Mersenne Mayhem

Author: supasuge

Challenge inspired by: Improved Lattice-Based Attack on Mersenne Low Hamming Ratio Search Problem

Please see the bottom for a detailed step-by-step explaination of the intended solution.

1. Background & AJPS Overview

The **AJPS** cryptosystem (Aggarwal–Joux–Prakash–Santha, Crypto 2018) is based on a Mersenne prime

$$p = 2^n - 1, \quad n \text{ prime}.$$

Secret keys are two sparse integers $f, g \in \mathbb{Z}/p\mathbb{Z}$ each of Hamming weight

$$\mathrm{Ham}(f) = \mathrm{Ham}(g) = w \approx \sqrt{n},$$

and the public key is

$$h \equiv f/g \pmod{p}.$$

Encryption in the **bit-by-bit** scheme uses extra sparse a, b (also weight~w):

$$c = (-1)^m \, (a\, h + b),$$

and decryption recovers m by comparing

$$d = \mathrm{Ham}(c\, g) \ \le 2w^2 \quad \Longleftrightarrow \quad m = 0.$$

In the **KEM** version, one publishes $(r, t = f\,r + g)$, encodes with an error-correcting code, and similar Hamming-distance tests recover the message.

---

2. Challenge Simplification

In this CTF challenge, we omit the bit-by-bit and KEM machinery and instead directly generate:

1. A Mersenne prime

$$p = 2^n - 1\,,\ n = 11213.$$

2. Two sparse secrets

$$f, g \in \{0, \ldots, p-1\}, \quad \mathrm{Ham}(f) = \mathrm{Ham}(g) = w = 10$$

3. Public key

$$h \equiv f/g \pmod{p}.$$

   Finally the flag is encrypted under AES-CBC with key derived as

$$\mathrm{secret} = f \cdot g \pmod{p}, \quad K = \mathrm{SHA3\_256}(\mathrm{secret}).$$

---

3. Hard Problem: MLHRSP

The underlying hard problem is the **Mersenne Low Hamming Ratio Search Problem**:

> **Given** $p = 2^n - 1$, w, and
>
> $$h \equiv f/g \pmod{p}$$
>
> with $\mathrm{Ham}(f) = \mathrm{Ham}(g) = w$,
> **find** (f, g).

In AJPS this is one subproblem; in the KEM one solves a related MLHCSP with $(r, t = f\,r + g)$.

---

4. Lattice-Based Attack as Bivariate Small-Root

1. **Polynomial formulation**

$$f - h\,g \equiv 0 \pmod{p} \implies F(x_1, x_2) = x_1 - h\,x_2, \quad F(f, g) \equiv 0$$

2. **Bounds**

$$|f| \leq X_1 = p^{\xi_1}, \quad |g| \leq X_2 = p^{\xi_2}, \quad \xi_1 = 0.31,\ \xi_2 = 0.69,\ \xi_1 + \xi_2 < 1.$$

3. **Shift polynomials** for $s = 2$:

$$g_i(x_1, x_2) = x_2^{s-i}\,F(x_1, x_2)^i\,p^{\,s-i}, \quad i = 0, 1, 2.$$

4. **Scaling**
   $x_1 \leftarrow X_1 x_1,\ x_2 \leftarrow X_2 x_2$ so the true root is "small."
5. **Lattice basis** $B \in \mathbb{Z}^{3\times 3}$ from coefficient vectors of $g_i(X_1 x_1, X_2 x_2)$.
6. **LLL-reduce** $B \to B'$. The shortest row yields an integer polynomial $H(x_1, x_2)$ vanishing at $(f/X_1, g/X_2)$.
7. **Recover** (f, g) by clearing denominators or solving $\tau = x_1/x_2$.
8. **Compute** secret $= f \cdot g \pmod{p}$, derive AES key, **decrypt** flag.

5. Differences from Real AJPS

Aspect	Real AJPS	CTF Challenge
Secret weight w	Chosen $\approx \sqrt{n}$ with $n > 4w^2$ or $n > 10w^2$	Fixed small $w = 10,\ \ n = 11213$
Encryption scheme	Bit-by-bit and/or KEM with ECC encoding	Single AES-CBC using $f \cdot g \pmod{p}$ as key
Public key material	$h = f/g$ (bit-by-bit) or (r, t) (KEM)	Only $h = f/g$
Security assumption	MLHRSP + MLHCSP	Only MLHRSP
Attack focus	Break indistinguishability or message recovery	Direct key recovery $\to$ AES key $\to$ flag recovery

---

6. Conclusion & Takeaways

- By casting $h\,g \equiv f \pmod{p}$ as a **bivariate small-root** problem with $\xi_1 + \xi_2 < 1$, a tiny 3×3 lattice (LLL) suffices.
- This covers **unbalanced** secrets ($f < p^{0.31}$, $g < p^{0.69}$) far beyond the $\sqrt{p}$ case.
- Once (f, g) are found, the AES-CBC flag is trivially decrypted after calculating the "secret" integer hashed via SHA3-256
- In **real AJPS**, additional parameters and error-correcting layers raise the bar beyond direct MLHRSP.

