2023 암호분석경진대회

3번 문제

목차

0. 문제풀이 코드 목록

* 3번 문제 관련 파일 목록

1. KUICS_3번/KUICS_3번.py: 문제풀이 코드(개인키 복구 공격 수행)

문제풀이 코드는 python으로 작성하였으며, 코드가 fault_message.txt 파일과 동일한 폴더 아래 위치할 경우 정상적으로 작동함

1. 문제 답

개인키 d **=** 0x1d66f23cec2d7987483d91f1aa951107dbad08fe92d6d096e4a42de85f86f3c1b339af83d5788b9d48e2b66ae88a9f6fbecb8b01658252e0a23d9364dc42d2d2c9759d71003dfb4869df81d6310e0087dcbafee6e42b1cab8cc27891eb12eea2ac9d0deafa9f14d22178a4dcfdc4abd511abd9e98eadc18e4a4016f7c5c6cb69

2. 접근 방법

2.1 문제 분석

해당 문제는 RSA 암호시스템을 사용하고 있으며 오류 주입 공격이 수행된 암호문들을 통해 개인 키를 복구하는 문제이다.

오류 위치 간의 차이가 작기 때문에, 오류가 발생한 위치들의 간격 단위로 각각 전수조사를 진행할 경우 효율적으로 개인키를 복구할 수 있다는 가정하에 문제 풀이를 진행하였다.

2.2 개인키 복구 전략 및 전수 조사

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

$$d' = x_{1024}...x_{k+1}0_k...0_1$$

(x_i는 원래의 개인키와 동일한 비트, 즉 찾아야하는 개인키의 일부분을 의미하고, 0_i은 오류 주입 공격으로 인하여 0으로 고정된 비트를 의미함)

이때, 개인키 d를 복구하기 위해서는 x_i에 해당하는 부분을 전수 조사하여, d의 상위비트부터 개인키의 일부를 복구 한 후, 다음 비트를 복구할 때 이전에 복구한 비트 정보를 이용하는 식으로 효율적인 전수 조사 진행이 가능하다.

즉, k가 큰 순서(현재 모르는 비트의 수가 적은 순서)대로 개인키 복구를 진행하다면, 임의의 단계에서 복구중인 개인키는 아래와 같이 표현할 수 있다.

$$d' = d...dx....x0...0$$

(d는 이전 단계에서 복구된 개인키 일부, x는 전수조사가 필요한 부분, 0은 0으로 고정된 비트를 의미함)

이때, 이전 단계에서 오류가 발생한 위치와 현재 단계에서 오류가 발생한 위치 사이의 부분(x로 표현된 비트들)만 전수조사를 하면 된다.

어느정도 효율적으로 전수조사가 가능한지, 다음과 같이 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))
```

오류 위치의 차이의 평균값은 6.389이며, 최대값은 9이기 때문에, 각 단계 별로 평균적으로 $2^{6.389}$번, 최악의 경우 2^9번 전수 조사를 진행하면 개인키를 복구할 수 있다는 것을 알 수 있고, 이는 보편적인 컴퓨터의 처리 속도를 고려하였을 때 짧은 시간 내에 해결이 가능하다.

실제로 각 단계마다 진행되는 전수조사는 다음과 같이 진행된다.

1) $d' = x_{1024}...x_{k+1}0_k...0_1$와 같이 k의 위치를 이용하여 오류가 주입된 개인키와 복구할 부분을 판단한다.

2) 복구할 부분중 이전 단계에서 복구된 상위비트에 대한 부분을 채운다 ($d' = d...dx....x0...0$)

3) 모르는 비트(x로 표시된 부분)에 대해 전수 조사를 진행하면서 $C^{d'} \bmod N = M'$과 동일한지 판단한다. 만약 같다면 개인키의 해당 부분이 성공적으로 복구된 것으로 판단하고, 다음 k에 대해서 1~3을 반복한다.
   (M'은 오류가 주입된 개인키로 복호화된 평문)

3. 실행 결과

코드 실행 결과는 다음과 같다. (Python 3.10 기준)
해당 문제풀이 코드는 M1 Pro 기준 약 40초 실행된 후 성공적으로 개인키 d를 복구하였다. 따라서 가정했던 대로 효율적인 시간내에 알고리즘이 수행되는 것을 확인할 수 있다.

```
[+]Possible private key found:d =0x1d66f23cec2d7987483d91f1aa951107dbad08fe92d6d096e4a42de85f
86f3c1b339af83d5788b9d48e2b66ae88a9f6fbecb8b01658252e0a23d9364dc42d2d2c9759d71003dfb4869d
f81d6310e0087dcbafee6e42b1cab8cc27891eb12eea2ac9d0deafa9f14d22178a4dcfdc4abd511abd9e98ead
c18e4a4016f7c5c6cb69
[*]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
```

또한, 복구한 개인키 d에 대하여 $C^d \bmod N = M$이 성립하기 때문에 성공적으로 개인키 d를 복구했음을 확인할 수 있다.