2023 암호분석경진대회

3번 문제

0. 문제 답

개인키 d = 0x1d66f23cec2d7987483d91f1aa951107dbad08fe92d6d096e4a42de85f86f3c1b339af83d57
88b9d48e2b66ae88a9f6fbecb8b01658252e0a23d9364dc42d2d2c9759d71003dfb4869df81d6310e0087d
cbafee6e42b1cab8cc27891eb12eea2ac9d0deafa9f14d22178a4dcfdc4abd511abd9e98eadc18e4a4016f7c5
c6cb69

1. 접근 방법

1.1 문제 분석

해당 문제는 RSA 암호를 사용하지만, 오류 주입 공격이 수행됐고, 이를 통해 개인 키를 복구하는 문제였다. 오류 위치 간의 차이가 작기 때문에, 전수 조사로 풀이 가능했다.

1.2 개인키 복구 전략 및 전수 조사

k 번째 비트에서 오류가 발생한 경우, 변형된 개인키 d'는 다음과 같이 표현할 수 있다.

$$d' = x_1 \ldots x_{1024-k} 0 \ldots 0$$

x는 원래의 개인키 비트, 즉 현재 모르는 비트를 의미하고, 0은 고정된 비트를 의미하게 된다.
x에 해당하는 부분을 전수 조사하여, 상위비트부터 개인키 일부를 복구 한 후, 다음 비트를 복구할 때 이전에 복구한 비트 정보를 이용하는 식으로 전수 조사 진행이 가능하다.
예를 들어, k가 큰 순서, 즉 현재 모르는 비트가 적은 순서로 개인키 복구를 진행하는데, 복구 중인 개인키는 아래와 같이 표현 가능하다.

$$d' = d \ldots dx \ldots . x 0 \ldots 0$$

d는 복구된 개인 키, x는 원래 개인키이지만 모르는 비트, 0은 고정된 비트를 의미하게 된다.
이렇게 모르는 비트 수를 줄임으로써 개인키를 계속해서 복구 가능하다.

이때, 전수조사가 현실적으로 가능한지 계산하기 위해, 오류 위치 간 차이의 평균과 최대값을 계산해보았다.

```
fault_index =[5,11,18,24,30,37,43,51,57,63,70,78,84,90,98,105,112,120,125,131,137,144,150,156,16
1,169,176,180,187,193,201,205,212,220,228,235,239,246,252,258,265,269,277,283,290,295,301,30
7,311,318,326,330,337,345,353,360,365,370,377,383,389,397,405,409,413,421,429,433,441,446,45
4,461,468,476,483,490,497,504,510,517,524,531,538,546,551,559,562,569,577,584,590,596,599,60
6,609,617,624,631,638,642,648,653,658,662,670,676,681,689,693,700,708,716,723,730,738,746,75
3,758,763,769,778,784,790,796,804,812,819,826,831,838,844,850,858,865,869,876,883,890,897,90
3,910,917,925,932,939,945,950,954,959,963,969,976,983,989,993,998,1005,1010,1016,1021]
dif =[]
for i in range(0,len(fault_index)-1):
    dif.append(fault_index[i+1]-fault_index[i])

print("Mean:",sum(dif)/(len(dif)-1))
print("Max:",max(dif))
```

각 스텝 당 평균적으로 $2^{6.389}$번, 최악의 경우 2^9번 전수 조사를 진행하면 되기 때문에, 현대 컴퓨터의 처리 속도를 고려하였을 때 짧은 시간 내에 해결 가능하다는 것을 확인할 수 있다.

복구 전략 순서는 다음과 같다.
1) k가 큰 순서대로 개인키 일부 비트를 복구 한다.
2) 이전 스텝에서 복구된 개인키 부분을 확인해 개인키 상위비트를 복구한다.
3) 각 스텝 마다 모르는 비트에 대해 전수 조사를 진행하면서 $C^{d'} \bmod N = M$과 동일한지 판단한다.

