\beta^2 t)/((t+1)\delta-\beta)}$.

On the other hand, by using LLL, we can obtain small vector v_1 such that $\|v_1\| \le 2^{\frac{\dim L-1}{4}} \cdot \det L^{1/\dim L}$. So we expect we can find a good polynomial by LLL with above lattice if around $X < 1/2N^{\beta^2/\delta}$. For detailed discussion about constant multiplication, see New RSA Vulnerabilities Using Lattice Reduction Methods, Thesis, A. May, 2003. Note that you do not forget to take into account for the factor $\sqrt{\omega}$ $(\omega = \dim L)$ for Howgrave-Graham bound. ■

Above theorem is theoretically clean, but we sometimes need parameter tuning for applying this to each task in practice. You know, especially ϵ is problematic if we use the above method as "magic" black box. I experienced $\beta = 0.5$ did not work. (For severe condition, we should set more sophisticated parameter setting such as $\beta = 0.499$.) Why we need for parameter tuning? This is because it involves asymptotic behavior. On the above proof, I states $\max X \simeq N^{(\beta^2 t)/((t+1)\delta-\beta)}$, if $t \to \infty$, then $\max X \simeq N^{\beta^2/\delta}$. And approximation and inequality discussion is involved for the proof, it goes worse. So we try to avoid parameter tuning.

For practice, we only have to construct lattice with specifically determined shift polynomials. Even if first choice of u, t are wrong, we can improve lattice quality just go up these parameters. When u goes up, $\det L^{1/\dim L}$ is decreasing, so we expect solution will find. Also, t goes up, we improve estimation of $\max X$. And we can check whether found polynomial is good or not with L1 norm Howgrave-Graham condition. These leads the following algorithm.

Implementation: Univariate Case

```
from sage.all import *

import time

from coppersmith_common import RRh, shiftpoly, genmatrix_from_shiftpolys, do_LLL,
filter_LLLresult_coppersmith
from rootfind_ZZ import rootfind_ZZ
from logger import logger
```

```python
### one variable coppersmith
def coppersmith_one_var_core(basepoly, bounds, beta, t, u, delta):
    logger.info("trying param: beta=%f, t=%d, u=%d, delta=%d", beta, t, u, delta)
    basepoly_vars = basepoly.parent().gens()
    basepoly = basepoly / basepoly.monomial_coefficient(basepoly_vars[0])

    shiftpolys = []
    for i in range(u-1, -1, -1):
        # x^i * f(x)^t
        shiftpolys.append(shiftpoly(basepoly, t, 0, [i]))
    for i in range(1, t+1, 1):
        for j in range(delta-1, -1, -1):
            # x^j * f(x)^(t-i) * N^i
            shiftpolys.append(shiftpoly(basepoly, t-i, i, [j]))

    mat = genmatrix_from_shiftpolys(shiftpolys, bounds)
    lll, trans = do_LLL(mat)
    result = filter_LLLresult_coppersmith(basepoly, beta, t, shiftpolys, lll, trans)
    return result


def coppersmith_onevariable(basepoly, bounds, beta, maxmatsize=100, maxu=8):
    if type(bounds) not in [list, tuple]:
        bounds = [bounds]

    N = basepoly.parent().characteristic()

    basepoly_vars = basepoly.parent().gens()
    if len(basepoly_vars) != 1:
        raise ValueError("not one variable poly")
    try:
        delta = basepoly.weighted_degree([1])
    except:
        delta = basepoly.degree()
```

```
    log_N_X = RRh(log(bounds[0], N))
    if log_N_X >= RRh(beta)**2/delta:
        raise ValueError("too much large bound")

    testimate = int(1/(((RRh(beta)**2)/delta)/log_N_X - 1))//2

    logger.debug("testimate: %d", testimate)
    t = min([maxmatsize//delta, max(testimate, 3)])

    whole_st = time.time()

    curfoundpols = []
    while True:
        if t*delta > maxmatsize:
            raise ValueError("maxmatsize exceeded(on coppersmith_one_var)")
        u0 = max([int((t+1)/RRh(beta) - t*delta), 0])
        for u_diff in range(0, maxu+1):
            u = u0 + u_diff
            if t*delta + u > maxmatsize:
                break
            foundpols = coppersmith_one_var_core(basepoly, bounds, beta, t, u, delta)
            if len(foundpols) == 0:
                continue

            curfoundpols += foundpols
            curfoundpols = list(set(curfoundpols))
            sol = rootfind_ZZ(curfoundpols, bounds)
            if sol != [] and sol is not None:
                whole_ed = time.time()
                logger.info("whole elapsed time: %f", whole_ed-whole_st)
                return sol
            elif len(curfoundpols) >= 2:
                whole_ed = time.time()
                logger.warning(f"failed. maybe, wrong pol was passed.")
                logger.info("whole elapsed time: %f", whole_ed-whole_st)
                return []
```

```
        t += 1
    # never reached here
    return None
```

The code imports the following functions. For details, see appendix.

