

Winter notes for CTF



Week 4

Astrageldon

2024-02-05

HAPPY SPRING FESTIVAL



and fuck the school :(

Contents

1	Crypto	3
1.1	A Toy 🧸 Implementation of Masaaki Shirase's Factorization Method (<i>Part II</i>)	3
1.1.1	A Slight Improvement by Astrageldon	3
1.1.2	The True and Thorough Method for the $4p - 1$ Factorization Problem	6

1 Crypto

1.1 A Toy Implementation of Masaaki Shirase's Factorization Method (Part II)

上回说到，计算出 division polynomial $\psi_n = (a_0 + a_1j) + (a_2 + a_3j)X$ 以后，构造如下的映射

$$\phi_n : \mathcal{S}_n^{D,\tau} \rightarrow \mathbb{Z}_n, \quad (a_0 + a_1j) + (a_2 + a_3j)X \mapsto c = b_0^2 + b_1^2s - b_0b_1t$$

$$\text{where } b_0 + b_1j = (a_0 + a_1j)^2 - (a_2 + a_3j)^2\tau$$

便可以得到 $\phi_n((a_0 + a_1j) + (a_2 + a_3j)X) \equiv 0 \pmod{p}$ ，这是因为根据韦达定理：

$$t = -(j_0 + j_1)$$

$$s = j_0j_1$$

其中 j_0, j_1 是方程 $(b_0 + b_1j)^2 = 0$ 的两根。换句话说，上述映射 ϕ_n 实际上将 ψ_n 映射到

$$c = (b_0 + b_1j_0)(b_0 + b_1j_1)$$

基于此，我们可以进行简单的扩展，将 $D = 2$ 的情况扩展到 D 更大的情况。

1.1.1 A Slight Improvement by Astrageldon

设 $\deg(H_D(j)) = d$, $m = d - 1$ ，则 $\mathcal{R}_n^D = \mathbb{Z}_n[j]/(H_D(j))$ 中的元素具有如下的形式：

$$\alpha_0 + \alpha_1j + \cdots + \alpha_mj^m \in \mathcal{R}_n^D$$

还是按照白势政明等人的方法，进一步构造 $\mathcal{S}_n^{D,\tau} = \mathcal{R}_n^D[X]/(X^2 - \tau)$ ，则 $\mathcal{S}_n^{D,\tau}$ 中的元素具有如下形式：


$$(a_0 + a_1j + \cdots + a_mj^m) + (\tilde{a}_0 + \tilde{a}_1j + \cdots + \tilde{a}_mj^m)X \in \mathcal{S}_n^{D,\tau}$$

乘以关于 X 的共轭便可以将 X 消去，得到完全关于 j 的表达式，对每个 j 的单项式取其系数得到

$$b_0 + b_1j + \cdots + b_mj^m$$

为了和 $H_D(j) = s_0 + s_1j + \cdots + s_dj^d$, $s_d = 1$ 产生联系，我们设 $H_D(j) = 0$ 在扩域 $\mathcal{S}_n^{D,\tau}$ 上具有 d 个根 j_0, \dots, j_m ，构造对称多项式：

$$\Phi_d(j_0, \dots, j_m) = (b_0 + b_1j_0 + \cdots + b_mj_0^m)(b_0 + b_1j_1 + \cdots + b_mj_1^m) \cdots (b_0 + b_1j_m + \cdots + b_mj_m^m)$$

根据对称多项式基本定理 (fundamental theorem of symmetric polynomials, )，任何对称多项式都可以被改写为关于基本多项式 e_1, \dots, e_d (elementary symmetric polynomials) 的多项式，这个改写的过程被称作“对称约化” (symmetry reduction)，Mathematica 与 SymPy 中都有相关的实现。比如，对于 $d = 3$ 的情况。

$$\Phi_3(j_0, j_1, j_2) = (b_0 + b_1j_0 + b_2j_0^2)(b_0 + b_1j_1 + b_2j_1^2)(b_0 + b_1j_2 + b_2j_2^2)$$

它可以被改写为

$$\begin{aligned}\tilde{\Phi}_3(e_1, e_2, e_3) = & b_0^3 + b_0^2 b_1 e_1 + b_0^2 b_2 e_1^2 + b_0 b_1 b_2 e_1 e_2 + \\ & b_0 b_2^2 e_2^2 + b_1 b_2^2 e_2 e_3 + b_2^3 e_3^2 + \\ & (-2b_0 b_2^2 + b_1^2 b_2) e_1 e_3 + \\ & (-2b_0^2 b_2 + b_0 b_1^2) e_2 + (-3b_0 b_1 b_2 + b_1^3) e_3\end{aligned}$$

其中

$$\begin{cases} e_1 = j_0 + j_1 + j_2 \\ e_2 = j_0 j_1 + j_0 j_2 + j_1 j_2 \\ e_3 = j_0 j_1 j_2 \end{cases}$$