2. 스크립트 작성

```
def load_fault_messages(filepath:str)->list[int]:
    lines =open(filepath,"r").readlines()
    messages =list(map(lambda line:int(line),lines))
    return messages

# RSA 파라미터 (공개키, 개인키))
(N,e),d  =(9863935015828774795617161402485800310673792978572337604798558848990322503167523
894228056337394090570193165117037936121955127889421302143464878311447324672133810720073 8
833103695814346217200966617698677137725683746838186561756004241774294434212453862492176 9
502763307002873489459411274945621762141259952187656 77,0x10001),None

# 평문, 암호문
M  =8684345636908686698383058753930515845376216074491905095625386038921316376845905096 94
897418428003715839777320992039152964978457490698575445047735001461035622200312721246616 9
135522169279492231954962510144676909892095883819057606351964935618910468692572435658048 5
663786964929657082417372715796601952630784727 51
C =37401583471056104381034405 1134135

# e값 추측에 대한 검증 (가장 많이 사용되는 e값인 0x10001로 추측)
# M^e mod N 이 C와 동일하면 정확한 추측으로 판정
assert pow(M,e,N)==C

# 오류 메세지 및 오류 위치 정보
# 처리에 용이하게 (M': 오류 메세지, k: 오류 위치) 형태로 변환
fault_messages =load_fault_messages('./fault_message.txt')
fault_index  =[5,11,18,24,30,37,43,51,57,63,70,78,84,90,98,105,112,120,125,131,137,144,150,156,161,16
9,176,180,187,193,201,205,212,220,228,235,239,246,252,258,265,269,277,283,290,295,301,307,311,318,
326,330,337,345,353,360,365,370,377,383,389,397,405,409,413,421,429,433,441,446,454,461,468,476,48
3,490,497,504,510,517,524,531,538,546,551,559,562,569,577,584,590,596,599,606,609,617,624,631,638,
642,648,653,658,662,670,676,681,689,693,700,708,716,723,730,738,746,753,758,763,769,778,784,790,79
6,804,812,819,826,831,838,844,850,858,865,869,876,883,890,897,903,910,917,925,932,939,945,950,954,
959,963,969,976,983,989,993,998,1005,1010,1016,1021]
faults =list(zip(fault_messages,fault_index))

# 맨 앞에 k가 0인 경우 (개인키에 에러가 없어 메세지가 정확히 복호화 된 경우) 추가
# 추가해야지 하위 5비트까지 복구 가능
message_index_pairs =[(M,0)]+faults

# d는 1024-bit 키로 문제에서 주어짐
BIT_LENGTH =1024 -1
d ='?'*BIT_LENGTH

# 개인키 복구 전략
for Mp,k in message_index_pairs[::-1]:
    # d' = d...d?...?0...0
    restored =d[0:(BIT_LENGTH-d.count('?'))]
    bits_to_restore =BIT_LENGTH-len(restored)-k
    error ='?'*k
    # ?...? 부분에 대해서 전수조사 진행
    for brute_force in range(2**bits_to_restore):
        possible_bits =bin(brute_force)[2:].zfill(bits_to_restore)
        dp =restored +possible_bits +error
        dp_int =int(dp.replace('?','0'),2)
        #print(f"  [*] Trying d': {dp}")
        if pow(C,dp_int,N)==Mp:
            print(f"[+] Successful guess with partial bits: {possible_bits}")
            print(f"[*] Current status of d: {d}")
            d =dp
            break
```

```python
# d를 정상적으로 복구한 경우 C^d mod N = M이 성립
d_int =int(d,2)
print(f"[+] Possible private key found: d = {hex(d_int)}")
print("[*] Checking with C^d mod N")
if pow(C,d_int,N)==M:
    print("[+] Private key recovery successful - assertion matched (C^d mod N = M).")
else:
    print("[0] Private key recovery unsuccessful - assertion failed (C^d mod N != M).")
```

3. 실행 결과

코드 실행 결과는 다음과 같다 (Python 3.10 기준)

```
[+]Possible private key found:d =0x1d66f23cec2d7987483d91f1aa951107dbad08fe92d6d096e4a42de85f86f3c1b339af83d5788b9d48e2b66ae88a9f6fbecb8b01658252e0a23d9364dc42d2d2c9759d71003dfb4869df81d6310e0087dcbafee6e42b1cab8cc27891eb12eea2ac9d0deafa9f14d22178a4dcfdc4abd511abd9e98eadc18e4a4016f7c5c6cb69
[*]Checking with C^d mod N
[+]Private key recovery successful -assertion matched (C^d mod N =M).
python3 ./solver.py  41.91s user 0.20s system 99%cpu 42.150 total
```

해당 알고리즘은 M1 Pro 기준 약 40초 안에 수행된다.