- shiftpoly(basepoly, baseidx, Nidx, varsidx_lst): generate shift polynomials as $\mathrm{basepoly}^{\mathrm{baseidx}} \cdot N^{\mathrm{Nidx}} \cdot {x_1}^{j_1} \cdots {x_n}^{j_n}$, whose j_i is varsidx_lst
- genmatrix_from_shiftpolys(shiftpolys, bounds): generate matrix corresponding to shiftpolys
- do_LLL(mat): output LLL result and transformation matrix from mat to LLL result
- filter_LLLresult_coppersmith(basepoly, beta, t, shiftpolys, lll, trans): output short polynomial which satisfies $\mathrm{output}(r) = 0$ over integer for solution r of $\mathrm{basepoly}(r) = 0 \pmod b$. This function only output polynomials with L1 norm $< b^t$
- rootfind_ZZ(pollst, bounds): find solution of pollst over integer on specific bounds

Above algorithm, we need to input basepoly, bounds, β. Choosing β is needed for checking L1 norm $< b^t$ (Since b is unknown, it uses $N^{\beta t}$ instead). I suggests the following β choosing for confirming $N^\beta \le b$:

- If $b = N$, then choose $\beta = 1.0$
- If $b = p$ such as $p \mid N$, then choose $(\mathrm{bitsize}(p) - 1)/(\mathrm{bitsize}(N))$

I used to use $\beta = 0.5, 0.499, \ldots$ for $N = pq$ with same bitsize of p, q in Sagemath small_roots input. In fact, for 2048bit N, $(\mathrm{bitsize}(p) - 1)/(\mathrm{bitsize}(N)) = 0.4995$. Note that zncoppersmith function on Pari/GP expects input as P (basepoly over integer), N, X (bounds), $B = N^\beta$.

Herrmann-May Method (Multivariate Linear Version)

An simple extension of the univariate case, we collect many shift polynomials and apply LLL. But it is just heuristic, then we may find solution or may not. This forces us to tune parameters without knowing goal, and we may fail to a rabbit hole. Sometimes it turns out that this heuristic does not work and another heuristic works. I do not want to search "lucky" anymore.

Instead, we see well analyzed method for multivariate linear polynomial case.

The paper Solving Linear Equations Modulo Divisors: On Factoring Given Any Bits, M. Herrmann and A. May, 2008 states the following:

Theorem [Herrmann and May]

Let N is a (large) positive integer, which has a divisor $b \geq N^{\beta}, 0 < \beta \leq 1$. Let $f(x_1, \ldots, x_n)$ be a linear multivariate polynomial, where the coefficient of x_1 for f is invertible over $\pmod{N}$. And let $0 < X_1, \ldots, X_n$ for an expected bound for a root of $f(x_1, \ldots, x_n)$. Then, we can find a solution r of the equation

$$f(r) = 0 \pmod{b} \; (|r_i| < X_i)$$

, if around $\log_N (X_1 \ldots X_n) \leq 1 - (1-\beta)^{\frac{n+1}{n}} - (n+1) \cdot (1 - (1-\beta)^{\frac{1}{n}}) \cdot (1-\beta)$.

Proof

The coefficient of x_i for $f(x_1, \ldots, x_n)$ is invertible, we can assume the coefficient of x_i for $f(x_1, \ldots, x_n)$ is 1. Write $f(x_1, \ldots, x_n) = x_1 + f_{12}x_2 + \cdots + f_{1n}x_n + f_{00}$. Let t, m are some non-negative integers (tuned later). Then, consider shift polynomials $g_{(i_2, \ldots, i_n, k)} = {x_2}^{i_2} \cdots {x_n}^{i_n} \cdot f^k \cdot N^{\max\{t-k,\, 0\}}$ with $\sum_{j=2}^{n} i_j \leq m - k$. Then, we can construct the following lattice L.

- each (column) element are corresponding to: ${X_1}^m, X_2 \cdot {X_1}^{m-1}, \ldots, X_n \cdot {X_1}^{m-1}, {X_2}^2 \cdot {X_1}^{m-2}, X_2 \cdot X_3 \cdot {X_1}^{m-2} \ldots, {X_n}^2 \cdot {X_1}^{m-2}, \ldots, {X_1}^{m-2}, \ldots, 1$
- each row are corresponding to: $g_{(0,0,\ldots,0,m)}, g_{(1,0,\ldots,0,m-1)}, \ldots, g_{(0,\ldots,0,m-1)}, g_{(2,0,\ldots,0,m-2)}, \ldots, g_{(0,\ldots,0,m-2)}, \ldots, g_{(0,\ldots,0,0)}$

Those vectors have triangular form. $\dim L = \binom{(n+1)+m-1}{m}$ ($(n+1)$ multichoose m). $\det L = \left(\prod_{i=1}^{n} {X_i}^{s_{x_i}}\right) \cdot N^{s_N}$, where $s_{x_i} = \sum_{\ell=0}^{m} \ell \cdot \binom{n+(m-\ell)-1}{m-\ell} = \binom{m+n}{m-1}$ and $s_N = \sum_{\ell=0}^{t} \ell \cdot \binom{n+(m-t+\ell)-1}{m-t+\ell} = t \cdot \binom{m+n}{n} - \binom{m+n}{m-1} + \binom{m-t+n}{m-t-1}$ [1].