Code explained

1. **Script initialization & Parameters**

- Mersenne prime/parameter setup

```
m_prime = 11213 # mersenne prime
xi1, xi2 = 0.31, 0.69
w = 10
```

- These values correspond to $n = 11213$, $p = 2^n - 1$, $\xi_1 = 0.31$, $\xi_2 = 0.69$, $w = 10$

Randomness Source

```
rand = SystemRandom()
```

2. **Challenge Parameter Generation (Simplification)**

```
def get_number(n, h):
    # choose exactly h set-bits in n bits, always including the top bit
    low_position = rand.sample(range(n-1), h-1)
    positions = low_positions + [n-1]
    # assemble the integer with 1<<pos

    ...
```

- This picks a sparse integer of Hamming weight h, matching "two sparse secrets" $\mathrm{Ham}(f) = \mathrm{Ham}(g) = w$

```
def gen_params(n, w, xi1, xi2, af=1):
```

```
    p = 2**n - 1

    bf = int(n*xi1); bg = int(n*xi2*af)

    f = get_number(bf, w)

    g = get_number(bg, w)  # repeat until gcd(f,g)=1

    h = inverse(g, p)*f % p

    return p, f, g, h
```

- This yields p, f, g, h as described previously

3. **Flag Encryption**

```
secret = (f*g) % p
K = sha3_256( secret.to_bytes(...) ).digest()
iv = get_random_bytes(16)
cipher = AES.new(K, AES.MODE_CBC, iv)
ciphertext = iv + cipher.encrypt(pad(flag,16))
```

- Here, the AES key is derived via $\text{Key} = \text{SHA3\_256}(f \cdot g \bmod p)$

4. **Polynomial and Lattice Construction**

```
def modular_bivariate_homogeneous(f, N, m, t, X, Y, ...):
    # f(x1,x2) = x1 - h x2
    # generate shifts g_i = x2^{m-i}· f^i · N^{m-i}
    shifts.append( y**(m-k) * f_**k * N**max(t-k,0) )
```

```
    L, monomials = create_lattice(pr, shifts, [X,Y])
    L = reduce_lattice(L)
    polynomials = reconstruct_polynomials(L, f, N**t, monomials, [X,Y])
    ...
    # solve g(t*y,1)=0 to recover (x1,x2)
```

- Step 1: Sets up $F(x_1, x_2) = x_1 - hx_2$
- Step 3: Builds shifts $g_i(x_1, x_2) = x_2^{8-i} F(x_1, x_2)^i p^{8-i}, s = m = t = 2$
- 4. Scaling $x_1 \leftarrow X_1 x_1,\ x_2 \leftarrow X_2 x_2$ so the true root is "small."
- Step 5: `create_lattice` collects their coefficient vectors into a single integer matrix B.
- Step 6: `reduce_lattice` (via LLL or `flatter`) shrinks $B \to B'$

Note: Step 2 & 7 is described below

5. **Solving and Root extraction**

```
def attack(p, h, xi1, xi2, s=5):
    X = int(RR(p)**xi1); Y = int(RR(p)**xi2)
    for x0,y0 in modular_bivariate_homogeneous(..., X, Y):
        if (x0 - h*y0) % p == 0:
            return ZZ(x0), ZZ(y0)
    return None,None
```

- Step 2 (bounds): $|f| \leq X_1 = p^{0.31}, |g| \leq X_2 = p^{0.69}$
- Step 7: After LLL reduction, we obtain an integer polynomial H vanishing at the scaled root; solving for $\tau = x_1/x_2$ recovers the rational (f, g).
- `attack()` simply returns these and checks congruence to confirm.

6. **Recovering flag from small-root polynomial**

Once (f, g) are known, recompute the 'secret': $\mathrm{secret} = f \cdot g, K = \mathrm{SHA3\_256}(\mathrm{secret})$, then very simply use this derived secret to decrypt the flag using AES-CBC.