Safe LCG

- 분야: Crypto
- 키워드: Inversive Congruential Generator, Linear System, LLL

배경

문제에서는 `chal.py`와 그 실행 결과인 `output.txt`를 제공합니다. `chal.py`를 읽어보면 평범한 truncated LCG로 보이지만, 주석 처리된 줄과 다르게 `self.x` 대신 `pow(self.x, -1, self.p)`를 사용하는 것을 볼 수 있습니다. 이는 Inversive Congruential Generator라고 불리는 비선형적인 난수 생성기 방식입니다.

이 문제를 풀기 위해서는 수식을 세운 뒤 이를 바탕으로 선형식을 세우고, 해당 식의 해가 작다는 사실을 이용해 LLL 알고리즘을 적용해 해를 구하는 방식을 이해하고 있어야 합니다. 풀이자는 이 문제를 통해 LLL을 적용하는 한 갈래를 배울 수 있고, LLL을 적용하는 과정에서 알아두면 좋은 테크닉을 공부할 수 있습니다.

분석

```python
from Crypto.Cipher import AES
from Crypto.Util.number import getPrime, long_to_bytes
from Crypto.Util.Padding import pad
from secrets import randbelow
import hashlib

class SafeLCG:
    def __init__(self):
        self.n = 256
        self.m = 96
        self.p = getPrime(self.n + self.m)
        self.a = randbelow(self.p)
        self.b = randbelow(self.p)
        self.x = randbelow(self.p)

    def next(self):
        # self.x = (self.a * self.x + self.b) % self.p
        # Ok, this must be safe... wait, it's not LCG anymore
        self.x = (self.a * pow(self.x, -1, self.p) + self.b) % self.p
        return self.x & ((1 << self.n) - 1)


lcg = SafeLCG()

for _ in range(10):
    print(lcg.next())

with open("flag", "rb") as f:
    flag = f.read()

key = hashlib.sha256(long_to_bytes(lcg.next())).digest()
cipher = AES.new(key, AES.MODE_ECB)
ct = cipher.encrypt(pad(flag, 16))

print(f"{lcg.p = }")
print(f"{lcg.a = }")
print(f"{lcg.b = }")
print(f"ct = {ct.hex()}")
```

코드는 매우 간단합니다. 총 256 + 96 = 352비트의 상태를 내부적으로 가지며, 매 `next()` 호출마다 $x_i = ax_{i-1}^{-1} + b \mod p$를 계산한 뒤 x_i의 하위 256비트를 반환합니다.

연속해서 생성된 총 10개의 난수를 출력하고, 이후 한 번 더 생성한 난수를 바탕으로 플래그를 AES-ECB 암호화해서 p, a, b 값과 함께 출력합니다.

풀이

작은 변수에 대한 선형식 세우기

우선 내부 상태 x_i와 실제로 제공되는 256비트 난수 o_i에 대해서, 96비트 에러 e_i를 포함해 다음과 같이 식을 세울 수 있습니다.

$$x_i = o_i + 2^{256}e_i$$

풀이자에게는 $o_1, o_2, \ldots, o_{10}$이 주어집니다.

이를 바탕으로 LCG식 $x_i = ax_{i-1}^{-1} + b \mod p$를 o_i, e_i에 대한 식으로 표현해봅시다.

$$x_i \equiv ax_{i-1}^{-1} + b \pmod p$$

$$(x_i - b)x_{i-1} \equiv a \pmod p$$

$$(o_i + 2^{256}e_i - b)(o_{i-1} + 2^{256}e_{i-1}) \equiv a \pmod p$$

$$2^{512}e_{i-1}e_i + 2^{256}(o_i - b)e_{i-1} + 2^{256}o_{i-1}e_i + (o_i - b)o_{i-1} - a \equiv 0 \pmod p$$

이 때 e_i의 크기는 96비트이므로, $e_{i-1}e_i$ 또한 192비트의 작은 수임을 알 수 있습니다. 그러므로 $y_i = e_ie_{i+1}$로 치환하면 다음과 같이 y_i, e_i에 대한 선형식을 얻을 수 있습니다.

$$2^{512}y_{i-1} + 2^{256}(o_i - b)e_{i-1} + 2^{256}o_{i-1}e_i + (o_i - b)o_{i-1} - a \equiv 0 \pmod p$$