We want to maximize $X_1 \dots X_n$ on m as $\det L^{1/(\dim L - n+1)} < N^{t\beta}$ for obtaining n good polynomials. By the analysis from the author, $\tau = 1-(1-\beta)^{\frac{1}{n}}$ $(t = \tau m)$ gives some optimal value. Then,

$$\max \log_N (X_1 \dots X_n) \simeq 1-(1-\beta)^{\frac{n+1}{n}} - (n+1)\cdot\left(1-(1-\beta)^{\frac{1}{n}}\right)\cdot(1-\beta) - \frac{n^{\frac{1}{\pi}}(1-\beta)^{-0.278465}}{m} + \beta \ln(1-\beta)\frac{n}{m}$$

Like univariate case, we expect we can find good n- polynomials by LLL with above lattice if around $\log_N (X_1 \dots X_n) \le 1-(1-\beta)^{\frac{n+1}{n}} - (n+1)\cdot\left(1-(1-\beta)^{\frac{1}{n}}\right)\cdot(1-\beta)$. For detailed, see the original paper. ■

Then, we can implement multivariate linear case straightforward. Note that, on the above proof, it forces the coefficient of x_1 to 1, but we can choose other term $x_2, \dots, x_n$. So we use all "monic-ed" polynomials on x_i. The proposion gurantees this can be improved quality. I do not think this tweak improves the quality so much, but I add it for retaining symmetry. Note that this addition does not increase much complexity for LLL cause linear dependent vectors are transformed to zero vectors.

Proposition

Let $v_1, \dots, v_\omega$ as a ω-dimensional basis for a lattice L, that is, L is full lattice. Let $w_1, \dots, w_{\omega'}$ as ω-dimensional vectors and L' is a lattice generated by $v_1, \dots, v_\omega, w_1, \dots, w_{\omega'}$.
Then, $\dim L' = \dim L = \omega$ and $\det L' \le \det L$.

Proof

L is full lattice and L' includes L, so $\omega \ge \dim L' \ge \dim L = \omega$. Let B_L as the basis matrix of L, and $B_{L'}$ as the basis matrix of L'. L' includes L, then we have some integer matrix A such that $B_L = A \cdot B_{L'}$. So $\det L = |\det B_L| = |\det A| \cdot |\det B_{L'}| = |\det A| \cdot \det L' \ge \det L'$. ■

It is important that assuming L is full lattice. If L is not full lattice, adding some vectors to L may increase the determinant. (As an example, see the lattice $(1,0),(0,2)$ as L' and L as $(1,0)$.) In our case, involving monomial set does not change (so dimension is same) and the lattice related to x_1 is full lattice.

Implementation: Multivariate Linear Case

```
from sage.all import *

import time
import itertools

from coppersmith_common import RRh, shiftpoly, genmatrix_from_shiftpolys, do_LLL,
filter_LLLresult_coppersmith
from rootfind_ZZ import rootfind_ZZ
from logger import logger


### multivariate linear coppersmith (herrmann-may)
def coppersmith_linear_core(basepoly, bounds, beta, t, m):
    logger.info("trying param: beta=%f, t=%d, m=%d", beta, t, m)
    basepoly_vars = basepoly.parent().gens()
    n = len(basepoly_vars)

    shiftpolys = []
    for i, basepoly_var in enumerate(basepoly_vars):
        basepoly_i = basepoly / basepoly.monomial_coefficient(basepoly_var)

        for k in range(m+1):
            for j in range(m-k+1):
                for xi_idx_sub in itertools.combinations_with_replacement(range(n-1), j):
                    xi_idx = [xi_idx_sub.count(l) for l in range(n-1)]
                    assert sum(xi_idx) == j
                    xi_idx.insert(i, 0)
                    # x2^i2 * ... * xn^in * f^k * N^max(t-k,0)
```

```
                shiftpolys.append(shiftpoly(basepoly_i, k, max(t-k, 0), xi_idx))

    mat = genmatrix_from_shiftpolys(shiftpolys, bounds)
    lll, trans = do_LLL(mat)
    result = filter_LLLresult_coppersmith(basepoly, beta, t, shiftpolys, lll, trans)
    return result


def coppersmith_linear(basepoly, bounds, beta, maxmatsize=100, maxm=8):
    if type(bounds) not in [list, tuple]:
        raise ValueError("not linear polynomial (on coppersmith_linear)")

    N = basepoly.parent().characteristic()

    basepoly_vars = basepoly.parent().gens()
    n = len(basepoly_vars)
    if n == 1:
        raise ValueError("one variable poly")

    if not set(basepoly.monomials()).issubset(set(list(basepoly_vars)+[1])):
        raise ValueError("non linear poly")

    log_N_X = RRh(log(product(bounds), N))
    log_N_X_bound = 1-(1-RRh(beta))**(RRh(n+1)/n) - (n+1)*(1-(1-RRh(beta))**(RRh(1)/n)) * (1-
RRh(beta))

    if log_N_X >= log_N_X_bound:
        raise ValueError("too much large bound")

    mestimate = (n*(-RRh(beta)*ln(1-beta) + ((1-RRh(beta))**(-0.278465))/pi)/(log_N_X_bound -
log_N_X))/(n+1.5)
    tau = 1 - (1-RRh(beta))**(RRh(1)/n)
    testimate = int(mestimate * tau + 0.5)

    logger.debug("testimate: %d", testimate)
    t = max(testimate, 1)
```

```python
    while True:
        if t == 1:
            break
        m = int(t/tau+0.5)
        if binomial(n+1+m-1, m) <= maxmatsize:
            break
        t -= 1

    whole_st = time.time()

    curfoundpols = []
    while True:
        m0 = int(t/tau+0.5)
        if binomial(n+1+m0-1, m0) > maxmatsize:
            raise ValueError("maxmatsize exceeded(on coppersmith_linear)")
        for m_diff in range(0, maxm+1):
            m = m0 + m_diff
            if binomial(n+1+m-1, m) > maxmatsize:
                break
            foundpols = coppersmith_linear_core(basepoly, bounds, beta, t, m)
            if len(foundpols) == 0:
                continue

            curfoundpols += foundpols
            curfoundpols = list(set(curfoundpols))
            sol = rootfind_ZZ(curfoundpols, bounds)
            if sol != [] and sol is not None:
                whole_ed = time.time()
                logger.info("whole elapsed time: %f", whole_ed-whole_st)
                return sol
            elif len(curfoundpols) >= 2 * n + 1:
                whole_ed = time.time()
                logger.warning(f"failed. maybe, wrong pol was passed.")
                logger.info("whole elapsed time: %f", whole_ed-whole_st)
                return []
```

```
        t += 1
    # never reached here
    return None
```