而根据韦达定理

$$e_i = (-1)^{2-i} s_{3-i}, \quad i = 1, 2, 3$$

将该式替换到上述的 $\tilde{\Phi}_3(e_1, e_2, e_3)$ 中便得到了 \mathbb{Z}_n 上只和 $b_0, b_1, b_2, s_0, s_1, s_2$ 有关的多项式 ϕ_3 ，此外，当 $E(\mathbb{F}_p)$ 的 j 不变量为 $H_D(j) = 0$ 的根（比如 $j_0 \neq 0, 1728$ ）时，有 $1/2$ 的概率 E 奇异，此时我们的 division polynomial ψ_n 满足

$$\psi_p \equiv \psi_n \equiv 0 \pmod{p} \quad (1)$$

由于 $E(\mathbb{Z}_n) = E(\mathbb{F}_p) \times E(\mathbb{F}_q)$ ，上述情况下的 (1) 式在 $E(\mathbb{F}_q)$ 与 $E(\mathbb{Z}_n)$ 上均成立，从而当 $\psi_n \neq 0$ 时， $\gcd(\psi_n, n) = p$ 。

然而，在扩域 $\mathcal{S}_n^{D, \tau}$ 或者 \mathcal{R}_n^D 上多项式与整数的最大公因数是针对每一个系数而言的，并没有太大意义（下一小节将提到），因此我们可以将 ψ_n 转移到 \mathbb{Z}_n 上，具体做法就是构造上述的多项式 ϕ ，此时由于

$$\mathcal{S}_n^{D, \tau} \ni (a_0 + a_1 j_0 + \cdots + a_m j_0^m) + (\tilde{a}_0 + \tilde{a}_1 j_0 + \cdots + \tilde{a}_m j_0^m) X_0 \equiv 0 \pmod{p}$$

可以得到

$$\mathcal{R}_n^D \ni b_0 + b_0 j_0 + \cdots + b_m j_0^m \equiv 0 \pmod{p}$$

从而

$$\mathbb{Z}_n \ni \phi_d(b_0, \cdots, b_m, s_0, \cdots, s_m) = \Phi_d(j_0, \cdots, j_m) \equiv 0 \pmod{p}$$

然而，该改进方法在实现过程中仍然存在很大改进的空间，SymPy 在对称约化的过程中，会试图将 $\Phi_d(j_0, \cdots, j_m)$ 展开，这意味着 $\mathcal{O}(d^d)$ 的时间复杂度，对于 d 较大（ $d > 6$ ）的 D 而言这是无法接受的，理应寻找 $\tilde{\Phi}_d(e_1, \cdots, e_d)$ 系数的规律。

此外，SageMath 对于多元商环的实现存在问题，导致计算 division polynomial 时效率极其低下，因此理想的方法是抄了 SageMath 的家研究底层逻辑，找到问题在哪，或者尝试使用一元商环的新算法（下一节会提到），换用其他软件，或者自己用 Python 手动实现一遍：）。

```
import random, sys
import sympy
sys.setrecursionlimit(int(2147483647))
def polynomial_egcd(f, g):
    old_r, r = f, g
    old_s, s = 1, 0
    old_t, t = 0, 1
```

```

while r != 0:
    try:
        q = old_r // r
        old_r, r = r, old_r - q * r
        old_s, s = s, old_s - q * s
        old_t, t = t, old_t - q * t
    except:
        raise ValueError("No inverse for r in Q.", r)

return old_r, old_s, old_t
def polynomial_inv_mod(f, g):
    g, s, _ = polynomial_egcd(f, g)

    if g.degree() > 1:
        raise ValueError("No polynomial inverse exists.")

    return s * g.lc()**-1

def algorithm4(D, N):
    N = int(N)
    Zn = Zmod(N)
    while 1:
        P.<X, j> = Zn[]
        H = hilbert_class_polynomial(-D)(j)
        R.<X, j> = P.quo(H)
        deg = H.degree()
        x0, r = [random.randint(1, N) for _ in range(2)]
        A = R(3*j*r^2 * polynomial_inv_mod((1728-j).lift(), H))
        B = R(2*j*r^3 * polynomial_inv_mod((1728-j).lift(), H))
        E = EllipticCurve(R, [A, B])
        tau = x0^3 + A*x0 + B
        S.<X, j> = R.quotient_ring(X^2 - tau)
        print("Degree = %s" % deg)
        print("Calculating division polynomial...")
        d = E.division_polynomial(N, x=X, two_torsion_multiplicity=0)
        Hcoeffs = H.coefficients()[::-1][::-1]
        bcoeffs = (d*d.lift().subs({X: -X})).lift().coefficients()[::-1]
        f = int(1)
        bs = sympy.symbols('b:%d' % deg)
        js = sympy.symbols('j:%d' % deg)
        for i in range(deg):
            f *= sum([int(bi)*js[i]**k for k, bi in enumerate(bcoeffs)])
        res, _, mappings_ = sympy.symmetrize(f, js, formal=1)
        mappings = [(item[0], int((-1)**(deg-i)*Hcoeffs[deg-i-1])) for i, item
                     in enumerate(mappings_)]
        mappings = dict(mappings)
        c = Zn(int(res.subs(mappings)))
        g = gcd(N, c)
        if g not in [0, 1, N]:
            return int(g)