격자 세우기

우선 간단하게 o_1, o_2에 대해서 위의 식을 바탕으로 격자를 생성해봅시다. 1~2번 행은 e_1, e_2, 3번 행은 y_1을 나타냅니다. 4번 행은 $\mathrm{mod} p$를 의미하고, 5번 행은 선형식의 상수항을 의미합니다.

$$\begin{bmatrix} 1 & 0 & 0 & 2^{256}(o_2 - b) \\ 0 & 1 & 0 & 2^{256}o_1 \\ 0 & 0 & 1 & 2^{512} \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & (o_2 - b)o_1 - a \end{bmatrix}$$

e_1, e_2의 크기는 96비트인 것에 비해, y_1의 크기는 192비트이기 때문에 변수간의 크기 차이가 발생합니다. 이를 균등화시키기 위해 1~2번 행의 1을 2^{96}으로 변경해줍니다.

$$\begin{bmatrix} 2^{96} & 0 & 0 & 2^{256}(o_2 - b) \\ 0 & 2^{96} & 0 & 2^{256}o_1 \\ 0 & 0 & 1 & 2^{512} \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & (o_2 - b)o_1 - a \end{bmatrix}$$

4번 열이 항상 0이 나오는 것을 보장하기 위해 4번 열의 모든 값에 매우 큰 값 B를 곱해줍니다. 아래에서 소개할 솔버에서는 2^{1024}를 사용합니다.

$$\begin{bmatrix} 2^{96} & 0 & 0 & 2^{256}(o_2 - b)B \\ 0 & 2^{96} & 0 & 2^{256}o_1B \\ 0 & 0 & 1 & 2^{512}B \\ 0 & 0 & 0 & pB \\ 0 & 0 & 0 & ((o_2 - b)o_1 - a)B \end{bmatrix}$$

결과로 나오게 될 벡터의 평균적인 원소 크기가 2^{192} 이므로 이에 맞춰 Kannan's embedding을 추가해, 마지막 행이 한 번만 더해지게 강제합니다. 또한 $2^{96}e_i, y_i$의 평균값인 2^{191}를 마지막 행에서 빼주는 Recentering을 적용해 LLL이 SVP를 더 잘 찾게끔 만들어줍니다.

- 평균을 빼주면 왜 더 잘 찾나요?
  - LLL은 격자 내에서 가장 짧은 벡터를 찾는 알고리즘입니다. 즉, 원점을 기준으로 가장 가까운 점을 찾는 것이기 때문에, 특정 변수가 $[0, a]$ 범위 내에 있을 경우 해당 변수가 a에 가까울수록 원점과는 멀어지게 됩니다. 하지만 해당 변수에 대해서 $-a/2$를 더해주게 되면, $[0, a]$ 범위가 $[-a/2, a/2]$로 변하게 되므로 평균적인 벡터의 길이를 더 짧게 만들어줄 수 있고, 이를 통해 LLL이 더 해를 잘 찾게 만들 수 있습니다.

$$\begin{bmatrix} 2^{96} & 0 & 0 & 2^{256}(o_2 - b)B & 0 \\ 0 & 2^{96} & 0 & 2^{256}o_1B & 0 \\ 0 & 0 & 1 & 2^{512}B & 0 \\ 0 & 0 & 0 & pB & 0 \\ -2^{191} & -2^{191} & -2^{191} & ((o_2 - b)o_1 - a)B & 2^{192} \end{bmatrix}$$

위의 격자를 그대로 사용하면 주어진 `output.txt` 파일에 대해서는 결과를 얻을 수 없지만, 직접 코드를 수정해 에러 e_i의의 크기가 더 작게끔 해본다면 해당 5x5 격자로도 충분히 해를 구할 수 있을 것입니다.