More General Case

A strategy for finding a root for modulus equation (and integer equation) on general case is proposed on A Strategy for Finding Roots of Multivariate Polynomials with New Applications in Attacking RSA Variants, E. Jochemez and A. May, 2006. But this method does not assure we could obtain solution. I think general multivarite polynomial root finding is complicated. You might consider a polynomial $f(x, y, z) = axy + byz + cx^2 + d$, which involves some cross terms xy, yz. We want to reduce these terms in case XY, YZ might be large, but it blows up the number of related monomials. Also we do not know which monomial ordering should be chose. (Which one should we choose for diagonal elements: yz or x^2 ?)

For obtaining good result of partial key exposure attacks, many authors propose different lattice construction. Some results are refined on A Tool Kit for Partial Key Exposure Attacks on RSA, †. Takayasu and N. Kunihiro, 2016. I do not think it is realistic to construct good lattice in our own during short period such as CTF. So we would like to search papers instead of tuning parameters. It might be worth to try using pre-checked good heuristic implementions, but devoting only one implementation is bad. If you determine to use heuristics, trying various methods may lead to win.

Or you can apply Herrmann-May with linearization. If XY, YZ, XZ are reasonably small, we can set new variables $U = XY, V = YZ, W = XZ$. Even if this construction might not be optimal, it could be better to try to complicated parameter tuning.

On the other hand, I states just simple extention of univariate case/multivariate linear case: very restrictive bivariate case

Proposition

Let N is a (large) positive integer, which has a divisor $b \geq N^{\beta}, 0 < \beta \leq 1$. Let a polynomial $f(x, y) = f_{x1}x + f_{y\delta}y^{\delta} + \ldots + f_{y1}y + f_{00}$ be a special form bivariate polynomial, where the coefficient of x for f is invertible over $\pmod N$. And let $0 < X, Y$ for an expected bound for a root of $f(x, y)$. Then, we can find a solution r of the equation

$$f(r) = 0 \pmod b \; (|r_1| < X, |r_2| < Y)$$

, if around $\log_N X, \log_N Y \leq \frac{(3\beta-2)+(1-\beta)^{\frac{3}{2}}}{1+\delta}$.

Proof

The coefficient of x for $f(x, y)$ is invertible, we can assume the coefficient of x for $f(x, y)$ is 1.
Rewrite $f(x, y) = x + f_{y\delta}y^{\delta} + \ldots + f_{y1}y + f_{00}$. Let t, m are some non-negative integers (tuned later). Then, consider shift polynomials $g_{(i,k)} = y^i f^k N^{\max\{t-k,\, 0\}}$ with $i \leq \delta \cdot (m - k)$. Then, we can construct the following lattice L.