```

1.1.2 The True and Thorough Method for the $4p - 1$ Factorization Problem

上一小节提到，我们的 division polynomial $\psi_n = w$ 在 $\mathcal{S}_n^{D,\tau}$ 上无法正常地与 n 进行 GCD 运算，因此我们应寻找不正常的 GCD 运算，比如 Extended GCD (EGCD, XGCD)。

多项式之间的 XGCD 算法如下：

```
def polynomial_egcd(f, g):
    old_r, r = f, g
    old_s, s = 1, 0
    old_t, t = 0, 1

    while r != 0:
        try:
            q = old_r // r
            old_r, r = r, old_r - q * r
            old_s, s = s, old_s - q * s
            old_t, t = t, old_t - q * t
        except:
            raise ValueError("No inverse for r in Q.", r)

    return old_r, old_s, old_t
```

类似于矩阵的高斯消元法，XGCD 的基本逻辑是搜集 GCD 过程使用的系数。

设 $f, h \in \mathbb{Z}[x]$ ，则 XGCD 输出三个变量： $g = \gcd(f, h)$ ， a 以及 b ，其中 $af + bh = g$ 。

$\mathbb{Z}[x]$ 上的多项式都能顺利地完 GCD 过程。

然而，在有限域 $\mathbb{Z}_n[x]$ 上，我们知道，一个元素 d 的除法能不能顺利完成取决于该元素是否存在逆元，而这一点在 $d \in \mathbb{Z}_n$ 且 $\gcd(d, n) \neq 1$ 时是做不到的。反过来说，当 $\mathbb{Z}_n[x]$ 上的 XGCD 出错时，往往意味着 $\gcd(d, n) \neq 1$ ，也就是说，我们找到了 n 的一个非平凡因子。

再来看 division polynomial w 的计算过程。在计算 w 的过程中，我们将其对 H_D 反复取模，因此，原来的 w 应该有如下的形式： $\tilde{w} = w + k \cdot H_D$ 。我们试图在 $\mathbb{Z}_n[j]$ 上对 w 与 $H_D(j)$ 进行 XGCD 过程，然而，注意到， \tilde{w} 要么为零，要么每个系数与 n 都有非平凡因子，即 $\gcd(n, \tilde{w}) > 1$ ，因此很显然地，XGCD 过程中第二次尝试除法时，程序会尝试求出 $\tilde{w} / (H_D - k)$ 中，也即计算 $H_D - k$ 中 $\leq \deg(\tilde{w})$ 次项的系数的逆（一般而言 $\deg(\tilde{w}) > \deg(H_D - k)$ ），这相当于求 $K \cdot p$ 的逆，显然是不存在的，于是我们可以判断出，我们得到了 n 的一个非平凡因子。因此我们可以对上述 XGCD 算法做如下改进：

```
def super_xgcd(f, g, n):
    old_r, r = f, g
    old_s, s = 1, 0
    old_t, t = 0, 1

    while r != 0:
        try:
            q = old_r // r
            old_r, r = r, old_r - q * r
            old_s, s = s, old_s - q * s
            old_t, t = t, old_t - q * t
```

```

    except:
        return GCD(r.lc(), n), None, None
    return old_r, old_s, old_t

```

根据前述原理，我们完全绕过了上一小节中对对称多项式的讨论，并且得到了更通用的算法。

```

def super_xgcd(f, g, n):
    old_r, r = f, g
    old_s, s = 1, 0
    old_t, t = 0, 1

    while r != 0:
        try:
            q = old_r // r
            old_r, r = r, old_r - q * r
            old_s, s = s, old_s - q * s
            old_t, t = t, old_t - q * t
        except:
            return GCD(r.lc(), n), None, None
    return old_r, old_s, old_t

def algorithm5(D, N):
    N = int(N)
    Zn = Zmod(N)
    while 1:
        P.<j> = Zn[]
        H = hilbert_class_polynomial(-D)(j)
        deg = H.degree()
        r = random.randint(1, N)
        A = 3*j*r^2 * polynomial_inv_mod((1728-j), H)
        B = 2*j*r^3 * polynomial_inv_mod((1728-j), H)
        R.<j> = P.quo(H)
        E = EllipticCurve(R, [A, B])
        print("Degree = %s" % deg)
        print("Calculating division polynomial...")
        d = E.division_polynomial(N, x=j, two_torsion_multiplicity=0)
        g = super_xgcd(d.lift(), H, N)[0]
        if g not in [0, 1, N]:
            return int(g)

```

它比白势政明等人提出的三个算法以及笔者在上一小节中提出的第四个算法都更加通用。

针对 $DV^2 + 1$ 型的素因子以及合理大小的 D 而言，我们有了高效的算法，于是，即便我们篡改 RSA 模数生成的程序，他人在不知道 D 的情况下也很难判断得出该程序经过了篡改（穷举 D 花费的时间太长，除非后续有人提出针对该问题的检测手段），因此 D 成为了我们破解该公私钥系统的后门口令。

至此，“RSA Backdoor”这一称号实至名归。