위의 격자를 o_3을 추가해 확장시킨다면 다음과 같은 구조가 될 것입니다.

$$\begin{bmatrix} 2^{96} & 0 & 0 & 0 & 0 & 2^{256}(o_2-b)B & 0 & 0 \\ 0 & 2^{96} & 0 & 0 & 0 & 2^{256}o_1B & 2^{256}(o_3-b)B & 0 \\ 0 & 0 & 2^{96} & 0 & 0 & 0 & 2^{256}o_2B & 0 \\ 0 & 0 & 0 & 1 & 0 & 2^{512}B & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 2^{512}B & 0 \\ 0 & 0 & 0 & 0 & 0 & pB & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & pB & 0 \\ -2^{191} & -2^{191} & -2^{191} & -2^{191} & -2^{191} & ((o_2-b)o_1-a)B & ((o_3-b)o_2-a)B & 2^{192} \end{bmatrix}$$

이를 더 확장해 10개의 난수값을 모두 사용한다면 충분히 e_i와 y_i를 복구하고, x_i 또한 알아내 문제를 해결할 수 있습니다.

솔버 코드

SageMath를 바탕으로 다음과 같이 작성했습니다.

```
from Crypto.Cipher import AES
from Crypto.Util.number import long_to_bytes
import hashlib

def solve(p, a, b, outputs, bits, trunc):
    count = len(outputs)
    M = 2^(bits - trunc)

    mat = [
        [0 for _ in range(count + count - 1 + count - 1 + 1)]
        for _ in range(count + count - 1 + count - 1 + 1)
    ]

    MUL = 2^trunc
    MUL2 = 2^1024

    for i in range(count - 1):
        t0 = int((outputs[i] * (outputs[i + 1] - b) - a) % p)
        t1 = int((M * (outputs[i + 1] - b)) % p)
        t2 = int((M * outputs[i]) % p)
        t3 = int(M^2 % p)

        mat[i][2 * count - 1 + i] = MUL2 * t1
        mat[i + 1][2 * count - 1 + i] = MUL2 * t2
        mat[count + i][2 * count - 1 + i] = MUL2 * t3
        mat[-1][2 * count - 1 + i] = MUL2 * t0

    for i in range(count - 1):
        mat[count + i][count + i] = 1
        mat[2 * count - 1 + i][2 * count - 1 + i] = MUL2 * p
        mat[-1][count + i] = -MUL^2 // 2

    for i in range(count):
        mat[i][i] = MUL
        mat[-1][i] = -MUL^2 // 2

    mat[-1][-1] = MUL^2

    mat = Matrix(ZZ, mat)

    res = mat.LLL()
    for row in res:
        if row[-1] in [MUL^2, -MUL^2]:
            row = row / (row[-1] // MUL^2)
            es = [ row[i] // MUL + MUL // 2 for i in range(count) ]
            flag = True

            for i in range(count - 1):
                mul = row[count + i] + MUL^2 // 2
                if es[i] * es[i + 1] % p != mul % p:
                    flag = False
                    break

            if flag:
                cur = int(es[-1] * M + outputs[-1])
                nxt = int( (GF(p)(a) / cur + b) )
                return nxt & (M - 1)

with open("output.txt", "r") as f:
    outputs = []
    for _ in range(10):
        outputs.append(int(f.readline().strip()))

    lcg = []
    for _ in range(3):
        lcg.append(int(f.readline().strip().split(" = ")[-1]))
    p, a, b = lcg

    ct = bytes.fromhex(f.readline().strip().split(" = ")[-1])
    out = solve(p, a, b, outputs, 256 + 96, 96)

    key = hashlib.sha256(long_to_bytes(out)).digest()
    cipher = AES.new(key, AES.MODE_ECB)
    print(cipher.decrypt(ct))
```

마치며...

Inversive Congruential Generator를 공격하는 방법에 더 공부하고 싶으신 분들은 하단의 레퍼런스를 읽어보시면 도움이 될 것입니다. 이 문제에서는 해당 논문을 읽지 않고도 풀 수 있는 범위로 파라미터를 제한했지만, 논문에서는 더 어려운 조건에서도 상태를 복구할 수 있는 방법을 제안합니다.

또한 LLL에 대해서는 알고 있었으나 Recentering이나 Kannan's embedding에 대해서 잘 모르셨던 분들을 위해 추가 레퍼런스를 달아둡니다.

레퍼런스

- https://www.ams.org/journals/mcom/2005-74-251/S0025-5718-04-01698-9/S0025-5718-04-01698-9.pdf
- https://www.iacr.org/archive/pkc2012/72930609/72930609.pdf
- https://github.com/rkm0959/Presentations/blob/main/lattice_survey.pdf
- https://eprint.iacr.org/2023/032.pdf

Translate

2025.05.03. 21:59:59