- each (column) element are corresponding to: $X^m, Y^{\delta}X^{m-1}, \ldots, X^{m-1}, Y^{2\delta} * X^{m-2}, \ldots, X^{m-2}, \ldots, 1$
- each row are corresponding to: $g_{(0,m)}, g_{(\delta,m-1)}, \ldots, g_{(0,m-1)}, g_{(2\delta,m-2)}, \ldots, g_{(0,m-2)}, \ldots, g_{(0,0)}$

Those vectors have triangular form. $\dim L = (m+1) \cdot (2 + \delta m)/2$. $\det(L) = X^{s_X} \cdot Y^{s_Y} \cdot N^{s_N}$, where $s_X = m \cdot (m+1) \cdot (\delta \cdot (m-1) + 3)/6, s_Y = \delta \cdot m \cdot (m+1) \cdot (\delta \cdot (2m+1) + 3)/12, s_N = t \cdot (t+1) \cdot (\delta \cdot (3m - t + 1) + 3)/6$
.

We want to maximize X, Y on m as $\det L^{1/(\dim L - 2 + 1)} < N^{t\beta}$ for obtaining 2 good polynomials. In this proof, we assume $X \simeq Y$. By rough calculus ($1/m \simeq 0$), $\tau = 1 - \sqrt{1-\beta}$ ($t = \tau m$) gives some optimal value. Then, $\max \log_N X, \max \log_N Y \simeq \frac{(3\beta-2)+(1-\beta)^{\frac{3}{2}}}{(1+\delta)+\frac{\delta}{2*m}}$. Like other case, we expect we can find good 2- polynomials if above condition satisfied. ■

Note that $\delta = 1$ in above case is just $n = 2$ for linear case. Constructed lattice and $\tau = 1 - (1-\beta)^{\frac{1}{2}}$ are exactly same, and the bounds of X, Y matches for large m.

This type of lattices can be constructed for another polynomials. For example, it can be applied to $f(x,y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$. In fact, the monomials f^m are $\{x^i y^j \mid i+j \le 2m\}$. Even if this structure might not always be applicable to other polynomials, we might expect some heuristic works. We may start $t = 1, 2, \ldots$ and $m = t/\tau$ as large multiple values of t respect to β. (For small β, m should be large value.)

Back to chronophobia

Then, I restate what we want to solve. Let $N = pq$ be a 1024 bit integer. We have a oracle named broken_token, which leaks $L = 200$ digits (about 664bits).
For the sake of this oracle, we know L-digits $u1, u2$, and we need to solve the following equation:

$$(u1 \cdot (10^{\mathrm{Ludown}}) + y)^2 - (u2 \cdot (10^{\mathrm{Lu2down}}) + x) = 0 \pmod N$$

, where x, y are small ($\le 10^{\mathrm{Ludown}}, 10^{\mathrm{Lu2down}}$). ($\mathrm{Ludown}, \mathrm{Lu2down} \le 108$ digits or about 359 bits)

As I just stated the proposition on general case section, we may solve this type of equation if $\log_2 Y, \log_2 X < 1024/3 = 341$ bits for $\beta = 1.0$. I just say it solve ALMOST, but not.

For extending the result, we consider the following equation (a, b are known):

$$f(x,y) := -x + y^2 + ay + b = 0 \pmod N$$

This type of equation was analyzed at Attacking Power Generators Using Unravelled Linearization: When Do We Output Too Much?, Herrman and May, 2009. Let $u := y^2 - x$ for linearization. Then,

$$f^2 = (u + ay + b)^2 = u^2 + a^2y^2 + b^2 + 2ayu + 2bu + 2aby = u^2 + 2ayu + (a^2 + 2b)u + a^2x + 2aby + b^2$$

Also, $yf = yu + au + ax + by$.

Then, we construct the lattice L with monomials $U^2, YU, U, X, Y, 1$. These shift polynomials are $f^2, yfN, fN, xN^2, yN^2, N^2$:

$$\begin{pmatrix} U^2 & 2aYU & (a^2+2b)U & a^2X & 2abY & b^2 \\ 0 & NYU & aNU & aNX & bNY & 0 \\ 0 & 0 & NU & 0 & aNY & bN \\ 0 & 0 & 0 & N^2X & 0 & 0 \\ 0 & 0 & 0 & 0 & N^2Y & 0 \\ 0 & 0 & 0 & 0 & 0 & N^2 \end{pmatrix}$$

$\dim L = 6$ and $\det L = U^4XY^2N^8$. Then, assuming $X \simeq Y, U \simeq X^2$, for about $X < N^{\frac{4}{11}}$, we can find good polynomial.

In our case, $1024 \cdot (4/11) = 372$, so we can solve the above problem by Coppersmith method.

I do not know above discussion assures we can construct $h(x, y)$ with shift polynomials of $f(x, y)$ (without linearlization) cause I do not construct a lattice directly. (lattice is complicated!) But outputs of lbc_toolkit may be reasonable. For solving chronophobia, we need the following shift polynomials (These are generated on the paremter $m = 2, d = 2.$):

$$f(x,y)^2, yf(x,y)N, xf(x,y) * N, xyN^2, x^2N^2, f(x,y)N, xN^2, yN^2, N^2$$

This shift polynomials have triangular form (full lattice). And the lattice can generate good polynomial related to the lattice L (with linearization).

