This challenge simulate a public key crypto schema based on the conjectured hardness of the Mersenne Low Hamming Ratio Assumption, first you need to reverse the binay to figure out its basic algorithms:

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$, which chooses the public parameters $\mathsf{pp} = (n, h)$ so that $p = 2^n - 1$ is prime and so as to achieve a λ-bit security level. In [AJPS17] the following lower bound is derived

$$\binom{n-1}{h-1} > 2^\lambda$$

which for instance is satisfied by $\lambda = 120, \mathsf{pp} = (n = 1279, h = 17)$.

$\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{sk}, \mathsf{pk})$, which picks F, G two n-bit strings chosen independently and uniformly at random from all n-bit strings of Hamming weight h, and returns $\mathsf{sk} \leftarrow G$ and $\mathsf{pk} \leftarrow H = F/G \bmod (2^n - 1)$.

$\mathsf{Encrypt}(\mathsf{pp}, \mathsf{pk}, b \in \{0,1\}) \to c$, which picks A, B two n-bit strings chosen independently and uniformly at random from all n-bit strings of Hamming weight h, then computes

$$c \leftarrow (-1)^b (AH + B) \bmod (2^n - 1).$$

The schema is vulnerable to Lenstra–Lenstra–Lovász lattice basis reduction algorithm and random partition, the observation at the beginning of this section is that using a balanced partition that is correct for F and another one that is correct for G, we can recover F and G from H.
Since F and G are unknown, we cannot construct a correct partition from them directly; but the probability that a random balanced partition is correct for F (resp. G) is lower bounded5 by 2^{-h} .
Assuming that F and G are independent, which they should be according to the key generation procedure, we found a correct partition for both F and G with a probability of 2^{-2h}, the steps are as follows:

1. Compute the size of the each non-zero blocks in f and g, we call these sizes $\boldsymbol{u} = \{u_i\}$ and $\boldsymbol{v} = \{v_i\}$ respectively, with $i = 0, \ldots, m/2 - 1$. Let $w = \max_i\{u_i, v_i\}$.
2. Construct the vector $\boldsymbol{s} = s_i$ as follows:

$$s_i = \begin{cases} 2^{w-v_i} & \text{if } i < m/2 \\ 2^{w-u_i} & \text{if } m/2 \le i < m \end{cases}$$

3. Construct the vector $\boldsymbol{a} = \{a_j\}$ as follows: let f_i (resp. g_i) denote the starting position of the non-zero blocks in F (rep. G), and set

$$a_j = \begin{cases} H \times 2^{g_i} \bmod p & \text{if } j < m/2 \\ p - 2^{f_i} & \text{if } m/2 \le j < m \end{cases}$$

4. Choose an integer K, and assemble the matrix $\boldsymbol{M}$ as follows:

$$\boldsymbol{M} = \begin{pmatrix} \mathrm{diag}(\boldsymbol{s}) & K\boldsymbol{a} \\ 0 & Kp \end{pmatrix}$$

5. Finally, we use LLL on M (using the Mathematica command `LatticeReduce`) and recover the reduced matrix's row that complies with the Hamming density of F and G. This row is expected to give the values of the non-zero blocks of F and G, and we can check its correctness by computing its Hamming weight, and checking that the ratio of the candidate values modulo p give H.

But here we can't get the parameters directly, since we know C_{n-1}^{h-1}should be greater than or equal to 2^{λ}, after doing some test we can soon figure out the only possible value of h is either 6 or 7, so we can just try this two numbers.

Another hinder is we don't know the value of n, p and public key, but we know the value of (nextprime(p)+nextprime(public key))*(nextprime(p) xor nextprime(public key)), here we can we can recover nextprime(p) and nextprime(public key) bit-by-bit from LSB, then given that (nextprime(p) – p) won't be too large compared with p, and we know that p is $2^n - 1$, where n is a small random number, too, so we can brute force the two number in a small search space to find what is the correct value of n and x satisfying $2^n - 1$ == nextprime(p) – x, then we can get n and x quickly, once we get x, we'll know the correct value of p, we can use the same way to brute force the public key to find what is the correct value of y satisfying md5(public key) == nextprime(public key) – y.

After got all the parameters we need, we'll find that the ciphertext parts are modified by adding a small nonce generated by a LCG, although we don't know the increment, multiplier and modulus of this LCG, we can find that the program print the first six nonces' value, which is enough for us to recover the increment, multiplier and modulus using a basic math trick :if we have few random multiples of n, with large probability their gcd will be equal to n, use this method and the extended Euclidean algorithm we can get all the parameters of this LCG, then we can get all the original ciphertext part and decrypt the ciphertext of the flag using the way we mentioned at first.