Then, we relook defund coppersmith. It turns out that the parameter $m = 2, d = 3$ works! This is cause shift polynomials are chosen as the following. (The parameter m is corresponding to our parameter $t = 2$. For obtaining x^2N^2, we should set $d = 2 + 1$.)

```
    for i in range(m+1):
        base = N^(m-i) * f^i
        for shifts in itertools.product(range(d), repeat=f.nvariables()):
```

```
            g = base * prod(map(power, f.variables(), shifts))
            G.append(g)
```

Though defund coppersmith can generate arbitrary shift polynomials, it may generate many useless shift polynomials for an input multivariate polynomial, so sometimes we could not compute LLL for large lattice in practice. lbc_toolkit outputs fairly reasonable shift polynomials, it includes a few useless shift polynomials, though.

How to Solve Future CTF Challenges?

With above discussion, I suggest the following basic strategy.

1. Construct input polynomial as **no cross term**. Or applying linearization if cross term bounds are small. (If not, search papers or change polynomial.)
2. try univariate case or linear case. parameters are chose based on above discussion, and go up parameters slightly (first m and then t)
3. try heuristic (lbc_toolkit) with going up paremters (first d and then m), in parallel, try defund one

Also, I suggest to print debug messages on each stage. If LLL takes much times, we know these parameters are too much. If passes LLL and not output solution, then we may check Howgrave-Graham bounds are satisfied (especially, β should be fairly restricted). If stucks on computing roots over integer, you might research how to find integer solution. Finding integer solution for multivariate polynomials are not easy in general. (general solver for diophantine equation does not exist.)

Conclusion

Lattice is so wild. The detailed discussions will help us for solving many tasks. We are waiting more discussion for specific examples...

Appendix: rootfind_ZZ

Finding roots over integer is not easy. For one variable polynomial, you only have to use Sagemath roots method. For multivarite polynomials, we do not know the efficient and exact method for root finiding task. So I implement three method:

1. solve_root_jacobian_newton:

- numerical method (cannot find all roots, but efficient)
- iteration method (Newton method: compute gradient (jacobian) and update point to close a root)
- possibly, no root found (converge local minima or divergence)

2. solve_root_hensel

- algebraic method (find all roots, slow)
- find root mod small p and update to mod large modulus
- possibly, cannot compute a root (too many candidates on modulus even if only a few roots over integer)

3. solve_root_triangulate

- algebraic method (try to find all roots, slow)
- compute Groebner basis and then find solution by solve function
- For finding all roots, sometimes requires manual manipulation (no general method)

```
from sage.all import *

from random import shuffle as random_shuffle
from itertools import product as itertools_product
import time

from logger import logger
```

```
def solve_root_onevariable(pollst, bounds):
    logger.info("start solve_root_onevariable")
    st = time.time()

    for f in pollst:
        f_x = f.parent().gens()[0]
        try:
            rt_ = f.change_ring(ZZ).roots()
            rt = [ele for ele, exp in rt_]
        except:
            f_QQ = f.change_ring(QQ)
            f_QQ_x = f_QQ.parent().gens()[0]
            rt_ = f_QQ.parent().ideal([f_QQ]).variety()
            rt = [ele[f_QQ_x] for ele in rt_]
        if rt != []:
            break
    result = []
    for rtele in rt:
        if any([pollst[i].subs({f_x: int(rtele)}) != 0 for i in range(len(pollst))]):
            continue
        if abs(int(rtele)) < bounds[0]:
            result.append(rtele)

    ed = time.time()
    logger.info("end solve_root_onevariable. elapsed %f", ed-st)

    return result


def solve_root_groebner(pollst, bounds):
    logger.info("start solve_root_groebner")
    st = time.time()

    # I heard degrevlex is faster computation for groebner basis, but idk real effect
    polrng_QQ = pollst[0].change_ring(QQ).parent().change_ring(order='degrevlex')
```

```
    vars_QQ = polrng_QQ.gens()
    G = Sequence(pollst, polrng_QQ).groebner_basis()
    try:
        # not zero-dimensional ideal raises error
        rt_ = G.ideal().variety()
    except:
        logger.warning("variety failed. not zero-dimensional ideal?")
        return None
    rt = [[int(ele[v]) for v in vars_QQ] for ele in rt_]

    vars_ZZ = pollst[0].parent().gens()
    result = []
    for rtele in rt:
        if any([pollst[i].subs({v: int(rtele[i]) for i, v in enumerate(vars_ZZ)}) != 0 for i in
range(len(pollst))]):
            continue
        if all([abs(int(rtele[i])) < bounds[i] for i in range(len(rtele))]):
            result.append(rtele)

    ed = time.time()
    logger.info("end solve_root_groebner. elapsed %f", ed-st)
    return result


def solve_ZZ_symbolic_linear_internal(sol_coefs, bounds):
    mult = prod(bounds)
    matele = []
    for i, sol_coef in enumerate(sol_coefs):
        denom = 1
        for sol_coef_ele in sol_coef:
            denom = LCM(denom, sol_coef_ele.denominator())
        for sol_coef_ele in sol_coef:
            matele.append(ZZ(sol_coef_ele * denom * mult))
        matele += [0]*i + [-mult*denom] + [0] * (len(bounds)-i-1)
    for idx, bd in enumerate(bounds):
        matele += [0]*len(sol_coefs[0]) + [0] * idx + [mult//bd] + [0]*(len(bounds)-idx-1)
```

```
    # const term
    matele += [0]*(len(sol_coefs[0])-1) + [mult] + [0]*len(bounds)
    mat = matrix(ZZ, len(sol_coefs)+len(bounds)+1, len(sol_coefs[0])+len(bounds), matele)
    logger.debug(f"start LLL for solve_ZZ_symbolic_linear_internal")
    mattrans = mat.transpose()
    lll, trans = mattrans.LLL(transformation=True)
    logger.debug(f"end LLL")
    for i in range(trans.nrows()):
        if all([lll[i, j] == 0 for j in range(len(sol_coefs))]):
            if int(trans[i,len(sol_coefs[0])-1]) in [1,-1]:
                linsolcoef = [int(trans[i,j])*int(trans[i,len(sol_coefs[0])-1]) for j in range(len(sol_coefs[0]))]
                logger.debug(f"linsolcoef found: {linsolcoef}")
                linsol = []
                for sol_coef in sol_coefs:
                    linsol.append(sum([ele*linsolcoef[idx] for idx, ele in enumerate(sol_coef)]))
                return [linsol]
    return []


def solve_root_triangulate(pollst, bounds):
    logger.info("start solve_root_triangulate")
    st = time.time()

    polrng_QQ = pollst[0].change_ring(QQ).parent().change_ring(order='lex')
    vars_QQ = polrng_QQ.gens()
    G = Sequence(pollst, polrng_QQ).groebner_basis()
    if len(G) == 0:
        return []

    symbolic_vars = [var(G_var) for G_var in G[0].parent().gens()]
    try:
        sols = solve([G_ele(*symbolic_vars) for G_ele in G], symbolic_vars, solution_dict=True)
    except:
        return None
```

```
    logger.debug(f"found sol on triangulate: {sols}")

    result = []
    # solve method returns parametrized solution. We treat only linear equation
    # TODO: use solver for more general integer equations (such as diophautus solver, integer
programming solver, etc.)
    for sol in sols:
        sol_args = set()
        for symbolic_var in symbolic_vars:
            sol_var = sol[symbolic_var]
            sol_args = sol_args.union(set(sol_var.args()))

        sol_args = list(sol_args)
        sol_coefs = []
        for symbolic_var in symbolic_vars:
            sol_var = sol[symbolic_var]
            sol_coefs_ele = []
            for sol_arg in sol_args:
                if sol_var.is_polynomial(sol_arg) == False:
                    logger.warning("cannot deal with non-polynomial equation")
                    return None
                if sol_var.degree(sol_arg) > 1:
                    logger.warning("cannot deal with high degree equation")
                    return None
                sol_var_coef_arg = sol_var.coefficient(sol_arg)
                if sol_var_coef_arg not in QQ:
                    logger.warning("cannot deal with multivariate non-linear equation")
                    return None
                sol_coefs_ele.append(QQ(sol_var_coef_arg))
            # constant term
            const = sol_var.subs({sol_arg: 0 for sol_arg in sol_args})
            if const not in QQ:
                return None
            sol_coefs_ele.append(const)
```

```
            sol_coefs.append(sol_coefs_ele)
        ZZsol = solve_ZZ_symbolic_linear_internal(sol_coefs, bounds)
        result += ZZsol

    ed = time.time()
    logger.info("end solve_root_triangulate. elapsed %f", ed-st)
    return result


def solve_root_jacobian_newton_internal(pollst, startpnt):
    # NOTE: Newton method's complexity is larger than BFGS, but for small variables Newton
method converges soon.
    pollst_Q = Sequence(pollst, pollst[0].parent().change_ring(QQ))
    vars_pol = pollst_Q[0].parent().gens()
    jac = jacobian(pollst_Q, vars_pol)

    if all([ele == 0 for ele in startpnt]):
        # just for prepnt != pnt
        prepnt = {vars_pol[i]: 1 for i in range(len(vars_pol))}
    else:
        prepnt = {vars_pol[i]: 0 for i in range(len(vars_pol))}
    pnt = {vars_pol[i]: startpnt[i] for i in range(len(vars_pol))}

    maxiternum = 1024
    iternum = 0
    while True:
        if iternum >= maxiternum:
            logger.warning("failed. maybe, going wrong way.")
            return None

        evalpollst = [(pollst_Q[i].subs(pnt)) for i in range(len(pollst_Q))]
        if all([int(ele) == 0 for ele in evalpollst]):
            break
        jac_eval = jac.subs(pnt)
        evalpolvec = vector(QQ, len(evalpollst), evalpollst)
        try:
```

```
            pnt_diff_vec = jac_eval.solve_right(evalpolvec)
        except:
            logger.warning("pnt_diff comp failed.")
            return None

        prepnt = {key:value for key,value in prepnt.items()}
        pnt = {vars_pol[i]: round(QQ(pnt[vars_pol[i]] - pnt_diff_vec[i])) for i in
range(len(pollst_Q))}

        if all([prepnt[vars_pol[i]] == pnt[vars_pol[i]] for i in range(len(vars_pol))]):
            logger.warning("point update failed. (converged local sol)")
            return None
        prepnt = {key:value for key,value in pnt.items()}
        iternum += 1
    return [int(pnt[vars_pol[i]]) for i in range(len(vars_pol))]


def solve_root_jacobian_newton(pollst, bounds):
    logger.info("start solve_root_jacobian newton")
    st = time.time()

    pollst_local = pollst[:]
    vars_pol = pollst[0].parent().gens()

    # not applicable to non-determined system
    if len(vars_pol) > len(pollst):
        return []

    for _ in range(10):
        random_shuffle(pollst_local)
        for signs in itertools_product([1, -1], repeat=len(vars_pol)):
            startpnt = [signs[i] * bounds[i] for i in range(len(vars_pol))]
            result = solve_root_jacobian_newton_internal(pollst_local[:len(vars_pol)],
startpnt)
            # filter too much small solution
            if result is not None:
```

```
                if all([abs(ele) < 2**16 for ele in result]):
                    continue
                ed = time.time()
                logger.info("end solve_root_jacobian newton. elapsed %f", ed-st)
                return [result]


def _solve_root_GF_smallp(pollst, smallp):
    Fsmallp = GF(smallp)
    polrng_Fsmallp = pollst[0].change_ring(Fsmallp).parent().change_ring(order='degrevlex')
    vars_Fsmallp = polrng_Fsmallp.gens()
    fieldpolys = [varele**smallp - varele for varele in vars_Fsmallp]
    pollst_Fsmallp = [polrng_Fsmallp(ele) for ele in pollst]
    G = pollst_Fsmallp[0].parent().ideal(pollst_Fsmallp + fieldpolys).groebner_basis()
    rt_ = G.ideal().variety()
    rt = [[int(ele[v].lift()) for v in vars_Fsmallp] for ele in rt_]
    return rt


def solve_root_hensel_smallp(pollst, bounds, smallp):
    logger.info("start solve_root_hensel")
    st = time.time()

    vars_ZZ = pollst[0].parent().gens()
    smallp_exp_max = max([int(log(ele, smallp)+0.5) for ele in bounds]) + 1
    # firstly, compute low order
    rt_lows = _solve_root_GF_smallp(pollst, smallp)
    for smallp_exp in range(1, smallp_exp_max+1, 1):
        cur_rt_low = []
        for rt_low in rt_lows:
            evalpnt = {vars_ZZ[i]:(smallp**smallp_exp)*vars_ZZ[i]+rt_low[i] for i in
range(len(vars_ZZ))}
            nextpollst = [pol.subs(evalpnt)/(smallp**smallp_exp) for pol in pollst]
            rt_up = _solve_root_GF_smallp(nextpollst, smallp)
            cur_rt_low += [tuple([smallp**smallp_exp*rt_upele[i] + rt_low[i] for i in
range(len(rt_low))]) for rt_upele in rt_up]
```

```
        rt_lows = list(set(cur_rt_low))
        if len(rt_lows) >= 800:
            logger.warning("too much root candidates found")
            return None

    result = []
    for rt in rt_lows:
        rtele = [[ele, ele - smallp**(smallp_exp_max+1)][ele >= smallp**smallp_exp_max] for ele
in rt]
        if any([pollst[i].subs({v: int(rtele[i]) for i, v in enumerate(vars_ZZ)}) != 0 for i in
range(len(pollst))]):
            continue
        if all([abs(int(rtele[i])) < bounds[i] for i in range(len(rtele))]):
            result.append(rtele)

    ed = time.time()
    logger.info("end solve_root_hensel. elapsed %f", ed-st)
    return result


def solve_root_hensel(pollst, bounds):
    for smallp in [2, 3, 5]:
        result = solve_root_hensel_smallp(pollst, bounds, smallp)
        if result != [] and result is not None:
            return result
    return None


## wrapper function
def rootfind_ZZ(pollst, bounds):
    vars_pol = pollst[0].parent().gens()
    if len(vars_pol) != len(bounds):
        raise ValueError("vars len is invalid (on rootfind_ZZ)")

    # Note: match-case statement introduced on python3.10, but not used for backward compati
    if len(vars_pol) == 1:
```

```
        return solve_root_onevariable(pollst, bounds)
    else:
        # first numeric
        result = solve_root_jacobian_newton(pollst, bounds)
        if result != [] and result is not None:
            return result

        # next hensel (fast if the number of solutions mod smallp**a are small. in not case,
cannot find solution)
        result = solve_root_hensel(pollst, bounds)
        if result != [] and result is not None:
            return result

        # last triangulate with groebner (slow, but sometimes solve when above methods does not
work)
        #return solve_root_groebner(pollst, bounds)
        return solve_root_triangulate(pollst, bounds)
```

---

1. These equalities can be proven by calculation like sum of multichoose multiplied by its argument. I used Explicit form for sum of "multichoose" functions. (involving Hockey-stick identity). ↩